7 OAuth2 and OpenID Connect

This chapter covers

- Enabling third-party access to your API with scoped tokens
- Integrating an OAuth2 Authorization Server for delegated authorization
- Validating OAuth2 access tokens with token introspection
- Implementing single sign-on with OAuth and OpenID Connect

In the last few chapters, you've implemented user authentication methods that are suitable for the Natter UI and your own desktop and mobile apps. Increasingly, APIs are being opened to third-party apps and clients from other businesses and organizations. Natter is no different, and your newly appointed CEO has decided that you can boost growth by encouraging an ecosystem of Natter API clients and services. In this chapter, you'll integrate an OAuth2 Authorization Server (AS) to allow your users to delegate access to third-party clients. By using scoped tokens, users can restrict which parts of the API those clients can access. Finally, you'll see how OAuth provides a standard way to centralize token-based authentication within your organization to achieve single sign-on across different APIs and services. The OpenID Connect standard builds on top of OAuth2 to provide a more complete authentication framework when you need finer control over how a user is authenticated.

In this chapter, you'll learn how to obtain a token from an AS to access an API, and how to validate those tokens in your API, using the Natter API as an example. You won't learn how to write your own AS, because this is beyond the scope of this book. Using OAuth2 to authorize service-to-service calls is covered in chapter 11.

LEARN ABOUT IT See OAuth2 in Action by Justin Richer and Antonio Sanso (Manning, 2017; **https://www.manning.com/books/oauth-2-in-action**) if you want to learn how an AS works in detail.

Because all the mechanisms described in this chapter are standards, the patterns will work with any standards-compliant AS with few changes.

See appendix A for details of how to install and configure an AS for use in this chapter.

7.1 Scoped tokens

In the bad old days, if you wanted to use a third-party app or service to access your email or bank account, you had little choice but to give them your username and password and hope they didn't misuse them. Unfortunately, some services did misuse those credentials. Even the ones that were trustworthy would have to store your password in a recoverable form to be able to use it, making potential compromise much more likely, as you learned in chapter 3. Token-based authentication provides a solution to this problem by allowing you to generate a long-lived token that you can give to the third-party service instead of your password. The service can then use the token to act on your behalf. When you stop using the service, you can revoke the token to prevent any further access.

Though using a token means that you don't need to give the third-party your password, the tokens you've used so far still grant full access to APIs as if you were performing actions yourself. The third-party service can use the token to do anything that you can do. But you may not trust a third-party to have full access, and only want to grant them partial access. When I ran my own business, I briefly used a third-party service to read transactions from my business bank account and import them into the accounting software I used. Although that service needed only read access to recent transactions, in practice it had full access to my account and could have transferred funds, cancelled payments, and performed many other actions. I stopped using the service and went back to manually entering transactions because the risk was too great.[1]

The solution to these issues is to restrict the API operations that can be performed with a token, allowing it to be used only within a well-defined scope. For example, you might let your accounting software read transactions that have occurred within the last 30 days, but not let it view or create new payments on the account. The scope of the access you've granted to the accounting software is therefore limited to read-only access to recent transactions. Typically, the scope of a token is represented as one or more string labels stored as an attribute of the token. For example, you might use the scope label `transactions:read` to allow read-access to transactions, and `payment :create` to allow setting up a new payment from an account. Because there may be more than one scope label associated with a token, they are often referred to as scopes. The scopes (labels)

of a token collectively define the scope of access it grants. Figure 7.1 shows some of the scope labels available when creating a personal access token on GitHub.

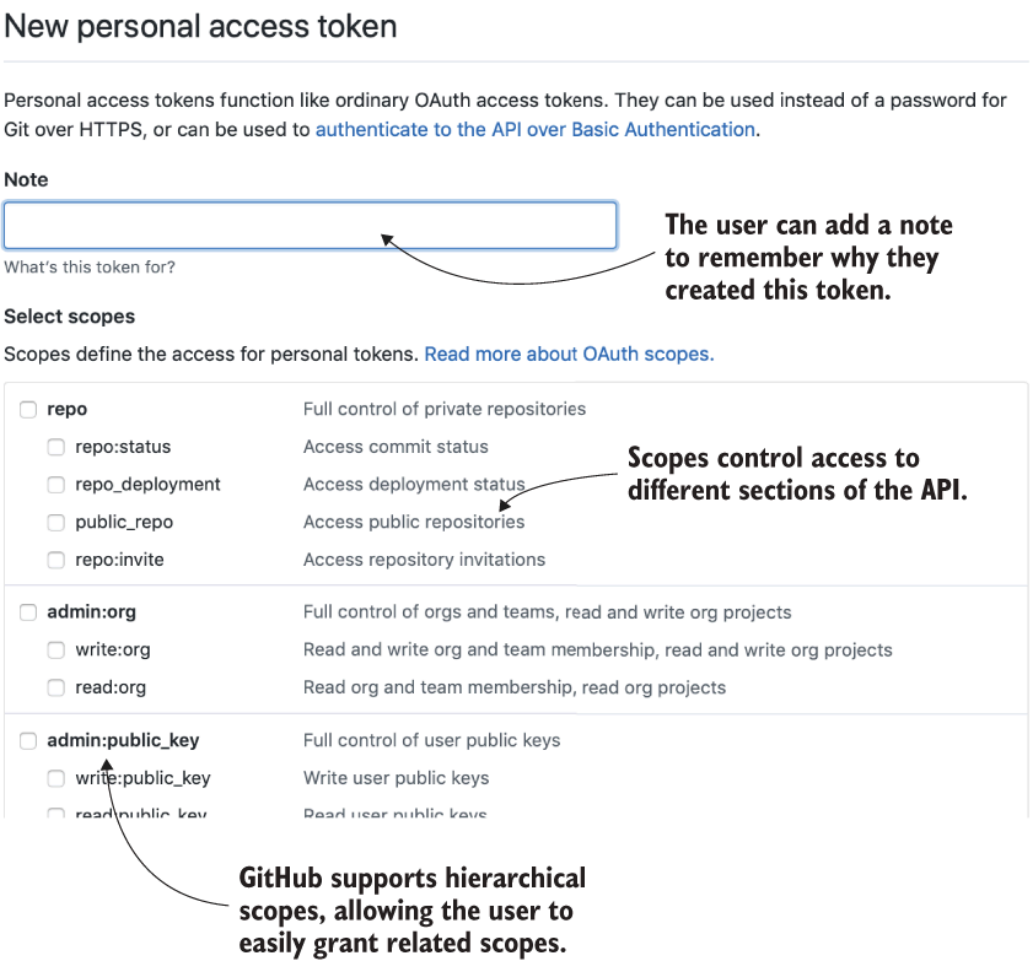

Figure 7.1 GitHub allows users to manually create scoped tokens, which they call personal access tokens. The tokens never expire but can be restricted to only allow access to parts of the GitHub API by setting the scope of the token.

DEFINITION A scoped token limits the operations that can be performed with that token. The set of operations that are allowed is known as the scope of the token. The scope of a token is specified by one or more scope labels, which are often referred to collectively as scopes.

7.1.1 Adding scoped tokens to Natter

Adapting the existing login endpoint to issue scoped tokens is very simple, as shown in listing 7.1. When a login request is received, if it contains a scope parameter then you can associate that scope with the token by storing it in the token attributes. You can define a default set of scopes to grant if the scope parameter is not specified. Open the TokenController.java file in your editor and update the login method to add support for scoped tokens, as in listing 7.1. At the top of the file, add a

new constant listing all the scopes. In Natter, you'll use scopes corresponding to each API operation:

```
private static final String DEFAULT_SCOPES =
    "create_space post_message read_message list_messages " +
  "delete_message add_member";
```

WARNING There is a potential privilege escalation issue to be aware of in this code. A client that is given a scoped token can call this endpoint to exchange it for one with more scopes. You'll fix that shortly by adding a new access control rule for the login endpoint to prevent this.

Listing 7.1 Issuing scoped tokens

```
public JSONObject login(Request request, Response response) {
    String subject = request.attribute("subject");
    var expiry = Instant.now().plus(10, ChronoUnit.MINUTES);

    var token = new TokenStore.Token(expiry, subject);
    var scope = request.queryParamOrDefault("scope", DEFAULT_SCOPES);    ❶
    token.attributes.put("scope", scope);                                ❶
    var tokenId = tokenStore.create(request, token);

    response.status(201);
    return new JSONObject()
            .put("token", tokenId);
}
```

❶ Store the scope in the token attributes, defaulting to all scopes if not specified.

To enforce the scope restrictions on a token, you can add a new access control filter that ensures that the token used to authorize a request to the API has the required scope for the operation being performed. This filter looks a lot like the existing permission filter that you added in chapter 3 and is shown in listing 7.2. (I'll discuss the differences between scopes and permissions in the next section.) To verify the scope, you need to perform several checks:

- First, check if the HTTP method of the request matches the method that this rule is for, so that you don't apply a scope for a POST request to a DELETE request or vice versa. This is needed because Spark's filters are matched only by the path and not the request method.
- You can then look up the scope associated with the token that authorized the current request from the scope attribute of the request. This works because the token validation code you wrote in chapter 4 copies any attributes from the token into the request, so the scope attribute will be copied across too.
- If there is no scope attribute, then the user directly authenticated the request with Basic authentication. In this case, you can skip the scope check and let the request proceed. Any client with access to the user's password would be able to issue themselves a token with any scope.
- Finally, you can verify that the scope of the token matches the required scope for this request, and if it doesn't, then you should return a 403 Forbidden error. The Bearer authentication scheme has a dedicated error code `insufficient_scope` to indicate that the caller needs a token with a different scope, so you can indicate that in the WWW-Authenticate header.

Open TokenController.java in your editor again and add the `requireScope` method from the listing.

Listing 7.2 Checking required scopes

```
public Filter requireScope(String method, String requiredScope) {
    return (request, response) -> {
        if (!method.equalsIgnoreCase(request.requestMethod()))     (1)
            return;                                                (1)
        var tokenScope = request.<String>attribute("scope");       (2)
        if (tokenScope == null) return;                            (2)
        if (!Set.of(tokenScope.split(" "))                         (3)
                .contains(requiredScope)) {                        (3)
            response.header("WWW-Authenticate",                    (3)
                    "Bearer error=\"insufficient_scope\"," +       (3)
                            "scope=\"" + requiredScope + "\"");    (3)
            halt(403);                                             (3)
        }
    };
}
```

(1) If the HTTP method doesn't match, then ignore this rule.

❷ If the token is unscoped, then allow all operations.

❸ If the token scope doesn't contain the required scope, then return a 403 Forbidden response.

You can now use this method to enforce which scope is required to perform certain operations, as shown in listing 7.3. Deciding what scopes should be used by your API, and exactly which scope should be required for which operations is a complex topic, discussed in more detail in the next section. For this example, you can use fine-grained scopes corresponding to each API operation: `create_space`, `post_message`, and so on. To avoid privilege escalation, you should require a specific scope to call the login endpoint, because this can be used to obtain a token with any scope, effectively bypassing the scope checks.[2] On the other hand, revoking a token by calling the logout endpoint should not require any scope. Open the Main.java file in your editor and add scope checks using the `tokenController.requireScope` method as shown in listing 7.3.

Listing 7.3 Enforcing scopes for operations

```
before("/sessions", userController::requireAuthentication);
before("/sessions",                                                    1
        tokenController.requireScope("POST", "full_access"));         1
post("/sessions", tokenController::login);
delete("/sessions", tokenController::logout);                          2

before("/spaces", userController::requireAuthentication);
before("/spaces",                                                      3
        tokenController.requireScope("POST", "create_space"));        3
post("/spaces", spaceController::createSpace);

before("/spaces/*/messages",                                           3
        tokenController.requireScope("POST", "post_message"));        3
before("/spaces/:spaceId/messages",
        userController.requirePermission("POST", "w"));
post("/spaces/:spaceId/messages", spaceController::postMessage);

before("/spaces/*/messages/*",                                         3
        tokenController.requireScope("GET", "read_message"));          3
before("/spaces/:spaceId/messages/*",
        userController.requirePermission("GET", "r"));
get("/spaces/:spaceId/messages/:msgId",
    spaceController::readMessage);

before("/spaces/*/messages",                                           3
```

```
        tokenController.requireScope("GET", "list_messages"));      ❸
before("/spaces/:spaceId/messages",
        userController.requirePermission("GET", "r"));
get("/spaces/:spaceId/messages", spaceController::findMessages);

before("/spaces/*/members",                                         ❸
        tokenController.requireScope("POST", "add_member"));        ❸
before("/spaces/:spaceId/members",
        userController.requirePermission("POST", "rwd"));
post("/spaces/:spaceId/members", spaceController::addMember);

before("/spaces/*/messages/*",                                      ❸
        tokenController.requireScope("DELETE", "delete_message")); ❸
before("/spaces/:spaceId/messages/*",
        userController.requirePermission("DELETE", "d"));
delete("/spaces/:spaceId/messages/:msgId",
    moderatorController::deletePost);
```

❶ Ensure that obtaining a scoped token itself requires a restricted scope.

❷ Revoking a token should not require any scope.

❸ Add scope requirements to each operation exposed by the API.

7.1.2 The difference between scopes and permissions

At first glance, it may seem that scopes and permissions are very similar, but there is a distinction in what they are used for, as shown in figure 7.2. Typically, an API is owned and operated by a central authority such as a company or an organization. Who can access the API and what they are allowed to do is controlled entirely by the central authority. This is an example of mandatory access control, because the users have no control over their own permissions or those of other users. On the other hand, when a user delegates some of their access to a third-party app or service, that is known as discretionary access control, because it's up to the user how much of their access to grant to the third party. OAuth scopes are fundamentally about discretionary access control, while traditional permissions (which you implemented using ACLs in chapter 3) can be used for mandatory access control.

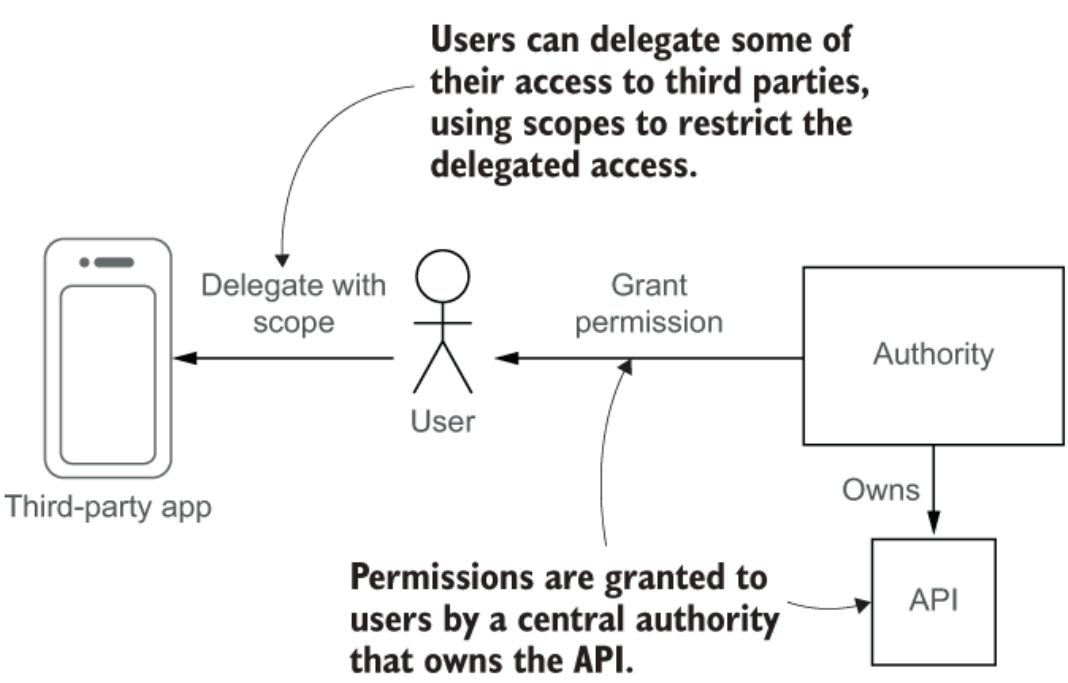

Figure 7.2 Permissions are typically granted by a central authority that owns the API being accessed. A user does not get to choose or change their own permissions. Scopes allow a user to delegate part of their authority to a third-party app, restricting how much access they grant using scopes.

DEFINITION With mandatory access control (MAC), user permissions are set and enforced by a central authority and cannot be granted by users themselves. With discretionary access control (DAC), users can delegate some of their permissions to other users. OAuth2 allows discretionary access control, also known as delegated authorization.

Whereas scopes are used for delegation, permissions may be used for either mandatory or discretionary access. File permissions in UNIX and most other popular operating systems can be set by the owner of the file to grant access to other users and so implement DAC. In contrast, some operating systems used by the military and governments have mandatory access controls that prevent somebody with only SECRET clearance from reading TOP SECRET documents, for example, regardless of whether the owner of the file wants to grant them access.[3] Methods for organizing and enforcing permissions for MAC are covered in chapter 8. OAuth scopes provide a way to layer DAC on top of an existing MAC security layer.

Putting the theoretical distinction between MAC and DAC to one side, the more practical distinction between scopes and permissions relates to how they are designed. The administrator of an API designs permissions to reflect the security goals for the system. These permissions reflect organiza-

tional policies. For example, an employee doing one job might have read and write access to all documents on a shared drive. Permissions should be designed based on access control decisions that an administrator may want to make for individual users, while scopes should be designed based on anticipating how users may want to delegate their access to third-party apps and services.

NOTE The delegated authorization in OAuth is about users delegating their authority to clients, such as mobile apps. The User Managed Access (UMA) extension of OAuth2 allows users to delegate access to other users.

An example of this distinction can be seen in the design of OAuth scopes used by Google for access to their Google Cloud Platform services. Services that deal with system administration jobs, such as the Key Management Service for handling cryptographic keys, only have a single scope that grants access to that entire API. Access to individual keys is managed through permissions instead. But APIs that provide access to individual user data, such as the Fitness API (**http://mng.bz/EEDJ**) are broken down into much more fine-grained scopes, allowing users to choose exactly which health statistics they wish to share with third parties, as shown in figure 7.3. Providing users with fine-grained control when sharing their data is a key part of a modern privacy and consent strategy and may be required in some cases by legislation such as the EU General Data Protection Regulation (GDPR).

Another distinction between scopes and permissions is that scopes typically only identify the set of API operations that can be performed, while permissions also identify the specific objects that can be accessed. For example, a client may be granted a `list_files` scope that allows it to call an API operation to list files on a shared drive, but the set of files returned may differ depending on the permissions of the user that authorized the token. This distinction is not fundamental, but reflects the fact that scopes are often added to an API as an additional layer on top of an existing permission system and are checked based on basic information in the HTTP request without knowledge of the individual data objects that will be operated on.

When choosing which scopes to expose in your API, you should consider what level of control your users are likely to need when delegating access. There is no simple answer to this question, and scope design typically requires several iterations of collaboration between security architects, user experience designers, and user representatives.

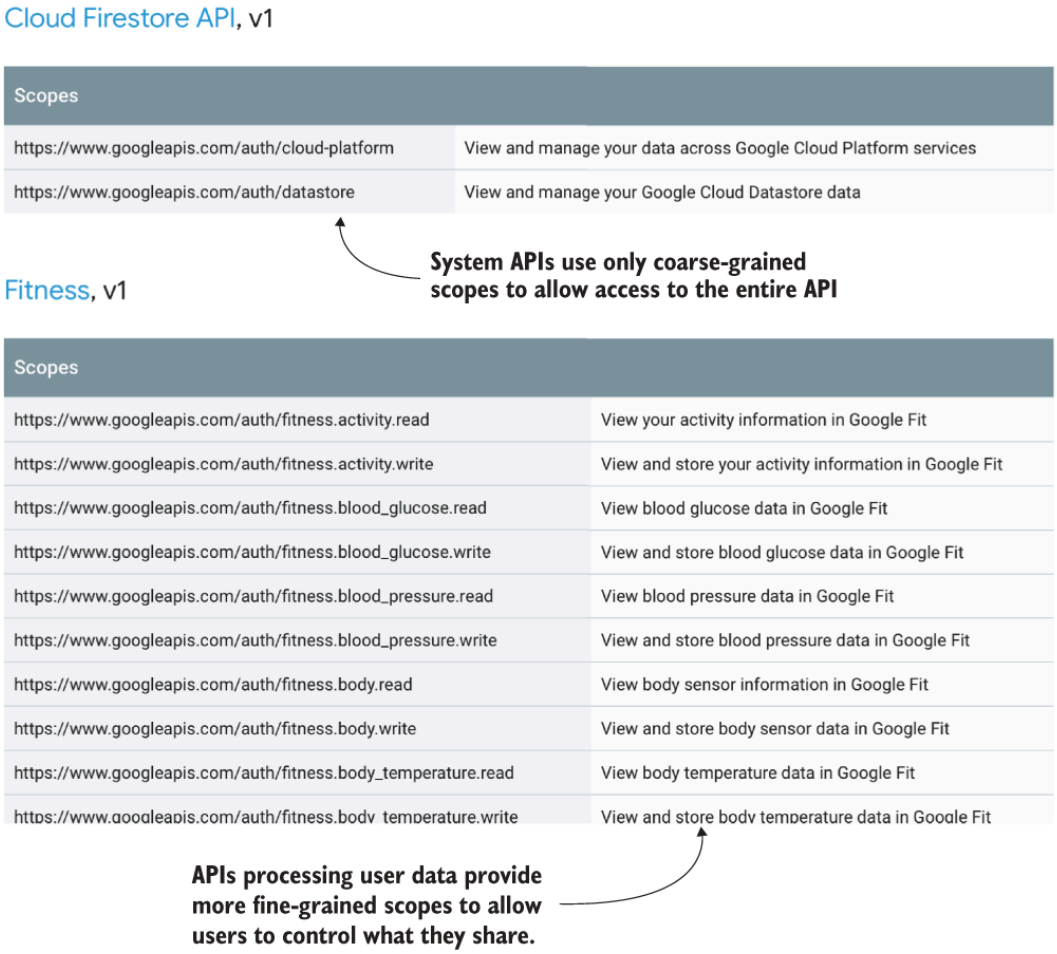

Figure 7.3 Google Cloud Platform OAuth scopes are very coarse-grained for system APIs such as database access or key management. For APIs that process user data, such as the Fitness API, many more scopes are defined, allowing users greater control over what they share with third-party apps and services.

LEARN ABOUT IT Some general strategies for scope design and documentation are provided in The Design of Web APIs by Arnaud Lauret (Manning, 2019; **https://www.manning.com/books/the-design-of-web-apis**).

Pop quiz

1. Which of the following are typical differences between scopes and permissions?
    1. Scopes are more fine-grained than permissions.
    2. Scopes are more coarse-grained than permissions.
    3. Scopes use longer names than permissions.
    4. Permissions are often set by a central authority, while scopes are designed for delegating access.
    5. Scopes typically only restrict the API operations that can be called. Permissions also restrict which objects can be accessed.

The answer is at the end of the chapter.

7.2 Introducing OAuth2

Although allowing your users to manually create scoped tokens for third-party applications is an improvement over sharing unscoped tokens or user credentials, it can be confusing and error-prone. A user may not know which scopes are required for that application to function and so may create a token with too few scopes, or perhaps delegate all scopes just to get the application to work.

A better solution is for the application to request the scopes that it requires, and then the API can ask the user if they consent. This is the approach taken by the OAuth2 delegated authorization protocol, as shown in figure 7.4. Because an organization may have many APIs, OAuth introduces the notion of an Authorization Server (AS), which acts as a central service for managing user authentication and consent and issuing tokens. As you'll see later in this chapter, this centralization provides significant advantages even if your API has no third-party clients, which is one reason why OAuth2 has become so popular as a standard for API security. The tokens that an application uses to access an API are known as access tokens in OAuth2, to distinguish them from other sorts of tokens that you'll learn about later in this chapter.

DEFINITION An access token is a token issued by an OAuth2 authorization server to allow a client to access an API.

OAuth uses specific terms to refer to the four entities shown in figure 7.4, based on the role they play in the interaction:

- The authorization server (AS) authenticates the user and issues tokens to clients.
- The user is known as the resource owner (RO), because it's typically their resources (documents, photos, and so on) that the third-party app is trying to access. This term is not always accurate, but it has stuck now.
- The third-party app or service is known as the client.
- The API that hosts the user's resources is known as the resource server (RS).

7.2.1 Types of clients

Before a client can ask for an access token it must first register with the AS and obtain a unique client ID. This can either be done manually by a system administrator, or there is a standard to allow clients to dynamically register with an AS (**https://tools.ietf .org/html/rfc7591**).

LEARN ABOUT IT OAuth2 in Action by Justin Richer and Antonio Sanso (Manning, 2017; **https://www.manning.com/books/oauth-2-in-action**) covers dynamic client registration in more detail.

There are two different types of clients:

- Public clients are applications that run entirely within a user's own device, such as a mobile app or JavaScript client running in a browser. The client is completely under the user's control.
- Confidential clients run in a protected web server or other secure location that is not under a user's direct control.

The main difference between the two is that a confidential client can have its own client credentials that it uses to authenticate to the authorization server. This ensures that an attacker cannot impersonate a legitimate client to try to obtain an access token from a user in a phishing attack. A mobile or browser-based application cannot keep credentials secret because any user that downloads the application could extract them.[4] For public clients, alternative measures are used to protect against these attacks, as you'll see shortly.

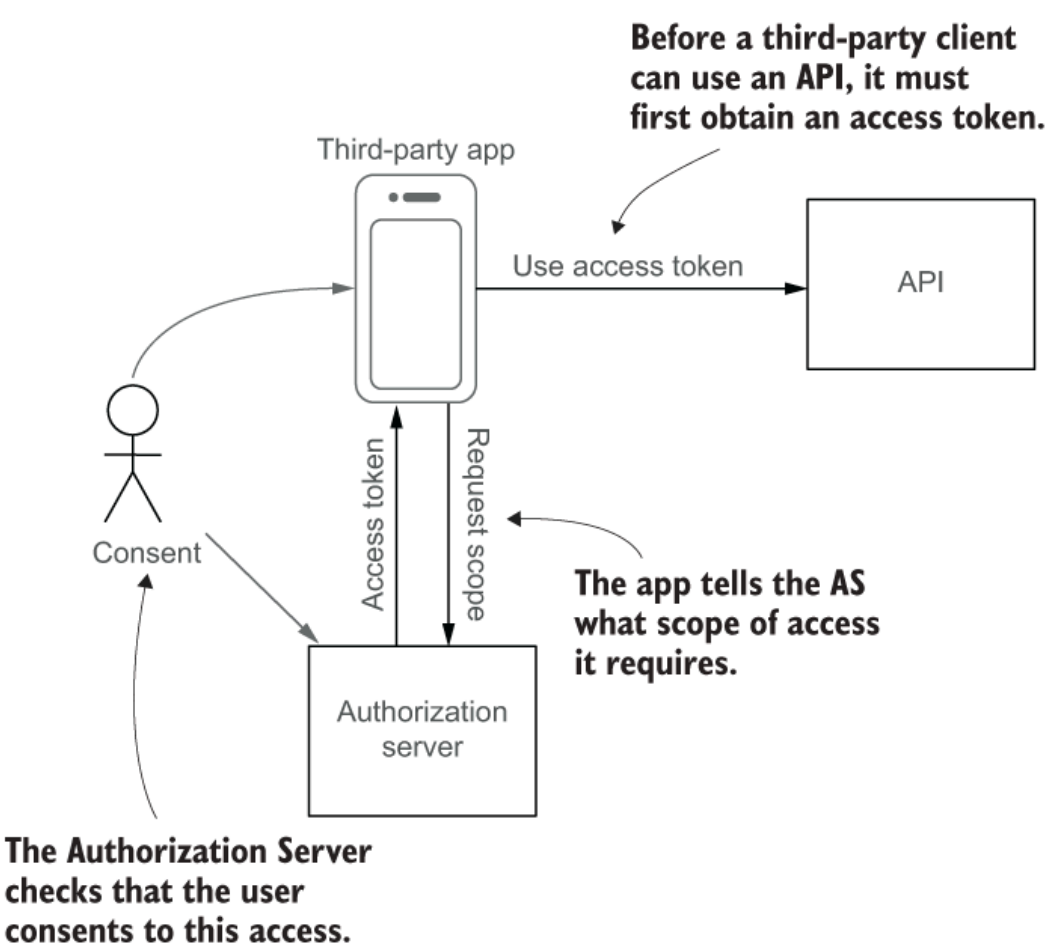

Figure 7.4 To access an API using OAuth2, an app must first obtain an access token from the Authorization Server (AS). The app tells the AS what scope of access it requires. The AS verifies that the user consents to this access and issues an access token to the app. The app can then use the access token to access the API on the user's behalf.

DEFINITION A confidential client uses client credentials to authenticate to the AS. Usually, this is a long random password known as a client secret, but more secure forms of authentication can be used, including JWTs and TLS client certificates.

Each client can typically be configured with the set of scopes that it can ask a user for. This allows an administrator to prevent untrusted apps from even asking for some scopes if they allow privileged access. For example, a bank might allow most clients read-only access to a user's recent transactions but require more extensive validation of the app's developer before the app can initiate payments.

7.2.2 Authorization grants

To obtain an access token, the client must first obtain consent from the user in the form of an authorization grant with appropriate scopes. The client then presents this grant to the AS's token endpoint to obtain an ac-

cess token. OAuth2 supports many different authorization grant types to support different kinds of clients:

- The Resource Owner Password Credentials (ROPC) grant is the simplest, in which the user supplies their username and password to the client, which then sends them directly to the AS to obtain an access token with any scope it wants. This is almost identical to the token login endpoint you developed in previous chapters and is not recommended for third-party clients because the user directly shares their password with the app--the very thing you were trying to avoid!

CAUTION ROPC can be useful for testing but should be avoided in most cases. It may be deprecated in future versions of the standard.

- In the Authorization Code grant, the client first uses a web browser to navigate to a dedicated authorization endpoint on the AS, indicating which scopes it requires. The AS then authenticates the user directly in the browser and asks for consent for the client access. If the user agrees then the AS generates an authorization code and gives it to the client to exchange for an access token at the token endpoint. The authorization code grant is covered in more detail in the next section.
- The Client Credentials grant allows the client to obtain an access token using its own credentials, with no user involved at all. This grant can be useful in some microservice communications patterns discussed in chapter 11.
- There are several additional grant types for more specific situations, such as the device authorization grant (also known as device flow) for devices without any direct means of user interaction. There is no registry of defined grant types, but websites such as **https://oauth.net/2/grant-types/** list the most commonly used types. The device authorization grant is covered in chapter 13. OAuth2 grants are extensible, so new grant types can be added when one of the existing grants doesn't fit.

What about the implicit grant?

The original definition of OAuth2 included a variation on the authorization code grant known as the implicit grant. In this grant, the AS returned an access token directly from the authorization endpoint, so that the client didn't need to call the token endpoint to exchange a code. This was allowed because when OAuth2 was standardized in 2012, CORS had not yet been finalized, so a browser-based client such as a single-page app

could not make a cross-origin call to the token endpoint. In the implicit grant, the AS redirects back from the authorization endpoint to a URI controlled by the client, with the access token included in the fragment component of the URI. This introduces some security weaknesses compared to the authorization code grant, as the access token may be stolen by other scripts running in the browser or leak through the browser history and other mechanisms. Since CORS is now widely supported by browsers, there is no need to use the implicit grant any longer and the OAuth Security Best Common Practice document (**https://tools.ietf.org/html/draft-ietf-oauth-security-topics**) now advises against its use.

An example of obtaining an access token using the ROPC grant type is as follows, as this is the simplest grant type. The client specifies the grant type (`password` in this case), it's client ID (for a public client), and the scope it's requesting as POST parameters in the `application/x-www-form-urlencoded` format used by HTML forms. It also sends the resource owner's username and password in the same way. The AS will authenticate the RO using the supplied credentials and, if successful, will return an access token in a JSON response. The response also contains metadata about the token, such as how long it's valid for (in seconds).

```
$ curl -d 'grant_type=password&client_id=test          (1)
➥ &scope=read_messages+post_message                   (1)
➥ &username=demo&password=changeit'                   (2)
➥ https://as.example.com:8443/oauth2/access_token
{
  "access_token":"I4d9xuSQABWthy71it8UaRNM2JA",       (3)
  "scope":"post_message read_messages",
  "token_type":"Bearer",
  "expires_in":3599}
```

(1) Specify the grant type, client ID, and requested scope as POST form fields.

(2) The RO's username and password are also sent as form fields.

(3) The access token is returned in a JSON response, along with its metadata.

7.2.3 Discovering OAuth2 endpoints

The OAuth2 standards don't define specific paths for the token and authorization endpoints, so these can vary from AS to AS. As extensions have been added to OAuth, several other endpoints have been added, along with several settings for new features. To avoid each client having to hard-code the locations of these endpoints, there is a standard way to discover these settings using a service discovery document published under a well-known location. Originally developed for the OpenID Connect profile of OAuth (which is covered later in this chapter), it has been adopted by OAuth2 (**https://tools.ietf.org/html/rfc8414**).

A conforming AS is required to publish a JSON document under the path /.well-known/oauth-authorization-server under the root of its web server.5 This JSON document contains the locations of the token and authorization endpoints and other settings. For example, if your AS is hosted as https:/ /as.example.com:8443, then a GET request to https:/ /as.example.com:8443/.well-known/oauth-authorization-server returns a JSON document like the following:

```
{
  "authorization_endpoint":
    "http://openam.example.com:8080/oauth2/authorize",
  "token_endpoint":
    "http://openam.example.com:8080/oauth2/access_token",
  ...
}
```

WARNING Because the client will send credentials and access tokens to many of these endpoints, it's critical that they are discovered from a trustworthy source. Only retrieve the discovery document over HTTPS from a trusted URL.

Pop quiz

2. Which two of the standard OAuth grants are now discouraged?
    1. The implicit grant
    2. The authorization code grant
    3. The device authorization grant
    4. Hugh Grant
    5. The Resource Owner Password Credentials (ROPC) grant
3. Which type of client should be used for a mobile app?

1. A public client
2. A confidential client

The answers are at the end of the chapter.

7.3 The Authorization Code grant

Though OAuth2 supports many different authorization grant types, by far the most useful and secure choice for most clients is the authorization code grant. With the implicit grant now discouraged, the authorization code grant is the preferred way for almost all client types to obtain an access token, including the following:

- Server-side clients, such as traditional web applications or other APIs. A server-side application should be a confidential client with credentials to authenticate to the AS.
- Client-side JavaScript applications that run in the browser, such as single-page apps. A client-side application is always a public client because it has no secure place to store a client secret.
- Mobile, desktop, and command-line applications. As for client-side applications, these should be public clients, because any secret embedded into the application can be extracted by a user.

In the authorization code grant, the client first redirects the user's web browser to the authorization endpoint at the AS, as shown in figure 7.5. The client includes its client ID and the scope it's requesting from the AS in this redirect. Set the `response_type` parameter in the query to `code` to request an authorization code (other settings such as `token` are used for the implicit grant). Finally, the client should generate a unique random `state` value for each request and store it locally (such as in a browser cookie). When the AS redirects back to the client with the authorization code it will include the same `state` parameter, and the client should check that it matches the original one sent on the request. This ensures that the code received by the client is the one it requested. Otherwise, an attacker may be able to craft a link that calls the client's redirect endpoint directly with an authorization code obtained by the attacker. This attack is like the Login CSRF attacks discussed in chapter 4, and the state parameter plays a similar role to an anti-CSRF token in that case. Finally, the client should include the URI that it wants the AS to redirect to with the authorization code. Typically, the AS will require the client's redirect URI to be pre-registered to prevent open redirect attacks.

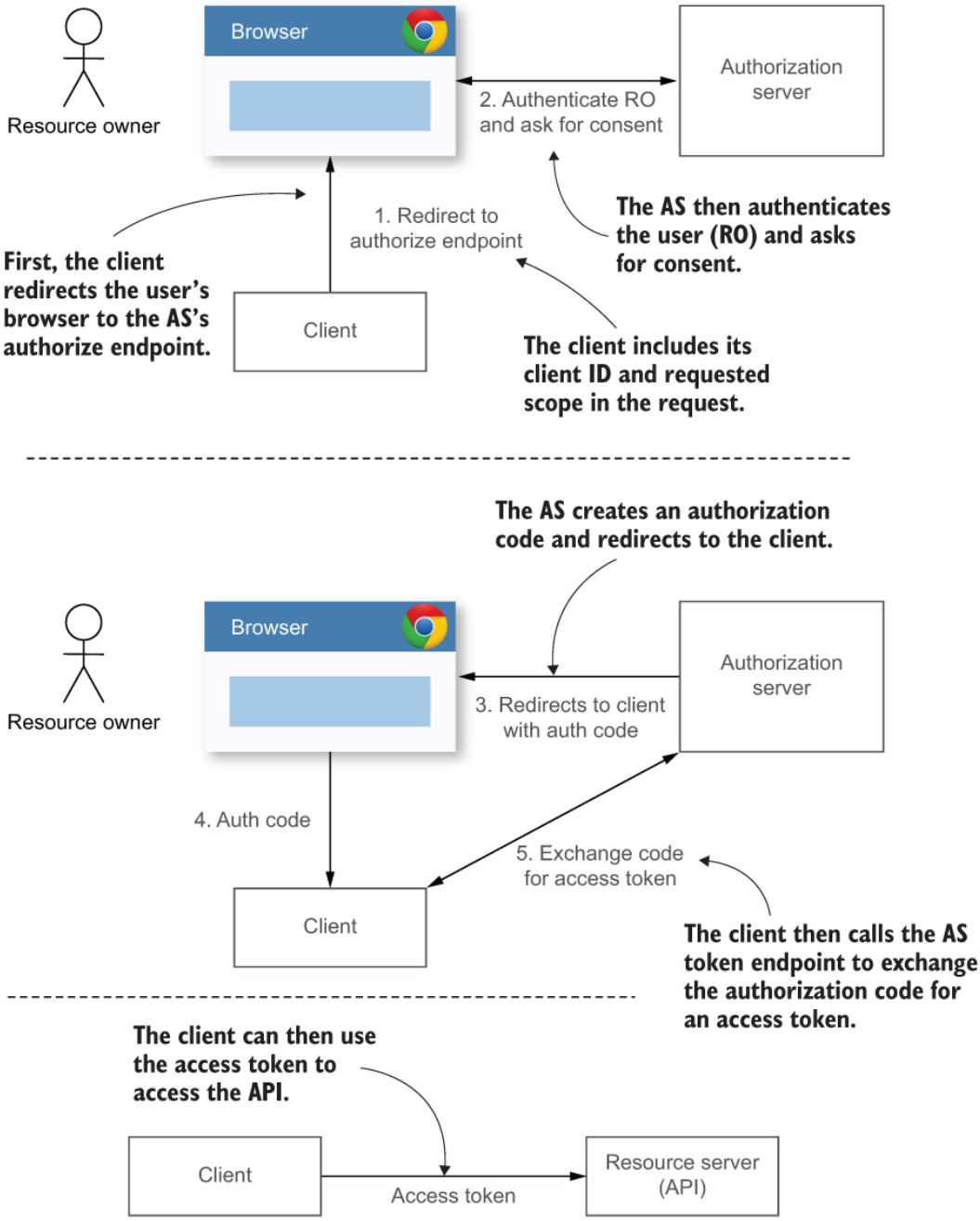

Figure 7.5 In the Authorization Code grant, the client first redirects the user's web browser to the authorization endpoint for the AS. The AS then authenticates the user and asks for consent to grant access to the application. If approved, then the AS redirects the web browser to a URI controlled by the client, including an authorization code. The client can then call the AS token endpoint to exchange the authorization code for an access token to use to access the API on the user's behalf.

DEFINITION An open redirect vulnerability is when a server can be tricked into redirecting a web browser to a URI under the attacker's control. This can be used for phishing because it initially looks like the user is going to a trusted site, only to be redirected to the attacker. You should require all redirect URIs to be pre-registered by trusted clients rather than redirecting to any URI provided in a request.

For a web application, this is simply a case of returning an HTTP redirect status code such as 303 See Other,[6] with the URI for the authorization endpoint in the Location header, as in the following example:

```
HTTP/1.1 303 See Other
Location: https://as.example.com/authorize?client_id=test          (1)
➥ &scope=read_messages+post_message                               (2)
➥ &state=t9kWoBWsYjbsNwY0ACJj0A                                   (3)
➥ &response_type=code                                             (4)
➥ &redirect_uri=https://client.example.net/callback               (5)
```

(1) The client_id parameter indicates the client.

(2) The scope parameter indicates the requested scope.

(3) Include a random state parameter to prevent CSRF attacks.

(4) Use the response_type parameter to obtain an authorization code.

(5) The client's redirection endpoint

For mobile and desktop applications, the client should launch the system web browser to carry out the authorization. The latest best practice advice for native applications (**https://tools.ietf.org/html/rfc8252**) recommends that the system browser be used for this, rather than embedding an HTML view within the application. This avoids users having to type their credentials into a UI under the control of a third-party app and allows users to reuse any cookies or other session tokens they may already have in the system browser for the AS to avoid having to login again. Both Android and iOS support using the system browser without leaving the current application, providing a similar user experience to using an embedded web view.

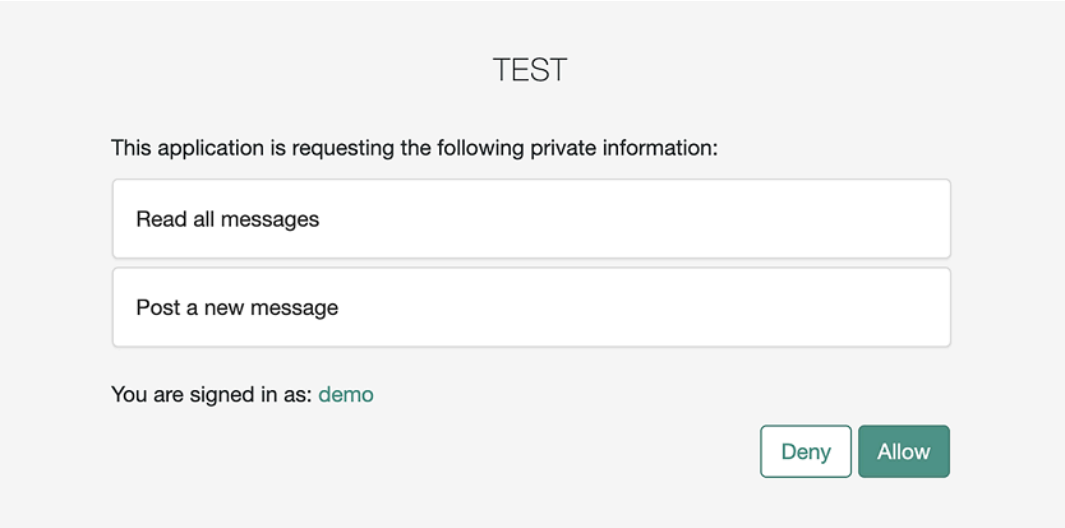

Figure 7.6 An example OAuth2 consent page indicating the name of the client requesting access and the scope it requires. The user can choose to allow or deny the request.

Once the user has authenticated in their browser, the AS will typically display a page telling the user which client is requesting access and the scope it requires, such as that shown in figure 7.6. The user is then given an opportunity to accept or decline the request, or possibly to adjust the scope of access that they are willing to grant. If the user approves, then the AS will issue an HTTP redirect to a URI controlled by the client application with the authorization code and the original `state` value as a query parameter:

```
HTTP/1.1 303 See Other
Location: https://client.example.net/callback?    (1)
➥ code=kdYfMS7H3sOO5y_sKhpdV6NFfik                (1)
➥ &state=t9kWoBWsYjbsNwY0ACJj0A                   (2)
```

(1) The AS redirects to the client with the authorization code.

(2) It includes the state parameter from the original request.

Because the authorization code is included in the query parameters of the redirect, it's vulnerable to being stolen by malicious scripts running in the browser or leaking in server access logs, browser history, or through the HTTP `Referer` header. To protect against this, the authorization code is usually only valid for a short period of time and the AS will enforce that it's used only once. If an attacker tries to use a stolen code after the legitimate client has used it, then the AS will reject the request and revoke any access tokens already issued with that code.

The client can then exchange the authorization code for an access token by calling the token endpoint on the AS. It sends the authorization code in the body of a POST request, using the `application/x-www-form-urlencoded` encoding used for HTML forms, with the following parameters:

- Indicate the authorization code grant type is being used by including `grant_type=authorization_code`.
- Include the client ID in the `client_id` parameter or supply client credentials to identify the client.
- Include the redirect URI that was used in the original request in the `redirect_uri` parameter.
- Finally, include the authorization code as the value of the `code` parameter.

This is a direct HTTPS call from the client to the AS rather than a redirect in the web browser, and so the access token returned to the client is protected against theft or tampering. An example request to the token endpoint looks like the following:

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic dGVzdDpwYXNzd29yZA==          ❶

grant_type=authorization_code&                     ❷
code=kdYfMS7H3sOO5y_sKhpdV6NFfik&                  ❷
redirect_uri=https://client.example.net/callback   ❸
```

❶ Supply client credentials for a confidential client.

❷ Include the grant type and authorization code.

❸ Provide the redirect URI that was used in the original request.

If the authorization code is valid and has not expired, then the AS will respond with the access token in a JSON response, along with some (optional) details about the scope and expiry time of the token:

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

{
  "access_token":"QdT8POxT2SReqKNtcRDicEgIgkk",    ❶
```

```
    "scope":"post_message read_messages",          ❷
    "token_type":"Bearer",
    "expires_in":3599}                             ❸
```

❶ The access token

❷ The scope of the access token, which may be different than requested

❸ The number of seconds until the access token expires

If the client is confidential, then it must authenticate to the token endpoint when it exchanges the authorization code. In the most common case, this is done by including the client ID and client secret as a username and password using HTTP Basic authentication, but alternative authentication methods are allowed, such as using a JWT or TLS client certificate. Authenticating to the token endpoint prevents a malicious client from using a stolen authorization code to obtain an access token.

Once the client has obtained an access token, it can use it to access the APIs on the resource server by including it in an `Authorization: Bearer` header just as you've done in previous chapters. You'll see how to validate an access token in your API in section 7.4.

7.3.1 Redirect URIs for different types of clients

The choice of redirect URI is an important security consideration for a client. For public clients that don't authenticate to the AS, the redirect URI is the only measure by which the AS can be assured that the authorization code is sent to the right client. If the redirect URI is vulnerable to interception, then an attacker may steal authorization codes.

For a traditional web application, it's simple to create a dedicated endpoint to use for the redirect URI to receive the authorization code. For a single-page app, the redirect URI should be the URI of the app from which client-side JavaScript can then extract the authorization code and make a CORS request to the token endpoint.

For mobile applications, there are two primary options:

- The application can register a private-use URI scheme with the mobile operating system, such as `myapp:/ /callback`. When the AS redirects to `myapp:/ /callback?code=...` in the system web browser, the operating system will launch the native app and pass it the callback URI. The native application can then extract the authorization code from this URI and call the token endpoint.
- An alternative is to register a portion of the path on the web domain of the app producer. For example, your app could register with the operating system that it will handle all requests to `https:/ /example.com/app/callback`. When the AS redirects to this HTTPS endpoint, the mobile operating system will launch the native app just as for a private-use URI scheme. Android calls this an App Link (**https://developer.android.com/training/app-links/**), while on iOS they are known as Universal Links (**https://developer.apple.com/ios/universal-links/**).

A drawback with private-use URI schemes is that any app can register to handle any URI scheme, so a malicious application could register the same scheme as your legitimate client. If a user has the malicious application installed, then the redirect from the AS with an authorization code may cause the malicious application to be activated rather than your legitimate application. Registered HTTPS redirect URIs on Android (App Links) and iOS (Universal Links) avoid this problem because an app can only claim part of the address space of a website if the website in question publishes a JSON document explicitly granting permission to that app. For example, to allow your iOS app to handle requests to https:/ /example.com/app/callback, you would publish the following JSON file to https:/ /example.com/.well-known/apple-app-site-association:

```
{
  "applinks": {
"apps": [],
"details": [
  { "appID": "9JA89QQLNQ.com.example.myapp",    ❶
    "paths": ["/app/callback"] }]               ❷
  }
}
```

❶ The ID of your app in the Apple App Store

❷ The paths on the server that the app can intercept

The process is similar for Android apps. This prevents a malicious app from claiming the same redirect URI, which is why HTTPS redirects are recommended by the OAuth Native Application Best Common Practice document (**https://tools.ietf.org/ html/rfc8252#section-7.2**).

For desktop and command-line applications, both Mac OS X and Windows support registering private-use URI schemes but not claimed HTTPS URIs at the time of writing. For non-native apps and scripts that cannot register a private URI scheme, the recommendation is that the application starts a temporary web server listening on the local loopback device (that is, http://127.0.0.1) on a random port, and uses that as its redirect URI. Once the authorization code is received from the AS, the client can shut down the temporary web server.

7.3.2 Hardening code exchange with PKCE

Before the invention of claimed HTTPS redirect URIs, mobile applications using private-use URI schemes were vulnerable to code interception by a malicious app registering the same URI scheme, as described in the previous section. To protect against this attack, the OAuth working group developed the PKCE standard (Proof Key for Code Exchange; **https://tools.ietf.org/html/rfc7636**), pronounced "pixy." Since then, formal analysis of the OAuth protocol has identified a few theoretical attacks against the authorization code flow. For example, an attacker may be able to obtain a genuine authorization code by interacting with a legitimate client and then using an XSS attack against a victim to replace their authorization code with the attacker's. Such an attack would be quite difficult to pull off but is theoretically possible. It's therefore recommended that all types of clients use PKCE to strengthen the authorization code flow.

The way PKCE works for a client is quite simple. Before the client redirects the user to the authorization endpoint, it generates another random value, known as the PKCE code verifier. This value should be generated with high entropy, such as a 32-byte value from a `SecureRandom` object in Java; the PKCE standard requires that the encoded value is at least 43 characters long and a maximum of 128 characters from a restricted set of characters. The client stores the code verifier locally, alongside the state parameter. Rather than sending this value directly to the AS, the client first hashes[7] it using the SHA-256 cryptographic hash function to create a code challenge (listing 7.4). The client then adds the code challenge as another query parameter when redirecting to the authorization endpoint.

Listing 7.4 Computing a PKCE code challenge

```
String addPkceChallenge(spark.Request request,
        String authorizeRequest) throws Exception {

    var secureRandom = new java.security.SecureRandom();
    var encoder = java.util.Base64.getUrlEncoder().withoutPadding();

    var verifierBytes = new byte[32];                                    (1)
    secureRandom.nextBytes(verifierBytes);                               (1)
    var verifier = encoder.encodeToString(verifierBytes);                (1)

    request.session(true).attribute("verifier", verifier);               (2)

    var sha256 = java.security.MessageDigest.getInstance("SHA-256");     (3)
    var challenge = encoder.encodeToString(                              (3)
            sha256.digest(verifier.getBytes("UTF-8")));                  (3)
    return authorizeRequest +
        "&code_challenge=" + challenge +                                 (4)
        "&code_challenge_method=S256";                                   (4)
}
```

(1) Create a random code verifier string.

(2) Store the verifier in a session cookie or other local storage.

(3) Create a code challenge as the SHA-256 hash of the code verifier string.

(4) Include the code challenge in the redirect to the AS authorization endpoint.

Later, when the client exchanges the authorization code at the token endpoint, it sends the original (unhashed) code verifier in the request. The AS will check that the SHA-256 hash of the code verifier matches the code challenge that it received in the authorization request. If they differ, then it rejects the request. PKCE is very secure, because even if an attacker intercepts both the redirect to the AS and the redirect back with the authorization code, they are not able to use the code because they cannot compute the correct code verifier. Many OAuth2 client libraries will automatically compute PKCE code verifiers and challenges for you, and it significantly improves the security of the authorization code grant so you should always use it when possible. Authorization servers that don't sup-

port PKCE should ignore the additional query parameters, because this is required by the OAuth2 standard.

7.3.3 Refresh tokens

In addition to an access token, the AS may also issue the client with a refresh token at the same time. The refresh token is returned as another field in the JSON response from the token endpoint, as in the following example:

```
$ curl -d 'grant_type=password
➥ &scope=read_messages+post_message
➥ &username=demo&password=changeit'
➥  -u test:password
➥ https://as.example.com:8443/oauth2/access_token
{
  "access_token":"B9KbdZYwajmgVxr65SzL-z2Dt-4",
  "refresh_token":"sBac5bgCLCjWmtjQ8Weji2mCrbI",      (1)
  "scope":"post_message read_messages",
  "token_type":"Bearer","expires_in":3599}
```

(1) A refresh token

When the access token expires, the client can then use the refresh token to obtain a fresh access token from the AS without the resource owner needing to approve the request again. Because the refresh token is sent only over a secure channel between the client and the AS, it's considered more secure than an access token that might be sent to many different APIs.

DEFINITION A client can use a refresh token to obtain a fresh access token when the original one expires. This allows an AS to issue short-lived access tokens without clients having to ask the user for a new token every time it expires.

By issuing a refresh token, the AS can limit the lifetime of access tokens. This has a minor security benefit because if an access token is stolen, then it can only be used for a short period of time. But in practice, a lot of damage could be done even in a short space of time by an automated attack, such as the Facebook attack discussed in chapter 6 (https://newsroom.fb.com/news/2018/09/security-update/). The primary benefit of refresh tokens is to allow the use of stateless access tokens such as JWTs. If the access token is short-lived, then the client is forced to peri-

odically refresh the token at the AS, providing an opportunity for the token to be revoked without the AS maintaining a large blocklist. The complexity of revocation is effectively pushed to the client, which must now handle periodically refreshing its access tokens.

To refresh an access token, the client calls the AS token endpoint passing in the refresh token, using the refresh token grant, and sending the refresh token and any client credentials, as in the following example:

```
$ curl -d 'grant_type=refresh_token                          (1)
➥ &refresh_token=sBac5bgCLCjWmtjQ8Weji2mCrbI'               (1)
➥ -u test:password                                          (2)
➥ https://as.example.com:8443/oauth2/access_token
{
  "access_token":"snGxj86QSYB7Zojt3G1b2aXN5UM",              (3)
  "scope":"post_message read_messages",
  "token_type":"Bearer","expires_in":3599}
```

(1) Use the refresh token grant and supply the refresh token.

(2) Include client credentials if using a confidential client.

(3) The AS returns a fresh access token.

The AS can often be configured to issue a new refresh token at the same time (revoking the old one), enforcing that each refresh token is used only once. This can be used to detect refresh token theft: when the attacker uses the refresh token, it will stop working for the legitimate client.

Pop quiz

4. Which type of URI should be preferred as the redirect URI for a mobile client?
    1. A claimed HTTPS URI
    2. A private-use URI scheme such as myapp:/ /cb
5. True or False: The authorization code grant should always be used in combination with PKCE.

The answers are at the end of the chapter.

7.4 Validating an access token

Now that you've learned how to obtain an access token for a client, you need to learn how to validate the token in your API. In previous chapters, it was simple to look up a token in the local token database. For OAuth2, this is no longer quite so simple when tokens are issued by the AS and not by the API. Although you could share a token database between the AS and each API, this is not desirable because sharing database access increases the risk of compromise. An attacker can try to access the database through any of the connected systems, increasing the attack surface. If just one API connected to the database has a SQL injection vulnerability, this would compromise the security of all.

Originally, OAuth2 didn't provide a solution to this problem and left it up to the AS and resource servers to decide how to coordinate to validate tokens. This changed with the publication of the OAuth2 Token Introspection standard (**https://tools.ietf .org/html/rfc7662**) in 2015, which describes a standard HTTP endpoint on the AS that the RS can call to validate an access token and retrieve details about its scope and resource owner. Another popular solution is to use JWTs as the format for access tokens, allowing the RS to locally validate the token and extract required details from the embedded JSON claims. You'll learn how to use both mechanisms in this section.

7.4.1 Token introspection

To validate an access token using token introspection, you simply make a POST request to the introspection endpoint of the AS, passing in the access token as a parameter. You can discover the introspection endpoint using the method in section 7.2.3 if the AS supports discovery. The AS will usually require your API (acting as the resource server) to register as a special kind of client and receive client credentials to call the endpoint. The examples in this section will assume that the AS requires HTTP Basic authentication because this is the most common requirement, but you should check the documentation for your AS to determine how the RS must authenticate.

TIP To avoid historical issues with ambiguous character sets, OAuth requires that HTTP Basic authentication credentials are first URL-encoded (as UTF-8) before being Base64-encoded.

Listing 7.5 shows the constructor and imports for a new token store that will use OAuth2 token introspection to validate an access token. You'll implement the remaining methods in the rest of this section. The `create` and `revoke` methods throw an exception, effectively disabling the login and logout endpoints at the API, forcing clients to obtain access tokens from the AS. The new store takes the URI of the token introspection endpoint, along with the credentials to use to authenticate. The credentials are encoded into an HTTP Basic authentication header ready to be used. Navigate to src/main/java/com/manning/apisecurityinaction/token and create a new file named OAuth2TokenStore.java. Type in the contents of listing 7.5 in your editor and save the new file.

Listing 7.5 The OAuth2 token store

```
package com.manning.apisecurityinaction.token;
import org.json.JSONObject;
import spark.Request;
import java.io.IOException;
import java.net.*;
import java.net.http.*;
import java.net.http.HttpRequest.BodyPublishers;
import java.net.http.HttpResponse.BodyHandlers;
import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.*;
import static java.nio.charset.StandardCharsets.UTF_8;
public class OAuth2TokenStore implements SecureTokenStore {

    private final URI introspectionEndpoint;                          (1)
    private final String authorization;

    private final HttpClient httpClient;

    public OAuth2TokenStore(URI introspectionEndpoint,                (1)
                            String clientId, String clientSecret) {
        this.introspectionEndpoint = introspectionEndpoint;           (1)

        var credentials = URLEncoder.encode(clientId, UTF_8) + ":" +  (2)
                URLEncoder.encode(clientSecret, UTF_8);               (2)
        this.authorization = "Basic " + Base64.getEncoder()           (2)
                .encodeToString(credentials.getBytes(UTF_8));         (2)

        this.httpClient = HttpClient.newHttpClient();
    }

    @Override
```

```
    public String create(Request request, Token token) {
        throw new UnsupportedOperationException();
    }

    @Override
    public void revoke(Request request, String tokenId) {
        throw new UnsupportedOperationException();
    }
}
```

❶ Inject the URI of the token introspection endpoint.

❷ Build up HTTP Basic credentials from the client ID and secret.

❸ Throw an exception to disable direct login and logout.

To validate a token, you then need to make a POST request to the introspection endpoint passing the token. You can use the HTTP client library in java.net.http, which was added in Java 11 (for earlier versions, you can use Apache HttpComponents, **https://hc.apache.org/httpcomponents-client-ga/**). Because the token is untrusted before the call, you should first validate it to ensure that it conforms to the allowed syntax for access tokens. As you learned in chapter 2, it's important to always validate all inputs, and this is especially important when the input will be included in a call to another system. The standard doesn't specify a maximum size for access tokens, but you should enforce a limit of around 1KB or less, which should be enough for most token formats (if the access token is a JWT, it could get quite large and you may need to increase that limit). The token should then be URL-encoded to include in the POST body as the `token` parameter. It's important to properly encode parameters when calling another system to prevent an attacker being able to manipulate the content of the request (see section 2.6 of chapter 2). You can also include a `token_type_hint` parameter to indicate that it's an access token, but this is optional.

TIP To avoid making an HTTP call every time a client uses an access token with your API, you can cache the response for a short period of time, indexed by the token. The longer you cache the response, the longer it may take your API to find out that a token has been revoked, so you should balance performance against security based on your threat model.

If the introspection call is successful, the AS will return a JSON response indicating whether the token is valid and metadata about the token, such

as the resource owner and scope. The only required field in this response is a Boolean `active` field, which indicates whether the token should be considered valid. If this is `false` then the token should be rejected, as in listing 7.6. You'll process the rest of the JSON response shortly, but for now open OAuth2TokenStore.java in your editor again and add the implementation of the read method from the listing.

Listing 7.6 Introspecting an access token

```
@Override
public Optional<Token> read(Request request, String tokenId) {
    if (!tokenId.matches("[\\x20-\\x7E]{1,1024}")) {
        return Optional.empty();
    }

    var form = "token=" + URLEncoder.encode(tokenId, UTF_8) +
            "&token_type_hint=access_token";

    var httpRequest = HttpRequest.newBuilder()
            .uri(introspectionEndpoint)
            .header("Content-Type", "application/x-www-form-urlencoded")
            .header("Authorization", authorization)
            .POST(BodyPublishers.ofString(form))
            .build();

    try {
        var httpResponse = httpClient.send(httpRequest,
                BodyHandlers.ofString());

        if (httpResponse.statusCode() == 200) {
            var json = new JSONObject(httpResponse.body());

            if (json.getBoolean("active")) {
                return processResponse(json);
            }
        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new RuntimeException(e);
    }

    return Optional.empty();
}
```

❶ Validate the token first.

❷ Encode the token into the POST form body.

❸ Call the introspection endpoint using your client credentials.

❹ Check that the token is still active.

Several optional fields are allowed in the JSON response, including all valid JWT claims (see chapter 6). The most important fields are listed in table 7.1. Because all these fields are optional, you should be prepared for them to be missing. This is an unfortunate aspect of the specification, because there is often no alternative but to reject a token if its scope or resource owner cannot be established. Thankfully, most AS software generates sensible values for these fields.

Table 7.1 Token introspection response fields

Field	Description
`scope`	The scope of the token as a string. If multiple scopes are specified then they are separated by spaces, such as `"read_messages post_message"`.
`sub`	An identifier for the resource owner (subject) of the token. This is a unique identifier, not necessarily human-readable.
`username`	A human-readable username for the resource owner.
`client_id`	The ID of the client that requested the token.
`exp`	The expiry time of the token, in seconds from the UNIX epoch.

Listing 7.7 shows how to process the remaining JSON fields by extracting the resource owner from the `sub` field, the expiry time from the `exp` field, and the scope from the `scope` field. You can also extract other fields of interest, such as the `client_id`, which can be useful information to add to audit logs. Open OAuth2TokenStore.java again and add the `processResponse` method from the listing.

Listing 7.7 Processing the introspection response

```
  private Optional<Token> processResponse(JSONObject response) {
      var expiry = Instant.ofEpochSecond(response.getLong("exp"));    ❶
      var subject = response.getString("sub");                        ❶

      var token = new Token(expiry, subject);
```

```
        token.attributes.put("scope", response.getString("scope"));     (1)
        token.attributes.put("client_id",                               (1)
                response.optString("client_id"));                       (1)

        return Optional.of(token);
    }
```

(1) Extract token attributes from the relevant fields in the response.

Although you used the `sub` field to extract an ID for the user, this may not always be appropriate. The authenticated subject of a token needs to match the entries in the `users` and `permissions` tables in the database that define the access control lists for Natter social spaces. If these don't match, then the requests from a client will be denied even if they have a valid access token. You should check the documentation for your AS to see which field to use to match your existing user IDs.

You can now switch the Natter API to use OAuth2 access tokens by changing the `TokenStore` in Main.java to use the `OAuth2TokenStore`, passing in the URI of your AS's token introspection endpoint and the client ID and secret that you registered for the Natter API (see appendix A for instructions):

```
var introspectionEndpoint =
     URI.create("https://as.example.com:8443/oauth2/introspect");
SecureTokenStore tokenStore = new OAuth2TokenStore(                     (1)
     introspectionEndpoint, clientId, clientSecret);                    (1)
var tokenController = new TokenController(tokenStore);
```

(1) Construct the token store, pointing at your AS.

You should make sure that the AS and the API have the same users and that the AS communicates the username to the API in the `sub` or `username` fields from the introspection response. Otherwise, the API may not be able to match the username returned from token introspection to entries in its access control lists (chapter 3). In many corporate environments, the users will not be stored in a local database but instead in a shared LDAP directory that is maintained by a company's IT department that both the AS and the API have access to, as shown in figure 7.7.

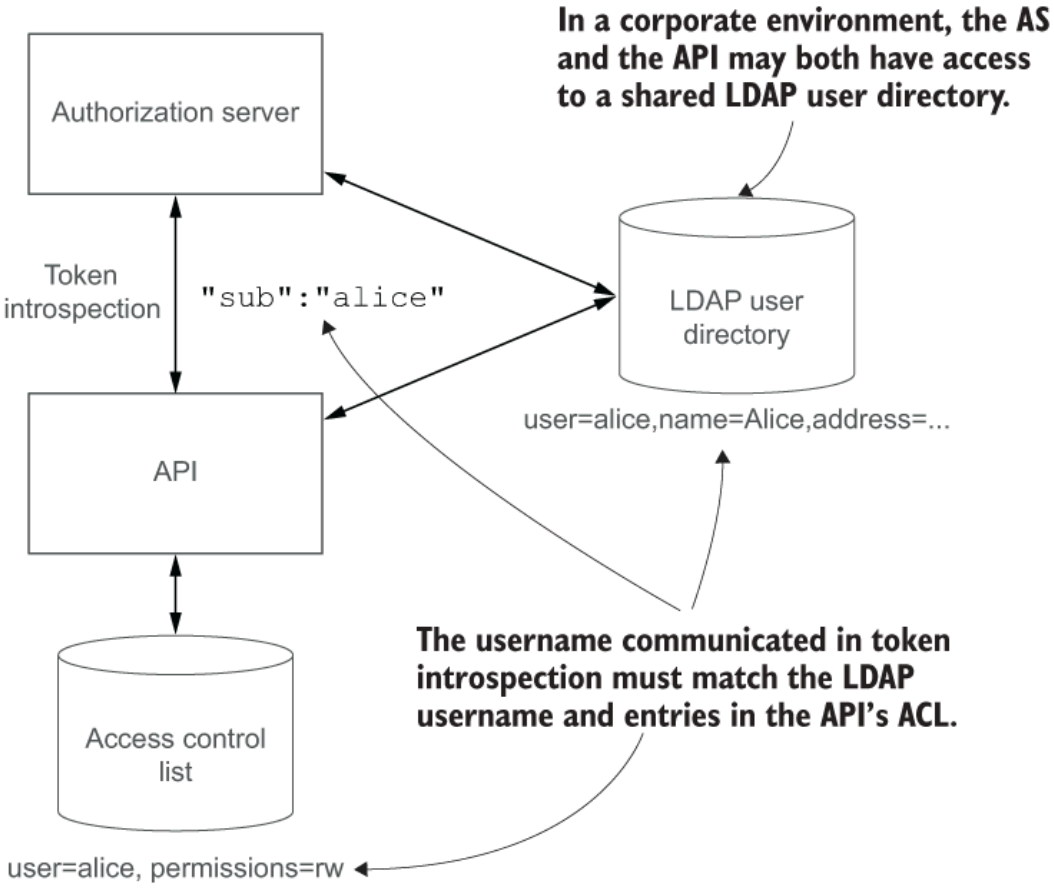

Figure 7.7 In many environments, the AS and the API will both have access to a corporate LDAP directory containing details of all users. In this case, the AS needs to communicate the username to the API so that it can find the matching user entry in LDAP and in its own access control lists.

In other cases, the AS and the API may have different user databases that use different username formats. In this case, the API will need some logic to map the username returned by token introspection into a username that matches its local database and ACLs. For example, if the AS returns the email address of the user, then this could be used to search for a matching user in the local user database. In more loosely coupled architectures, the API may rely entirely on the information returned from the token introspection endpoint and not have access to a user database at all.

Once the AS and the API are on the same page about usernames, you can obtain an access token from the AS and use it to access the Natter API, as in the following example using the ROPC grant:

```
$ curl -u test:password \                                          (1)
  -d 'grant_type=password&scope=create_space+post_message          (1)
➥ &username=demo&password=changeit' \                             (1)
https://openam.example.com:8443/openam/oauth2/access_token
{"access_token":"_Avja0SO-6vAz-caub31eh5RLDU",
```

```
 "scope":"post_message create_space",
  "token_type":"Bearer","expires_in":3599}
$ curl -H 'Content-Type: application/json' \
 -H 'Authorization: Bearer _Avja0SO-6vAz-caub31eh5RLDU' \          ❷
 -d '{"name":"test","owner":"demo"}' https://localhost:4567/spaces
{"name":"test","uri":"/spaces/1"}
```

❶ Obtain an access token using ROPC grant.

❷ Use the access token to perform actions with the Natter API.

Attempting to perform an action that is not allowed by the scope of the access token will result in a 403 Forbidden error due to the access control filters you added at the start of this chapter:

```
$ curl -i -H 'Authorization: Bearer _Avja0SO-6vAz-caub31eh5RLDU' \
  https://localhost:4567/spaces/1/messages
HTTP/1.1 403 Forbidden                                             ❶
Date: Mon, 01 Jul 2019 10:22:17 GMT
WWW-Authenticate: Bearer
➥ error="insufficient_scope",scope="list_messages"                ❷
```

❶ The request is forbidden.

❷ The error message tells the client the scope it requires.

7.4.2 Securing the HTTPS client configuration

Because the API relies entirely on the AS to tell it if an access token is valid, and the scope of access it should grant, it's critical that the connection between the two be secure. While this connection should always be over HTTPS, the default connection settings used by Java are not as secure as they could be:

- The default settings trust server certificates signed by any of the main public certificate authorities (CAs). Typically, the AS will be running on your own internal network and issued with a certificate by a private CA for your organization, so it's unnecessary to trust all of these public CAs.

- The default TLS settings include a wide variety of cipher suites and protocol versions for maximum compatibility. Older versions of TLS, and some cipher suites, have known security weaknesses that should be avoided where possible. You should disable these less secure options and re-enable them only if you must talk to an old server that cannot be upgraded.

TLS cipher suites

A TLS cipher suite is a collection of cryptographic algorithms that work together to create the secure channel between a client and a server. When a TLS connection is first established, the client and server perform a handshake, in which the server authenticates to the client, the client optionally authenticates to the server, and they agree upon a session key to use for subsequent messages. The cipher suite specifies the algorithms to be used for authentication, key exchange, and the block cipher and mode of operation to use for encrypting messages. The cipher suite to use is negotiated as the first part of the handshake.

For example, the TLS 1.2 cipher suite `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` specifies that the two parties will use the Elliptic Curve Diffie-Hellman (ECDH) key agreement algorithm (using ephemeral keys, indicated by the final E), with RSA signatures for authentication, and the agreed session key will be used to encrypt messages using AES in Galois/Counter Mode. (SHA-256 is used as part of the key agreement.)

In TLS 1.3, cipher suites only specify the block cipher and hash function used, such as `TLS_AES_128_GCM_SHA256`. The key exchange and authentication algorithms are negotiated separately.

The latest and most secure version of TLS is version 1.3, which was released in August 2018. This replaced TLS 1.2, released exactly a decade earlier. While TLS 1.3 is a significant improvement over earlier versions of the protocol, it's not yet so widely adopted that support for TLS 1.2 can be dropped completely. TLS 1.2 is still a very secure protocol, but for maximum security you should prefer cipher suites that offer forward secrecy and avoid older algorithms that use AES in CBC mode, because these are more prone to attacks. Mozilla provides recommendations for secure TLS configuration options (**https://wiki.mozilla.org/Security/Server_Side_TLS**), along with a tool for automatically generating configuration files for various web servers,

load balancers, and reverse proxies. The configuration used in this section is based on Mozilla's Intermediate settings. If you know that your AS software is capable of TLS 1.3, then you could opt for the Modern settings and remove the TLS 1.2 support.

DEFINITION A cipher suite offers forward secrecy if the confidentiality of data transmitted using that cipher suite is protected even if one or both of the parties are compromised afterwards. All cipher suites provide forward secrecy in TLS 1.3. In TLS 1.2, these cipher suites start with `TLS_ECDHE_` or `TLS_DHE_`.

To configure the connection to trust only the CA that issued the server certificate used by your AS, you need to create a `javax.net.ssl.TrustManager` that has been initialized with a `KeyStore` that contains only that one CA certificate. For example, if you're using the `mkcert` utility from chapter 3 to generate the certificate for your AS, then you can use the following command to import the root CA certificate into a keystore:

```
$ keytool -import -keystore as.example.com.ca.p12 \
    -alias ca -file "$(mkcert -CAROOT)/rootCA.pem"
```

This will ask you whether you want to trust the root CA certificate and then ask you for a password for the new keystore. Accept the certificate and type in a suitable password, then copy the generated keystore into the Natter project root directory.

Certificate chains

When configuring the trust store for your HTTPS client, you could choose to directly trust the server certificate for that server. Although this seems more secure, it means that whenever the server changes its certificate, the client would need to be updated to trust the new one. Many server certificates are valid for only 90 days. If the server is ever compromised, then the client will continue trusting the compromised certificate until it's manually updated to remove it from the trust store.

To avoid these problems, the server certificate is signed by a CA, which itself has a (self-signed) certificate. When a client connects to the server it receives the server's current certificate during the handshake. To verify this certificate is genuine, it looks up the corresponding CA certificate in

the client trust store and checks that the server certificate was signed by that CA and is not expired or revoked.

In practice, the server certificate is often not signed directly by the CA. Instead, the CA signs certificates for one or more intermediate CAs, which then sign server certificates. The client may therefore have to verify a chain of certificates until it finds a certificate of a root CA that it trusts directly. Because CA certificates might themselves be revoked or expire, in general the client may have to consider multiple possible certificate chains before it finds a valid one. Verifying a certificate chain is complex and error-prone with many subtle details so you should always use a mature library to do this.

In Java, overall TLS settings can be configured explicitly using the `javax.net.ssl.SSLParameters` class[8] (listing 7.8). First construct a new instance of the class, and then use the setter methods such as `setCipherSuites(String[])` that allows TLS versions and cipher suites. The configured parameters can then be passed when building the `HttpClient` object. Open OAuth2TokenStore.java in your editor and update the constructor to configure secure TLS settings.

Listing 7.8 Securing the HTTPS connection

```
import javax.net.ssl.*;
import java.security.*;
import java.net.http.*;
var sslParams = new SSLParameters();
sslParams.setProtocols(                                          (1)
        new String[] { "TLSv1.3", "TLSv1.2" });                  (1)
sslParams.setCipherSuites(new String[] {
        "TLS_AES_128_GCM_SHA256",                                (2)
        "TLS_AES_256_GCM_SHA384",                                (2)
        "TLS_CHACHA20_POLY1305_SHA256",                          (2)

        "TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256",               (3)
        "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256",                 (3)
        "TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384",               (3)
        "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384",                 (3)
        "TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256",         (3)
        "TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256"            (3)
});
sslParams.setUseCipherSuitesOrder(true);
sslParams.setEndpointIdentificationAlgorithm("HTTPS");
try {
    var trustedCerts = KeyStore.getInstance("PKCS12");           (4)
```

```
        trustedCerts.load(                                          ❹
                new FileInputStream("as.example.com.ca.p12"),       ❹
                "changeit".toCharArray());                          ❹
        var tmf = TrustManagerFactory.getInstance("PKIX");          ❹
        tmf.init(trustedCerts);                                     ❹
        var sslContext = SSLContext.getInstance("TLS");             ❹
        sslContext.init(null, tmf.getTrustManagers(), null);        ❹
        this.httpClient = HttpClient.newBuilder()
                .sslParameters(sslParams)                           ❺
                .sslContext(sslContext)                             ❺
                .build();
    } catch (GeneralSecurityException | IOException e) {
        throw new RuntimeException(e);
    }
```

❶ Allow only TLS 1.2 or TLS 1.3.

❷ Configure secure cipher suites for TLS 1.3 . . .

❸ . . . and for TLS 1.2.

❹ The SSLContext should be configured to trust only the CA used by your AS.

❺ Initialize the HttpClient with the chosen TLS parameters.

7.4.3 Token revocation

Just as for token introspection, there is an OAuth2 standard for revoking an access token (**https://tools.ietf.org/html/rfc7009**). While this could be used to implement the `revoke` method in the `OAuth2TokenStore`, the standard only allows the client that was issued a token to revoke it, so the RS (the Natter API in this case) cannot revoke a token on behalf of a client. Clients should directly call the AS to revoke a token, just as they do to get an access token in the first place.

Revoking a token follows the same pattern as token introspection: the client makes a POST request to a revocation endpoint at the AS, passing in the token in the request body, as shown in listing 7.9. The client should include its client credentials to authenticate the request. Only an HTTP status code is returned, so there is no need to parse the response body.

Listing 7.9 Revoking an OAuth access token

```
package com.manning.apisecurityinaction;

import java.net.*;
import java.net.http.*;
import java.net.http.HttpResponse.BodyHandlers;
import java.util.Base64;

import static java.nio.charset.StandardCharsets.UTF_8;

public class RevokeAccessToken {

    private static final URI revocationEndpoint =
            URI.create("https://as.example.com:8443/oauth2/token/revoke")

    public static void main(String...args) throws Exception {

        if (args.length != 3) {
            throw new IllegalArgumentException(
                    "RevokeAccessToken clientId clientSecret token");
        }

        var clientId = args[0];
        var clientSecret = args[1];
        var token = args[2];

        var credentials = URLEncoder.encode(clientId, UTF_8) +    (1)
                ":" + URLEncoder.encode(clientSecret, UTF_8);     (1)
        var authorization = "Basic " + Base64.getEncoder()        (1)
                .encodeToString(credentials.getBytes(UTF_8));     (1)

        var httpClient = HttpClient.newHttpClient();

        var form = "token=" + URLEncoder.encode(token, UTF_8) +   (2)
                "&token_type_hint=access_token";                  (2)

        var httpRequest = HttpRequest.newBuilder()
                .uri(revocationEndpoint)
                .header("Content-Type",
                    "application/x-www-form-urlencoded")
                .header("Authorization", authorization)           (3)
                .POST(HttpRequest.BodyPublishers.ofString(form))
                .build();

        httpClient.send(httpRequest, BodyHandlers.discarding());
    }
}
```

❶ Encode the client's credentials for Basic authentication.

❷ Create the POST body using URL-encoding for the token.

❸ Include the client credentials in the revocation request.

Pop quiz

6. Which standard endpoint is used to determine if an access token is valid?
    1. The access token endpoint
    2. The authorization endpoint
    3. The token revocation endpoint
    4. The token introspection endpoint
7. Which parties are allowed to revoke an access token using the standard revocation endpoint?
    1. Anyone
    2. Only a resource server
    3. Only the client the token was issued to
    4. A resource server or the client the token was issued to

The answers are at the end of the chapter.

7.4.4 JWT access tokens

Though token introspection solves the problem of how the API can determine if an access token is valid and the scope associated with that token, it has a downside: the API must make a call to the AS every time it needs to validate a token. An alternative is to use a self-contained token format such as JWTs that were covered in chapter 6. This allows the API to validate the access token locally without needing to make an HTTPS call to the AS. While there is not yet a standard for JWT-based OAuth2 access tokens (although one is being developed; see **http://mng.bz/5pW4**), it's common for an AS to support this as an option.

To validate a JWT-based access token, the API needs to first authenticate the JWT using a cryptographic key. In chapter 6, you used symmetric HMAC or authenticated encryption algorithms in which the same key is used to both create and verify messages. This means that any party that can verify a JWT is also able to create one that will be trusted by all other parties. Although this is suitable when the API and AS exist within the same trust boundary, it becomes a security risk when the APIs are in different trust boundaries. For example, if the AS is in a different datacenter

to the API, the key must now be shared between those two datacenters. If there are many APIs that need access to the shared key, then the security risk increases even further because an attacker that compromises any API can then create access tokens that will be accepted by all of them.

To avoid these problems, the AS can switch to public key cryptography using digital signatures, as shown in figure 7.8. Rather than having a single shared key, the AS instead has a pair of keys: a private key and a public key. The AS can sign a JWT using the private key, and then anybody with the public key can verify that the signature is genuine. However, the public key cannot be used to create a new signature and so it's safe to share the public key with any API that needs to validate access tokens. For this reason, public key cryptography is also known as asymmetric cryptography, because the holder of a private key can perform different operations to the holder of a public key. Given that only the AS needs to create new access tokens, using public key cryptography for JWTs enforces the principle of least authority (POLA; see chapter 2) as it ensures that APIs can only verify access tokens and not create new ones.

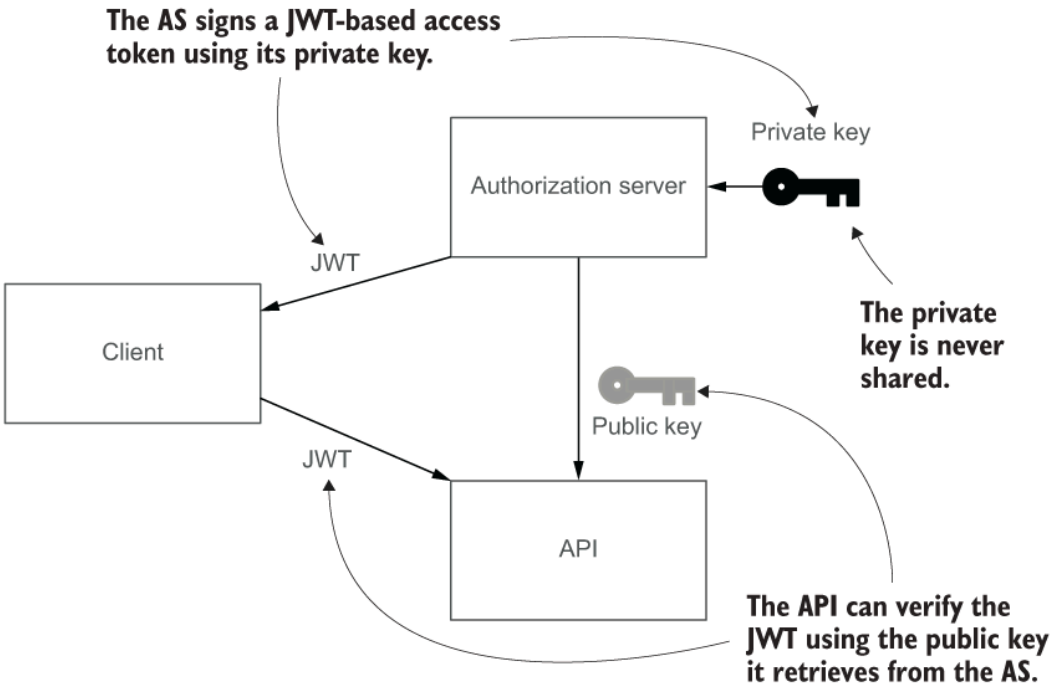

Figure 7.8 When using JWT-based access tokens, the AS signs the JWT using a private key that is known only to the AS. The API can retrieve a corresponding public key from the AS to verify that the JWT is genuine. The public key cannot be used to create a new JWT, ensuring that access tokens can be issued only by the AS.

TIP Although public key cryptography is more secure in this sense, it's also more complicated with more ways to fail. Digital signatures are also

much slower than HMAC and other symmetric algorithms--typically 10-100x slower for equivalent security.

RETRIEVING THE PUBLIC KEY

The API can be directly configured with the public key of the AS. For example, you could create a keystore that contains the public key, which the API can read when it first starts up. Although this will work, it has some disadvantages:

- A Java keystore can only contain certificates, not raw public keys, so the AS would need to create a self-signed certificate purely to allow the public key to be imported into the keystore. This adds complexity that would not otherwise be required.
- If the AS changes its public key, which is recommended, then the keystore will need to be manually updated to list the new public key and remove the old one. Because some access tokens using the old key may still be in use, the keystore may have to list both public keys until those old tokens expire. This means that two manual updates need to be performed: one to add the new public key, and a second update to remove the old public key when it's no longer needed.

Although you could use X.509 certificate chains to establish trust in a key via a certificate authority, just as for HTTPS in section 7.4.2, this would require the certificate chain to be attached to each access token JWT (using the standard `x5c` header described in chapter 6). This would increase the size of the access token beyond reasonable limits--a certificate chain can be several kilobytes in size. Instead, a common solution is for the AS to publish its public key in a JSON document known as a JWK Set (**https://tools.ietf.org/html/rfc7517**). An example JWK Set is shown in listing 7.10 and consists of a JSON object with a single `keys` attribute, whose value is an array of JSON Web Keys (see chapter 6). The API can periodically fetch the JWK Set from an HTTPS URI provided by the AS. The API can trust the public keys in the JWK Set because they were retrieved over HTTPS from a trusted URI, and that HTTPS connection was authenticated using the server certificate presented during the TLS handshake.

Listing 7.10 An example JWK Set

```
{"keys": [                                    ❶
    {
      "kty": "EC",                            ❷
```

```
      "kid": "I4x/IijvdDsUZMghwNq2gC/7pYQ=",
      "use": "sig",
      "x": "k5wSvW_6JhOuCj-9PdDWdEA4oH90RSmC2GTliiUHAhXj6rmTdE2S-
➥ _zGmMFxufuV",
      "y": "XfbR-tRoVcZMCoUrkKtuZUIyfCgAy8b0FWnPZqevwpdoTzGQBOXSN
➥ i6uItN_o4tH",
      "crv": "P-384",
      "alg": "ES384"
    },
    {
      "kty": "RSA",                                    ❸
      "kid": "wU3ifIIaLOUAReRB/FG6eM1P1QM=",
      "use": "sig",
      "n": "10iGQ5l5IdqBP1l5wb5BDBZpSyLs4y_Um-kGv_se0BkRkwMZavGD_Nqjq8x3-
➥ fKNI45nU7E7COAh8gjn6LCXfug57EQfi0gOgKhOhVcLmKqIEXPmqeagvMndsXWIy6k8WP
➥ PwBzSkN5PDLKBXKG_X1BwVvOE9276nrx6lJq3CgNbmiEihovNt_6g5pCxiSarIk2uaG3T
➥ 3Ve6hUJrM0W35QmqrNM9rL3laPgXtCuz4sJJN3rGnQq_25YbUawW9L1MTVbqKxWiyN5Wb
➥ XoWUg8to1DhoQnXzDymIMhFa45NTLhxtdH9CDprXWXWBaWzo8mIFes5yI4AJW4ZSg1PPO
➥ 2UJSQ",
      "e": "AQAB",
      "alg": "RS256"
    }
]}
```

❶ The JWK Set has a “keys” attribute, which is an array of JSON Web Keys.

❷ An elliptic curve public key

❸ An RSA public key

Many JWT libraries have built-in support for retrieving keys from a JWK Set over HTTPS, including periodically refreshing them. For example, the Nimbus JWT library that you used in chapter 6 supports retrieving keys from a JWK Set URI using the `RemoteJWKSet` class:

```
var jwkSetUri = URI.create("https://as.example.com:8443/jwks_uri");
var jwkSet = new RemoteJWKSet(jwkSetUri);
```

Listing 7.11 shows the configuration of a new `SignedJwtAccessTokenStore` that will validate an access token as a signed JWT. The constructor takes a URI for the endpoint on the AS to retrieve the JWK Set from and constructs a `RemoteJWKSet` based on this. It also takes in the expected issuer and audience values of the JWT, and the

JWS signature algorithm that will be used. As you'll recall from chapter 6, there are attacks on JWT verification if the wrong algorithm is used, so you should always strictly validate that the algorithm header has an expected value. Open the src/main/java/com/manning/apisecurityinaction/token folder and create a new file SignedJwtAccessTokenStore.java with the contents of listing 7.11. You'll fill in the details of the `read` method shortly.

TIP If the AS supports discovery (see section 7.2.3), then it may advertise its JWK Set URI as the `jwks_uri` field of the discovery document.

Listing 7.11 The SignedJwtAccessTokenStore

```
package com.manning.apisecurityinaction.token;
import com.nimbusds.jose.*;
import com.nimbusds.jose.jwk.source.*;
import com.nimbusds.jose.proc.*;
import com.nimbusds.jwt.proc.DefaultJWTProcessor;
import spark.Request;
import java.net.*;
import java.text.ParseException;
import java.util.Optional;

public class SignedJwtAccessTokenStore implements SecureTokenStore {

    private final String expectedIssuer;
    private final String expectedAudience;
    private final JWSAlgorithm signatureAlgorithm;
    private final JWKSource<SecurityContext> jwkSource;

    public SignedJwtAccessTokenStore(String expectedIssuer,
                                     String expectedAudience,
                                     JWSAlgorithm signatureAlgorithm,
                                     URI jwkSetUri)
            throws MalformedURLException {
        this.expectedIssuer = expectedIssuer;                    (1)
        this.expectedAudience = expectedAudience;                (1)
        this.signatureAlgorithm = signatureAlgorithm;            (1)
        this.jwkSource = new RemoteJWKSet<>(jwkSetUri.toURL());  (2)
    }

    @Override
    public String create(Request request, Token token) {
        throw new UnsupportedOperationException();
    }
```

```
    @Override
    public void revoke(Request request, String tokenId) {
        throw new UnsupportedOperationException();
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
        // See listing 7.12
    }
}
```

❶ Configure the expected issuer, audience, and JWS algorithm.

❷ Construct a RemoteJWKSet to retrieve keys from the JWK Set URI.

A JWT access token can be validated by configuring the processor class to use the `RemoteJWKSet` as the source for verification keys (`ES256` is an example of a JWS signature algorithm):

```
    var verifier = new DefaultJWTProcessor<>();
    var keySelector = new JWSVerificationKeySelector<>(
            JWSAlgorithm.ES256, jwkSet);
    verifier.setJWSKeySelector(keySelector);
    var claims = verifier.process(tokenId, null);
```

After verifying the signature and the expiry time of the JWT, the processor returns the JWT Claims Set. You can then verify that the other claims are correct. You should check that the JWT was issued by the AS by validating the `iss` claim, and that the access token is meant for this API by ensuring that an identifier for the API appears in the audience (`aud`) claim (listing 7.12).

In the normal OAuth2 flow, the AS is not informed by the client which APIs it intends to use the access token for,[9] and so the audience claim can vary from one AS to another. Consult the documentation for your AS software to configure the intended audience. Another area of disagreement between AS software is in how the scope of the token is communicated. Some AS software produces a string `scope` claim, whereas others produce a JSON array of strings. Some others may use a different field entirely, such as `scp` or `scopes`. Listing 7.12 shows how to handle a scope claim that may either be a string or an array of strings. Open SignedJwtAccessTokenStore.java in your editor again and update the `read` method based on the listing.

Listing 7.12 Validating signed JWT access tokens

```
@Override
public Optional<Token> read(Request request, String tokenId) {
    try {
        var verifier = new DefaultJWTProcessor<>();
        var keySelector = new JWSVerificationKeySelector<>(
                signatureAlgorithm, jwkSource);
        verifier.setJWSKeySelector(keySelector);

        var claims = verifier.process(tokenId, null);          (1)

        if (!issuer.equals(claims.getIssuer())) {              (2)
            return Optional.empty();                           (2)
        }
        if (!claims.getAudience().contains(audience)) {        (2)
            return Optional.empty();                           (2)
        }

        var expiry = claims.getExpirationTime().toInstant();   (3)
        var subject = claims.getSubject();                     (3)
        var token = new Token(expiry, subject);                (3)

        String scope;                                          (4)
        try {                                                  (4)
            scope = claims.getStringClaim("scope");            (4)
        } catch (ParseException e) {                           (4)
            scope = String.join(" ",                           (4)
                claims.getStringListClaim("scope"));           (4)
        }                                                      (4)
        token.attributes.put("scope", scope);                  (4)
        return Optional.of(token);

    } catch (ParseException | BadJOSEException | JOSEException e) {
        return Optional.empty();
    }
}
```

(1) Verify the signature first.

(2) Ensure the issuer and audience have expected values.

(3) Extract the JWT subject and expiry time.

(4) The scope may be either a string or an array of strings.

CHOOSING A SIGNATURE ALGORITHM

The JWS standard that JWT uses for signatures supports many different public key signature algorithms, summarized in table 7.2. Because public key signature algorithms are expensive and usually limited in the amount of data that can be signed, the contents of the JWT is first hashed using a cryptographic hash function and then the hash value is signed. JWS provides variants for different hash functions when using the same underlying signature algorithm. All the allowed hash functions provide adequate security, but SHA-512 is the most secure and may be slightly faster than the other choices on 64-bit systems. The exception to this rule is when using ECDSA signatures, because JWS specifies elliptic curves to use along with each hash function; the curve used with SHA-512 has a significant performance penalty compared with the curve used for SHA-256.

Table 7.2 JWS signature algorithms

JWS Algorithm	Hash function	Signature algorithm
`RS256`	SHA-256	RSA with PKCS#1 v1.5 padding
`RS384`	SHA-384	
`RS512`	SHA-512	
`PS256`	SHA-256	RSA with PSS padding
`PS384`	SHA-384	
`PS512`	SHA-512	
`ES256`	SHA-256	ECDSA with the NIST P-256 curve
`ES384`	SHA-384	ECDSA with the NIST P-384 curve
`ES512`	SHA-512	ECDSA with the NIST P-521 curve
`EdDSA`	SHA-512 / SHAKE256	EdDSA with either the Ed25519 or Ed448 curves

Of these choices, the best is `EdDSA`, based on the Edwards Curve Digital Signature Algorithm (**https://tools.ietf.org/html/rfc8037**). EdDSA signatures are fast to produce and verify, produce compact signatures, and are designed to be implemented securely against side-channel attacks. Not all JWT libraries or AS software supports EdDSA signatures yet. The older ECDSA standard for elliptic curve digital signatures has wider support, and shares some of the same properties as EdDSA, but is slightly slower and harder to implement securely.

WARNING ECDSA signatures require a unique random nonce for each signature. If a nonce is repeated, or even just a few bits are not com-

pletely random, then the private key can be reconstructed from the signature values. This kind of bug was used to hack the Sony PlayStation 3, steal Bitcoin cryptocurrency from wallets on Android mobile phones, among many other cases. Deterministic ECDSA signatures (**https://tools.ietf.org/html/rfc6979**) can be used to prevent this, if your library supports them. EdDSA signatures are also immune to this issue.

RSA signatures are expensive to produce, especially for secure key sizes (a 3072-bit RSA key is roughly equivalent to a 256-bit elliptic curve key or a 128-bit HMAC key) and produce much larger signatures than the other options, resulting in larger JWTs. On the other hand, RSA signatures can be validated very quickly. The variants of RSA using PSS padding should be preferred over those using the older PKCS#1 version 1.5 padding but may not be supported by all libraries.

7.4.5 Encrypted JWT access tokens

In chapter 6, you learned that authenticated encryption can be used to provide the benefits of encryption to hide confidential attributes and authentication to ensure that a JWT is genuine and has not been tampered with. Encrypted JWTs can be useful for access tokens too, because the AS may want to include attributes in the access token that are useful for the API for making access control decisions, but which should be kept confidential from third-party clients or from the user themselves. For example, the AS may include the resource owner's email address in the token for use by the API, but this information should not be leaked to the third-party client. In this case the AS can encrypt the access token JWT by using an encryption key that only the API can decrypt.

Unfortunately, none of the public key encryption algorithms supported by the JWT standards provide authenticated encryption,**10** because this is less often implemented for public key cryptography. The supported algorithms provide only confidentiality and so must be combined with a digital signature to ensure the JWT is not tampered with or forged. This is done by first signing the claims to produce a signed JWT, and then encrypting that signed JWT to produce a nested JOSE structure (figure 7.9). The downside is that the resulting JWT is much larger than it would be if it was just signed and requires two expensive public key operations to first decrypt the outer encrypted JWE and then verify the inner signed JWT. You shouldn't use the same key for encryption and signing, even if the algorithms are compatible.

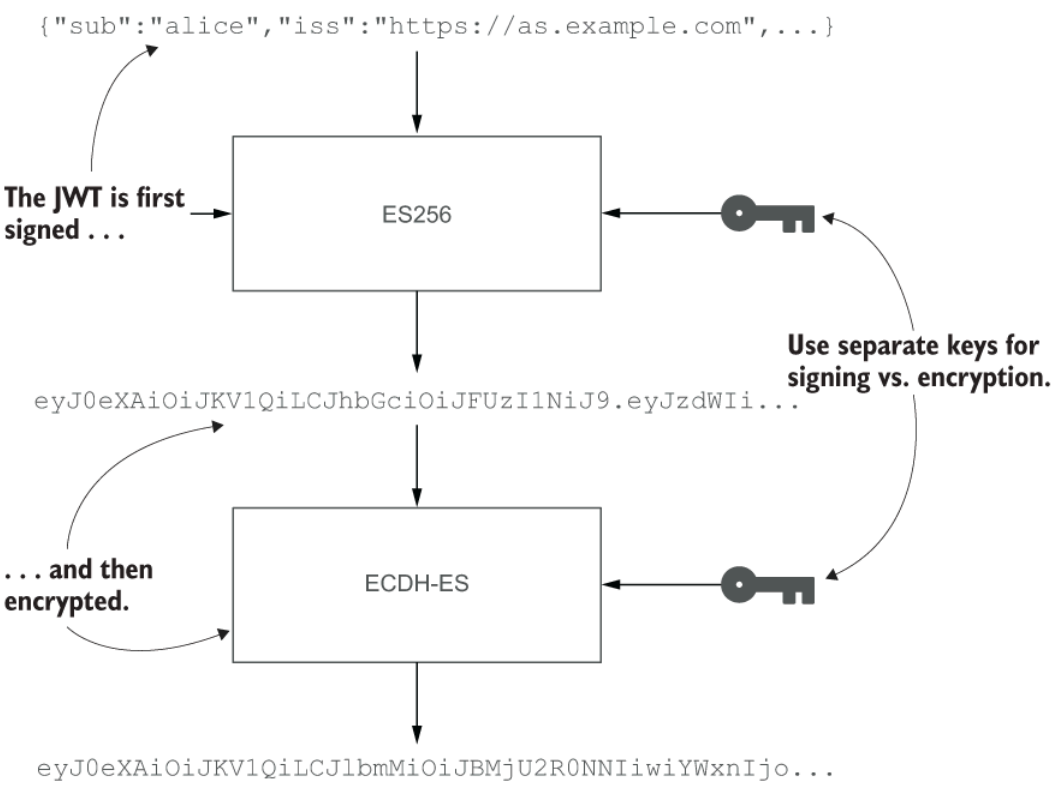

Figure 7.9 When using public key cryptography, a JWT needs to be first signed and then encrypted to ensure confidentiality and integrity as no standard algorithm provides both properties. You should use separate keys for signing and encryption even if the algorithms are compatible.

The JWE specifications include several public key encryption algorithms, shown in table 7.3. The details of the algorithms can be complicated, and several variations are included. If your software supports it, it's best to avoid the RSA encryption algorithms entirely and opt for `ECDH-ES` encryption. ECDH-ES is based on Elliptic Curve Diffie-Hellman key agreement, and is a secure and performant choice, especially when used with the X25519 or X448 elliptic curves (**https://tools.ietf.org/html/rfc8037**), but these are not yet widely supported by JWT libraries.

Table 7.3 JOSE public key encryption algorithms

JWE Algorithm	Details	Comments
`RSA1_5`	RSA with PKCS#1 v1.5 padding	This mode is insecure and should not be used.
`RSA-OAEP`	RSA with OAEP padding using SHA-1	OAEP is secure but RSA decryption is slow, and encryption produces large JWTs.
`RSA-OAEP-256`	RSA with OAEP padding using SHA-256	
`ECDH-ES`	Elliptic Curve Integrated Encryption Scheme (ECIES)	A secure encryption algorithm but the `epk` header it adds can be bulky. Best when used with the X25519 or X448 curves.
`ECDH-ES+A128KW`	ECDH-ES with an extra AES key-wrapping step	
`ECDH-ES+A192KW`		
`ECDH-ES+A256KW`		

WARNING Most of the JWE algorithms are secure, apart from `RSA1_5` which uses the older PKCS#1 version 1.5 padding algorithm. There are known attacks against this algorithm, so you should not use it. This padding mode was replaced by Optimal Asymmetric Encryption Padding (OAEP) that was standardized in version 2 of PKCS#1. OAEP uses a hash function internally, so there are two variants included in JWE: one using SHA-1, and one using SHA-256. Because SHA-1 is no longer considered secure, you should prefer the SHA-256 variant, although there are no known attacks against it when used with OAEP. However, even OAEP has some downsides because it's a complicated algorithm and less widely implemented. RSA encryption also produces larger ciphertext than other modes and the decryption operation is very slow, which is a problem for an access token that may need to be decrypted many times.

7.4.6 Letting the AS decrypt the tokens

An alternative to using public key signing and encryption would be for the AS to encrypt access tokens with a symmetric authenticated encryption algorithm, such as the ones you learned about in chapter 6. Rather than sharing this symmetric key with every API, they instead call the token introspection endpoint to validate the token rather than verifying it

locally. Because the AS does not need to perform a database lookup to validate the token, it may be easier to horizontally scale the AS in this case by adding more servers to handle increased traffic.

This pattern allows the format of access tokens to change over time because only the AS validates tokens. In software engineering terms, the choice of token format is encapsulated by the AS and hidden from resource servers, while with public key signed JWTs, each API knows how to validate tokens, making it much harder to change the representation later. More sophisticated patterns for managing access tokens for microservice environments are covered in part 4.

Pop quiz

8. Which key is used to validate a public key signature?
    1. The public key
    2. The private key

The answer is at the end of the chapter.

7.5 Single sign-on

One of the advantages of OAuth2 is the ability to centralize authentication of users at the AS, providing a single sign-on (SSO) experience (figure 7.10). When the user's client needs to access an API, it redirects the user to the AS authorization endpoint to get an access token. At this point the AS authenticates the user and asks for consent for the client to be allowed access. Because this happens within a web browser, the AS typically creates a session cookie, so that the user does not have to login again.

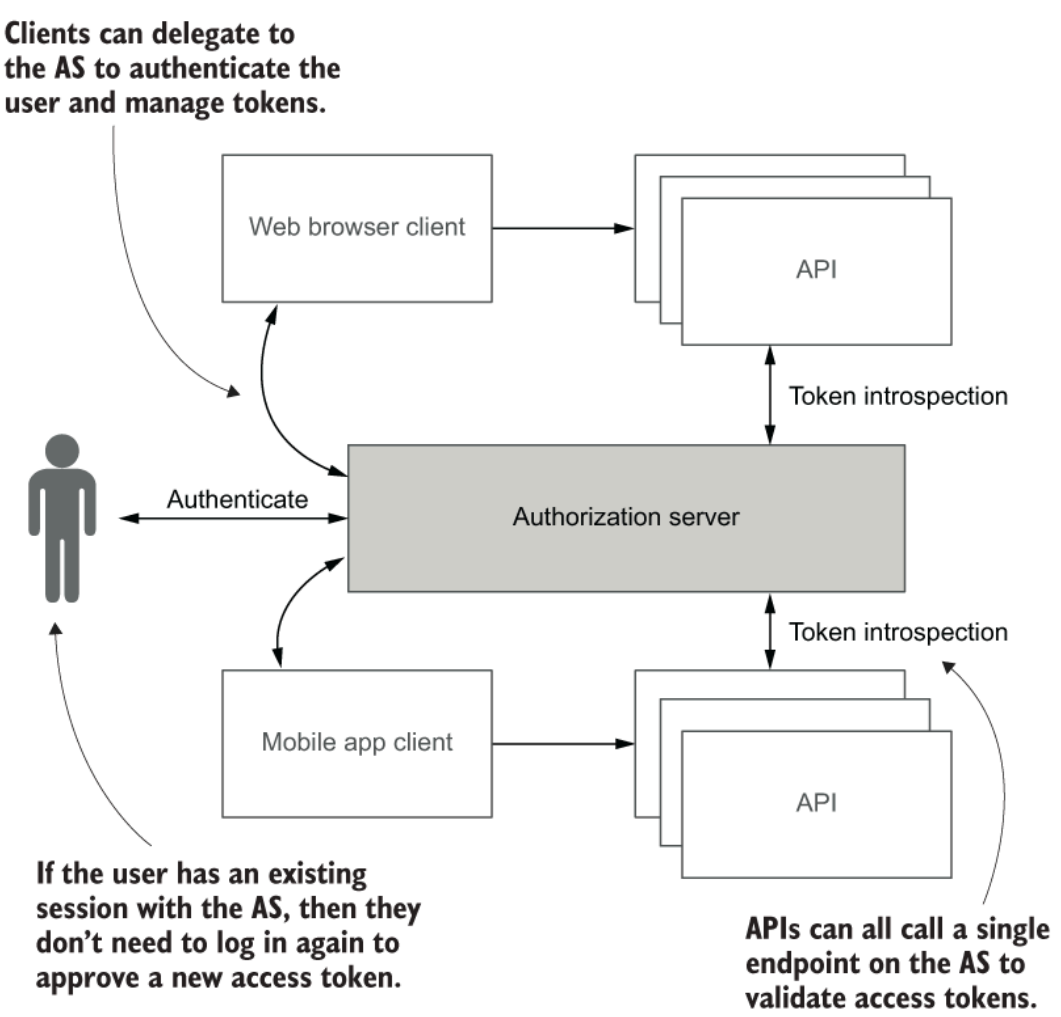

Figure 7.10 OAuth2 enables single sign-on for users. As clients delegate to the AS to get access tokens, the AS is responsible for authenticating all users. If the user has an existing session with the AS, then they don't need to be authenticated again, providing a seamless SSO experience.

If the user then starts using a different client, such as a different web application, they will be redirected to the AS again. But this time the AS will see the existing session cookie and won't prompt the user to log in. This even works for mobile apps from different developers if they are installed on the same device and use the system browser for OAuth flows, as recommended in section 7.3. The AS may also remember which scopes a user has granted to clients, allowing the consent screen to be skipped when a user returns to that client. In this way, OAuth can provide a seamless SSO experience for users replacing traditional SSO solutions. When the user logs out, the client can revoke their access or refresh token using the OAuth token revocation endpoint, which will prevent further access.

WARNING Though it might be tempting to reuse a single access token to provide access to many different APIs within an organization, this increases the risk if a token is ever stolen. Prefer to use separate access tokens for each different API.

7.6 OpenID Connect

OAuth can provide basic SSO functionality, but the primary focus is on delegated third-party access to APIs rather than user identity or session management. The OpenID Connect (OIDC) suite of standards (**https://openid.net/developers/specs/**) extend OAuth2 with several features:

- A standard way to retrieve identity information about a user, such as their name, email address, postal address, and telephone number. The client can access a UserInfo endpoint to retrieve identity claims as JSON using an OAuth2 access token with standard OIDC scopes.
- A way for the client to request that the user is authenticated even if they have an existing session, and to ask for them to be authenticated in a particular way, such as with two-factor authentication. While obtaining an OAuth2 access token may involve user authentication, it's not guaranteed that the user was even present when the token was issued or how recently they logged in. OAuth2 is primarily a delegated access protocol, whereas OIDC provides a full authentication protocol. If the client needs to positively authenticate a user, then OIDC should be used.
- Extensions for session management and logout, allowing clients to be notified when a user logs out of their session at the AS, enabling the user to log out of all clients at once (known as single logout).

Although OIDC is an extension of OAuth, it rearranges the pieces a bit because the API that the client wants to access (the UserInfo endpoint) is part of the AS itself (figure 7.11). In a normal OAuth2 flow, the client would first talk to the AS to obtain an access token and then talk to the API on a separate resource server.

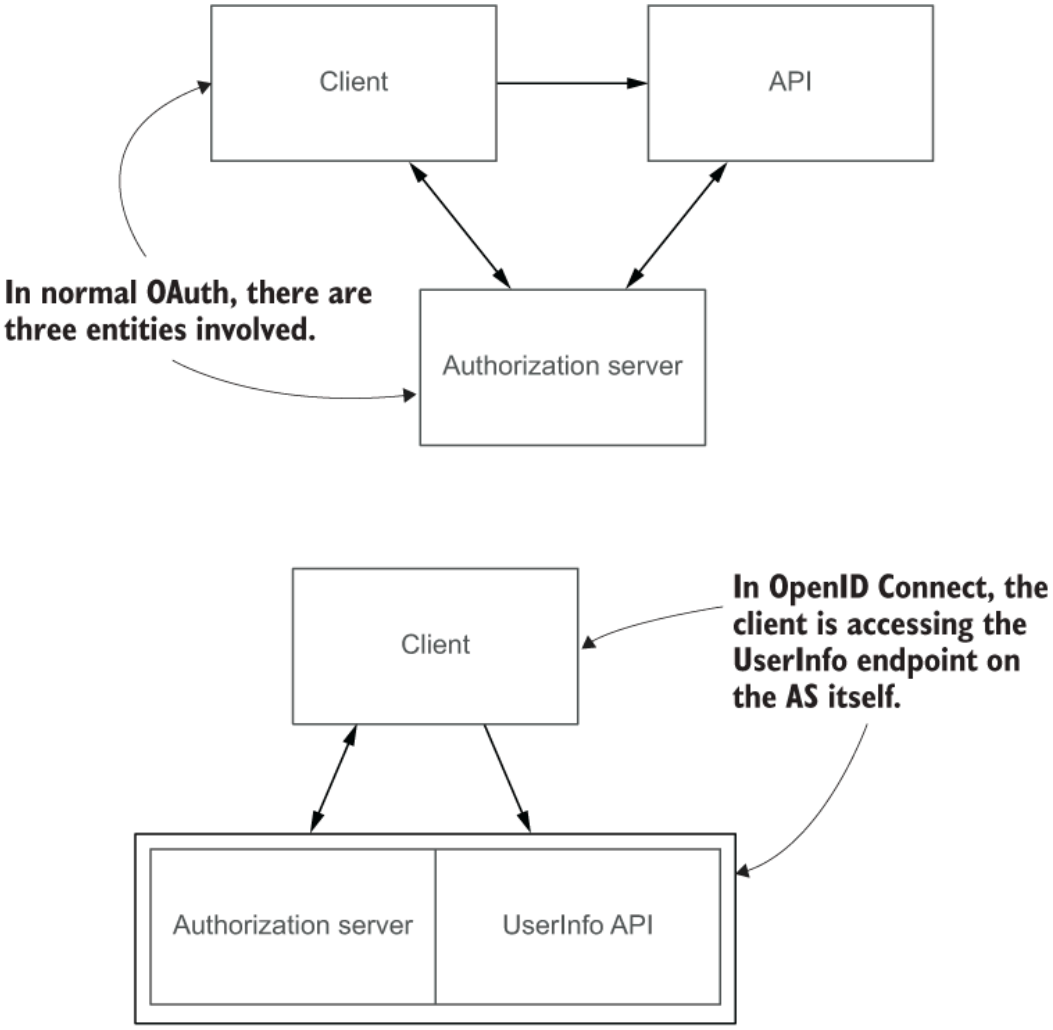

Figure 7.11 In OpenID Connect, the client accesses APIs on the AS itself, so there are only two entities involved compared to the three in normal OAuth. The client is known as the Relying Party (RP), while the combined AS and API is known as an OpenID Provider (OP).

DEFINITION In OIDC, the AS and RS are combined into a single entity known as an OpenID Provider (OP). The client is known as a Relying Party (RP).

The most common use of OIDC is for a website or app to delegate authentication to a third-party identity provider. If you've ever logged into a website using your Google or Facebook account, you're using OIDC behind the scenes, and many large social media companies now support this.

7.6.1 ID tokens

If you follow the OAuth2 recommendations in this chapter, then finding out who a user is involves three roundtrips to the AS for the client:

1. First, the client needs to call the authorization endpoint to get an authorization code.

2. Then the client exchanges the code for an access token.
3. Finally, the client can use the access token to call the UserInfo endpoint to retrieve the identity claims for the user.

This is a lot of overhead before you even know the user's name, so OIDC provides a way to return some of the identity and authentication claims about a user as a new type of token known as an ID token, which is a signed and optionally encrypted JWT. This token can be returned directly from the token endpoint in step 2, or even directly from the authorization endpoint in step 1, in a variant of the implicit flow. There is also a hybrid flow in which the authorization endpoint returns an ID token directly along with an authorization code that the client can then exchange for an access token.

DEFINITION An ID token is a signed and optionally encrypted JWT that contains identity and authentication claims about a user.

To validate an ID token, the client should first process the token as a JWT, decrypting it if necessary and verifying the signature. When a client registers with an OIDC provider, it specifies the ID token signing and encryption algorithms it wants to use and can supply public keys to be used for encryption, so the client should ensure that the received ID token uses these algorithms. The client should then verify the standard JWT claims in the ID token, such as the expiry, issuer, and audience values as described in chapter 6. OIDC defines several additional claims that should also be verified, described in table 7.4.

Table 7.4 ID token standard claims

Claim	Purpose	Notes
`azp`	Authorized Party	An ID token can be shared with more than one party and so have multiple values in the audience claim. The `azp` claim lists the client the ID token was initially issued to. A client directly interacting with an OIDC provider should verify that it's the authorized party if more than one party is in the audience.
`auth_time`	User authentication time	The time at which the user was authenticated as seconds from the UNIX epoch.
`nonce`	Anti-replay nonce	A unique random value that the client sends in the authentication request. The client should verify that the same value is included in the ID token to prevent replay attacks--see section 7.6.2 for details.
`acr`	Authentication context Class Reference	Indicates the overall strength of the user authentication performed. This is a string and specific values are defined by the OP or by other standards.
`amr`	Authentication Methods References	An array of strings indicating the specific methods used. For example, it might contain `["password", "otp"]` to indicate that the user supplied a password and a one-time password.

When requesting authentication, the client can use extra parameters to the authorization endpoint to indicate how the user should be authenticated. For example, the `max_time` parameter can be used to indicate how recently the user must have authenticated to be allowed to reuse an existing login session at the OP, and the `acr_values` parameter can be used to indicate acceptable authentication levels of assurance. The `prompt=login` parameter can be used to force reauthentication even if the user has an existing session that would satisfy any other constraints specified in the authentication request, while `prompt=none` can be used to check if the user is currently logged in without authenticating them if they are not.

WARNING Just because a client requested that a user be authenticated in a certain way does not mean that they will be. Because the request parameters are exposed as URL query parameters in a redirect, the user could alter them to remove some constraints. The OP may not be able to

satisfy all requests for other reasons. The client should always check the claims in an ID token to make sure that any constraints were satisfied.

7.6.2 Hardening OIDC

While an ID token is protected against tampering by the cryptographic signature, there are still several possible attacks when an ID token is passed back to the client in the URL from the authorization endpoint in either the implicit or hybrid flows:

- The ID token might be stolen by a malicious script running in the same browser, or it might leak in server access logs or the HTTP `Referer` header. Although an ID token does not grant access to any API, it may contain personal or sensitive information about the user that should be protected.
- An attacker may be able to capture an ID token from a legitimate login attempt and then replay it later to attempt to login as a different user. A cryptographic signature guarantees only that the ID token was issued by the correct OP but does not by itself guarantee that it was issued in response to this specific request.

The simplest defense against these attacks is to use the authorization code flow with PKCE as recommended for all OAuth2 flows. In this case the ID token is only issued by the OP from the token endpoint in response to a direct HTTPS request from the client. If you decide to use a hybrid flow to receive an ID token directly in the redirect back from the authorization endpoint, then OIDC includes several protections that can be employed to harden the flow:

- The client can include a random `nonce` parameter in the request and verify that the same nonce is included in the ID token that is received in response. This prevents replay attacks as the nonce in a replayed ID token will not match the fresh value sent in the new request. The nonce should be randomly generated and stored on the client just like the OAuth `state` parameter and the PKCE `code_challenge`. (Note that the nonce parameter is unrelated to a nonce used in encryption as covered in chapter 6.)

- The client can request that the ID token is encrypted using a public key supplied during registration or using AES encryption with a key derived from the client secret. This prevents sensitive personal information being exposed if the ID token is intercepted. Encryption alone does not prevent replay attacks, so an OIDC nonce should still be used in this case.
- The ID token can include `c_hash` and `at_hash` claims that contain cryptographic hashes of the authorization code and access token associated with a request. The client can compare these to the actual authorization code and access token it receives to make sure that they match. Together with the nonce and cryptographic signature, this effectively prevents an attacker swapping the authorization code or access token in the redirect URL when using the hybrid or implicit flows.

TIP You can use the same random value for the OAuth `state` and OIDC `nonce` parameters to avoid having to generate and store both on the client.

The additional protections provided by OIDC can mitigate many of the problems with the implicit grant. But they come at a cost of increased complexity compared with the authorization code grant with PKCE, because the client must perform several complex cryptographic operations and check many details of the ID token during validation. With the auth code flow and PKCE, the checks are performed by the OP when the code is exchanged for access and ID tokens.

7.6.3 Passing an ID token to an API

Given that an ID token is a JWT and is intended to authenticate a user, it's tempting to use them for authenticating users to your API. This can be a convenient pattern for first-party clients, because the ID token can be used directly as a stateless session token. For example, the Natter web UI could use OIDC to authenticate a user and then store the ID token as a cookie or in local storage. The Natter API would then be configured to accept the ID token as a JWT, verifying it with the public key from the OP. An ID token is not appropriate as a replacement for access tokens when dealing with third-party clients for the following reasons:

- ID tokens are not scoped, and the user is asked only for consent for the client to access their identity information. If the ID token can be used to access APIs then any client with an ID token can act as if they are the user without any restrictions.

- An ID token authenticates a user to the client and is not intended to be used by that client to access an API. For example, imagine if Google allowed access to its APIs based on an ID token. In that case, any website that allowed its users to log in with their Google account (using OIDC) would then be able to replay the ID token back to Google's own APIs to access the user's data without their consent.
- To prevent these kinds of attacks, an ID token has an audience claim that only lists the client. An API should reject any JWT that does not list that API in the audience.
- If you're using the implicit or hybrid flows, then the ID token is exposed in the URL during the redirect back from the OP. When an ID token is used for access control, this has the same risks as including an access token in the URL as the token may leak or be stolen.

You should therefore not use ID tokens to grant access to an API.

NOTE Never use ID tokens for access control for third-party clients. Use access tokens for access and ID tokens for identity. ID tokens are like usernames; access tokens are like passwords.

Although you shouldn't use an ID token to allow access to an API, you may need to look up identity information about a user while processing an API request or need to enforce specific authentication requirements. For example, an API for initiating financial transactions may want assurance that the user has been freshly authenticated using a strong authentication mechanism. Although this information can be returned from a token introspection request, this is not always supported by all authorization server software. OIDC ID tokens provide a standard token format to verify these requirements. In this case, you may want to let the client pass in a signed ID token that it has obtained from a trusted OP. When this is allowed, the API should accept the ID token only in addition to a normal access token and make all access control decisions based on the access token.

When the API needs to access claims in the ID token, it should first verify that it's from a trusted OP by validating the signature and issuer claims. It should also ensure that the subject of the ID token exactly matches the resource owner of the access token or that there is some other trust relationship between them. Ideally, the API should then ensure that its own identifier is in the audience of the ID token and that the client's identifier is the authorized party (`azp` claim), but not all OP software supports setting these values correctly in this case. Listing 7.13 shows an example of

validating the claims in an ID token against those in an access token that has already been used to authenticate the request. Refer to the `SignedJwtAccessToken` store for details on configuring the JWT verifier.

Listing 7.13 Validating an ID token

```
var idToken = request.headers("X-ID-Token");                          (1)
var claims = verifier.process(idToken, null);                         (1)

if (!expectedIssuer.equals(claims.getIssuer())) {                     (2)
    throw new IllegalArgumentException(                               (2)
            "invalid id token issuer");                               (2)
}
if (!claims.getAudience().contains(expectedAudience)) {               (2)
    throw new IllegalArgumentException(                               (2)
            "invalid id token audience");                             (2)
}

var client = request.attribute("client_id");                          (3)
var azp = claims.getStringClaim("azp");                               (3)
if (client != null && azp != null && !azp.equals(client)) {           (3)
    throw new IllegalArgumentException(                               (3)
            "client is not authorized party");                        (3)
}

var subject = request.attribute("subject");                           (4)
if (!subject.equals(claims.getSubject())) {                           (4)
    throw new IllegalArgumentException(                               (4)
            "subject does not match id token");                       (4)
}

request.attribute("id_token.claims", claims);                         (5)
```

(1) Extract the ID token from the request and verify the signature.

(2) Ensure the token is from a trusted issuer and that this API is the intended audience.

(3) If the ID token has an azp claim, then ensure it's for the same client that is calling the API.

(4) Check that the subject of the ID token matches the resource owner of the access token.

❺ Store the verified ID token claims in the request attributes for further processing.

Answers to pop quiz questions

1. d and e. Whether scopes or permissions are more fine-grained varies from case to case.
2. a and e. The implicit grant is discouraged because of the risk of access tokens being stolen. The ROPC grant is discouraged because the client learns the user's password.
3. a. Mobile apps should be public clients because any credentials embedded in the app download can be easily extracted by users.
4. a. Claimed HTTPS URIs are more secure.
5. True. PKCE provides security benefits in all cases and should always be used.
6. d.
7. c.
8. a. The public key is used to validate a signature.

Summary

- Scoped tokens allow clients to be given access to some parts of your API but not others, allowing users to delegate limited access to third-party apps and services.
- The OAuth2 standard provides a framework for third-party clients to register with your API and negotiate access with user consent.
- All user-facing API clients should use the authorization code grant with PKCE to obtain access tokens, whether they are traditional web apps, SPAs, mobile apps, or desktop apps. The implicit grant should no longer be used.
- The standard token introspection endpoint can be used to validate an access token, or JWT-based access tokens can be used to reduce network roundtrips. Refresh tokens can be used to keep token lifetimes short without disrupting the user experience.
- The OpenID Connect standard builds on top of OAuth2, providing a comprehensive framework for offloading user authentication to a dedicated service. ID tokens can be used for user identification but should be avoided for access control.

---

[1] In some countries, banks are being required to provide secure API access to transactions and payment services to third-party apps and ser-

vices. The UK's Open Banking initiative and the European Payment Services Directive 2 (PSD2) regulations are examples, both of which mandate the use of OAuth2.

2. An alternative way to eliminate this risk is to ensure that any newly issued token contains only scopes that are in the token used to call the login endpoint. I'll leave this as an exercise.

3. Projects such as SELinux (**https://selinuxproject.org/page/Main_Page**) and AppArmor (**https://apparmor .net/**) bring mandatory access controls to Linux.

4. A possible solution to this is to dynamically register each individual instance of the application as a new client when it starts up so that each gets its own unique credentials. See chapter 12 of OAuth2 in Action (Manning, 2017) for details.

5. AS software that supports the OpenID Connect standard may use the path /.well-known/openid-configuration instead. It is recommended to check both locations.

6. The older 302 Found status code is also often used, and there is little difference between them.

7. There is an alternative method in which the client sends the original verifier as the challenge, but this is less secure.

8. Recall from chapter 3 that earlier versions of TLS were called SSL, and this terminology is still widespread.

9. As you might expect by now, there is a proposal to allow the client to indicate the resource servers it intends to access: **http://mng.bz/6ANG**

10. **I have proposed adding public key authenticated encryption to JOSE and JWT, but the proposal is still a draft at this stage. See http://mng.bz/oRGN**.