Red Hat Single Sign-On 7.1

Authorization Services Guide

For Use with Red Hat Single Sign-On 7.1

Red Hat Customer Content
Services

Red Hat Single Sign-On7.1 Authorization Services Guide

For Use with Red Hat Single Sign-On 7.1

Legal Notice

Abstract

This guide consists of information for authorization services for Red Hat Single Sign-On 7.1

Table of Contents

CHAPTER 1. OVERVIEW

Note

Authorization Services is a Technology Preview feature and is not fully supported. This feature is disabled by default.

To enable Authorization Services add the **`standalone/configuration/profile.properties`** file with the contents **`profile=preview`** or start the server with **`-Dkeycloak.profile=preview`** to enable all technology preview features.

Red Hat Single Sign-On supports fine-grained authorization policies and is able to combine different access control mechanisms such as:

- **Attribute-based access control (ABAC)**
- **Role-based access control (RBAC)**
- **User-based access control (UBAC)**
- **Context-based access control (CBAC)**
- **Rule-based access control**
  - Using Javascript
  - Using JBoss Drools
- **Time-based access control**
- **Support for custom access control mechanisms (ACMs) through a Policy Provider Service Provider Interface (SPI)**

Red Hat Single Sign-On is based on a set of administrative UIs and a RESTful API, and provides the necessary means to create permissions for your protected resources and scopes, associate those permissions with authorization policies, and enforce authorization decisions in your applications and services.

Resource servers (applications or services serving protected resources) usually rely on some kind of information to decide if access should be granted to a protected resource. For RESTful-based resource servers, that information is usually obtained from a security token, usually sent as a bearer token on every request to the server. For web applications that rely on a session to authenticate users, that information is usually stored in a user’s session and retrieved from there for each request.

Frequently, resource servers only perform authorization decisions based on role-based access control (RBAC), where the roles granted to the user trying to access protected resources are checked against the roles mapped to these same resources. While roles are very useful and used by applications, they also have a few limitations:

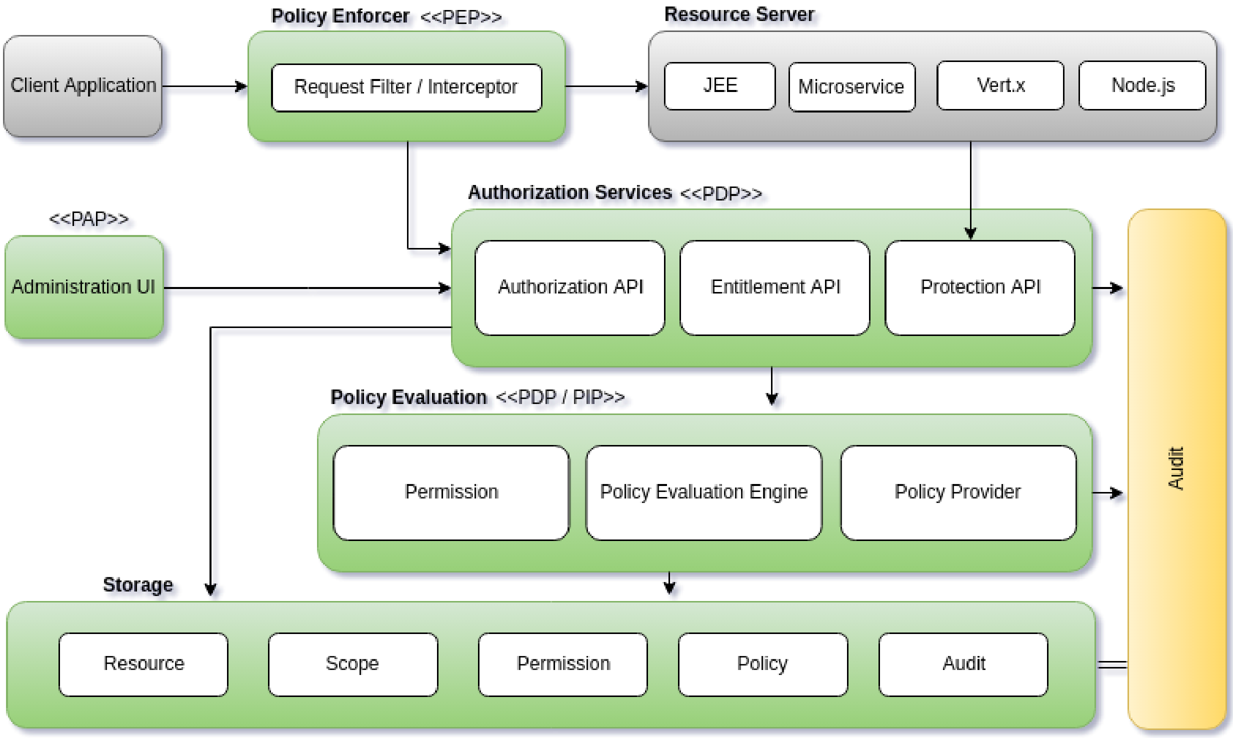
- Resources and roles are tightly coupled and changes to roles (such as adding, removing, or changing an access context) can impact multiple resources
- Changes to your security requirements can imply deep changes to application code to reflect these changes
- Depending on your application size, role management might become difficult and error-prone

- It is not the most flexible access control mechanism. Roles do not represent who you are and lack contextual information. If you have been granted a role, you have at least some access.

Considering that today we need to consider heterogeneous environments where users are distributed across different regions, with different local policies, using different devices, and with a high demand for information sharing, Red Hat Single Sign-On Authorization Services can help you improve the authorization capabilities of your applications and services by providing:

- Resource protection using fine-grained authorization policies and different access control mechanisms
- Centralized Resource, Permission, and Policy Management
- Centralized Policy Decision Point
- REST security based on a set of REST-based authorization services
- Authorization workflows and User-Managed Access
- The infrastructure to help avoid code replication across projects (and redeploys) and quickly adapt to changes in your security requirements.

1.1. ARCHITECTURE



From a design perspective, Authorization Services is based on a well-defined set of authorization patterns providing these capabilities:

- **Policy Administration Point (PAP)**

Provides a set of UIs based on the Red Hat Single Sign-On Administration Console to manage resource servers, resources, scopes, permissions, and policies. Part of this is also accomplished remotely through the use of the Protection API.

- **Policy Decision Point (PDP)**

  Provides a distributable policy decision point to where authorization requests are sent and policies are evaluated accordingly with the permissions being requested. Part of this is also accomplished remotely through the use of the Entitlement APIs.

- **Policy Enforcement Point (PEP)**

  Provides implementations for different environments to actually enforce authorization decisions at the resource server side. Red Hat Single Sign-On provides some built-in Policy Enforcers.

- **Policy Information Point (PIP)**

  Being based on Red Hat Single Sign-On Authentication Server, you can obtain attributes from identities and runtime environment during the evaluation of authorization policies.

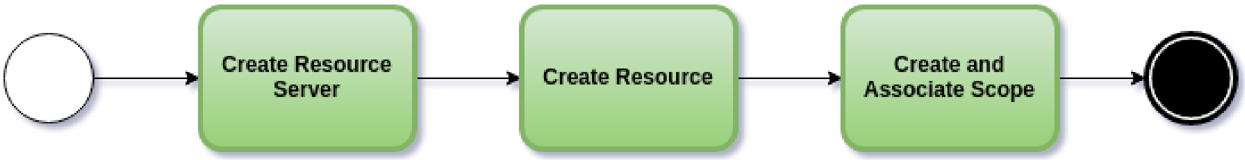
1.1.1. The Authorization Process

Three main processes define the necessary steps to understand how to use Red Hat Single Sign-On to enable fine-grained authorization to your applications:

- **Resource Management**
- **Permission and Policy Management**
- **Policy Enforcement**

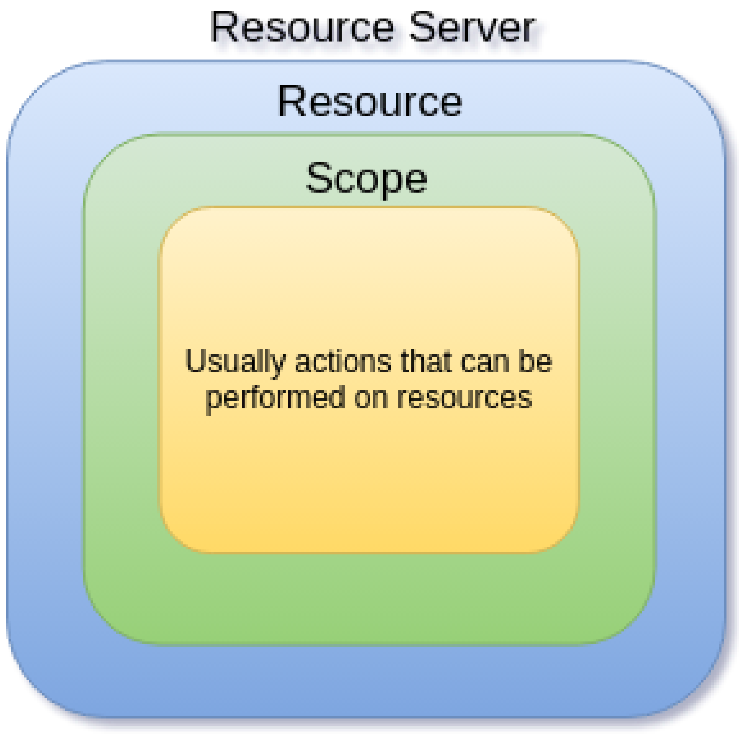
1.1.1.1. Resource Management

Resource Management involves all the necessary steps to define what is being protected.



First, you need to specify Red Hat Single Sign-On what are you looking to protect, which usually represents a web application or a set of one or more services. For more information on resource servers see Terminology.

Resource servers are managed using the Red Hat Single Sign-On Administration Console. There you can enable any registered client application as a resource server and start managing the resources and scopes you want to protect.



A resource can be a web page, a RESTFul resource, a file in your file system, an EJB, and so on. They can represent a group of resources (just like a Class in Java) or they can represent a single and specific resource.

For instance, you might have a *Bank Account* resource that represents all banking accounts and use it to define the authorization policies that are common to all banking accounts. However, you might want to define specific policies for *Alice Account* (a resource instance that belongs to a customer), where only the owner is allowed to access some information or perform an operation.

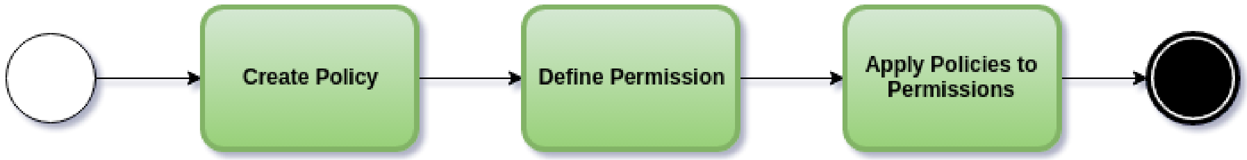
Resources can be managed using the Red Hat Single Sign-On Administration Console or the Protection API. In the latter case, resource servers are able to manage their resources remotely.

Scopes usually represent the actions that can be performed on a resource, but they are not limited to that. You can also use scopes to represent one or more attributes within a resource.

1.1.1.2. Permission and Policy Management

Once you have defined your resource server and all the resources you want to protect, you must set up permissions and policies.

This process involves all the necessary steps to actually define the security and access requirements that govern your resources.



Policies define the conditions that must be satisfied to access or perform operations on something (resource or scope), but they are not tied to what they are protecting. They are generic and can be reused to build permissions or even more complex policies.

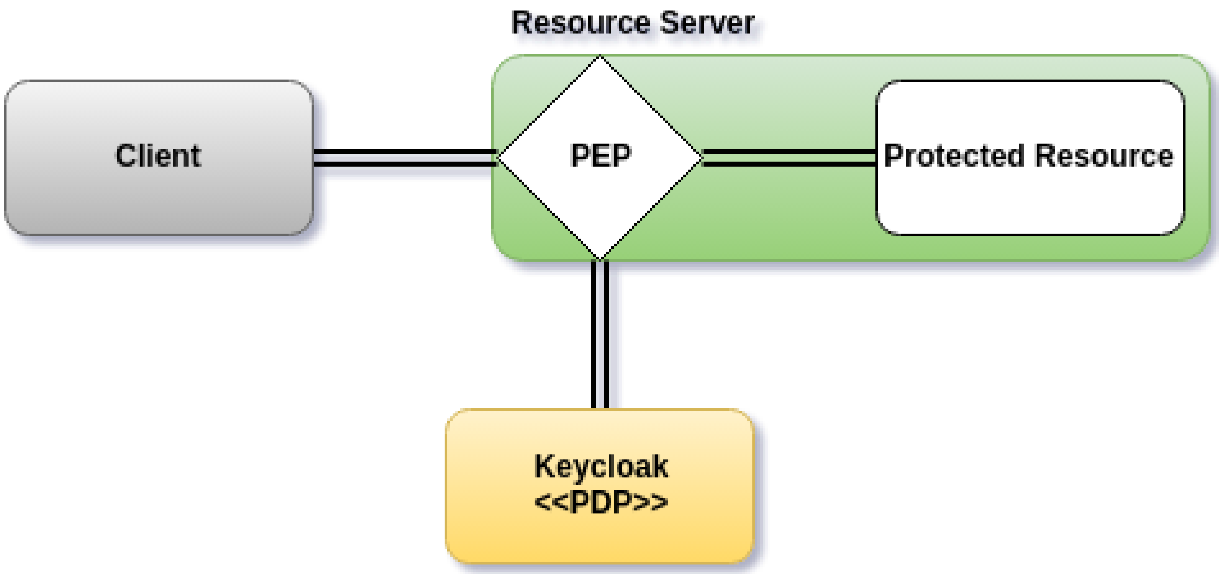
For instance,to allow access to a group of resources only for users granted with a role "User Premium,"" you can use RBAC (Role-based Access Control).

Red Hat Single Sign-On provides a few built-in policy types (and their respective policy providers) covering the most common access control mechanisms. You can even create policies based on rules written using JavaScript or JBoss Drools.

Once you have your policies defined, you can start defining your permissions. Permissions are coupled with the resource they are protecting. Here you specify what you want to protect (resource or scope) and the policies that must be satisfied to grant or deny permission.

1.1.1.3. Policy Enforcement

Policy Enforcement involves the necessary steps to actually enforce authorization decisions to a resource server. This is achieved by enabling a **Policy Enforcement Point** or PEP at the resource server that is capable of communicating with the authorization server, ask for authorization data and control access to protected resources based on the decisions and permissions returned by the server.



Red Hat Single Sign-On provides some built-in Policy Enforcers implementations that you can use to protect your applications depending on the platform they are running on.

1.1.2. Authorization Services

Authorization services consist of the following RESTFul APIs:

- **Protection API**
- **Authorization API**
- **Entitlement API**

Each of these services provides a specific API covering the different steps involved in the authorization process.

1.1.2.1. Protection API

The **Protection API** is a UMA-compliant endpoint providing a small set of operations for resource servers to help them manage their resources and scopes. Only resource servers are allowed to access this API, which also requires a **uma_protection** scope.

The operations provided by the Protection API can be organized in two main groups:

- **Resource Management**
  - Create Resource
  - Delete Resource
  - Find by Id
  - Find All
  - Find with filters (for example, search by name, type, or URI)
- **Permission Management**
  - Issue Permission Tickets

Note

By default, Remote Resource Management is enabled. You can change that using the Red Hat Single Sign-On Administration Console and only allow resource management through the console.

When using the UMA protocol, the issuance of Permission Tickets by the Protection API is an important part of the whole authorization process. As described in a subsequent section, they represent the permissions being requested by the client and that are sent to the server to obtain a final token with all permissions granted during the evaluation of the permissions and policies associated with the resources and scopes being requested.

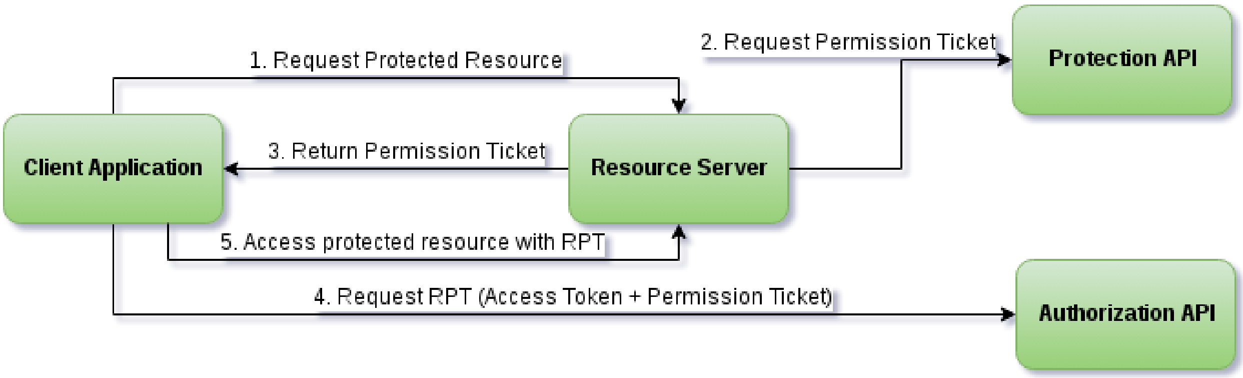
For more information, see Protection API.

1.1.2.2. Authorization API

The Authorization API is also a UMA-compliant endpoint providing a single operation that exchanges an Access Token and Permission Ticket with a Requesting Party Token (RPT).

The RPT contains all permissions granted to a client and can be used to call a resource server to get access to its protected resources.

When requesting an RPT you can also provide a previously issued RPT. In this case, the resulting RPT will consist of the union of the permissions from the previous RPT and the new ones within a permission ticket.


1. Request Protected Resource
2. Request Permission Ticket
Protection API
Client Application
3. Return Permission Ticket
Resource Server
5. Access protected resource with RPT
4. Request RPT (Access Token + Permission Ticket)
Authorization API

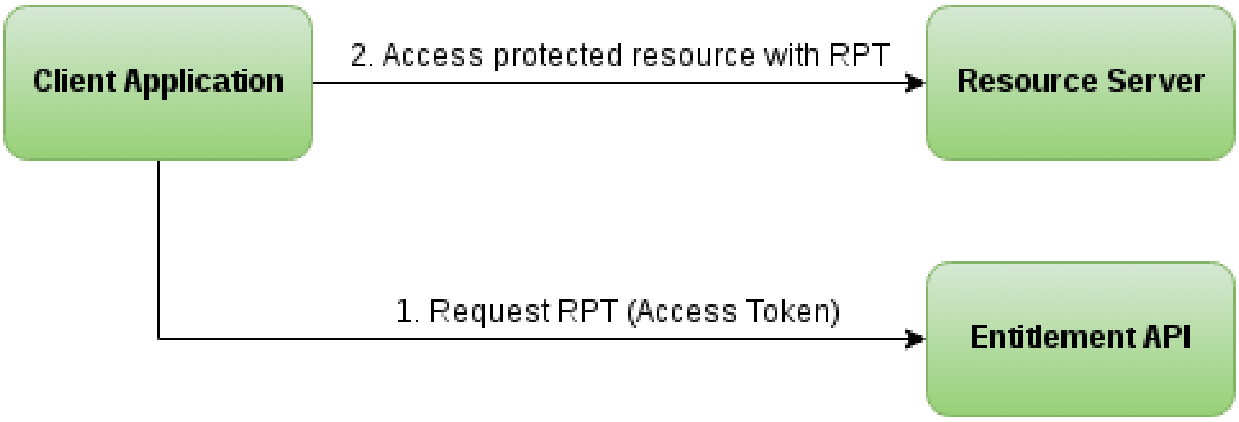
For more information, see Authorization API.

1.1.3. Entitlement API

1.1.3. Entitlement API

The Entitlement API provides a 1-legged protocol to issue RPTs. Unlike the Authorization API, the Entitlement API only expects an access token.

From this API you can obtain all the entitlements or permissions for a user (based on the resources managed by a given resource server) or just the entitlements for a set of one or more resources.


Client Application
2. Access protected resource with RPT
Resource Server
1. Request RPT (Access Token)
Entitlement API

For more information see Entitlement API.

1.2. TERMINOLOGY

Before going further, it is important to understand these terms and concepts introduced by Red Hat Single Sign-On Authorization Services.

1.2.1. Resource Server

Per OAuth2 terminology, a resource server is the server hosting the protected resources and capable of accepting and responding to protected resource requests.

Resource servers usually rely on some kind of information to decide whether access to a protected resource should be granted. For RESTful-based resource servers, that information is usually carried in a security token, typically sent as a bearer token along with every request to the server. Web applications that rely on a session to authenticate users usually store that information in the user’s session and retrieve it from there for each request.

In Red Hat Single Sign-On, any **confidential** client application can act as a resource server. This client’s resources and their respective scopes are protected and governed by a set of authorization policies.

1.2.2. Resource

A resource is part of the assets of an application and the organization. It can be a set of one or more endpoints, a classic web resource such as an HTML page, and so on. In authorization policy terminology, a resource is the *object* being protected.

Every resource has a unique identifier that can represent a single resource or a set of resources. For instance, you can manage a *Banking Account Resource* that represents and defines a set of authorization policies for all banking accounts. But you can also have a different resource named *Alice’s Banking Account*, which represents a single resource owned by a single customer, which can have its own set of authorization policies.

1.2.3. Scope

A resource’s scope is a bounded extent of access that is possible to perform on a resource. In authorization policy terminology, a scope is one of the potentially many *verbs* that can logically apply to a resource.

It usually indicates what can be done with a given resource. Example of scopes are view, edit, delete, and so on. However, scope can also be related to specific information provided by a resource. In this case, you can have a project resource and a cost scope, where the cost scope is used to define specific policies and permissions for users to access a project’s cost.

1.2.4. Permission

Consider this simple and very common permission:

A permission associates the object being protected with the policies that must be evaluated to determine whether access is granted.

- **X** CAN DO **Y** ON RESOURCE **Z**
    - where …
        - **X** represents one or more users, roles, or groups, or a combination of them. You can also use claims and context here.
        - **Y** represents an action to be performed, for example, write, view, and so on.
        - **Z** represents a protected resource, for example, "/accounts".

Red Hat Single Sign-On provides a rich platform for building a range of permission strategies ranging from simple to very complex, rule-based dynamic permissions. It provides flexibility and helps to:

- Reduce code refactoring and permission management costs
- Support a more flexible security model, helping you to easily adapt to changes in your security requirements
- Make changes at runtime; applications are only concerned about the resources and scopes being protected and not how they are protected.

1.2.5. Policy

A policy defines the conditions that must be satisfied to grant access to an object. Unlike permissions, you do not specify the object being protected but rather the conditions that must be satisfied for access to a given object (for example, resource, scope, or both). Policies are strongly related to the different access control mechanisms (ACMs) that you can use to protect your resources. With policies, you can implement strategies for attribute-based access control (ABAC), role-based access control (RBAC), context-based access control, or any combination of these.

Red Hat Single Sign-On leverages the concept of policies and how you define them by providing the concept of aggregated policies, where you can build a "policy of policies" and still control the behavior of the evaluation. Instead of writing one large policy with all the conditions that must be satisfied for access to a given resource, the policies implementation in Red Hat Single Sign-On Authorization Services follows the divide-and-conquer technique. That is, you can create individual policies, then reuse them with different permissions and build more complex policies by combining individual policies.

1.2.6. Policy Provider

Policy providers are implementations of specific policy types. Red Hat Single Sign-On provides built-in policies, backed by their corresponding policy providers, and you can create your own policy types to support your specific requirements.

Red Hat Single Sign-On provides a SPI (Service Provider Interface) that you can use to plug in your own policy provider implementations.

1.2.7. Permission Ticket

A permission ticket is a special type of token defined by the OAuth2's User-Managed Access (UMA) Profile specification that provides an opaque structure whose form is determined by the authorization server. This structure represents the resources and/or scopes being requested by a client as well as the policies that must be applied to a request for authorization data (requesting party token [RPT]).

In UMA, permission tickets are crucial to support person-to-person sharing and also person-to-organization sharing. Using permission tickets for authorization workflows enables a range of scenarios from simple to complex, where resource owners and resource servers have complete control over their resources based on fine-grained policies that govern the access to these resources.

In the UMA workflow, permission tickets are issued by the authorization server to a resource server, which returns the permission ticket to the client trying to access a protected resource. Once the client receives the ticket, it can make a request for an RPT (a final token holding authorization data) by sending the ticket back to the authorization server.

For more information on permission tickets, see Authorization API and the UMA specification.

CHAPTER 2. GETTING STARTED

Before you can use this tutorial, you need to complete the installation of Red Hat Single Sign-On and create the initial admin user as shown in the Getting Started tutorial. There is one caveat to this. You have to run a separate JBoss EAP 7 instance on the same machine as Red Hat Single Sign-On Server. This separate instance will run your Java Servlet application. Because of this you will have to run the Red Hat Single Sign-On under a different port so that there are no port conflicts when running on the same machine. Use the **`jboss.socket.binding.port-offset`** system property on the command line. The value of this property is a number that will be added to the base value of every port opened by Red Hat Single Sign-On Server.

To boot Red Hat Single Sign-On Server:

Linux/Unix

```
$ .../bin/standalone.sh -Djboss.socket.binding.port-offset=100
```

Windows

```
> ...\bin\standalone.bat -Djboss.socket.binding.port-offset=100
```

For more details about how to install and configure a JBoss EAP 7, please follow the steps on the Securing Applications and Services Guide tutorial.

After installing and booting both servers you should be able to access Red Hat Single Sign-On Admin Console at http://localhost:8180/auth/admin/ and also the JBoss EAP 7 instance at http://localhost:8080.

2.1. SECURING A SERVLET APPLICATION

The purpose of this getting started guide is to get you up and running as quickly as possible so that you can experiment with and test various authorization features provided by Red Hat Single Sign-On. This quick tour relies heavily on the default database and server configurations and does not cover complex deployment options. For more information on features or configuration options, see the appropriate sections in this documentation.

This guide explains key concepts about Red Hat Single Sign-On Authorization Services:

- Enabling fine-grained authorization for a client application
- Configuring a client application to be a resource server, with protected resources
- Defining permissions and authorization policies to govern access to protected resources
- Enabling policy enforcement in your applications.

2.2. CREATING A REALM AND A USER

The first step is to create a realm and a user in that realm. The realm consists of:

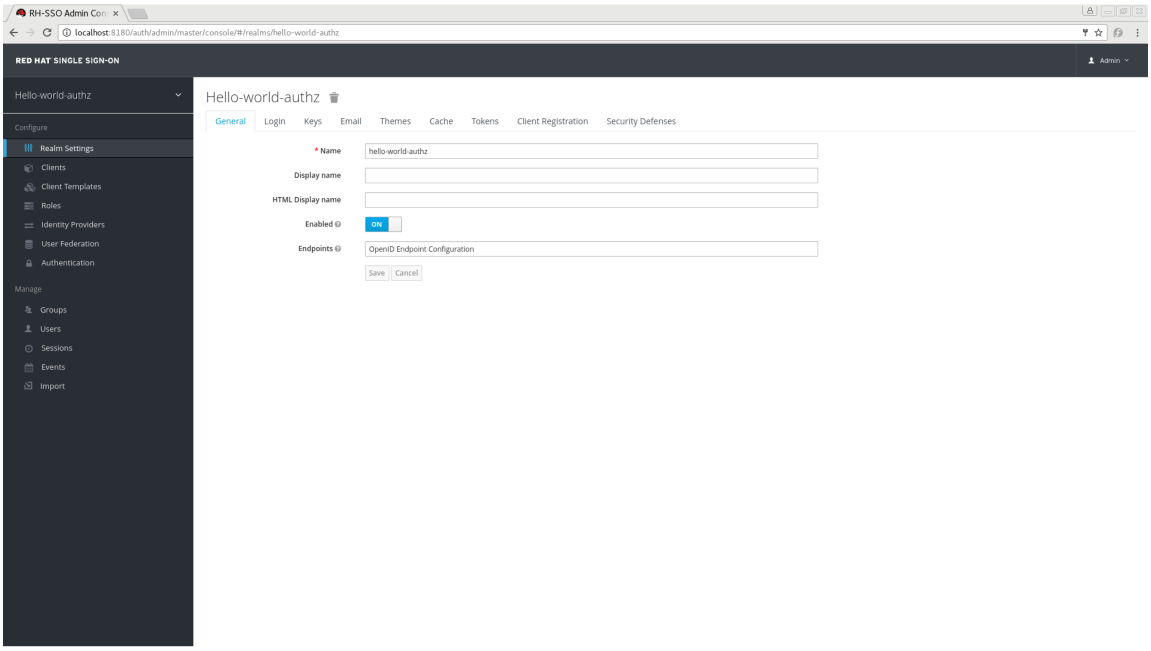
- A single user

- A single client application, which then becomes a resource server for which you need to enable authorization services.

To create a realm and a user complete the following steps:

1. Create a realm with a name **hello-world-authz**. Once created, a page similar to the following is displayed:

Realm hello-world-authz



* Name: hello-world-authz

Display name:

HTML Display name:

Enabled: ON

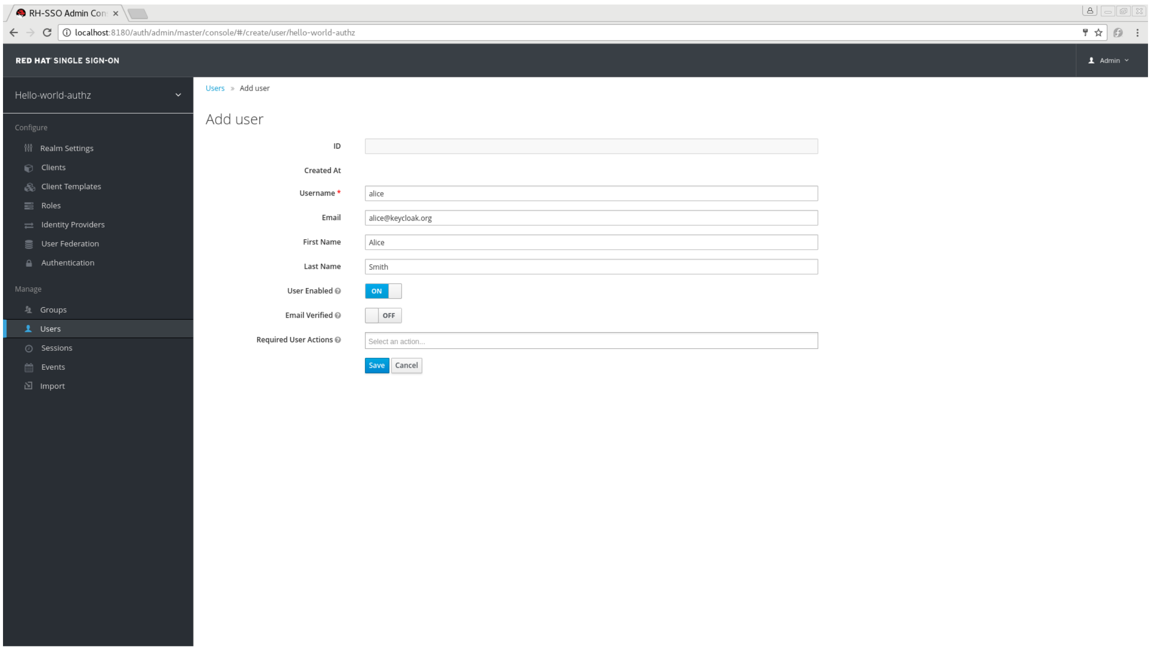
Endpoints: OpenID Endpoint Configuration

2. Create a user for your newly created realm. Click **Users**. The user list page opens.

3. On the right side of the empty user list, click **Add User**.

4. To create a new user, complete the **Username**, **Email**, **First Name**, and **Last Name** fields. Click the **User Enabled** switch to **On**, and then click **Save**.

Add User



localhost:8180/auth/admin/master/console/#/create/user/hello-world-authz

Users » Add user

Add user

ID

Created At

Username * alice

Email alice@keycloak.org

First Name Alice

Last Name Smith

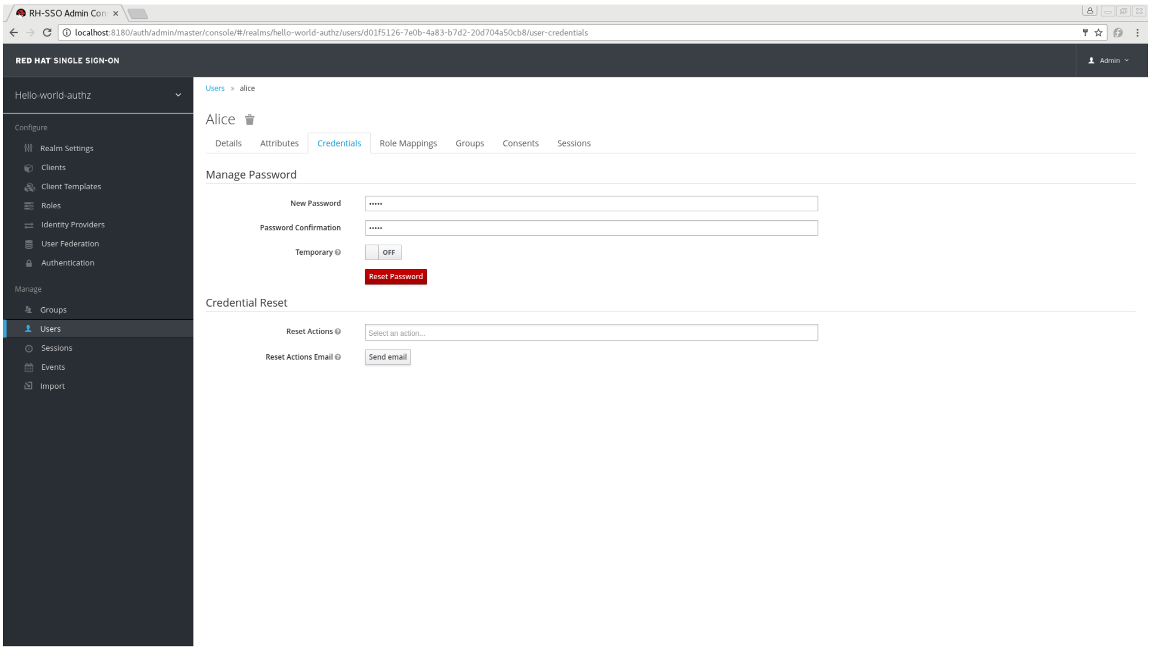
User Enabled ON

Email Verified OFF

Required User Actions Select an action...

5. Set a password for the user by clicking the **Credentials** tab.

Set User Password



localhost:8180/auth/admin/master/console/#/realms/hello-world-authz/users/d01f5126-7e0b-4a83-b7d2-20d704a50cb8/user-credentials

Users » alice

Alice

Manage Password

New Password •••••

Password Confirmation •••••

Temporary OFF

Credential Reset

Reset Actions Select an action...

6. Complete the **New Password** and **Password Confirmation** fields with a password and click the **Temporary** switch to **OFF**.

7. Click **Reset Password** to set the user’s password.

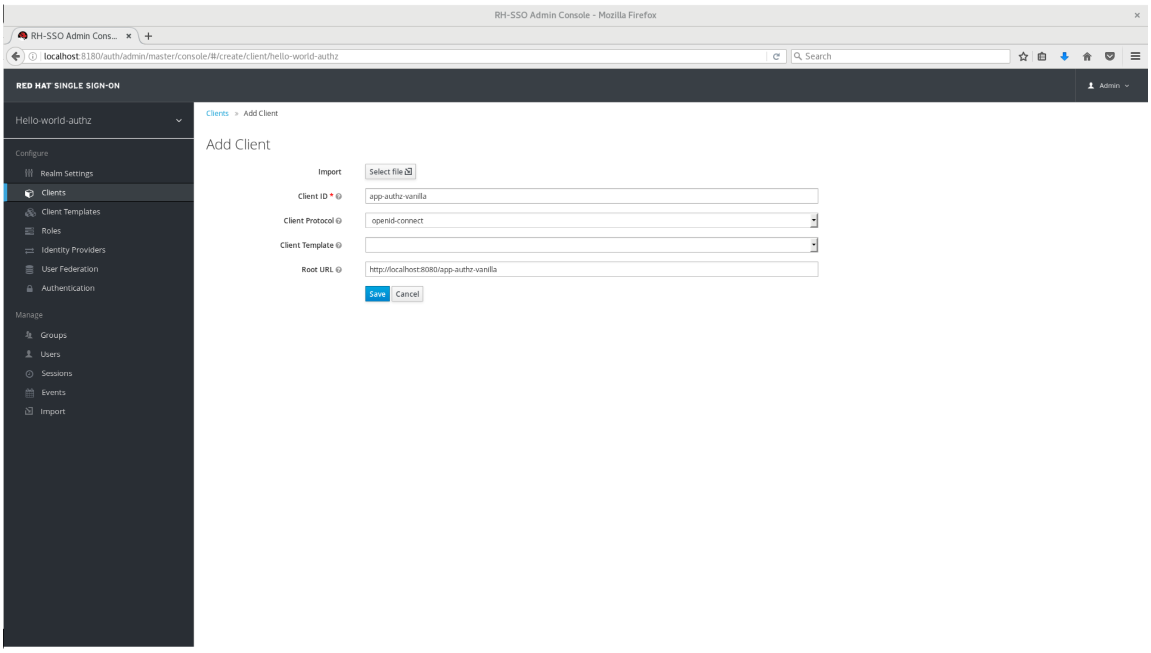
2.3. ENABLING AUTHORIZATION SERVICES

You can enable authorization services in an existing client application configured to use the OpenID Connect Protocol. You can also create a new client.

To create a new client, complete the following steps:

1. Click **Clients** to start creating a new client application and fill in the **Client ID**, **Client Protocol**, and **Root URL** fields.

Create Client Application



localhost:8180/auth/admin/master/console/#/create/client/hello-world-authz

Add Client

Import: Select file

Client ID *: app-authz-vanilla

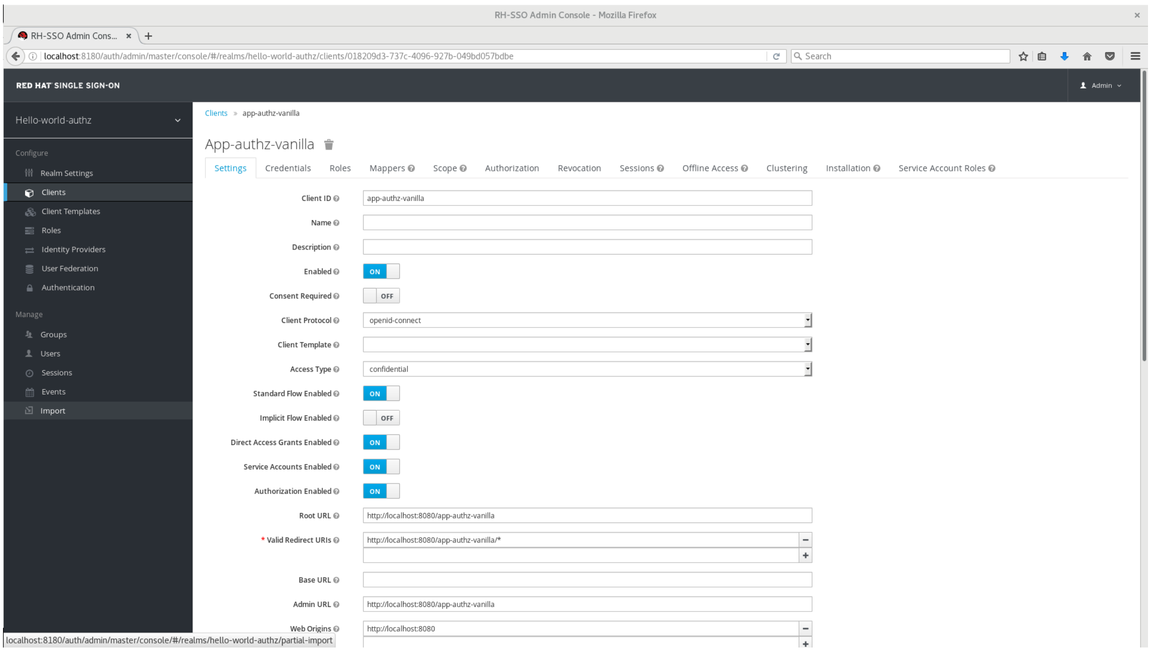
Client Protocol: openid-connect

Client Template:

Root URL: http://localhost:8080/app-authz-vanilla

2. Click **Save**. The Client Details page is displayed.

Client Details



localhost:8180/auth/admin/master/console/#/realms/hello-world-authz/clients/018209d3-737c-4096-927b-049bd057bdbe

App-authz-vanilla

Client ID: app-authz-vanilla

Name:

Description:

Enabled: ON

Consent Required: OFF

Client Protocol: openid-connect

Client Template:

Access Type: confidential

Standard Flow Enabled: ON

Implicit Flow Enabled: OFF

Direct Access Grants Enabled: ON

Service Accounts Enabled: ON

Authorization Enabled: ON

Root URL: http://localhost:8080/app-authz-vanilla

* Valid Redirect URIs: http://localhost:8080/app-authz-vanilla/*

Base URL:

Admin URL: http://localhost:8080/app-authz-vanilla

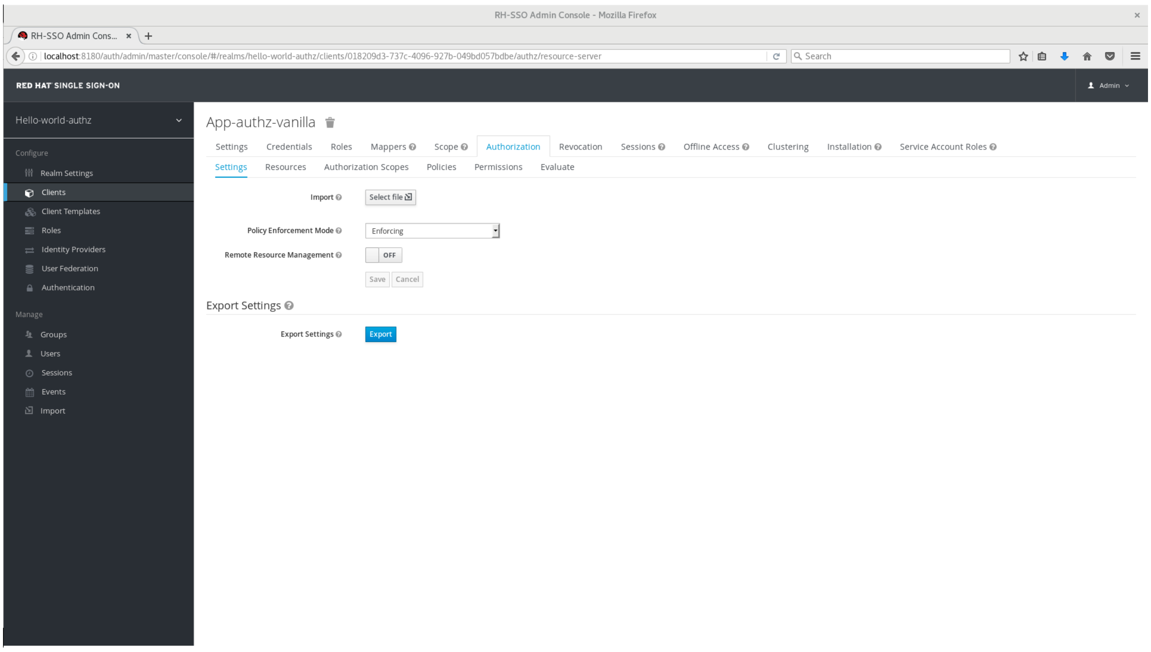
Web Origins: http://localhost:8080

localhost:8180/auth/admin/master/console/#/realms/hello-world-authz/partial-import

3. On the Client Details page, click the **Authorization Enabled** switch to **ON**, and then click **Save**. A new **Authorization** tab is displayed for the client.

4. Click the **Authorization** tab and an Authorization Settings page similar to the following is displayed:

Authorization Settings



Import

Policy Enforcement Mode: Enforcing

Remote Resource Management: OFF

Export Settings

Export Settings

When you enable authorization services for a client application, Red Hat Single Sign-On automatically creates several default settings for your client authorization configuration.

For more information about authorization configuration, see Enabling Authorization Services.

2.4. BUILD, DEPLOY, AND TEST YOUR APPLICATION

Now that the **app-authz-vanilla** resource server (or client) is properly configured and authorization services are enabled, it can be deployed to the server.

The project and code for the application you are going to deploy is available in Quickstarts for the Red Hat Single Sign-On (SSO) Server. You will need the following installed on your machine and available in your PATH before you can continue:

- Java JDK 8
- Apache Maven 3.1.1 or higher
- Git

You can obtain the code by cloning the repository at https://github.com/redhat-developer/redhat-sso-quickstarts. Use the branch matching the version of Red Hat Single Sign-On in use. Follow these steps to download the code.

Clone Project

```
$ git clone https://github.com/redhat-developer/redhat-sso-quickstarts
```

The application we are about to build and deploy is located at

```
$ cd redhat-sso-quickstarts/app-authz-jee-vanilla
```

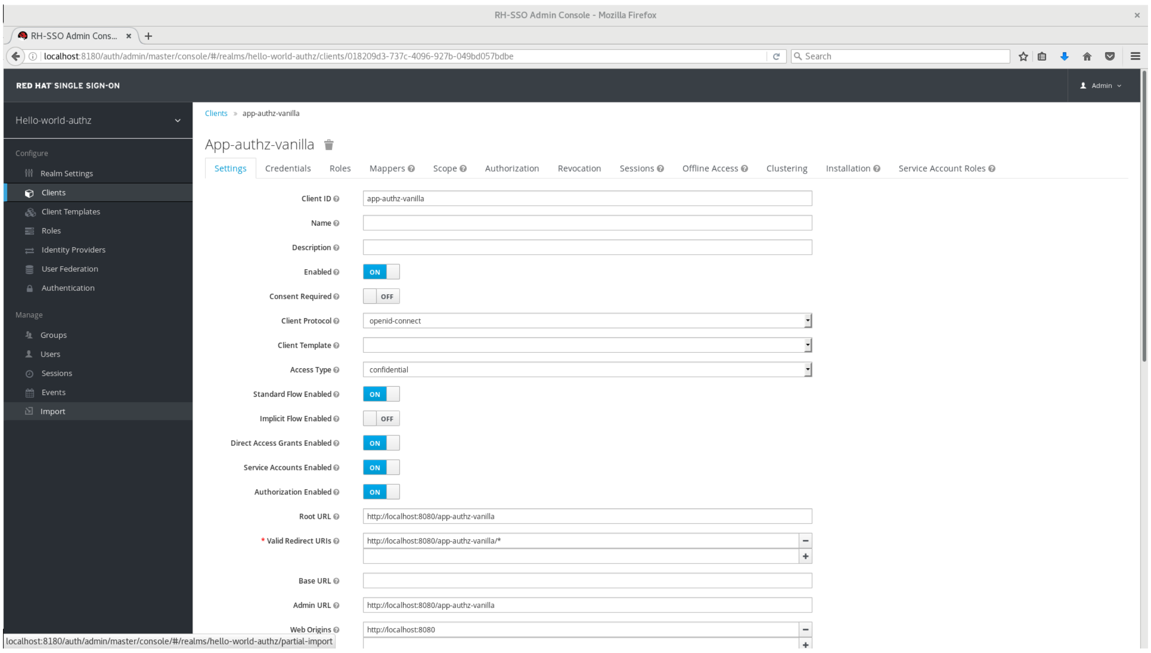
2.4.1. Obtaining the Adapter Configuration

You must first obtain the adapter configuration before building and deploying the application.

To obtain the adapter configuration from the Red Hat Single Sign-On Administration Console, complete the following steps.

1. Click **Clients**. In the client listing, click the **app-authz-vanilla** client application. The Client Details page opens.

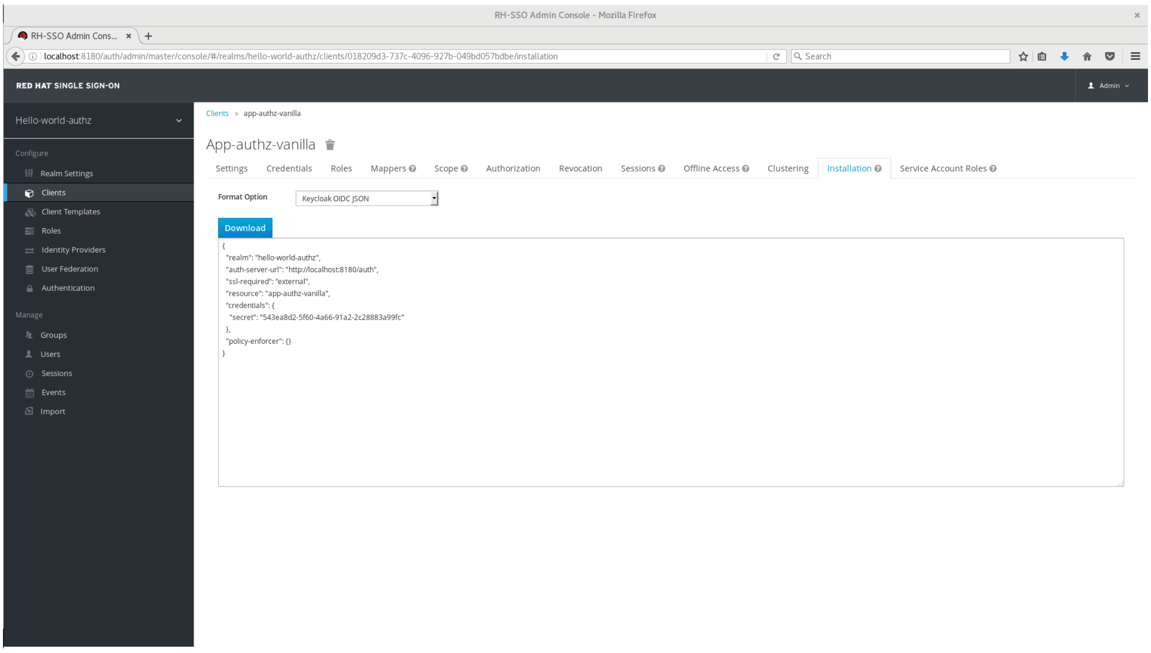
Client Details



Client ID: app-authz-vanilla
Name:
Description:
Enabled: ON
Consent Required: OFF
Client Protocol: openid-connect
Client Template:
Access Type: confidential
Standard Flow Enabled: ON
Implicit Flow Enabled: OFF
Direct Access Grants Enabled: ON
Service Accounts Enabled: ON
Authorization Enabled: ON
Root URL: http://localhost:8080/app-authz-vanilla
* Valid Redirect URIs: http://localhost:8080/app-authz-vanilla/*
Base URL:
Admin URL: http://localhost:8080/app-authz-vanilla
Web Origins: http://localhost:8080

2. Click the **Installation** tab. From the Format Option dropdown list, select **Keycloak OIDC JSON**. The adapter configuration is displayed in JSON format. Click **Download**.

Adapter Configuration



Format Option: Keycloak OIDC JSON

```
{
  "realm": "hello-world-authz",
  "auth-server-url": "http://localhost:8180/auth",
  "ssl-required": "external",
  "resource": "app-authz-vanilla",
  "credentials": {
    "secret": "543ea8d2-5f60-4a66-91a2-2c28883a99fc"
  },
  "policy-enforcer": {}
}
```

3. Move the file **`keycloak.json`** to the **`app-authz-jee-vanilla/config`** directory.

4. (optional) By default, the policy enforcer responds with a **`403`** status code when the user lacks permission to access protected resources on the resource server. However, you can

also specify a redirection URL for unauthorized users. To specify a redirection URL, edit the **keycloak.json** file you updated in step 3 and replace the **policy-enforcer** configuration with the following:

```
"policy-enforcer": {
    "on-deny-redirect-to" : "/app-authz-vanilla/error.jsp"
}
```

This change specifies to the policy enforcer to redirect users to a **/app-authz-vanilla/error.jsp** page if a user does not have the necessary permissions to access a protected resource, rather than an unhelpful **403 Unauthorized** message.

2.4.2. Building and Deploying the Application

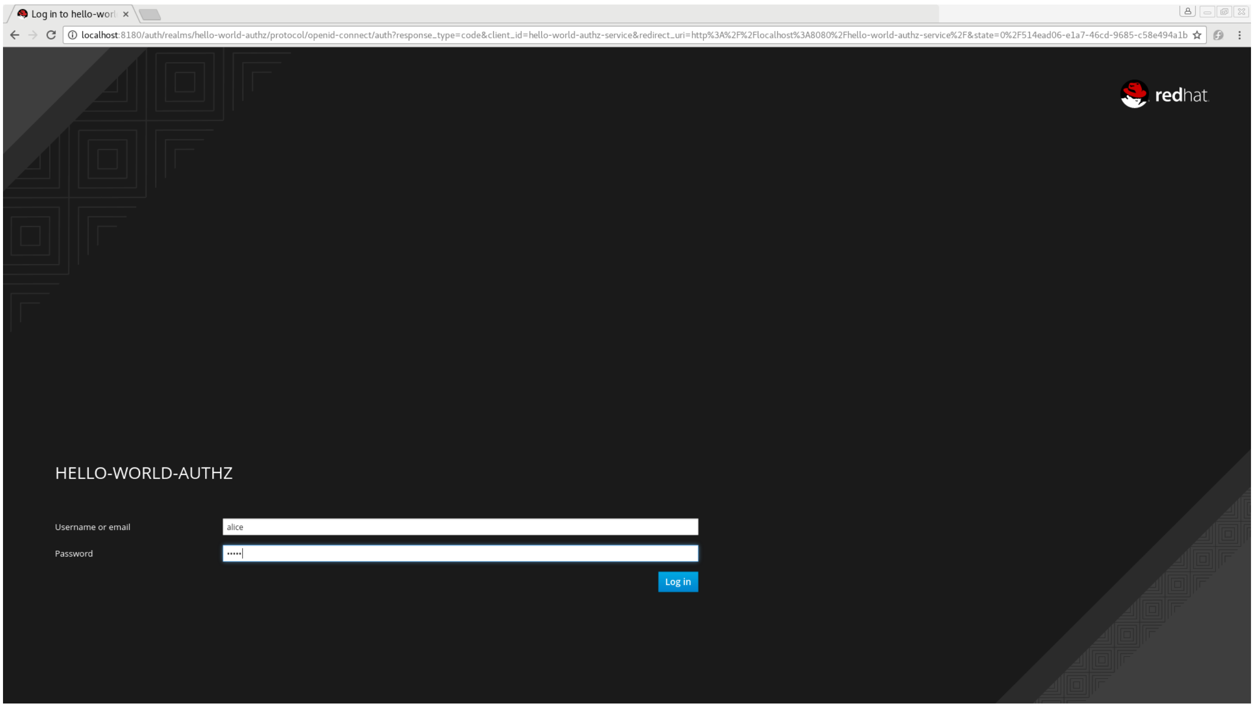
To build and deploy the application execute the following command:

```
$ cd redhat-sso-quickstarts/app-authz-jee-vanilla
$ mvn clean package wildfly:deploy
```

2.4.3. Testing the Application

If your application was successfully deployed you can access it at http://localhost:8080/app-authz-vanilla. The Red Hat Single Sign-On Login page opens.

Login Page



localhost:8180/auth/realms/hello-world-authz/protocol/openid-connect/auth?response_type=code&client_id=hello-world-authz-service&redirect_uri=http%3A%2F%2Flocalhost%3A8080%2Fhello-world-authz-service%2F&state=0%2F514ead06-e1a7-46cd-9685-c58e494a1b

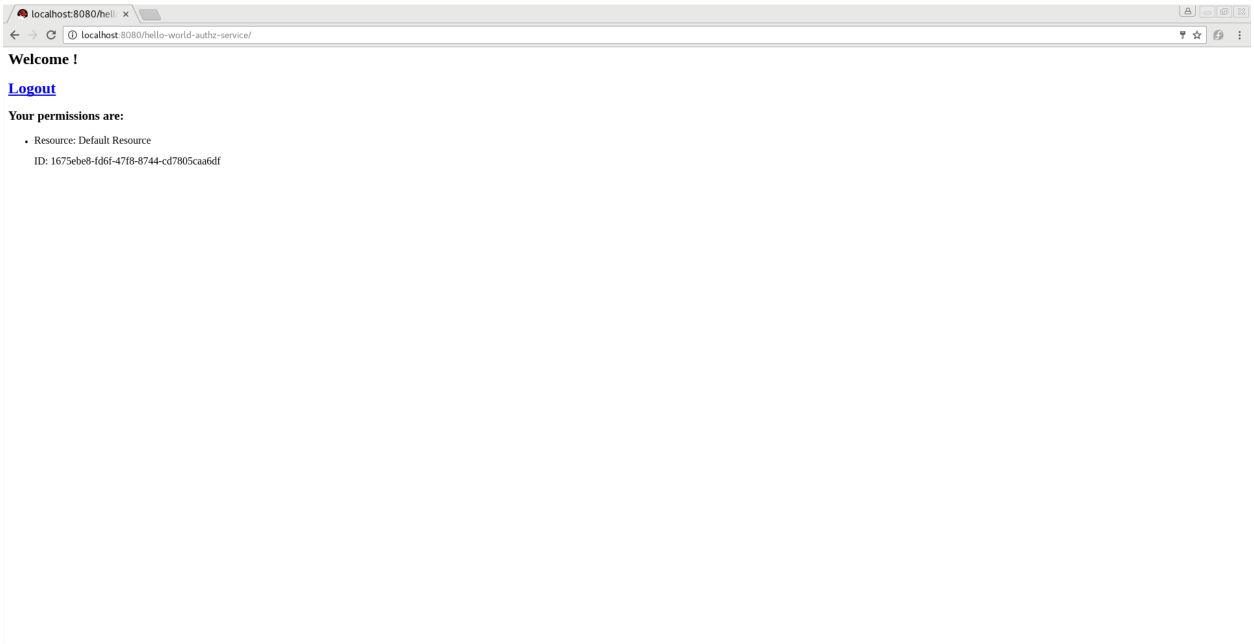
HELLO-WORLD-AUTHZ

Username or email: alice

Password: •••••

Log in as **alice** using the password you specified for that user. After authenticating, the following page is displayed:

Hello World Authz Main Page



Welcome !

Logout

Your permissions are:

- Resource: Default Resource

  ID: 1675ebe8-fd6f-47f8-8744-cd7805caa6df

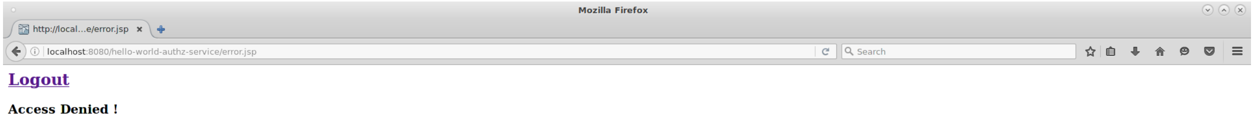
The default settings defined by Red Hat Single Sign-On when you enable authorization services for a client application provide a simple policy that always grants access to the resources protected by this policy.

You can start by changing the default permissions and policies and test how your application responds, or even create new policies using the different policy types provided by Red Hat Single Sign-On.

There are a plenty of things you can do now to test this application. For example, you can change the default policy by clicking the Authorization tab for the client, then **`Policies`** tab, then click on **`Default Policy`** in the list to allow you to change it as follows:

```
// The default value is $evaluation.grant(),
// let's see what happens when we change it to $evaluation.deny()
$evaluation.deny();
```

Now, log out of the demo application and log in again. You can no longer access the application.



Mozilla Firefox
http://local...e/error.jsp
localhost:8080/hello-world-authz-service/error.jsp
Search

Logout

Access Denied !

Let’s fix that now, but instead of changing the **Default Policy** code we are going to change the **Logic** to **Negative** using the dropdown list below the policy code text area. That re-enables access to the application as we are negating the result of that policy, which is by default denying all requests for access. Again, before testing this change, be sure to log out and log in again.

2.4.4. Next Steps

There are additional things you can do, such as:

- Create a scope, define a policy and permission for it, and test it on the application side. Can the user perform an action (or anything else represented by the scope you created)?
- Create different types of policies such as rule-based, and associate these policies with the **Default Permission**.
- Apply multiple policies to the **Default Permission** and test the behavior. For example, combine multiple policies and change the **Decision Strategy** accordingly.
- For more information about how to view and test permissions inside your application see Obtaining the Authorization Context.

CHAPTER 3. MANAGING RESOURCE SERVERS

According to the OAuth2 specification, a resource server is a server hosting the protected resources and capable of accepting and responding to protected resource requests.

In Red Hat Single Sign-On, resource servers are provided with a rich platform for enabling fine-grained authorization for their protected resources, where authorization decisions can be made based on different access control mechanisms.

Any client application can be configured to support fine-grained permissions. In doing so, you are conceptually turning the client application into a resource server.

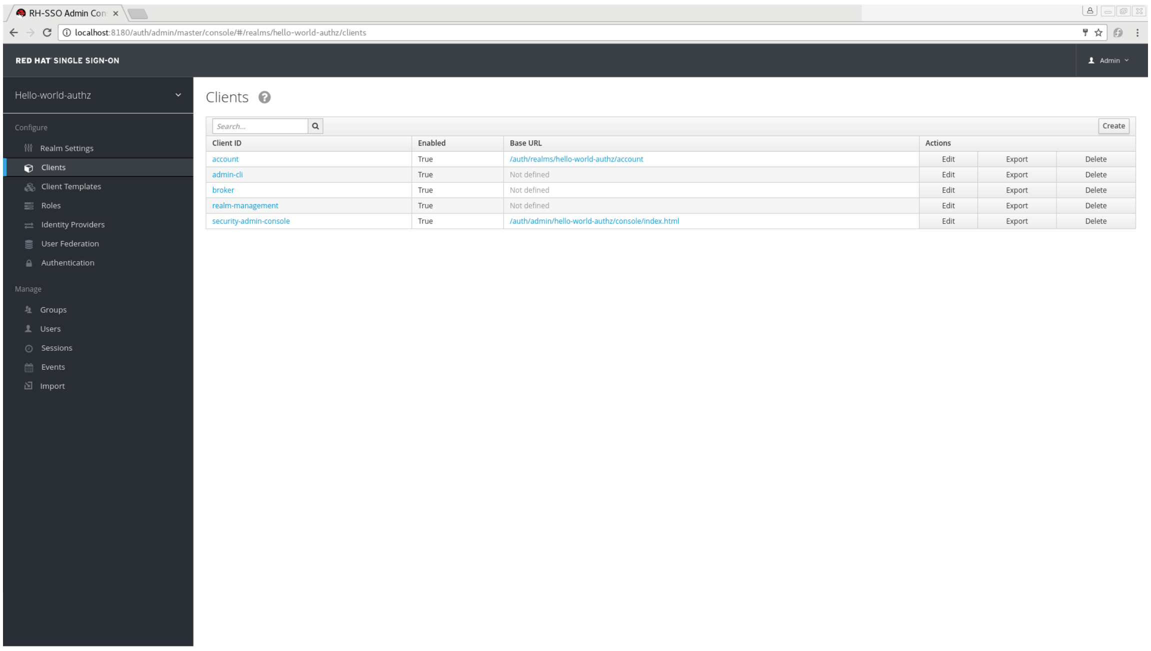
3.1. CREATING A CLIENT APPLICATION

The first step to enable Red Hat Single Sign-On Authorization Services is to create the client application that you want to turn into a resource server.

To create a client application, complete the following steps:

1. Click **Clients**.

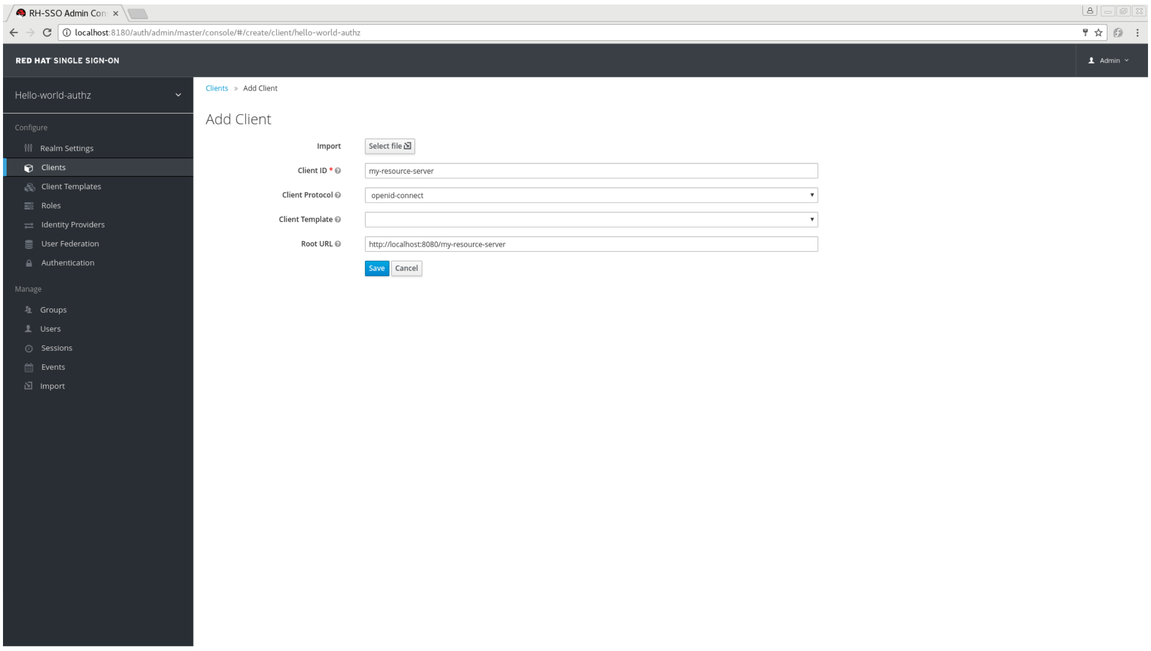
Clients



Client ID	Enabled	Base URL	Actions		
account	True	/auth/realms/hello-world-authz/account	Edit	Export	Delete
admin-cli	True	Not defined	Edit	Export	Delete
broker	True	Not defined	Edit	Export	Delete
realm-management	True	Not defined	Edit	Export	Delete
security-admin-console	True	/auth/admin/hello-world-authz/console/index.html	Edit	Export	Delete

2. On this page, click **Create**.

Create Client



localhost:8180/auth/admin/master/console/#/create/client/hello-world-authz

Add Client

Import: Select file

Client ID *: my-resource-server

Client Protocol: openid-connect

Client Template:

Root URL: http://localhost:8080/my-resource-server

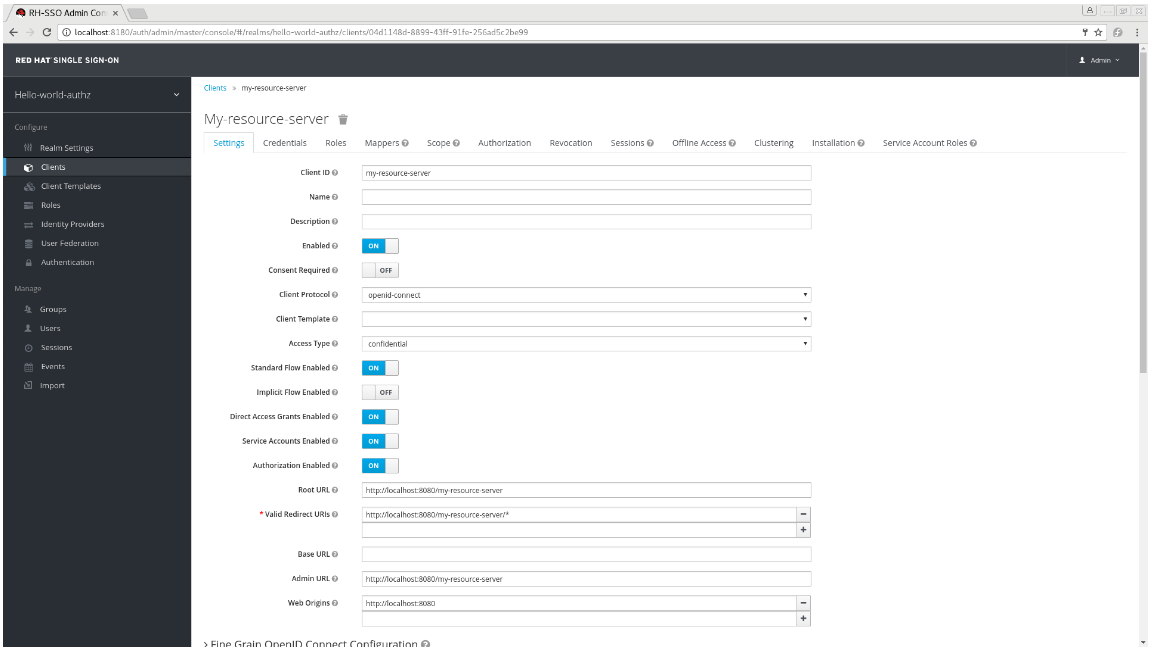
3. Type the **`Client ID`** of the client. For example, *my-resource-server*.

4. Type the **`Root URL`** for your application. For example:

```
http://${host}:${port}/my-resource-server
```

5. Click **Save**. The client is created and the client Settings page opens. A page similar to the following is displayed:

Client Settings



localhost:8180/auth/admin/master/console/#/realms/hello-world-authz/clients/04d1148d-8899-43ff-91fe-256ad5c2be99

My-resource-server

Client ID: my-resource-server

Name:

Description:

Enabled: ON

Consent Required: OFF

Client Protocol: openid-connect

Client Template:

Access Type: confidential

Standard Flow Enabled: ON

Implicit Flow Enabled: OFF

Direct Access Grants Enabled: ON

Service Accounts Enabled: ON

Authorization Enabled: ON

Root URL: http://localhost:8080/my-resource-server

* Valid Redirect URIs: http://localhost:8080/my-resource-server/*

Base URL:

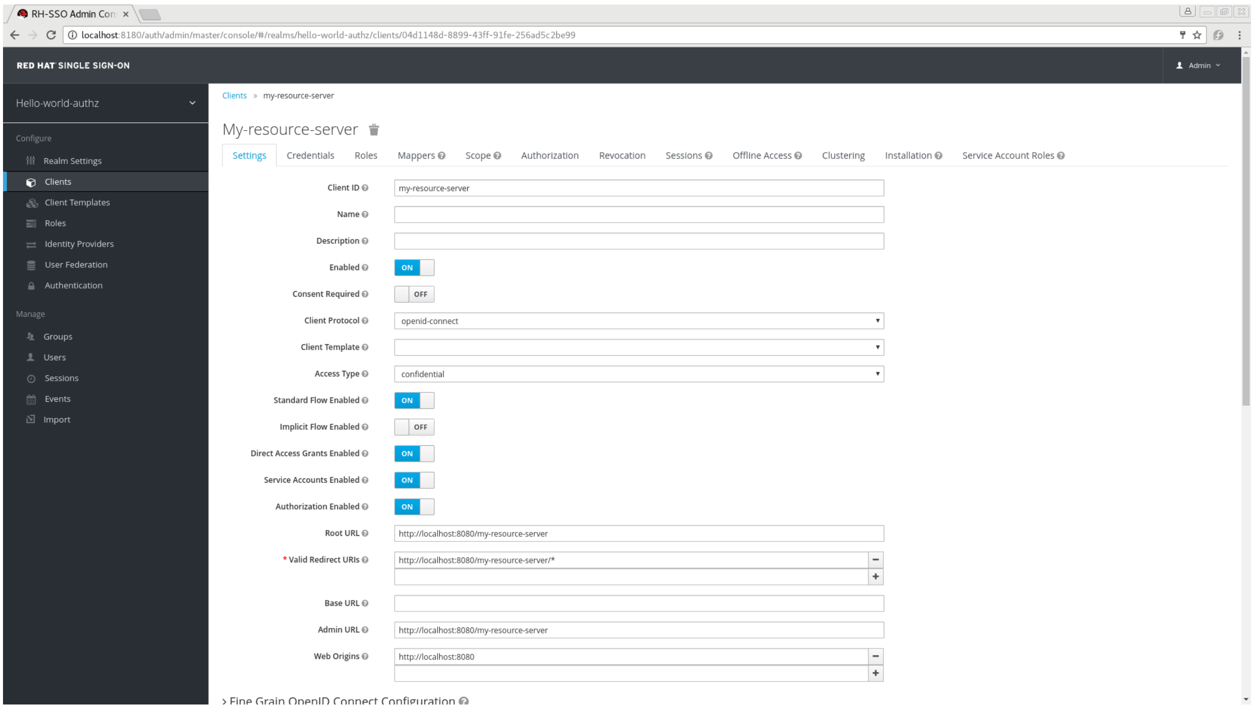
Admin URL: http://localhost:8080/my-resource-server

Web Origins: http://localhost:8080

3.2. ENABLING AUTHORIZATION SERVICES

To turn your OIDC Client Application into a resource server and enable fine-grained authorization, click the **Authorization Enabled** switch to **ON** and click **Save**.

Enabling Authorization Services



localhost:8180/auth/admin/master/console/#/realms/hello-world-authz/clients/04d1148d-8899-43ff-91fe-256ad5c2be99

My-resource-server

Client ID: my-resource-server

Name:

Description:

Enabled: ON

Consent Required: OFF

Client Protocol: openid-connect

Client Template:

Access Type: confidential

Standard Flow Enabled: ON

Implicit Flow Enabled: OFF

Direct Access Grants Enabled: ON

Service Accounts Enabled: ON

Authorization Enabled: ON

Root URL: http://localhost:8080/my-resource-server

* Valid Redirect URIs: http://localhost:8080/my-resource-server/*

Base URL:

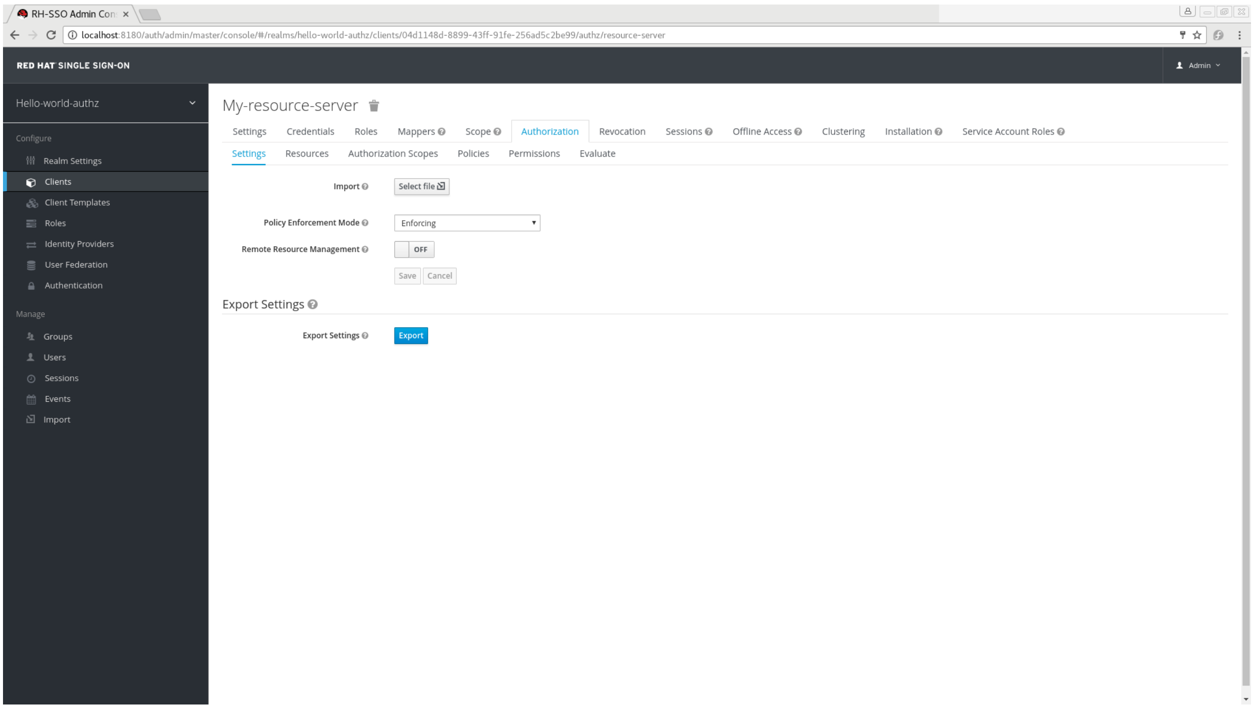
Admin URL: http://localhost:8080/my-resource-server

Web Origins: http://localhost:8080

› Fine Grain OpenID Connect Configuration

A new Authorization tab is displayed for this client. Click the **Authorization** tab and a page similar to the following is displayed:

Resource Server Settings



localhost:8180/auth/admin/master/console/#/realms/hello-world-authz/clients/04d1148d-8899-43ff-91fe-256ad5c2be99/authz/resource-server

My-resource-server

Import: Select file

Policy Enforcement Mode: Enforcing

Remote Resource Management: OFF

Export Settings

Export Settings: Export

The Authorization tab contains additional sub-tabs covering the different steps that you must follow to actually protect your application’s resources. Each tab is covered separately by a specific topic in this documentation. But here is a quick description about each one:

- **Settings**

  General settings for your resource server. For more details about this page see the Resource Server Settings section.

- **Resource**

  From this page, you can manage your application’s resources.

- **Scope**

  From this page, you can manage scopes.

- **Policies**

  From this page, you can manage authorization policies and define the conditions that must be met to grant a permission.

- **Permissions**

  From this page, you can manage the permissions for your protected resources and scopes by linking them with the policies you created.

- **Evaluate**

  From this page, you can simulate authorization requests and view the result of the evaluation of the permissions and authorization policies you have defined.

3.2.1. Resource Server Settings

On the Resource Server Settings page, you can configure the policy enforcement mode, allow remote resource management, and export the authorization configuration settings.

- **Policy Enforcement Mode**

  Specifies how policies are enforced when processing authorization requests sent to the server.

  - **Enforcing**

    (default mode) Requests are denied by default even when there is no policy associated with a given resource.

  - **Permissive**

    Requests are allowed even when there is no policy associated with a given resource.

  - **Disabled**

    Disables the evaluation of all policies and allows access to all resources.

- **Allow Remote Resource Management**

  Specifies whether resources can be managed remotely by the resource server. If false, resources can be managed only from the administration console.

- **Export Settings**

  You can export the authorization configuration settings to a JSON file. Click **Export** to display the complete JSON configuration for download. The configuration file contains everything defined for a resource server: protected resources, scopes, permissions, and policies.

3.3. DEFAULT CONFIGURATION

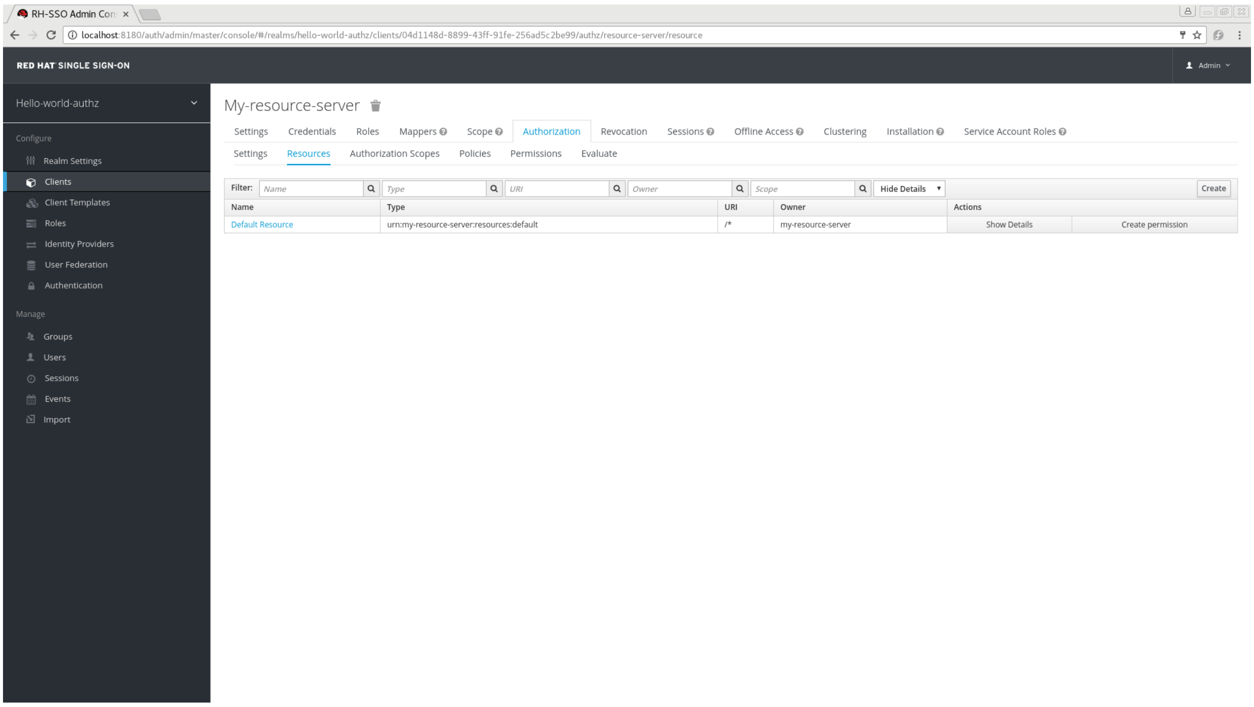
When you create a resource server, Red Hat Single Sign-On creates a default configuration for your newly created resource server.

The default configuration consists of:

- A default protected resource representing all resources in your application.
- A policy that always grants access to the resources protected by this policy.
- A permission that governs access to all resources based on the default policy.

The default protected resource is referred to as the **default resource** and you can view it if you navigate to the **Resources** tab.

Default Resource



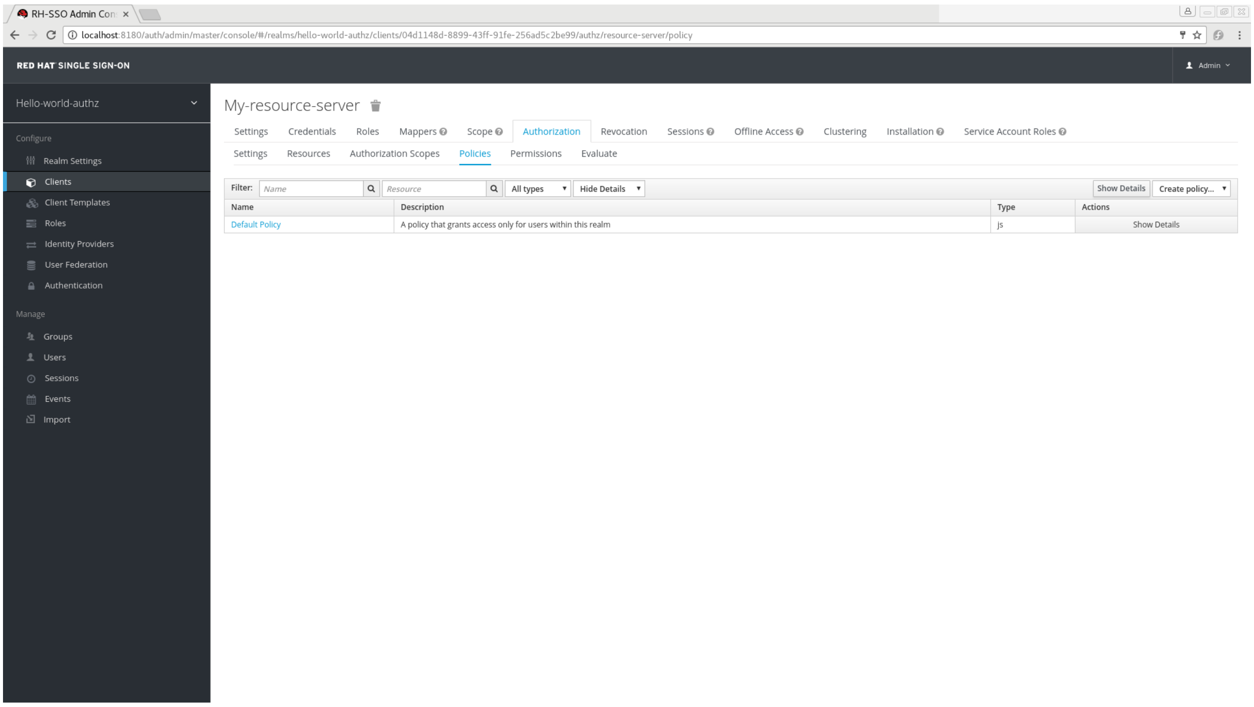
Name	Type	URI	Owner	Actions	
Default Resource	urn:my-resource-server:resources:default	/*	my-resource-server	Show Details	Create permission

This resource defines a **Type**, namely **`urn:my-resource-server:resources:default`** and a **`URI`** **`/*`**. Here, the **`URI`** field defines a wildcard pattern that indicates to Red Hat Single Sign-On that this resource represents all the paths in your application. In other words, when enabling policy enforcement for your application, all the permissions associated with the resource will be examined before granting access.

The **Type** mentioned previously defines a value that can be used to create typed resource permissions that must be applied to the default resource or any other resource you create using the same type.

The default policy is referred to as the **only from realm policy** and you can view it if you navigate to the **Policies** tab.

Default Policy



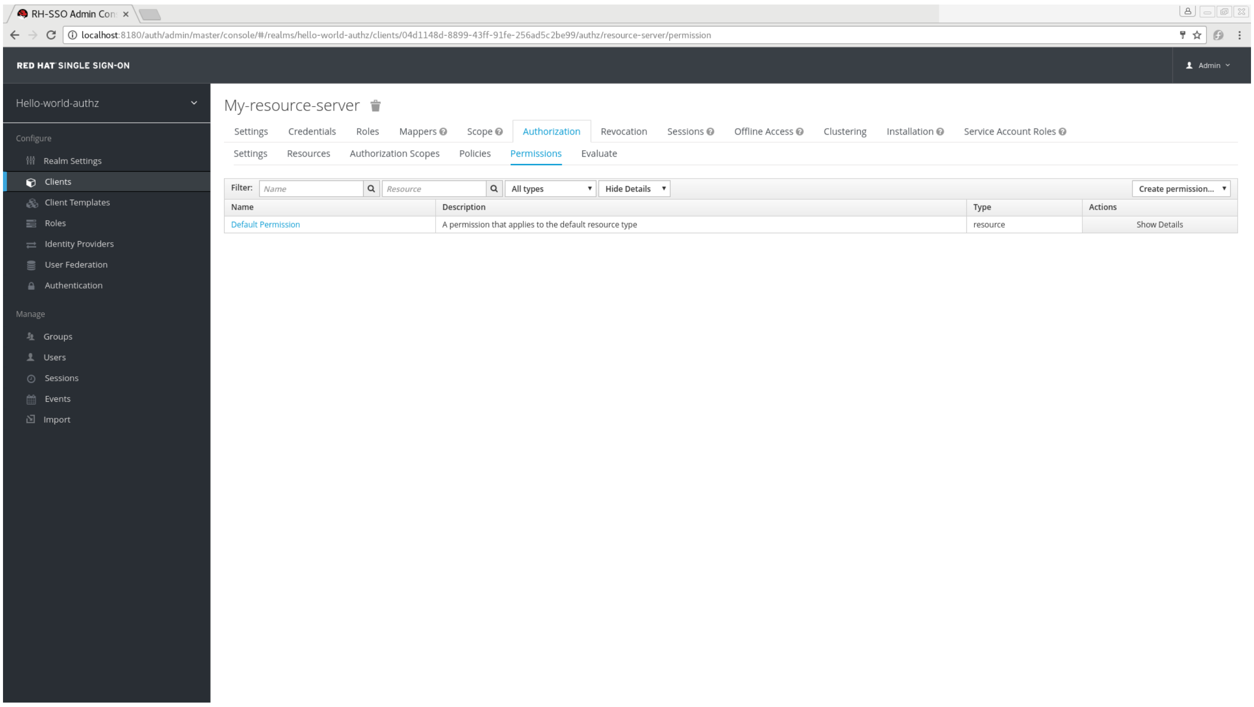
Name	Description	Type	Actions
Default Policy	A policy that grants access only for users within this realm	js	Show Details

This policy is a JavaScript-based policy defining a condition that always grants access to the resources protected by this policy. If you click this policy you can see that it defines a rule as follows:

```
// by default, grants any permission associated with this policy
$evaluation.grant();
```

Lastly, the default permission is referred to as the **default permission** and you can view it if you navigate to the **Permissions** tab.

Default Permission



Name	Description	Type	Actions
Default Permission	A permission that applies to the default resource type	resource	Show Details

This permission is a resource-based permission, defining a set of one or more policies that are applied to all resources with a given type.

2.2.1. Changing the Default Configuration

3.3.1. Changing the Default Configuration

You can change the default configuration by removing the default resource, policy, or permission definitions and creating your own.

Note

The default configuration defines a resource that maps to all paths in your application. If you are about to write permissions to your own resources, be sure to remove the **Default Resource** or change its `URI` field to a more specific path in your application. Otherwise, the policy associated with the default resource (which by default always grants access) will allow Red Hat Single Sign-On to grant access to any protected resource.

3.4. EXPORT AND IMPORT AUTHORIZATION CONFIGURATION

The configuration settings for a resource server (or client) can be exported and downloaded. You can also import an existing configuration file for a resource server. Importing and exporting a configuration file is helpful when you want to create an initial configuration for a resource server or to update an existing configuration. The configuration file contains definitions for:

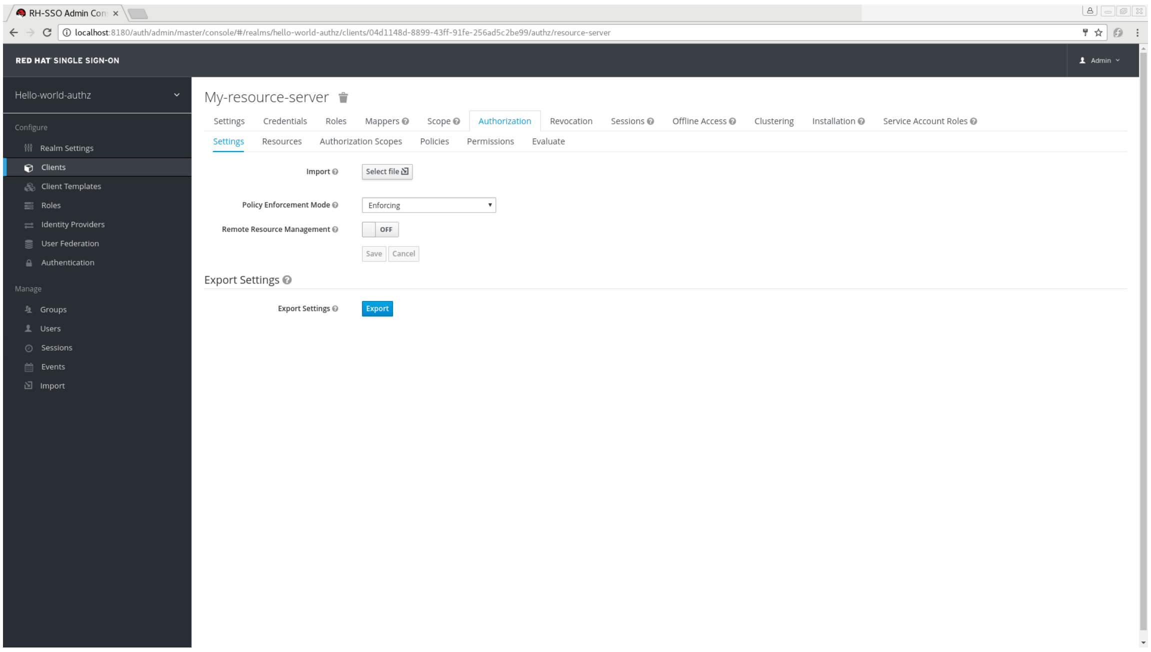
- Protected resources and scopes
- Policies
- Permissions

3.4.1. Exporting a Configuration File

To export a configuration file, complete the following steps:

1. Navigate to the **Resource Server Settings** page.

Resource Server Settings



localhost:8180/auth/admin/master/console/#/realms/hello-world-authz/clients/04d1148d-8899-43ff-91fe-256ad5c2be99/authz/resource-server

My-resource-server

Import: Select file

Policy Enforcement Mode: Enforcing

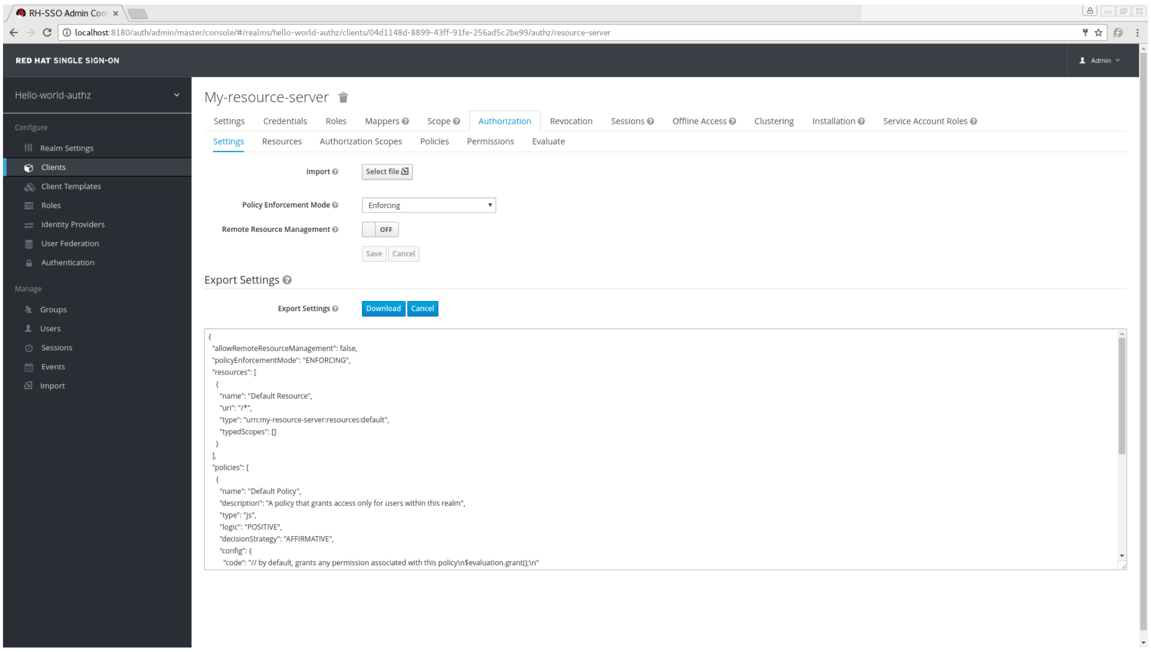
Remote Resource Management: OFF

Export Settings

Export Settings: Export

2. On this page, in the Export Settings section, click **Export**.

Export Settings



Import: Select file

Policy Enforcement Mode: Enforcing

Remote Resource Management: OFF

Export Settings

```
{
 "allowRemoteResourceManagement": false,
 "policyEnforcementMode": "ENFORCING",
 "resources": [
  {
   "name": "Default Resource",
   "uri": "/*",
   "type": "urn:my-resource-server:resources:default",
   "typedScopes": []
  }
 ],
 "policies": [
  {
   "name": "Default Policy",
   "description": "A policy that grants access only for users within this realm",
   "type": "js",
   "logic": "POSITIVE",
   "decisionStrategy": "AFFIRMATIVE",
   "config": {
    "code": "// by default, grants any permission associated with this policy\n$evaluation.grant();\n"
```

The configuration file is exported in JSON format and displayed in a text area, from which you can copy and paste. You can also click **Download** to download the configuration file and save it.

3.4.2. Importing a Configuration File

To import a configuration file for a resource server, click **Select file** to select a file containing the configuration you want to import.

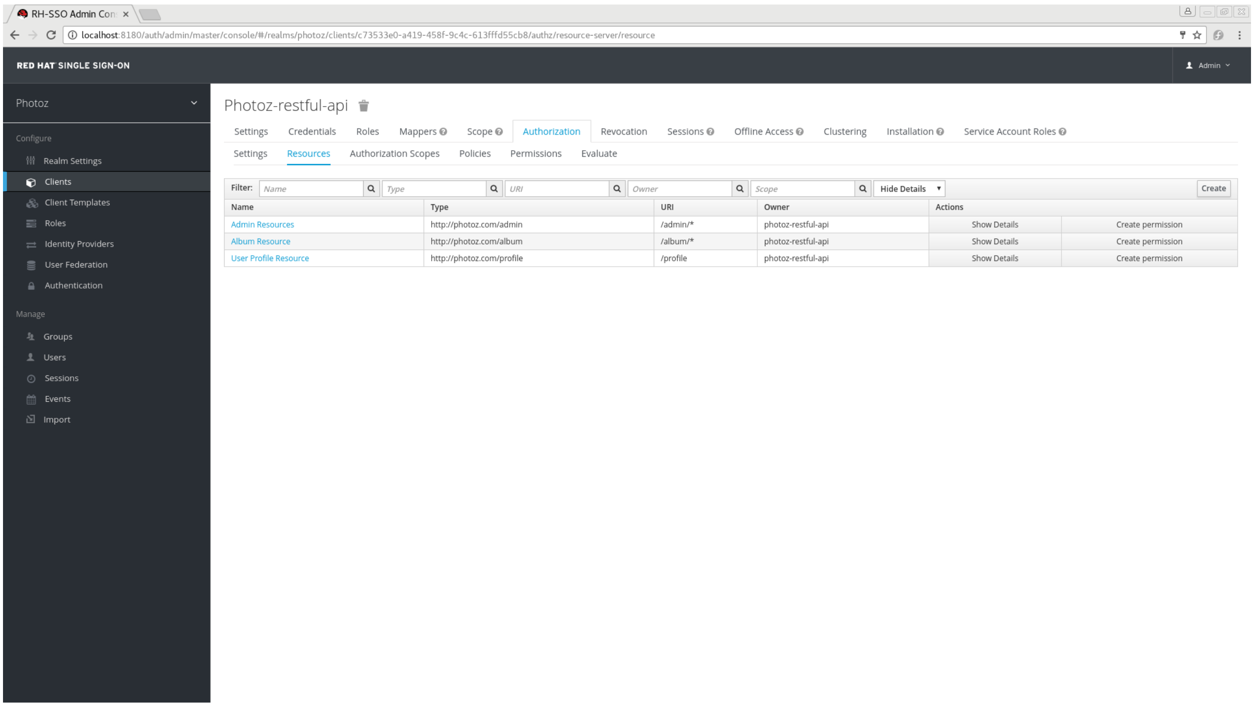
CHAPTER 4. MANAGING RESOURCES AND SCOPES

Resource management is straightforward and generic. After creating a resource server, you can start creating the resources and scopes that you want to protect. Resources and scopes can be managed by navigating to the **Resource** and **Scope** tabs, respectively.

4.1. VIEWING RESOURCES

On the **Resource** page, you see a list of the resources associated with a resource server.

Resources



Name	Type	URI	Owner	Actions	
Admin Resources	http://photoz.com/admin	/admin/*	photoz-restful-api	Show Details	Create permission
Album Resource	http://photoz.com/album	/album/*	photoz-restful-api	Show Details	Create permission
User Profile Resource	http://photoz.com/profile	/profile	photoz-restful-api	Show Details	Create permission

The resource list provides information about the protected resources, such as:

- Type
- URI
- Owner
- Associated scopes, if any
- Associated permissions

From this list, you can also directly create a permission by clicking **Create Permission** for the resource for which you want to create the permission.

Note

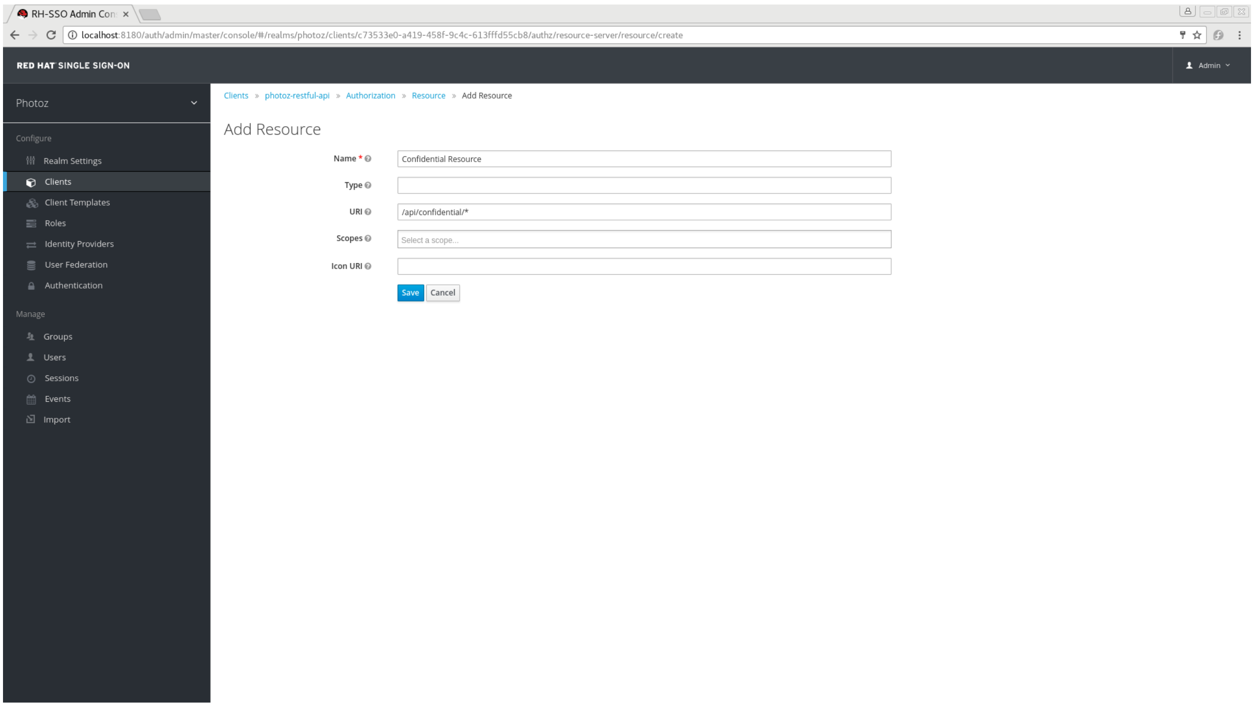
Before creating permissions for your resources, be sure you have already defined the policies that you want to associate with the permission.

4.2. CREATING RESOURCES

Creating a resource is straightforward and generic. Your main concern is the granularity of the resources you create. In other words, resources can be created to represent a set of one or more resources and the way you define them is crucial to managing permissions.

To create a new resource, click **Create** in the right upper corner of the resource listing.

Add Resource



localhost:8180/auth/admin/master/console/#/realms/photoz/clients/c73533e0-a419-458f-9c4c-613fffd55cb8/authz/resource-server/resource/create

Add Resource

Name *: Confidential Resource

Type:

URI: /api/confidential/*

Scopes: Select a scope...

Icon URI:

In Red Hat Single Sign-On, a resource defines a small set of information that is common to different types of resources, such as:

- **Name**

  A human-readable and unique string describing this resource.

- **Type**

  A string uniquely identifying the type of a set of one or more resources. The type is a *string* used to group different resource instances. For example, the default type for the default resource that is automatically created is **`urn:resource-server-name:resources:default`**

- **URI**

  A URI that provides the location/address for the resource. For HTTP resources, the URI is usually the relative path used to serve these resources.

- **Scopes**

  One or more scopes to associate with the resource.

4.2.1. Typed Resources

The type field of a resource can be used to group different resources together, so they can be protected using a common set of permissions.

4.2.2. Resource Owners

Resources also have an owner. By default, resources are owned by the resource server.

However, resources can also be associated with users, so you can create permissions based on the resource owner. For example, only the resource owner is allowed to delete or update a given resource.

4.2.3. Managing Resources Remotely

Resource management is also exposed through the Protection API to allow resource servers to remotely manage their resources.

When using the Protection API, resource servers can be implemented to manage resources owned by their users. In this case, you can specify the user identifier to configure a resource as belonging to a specific user.

Note

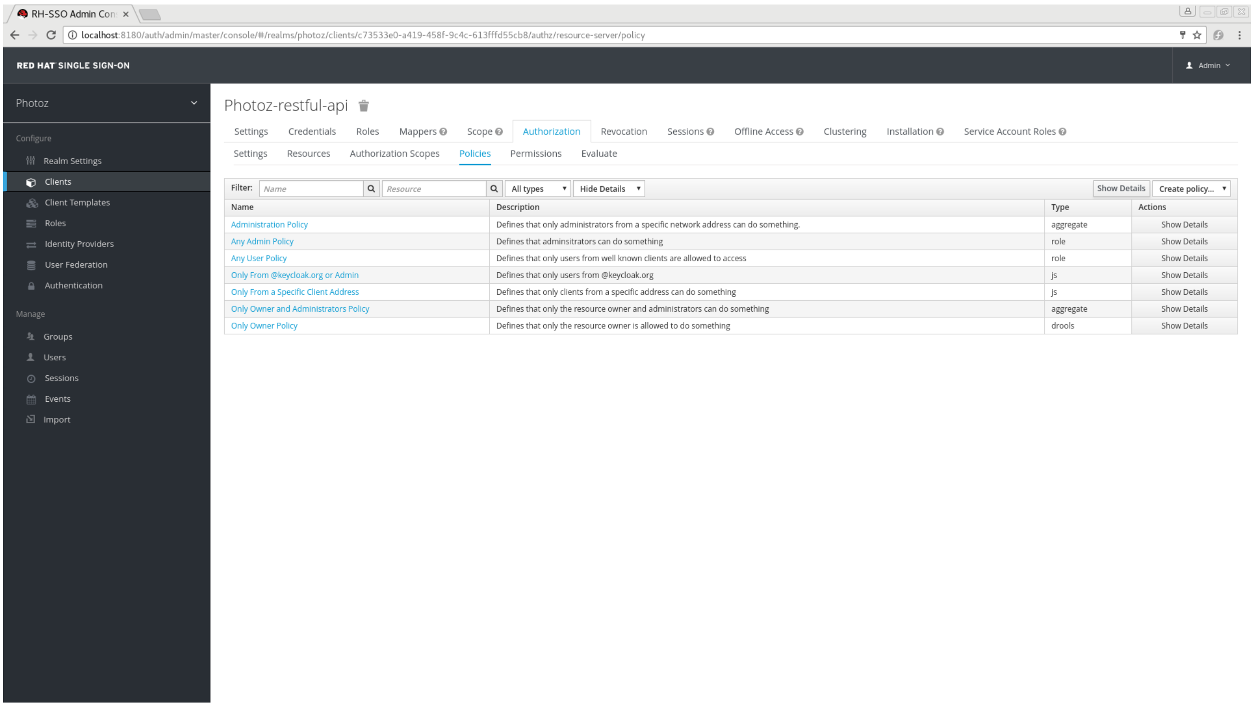
Red Hat Single Sign-On provides resource servers complete control over their resources. In the future, we should be able to allow users to control their own resources as well as approve authorization requests and manage permissions, especially when using the UMA protocol.

CHAPTER 5. MANAGING POLICIES

As mentioned previously, policies define the conditions that must be satisfied before granting access to an object.

You can view all policies associated with a resource server by clicking the **Policy** tab when editing a resource server.

Policies



Name	Description	Type	Actions
Administration Policy	Defines that only administrators from a specific network address can do something.	aggregate	Show Details
Any Admin Policy	Defines that adminsitrators can do something	role	Show Details
Any User Policy	Defines that only users from well known clients are allowed to access	role	Show Details
Only From @keycloak.org or Admin	Defines that only users from @keycloak.org	js	Show Details
Only From a Specific Client Address	Defines that only clients from a specific address can do something	js	Show Details
Only Owner and Administrators Policy	Defines that only the resource owner and administrators can do something	aggregate	Show Details
Only Owner Policy	Defines that only the resource owner is allowed to do something	drools	Show Details

On this tab, you can view the list of previously created policies as well as create and edit a policy.

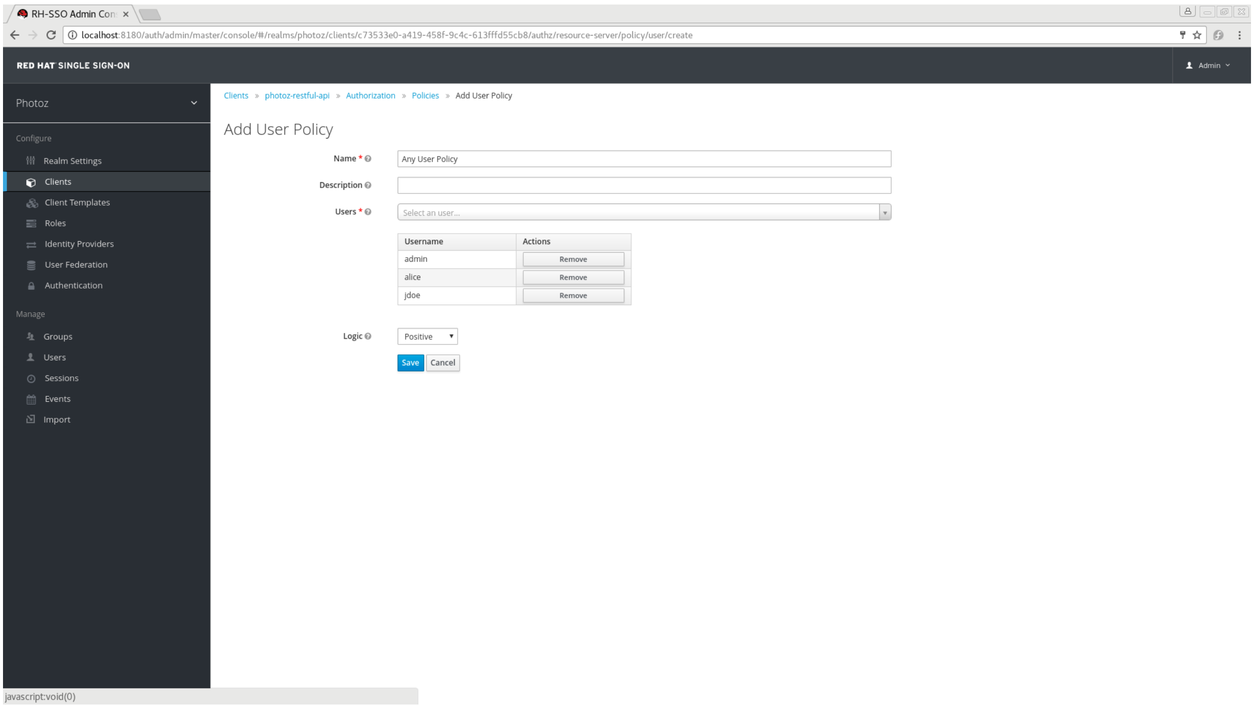
To create a new policy, in the upper right corner of the policy list, select a policy type from the `Create policy` dropdown list. Details about each policy type are described in this section.

5.1. USER-BASED POLICY

You can use this type of policy to define conditions for your permissions where a set of one or more users is permitted to access an object.

To create a new user-based policy, select **User-Based** in the dropdown list in the upper right corner of the permission listing.

Add a User-Based Policy


RH-SSO Admin Con
localhost:8180/auth/admin/master/console/#/realms/photoz/clients/c73533e0-a419-458f-9c4c-613fffd55cb8/authz/resource-server/policy/user/create
RED HAT SINGLE SIGN-ON
Admin
Photoz
Configure
Realm Settings
Clients
Client Templates
Roles
Identity Providers
User Federation
Authentication
Manage
Groups
Users
Sessions
Events
Import
Clients » photoz-restful-api » Authorization » Policies » Add User Policy
Save
Cancel
javascript:void(0)

Add User Policy

Name * Any User Policy

Description

Users * Select an user...

Username	Actions
admin	Remove
alice	Remove
jdoe	Remove

Logic Positive

5.1.1. Configuration

- **Name**

  A human-readable and unique string identifying the policy. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

  A string containing details about this policy.

- **Users**

  Specifies which users are given access by this policy.

- **Logic**

  The Logic of this policy to apply after the other conditions have been evaluated.

5.2. ROLE-BASED POLICY

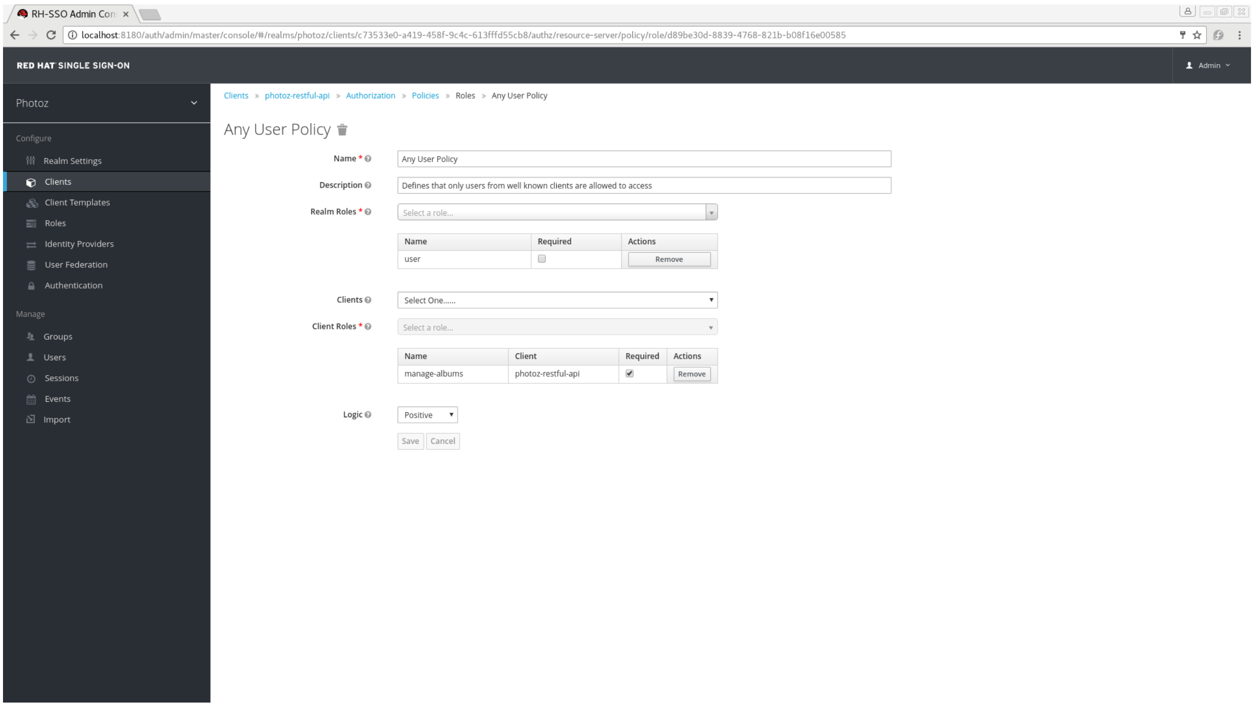
You can use this type of policy to define conditions for your permissions where a set of one or more roles is permitted to access an object.

By default, roles added to this policy are not specified as required and the policy will grant access if the user requesting access has been granted any of these roles. However, you can specify a specific role as required if you want to enforce a specific role. You can also combine required and non-required roles, regardless of whether they are realm or client roles.

Role policies can be useful when you need more restricted role-based access control (RBAC), where specific roles must be enforced to grant access to an object. For instance, you can enforce that a user must consent to allowing a client application (which is acting on the user’s behalf) to access the user’s resources. You can use Red Hat Single Sign-On Client Scope Mapping to enable consent pages or even enforce clients to explicitly provide a scope when obtaining access tokens from a Red Hat Single Sign-On server.

To create a new role-based policy, select **Role-Based** in the dropdown list in the upper right corner of the permission listing.

Add Role-Based Policy



RH-SSO Admin Con
localhost:8180/auth/admin/master/console/#/realms/photoz/clients/c73533e0-a419-458f-9c4c-613fffd55cb8/authz/resource-server/policy/role/d89be30d-8839-4768-821b-b08f16e00585
RED HAT SINGLE SIGN-ON
Admin
Photoz
Configure
Realm Settings
Clients
Client Templates
Roles
Identity Providers
User Federation
Authentication
Manage
Groups
Users
Sessions
Events
Import
Clients » photoz-restful-api » Authorization » Policies » Roles » Any User Policy
Save
Cancel

Any User Policy

Name * : Any User Policy

Description : Defines that only users from well known clients are allowed to access

Realm Roles * : Select a role...

Name	Required	Actions
user	□	Remove

Clients : Select One......

Client Roles * : Select a role...

Name	Client	Required	Actions
manage-albums	photoz-restful-api	☑	Remove

Logic : Positive

5.2.1. Configuration

- **Name**

  A human-readable and unique string describing the policy. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

  A string containing details about this policy.

- **Realm Roles**

  Specifies which **realm** roles are permitted by this policy.

- **Client Roles**

  Specifies which **client** roles are permitted by this policy. To enable this field must first select a **`Client`**.

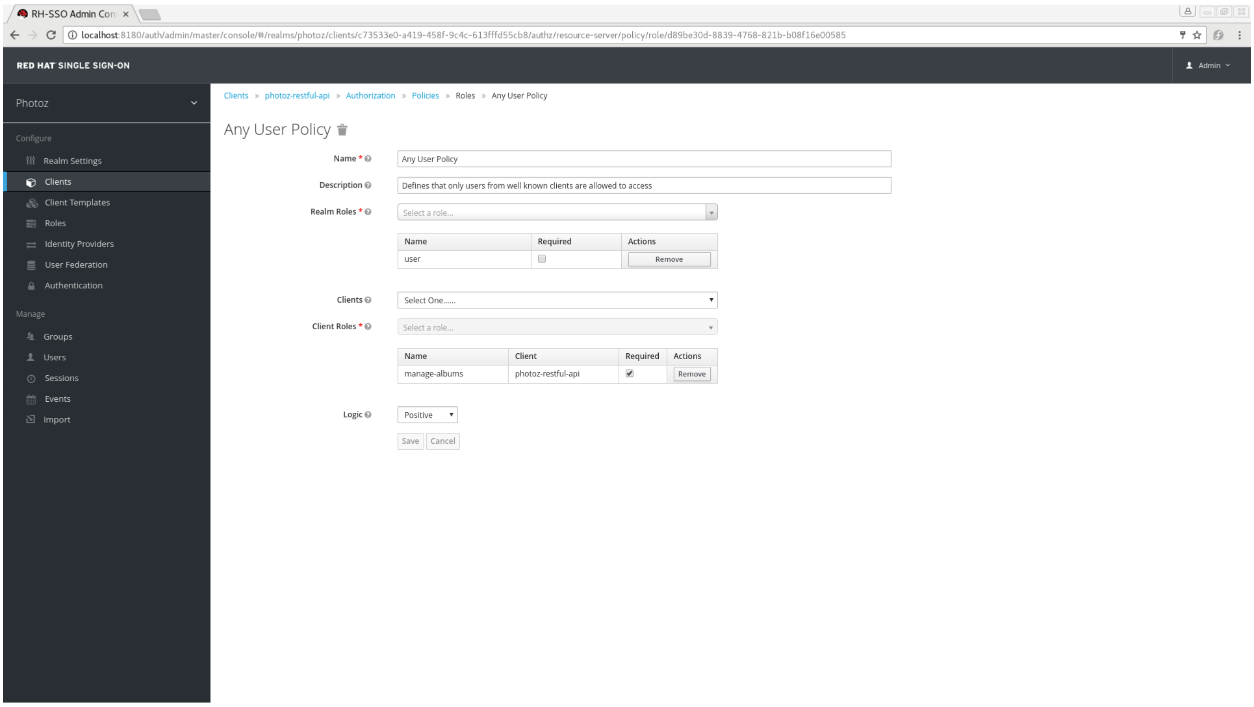
- **Logic**

  The Logic of this policy to apply after the other conditions have been evaluated.

5.3. DEFINING A ROLE AS REQUIRED

When creating a role-based policy, you can specify a specific role as **`Required`**. When you do that, the policy will grant access only if the user requesting access has been granted **all** the **required** roles. Both realm and client roles can be configured as such.

Example of Required Role



localhost:8180/auth/admin/master/console/#/realms/photoz/clients/c73533e0-a419-458f-9c4c-613fffd55cb8/authz/resource-server/policy/role/d89be30d-8839-4768-821b-b08f16e00585

Clients » photoz-restful-api » Authorization » Policies » Roles » Any User Policy

Any User Policy

Name *: Any User Policy

Description: Defines that only users from well known clients are allowed to access

Realm Roles *: Select a role...

Name	Required	Actions
user	□	Remove

Clients: Select One......

Client Roles *: Select a role...

Name	Client	Required	Actions
manage-albums	photoz-restful-api	☑	Remove

Logic: Positive

To specify a role as required, select the **`Required`** checkbox for the role you want to configure as required.

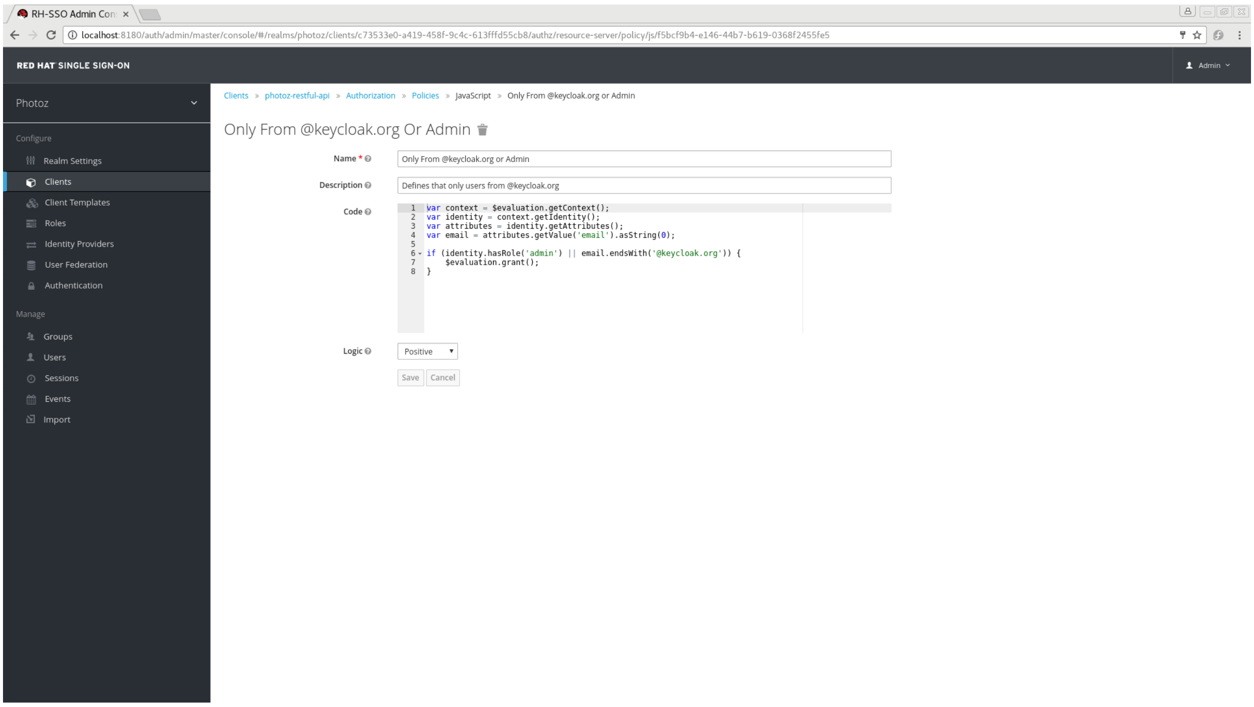
Required roles can be useful when your policy defines multiple roles but only a subset of them are mandatory. In this case, you can combine realm and client roles to enable an even more fine-grained role-based access control (RBAC) model for your application. For example, you can have policies specific for a client and require a specific client role associated with that client. Or you can enforce that access is granted only in the presence of a specific realm role. You can also combine both approaches within the same policy.

5.4. JAVASCRIPT-BASED POLICY

You can use this type of policy to define conditions for your permissions using JavaScript. It is one of the rule-based policy types supported by Red Hat Single Sign-On, and provides flexibility to write any policy based on the Evaluation API.

To create a new JavaScript-based policy, select **JavaScript** in the dropdown list in the upper right corner of the permission listing.

Add JavaScript Policy


RH-SSO Admin Con
localhost:8180/auth/admin/master/console/#/realms/photoz/clients/c73533e0-a419-458f-9c4c-613fffd55cb8/authz/resource-server/policy/js/f5bcf9b4-e146-44b7-b619-0368f2455fe5
RED HAT SINGLE SIGN-ON
Admin
Photoz
Configure
Realm Settings
Clients
Client Templates
Roles
Identity Providers
User Federation
Authentication
Manage
Groups
Users
Sessions
Events
Import
Clients » photoz-restful-api » Authorization » Policies » JavaScript » Only From @keycloak.org or Admin
Save
Cancel

Only From @keycloak.org Or Admin

Name * : Only From @keycloak.org or Admin

Description : Defines that only users from @keycloak.org

Code :

```
var context = $evaluation.getContext();
var identity = context.getIdentity();
var attributes = identity.getAttributes();
var email = attributes.getValue('email').asString(0);

if (identity.hasRole('admin') || email.endsWith('@keycloak.org')) {
    $evaluation.grant();
}
```

Logic : Positive

5.4.1. Configuration

- **Name**

  A human-readable and unique string describing the policy. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

  A string containing details about this policy.

- **Code**

  The JavaScript code providing the conditions for this policy.

- **Logic**

  The Logic of this policy to apply after the other conditions have been evaluated.

5.4.2. Examples

Here is a simple example of a JavaScript-based policy that uses attribute-based access control (ABAC) to define a condition based on an attribute obtained from the execution context:

```
var context = $evaluation.getContext();
var contextAttributes = context.getAttributes();

if (contextAttributes.containsValue('kc.client.network.ip_address',
'127.0.0.1')) {
    $evaluation.grant();
}
```

You can also use role-based access control (RBAC):

```
var identity = $evaluation.getIdentity();

if (identity.hasRole('keycloak_user')) {
    $evaluation.grant();
}
```

Or a combination of several access control mechanisms:

```
var context = $evaluation.getContext();
var identity = context.getIdentity();
var attributes = identity.getAttributes();
var email = attributes.getValue('email').asString(0);

if (identity.hasRole('admin') || email.endsWith('@keycloak.org')) {
    $evaluation.grant();
}
```

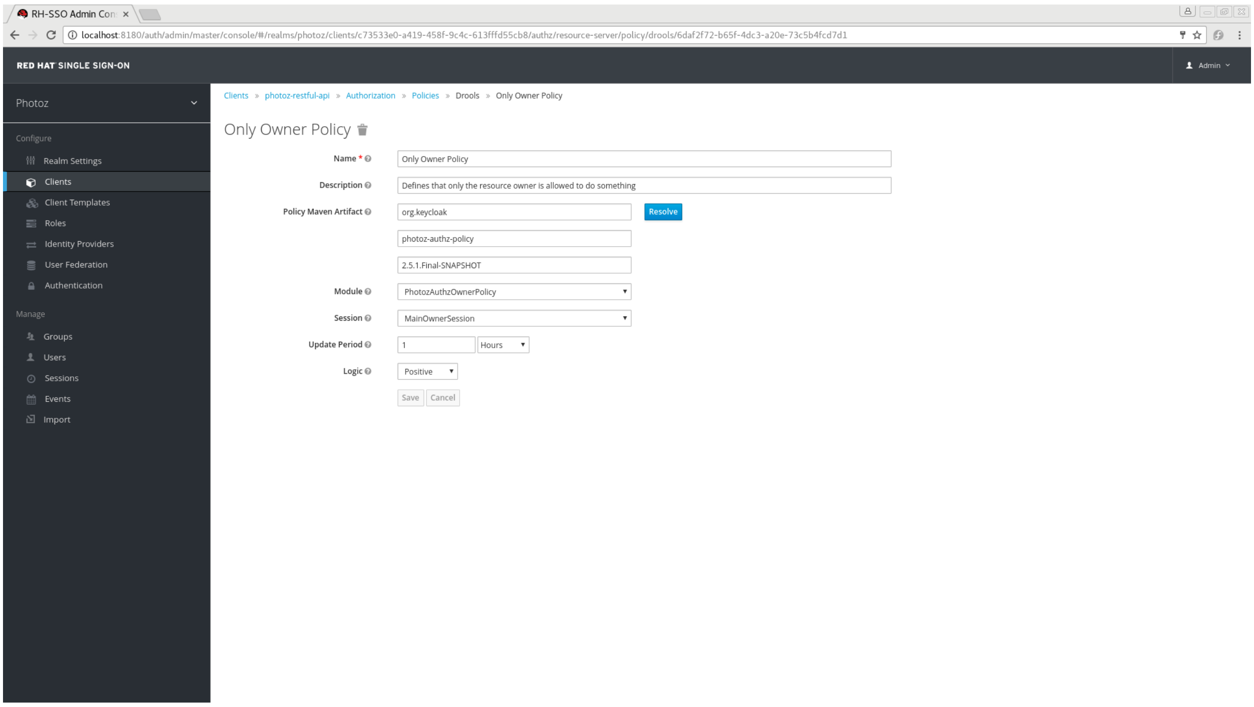
When writing your own rules, keep in mind that the **$evaluation** object is an object implementing **org.keycloak.authorization.policy.evaluation.Evaluation**. For more information about what you can access from this interface, see the Evaluation API.

5.5. RULE-BASED POLICY

With this type of policy you can define conditions for your permissions using Drools, which is a rule evaluation environment. It is one of the *Rule-Based* policy types supported by Red Hat Single Sign-On, and provides flexibility to write any policy based on the Evaluation API.

To create a new Rule-based policy, in the dropdown list in the right upper corner of the permission listing, select **Rule**.

Add Rule Policy



localhost:8180/auth/admin/master/console/#/realms/photoz/clients/c73533e0-a419-458f-9c4c-613fffd55cb8/authz/resource-server/policy/drools/6daf2f72-b65f-4dc3-a20e-73c5b4fcd7d1

Clients » photoz-restful-api » Authorization » Policies » Drools » Only Owner Policy

Only Owner Policy

Name *: Only Owner Policy

Description: Defines that only the resource owner is allowed to do something

Policy Maven Artifact: org.keycloak / photoz-authz-policy / 2.5.1.Final-SNAPSHOT

Module: PhotozAuthzOwnerPolicy

Session: MainOwnerSession

Update Period: 1 Hours

Logic: Positive

5.5.1. Configuration

- **Name**

  A human-readable and unique string describing the policy. We strongly suggest that you use names that are closely related with your business and security requirements, so you can identify them more easily and also know what they actually mean.

- **Description**

  A string with more details about this policy.

- **Policy Maven Artifact**

  A Maven groupId-artifactId-version (GAV) pointing to an artifact where the rules are defined. Once you have provided the GAV, you can click **Resolve** to load both **Module** and **Session** fields.

  - Group Id

    The groupId of the artifact.

  - Artifact Id

    The artifactId of the artifact.

  - Version

    The version of the artifact.

- **Module**

  The module used by this policy. You must provide a module to select a specific session from which rules will be loaded.

- **Session**

  The session used by this policy. The session provides all the rules to evaluate when processing the policy.

- **Update Period**

  Specifies an interval for scanning for artifact updates.

- **Logic**

  The Logic of this policy to apply after the other conditions have been evaluated.

5.5.2. Examples

Here is a simple example of a Drools-based policy that uses attribute-based access control (ABAC) to define a condition that evaluates to a GRANT only if the authenticated user is the owner of the requested resource:

```
import org.keycloak.authorization.policy.evaluation.Evaluation;
rule "Authorize Resource Owner"
    dialect "mvel"
    when
       $evaluation : Evaluation(
           $identity: context.identity,
           $permission: permission,
           $permission.resource != null &&
$permission.resource.owner.equals($identity.id)
       )
    then
        $evaluation.grant();
end
```

You can even use another variant of ABAC to obtain attributes from the identity and define a condition accordingly:

```
import org.keycloak.authorization.policy.evaluation.Evaluation;
rule "Authorize Using Identity Information"
    dialect "mvel"
    when
       $evaluation : Evaluation(
           $identity: context.identity,
           identity.attributes.containsValue("someAttribute",
"you_can_access")
       )
    then
        $evaluation.grant();
end
```

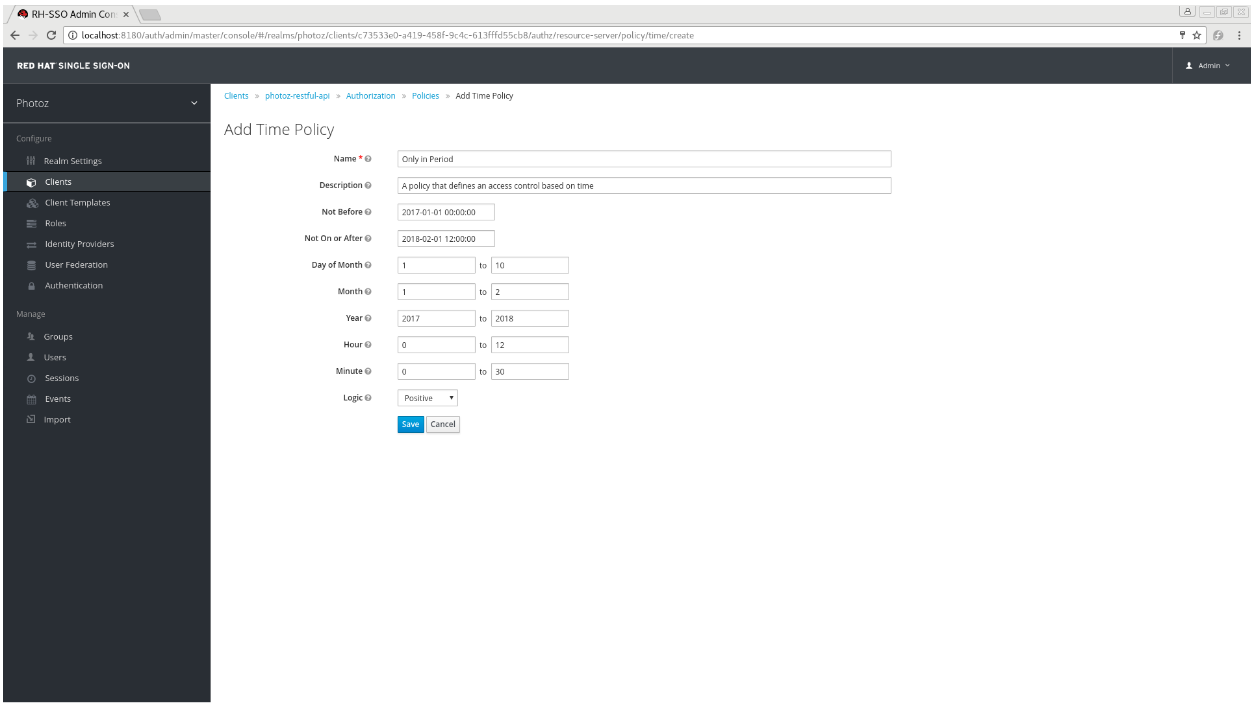
For more information about what you can access from the **`org.keycloak.authorization.policy.evaluation.Evaluation`** interface, see Evaluation API.

5.6. TIME-BASED POLICY

You can use this type of policy to define time conditions for your permissions.

To create a new time-based policy, select **Time** in the dropdown list in the upper right corner of the permission listing.

Add Time Policy



localhost:8180/auth/admin/master/console/#/realms/photoz/clients/c73533e0-a419-458f-9c4c-613fffd55cb8/authz/resource-server/policy/time/create

Clients » photoz-restful-api » Authorization » Policies » Add Time Policy

Add Time Policy

Name *	Only in Period		
Description	A policy that defines an access control based on time		
Not Before	2017-01-01 00:00:00		
Not On or After	2018-02-01 12:00:00		
Day of Month	1	to	10
Month	1	to	2
Year	2017	to	2018
Hour	0	to	12
Minute	0	to	30
Logic	Positive		

5.6.1. Configuration

- **Name**

  A human-readable and unique string describing the policy. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

  A string containing details about this policy.

- **Not Before**

  Defines the time before which access must **not** be granted. Permission is granted only if the current date/time is later than or equal to this value.

- **Not On or After**

  Defines the time after which access must **not** be granted. Permission is granted only if the current date/time is earlier than or equal to this value.

- **Day of Month**

  Defines the day of month that access must be granted. You can also specify a range of dates. In this case, permission is granted only if the current day of the month is between or equal to the two values specified.

- **Month**

  Defines the month that access must be granted. You can also specify a range of months. In this case, permission is granted only if the current month is between or equal to the two values specified.

- **Year**

  Defines the year that access must be granted. You can also specify a range of years. In this case, permission is granted only if the current year is between or equal to the two values specified.

- **Hour**

  Defines the hour that access must be granted. You can also specify a range of hours. In this case, permission is granted only if current hour is between or equal to the two values specified.

- **Minute**

  Defines the minute that access must be granted. You can also specify a range of minutes. In this case, permission is granted only if the current minute is between or equal to the two values specified.

- **Logic**

  The Logic of this policy to apply after the other conditions have been evaluated.

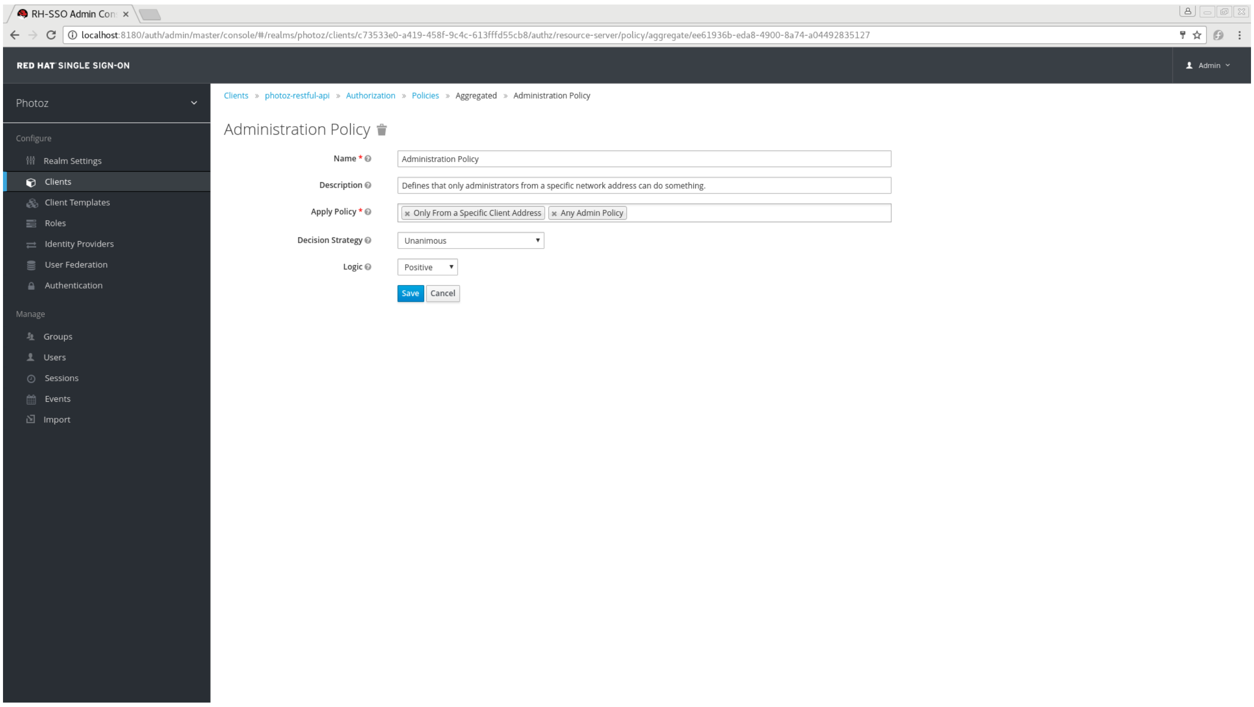
Access is only granted if all conditions are satisfied. Red Hat Single Sign-On will perform an *AND* based on the outcome of each condition.

5.7. AGGREGATED POLICY

As mentioned previously, Red Hat Single Sign-On allows you to build a policy of policies, a concept referred to as policy aggregation. You can use policy aggregation to reuse existing policies to build more complex ones and keep your permissions even more decoupled from the policies that are evaluated during the processing of authorization requests.

To create a new aggregated policy, select **Aggregated** in the dropdown list located in the right upper corner of the permission listing.

Add an Aggregated Policy


RH-SSO Admin Con
localhost:8180/auth/admin/master/console/#/realms/photoz/clients/c73533e0-a419-458f-9c4c-613fffd55cb8/authz/resource-server/policy/aggregate/ee61936b-eda8-4900-8a74-a04492835127
RED HAT SINGLE SIGN-ON
Admin
Photoz
Configure
Realm Settings
Clients
Client Templates
Roles
Identity Providers
User Federation
Authentication
Manage
Groups
Users
Sessions
Events
Import
Clients » photoz-restful-api » Authorization » Policies » Aggregated » Administration Policy
Save
Cancel

Administration Policy

Name * : Administration Policy

Description : Defines that only administrators from a specific network address can do something.

Apply Policy * : Only From a Specific Client Address, Any Admin Policy

Decision Strategy : Unanimous

Logic : Positive

Let’s suppose you have a resource called *Confidential Resource* that can be accessed only by users from the *keycloak.org* domain and from a certain range of IP addresses. You can create a single policy with both conditions. However, you want to reuse the domain part of this policy to apply to permissions that operates regardless of the originating network.

You can create separate policies for both domain and network conditions and create a third policy based on the combination of these two policies. With an aggregated policy, you can freely combine other policies and then apply the new aggregated policy to any permission you want.

Note

When creating aggregated policies, be mindful that you are not introducing a circular reference or dependency between policies. If a circular dependency is detected, you cannot create or update the policy.

5.7.1. Configuration

- **Name**

  A human-readable and unique string describing the policy. We strongly suggest that you use names that are closely related with your business and security requirements, so you can identify them more easily and also know what they mean.

- **Description**

A string with more details about this policy.

- **Apply Policy**

  Defines a set of one or more policies to associate with a policy.

- **Decision Strategy**

  The decision strategy for this permission.

- **Logic**

  The Logic of this policy to apply after the other conditions have been evaluated.

5.7.2. Decision Strategy for Aggregated Policies

When creating aggregated policies, you can also define the decision strategy that will be used to determine the final decision based on the outcome from each policy.

- **Unanimous**

  The default strategy if none is provided. In this case, *all* policies must evaluate to a positive decision for the final decision to be also positive.

- **Affirmative**

  In this case, *at least one* policy must evaluate to a positive decision in order for the final decision to be also positive.

- **Consensus**

  In this case, the number of positive decisions must be greater than the number of negative decisions. If the number of positive and negative decisions is the same, the final decision will be negative.

5.8. POSITIVE AND NEGATIVE LOGIC

Policies can be configured with positive or negative logic. Briefly, you can use this option to define whether the policy result should be kept as it is or be negated.

For example, suppose you want to create a policy where only users **not** granted with a specific role should be given access. In this case, you can create a role-based policy using that role and set its **Logic** field to **Negative**. If you keep **Positive**, which is the default behavior, the policy result will be kept as it is.

5.9. POLICY EVALUATION API

When writing rule-based policies using JavaScript or JBoss Drools, Red Hat Single Sign-On provides an Evaluation API that provides useful information to help determine whether a permission should be granted.

This API consists of a few interfaces that provides you access to information such as:

- The permission being requested
- The identity that is requesting the permission, from which you can obtain claims/attributes

- Runtime environment and any other attribute associated with the execution context

The main interface is **org.keycloak.authorization.policy.evaluation.Evaluation**, which defines the following contract:

```
public interface Evaluation {

    /**
     * Returns the {@link ResourcePermission} to be evaluated.
     *
     * @return the permission to be evaluated
     */
    ResourcePermission getPermission();

    /**
     * Returns the {@link EvaluationContext}. Which provides access to
the whole evaluation runtime context.
     *
     * @return the evaluation context
     */
    EvaluationContext getContext();

    /**
     * Grants the requested permission to the caller.
     */
    void grant();

    /**
     * Denies the requested permission.
     */
    void deny();
}
```

When processing an authorization request, Red Hat Single Sign-On creates an **`Evaluation`** instance before evaluating any policy. This instance is then passed to each policy to determine whether access is **GRANT** or **DENY**.

Policies determine this by invoking the **`grant()`** or **`deny()`** methods on an **`Evaluation`** instance. By default, the state of the **`Evaluation`** instance is denied, which means that your policies must explicitly invoke the **`grant()`** method to indicate to the policy evaluation engine that permission should be granted.

For more information about the Evaluation API see the JavaDocs.

5.9.1. The Evaluation Context

The evaluation context provides useful information to policies during their evaluation.

```
public interface EvaluationContext {

    /**
     * Returns the {@link Identity} that represents an entity (person or
non-person) to which the permissions must be granted, or not.
     *
     * @return the identity to which the permissions must be granted, or
not
```

```
     */
    Identity getIdentity();

    /**
     * Returns all attributes within the current execution and runtime
environment.
     *
     * @return the attributes within the current execution and runtime
environment
     */
    Attributes getAttributes();
}
```

From this interface, policies can obtain:

- The authenticated **`Identity`**
- Information about the execution context and runtime environment

The **`Identity`** is built based on the OAuth2 Access Token that was sent along with the authorization request, and this construct has access to all claims extracted from the original token. For example, if you are using a *Protocol Mapper* to include a custom claim in a OAuth2 Access Token you can also access this claim from a policy and use it to build your conditions.

The **`EvaluationContext`** also gives you access to attributes related to both the execution and runtime environments. For now, there only a few built-in attributes.

Table 5.1. Execution and Runtime Attributes

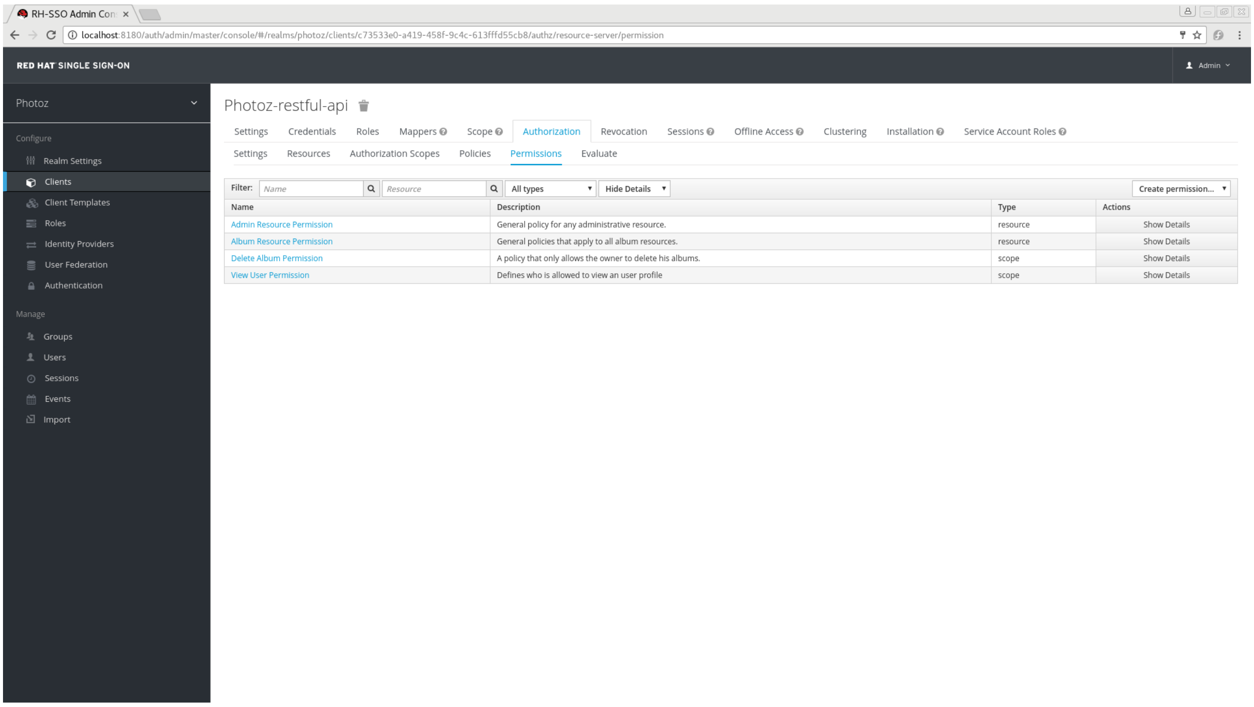
Name	Description	Type
kc.time.date_time	Current date and time	String. Format **`MM/dd/yyyy hh:mm:ss`**
kc.client.network.ip_address	IPv4 address of the client	String
kc.client.network.host	Client’s host name	String
kc.client.id	The client id	String
kc.client.user_agent	The value of the 'User-Agent' HTTP header	String[]
kc.realm.name	The name of the realm	String

CHAPTER 6. MANAGING PERMISSIONS

A permission associates the object being protected and the policies that must be evaluated to decide whether access should be granted.

After creating the resources you want to protect and the policies you want to use to protect these resources, you can start managing permissions. To manage permissions, click the **Permissions** tab when editing a resource server.

Permissions



Name	Description	Type	Actions
Admin Resource Permission	General policy for any administrative resource.	resource	Show Details
Album Resource Permission	General policies that apply to all album resources.	resource	Show Details
Delete Album Permission	A policy that only allows the owner to delete his albums.	scope	Show Details
View User Permission	Defines who is allowed to view an user profile	scope	Show Details

Permissions can be created to protect two main types of objects:

- **Resources**
- **Scopes**

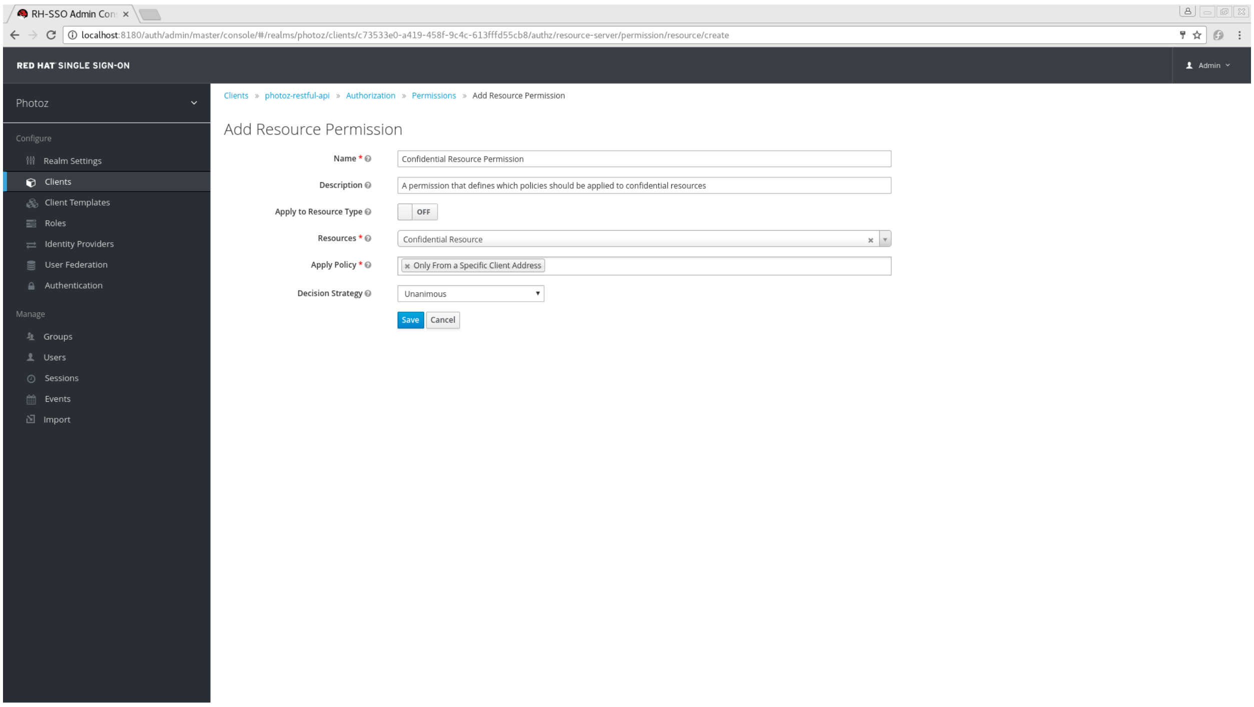
To create a permission, select the permission type you want to create from the dropdown list in the upper right corner of the permission listing. The following sections describe these two types of objects in more detail.

6.1. CREATING RESOURCE-BASED PERMISSIONS

A resource-based permission defines a set of one or more resources to protect using a set of one or more authorization policies.

To create a new resource-based permission, select **Resource-based** in the dropdown list in the upper right corner of the permission listing.

Add Resource-Based Permission



RH-SSO Admin Con
localhost:8180/auth/admin/master/console/#/realms/photoz/clients/c73533e0-a419-458f-9c4c-613fffd55cb8/authz/resource-server/permission/resource/create
RED HAT SINGLE SIGN-ON
Admin
Photoz
Configure
Realm Settings
Clients
Client Templates
Roles
Identity Providers
User Federation
Authentication
Manage
Groups
Users
Sessions
Events
Import
Clients » photoz-restful-api » Authorization » Permissions » Add Resource Permission
Save
Cancel

Add Resource Permission

Name * Confidential Resource Permission

Description A permission that defines which policies should be applied to confidential resources

Apply to Resource Type OFF

Resources * Confidential Resource

Apply Policy * Only From a Specific Client Address

Decision Strategy Unanimous

6.1.1. Configuration

- **Name**

  A human-readable and unique string describing the permission. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

  A string containing details about this permission.

- **Apply To Resource Type**

  Specifies if the permission is applied to all resources with a given type. When selecting this field, you are prompted to enter the resource type to protect.

  - Resource Type

    Defines the resource type to protect. When defined, this permission is evaluated for all resources matching that type.

- **Resources**

  Defines a set of one or more resources to protect.

- **Apply Policy**

  Defines a set of one or more policies to associate with a permission.

- **Decision Strategy**

  The Decision Strategy for this permission.

6.1.2. Typed Resource Permission

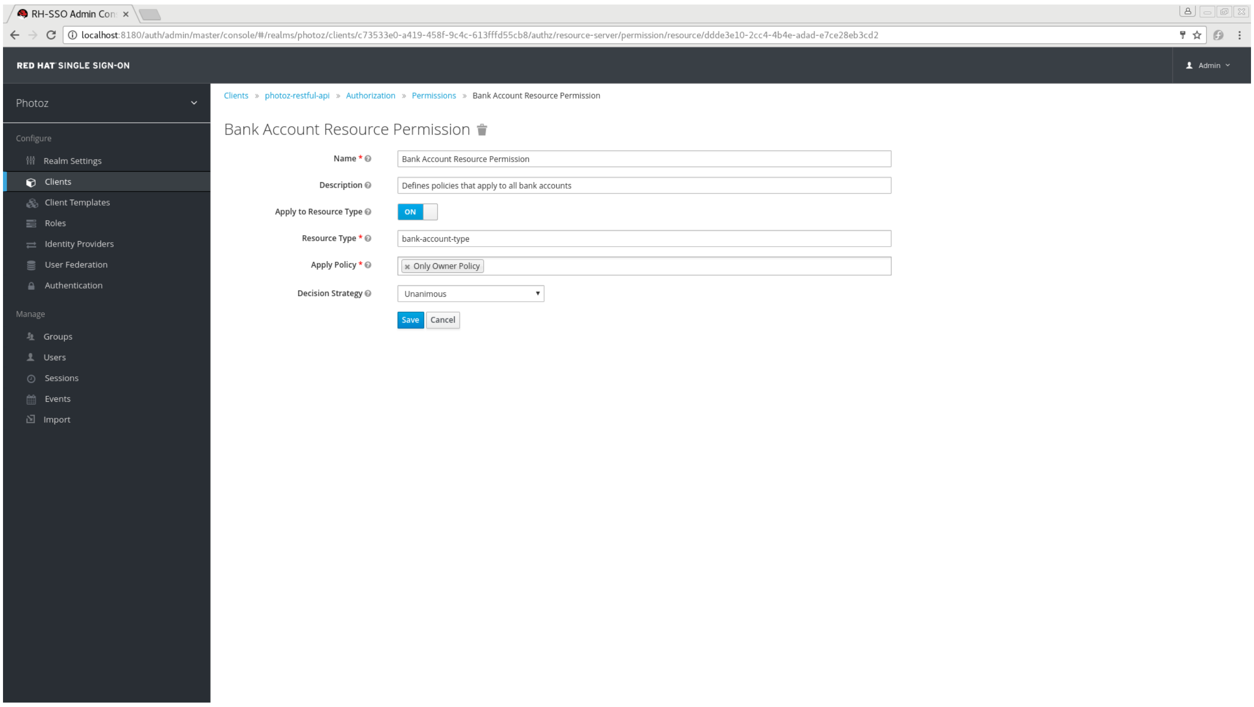
Resource permissions can also be used to define policies that are to be applied to all resources with a given type. This form of resource-based permission can be useful when you have resources sharing common access requirements and constraints.

Frequently, resources within an application can be categorized (or typed) based on the data they encapsulate or the functionality they provide. For example, a financial application can manage different banking accounts where each one belongs to a specific customer. Although they are different banking accounts, they share common security requirements and constraints that are globally defined by the banking organization. With typed resource permissions, you can define common policies to apply to all banking accounts, such as:

- Only the owner can manage his account
- Only allow access from the owner’s country and/or region
- Enforce a specific authentication method

To create a typed resource permission, click Apply to Resource Type when creating a new resource-based permission. With **`Apply to Resource Type`** set to **`On`**, you can specify the type that you want to protect as well as the policies that are to be applied to govern access to all resources with type you have specified.

Example of a Typed Resource Permission



localhost:8180/auth/admin/master/console/#/realms/photoz/clients/c73533e0-a419-458f-9c4c-613fffd55cb8/authz/resource-server/permission/resource/ddde3e10-2cc4-4b4e-adad-e7ce28eb3cd2

Clients » photoz-restful-api » Authorization » Permissions » Bank Account Resource Permission

Bank Account Resource Permission

Name *: Bank Account Resource Permission

Description: Defines policies that apply to all bank accounts

Apply to Resource Type: ON

Resource Type *: bank-account-type

Apply Policy *: Only Owner Policy

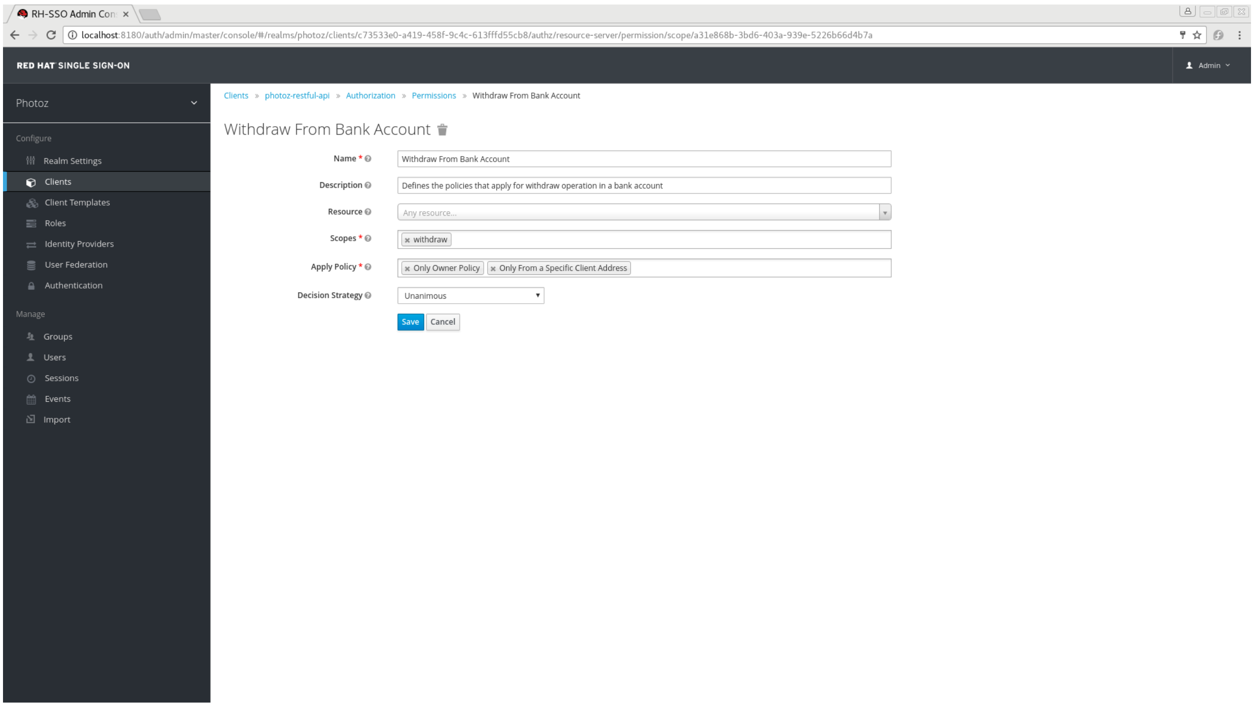
Decision Strategy: Unanimous

6.2. CREATING SCOPE-BASED PERMISSIONS

A scope-based permission defines a set of one or more scopes to protect using a set of one or more authorization policies. Unlike resource-based permissions, you can use this permission type to create permissions not only for a resource, but also for the scopes associated with it, providing more granularity when defining the permissions that govern your resources and the actions that can be performed on them.

To create a new scope-based permission, select **Scope-based** in the dropdown list in the upper right corner of the permission listing.

Add Scope-Based Permission



RH-SSO Admin Con
localhost:8180/auth/admin/master/console/#/realms/photoz/clients/c73533e0-a419-458f-9c4c-613fffd55cb8/authz/resource-server/permission/scope/a31e868b-3bd6-403a-939e-5226b66d4b7a
RED HAT SINGLE SIGN-ON
Admin
Photoz
Configure
Realm Settings
Clients
Client Templates
Roles
Identity Providers
User Federation
Authentication
Manage
Groups
Users
Sessions
Events
Import
Clients » photoz-restful-api » Authorization » Permissions » Withdraw From Bank Account

Withdraw From Bank Account

Name *: Withdraw From Bank Account

Description: Defines the policies that apply for withdraw operation in a bank account

Resource: Any resource...

Scopes *: withdraw

Apply Policy *: Only Owner Policy, Only From a Specific Client Address

Decision Strategy: Unanimous

6.2.1. Configuration

- **Name**

  A human-readable and unique string describing the permission. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

  A string containing details about this permission.

- **Resource**

  Restricts the scopes to those associated with the selected resource. If none is selected, all scopes are available.

- **Scopes**

  Defines a set of one or more scopes to protect.

- **Apply Policy**

  Defines a set of one or more policies to associate with a permission.

- **Decision Strategy**

  The Decision Strategy for this permission.

6.3. POLICY DECISION STRATEGIES

When associating policies with a permission, you can also define a decision strategy to specify how to evaluate the outcome of the associated policies to determine access.

- **Unanimous**

The default strategy if none is provided. In this case, *all* policies must evaluate to a positive decision for the final decision to be also positive.

- **Affirmative**

  In this case, *at least one* policy must evaluate to a positive decision for the final decision to be also positive.

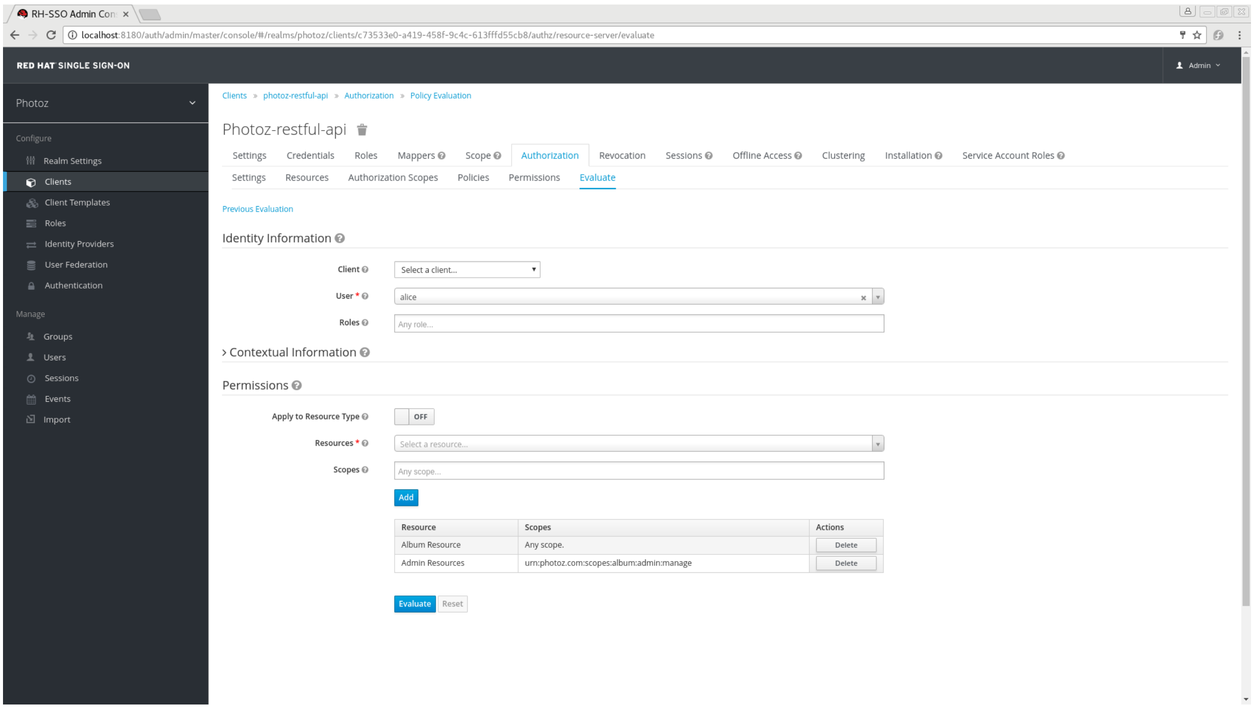
- **Consensus**

  In this case, the number of positive decisions must be greater than the number of negative decisions. If the number of positive and negative decisions is equal, the final decision will be negative.

CHAPTER 7. EVALUATING AND TESTING POLICIES

When designing your policies, you can simulate authorization requests to test how your policies are being evaluated.

You can access the Policy Evaluation Tool by clicking the **`Evaluate`** tab when editing a resource server. There you can specify different inputs to simulate real authorization requests and test the effect of your policies.



Identity Information

Client: Select a client...

User *: alice

Roles: Any role...

› Contextual Information

Permissions

Apply to Resource Type: OFF

Resources *: Select a resource...

Scopes: Any scope...

Resource	Scopes	Actions
Album Resource	Any scope.	Delete
Admin Resources	urn:photoz.com:scopes:album:admin:manage	Delete

7.1. PROVIDING IDENTITY INFORMATION

The **Identity Information** filters can be used to specify the user requesting permissions.

You can also click **Entitlement** to obtain all permissions for the user you selected.

7.2. PROVIDING CONTEXTUAL INFORMATION

The **Contextual Information** filters can be used to define additional attributes to the evaluation context, so that policies can obtain these same attributes.

7.3. PROVIDING THE PERMISSIONS

The **Permissions** filters can be used to build an authorization request. You can request permissions for a set of one or more resources and scopes. If you want to simulate authorization requests based on all protected resources and scopes, click **Add** without specifying any **`Resources`** or **`Scopes`**.

When you’ve specified your desired values, click **Evaluate**.

CHAPTER 8. AUTHORIZATION SERVICES

Red Hat Single Sign-On Authorization Services are based on OAuth2’s User-Managed Access (UMA) Profile.

This section describes the different RESTful endpoints that you can interact with to enable fine-grained authorization for your applications and services.

8.1. PROTECTION API

The Protection API provides a UMA-compliant set of endpoints providing:

- **Resource Registration**

  With this endpoint, resource servers can manage their resources remotely and enable policy enforcers to query the server for the resources that need protection.

- **Permission Registration**

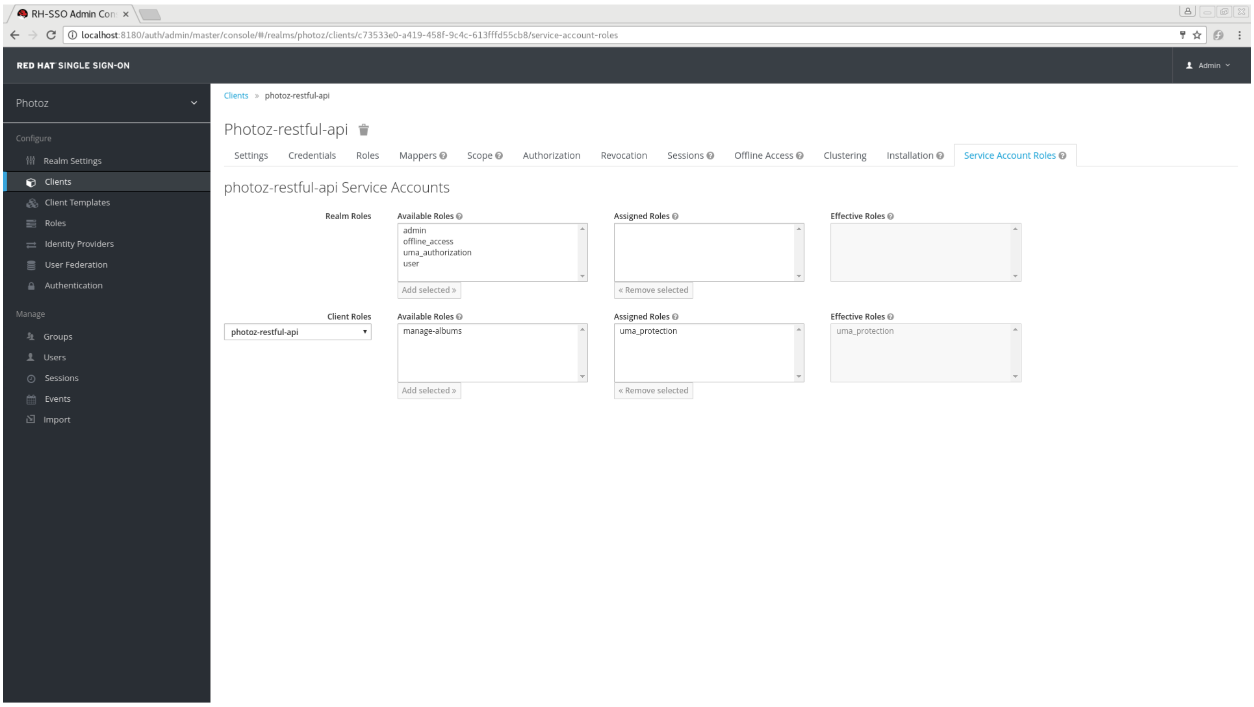
  In the UMA protocol, resource servers access this endpoint, which issues permission tickets.

An important requirement for this API is that *only* resource servers are allowed to access its endpoints using a special OAuth2 access token called a protection API token (PAT). In UMA, a PAT is a token with the scope **uma_protection**.

8.1.1. What is a PAT and How to Obtain It

A **protection API token** (PAT) is a special OAuth2 access token with a scope defined as **uma_protection**. When you create a resource server, Red Hat Single Sign-On automatically creates a role, *uma_protection*, for the corresponding client application and associates it with the client’s service account.

Service Account granted with uma_protection role



localhost:8180/auth/admin/master/console/#/realms/photoz/clients/c73533e0-a419-458f-9c4c-613fffd55cb8/service-account-roles

Clients » photoz-restful-api

Photoz-restful-api

photoz-restful-api Service Accounts

Realm Roles
Available Roles: admin, offline_access, uma_authorization, user
Assigned Roles:
Effective Roles:

Client Roles: photoz-restful-api
Available Roles: manage-albums
Assigned Roles: uma_protection
Effective Roles: uma_protection

Resource servers can obtain a PAT from Red Hat Single Sign-On like any other OAuth2 access token. For example, using curl:

```
curl -X POST \
    -H "Authorization: Basic
aGVsbG8td29ybGQtYXV0aHotc2VydmljZTpwYXNzd29yZA==" \
    -H "Content-Type: application/x-www-form-urlencoded" \
    -d 'grant_type=client_credentials' \
    "http://localhost:8080/auth/realms/${realm_name}/protocol/openid-
connect/token"
```

The example above is using the **client_credentials** grant type to obtain a PAT from the server. As a result, the server returns a response similar to the following:

```
{
  "access_token": ${PAT},
  "expires_in": 300,
  "refresh_expires_in": 1800,
  "refresh_token": ${refresh_token},
  "token_type": "bearer",
  "id_token": ${id_token},
  "not-before-policy": 0,
  "session_state": "ccea4a55-9aec-4024-b11c-44f6f168439e"
}
```

Note

Red Hat Single Sign-On can authenticate your client application in different ways. For simplicity, the **client_credentials** grant type is used here, which requires a *client_id* and a *client_secret*. You can choose to use any supported authentication method.

8.1.2. Managing Resources

Resource servers can manage their resources remotely using a UMA-compliant endpoint.

```
http://${host}:${port}/auth/realms/${realm_name}/authz/protection/resourc
e_set
```

This endpoint provides registration operations outlined as follows (entire path omitted for clarity):

- Create resource set description: POST /resource_set
- Read resource set description: GET /resource_set/{_id}
- Update resource set description: PUT /resource_set/{_id}
- Delete resource set description: DELETE /resource_set/{_id}
- List resource set descriptions: GET /resource_set
- List resource set descriptions using a filter: GET /resource_set?filter=${filter}

For more information about the contract for each of these operations, see UMA Resource Set Registration.

8.1.3. Managing Permission Requests

Resource servers using the UMA protocol can use a specific endpoint to manage permission requests. This endpoint provides a UMA-compliant flow for registering permission requests and obtaining a permission ticket.

```
http://${host}:${port}/auth/realms/${realm_name}/authz/protection/permiss
ion
```

A permission ticket is a special security token type representing a permission request. Per the UMA specification, a permission ticket is:

A correlation handle that is conveyed from an authorization server to a resource server, from a resource server to a client, and ultimately from a client back to an authorization server, to enable the authorization server to assess the correct policies to apply to a request for authorization data.

Note

Permission ticket support is limited. In the full UMA protocol, resource servers can register permission requests in the server to support authorization flows where a resource owner (the user that owns a resource being requested) can approve access to his resources by third parties, among other ways. This represents one of the main features of the UMA specification: resource owners can control their own resources and the policies that govern them. Currently Red Hat Single Sign-On UMA implementation support is very limited in this regard. For example, the system does not store permission tickets on the server and we are essentially using UMA to provide API security and base our authorization offerings. In the future, full support of UMA and other use cases is planned.

In most cases, you won’t need to deal with this endpoint directly. Red Hat Single Sign-On provides a policy enforcer that enables UMA for your resource server so it can obtain a permission ticket from the authorization server, return this ticket to client application, and enforce authorization decisions based on a final requesting party token (RPT).

8.2. AUTHORIZATION API

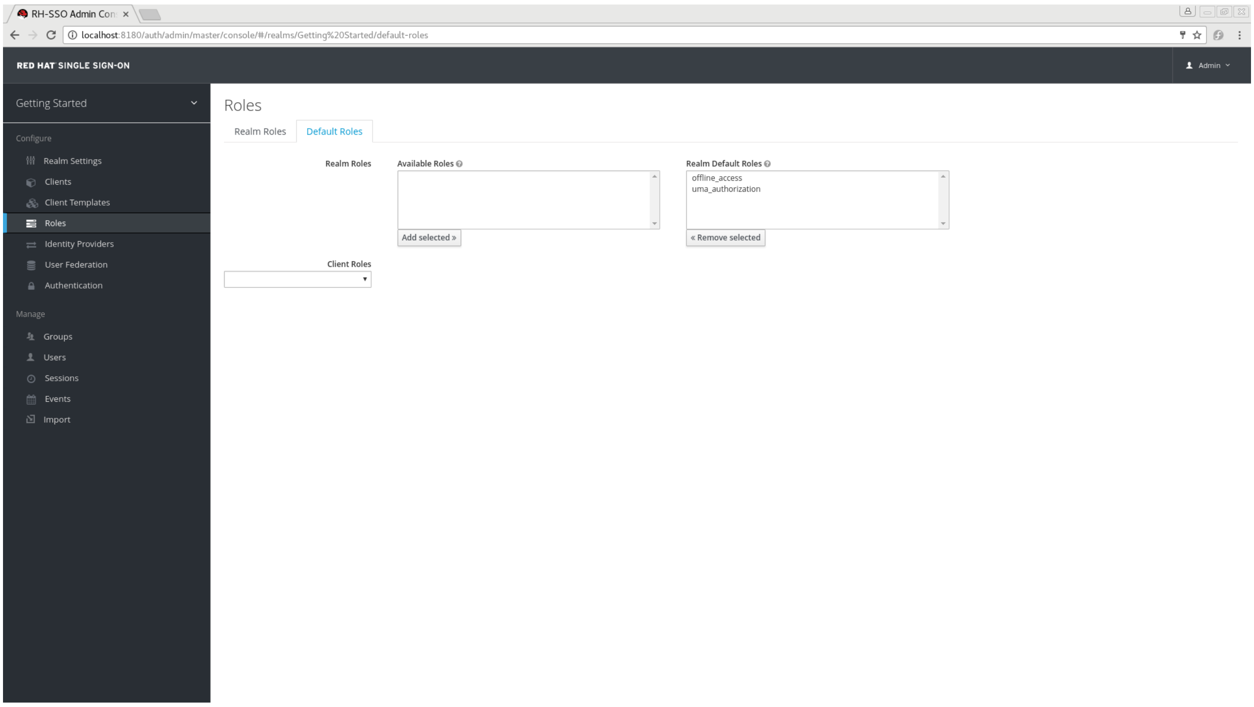
The Authorization API provides a UMA-compliant endpoint for obtaining authorization data from the server, where the authorization data represents the result of the evaluation of all permissions and authorization policies associated with the resources being requested.

Unlike the Protection API, any client application can access the Authorization API endpoint, which requires a special OAuth2 access token called an authorization API token (AAT). In UMA, an AAT is a token with the scope **uma_authorization**.

8.2.1. What is an AAT and How to Obtain It

An authorization API token (AAT) is a special OAuth2 access token with the scope **uma_authorization**. When you create a user, Red Hat Single Sign-On automatically assigns the role *uma_authorization* to the user. The *uma_authorization* role is a default realm role.

Default Role uma_authorization



Realm Roles

Available Roles

Realm Default Roles: offline_access, uma_authorization

Client Roles

An AAT enables a client application to query the server for user permissions.

Client applications can obtain an AAT from Red Hat Single Sign-On like any other OAuth2 access token. Usually, client applications obtain AATs after the user is successfully authenticated in Red Hat Single Sign-On. By default, the *authorization_code* grant type is used to authenticate users, and the server will issue an OAuth2 access token to the client application acting on their behalf.

The example below uses the Resource Owner Password Credentials Grant Type to request an AAT:

```
curl -X POST \
    -H "Authorization: Basic
aGVsbG8td29ybGQtYXV0aHotc2VydmljZTpwYXNzd29yZA==" \
    -H "Content-Type: application/x-www-form-urlencoded" \
    -d
'username=${username}&password=${user_password}&grant_type=password' \
    "http://localhost:8080/auth/realms/${realm_name}/protocol/openid-
connect/token"
```

As a result, the server response is:

```
{
  "access_token": ${AAT},
  "expires_in": 300,
  "refresh_expires_in": 1800,
  "refresh_token": ${refresh_token},
  "token_type": "bearer",
  "id_token": ${id_token},
  "not-before-policy": 0,
  "session_state": "3cad2afc-855b-47b7-8e4d-a21c66e312fb"
}
```

8.2.2. Requesting Authorization Data and Token

Client applications using the UMA protocol can use a specific endpoint to obtain a special security token called a requesting party token (RPT). This token consists of all the permissions granted to a user as a result of the evaluation of the permissions and authorization policies associated with the resources being requested. With an RPT, client applications can gain access to protected resources at the resource server.

```
http://${host}:${port}/auth/realms/${realm_name}/authz/authorize
```

When requesting an RPT, you need to provide two things:

- A permission ticket with the resources you want to access
- The authorization API token (AAT) (as a bearer token) representing a user’s identity and his consent to access authorization data on his behalf.

```
curl -X POST
    -H "Authorization: Bearer ${AAT}" -d '{
    "ticket" : ${PERMISSION_TICKET}
}' "http://localhost:8080/auth/realms/hello-world-authz/authz/authorize"
```

As a result, the server response is:

```
{"rpt":"${RPT}"}
```

8.2.2.1. Requesting Party Token

A Requesting Party Token (RPT) is a JSON web token (JWT) digitally signed using JSON Web Signature (JWS). The token is built based on the AAT sent by the client during the authorization process.

When you decode an RPT you will see something like:

```
{
  "authorization": {
      "permissions": [
        {
          "resource_set_id": "d2fe9843-6462-4bfc-baba-b5787bb6e0e7",
          "resource_set_name": "Hello World Resource"
        }
      ]
  },
  "jti": "d6109a09-78fd-4998-bf89-95730dfd0892-1464906679405",
  "exp": 1464906971,
  "nbf": 0,
  "iat": 1464906671,
  "sub": "f1888f4d-5172-4359-be0c-af338505d86c",
  "typ": "kc_ett",
  "azp": "hello-world-authz-service"
}
```

From this token you can obtain all permissions granted by the server from the **permissions** claim.

8.3. ENTITLEMENT API

The Entitlement API provides a 1-legged protocol for obtaining authorization data from the server, where the authorization data represents the result of the evaluation of all permissions and authorization policies associated with the resources being requested.

Unlike the *Authorization API*, the Entitlement API is not UMA-compliant and does not require permission tickets.

The purpose of this API is provide a more lightweight API for obtaining authorization data, where a client in possession of a valid OAuth2 access token is able to obtain the necessary authorization data on behalf of its users.

8.3.1. Requesting Entitlements

Client applications can use a specific endpoint to obtain a special security token called a requesting party token (RPT). This token consists of all the entitlements (or permissions) for a user as a result of the evaluation of the permissions and authorization policies associated with the resources being requested. With an RPT, client applications can gain access to protected resources at the resource server.

```
http://${host}:${port}/auth/realms/${realm_name}/authz/entitlement
```

8.3.1.1. Obtaining Entitlements

The easiest way to obtain entitlements for a specific user is using an HTTP GET request. For example, using curl:

```
curl -X GET \
    -H "Authorization: Bearer ${access_token}" \
    "http://localhost:8080/auth/realms/hello-world-
authz/authz/entitlement/${resource_server_id}"
```

Note

When requesting entitlements using this endpoint, you must provide the access_token (as a bearer token) representing a user’s identity and his consent to access authorization data on his behalf.

In the curl example, **${resource_server_id}** is the **client_id** registered with the client application acting as a resource server.

As a result, the server response is:

```
{
  "rpt": ${RPT}
}
```

Using this method to obtain entitlements, the server responds to the requesting client with **all** entitlements for a user, based on the evaluation of the permissions and authorization policies associated with the resources managed by the resource server.

8.3.1.2. Obtaining Entitlements for a Specific Set of Resources

You can also use the entitlements endpoint to obtain a user’s entitlements for a set of one or more resources. For example, using curl:

```
curl -X POST -H "Authorization: Bearer ${access_token}" -d '{
    "permissions" : [
        {
            "resource_set_name" : "Hello World Resource"
        }
    ]
}' "http://localhost:8080/auth/realms/hello-world-
authz/authz/entitlement/hello-world-authz-service"
```

As a result, the server response is:

```
{
  "rpt": ${RPT}
}
```

Unlike the GET version, the server responds with an RPT holding the permissions granted during the evaluation of the permissions and authorization policies associated with the resources being requested.

When requesting entitlements, you can also specify the scopes you want to access. For example, using curl:

```
curl -X POST -H "Authorization: Bearer ${access_token}" -d '{
    "permissions" : [
        {
            "resource_set_name" : "Hello World Resource",
            "scopes" : [
                "urn:my-app.com:scopes:view"
            ]
        }
    ]
}' "http://localhost:8080/auth/realms/hello-world-
authz/authz/entitlement/hello-world-authz-service"
```

8.3.1.3. Requesting Party Token

A requesting party token (RPT) is a JSON web token (JWT) digitally signed using JSON web signature (JWS). The token is built based on the access_token sent by the client during the authorization process.

When you decode an RPT, you see a payload similar to the following:

```
{
  "authorization": {
      "permissions": [
        {
          "resource_set_id": "d2fe9843-6462-4bfc-baba-b5787bb6e0e7",
          "resource_set_name": "Hello World Resource"
        }
      ]
  },
  "jti": "d6109a09-78fd-4998-bf89-95730dfd0892-1464906679405",
    "exp": 1464906971,
    "nbf": 0,
    "iat": 1464906671,
    "sub": "f1888f4d-5172-4359-be0c-af338505d86c",
    "typ": "kc_ett",
    "azp": "hello-world-authz-service"
}
```

From this token you can obtain all permissions granted by the server from the **permissions** claim.

8.4. INTROSPECTING A REQUESTING PARTY TOKEN

Sometimes you might want to introspect a requesting party token (RPT) to check its validity or obtain the permissions within the token to enforce authorization decisions on the resource server side.

There are two main use cases where token introspection can help you:

- When client applications need to query the token validity to obtain a new one with the same or additional permissions
- When enforcing authorization decisions at the resource server side, especially when none of the built-in policy enforcers fits your application

8.4.1. Obtaining Information about an RPT

The token introspection is essentially a OAuth2 token introspection-compliant endpoint from which you can obtain information about an RPT.

```
http://${host}:${port}/auth/realms/${realm_name}/protocol/openid-
connect/token/introspect
```

To introspect an RPT using this endpoint, you can send a request to the server as follows:

```
curl -X POST \
    -H "Authorization: Basic
aGVsbG8td29ybGQtYXV0aHotc2VydmljZTpzZWNyZXQ=" \
    -H "Content-Type: application/x-www-form-urlencoded" \
    -d 'token_type_hint=requesting_party_token&token=${RPT}' \
    "http://localhost:8080/auth/realms/hello-world-authz/protocol/openid-
connect/token/introspect"
```

Note

The request above is using HTTP BASIC and passing the client’s credentials (client ID and secret) to authenticate the client attempting to introspect the token, but you can use any other client authentication method supported by Red Hat Single Sign-On.

The introspection endpoint expects two parameters:

- **token_type_hint**

Use **requesting_party_token** as the value for this parameter, which indicates that you want to introspect an RPT.

- **token**

  Use the token string as it was returned by the server during the authorization process as the value for this parameter.

As a result, the server response is:

```
{
  "permissions": [
    {
      "resource_set_id": "90ccc6fc-b296-4cd1-881e-089e1ee15957",
      "resource_set_name": "Hello World Resource"
    }
  ],
  "exp": 1465314139,
  "nbf": 0,
  "iat": 1465313839,
  "aud": "hello-world-authz-service",
  "active": true
}
```

If the RPT is not active, this response is returned instead:

```
{
  "active": false
}
```

8.4.2. Do I Need to Invoke the Server Every Time I Want to Introspect an RPT?

No. Both Entitlement APIs use the JSON web token (JWT) specification as the default format for RPTs.

If you want to validate these tokens without a call to the remote introspection endpoint, you can decode the RPT and query for its validity locally. Once you decode the token, you can also use the permissions within the token to enforce authorization decisions.

This is essentially what the policy enforcers do. Be sure to:

- Validate the signature of the RPT (based on the realm’s public key)
- Query for token validity based on its *exp*, *iat*, and *aud* claims

8.5. AUTHORIZATION CLIENT JAVA API

If you are using Java, you can access all Red Hat Single Sign-On Authorization Services using a client API.

8.5.1. Maven Dependency

```
<dependencies>
    <dependency>
```

```
        <groupId>org.keycloak</groupId>
        <artifactId>keycloak-authz-client</artifactId>
        <version>${KEYCLOAK_VERSION}</version>
    </dependency>
</dependencies>
```

8.5.2. Configuration

The client configuration is defined in a JSON file as follows:

```
{
  "realm": "hello-world-authz",
  "auth-server-url" : "http://localhost:8080/auth",
  "resource" : "hello-world-authz-service",
  "credentials": {
    "secret": "secret"
  }
}
```

- **realm** (required)

  The name of the realm.

- **auth-server-url** (required)

  The base URL of the Red Hat Single Sign-On server. All other Red Hat Single Sign-On pages and REST service endpoints are derived from this. It is usually in the form https://host:port/auth.

- **resource** (required)

  The client-id of the application. Each application has a client-id that is used to identify the application.

- **credentials** (required) Specifies the credentials of the application. This is an object notation where the key is the credential type and the value is the value of the credential type. Currently only secret/password is supported.

8.5.3. Obtaining User Entitlements

Here is an example illustrating how to obtain user entitlements:

```
// create a new instance based on the configuration defined in keycloak-
authz.json
AuthzClient authzClient = AuthzClient.create();

// obtain an Entitlement API Token to get access to the Entitlement API.
// this token is an access token issued to a client on behalf of an user
// with a scope = kc_entitlement
String eat = getEntitlementAPIToken(authzClient);

// send the entitlement request to the server to
// obtain an RPT with all permissions granted to the user
EntitlementResponse response = authzClient.entitlement(eat)
    .getAll("hello-world-authz-service");
String rpt = response.getRpt();
```

```
System.out.println("You got a RPT: " + rpt);

// now you can use the RPT to access protected resources on the resource
server
```

Here is an example illustrating how to obtain user entitlements for a set of one or more resources:

```
// create a new instance based on the configuration defined in keycloak-
authz.json
AuthzClient authzClient = AuthzClient.create();

// obtain an Entitlement API Token to get access to the Entitlement API.
// this token is an access token issued to a client on behalf of an user
// with a scope = kc_entitlement
String eat = getEntitlementAPIToken(authzClient);

// create an entitlement request
EntitlementRequest request = new EntitlementRequest();
PermissionRequest permission = new PermissionRequest();

permission.setResourceSetName("Hello World Resource");

request.addPermission(permission);

// send the entitlement request to the server to obtain an RPT
// with all permissions granted to the user
EntitlementResponse response = authzClient.entitlement(eat)
    .get("hello-world-authz-service", request);
String rpt = response.getRpt();

System.out.println("You got a RPT: " + rpt);
```

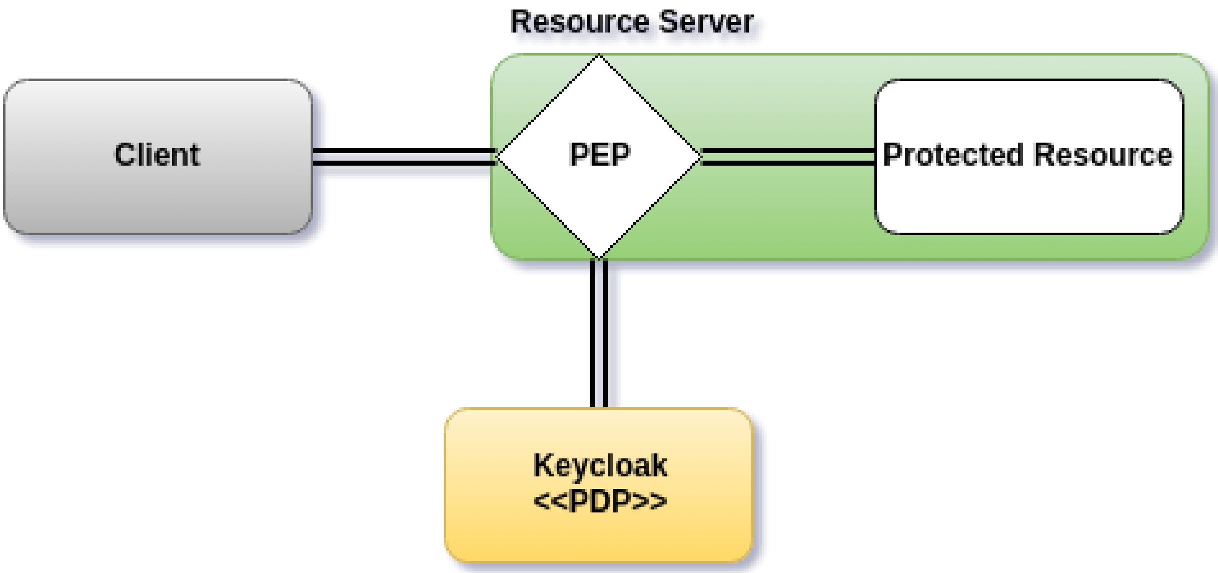
8.5.4. Creating a Resource Using the Protection API

```
// create a new instance based on the configuration defined in keycloak-
authz.json
AuthzClient authzClient = AuthzClient.create();

// create a new resource representation with the information we want
ResourceRepresentation newResource = new ResourceRepresentation();

newResource.setName("New Resource");
newResource.setType("urn:hello-world-authz:resources:example");

newResource.addScope(new ScopeRepresentation("urn:hello-world-
authz:scopes:view"));

ProtectedResource resourceClient = authzClient.protection().resource();
Set<String> existingResource = resourceClient
    .findByFilter("name=" + newResource.getName());

if (!existingResource.isEmpty()) {
    resourceClient.delete(existingResource.iterator().next());
}
```

```
// create the resource on the server
RegistrationResponse response = resourceClient.create(newResource);
String resourceId = response.getId();

// query the resource using its newly generated id
ResourceRepresentation resource =
resourceClient.findById(resourceId).getResourceDescription();
```

CHAPTER 9. POLICY ENFORCERS

Policy Enforcement Point (PEP) is a design pattern and as such you can implement it in different ways. Red Hat Single Sign-On provides all the necessary means to implement PEPs for different platforms, environments, and programming languages. Red Hat Single Sign-On Authorization Services presents a RESTful API, and leverages OAuth2 authorization capabilities for fine-grained authorization using a centralized authorization server.



9.1. RED HAT SINGLE SIGN-ON ADAPTER POLICY ENFORCER

You can enforce authorization decisions for your applications if you are using Red Hat Single Sign-On OIDC adapters.

When you enable policy enforcement for your Red Hat Single Sign-On application, the corresponding adapter intercepts all requests to your application and enforces the authorization decisions obtained from the server.

Policy enforcement is strongly linked to your application’s paths and the resources you created for a resource server using the Red Hat Single Sign-On Administration Console. By default, when you create a resource server, Red Hat Single Sign-On creates a default configuration for your resource server so you can enable policy enforcement quickly.

The default configuration allows access for all resources in your application provided the authenticated user belongs to the same realm as the resource server being protected.

9.1.1. Policy Enforcement Configuration

To enable policy enforcement for your application, add the following property to your **keycloak.json** file:

keycloak.json

```
{
  "policy-enforcer": {}
}
```

Or a little more verbose if you want to manually define the resources being protected:

```
{
  "policy-enforcer": {
    "user-managed-access" : {},
    "enforcement-mode" : "ENFORCING"
    "paths": [
      {
        "path" : "/someUri/*",
        "methods" : [
          {
            "method": "GET",
            "scopes" : ["urn:app.com:scopes:view"]
          },
          {
            "method": "POST",
            "scopes" : ["urn:app.com:scopes:create"]
          }
        ]
      },
      {
        "name" : "Some Resource",
        "path" : "/usingPattern/{id}",
        "methods" : [
          {
            "method": "DELETE",
            "scopes" : ["urn:app.com:scopes:delete"]
          }
        ]
      },
      {
        "path" : "/exactMatch"
      },
      {
        "name" : "Admin Resources",
        "path" : "/usingWildCards/*"
      }
    ]
  }
}
```

Here is a description of each configuration option:

- **policy-enforcer**

  Specifies the configuration options that define how policies are actually enforced and optionally the paths you want to protect. If not specified, the policy enforcer queries the server for all resources associated with the resource server being protected. In this case, you need to ensure the resources are properly configured with a URI property that matches the paths you want to protect.

  - **user-managed-access**

    Specifies that the adapter uses the UMA protocol. If specified, the adapter queries the server for permission tickets and return them to clients according to the UMA specification. If not specified, the adapter relies on the requesting party token (RPT) sent to the server to enforce permissions.

- **enforcement-mode**

  Specifies how policies are enforced.

  - **ENFORCING**

    (default mode) Requests are denied by default even when there is no policy associated with a given resource.

  - **PERMISSIVE**

    Requests are allowed even when there is no policy associated with a given resource.

  - **DISABLED**

    Completely disables the evaluation of policies and allows access to any resource.

- **on-deny-redirect-to**

  Defines a URL where a client request is redirected when an "access denied" message is obtained from the server. By default, the adapter responds with a 403 HTTP status code.

- **paths**

  Specifies the paths to protect.

  - **name**

    The name of a resource on the server that is to be associated with a given path. When used in conjunction with a **path**, the policy enforcer ignores the resource's **URI** property and uses the path you provided instead.

  - **path**

    (required) A URI relative to the application's context path. If this option is specified, the policy enforcer queries the server for a resource with a **URI** with the same value. Currently a very basic logic for path matching is supported. Examples of valid paths are:

    - Wildcards: `/*`
    - Suffix: `/*.html`
    - Sub-paths: `/path/*`
    - Path parameters: /resource/{id}
    - Exact match: /resource

  - **methods** The HTTP methods (for example, GET, POST, PATCH) to protect and how they are associated with the scopes for a given resource in the server. +[/"]

    - **method**

      The name of the HTTP method.

    - **scopes**

      An array of strings with the scopes associated with the method. When you associate scopes with a specific method, the client trying to access a protected resource (or path) must provide an RPT that grants permission to all scopes specified in the list. For example, if you define a method *POST* with a scope *create*, the RPT must contain a permission granting access to the *create* scope when performing a POST to the path.

- **enforcement-mode**

  Specifies how policies are enforced.

  - **ENFORCING**

    (default mode) Requests are denied by default even when there is no policy associated with a given resource.

  - **DISABLED**

    Disables the evaluation of policies for a path

9.1.2. Protecting a Stateless Service Using a Bearer Token

If the adapter is configured with the **`bearer-only`** configuration option, the policy enforcer decides whether a request to access a protected resource is allowed or denied based on the permissions of the bearer token.

1. HTTP GET example passing an RPT as a bearer token

```
GET /my-resource-server/my-protected-resource HTTP/1.1
Host: host.com
Authorization: Bearer ${RPT}
...
```

In this example, a **keycloak.json** file in your application is similar to the following:

Example of WEB-INF/keycloak.json with the bearer-only configuration option

```
...
"bearer-only" : true,
...
```

9.1.2.1. Authorization Response

When a client tries to access a resource server with a bearer token that is lacking permissions to access a protected resource, the resource server responds with a **401** status code and a **`WWW-Authenticate`** header. The value of the **`WWW-Authenticate`** header depends on the authorization protocol in use by the resource server.

Here is an example of a response from a resource server that is using UMA as the authorization protocol:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: UMA realm="photoz-restful-
api",as_uri="http://localhost:8080/auth/realms/photoz/authz/authorize",tic
ket="${PERMISSION_TICKET}"
```

And another example when the resource server is using the Entitlement protocol:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: KC_ETT realm="photoz-restful-
api",as_uri="http://localhost:8080/auth/realms/photoz/authz/entitlement"
```

Once a client receives a response from the server, it examines the status code and **`WWW-Authenticate`** header to obtain an RPT from the Red Hat Single Sign-On Server.

9.1.3. Obtaining the Authorization Context

When policy enforcement is enabled, the permissions obtained from the server are available through **`org.keycloak.AuthorizationContext`**. This class provides several methods you can use to obtain permissions and ascertain whether a permission was granted for a particular resource or scope.

Obtaining the Authorization Context in a Servlet Container

```
    HttpServletRequest request = ... // obtain
javax.servlet.http.HttpServletRequest
    KeycloakSecurityContext keycloakSecurityContext =
        (KeycloakSecurityContext) request
            .getAttribute(KeycloakSecurityContext.class.getName());
    AuthorizationContext authzContext =
        keycloakSecurityContext.getAuthorizationContext();
```

Note

For more details about how you can obtain a **`KeycloakSecurityContext`** consult the adapter configuration. The example above should be sufficient to obtain the context when running an application using any of the servlet containers supported by Red Hat Single Sign-On.

The authorization context helps give you more control over the decisions made and returned by the server. For example, you can use it to build a dynamic menu where items are hidden or shown depending on the permissions associated with a resource or scope.

```
if (authzContext.hasResourcePermission("Project Resource")) {
    // user can access the Project Resource
}

if (authzContext.hasResourcePermission("Admin Resource")) {
    // user can access administration resources
}

if (authzContext.hasScopePermission("urn:project.com:project:create")) {
    // user can create new projects
}
```

The **`AuthorizationContext`** represents one of the main capabilities of Red Hat Single Sign-On Authorization Services. From the examples above, you can see that the protected resource is not directly associated with the policies that govern them.

Consider some similar code using role-based access control (RBAC):

```
if (User.hasRole('user')) {
    // user can access the Project Resource
}

if (User.hasRole('admin')) {
    // user can access administration resources
}

if (User.hasRole('project-manager')) {
    // user can create new projects
}
```

Although both examples address the same requirements, they do so in different ways. In RBAC, roles only *implicitly* define access for their resources. With Red Hat Single Sign-On you gain the capability to create more manageable code that focuses directly on your resources whether you are using RBAC, attribute-based access control (ABAC), or any other BAC variant. Either you have the permission for a given resource or scope, or you don't.

Now, suppose your security requirements have changed and in addition to project managers, PMOs can also create new projects.

Security requirements change, but with Red Hat Single Sign-On there is no need to change your application code to address the new requirements. Once your application is based on the resource and scope identifier, you need only change the configuration of the permissions or policies associated with a particular resource in the authorization server. In this case, the permissions and policies associated with the **`Project Resource`** and/or the scope **`urn:project.com:project:create`** would be changed.

9.1.4. JavaScript Integration

The Red Hat Single Sign-On Server comes with a JavaScript library you can use to interact with a resource server protected by a policy enforcer. This library is based on the Red Hat Single Sign-On JavaScript adapter, which can be integrated to allow your client to obtain permissions from a Red Hat Single Sign-On Server.

You can obtain this library from a running a Red Hat Single Sign-On Server instance by including the following **`script`** tag in your web page:

```
<script src="http://.../auth/js/keycloak-authz.js"></script>
```

Once you do that, you can create a **`KeycloakAuthorization`** instance as follows:

```
var keycloak = ... // obtain a Keycloak instance from keycloak.js library
var authorization = new KeycloakAuthorization(keycloak);
```

The **keycloak-authz.js** library provides two main features:

- Handle responses from a resource server protected by a Red Hat Single Sign-On Policy Enforcer and obtain a requesting party token (RPT) with the necessary permissions to gain access to the protected resources on the resource server.
    - In this case, the library can handle whatever authorization protocol the resource server is using: Entitlements.
- Obtain permissions from a Red Hat Single Sign-On Server using the Entitlement API.

In both cases, the library allows you to easily interact with both resource server and Red Hat Single Sign-On Authorization Services to obtain tokens with permissions your client can use as bearer tokens to access the protected resources on a resource server.

9.1.4.1. Handling Authorization Responses from a Resource Server

If a resource server is protected by a policy enforcer, it responds to client requests based on the permissions carried along with a bearer token. Typically, when you try to access a resource server with a bearer token that is lacking permissions to access a protected resource, the resource server responds with a **401** status code and a **`WWW-Authenticate`** header.

The value of the **`WWW-Authenticate`** header depends on the authorization protocol in use by the resource server. Whatever protocol is in use, you can use a **`KeycloakAuthorization`** instance to handle responses as follows:

```
var wwwAuthenticateHeader = ... // extract WWW-Authenticate Header from
the response in case of a 401 status code
authorization.authorize(wwwAuthenticateHeader).then(function (rpt) {
    // onGrant callback function.
    // If authorization was successful you'll receive an RPT
    // with the necessary permissions to access the resource server
}, function () {
    // onDeny callback function.
    // Called when the authorization request is denied by the server
}, function () {
    // onError callback function. Called when the server responds
unexpectedly
});
```

The **`authorize`** function is completely asynchronous and supports a few callback functions to receive notifications from the server:

- **`onGrant`**: The first argument of the function. If authorization was successful and the server returned an RPT with the requested permissions, the callback receives the RPT.

- **`onDeny`**: The second argument of the function. Only called if the server has denied the authorization request.

- **`onError`**: The third argument of the function. Only called if the server responds unexpectedly.

Most applications should use the **`onGrant`** callback to retry a request after a 401 response. Subsequent requests should include the RPT as a bearer token for retries.

9.1.4.2. Obtaining Entitlements

The keycloak-authz.js library provides an **`entitlement`** function that you can use to obtain an RPT from the server using the Entitlement API.

```
authorization.entitlement('my-resource-server-id').then(function (rpt) {
    // onGrant callback function.
    // If authorization was successful you'll receive an RPT
    // with the necessary permissions to access the resource server
});
```

When using the **`entitlement`** function, you must provide the *client_id* of the resource server you want to access.

The **`entitlement`** function is completely asynchronous and supports a few callback functions to receive notifications from the server:

- **`onGrant`**: The first argument of the function. If authorization was successful and the server returned an RPT with the requested permissions, the callback receives the RPT.
- **`onDeny`**: The second argument of the function. Only called if the server has denied the authorization request.
- **`onError`**: The third argument of the function. Only called if the server responds unexpectedly.

9.1.4.3. Obtaining the RPT

If you have already obtained an RPT using any of the authorization functions provided by the library, you can always obtain the RPT as follows from the authorization object (assuming that it has been initialized by one of the techniques shown earlier):

```
var rpt = authorization.rpt;
```

9.1.5. Setting Up TLS/HTTPS

When the server is using HTTPS, ensure your adapter is configured as follows:

keycloak.json

```
{
  "truststore": "path_to_your_trust_store",
  "truststore-password": "trust_store_password"
}
```

The configuration above enables TLS/HTTPS to the Authorization Client, making possible to access a Red Hat Single Sign-On Server remotely using the HTTPS scheme.

Note

It is strongly recommended that you enable TLS/HTTPS when accessing the Red Hat Single Sign-On Server endpoints.