8 Identity-based access control

This chapter covers

- Organizing users into groups
- Simplifying permissions with role-based access control
- Implementing more complex policies with attribute-based access control
- Centralizing policy management with a policy engine

As Natter has grown, the number of access control list (ACL; chapter 3) entries has grown too. ACLs are simple, but as the number of users and objects that can be accessed through an API grows, the number of ACL entries grows along with them. If you have a million users and a million objects, then in the worst case you could end up with a billion ACL entries listing the individual permissions of each user for each object. Though that approach can work with fewer users, it becomes more of a problem as the user base grows. This problem is particularly bad if permissions are centrally managed by a system administrator (mandatory access control, or MAC, as discussed in chapter 7), rather than determined by individual users (discretionary access control, or DAC). If permissions are not removed when no longer required, users can end up accumulating privileges, violating the principle of least privilege. In this chapter you'll learn about alternative ways of organizing permissions in the identity-based access control model. In chapter 9, we'll look at alternative non-identity-based access control models.

DEFINITION Identity-based access control (IBAC) determines what you can do based on who you are. The user performing an API request is first authenticated and then a check is performed to see if that user is authorized to perform the requested action.

8.1 Users and groups

One of the most common approaches to simplifying permission management is to collect related users into groups, as shown in figure 8.1. Rather than the subject of an access control decision always being an individual user, groups allow permissions to be assigned to collections of users.

There is a many-to-many relationship between users and groups: a group can have many members, and a user can belong to many groups. If the membership of a group is defined in terms of subjects (which may be either users or other groups), then it is also possible to have groups be members of other groups, creating a hierarchical structure. For example, you might define a group for employees and another one for customers. If you then add a new group for project managers, you could add this group to the employees' group: all project managers are employees.

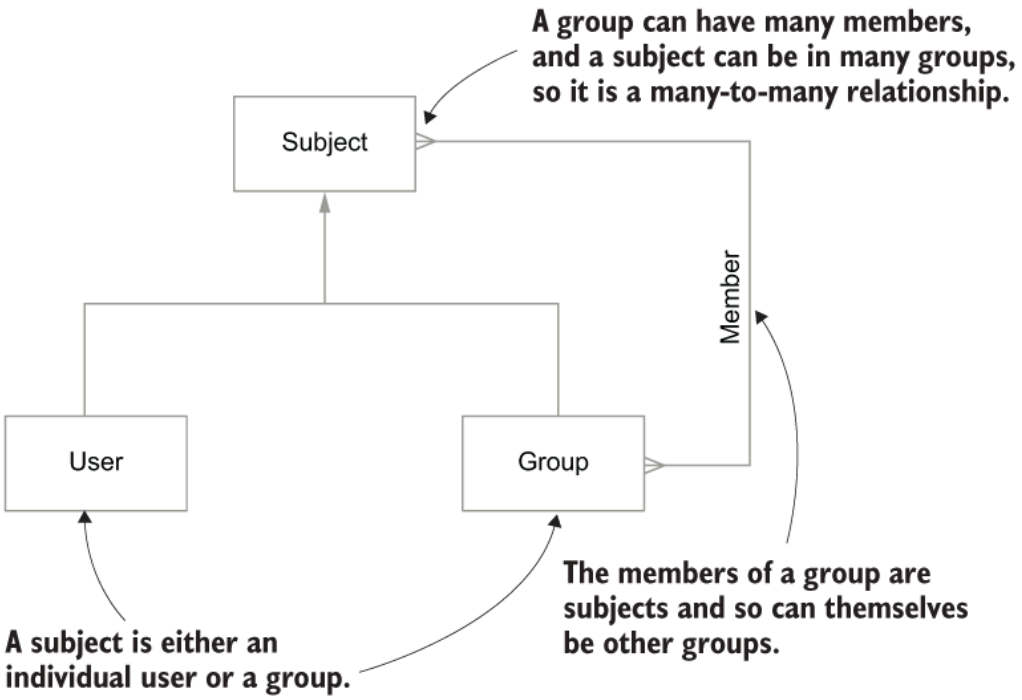

Figure 8.1 Groups are added as a new type of subject. Permissions can then be assigned to individual users or to groups. A user can be a member of many groups and each group can have many members.

The advantage of groups is that you can now assign permissions to groups and be sure that all members of that group have consistent permissions. When a new software engineer joins your organization, you can simply add them to the "software engineers" group rather than having to remember all the individual permissions that they need to get their job done. And when they change jobs, you simply remove them from that group and add them to a new one.

UNIX groups

Another advantage of groups is that they can be used to compress the permissions associated with an object in some cases. For example, the UNIX file system stores permissions for each file as a simple triple of permissions for the current user, the user's group, and anyone else. Rather than storing permissions for many individual users, the owner of the file

can assign permissions to only a single pre-existing group, dramatically reducing the amount of data that must be stored for each file. The downside of this compression is that if a group doesn't exist with the required members, then the owner may have to grant access to a larger group than they would otherwise like to.

The implementation of simple groups is straightforward. Currently in the Natter API you have written, there is a `users` table and a `permissions` table that acts as an ACL linking users to permissions within a space. To add groups, you could first add a new table to indicate which users are members of which groups:

```
CREATE TABLE group_members(
    group_id VARCHAR(30) NOT NULL,
    user_id VARCHAR(30) NOT NULL REFERENCES users(user_id));
CREATE INDEX group_member_user_idx ON group_members(user_id);
```

When the user authenticates, you can then look up the groups that user is a member of and add them as an additional request attribute that can be viewed by other processes. Listing 8.1 shows how groups could be looked up in the `authenticate()` method in `UserController` after the user has successfully authenticated.

Listing 8.1 Looking up groups during authentication

```
    if (hash.isPresent() && SCryptUtil.check(password, hash.get())) {
        request.attribute("subject", username);

        var groups = database.findAll(String.class,                  (1)
            "SELECT DISTINCT group_id FROM group_members " +         (1)
                    "WHERE user_id = ?", username);                  (1)
        request.attribute("groups", groups);                         (2)
    }
```

(1) Look up all groups that the user belongs to.

(2) Set the user's groups as a new attribute on the request.

You can then either change the `permissions` table to allow either a user or group ID to be used (dropping the foreign key constraint to the users table):

```
CREATE TABLE permissions(
    space_id INT NOT NULL REFERENCES spaces(space_id),
```

```
    user_or_group_id VARCHAR(30) NOT NULL,                ❶
    perms VARCHAR(3) NOT NULL);
```

❶ Allow either a user or group ID.

or you can create two separate permission tables and define a view that performs a union of the two:

```
CREATE TABLE user_permissions(...);
CREATE TABLE group_permissions(...);
CREATE VIEW permissions(space_id, user_or_group_id, perms) AS
    SELECT space_id, user_id, perms FROM user_permissions
    UNION ALL
    SELECT space_id, group_id, perms FROM group permissions;
```

To determine if a user has appropriate permissions, you would query first for individual user permissions and then for permissions associated with any groups the user is a member of. This can be accomplished in a single query, as shown in listing 8.2, which adjusts the `requirePermission` method in `UserController` to take groups into account by building a dynamic SQL query that checks the permissions table for both the username from the subject attribute of the request and any groups the user is a member of. Dalesbred has support for safely constructing dynamic queries in its `QueryBuilder` class, so you can use that here for simplicity.

TIP When building dynamic SQL queries, be sure to use only placeholders and never include user input directly in the query being built to avoid SQL injection attacks, which are discussed in chapter 2. Some databases support temporary tables, which allow you to insert dynamic values into the temporary table and then perform a SQL `JOIN` against the temporary table in your query. Each transaction sees its own copy of the temporary table, avoiding the need to generate dynamic queries.

Listing 8.2 Taking groups into account when looking up permissions

```
public Filter requirePermission(String method, String permission) {
    return (request, response) -> {
        if (!method.equals(request.requestMethod())) {
            return;
        }

        requireAuthentication(request, response);

        var spaceId = Long.parseLong(request.params(":spaceId"));
```

```
        var username = (String) request.attribute("subject");
        List<String> groups = request.attribute("groups");                  (1)

        var queryBuilder = new QueryBuilder(                                (2)
                "SELECT perms FROM permissions " +                          (2)
                    "WHERE space_id = ? " +                                 (2)
                    "AND (user_or_group_id = ?", spaceId, username);        (2)
        for (var group : groups) {                                          (3)
            queryBuilder.append(" OR user_or_group_id = ?", group);         (3)
        }                                                                   (3)
        queryBuilder.append(")");                                           (3)

        var perms = database.findAll(String.class,
                queryBuilder.build());
        if (perms.stream().noneMatch(p -> p.contains(permission))) {        (4)
            halt(403);                                                      (4)
        }
    };
}
```

(1) Look up the groups the user is a member of.

(2) Build a dynamic query to check permissions for the user.

(3) Include any groups in the query.

(4) Fail if none of the permissions for the user or groups allow this action.

You may be wondering why you would split out looking up the user's groups during authentication to then just use them in a second query against the `permissions` table during access control. It would be more efficient instead to perform a single query that automatically checked the groups for a user using a `JOIN` or sub-query against the group membership table, such as the following:

```
SELECT perms FROM permissions
 WHERE space_id = ?
   AND (user_or_group_id = ?      (1)
    OR user_or_group_id IN        (2)
    (SELECT DISTINCT group_id     (2)
       FROM group_members         (2)
      WHERE user_id = ?))         (2)
```

(1) Check for permissions for this user directly.

❷ Check for permissions for any groups the user is a member of.

Although this query is more efficient, it is unlikely that the extra query of the original design will become a significant performance bottleneck. But combining the queries into one has a significant drawback in that it violates the layering of authentication and access control. As far as possible, you should ensure that all user attributes required for access control decisions are collected during the authentication step, and then decide if the request is authorized using these attributes. As a concrete example of how violating this layering can cause problems, consider what would happen if you changed your API to use an external user store such as LDAP (discussed in the next section) or an OpenID Connect identity provider (chapter 7). In these cases, the groups that a user is a member of are likely to be returned as additional attributes during authentication (such as in the ID token JWT) rather than exist in the API's own database.

8.1.1 LDAP groups

In many large organizations, including most companies, users are managed centrally in an LDAP (Lightweight Directory Access Protocol) directory. LDAP is designed for storing user information and has built-in support for groups. You can learn more about LDAP at **https://ldap.com/basic-ldap-concepts/**. The LDAP standard defines the following two forms of groups:

1. Static groups are defined using the `groupOfNames` or `groupOfUniqueNames` object classes,**1** which explicitly list the members of the group using the `member` or `uniqueMember` attributes. The difference between the two is that `groupOfUniqueNames` forbids the same member being listed twice.
2. Dynamic groups are defined using the `groupOfURLs` object class, where the membership of the group is given by a collection of LDAP URLs that define search queries against the directory. Any entry that matches one of the search URLs is a member of the group.

Some directory servers also support virtual static groups, which look like static groups but query a dynamic group to determine the membership. Dynamic groups can be useful when groups become very large, because they avoid having to explicitly list every member of the group, but they can cause performance problems as the server needs to perform potentially expensive search operations to determine the members of a group.

To find which static groups a user is a member of in LDAP, you must perform a search against the directory for all groups that have that user's distinguished name as a value of their `member` attribute, as shown in listing 8.3. First, you need to connect to the LDAP server using the Java Naming and Directory Interface (JNDI) or another LDAP client library. Normal LDAP users typically are not permitted to run searches, so you should use a separate JNDI `InitialDirContext` for looking up a user's groups, configured to use a connection user that has appropriate permissions. To find the groups that a user is in, you can use the following search filter, which finds all LDAP `groupOfNames` entries that contain the given user as a member:

```
(&(objectClass=groupOfNames)(member=uid=test,dc=example,dc=org))
```

To avoid LDAP injection vulnerabilities (explained in chapter 2), you can use the facilities in JNDI to let search filters have parameters. JNDI will then make sure that any user input in these parameters is properly escaped before passing it to the LDAP directory. To use this, replace the user input in the field with a numbered parameter (starting at 0) in the form `{0}` or `{1}` or `{2}`, and so on, and then supply an `Object` array with the actual arguments to the `search` method. The names of the groups can then be found by looking up the CN (Common Name) attribute on the results.

Listing 8.3 Looking up LDAP groups for a user

```
import javax.naming.*;
import javax.naming.directory.*;
import java.util.*;

private List<String> lookupGroups(String username)
        throws NamingException {
    var props = new Properties();
    props.put(Context.INITIAL_CONTEXT_FACTORY,                  (1)
            "com.sun.jndi.ldap.LdapCtxFactory");                (1)
    props.put(Context.PROVIDER_URL, ldapUrl);                   (1)
    props.put(Context.SECURITY_AUTHENTICATION, "simple");       (1)
    props.put(Context.SECURITY_PRINCIPAL, connUser);            (1)
    props.put(Context.SECURITY_CREDENTIALS, connPassword);      (1)

    var directory = new InitialDirContext(props);               (1)

    var searchControls = new SearchControls();
    searchControls.setSearchScope(
```

```
                SearchControls.SUBTREE_SCOPE);
        searchControls.setReturningAttributes(
                new String[]{"cn"});

        var groups = new ArrayList<String>();
        var results = directory.search(
            "ou=groups,dc=example,dc=com",
            "(&(objectClass=groupOfNames)" +                    (2)
            "(member=uid={0},ou=people,dc=example,dc=com))",    (2)
            new Object[]{ username },                           (3)
            searchControls);

        while (results.hasMore()) {
            var result = results.next();
            groups.add((String) result.getAttributes()          (4)
                .get("cn").get(0));                             (4)
        }

        directory.close();

        return groups;
    }
```

(1) Set up the connection details for the LDAP server.

(2) Search for all groups with the user as a member.

(3) Use query parameters to avoid LDAP injection vulnerabilities.

(4) Extract the CN attribute of each group the user is a member of.

To make looking up the groups a user belongs to more efficient, many directory servers support a virtual attribute on the user entry itself that lists the groups that user is a member of. The directory server automatically updates this attribute as the user is added to and removed from groups (both static and dynamic). Because this attribute is nonstandard, it can have different names but is often called `isMemberOf` or something similar. Check the documentation for your LDAP server to see if it provides such an attribute. Typically, it is much more efficient to read this attribute than to search for the groups that a user is a member of.

TIP If you need to search for groups regularly, it can be worthwhile to cache the results for a short period to prevent excessive searches on the directory.

Pop quiz

1. True or False: In general, can groups contain other groups as members?
2. Which three of the following are common types of LDAP groups?
    1. Static groups
    2. Abelian groups
    3. Dynamic groups
    4. Virtual static groups
    5. Dynamic static groups
    6. Virtual dynamic groups
3. Given the following LDAP filter:

    ```
    (&(objectClass=#A)(member=uid=alice,dc=example,dc=com))
    ```

    which one of the following object classes would be inserted into the position marked #A to search for static groups Alice belongs to?
    1. `group`
    2. `herdOfCats`
    3. `groupOfURLs`
    4. `groupOfNames`
    5. `gameOfThrones`
    6. `murderOfCrows`
    7. `groupOfSubjects`

The answers are at the end of the chapter.

8.2 Role-based access control

Although groups can make managing large numbers of users simpler, they do not fully solve the difficulties of managing permissions for a complex API. First, almost all implementations of groups still allow permissions to be assigned to individual users as well as to groups. This means that to work out who has access to what, you still often need to examine the permissions for all users as well as the groups they belong to. Second, because groups are often used to organize users for a whole organization (such as in a central LDAP directory), they sometimes cannot be very useful distinctions for your API. For example, the LDAP directory might just have a group for all software engineers, but your API needs to distinguish between backend and frontend engineers, QA, and scrum masters. If you cannot change the centrally managed groups, then you are back to managing permissions for individual users. Finally, even when groups are a good fit for an API, there may be large numbers of fine-grained permissions assigned to each group, making it difficult to review the permissions.

To address these drawbacks, role-based access control (RBAC) introduces the notion of role as an intermediary between users and permissions, as shown in figure 8.2. Permissions are no longer directly assigned to users (or to groups). Instead, permissions are assigned to roles, and then roles are assigned to users. This can dramatically simplify the management of permissions, because it is much simpler to assign somebody the "moderator" role than to remember exactly which permissions a moderator is supposed to have. If the permissions change over time, then you can simply change the permissions associated with a role without needing to update the permissions for many users and groups individually.

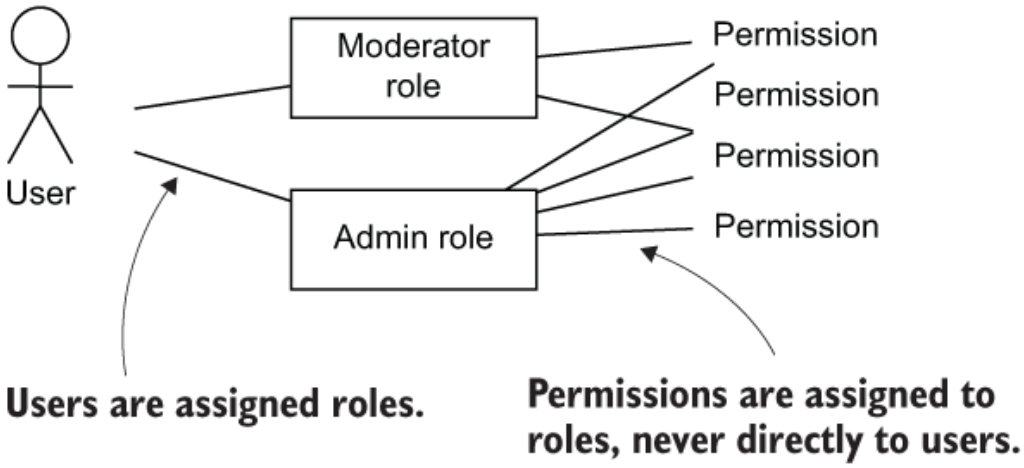

Figure 8.2 In RBAC, permissions are assigned to roles rather than directly to users. Users are then assigned to roles, depending on their required level of access.

In principle, everything that you can accomplish with RBAC could be accomplished with groups, but in practice there are several differences in how they are used, including the following:

- Groups are used primarily to organize users, while roles are mainly used as a way to organize permissions.
- As discussed in the previous section, groups tend to be assigned centrally, whereas roles tend to be specific to a particular application or API. As an example, every API may have an `admin` role, but the set of users that are administrators may differ from API to API.
- Group-based systems often allow permissions to be assigned to individual users, but RBAC systems typically don't allow that. This restriction can dramatically simplify the process of reviewing who has access to what.

- RBAC systems split the definition and assigning of permissions to roles from the assignment of users to those roles. It is much less error-prone to assign a user to a role than to work out which permissions each role should have, so this is a useful separation of duties that improves security.
- Roles may have a dynamic element. For example, some military and other environments have the concept of a duty officer, who has particular privileges and responsibilities only during their shift. When the shift ends, they hand over to the next duty officer, who takes on that role.

RBAC is almost always used as a form of mandatory access control, with roles being described and assigned by whoever controls the systems that are being accessed. It is much less common to allow users to assign roles to other users the way they can with permissions in discretionary access control approaches. Instead, it is common to layer a DAC mechanism such as OAuth2 (chapter 7) over an underlying RBAC system so that a user with a moderator role, for example, can delegate some part of their permissions to a third party. Some RBAC systems give users some discretion over which roles they use when performing API operations. For example, the same user may be able to send messages to a chatroom as themselves or using their role as Chief Financial Officer when they want to post an official statement. The NIST (National Institute of Standards and Technology) standard RBAC model (**http://mng.bz/v9eJ**) includes a notion of session, in which a user can choose which of their roles are active at a given time when making API requests. This works similarly to scoped tokens in OAuth, allowing a session to activate only a subset of a user's roles, reducing the damage if the session is compromised. In this way, RBAC also better supports the principle of least privilege than groups because a user can act with only a subset of their full authority.

8.2.1 Mapping roles to permissions

There are two basic approaches to mapping roles to lower-level permissions inside your API. The first is to do away with permissions altogether and instead to just annotate each operation in your API with the role or roles that can call that operation. In this case, you'd replace the existing `requirePermission` filter with a new `requireRole` filter that enforced role requirements instead. This is the approach taken in Java Enterprise Edition (Java EE) and the JAX-RS framework, where methods can be annotated with the `@RolesAllowed` annotation to describe which roles can call that method via an API, as shown in listing 8.4.

Listing 8.4 Annotating methods with roles in Java EE

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import javax.annotation.security.*;                    ❶

@DeclareRoles({"owner", "moderator", "member"})        ❷
@Path("/spaces/{spaceId}/members")
public class SpaceMembersResource {

    @POST
    @RolesAllowed("owner")                             ❸
    public Response addMember() { .. }

    @GET
    @RolesAllowed({"owner", "moderator"})              ❸
    public Response listMembers() { .. }
}
```

❶ Role annotations are in the javax.annotation.security package.

❷ Declare roles with the @DeclareRoles annotation.

❸ Describe role restrictions with the @RolesAllowed annotation.

The second approach is to retain an explicit notion of lower-level permissions, like those currently used in the Natter API, and to define an explicit mapping from roles to permissions. This can be useful if you want to allow administrators or other users to define new roles from scratch, and it also makes it easier to see exactly what permissions a role has been granted without having to examine the source code of the API. Listing 8.5 shows the SQL needed to define four new roles based on the existing Natter API permissions:

- The social space owner has full permissions.
- A moderator can read posts and delete offensive posts.
- A normal member can read and write posts, but not delete any.
- An observer is only allowed to read posts and not write their own.

Open src/main/resources/schema.sql in your editor and add the lines from listing 8.5 to the end of the file and click save. You can also delete the existing `permissions` table (and associated `GRANT` statements) if you wish.

Listing 8.5 Role permissions for the Natter API

```
CREATE TABLE role_permissions(                          (1)
    role_id VARCHAR(30) NOT NULL PRIMARY KEY,           (1)
    perms VARCHAR(3) NOT NULL                           (1)
);
INSERT INTO role_permissions(role_id, perms)
    VALUES ('owner', 'rwd'),                            (2)
           ('moderator', 'rd'),                         (2)
           ('member', 'rw'),                            (2)
           ('observer', 'r');                           (2)
GRANT SELECT ON role_permissions TO natter_api_user;    (3)
```

(1) Each role grants a set of permissions.

(2) Define roles for Natter social spaces.

(3) Because the roles are fixed, the API is granted read-only access.

8.2.2 Static roles

Now that you've defined how roles map to permissions, you just need to decide how to map users to roles. The most common approach is to statically define which users (or groups) are assigned to which roles. This is the approach taken by most Java EE application servers, which define configuration files to list the users and groups that should be assigned different roles. You can implement the same kind of approach in the Natter API by adding a new table to map users to roles within a social space. Roles in the Natter API are scoped to each social space so that the owner of one social space cannot make changes to another.

DEFINITION When users, groups, or roles are confined to a subset of your application, this is known as a security domain or realm.

Listing 8.6 shows the SQL to create a new table to map a user in a social space to a role. Open schema.sql again and add the new table definition to the file. The `user_roles` table, together with the `role_permissions` table, take the place of the old `permissions` table. In the Natter API, you'll restrict a user to having just one role within a space, so you can add a primary key constraint on the `space_id` and `user_id` fields. If you wanted to allow more than one role you could leave this out and manually add an index on those fields instead. Don't forget to grant permissions to the Natter API database user.

Listing 8.6 Mapping static roles

```
CREATE TABLE user_roles(                                                    (1)
    space_id INT NOT NULL REFERENCES spaces(space_id),                      (1)
    user_id VARCHAR(30) NOT NULL REFERENCES users(user_id),                 (1)
    role_id VARCHAR(30) NOT NULL REFERENCES role_permissions(role_id),      (1)
    PRIMARY KEY (space_id, user_id)                                         (2)
);
GRANT SELECT, INSERT, DELETE ON user_roles TO natter_api_user;              (3)
```

(1) Natter restricts each user to have only one role.

(2) Map users to roles within a space.

(3) Grant permissions to the Natter database user.

To grant roles to users, you need to update the two places where permissions are currently granted inside the `SpaceController` class:

- In the `createSpace` method, the owner of the new space is granted full permissions. This should be updated to instead grant the `owner` role.
- In the `addMember` method, the request contains the permissions for the new member. This should be changed to accept a role for the new member instead.

The first task is accomplished by opening the SpaceController.java file and finding the line inside the `createSpace` method where the insert into the permissions table statement is. Remove those lines and replace them instead with the following to insert a new role assignment:

```
    database.updateUnique(
        "INSERT INTO user_roles(space_id, user_id, role_id) " +
                "VALUES(?, ?, ?)", spaceId, owner, "owner");
```

Updating `addMember` involves a little more code, because you should ensure that you validate the new role. Add the following line to the top of the class to define the valid roles:

```
    private static final Set<String> DEFINED_ROLES =
            Set.of("owner", "moderator", "member", "observer");
```

You can now update the implementation of the `addMember` method to be role-based instead of permission-based, as shown in listing 8.7. First, extract the desired role from the request and ensure it is a valid role name. You can default to the `member` role if none is specified as this is the normal role for most members. It is then simply a case of inserting the role into the `user_roles` table instead of the old `permissions` table and returning the assigned role in the response.

Listing 8.7 Adding new members with roles

```
public JSONObject addMember(Request request, Response response) {
  var json = new JSONObject(request.body());
  var spaceId = Long.parseLong(request.params(":spaceId"));
  var userToAdd = json.getString("username");
  var role = json.optString("role", "member");                    (1)

  if (!DEFINED_ROLES.contains(role)) {                            (1)
    throw new IllegalArgumentException("invalid role");           (1)
  }

  database.updateUnique(
          "INSERT INTO user_roles(space_id, user_id, role_id)" +  (2)
                  " VALUES(?, ?, ?)", spaceId, userToAdd, role);  (2)

  response.status(200);
  return new JSONObject()
          .put("username", userToAdd)
          .put("role", role);                                     (3)
}
```

(1) Extract the role from the input and validate it.

(2) Insert the new role assignment for this space.

(3) Return the role in the response.

8.2.3 Determining user roles

The final step of the puzzle is to determine which roles a user has when they make a request to the API and the permissions that each role allows. This can be found by looking up the user in the `user_roles` table to discover their role for a given space, and then looking up the permissions assigned to that role in the `role_permissions` table. In contrast to the situation with groups in section 8.1, roles are usually specific to an API, so it is less likely that you would be told a user's roles as part of authentica-

tion. For this reason, you can combine the lookup of roles and the mapping of roles into permissions into a single database query, joining the two tables together, as follows:

```
SELECT rp.perms
  FROM role_permissions rp
  JOIN user_roles ur
    ON ur.role_id = rp.role_id
 WHERE ur.space_id = ? AND ur.user_id = ?
```

Searching the database for roles and permissions can be expensive, but the current implementation will repeat this work every time the `requirePermission` filter is called, which could be several times while processing a request. To avoid this issue and simplify the logic, you can extract the permission look up into a separate filter that runs before any permission checks and stores the permissions in a request attribute. Listing 8.8 shows the new `lookupPermissions` filter that performs the mapping from user to role to permissions, and then updated `requirePermission` method. By reusing the existing permissions checks, you can add RBAC on top without having to change the access control rules. Open UserController.java in your editor and update the `requirePermission` method to match the listing.

Listing 8.8 Determining permissions based on roles

```
public void lookupPermissions(Request request, Response response) {
    requireAuthentication(request, response);
    var spaceId = Long.parseLong(request.params(":spaceId"));
    var username = (String) request.attribute("subject");

    var perms = database.findOptional(String.class,                    (1)
            "SELECT rp.perms " +                                       (1)
            "  FROM role_permissions rp JOIN user_roles ur" +          (1)
            "    ON rp.role_id = ur.role_id" +                         (1)
            " WHERE ur.space_id = ? AND ur.user_id = ?",               (1)
            spaceId, username).orElse("");                             (1)
    request.attribute("perms", perms);                                 (2)
}

public Filter requirePermission(String method, String permission) {
    return (request, response) -> {
        if (!method.equals(request.requestMethod())) {
            return;
        }
```

```
            var perms = request.<String>attribute("perms");
            if (!perms.contains(permission)) {
                halt(403);
            }
        };
    }
```

❶ Determine user permissions by mapping user to role to permissions.

❷ Store permissions in a request attribute.

❸ Retrieve permissions from the request before checking.

You now need to add calls to the new filter to ensure permissions are looked up. Open the Main.java file and add the following lines to the `main` method, before the definition of the `postMessage` operation:

```
before("/spaces/:spaceId/messages",
    userController::lookupPermissions);
before("/spaces/:spaceId/messages/*",
    userController::lookupPermissions);
before("/spaces/:spaceId/members",
    userController::lookupPermissions);
```

If you restart the API server you can now add users, create spaces, and add members using the new RBAC approach. All the existing permission checks on API operations are still enforced, only now they are managed using roles instead of explicit permission assignments.

8.2.4 Dynamic roles

Though static role assignments are the most common, some RBAC systems allow more dynamic queries to determine which roles a user should have. For example, a call center worker might be granted a role that allows them access to customer records so that they can respond to customer support queries. To reduce the risk of misuse, the system could be configured to grant the worker this role only during their contracted working hours, perhaps based on their shift times. Outside of these times the user would not be granted the role, and so would be denied access to customer records if they tried to access them.

Although dynamic role assignments have been implemented in several systems, there is no clear standard for how to build dynamic roles.

Approaches are usually based on database queries or perhaps based on rules specified in a logical form such as Prolog or the Web Ontology Language (OWL). When more flexible access control rules are required, attribute-based access control (ABAC) has largely replaced RBAC, as discussed in section 8.3. NIST has attempted to integrate ABAC with RBAC to gain the best of both worlds (**http://mng.bz/4BMa**), but this approach is not widely adopted.

Other RBAC systems implement constraints, such as making two roles mutually exclusive; a user can't have both roles at the same time. This can be useful for enforcing separation of duties, such as preventing a system administrator from also managing audit logs for a sensitive system.

Pop quiz

4. Which of the following are more likely to apply to roles than to groups?
    1. Roles are usually bigger than groups.
    2. Roles are usually smaller than groups.
    3. All permissions are assigned using roles.
    4. Roles better support separation of duties.
    5. Roles are more likely to be application specific.
    6. Roles allow permissions to be assigned to individual users.
5. What is a session used for in the NIST RBAC model? Pick one answer.
    1. To allow users to share roles.
    2. To allow a user to leave their computer unlocked.
    3. To allow a user to activate only a subset of their roles.
    4. To remember the users name and other identity attributes.
    5. To allow a user to keep track of how long they have worked.
6. Given the following method definition

```
@<annotation here>
public Response adminOnlyMethod(String arg);
```

    what annotation value can be used in the Java EE and JAX-RS role system to restrict the method to only be called by users with the ADMIN role?
    1. `@DenyAll`
    2. `@PermitAll`
    3. `@RunAs("ADMIN")`
    4. `@RolesAllowed("ADMIN")`
    5. `@DeclareRoles("ADMIN")`

The answers are at the end of the chapter.

8.3 Attribute-based access control

Although RBAC is a very successful access control model that has been widely deployed, in many cases the desired access control policies cannot be expressed through simple role assignments. Consider the call center agent example from section 8.2.4. As well as preventing the agent from accessing customer records outside of their contracted working hours, you might also want to prevent them accessing those records if they are not actually on a call with that customer. Allowing each agent to access all customer records during their working hours is still more authority than they really need to get their job done, violating the principle of least privilege. It may be that you can determine which customer the call agent is talking to from their phone number (caller ID), or perhaps the customer enters an account number using the keypad before they are connected to an agent. You'd like to only allow the agent access to just that customer's file for the duration of the call, perhaps allowing five minutes afterward for them to finishing writing any notes.

To handle these kinds of dynamic access control decisions, an alternative to RBAC has been developed known as ABAC: attribute-based access control. In ABAC, access control decisions are made dynamically for each API request using collections of attributes grouped into four categories:

- Attributes about the subject; that is, the user making the request. This could include their username, any groups they belong to, how they were authenticated, when they last authenticated, and so on.
- Attributes about the resource or object being accessed, such as the URI of the resource or a security label (TOP SECRET, for example).
- Attributes about the action the user is trying to perform, such as the HTTP method.
- Attributes about the environment or context in which the operation is taking place. This might include the local time of day, or the location of the user performing the action.

The output of ABAC is then an allow or deny decision, as shown in figure 8.3.

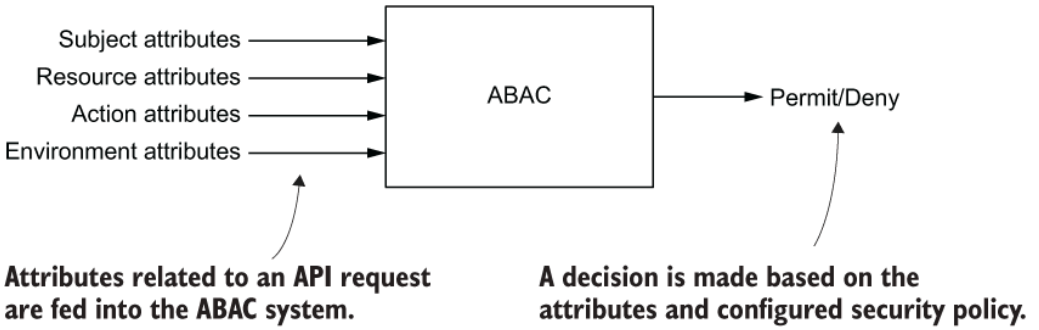

Figure 8.3 In an ABAC system, access control decisions are made dynamically based on attributes describing the subject, resource, action, and environment or context of the API request.

Listing 8.9 shows example code for gathering attribute values to feed into an ABAC decision process in the Natter API. The code implements a Spark filter that can be included before any API route definition in place of the existing `requirePermission` filters. The actual implementation of the ABAC permission check is left abstract for now; you will develop implementations in the next sections. The code collects attributes into the four attribute categories described above by examining the Spark `Request` object and extracting the username and any groups populated during authentication. You can include other attributes, such as the current time, in the environment properties. Extracting these kind of environmental attributes makes it easier to test the access control rules because you can easily pass in different times of day in your tests. If you're using JWTs (chapter 6), then you might want to include claims from the JWT Claims Set in the subject attributes, such as the issuer or the issued-at time. Rather than using a simple `boolean` value to indicate the decision, you should use a custom `Decision` class. This is used to combine decisions from different policy rules, as you'll see in section 8.3.1.

Listing 8.9 Gathering attribute values

```
package com.manning.apisecurityinaction.controller;

import java.time.LocalTime;
import java.util.Map;
import spark.*;
import static spark.Spark.halt;
public abstract class ABACAccessController {
    public void enforcePolicy(Request request, Response response) {

        var subjectAttrs = new HashMap<String, Object>();              (1)
        subjectAttrs.put("user", request.attribute("subject"));        (1)
        subjectAttrs.put("groups", request.attribute("groups"));       (1)
```

```
            var resourceAttrs = new HashMap<String, Object>();                  (1)
            resourceAttrs.put("path", request.pathInfo());                      (1)
            resourceAttrs.put("space", request.params(":spaceId"));             (1)

            var actionAttrs = new HashMap<String, Object>();                    (1)
            actionAttrs.put("method", request.requestMethod());                 (1)

            var envAttrs = new HashMap<String, Object>();                       (1)
            envAttrs.put("timeOfDay", LocalTime.now());                         (1)
            envAttrs.put("ip", request.ip());                                   (1)

            var decision = checkPermitted(subjectAttrs, resourceAttrs,          (2)
                    actionAttrs, envAttrs);                                     (2)

            if (!decision.isPermitted()) {                                      (3)
                halt(403);                                                      (3)
            }
        }

        abstract Decision checkPermitted(
                Map<String, Object> subject,
                Map<String, Object> resource,
                Map<String, Object> action,
                Map<String, Object> env);

        public static class Decision {                                          (4)
        }
    }
```

(1) Gather relevant attributes and group them into categories.

(2) Check whether the request is permitted.

(3) If not, halt with a 403 Forbidden error.

(4) The Decision class will be described next.

8.3.1 Combining decisions

When implementing ABAC, typically access control decisions are structured as a set of independent rules describing whether a request should be permitted or denied. If more than one rule matches a request, and they have different outcomes, then the question is which one should be preferred. This boils down to the two following questions:

- What should the default decision be if no access control rules match the request?
- How should conflicting decisions be resolved?

The safest option is to default to denying requests unless explicitly permitted by some access rule, and to give deny decisions priority over permit decisions. This requires at least one rule to match and decide to permit the action and no rules to decide to deny the action for the request to be allowed. When adding ABAC on top of an existing access control system to enforce additional constraints that cannot be expressed in the existing system, it can be simpler to instead opt for a default permit strategy where requests are permitted to proceed if no ABAC rules match at all. This is the approach you'll take with the Natter API, adding additional ABAC rules that deny some requests and let all others through. In this case, the other requests may still be rejected by the existing RBAC permissions enforced earlier in the chapter.

The logic for implementing this default permit with deny overrides strategy is shown in the Decision class in listing 8.10. The `permit` variable is initially set to `true` but any call to the `deny()` method will set it to false. Calls to the `permit()` method are ignored because this is the default unless another rule has called `deny()` already, in which case the deny should take precedence. Open ABACAccessController.java in your editor and add the `Decision` class as an inner class.

Listing 8.10 Implementing decision combining

```
public static class Decision {
    private boolean permit = true;     ❶
    public void deny() {               ❷
        permit = false;                ❷
    }
    public void permit() {             ❸
    }

    boolean isPermitted() {
        return permit;
    }
}
```

❶ Default to permit

❷ An explicit deny decision overrides the default.

❸ Explicit permit decisions are ignored.

8.3.2 Implementing ABAC decisions

Although you could implement ABAC access control decisions directly in Java or another programming language, it's often clearer if the policy is expressed in the form of rules or domain-specific language (DSL) explicitly designed to express access control decisions. In this section you'll implement a simple ABAC decision engine using the Drools (**https://drools.org**) business rules engine from Red Hat. Drools can be used to write all kinds of business rules and provides a convenient syntax for authoring access control rules.

TIP Drools is part of a larger suite of tools marketed under the banner "Knowledge is Everything," so many classes and packages used in Drools include the `kie` abbreviation in their names.

To add the Drools rule engine to the Natter API project, open the pom.xml file in your editor and add the following dependencies to the `<dependencies>` section:

```
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-api</artifactId>
  <version>7.26.0.Final</version>
</dependency>
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-core</artifactId>
  <version>7.26.0.Final</version>
</dependency>
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-compiler</artifactId>
  <version>7.26.0.Final</version>
</dependency>
```

When it starts up, Drools will look for a file called kmodule.xml on the classpath that defines the configuration. You can use the default configuration, so navigate to the folder src/main/resources and create a new folder named META-INF under resources. Then create a new file called kmodule.xml inside the src/main/resource/META-INF folder with the following contents:

```
<?xml version="1.0" encoding="UTF-8" ?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>
```

You can now implement a version of the `ABACAccessController` class that evaluates decisions using Drools. Listing 8.11 shows code that implements the `checkPermitted` method by loading rules from the classpath using `KieServices.get().getKieClasspathContainer()`.

To query the rules for a decision, you should first create a new KIE session and set an instance of the Decision class from the previous section as a global variable that the rules can access. Each rule can then call the `deny()` or `permit()` methods on this object to indicate whether the request should be allowed. The attributes can then be added to the working memory for Drools using the `insert()` method on the session. Because Drools prefers strongly typed values, you can wrap each set of attributes in a simple wrapper class to distinguish them from each other (described shortly). Finally, call `session.fireAllRules()` to evaluate the rules against the attributes and then check the value of the decision variable to determine the final decision. Create a new file named DroolsAccessController.java inside the controller folder and add the contents of listing 8.11.

Listing 8.11 Evaluating decisions with Drools

```
package com.manning.apisecurityinaction.controller;
import java.util.*;
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;

public class DroolsAccessController extends ABACAccessController {

    private final KieContainer kieContainer;

    public DroolsAccessController() {
        this.kieContainer = KieServices.get().getKieClasspathContainer();
    }

    @Override
    boolean checkPermitted(Map<String, Object> subject,
                           Map<String, Object> resource,
                           Map<String, Object> action,
                           Map<String, Object> env) {

        var session = kieContainer.newKieSession();
```

```
            try {
                var decision = new Decision();
                session.setGlobal("decision", decision);

                session.insert(new Subject(subject));
                session.insert(new Resource(resource));
                session.insert(new Action(action));
                session.insert(new Environment(env));

                session.fireAllRules();
                return decision.isPermitted();
            } finally {
                session.dispose();
            }
        }
    }
```

❶ Load all rules found in the classpath.

❷ Start a new Drools session.

❸ Create a Decision object and set it as a global variable named "decision."

❹ Insert facts for each category of attributes.

❺ Run the rule engine to see which rules match the request and check the decision.

❻ Dispose of the session when finished.

As mentioned, Drools likes to work with strongly typed values, so you can wrap each collection of attributes in a distinct class to make it simpler to write rules that match each one, as shown in listing 8.12. Open DroolsAccessController.java in your editor again and add the four wrapper classes from the following listing as inner classes to the `DroolsAccessController` class.

Listing 8.12 Wrapping attributes in types

```
    public static class Subject extends HashMap<String, Object> {    ❶
        Subject(Map<String, Object> m) { super(m); }                  ❶
    }                                                                 ❶

    public static class Resource extends HashMap<String, Object> {   ❷
```

```
    Resource(Map<String, Object> m) { super(m); }
}

public static class Action extends HashMap<String, Object> {
    Action(Map<String, Object> m) { super(m); }
}

public static class Environment extends HashMap<String, Object> {
    Environment(Map<String, Object> m) { super(m); }
}
```

❶ Wrapper for subject-related attributes

❷ Wrapper for resource-related attributes

You can now start writing access control rules. Rather than reimplementing all the existing RBAC access control checks, you will just add an additional rule that prevents moderators from deleting messages outside of normal office hours. Create a new file accessrules.drl in the folder src/main/resources to contain the rules. Listing 8.13 lists the example rule. As for Java, a Drools rule file can contain a `package` and `import` statements, so use those to import the `Decision` and wrapper class you've just created. Next, you need to declare the global `decision` variable that will be used to communicate the decision by the rules. Finally, you can implement the rules themselves. Each rule has the following form:

```
rule "description"
    when
        conditions
    then
        actions
end
```

The description can be any useful string to describe the rule. The conditions of the rule match classes that have been inserted into the working memory and consist of the class name followed by a list of constraints inside parentheses. In this case, because the classes are maps, you can use the `this["key"]` syntax to match attributes inside the map. For this rule, you should check that the HTTP method is DELETE and that the hour field of the `timeOfDay` attribute is outside of the allowed 9-to-5 working hours. If the rule matches, the action of the rule will call the `deny()` method of the `decision` global variable. You can find more detailed information

about writing Drools rules on the **https://drools.org** website, or from the book Mastering JBoss Drools 6, by Mauricio Salatino, Mariano De Maio, and Esteban Aliverti (Packt, 2016).

Listing 8.13 An example ABAC rule

```
package com.manning.apisecurityinaction.rules;          (1)

import com.manning.apisecurityinaction.controller.
➥ DroolsAccessController.*;                            (1)
import com.manning.apisecurityinaction.controller.
➥ ABACAccessController.Decision;                       (1)

global Decision decision;                               (2)

rule "deny moderation outside office hours"             (3)
    when                                                (3)
        Action( this["method"] == "DELETE" )            (4)
        Environment( this["timeOfDay"].hour < 9         (4)
            || this["timeOfDay"].hour > 17 )            (4)
    then                                                (3)
        decision.deny();                                (5)
end
```

(1) Add package and import statements just like Java.

(2) Declare the decision global variable.

(3) A rule has a description, a when section with patterns, and a then section with actions.

(4) Patterns match the attributes.

(5) The action can call the permit or deny methods on the decision.

Now that you have written an ABAC rule you can wire up the main method to apply your rules as a Spark `before()` filter that runs before the other access control rules. The filter will call the `enforcePolicy` method inherited from the `ABACAccessController` (listing 8.9), which populates the attributes from the requests. The base class then calls the `checkDecision` method from listing 8.11, which will use Drools to evaluate the rules. Open Main.java in your editor and add the following lines to the `main()` method just before the route definitions in that file:

```
    var droolsController = new DroolsAccessController();
    before("/*", droolsController::enforcePolicy);
```

Restart the API server and make some sample requests to see if the policy is being enforced and is not interfering with the existing RBAC permission checks. To check that DELETE requests are being rejected outside of office hours, you can either adjust your computer's clock to a different time, or you can adjust the time of day environment attribute to artificially set the time of day to 11 p.m. Open ABACAccessController .java and change the definition of the `timeOfDay` attribute as follows:

```
envAttrs.put("timeOfDay", LocalTime.now().withHour(23));
```

If you then try to make any DELETE request to the API it'll be rejected:

```
$ curl -i -X DELETE \
  -u demo:password https://localhost:4567/spaces/1/messages/1
HTTP/1.1 403 Forbidden
...
```

TIP It doesn't matter if you haven't implemented any DELETE methods in the Natter API, because the ABAC rules will be applied before the request is matched to any endpoints (even if none exist). The Natter API implementation in the GitHub repository accompanying this book has implementations of several additional REST requests, including DELETE support, if you want to try it out.

8.3.3 Policy agents and API gateways

ABAC enforcement can be complex as policies increase in complexity. Although general-purpose rule engines such as Drools can simplify the process of writing ABAC rules, specialized components have been developed that implement sophisticated policy enforcement. These components are typically implemented either as a policy agent that plugs into an existing application server, web server, or reverse proxy, or else as standalone gateways that intercept requests at the HTTP layer, as illustrated in figure 8.4.

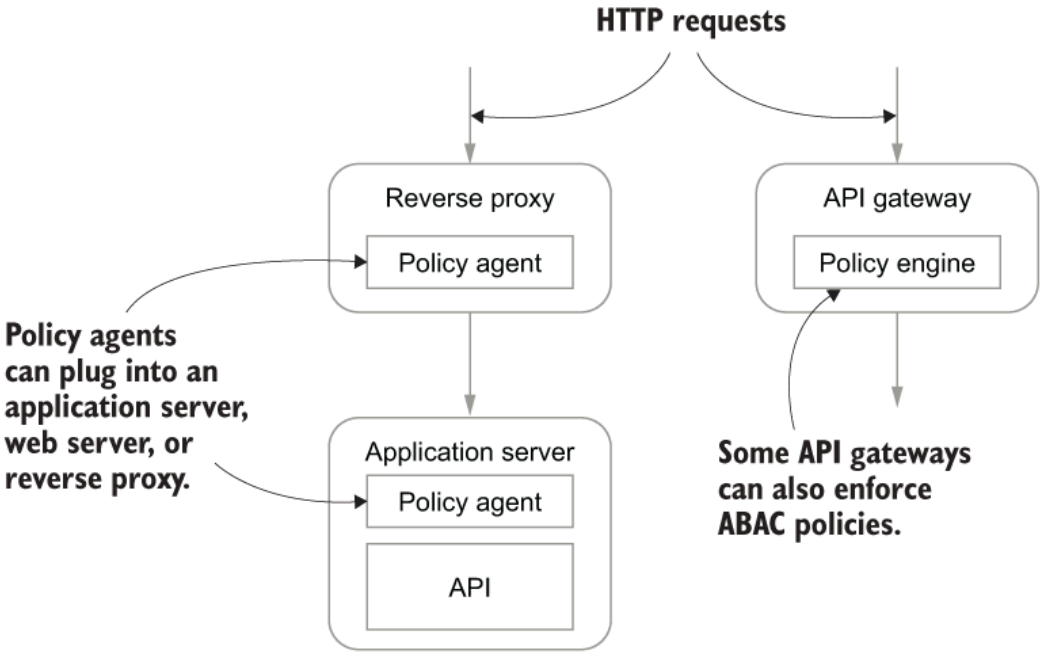

Figure 8.4 A policy agent can plug into an application server or reverse proxy to enforce ABAC policies. Some API gateways can also enforce policy decisions as standalone components.

For example, the Open Policy Agent (OPA, **https://www.openpolicyagent.org**) implements a policy engine using a DSL designed to make expressing access control decisions easy. It can be integrated into an existing infrastructure either using its REST API or as a Go library, and integrations have been written for various reverse proxies and gateways to add policy enforcement.

8.3.4 Distributed policy enforcement and XACML

Rather than combining all the logic of enforcing policies into the agent itself, another approach is to centralize the definition of policies in a separate server, which provides a REST API for policy agents to connect to and evaluate policy decisions. By centralizing policy decisions, a security team can more easily review and adjust policy rules for all APIs in an organization and ensure consistent rules are applied. This approach is most closely associated with XACML, the eXtensible Access-Control Markup Language (see **http://mng.bz/Qx2w**), which defines an XML-based language for policies with a rich set of functions for matching attributes and combining policy decisions. Although the XML format for defining policies has fallen somewhat out of favor in recent years, XACML also defined a reference architecture for ABAC systems that has been very influential and is now incorporated into NIST's recommendations for ABAC (**http://mng.bz/X0YG**).

DEFINITION XACML is the eXtensible Access-Control Markup Language, a standard produced by the OASIS standards body. XACML defines a rich XML-based policy language and a reference architecture for distributed policy enforcement.

The core components of the XACML reference architecture are shown in figure 8.5, and consist of the following functional components:

- A Policy Enforcement Point (PEP) acts like a policy agent to intercept requests to an API and reject any requests that are denied by policy.
- The PEP talks to a Policy Decision Point (PDP) to determine if a request should be allowed. The PDP contains a policy engine like those you've seen already in this chapter.
- A Policy Information Point (PIP) is responsible for retrieving and caching values of relevant attributes from different data sources. These might be local databases or remote services such as an OIDC UserInfo endpoint (see chapter 7).
- A Policy Administration Point (PAP) provides an interface for administrators to define and manage policies.

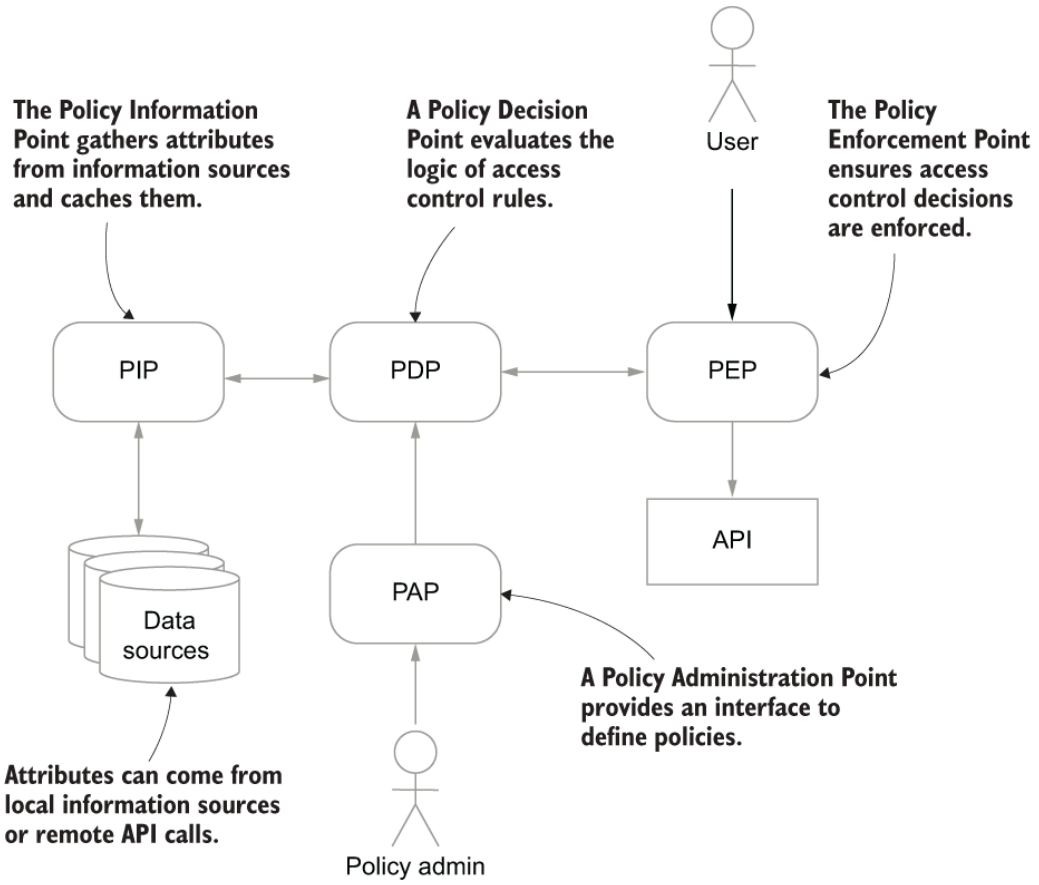

Figure 8.5 XACML defines four services that cooperate to implement an ABAC system. The Policy Enforcement Point (PEP) rejects requests that are denied by the Policy Decision Point (PDP). The Policy Information Point (PIP) retrieves attributes that are relevant to policy decisions. A

Policy Administration Point (PAP) can be used to define and manage policies.

The four components may be collocated or can be distributed on different machines. In particular, the XACML architecture allows policy definitions to be centralized within an organization, allowing easy administration and review. Multiple PEPs for different APIs can talk to the PDP via an API (typically a REST API), and XACML supports the concept of policy sets to allow policies for different PEPs to be grouped together with different combining rules. Many vendors offer implementations of the XACML reference architecture in some form, although often without the standard XML policy language, providing policy agents or gateways and PDP services that you can install into your environment to add ABAC access control decisions to existing services and APIs.

8.3.5 Best practices for ABAC

Although ABAC provides an extremely flexible basis for access control, its flexibility can also be a drawback. It's easy to develop overly complex rules, making it hard to determine exactly who has access to what. I have heard of deployments with many thousands of policy rules. Small changes to rules can have dramatic impacts, and it can be hard to predict how rules will combine. As an example, I once worked on a system that implemented ABAC rules in the form of XPath expressions that were applied to incoming XML messages; if a message matched any rule, it was rejected.

It turned out that a small change to the document structure made by another team caused many of the rules to no longer match, which allowed invalid requests to be processed for several weeks before somebody noticed. It would've been nice to be able to automatically tell when these XPath expressions could no longer match any messages, but due to the flexibility of XPath, this turns out to be impossible to determine automatically in general, and all our tests continued using the old format. This anecdote shows the potential downside of flexible policy evaluation engines, but they are still a very powerful way to structure access control logic.

To maximize the benefits of ABAC while limiting the potential for mistakes, consider adopting the following best practices:

- Layer ABAC over a simpler access control technology such as RBAC. This provides a defense-in-depth strategy so that a mistake in the ABAC rules doesn't result in a total loss of security.
- Implement automated testing of your API endpoints so that you are alerted quickly if a policy change results in access being granted to unintended parties.
- Ensure access control policies are maintained in a version control system so that they can be easily rolled back if necessary. Ensure proper review of all policy changes.
- Consider which aspects of policy should be centralized and which should be left up to individual APIs or local policy agents. Though it can be tempting to centralize everything, this can introduce a layer of bureaucracy that can make it harder to make changes. In the worst case, this can violate the principle of least privilege because overly broad policies are left in place due to the overhead of changing them.
- Measure the performance overhead of ABAC policy evaluation early and often.

Pop quiz

7. Which are the four main categories of attributes used in ABAC decisions?
    1. Role
    2. Action
    3. Subject
    4. Resource
    5. Temporal
    6. Geographic
    7. Environment
8. Which one of the components of the XACML reference architecture is used to define and manage policies?
    1. Policy Decision Point
    2. Policy Retrieval Point
    3. Policy Demolition Point
    4. Policy Information Point
    5. Policy Enforcement Point
    6. Policy Administration Point

The answers are at the end of the chapter.

Answers to pop quiz questions

1. True. Many group models allow groups to contain other groups, as discussed in section 8.1.
2. a, c, d. Static and dynamic groups are standard, and virtual static groups are nonstandard but widely implemented.
3. d. `groupOfNames` (or `groupOfUniqueNames`).
4. c, d, e. RBAC only assigns permissions using roles, never directly to individuals. Roles support separation of duty as typically different people define role permissions than those that assign roles to users. Roles are typically defined for each application or API, while groups are often defined globally for a whole organization.
5. c. The NIST model allows a user to activate only some of their roles when creating a session, which enables the principle of least privilege.
6. d. The `@RolesAllowed` annotation determines which roles can all the method.
7. b, c, d, and g. Subject, Resource, Action, and Environment.
8. f. The Policy Administration Point is used to define and manage policies.

Summary

- Users can be collected into groups on an organizational level to make them easier to administer. LDAP has built-in support for managing user groups.
- RBAC collects related sets of permissions on objects into roles which can then be assigned to users or groups and later revoked. Role assignments may be either static or dynamic.
- Roles are often specific to an API, while groups are more often defined statically for a whole organization.
- ABAC evaluates access control decisions dynamically based on attributes of the subject, the resource they are accessing, the action they are attempting to perform, and the environment or context in which the request occurs (such as the time or location).
- ABAC access control decisions can be centralized using a policy engine. The XACML standard defines a common model for ABAC architecture, with separate components for policy decisions (PDP), policy information (PIP), policy administration (PAP), and policy enforcement (PEP).

---

[1] An object class in LDAP defines the schema of a directory entry, describing which attributes it contains.