KeyCloak – щит от JBOSS для WEB приложений.

Из общения двух программистов:

* Кажется, у нас дыра в безопасности!
* Слава Богу, хоть что-то у нас в безопасности...

# Введение

Пару лет назад мы уже затрагивали тему безопасности. Тогда в рамках исследовательских работ был реализован собственный Service Provider. В качестве сервера аутентификации был использован Shibboleth, коммуникация с которым велась по протоколу SAML 2.0.

Сегодня речь снова пойдет о безопасности в сфере веб приложений. Мы сделаем небольшой обзор продукта KeyCloak (доселе оставленным без внимания сообществом Habr). В качестве практической ценности будет разобран пример, как защитить простое JEE приложение средствами KeyCloak, а также как осуществить взаимодействие между двумя защищенными приложениями (SSO).

# KeyCloak и PicketLink – два продукта под одной крышей

В настоящее время JBoss ведет разработку двух продуктов в области безопасности: KeyCloak и PicketLink. Вероятно, в ближайшем будущем оба продукты объединят в один, о чем уже давненько ходят разговоры <http://picketlink.org/news/2015/03/10/PicketLink-and-Keycloak-project-merge/>.  
  
Сегодня мы не будем останавливаться на проведении детального анализа сравнения двух продуктов (кое-что на эту тему можно посмотреть тут <http://planet.jboss.org/post/what_is_the_difference_between_picketlink_and_keycloak>). Буквально в двух словах отметим, что PicketLink позволяет построить уровень безопасности используя программную модель конфигурации. Таким образом PicketLink предоставляет набор библиотек, хорошо документированный API и достаточно широкий набор примеров, оформленных в виде quick start приложений. Все это вы найдете на официальном сайте <http://picketlink.org/>. Придется потратить кое-какое время, чтобы разобраться с API и научиться правильно его использовать. Около года назад у нас был опыт применения PicketLink в качестве idP сервера для построения SSO на протоколе SAML 2.0. Также средствами PicketLink был сконфигурирован STS сервис для защиты REST сервисов. Однако это отдельная тема для обсуждения ☺  
  
KeyCloak, в свою очередь, предоставляет почти все возможности из коробки, предоставляя простой административный интерфейс. Его следует использовать в типичных случаях, когда требования укладываются в типичные сценарии. На нем сегодня мы и остановимся поподробнее.

# KeyCloak – берем все из коробки…

Keycloak (<http://keycloak.jboss.org/>) - это open-source сервер аутентификации и управления учетными записями (IDM) от JBoss, построенный на базе спецификаций OAuth 2.0, Open ID Connect, JSON Web Token (JWT) и SAML 2.0.  
Список фич KeyCloak достаточно большой, который включает поддержку SSO, Social Login, интеграция с LDAP серверами, управление пользователями, группами и ролями, и много других плюшек. Полный список фич можно посмотреть тут -> <http://keycloak.github.io/docs/userguide/keycloak-server/html_single/index.html#Overview>.

## Как работает аутентификация в KeyCloak?

После этапа настройки приложения и KeyCloak сервера схема авторизации выглядит так:



1. Шаг-1: Запрос защищенного ресурса. Пользователь в браузере обращается по URL к закрытому ресурсу.
2. Шаг-2: Закрытое приложение перенаправляет неавторизованного пользователя на сервер аутентификации KeyCloak.
3. Шаг-3: KeyCloak отображает страницу аутентификации (логин/пароль, социальный логин, и т.д.).
4. Шаг-4: Пользователь проходит этап аутентификации. Для простоты будем считать, что вводит логин и пароль.
5. Шаг 5: KeyCloak выдает временный токен (секрет) и делает редирект на страницу защищенного приложения.
6. Шаг-6 и Шаг-7: Приложение проверяет валидность временного токена и меняет временный на постоянный JWT токен.
7. Шаг 8: На защищенном приложении проходит этап формирования контекста безопасности. Пользователю отображается защищенного ресурса.

Как мы видим, все достаточно стандартно.

## Немного о JWT.

JWT (JSON Web Token) - молодой открытый стандарт (<https://tools.ietf.org/html/rfc7519>), который определяет компактный и автономный способ для защищенной передачи информации между сторонами в виде JSON-объекта.

Основные свойства:

1. Компактный. Действительно в отличие от SAML сообщений, формат JWT намного проще. Состоит из трех частей: заголовок, основная информация и цифровая подпись.
2. Емкий. Содержит информацию по аутентифицированному пользователю, включая роли.
3. Самодостаточный. Не требуется обращаться к единому серверу для проверки токена (STS, idP). Сервис/приложение может осуществлять эту проверку самостоятельно.

Согласно стандарту токен состоит из трех частей в base-64 формате, разделенных точками. Первая часть называется заголовком (header), в которой содержится тип токена и название хэш алогиритма для получения цифровой подписи. Вторая часть хранит ценную информацию (пользователь, атрибуты, роли и т.д.). Третья часть – цифровая подпись. Более детальную информацию можно посмотреть тут <http://jwt.io/introduction/>. Были посты по этой теме и на хабре (например, <http://habrahabr.ru/post/243427/>).

## Настройка SSO

Получив JWT токен, мы по сути дела уже имеем готовый SSO. Несколько простых конфигурационных шагов - и мы сможем использовать защищенные данные другого приложения.

# Практическая часть.

В рамках практической части будут выполнены следующие шаги:

1. Сначала создадим простое веб приложение (video-app), которое отображает простой список видео объектов.
2. Далее установим и настроим KeyCloak.
3. Защитим приложение средствами KeyCloak.
4. Реализуем REST сервис в виде отдельного приложения (video-rest).
5. Защитим REST сервис средствами KeyCloak. Будем использовать bearer токен, для доступа к REST сервису.
6. Построим SSO: обновим приложение video-app, чтобы в качестве данных использовались данные из приложения video-rest.

Мы будем строить приложения с нуля простыми итерациями. Код готовой практической части можно взять здесь -> <https://github.com/EBTRussia/keycloak-demo.git>

## Используемые технологии

1. Maven (версия >= 3.2) как средство сборки.
2. WildFly (версия 9.0.1) в качестве сервера для деплоя приложений (включая KeyCloak). Взять можно тут <http://wildfly.org/downloads/>.
3. KeyCloak (версия 1.6.1)
4. RestEasy как имплементация JAX-RS для построения REST сервиса.  
   Идет в комплекте с WildFly.
5. JSP/JSTL – для проектирования вьюшек.
6. Чуть-чуть CSS и JS.

Мы предполагаем, что у читателя есть какой-то опыт с указанными технологиями (кроме KeyCloak). Поэтому комментировать моменты из серии что-такое JAX-RS или EJB в данной статье мы не будем.

## Родительский модуль – keycloak-demo

Родительский модуль хранит версии общих библиотек, версию JVM.

### Файл зависимости – pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.ebt.ressearch.keycloak</groupId>

<artifactId>keycloak-demo</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>pom</packaging>

<modules>

<module>common</module>

<module>video-app</module>

<module>video-rest</module>

</modules>

<properties>

<version.java>1.8</version.java>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<version.wildfly.maven.plugin>1.0.2.Final</version.wildfly.maven.plugin>

<version.compiler.plugin>3.1</version.compiler.plugin>

<version.war.plugin>2.5</version.war.plugin>

<version.jboss.bom>9.0.1.Final</version.jboss.bom>

<version.keycloak>1.6.1.Final</version.keycloak>

<version.jstl>1.2</version.jstl>

</properties>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.3.2</version>

<configuration>

<source>${version.java}</source>

<target>${version.java}</target>

</configuration>

</plugin>

</plugins>

</build>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.wildfly.bom</groupId>

<artifactId>jboss-javaee-7.0-wildfly-with-tools</artifactId>

<version>${version.jboss.bom}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<dependency>

<groupId>org.keycloak</groupId>

<artifactId>keycloak-core</artifactId>

<version>${version.keycloak}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.keycloak</groupId>

<artifactId>keycloak-adapter-core</artifactId>

<version>${version.keycloak}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.keycloak</groupId>

<artifactId>keycloak-services</artifactId>

<version>${version.keycloak}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.keycloak</groupId>

<artifactId>keycloak-jboss-adapter-core</artifactId>

<version>${version.keycloak}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>${version.jstl}</version>

<scope>provided</scope>

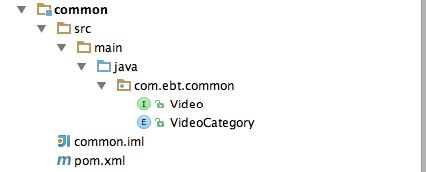
</dependency>

</dependencies>

</dependencyManagement>

</project>

## Модуль общих ресурсов (common)

В данном модуле определим общие интерфейсы, которые будут использоваться в остальных модулях/приложениях нашего практического примера.   
  


### Файл зависимостей – pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.ebt.ressearch.keycloak</groupId>

<artifactId>keycloak-demo</artifactId>

<version>1.0-SNAPSHOT</version>

</parent>

<artifactId>common</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

</project>

### Интерфейс объекта видео – Video.java

package com.ebt.common;

/\*\*

\* Интерфейс объектов видео.

\*

\* @author East Banc Technologies (http://eastbanctech.ru/)

\*/

public interface Video {

String getSource();

String getId();

String getTitle();

String getUrl();

Double getRating();

VideoCategory getCategory();

}

### Категория видео - VideoCategory.java

package com.ebt.common;

/\*\*

\* Видео категория для объекта Видео.

\*

\* @author East Banc Technologies (http://eastbanctech.ru/)

\*/

public enum VideoCategory {

SPORT,

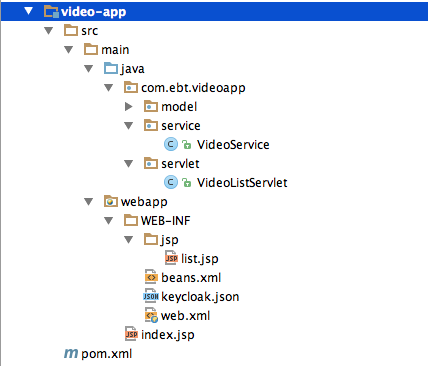
CARS,

MUSIC,

}

## Приложение для отображения списка видео (video-app)

Простое приложение отображения списка видео. Структура готового приложения будет выглядеть так.



### Файл зависимостей – pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.ebt.ressearch.keycloak</groupId>

<artifactId>keycloak-demo</artifactId>

<version>1.0-SNAPSHOT</version>

</parent>

<artifactId>video-app</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>war</packaging>

<build>

<finalName>video-app</finalName>

<plugins>

<plugin>

<groupId>org.wildfly.plugins</groupId>

<artifactId>wildfly-maven-plugin</artifactId>

<version>${version.wildfly.maven.plugin}</version>

</plugin>

</plugins>

</build>

<dependencies>

<dependency>

<groupId>com.ebt.ressearch.keycloak</groupId>

<artifactId>common</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>javax.enterprise</groupId>

<artifactId>cdi-api</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.jboss.spec.javax.annotation</groupId>

<artifactId>jboss-annotations-api\_1.2\_spec</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.jboss.resteasy</groupId>

<artifactId>resteasy-jaxrs</artifactId>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

<dependency>

<groupId>org.keycloak</groupId>

<artifactId>keycloak-core</artifactId>

</dependency>

<dependency>

<groupId>org.keycloak</groupId>

<artifactId>keycloak-adapter-core</artifactId>

</dependency>

<dependency>

<groupId>org.keycloak</groupId>

<artifactId>keycloak-services</artifactId>

</dependency>

<dependency>

<groupId>org.keycloak</groupId>

<artifactId>keycloak-jboss-adapter-core</artifactId>

</dependency>

</dependencies>

</project>

### Имплементация модели – VideoImpl.java

package com.ebt.videoapp.model;

import com.ebt.common.Video;

import com.ebt.common.VideoCategory;

/\*\*

\* Объект видео.

\*

\* @author East Banc Technologies (http://eastbanctech.ru/)

\*/

public class VideoImpl implements Video {

private String id;

private String title;

private String url;

private Double rating;

private VideoCategory category;

private String source;

// Объявление get & set методов

}

*Для экономии места get и set методы не описываем.*

### Сервис для работы с видео объектами – VideoService.java

package com.ebt.videoapp.service;

import com.ebt.common.Video;

import com.ebt.common.VideoCategory;

import com.ebt.videoapp.model.VideoImpl;

import java.util.Collections;

import java.util.LinkedList;

import java.util.List;

/\*\*

\* Сервис для управления объектами Видео.

\*

\* @author East Banc Technologies (http://eastbanctech.ru/)

\*/

public class VideoService {

private List<Video> list = new LinkedList<>();

public VideoService() {

VideoImpl video = new VideoImpl();

video.setTitle("Красная Феррари");

video.setUrl("http://www.youtube.com/watch?v=YJDz2-tT8b4");

video.setCategory(VideoCategory.CARS);

video.setRating(0.75);

video.setSource("VIDEO-APP");

list.add(video);

video = new VideoImpl();

video.setTitle("Lamborghini Aventador LP700-4");

video.setUrl("http://www.youtube.com/watch?v=ujn7jEQ4ib4");

video.setCategory(VideoCategory.CARS);

video.setRating(0.65);

video.setSource("VIDEO-APP");

list.add(video);

video = new VideoImpl();

video.setTitle("Lady Gaga - Bad Romance");

video.setUrl("http://www.youtube.com/watch?v=qrO4YZeyl0I");

video.setCategory(VideoCategory.MUSIC);

video.setRating(0.89);

video.setSource("VIDEO-APP");

list.add(video);

video = new VideoImpl();

video.setTitle("Shakira - La La La");

video.setUrl("http://www.youtube.com/watch?v=7-7knsP2n5w");

video.setCategory(VideoCategory.MUSIC);

video.setRating(0.88);

video.setSource("VIDEO-APP");

list.add(video);

video = new VideoImpl();

video.setTitle("Zlatan Ibrahimovic Goals & Skills");

video.setUrl("http://www.youtube.com/watch?v=ijAuwXZnxXc");

video.setCategory(VideoCategory.SPORT);

video.setRating(0.74);

video.setSource("VIDEO-APP");

list.add(video);

video = new VideoImpl();

video.setTitle("Goodbye Steven Gerrard - You're Irreplaceable");

video.setUrl("http://www.youtube.com/watch?v=bADTiAUWygA");

video.setCategory(VideoCategory.SPORT);

video.setRating(0.90);

video.setSource("VIDEO-APP");

list.add(video);

Collections.sort(list, (o1, o2) -> Double.compare(o1.getRating(), o2.getRating()));

}

public List<Video> list() {

return list;

}

}

Сервис построен на коллекции объектов. Для простоты создаем коллекцию прямо в сервисы. Понятно, что в реальном приложении будет промежуточный DAO слой для работы с БД или другим источником данных.

### Сервлет для отображения списка видео

package com.ebt.videoapp.servlet;

import com.ebt.videoapp.service.VideoService;

import org.keycloak.KeycloakSecurityContext;

import javax.annotation.security.DeclareRoles;

import javax.inject.Inject;

import javax.servlet.ServletException;

import javax.servlet.annotation.HttpConstraint;

import javax.servlet.annotation.ServletSecurity;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

/\*\*

\* Сервлет показывает список видео, используя в качестве данных внутренний сервис.

\*

\* @author East Banc Technologies (http://eastbanctech.ru/)

\*/

@WebServlet("/video-list-servlet")

public class VideoListServlet extends HttpServlet {

@Inject

private VideoService videoService;

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

req.setAttribute("list", videoService.list());

getServletContext().getRequestDispatcher("/WEB-INF/jsp/list.jsp").forward(req, resp);

}

}

Несколько комментариев :

1. Создаем наследник класса HttpServlet.
2. Определяем логику в методе doGet().
3. Используем аннотацию @WebServlet для привязки запроса к сервлету. Тем самым избавляемся от необходимости определять сервлет и задавать маппинг в web.xml.

Примечание: Аннотация @WebServlet появилась в спецификации Servlet API 3.0 и позволяет конфигурировать сервлеты прямо в Java коде.

### Отображение результатов – list.jsp

<%@ page contentType="text/html;charset=UTF-8" language="java" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>

<head>

<title>Список видео</title>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css"

integrity="sha512-dTfge/zgoMYpP7QbHy4gWMEGsbsdZeCXz7irItjcC3sPUFtf0kuFbDz/ixG7ArTxmDjLXDmezHubeNikyKGVyQ=="

crossorigin="anonymous">

</head>

<body>

<h2>Данные</h2>

<table class="table table-striped">

<thead>

<tr>

<th>Источник</th>

<th>Название</th>

<th>Категория</th>

<th>Рейтинг</th>

<th>Ссылка</th>

</tr>

</thead>

<tbody>

<c:forEach var="video" items="${list}">

<tr>

<td>${video.source}</td>

<td>${video.title}</td>

<td>${video.category}</td>

<td>${video.rating}</td>

<td><a href="${video.url}" target="\_new">Смотреть</a></td>

</tr>

</c:forEach>

</tbody>

</table>

</body>

</html>

### Включаем поддержку CDI – beans.xml

<?xml version="1.0" encoding="UTF-8"?>

<!-- enable CDI -->  
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://xmlns.jcp.org/xml/ns/javaee

http://xmlns.jcp.org/xml/ns/javaee/beans\_1\_1.xsd" bean-discovery-mode="all">

</beans>

### Дефолтный файл приложения – index.jsp

Перенаправляем пользователя на страницу со списком.

<%@ page contentType="text/html;charset=UTF-8" language="java" %>

<html>

<body onload="document.location='<%=request.getContextPath()%>/video-list-servlet'">

</body>

</html>

### Файл конфигурации веб приложения - web.xml

На текущий момент сделаем пустой файл. Далее контент будет изменен для конфигурации метода аутентификации.

<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd">

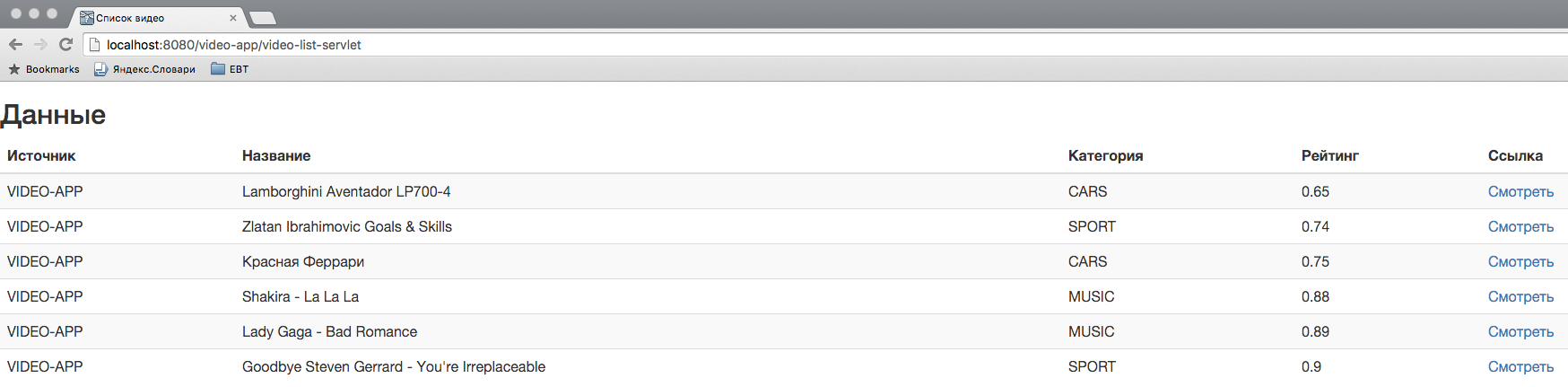
</web-app>

### Сборка и деплой

Собираем проект через maven: *mvn clean install*.

Деплоим собранное приложение в сервер приложений WildFly. Для этого копируем собранный *video-app.war* в *${WildFlyHome}/standalone/deployments*.

В броузере открываем страницу <http://localhost:8080/video-app/video-list-servlet> и проверяем, что страница отображается корректно.



## Установка и конфигурация сервера KeyCloak

### Установка KeyCloak в WildFly

Шаги взяты из инструкции (<http://keycloak.github.io/docs/userguide/keycloak-server/html_single/index.html>):

1. На сайте KeyCloak (<http://keycloak.jboss.org/keycloak/downloads>) находим файл патча WildFly сервера. Т.к. используем версию KeyCloak 1.6.1, нужен файл keycloak-overlay-1.6.1.Final.zip.  
     
   Скачиваем файл в <WILDFLY\_HOME> и распаковываем.   
     
   Этим шагом мы установили сервер KeyCloak в WildFly. Приложение будет доступен по корневому контексту **/auth**.
2. Устанавливаем адаптер для поддержки KeyCloak в веб приложениях
   1. Качаем адаптер для wildfly, файл "keycloak-wf9-adapter-dist-1.6.1.Final.zip" с сайта KeyCloak -> <http://keycloak.jboss.org/keycloak/downloads.html?dir=0%3Dadapters/keycloak-oidc%3B>   
      Скачиваем файл в <WILDFLY\_HOME> и распаковываем.

Подключили возможность использовать KeyCloak в настройке безопасности приложений (будет понятно чуть позже).

1. Подключаем KeyCloak к дефолтному профилю WildFly:
   1. Запускаем WildFly : *<WILDFLY\_HOME>/bin/standalone.sh*
   2. Переходим в папку <WILDFLY\_HOME>/bin и запускаем команды:  
      ./jboss-cli.sh -c --file=keycloak-install.cli

./jboss-cli.sh -c --file=adapter-install.cli

* 1. Перезапускаем сервер WildFly.

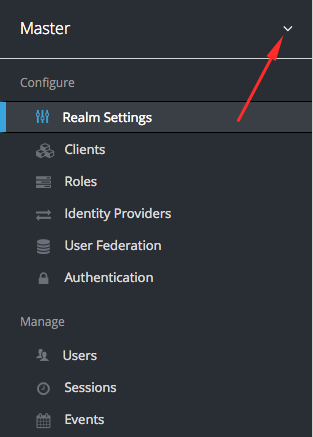
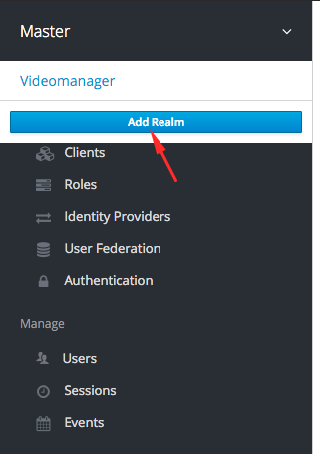
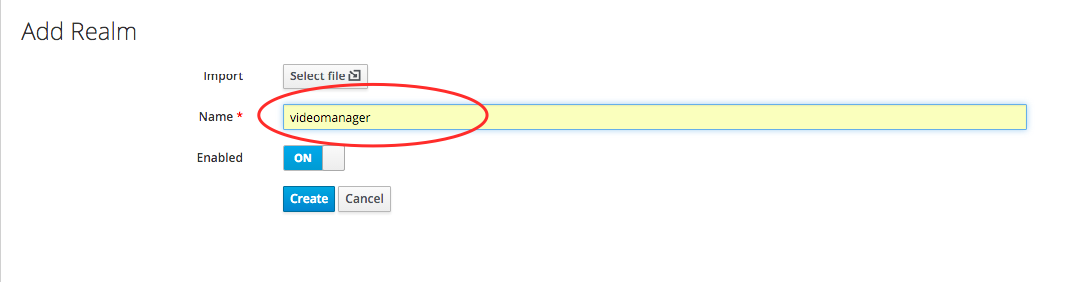
### Создаем Realm

Realm – область безопасности, которую мы определяем и настраиваем в KeyCloak. Realm’ы в KeyCloak позволяют создавать несколько различных конфигураций безопасности.   
  
Все настройки будут производится в рамках созданного Realm: как будет проходить аутентификация, какие приложения будут ее использовать, какие пользователи могут проходить аутентификацию, какие атрибуты будут возвращаться и т.д.

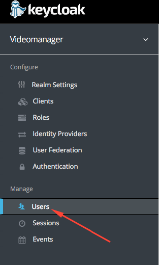
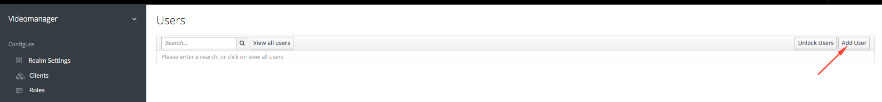
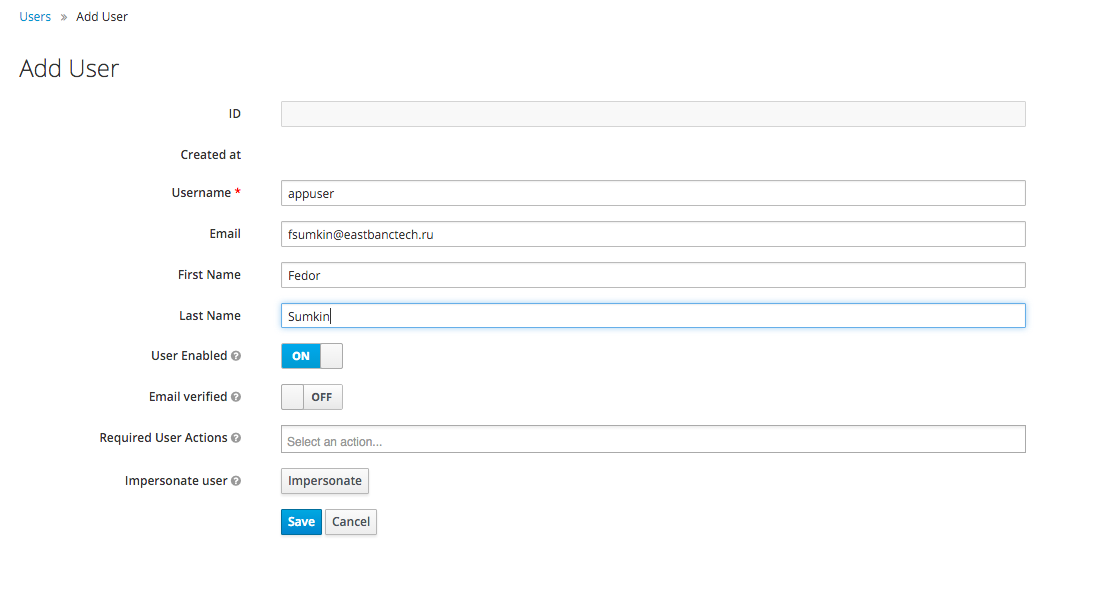
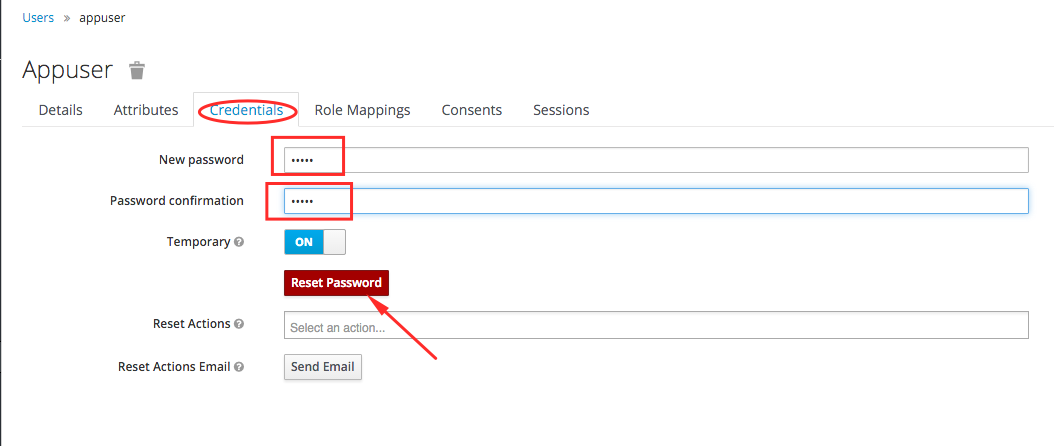
Шаги для создания Realm:

1. Запускаем WildFly сервер (если он остановлен).
2. Идем в административную панель Keycloak

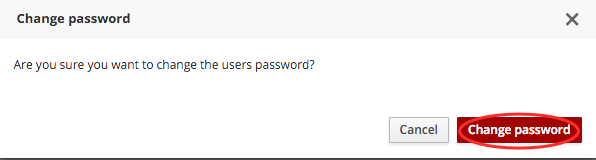
<http://localhost:8080/auth/admin/index.html>

1. Вводим логин и установленный пароль. (По умолчанию используется admin/admin, при первом запуске система попросит изменить пароль).
2. Раскрываем список существующих областей (галочка в левом верхнем углу)  
     
   
3. Появляется список существующих Realm’ов и кнопка для создания нового. Кликаем на нее.  
   
4. Создаем новый Realm с именем *videomanager* и жмем кнопку *Create*.  
   

### Создание нового пользователя

1. Кликаем на пункт Users в левом нижнем меню (секция Manage)  
   
2. Нажимаем кнопку создать пользователя  
   
3. Создаем пользователя *appuser*, определяем основные свойства (имя, фамилия, email) и жмем на кнопку сохранить.  
     
   
4. Открываем секцию Credentials. Проставляем пароль, подтверждение пароля для проверки и жмем кнопку Rest Password. 

Система покажет сообщение с просьбой подтвердить смену пароля.



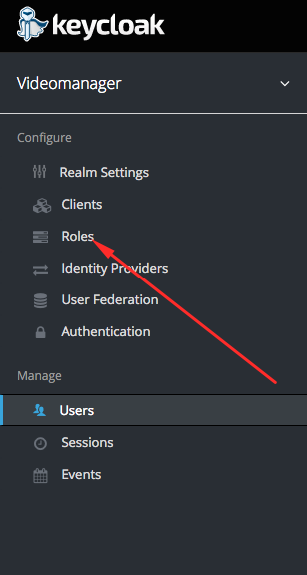
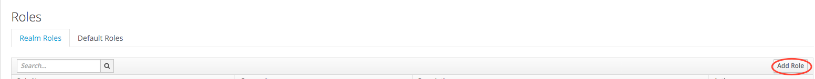
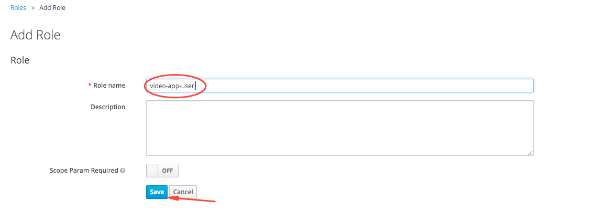
## Защита приложения video-app через KeyCloak

1. Идем в административную панель Keycloak

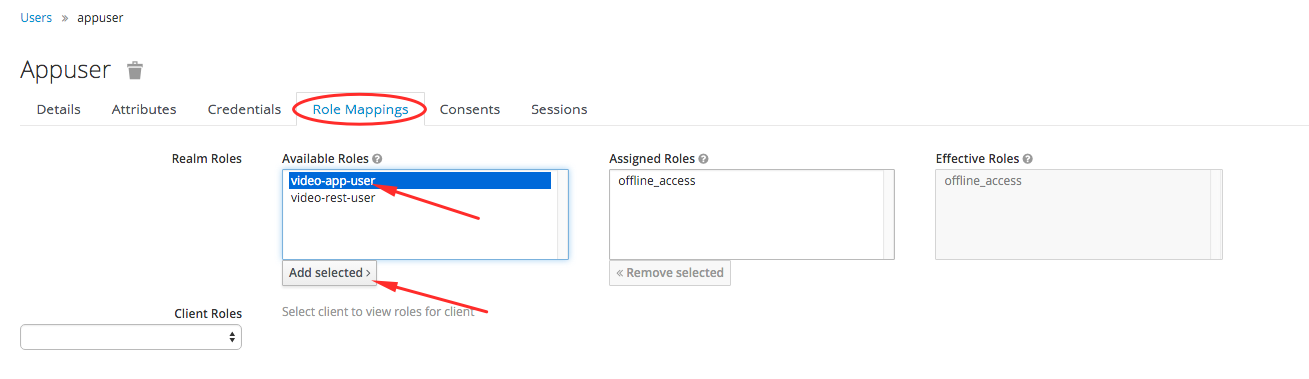
<http://localhost:8080/auth/admin/index.html>

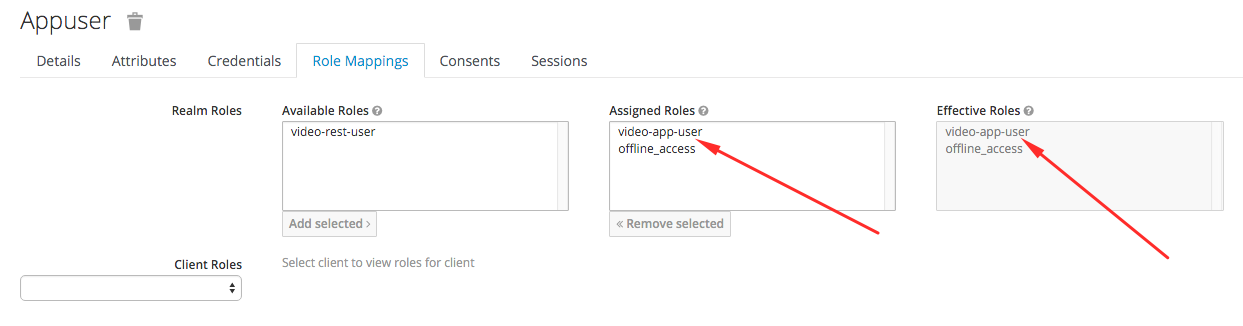
1. Открываем созданный нами Realm *videomanager. Все дальнейшие настройки будут проводится для данного Realm.*

### Добавление новой роли

1. Кликаем на пункт *Roles* в левом среднем меню (секция Configure)  
   
2. В правой части нажимаем на кнопку *Add Role*
3. Создаем Роль *video-app-user* и нажимаем кнопку *Save*.  
   

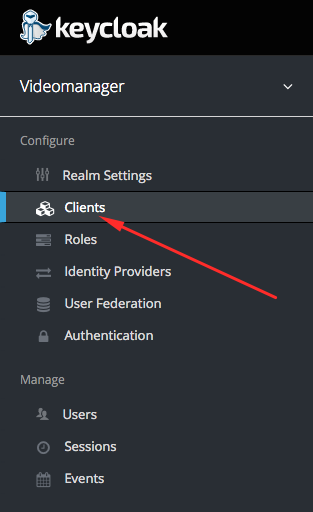
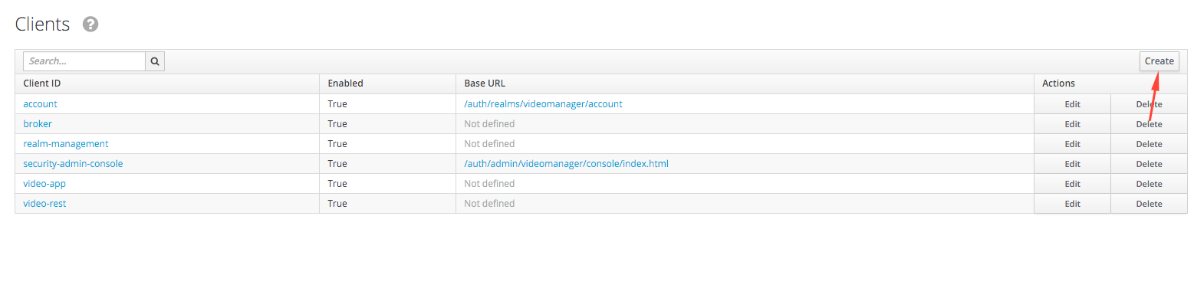
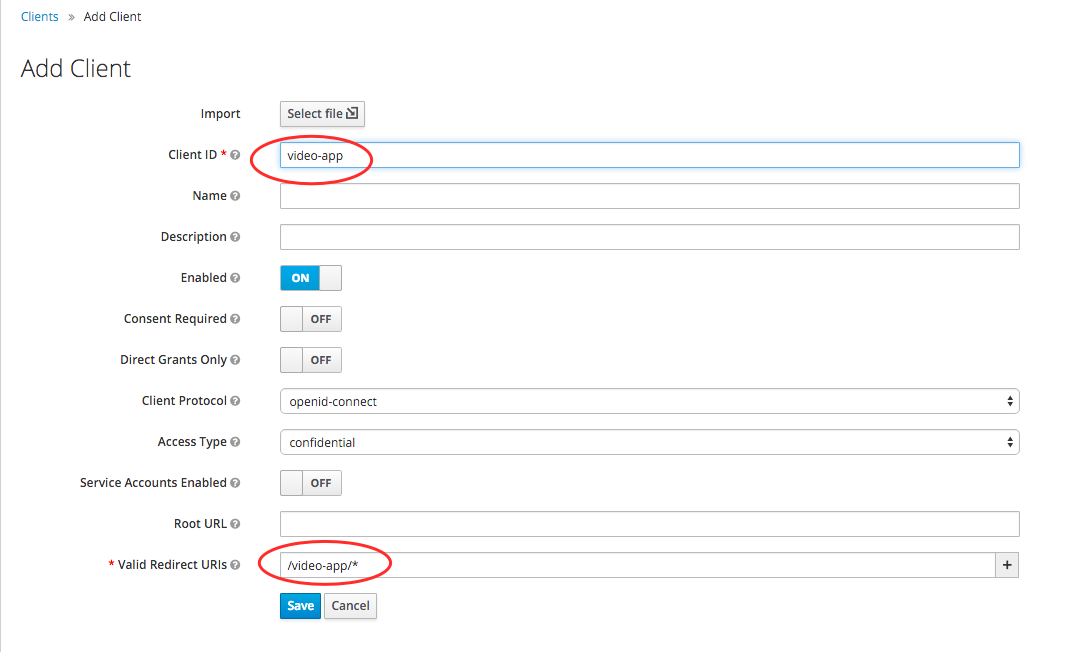
### Добавление роли пользователю

1. Открываем страницу редактирования пользователя *appuser* (который мы создали ранее).
2. Переходим на вкладку *Role Mappings.* В секции Available Roles выбираем роль video-app-user и нажимаем кнопку Add Selected.  
     
      
   Сохранение произойдет автоматически.



### Определяем приложение video-app в KeyCloak

Приложения, которые мы хотим защитить, описываются в секции Clients для выбранного Realm’а. Здесь мы будем описывать как будет проходить аутентификация в данное приложение, куда и какую информацию отправлять приложению после успешной аутентификации и т.д.

1. В секции Configure кликаем на пункт *Clients*.  
   
2. Отображается список приложений. В правом верхнем углу нажимаем кнопку *Create*.  
   
3. Определяем свойства клиента. На текущий момент нужно задать ID приложения и URL для возврата после аутентификации.   
     
   

Поле Valid Redirect URIs нужно для идентификации тех точек приложения куда допускается отправка результата после аутентификации (JWT токеном). Если у вас несколько таких точек, то все их нужно перечислить тут. Также можно задать маску: /video-app/\*. Это дает право KeyCloak делать редирект на любой ресурс, соответствующий данной маске.

### Добавляем слой безопасности в приложение video-app

#### Защищаем Сервлет VideoListServlet

Для этого просто добавляем на класс VideoListServlet аннотации DeclaredRoles, ServletSecurity.

@WebServlet("/video-list-servlet")

@DeclareRoles("video-app-user")

@ServletSecurity(@HttpConstraint(rolesAllowed = {"video-app-user"}))

public class VideoListServlet extends HttpServlet {  
}

Опять же можно использовать средства web.xml, но мы решили обойтись аннотациями.

#### Конфигурируем способ аутентификации

Открываем web.xml и добавляем запись следующего вида:

<login-config>

<auth-method>KEYCLOAK</auth-method>

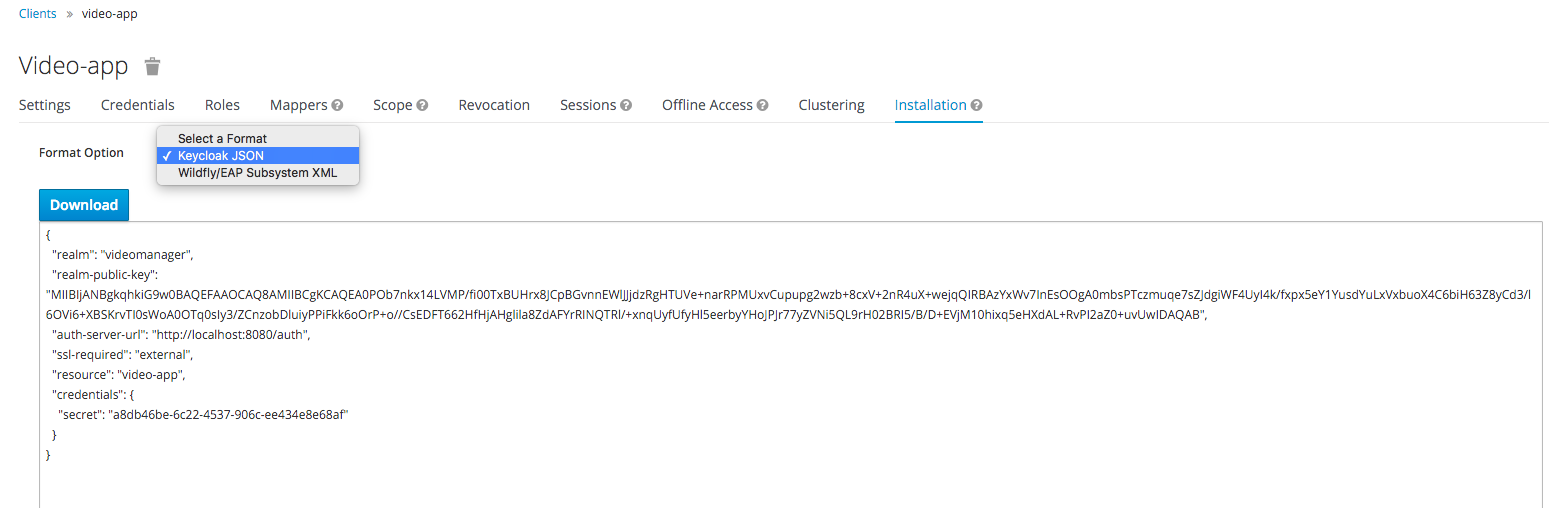
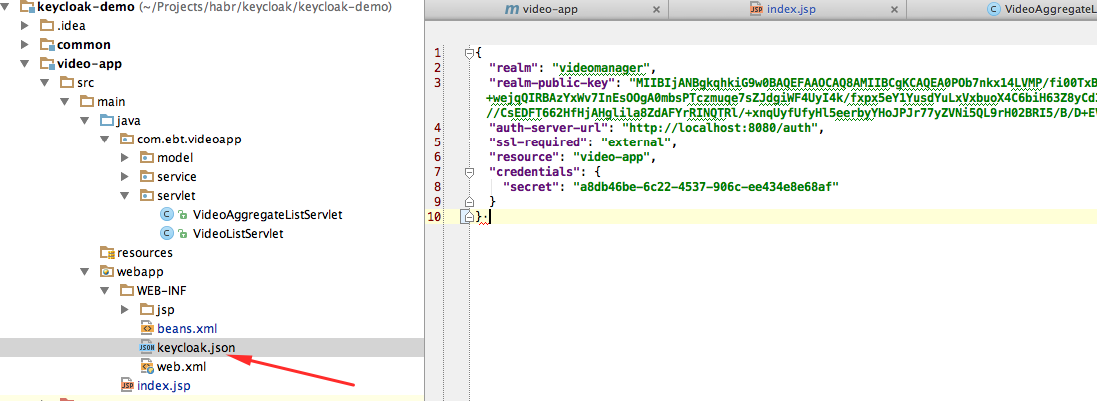
<realm-name>videomanager</realm-name>

</login-config>

Аннотация для конфигурации метода аутентификации пока нет. Здесь мы описываем, что   
для аутентификации будет использоваться KEYCLOAK. Этом метод аутентификации был добавлен в сервер приложений WildFly на шаге Установка и конфигурация сервера KeyCloak.

#### Интеграция с KeyCloak

Осталось добавить данные для интеграции с KeyCloak. Этот шаг делается очень просто.

1. Открываем в KeyCloak настройку клиента video-app. Переходим во вкладку Installation, выбираем KeyCloak JSON формат.   
     
   
2. Система покажет JSON, который нужно скопировать и поместить в файл web-app/WEB-INF/keycloak.json приложения video-app.  
     
    

### Сборка и деплой

Снова собираем и деплоим приложение.

Открываем страницу <http://localhost:8080/video-app/video-list-servlet>.  
Если все сконфигурировано верно, то система перенаправит вас на страницу аутентификации от KeyCloak. Вводим логин и пароль, созданного для Realm *videomanager* пользователя.

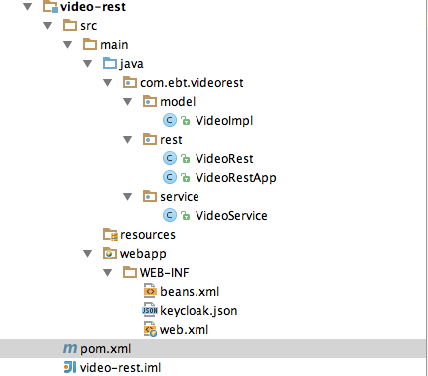
Примечание: При первом логине в приложение KeyCloak запросит обновить пароль у пользователя. Меняем и жмем Submit.  
  


Если все прошло успешно, вы снова увидите список видео.

## REST сервис, как источник видео – (video-rest)

Создадим простой REST сервис, который будет возвращать список видео.

Структура приложения будет выглядеть так.



### Файл зависимостей – pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.ebt.ressearch.keycloak</groupId>

<artifactId>keycloak-demo</artifactId>

<version>1.0-SNAPSHOT</version>

</parent>

<artifactId>video-rest</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>war</packaging>

<build>

<finalName>video-rest</finalName>

<plugins>

<plugin>

<groupId>org.wildfly.plugins</groupId>

<artifactId>wildfly-maven-plugin</artifactId>

<version>${version.wildfly.maven.plugin}</version>

</plugin>

</plugins>

</build>

<dependencies>

<dependency>

<groupId>com.ebt.ressearch.keycloak</groupId>

<artifactId>common</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>javax.enterprise</groupId>

<artifactId>cdi-api</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.jboss.spec.javax.annotation</groupId>

<artifactId>jboss-annotations-api\_1.2\_spec</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.jboss.spec.javax.ws.rs</groupId>

<artifactId>jboss-jaxrs-api\_2.0\_spec</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.keycloak</groupId>

<artifactId>keycloak-core</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.keycloak</groupId>

<artifactId>keycloak-adapter-core</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.jboss.spec.javax.ejb</groupId>

<artifactId>jboss-ejb-api\_3.1\_spec</artifactId>

<version>1.0.2.Final</version>

</dependency>

</dependencies>

</project>

### Имплементация модели - VideoImpl.java

Аналогично имплементации в приложении video-app (с точности до имени пакета).

### Сервис для работы с видео объектами - VideoService.java

Аналогично имплементации в приложении video-app.

### Создание REST приложения – VideoRest.java

package com.ebt.videorest.rest;

import javax.ws.rs.ApplicationPath;

import javax.ws.rs.core.Application;

/\*\*

\* Определение JAX-RS приложения.

\*

\* @author East Banc Technologies (http://eastbanctech.ru/)

\*/

@ApplicationPath("/")

public class VideoRestApp extends Application {

}

Определение JAX-RS приложение, по которому сервер приложений WildFly поймет, что мы объявляем REST сервисы.

### Создание REST сервиса – VideoRest.java

package com.ebt.videorest.rest;

import com.ebt.common.Video;

import com.ebt.videorest.service.VideoService;

import javax.annotation.security.RolesAllowed;

import javax.ejb.Stateless;

import javax.inject.Inject;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import java.util.List;

/\*\*

\* REST сервис.

\*

\* @author East Banc Technologies (http://eastbanctech.ru/)

\*/

@Path("/")

@Produces("application/json")

public class VideoRest {

@Inject

private VideoService videoService;

@GET

@Path("/list")

public List<Video> get() {

return videoService.list();

}

}

### Включаем поддержку CDI – beans.xml

В точности файл из приложения video-app.

### Файл конфигурации веб приложения - web.xml

<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd">

</web-app>

### Сборка и деплой

Собираем приложение через maven и деплоим в сервер в WildFly.

Открываем страницу <http://localhost:8080/video-rest/list>. Если все настроено верно, то мы увидим список видео объектов в формате JSON.  
  


## Защита приложения video-rest через KeyCloak

### Добавление новой роли

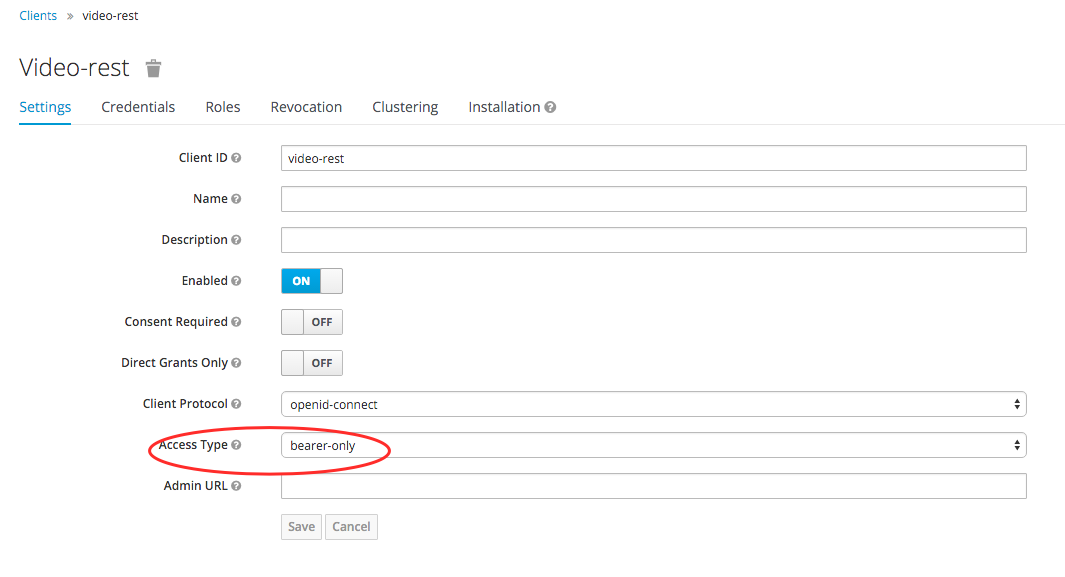
Добавляем новую роль video-rest-user.

### Добавление роли пользователю

Добавляем роль video-rest-user пользователю appuser.

### Определяем приложение video-rest в KeyCloak

Определяем video-rest в KeyCloak аналогично пункту **Определяем приложение video-app в KeyCloak** с одним замечанием. В AccessType выбираем значение *bearer-only*.   
Это означает, что приложение не будет инициировать процесс аутентификации в KeyCloak и ожидает получение пользователя и его атрибутов из JWT токенов.



### Добавляем слой безопасности в приложение video-rest

#### Защищаем сервис возврата списка видео объектов – VideoRest.java

Помечаем метод аннотацией @RolesAllowed. Сам класс сервиса помечаем аннотацией @Stateless, чтобы сделать его EJB (иначе аннотация *@RolesAllowed* работать не будет).   
  
В итоге получаем такой сервис:   
package com.ebt.videorest.rest;

import com.ebt.common.Video;

import com.ebt.videorest.service.VideoService;

import javax.annotation.security.RolesAllowed;

import javax.ejb.Stateless;

import javax.inject.Inject;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import java.util.List;

/\*\*

\* REST сервис.

\*

\* @author East Banc Technologies (http://eastbanctech.ru/)

\*/

@Path("/")

@Produces("application/json")

@Stateless

public class VideoRest {

@Inject

private VideoService videoService;

@GET

@Path("/list")

@RolesAllowed("video-rest-user")

public List<Video> get() {

return videoService.list();

}

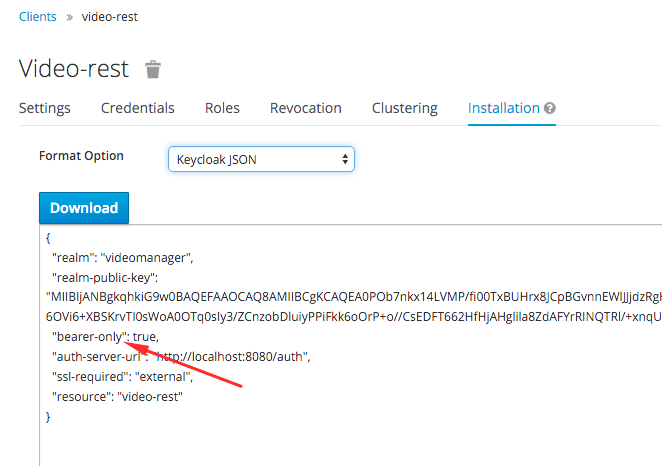
}

#### Конфигурируем способ аутентификации

Аналогично конфигурации приложения video-app.

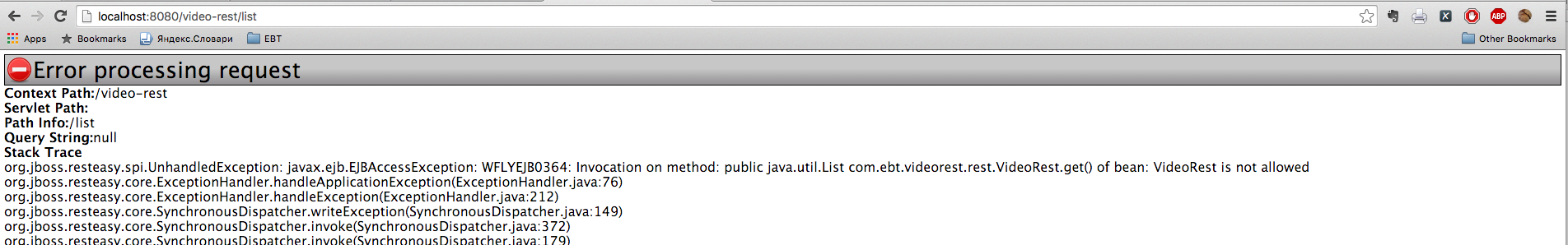
#### Интеграция с KeyCloak

Повторить те же шаги для приложения video-rest, что и для конфигурации с KeyCloak для приложения video-app.

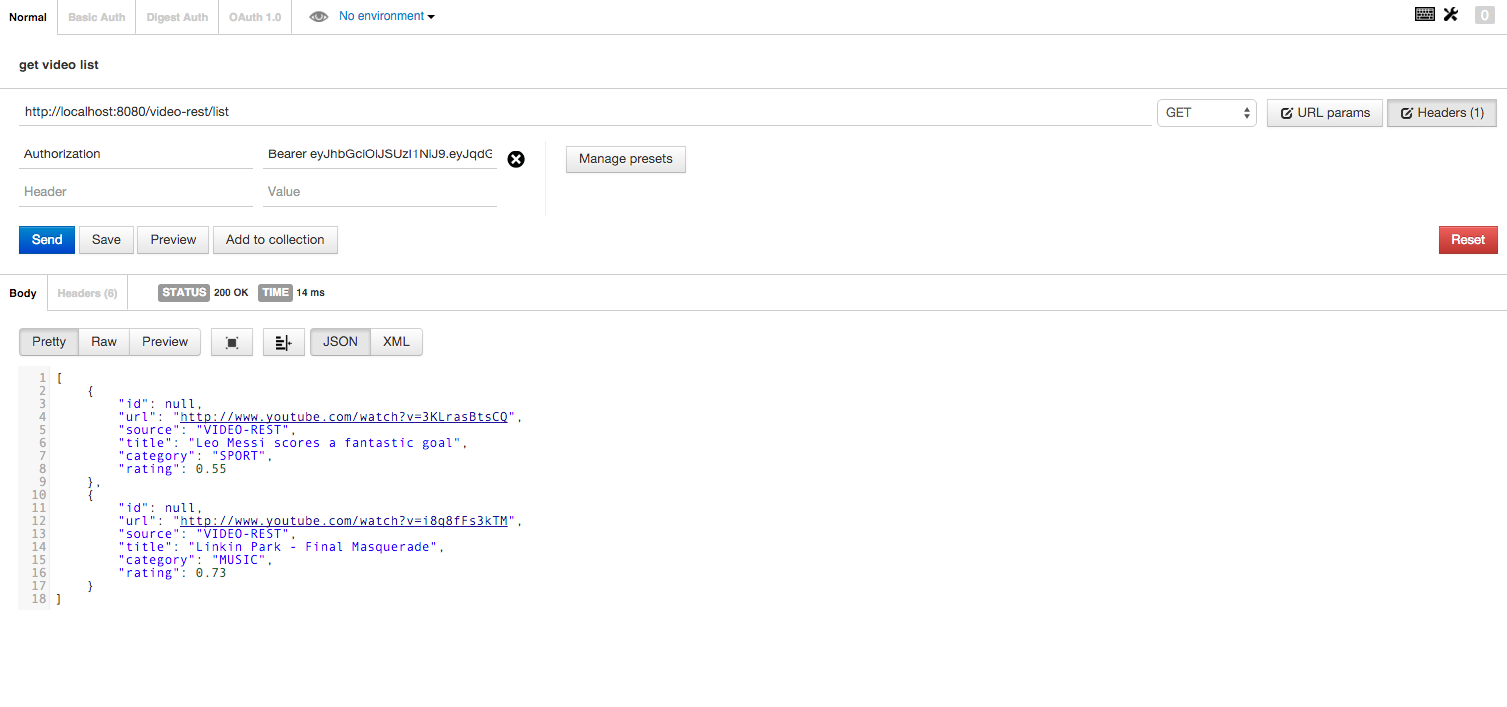
Обратите внимание, что в файле конфигурации параметр "bearer-only" должен быть установлен в значение "true".   
  


### Сборка и деплой

Собираем и деплоим приложение.

Открываем приложение по адресу <http://localhost:8080/video-rest/list>   
  
Система должна вывести сообщение об ошибке о недостаточности прав доступа.  
  


Если воспользоваться клиентом для REST запросов, в котором указать заголовок *Authorization* со значением *Bearer ${значение JWT токена},* получим список видео.   
Значение JWT можно получить из приложение video-app, используя метод getTokenString() объекта KeycloakSecurityContext. Объект класса получаем из контекста запроса.   
  
KeycloakSecurityContext ksc = (KeycloakSecurityContext) req.getAttribute(KeycloakSecurityContext.class.getName());

String token = ksc.getTokenString();  
  


На скриншоте показан вызов REST сервиса с JWT токеном. В качестве клиента использовалось расширение *Postman* для *Chrome*.

## SSO – интеграция video-app с video-rest

Изменяем логику сервлета VideoListServlet в приложении video-app для получения данных из REST сервиса. Будем выводить общий список видео, построенный на основе двух источников.

### Модифицируем сервлет VideoListServlet.java

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

Client client = ClientBuilder.newBuilder().build();

WebTarget target = client.target("http://localhost:8080/video-rest/list");

GenericType<List<VideoImpl>> listGenericType = new GenericType<List<VideoImpl>>() {

};

KeycloakSecurityContext ksc = (KeycloakSecurityContext) req.getAttribute(KeycloakSecurityContext.class.getName());

List<VideoImpl> list = target.request().header("Authorization", "Bearer " + ksc.getTokenString()).get(listGenericType);

// merge lists

List<Video> mergeList = new ArrayList<>();

mergeList.addAll(list);

mergeList.addAll(videoService.list());

req.setAttribute("list", mergeList);

req.setAttribute("ksc", ksc);

getServletContext().getRequestDispatcher("/WEB-INF/jsp/list.jsp").forward(req, resp);

}

Во фрагменте кода, получаем контекст безопасности, созданный в процессе авторизации средствами KeyCloak.

KeycloakSecurityContext ksc = (KeycloakSecurityContext) req.getAttribute(KeycloakSecurityContext.class.getName());

Далее обращаемся к защищенному ресурсу, используя JWT токен:  
List<VideoImpl> list = target.request().header("Authorization", "Bearer " + ksc.getTokenString()).get(listGenericType);

### Сборка и деплой

Собираем и деплоим приложение.   
Открываем страницу списка видео <http://localhost:8080/video-app/video-list-servlet>.



На скриншоте показано, что видео из разных источников были агрегированы в один список.

Вот собственно и все, практическая часть завершена. Были разработаны и защищены два веб приложения, а также организовано взаимодействие между этими приложениями с помощью JWT токенов от KeyCloak.

# Заключение

Сегодня мы показали каким образом можно использовать KeyCloak для защиты веб приложений. Большая часть статьи была потрачена на имплементацию практического примера. Надеемся, что статья будет полезная и сэкономит время тем, кто захочет подключить KeyCloak в свои веб системы или продолжить дальнейшее исследование этого продукта.

По работе с KeyCloak сделали такие выводы:

1. *KeyCloak* предоставляет возможность быстро настроить типичный уровень безопасности (когда не нужно изобретать чего-то хитрого).
2. *KeyCloak* позволяет изолировать управление пользователями и прав от логики приложения.
3. Сама настройка делается через интуитивно понятный интерфейс пользователя.
4. Достаточно много приятных плюшек из серии логин через социальные сети, политики пароля и прочее (не были затронуты тут).
5. Мы смогли объявлять роли только на уровне Realm. Роли, объявленные в рамках клиента, почему-то не захотели работать. С этим нужно разбираться.