# Création et déploiement d’un WAR dans Tomcat

# Démonstration 3 du module 10

|  |
| --- |
| Les objectifs de cette démonstration sont   * Configurer pour créer le WAR * Configurer l’API en 100% sécurisée * Utiliser AntPathRequestMatcher pour séparer SecurityFilterChain et ServletContext * Déployer le WAR dans Tomcat |

Contexte

* Nous continuons notre application précédente : demo-nosql.
  + Si votre application n’est pas complète, vous pouvez reprendre le code Java dans les ressources
  + Attention, le paramétrage du projet Spring Boot n’est pas inclus, il vous faut le compléter
  + Pensez aux 2 propriétés de GitHub pour OAuth 2.0 dans application.yml
* Dans cette itération, nous voulons créer un WAR

Déroulement

# Configuration des logs

* Nous voulons créer un WAR. Il faut éviter que l’application perde trop de temps à tracer des logs.
  + Pour éviter de saturer les ressources, nous allons changer le niveau de trace de MongoTemplate et le passer en ERROR uniquement
* Dans le fichier de configuration de Spring Boot : application.yml :

#Log

logging:

level:

root: WARN

org:

springframework:

data:

mongodb:

core:

MongoTemplate: ERROR

# Configuration du WAR :

Nous voulons gérer notre propre WAR et le numéro de version.

Dans build.gradle :

* Ajouter L’id war

plugins {

id 'java'

// DEPLOIEMENT WAR TOMCAT

id 'war'

id 'org.springframework.boot' *version* '3.1.4'

id 'io.spring.dependency-management' *version* '1.1.3'

}

* Et la production sous Tomcat :
  + Starter pour les serveurs web Java

// DEMO DEPLOIEMENT WAR TOMCAT

// Starter Tomcat

providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'

* Mettre à jour le numéro de version si besoin

*version* = '1.0.1'

* Ajouter un bloc de déclaration après le bloc java nommé : taks.war pour définir le nom du war désiré

*tasks*.war{

archiveBaseName.set("demo-nosql")

}

* Faire un « Refresh Gradle Project »

# Configuration du ServletContext pour Spring Boot et Tomcat :

* Ajouter sur la classe principale de SpringBoot :
  + L’héritage de SpringBootServletInitializer
  + Et redéfinir la méthode « configure » qui permettra d’associer ServletContext et Spring Boot et ainsi configurer l’exécution dans un Tomcat :

**package** fr.eni.demo;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.boot.builder.SpringApplicationBuilder;

**import** org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

@SpringBootApplication

**public** **class** DemoNosqlApplication **extends** SpringBootServletInitializer {

@Override

**protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {

**return** application.sources(DemoNosqlApplication.**class**);

}

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(DemoNosqlApplication.**class**, args);

}

}

# Contraintes de sécurité

Le DispatchServlet peut rentrer en contrainte avec le SecurityFilterChain.

* Dès qu’on utilise requestMatchers
* Car les 2 peuvent réagir à des URLs

Pour éviter cela, il y a 2 solutions :

1. Faire que l’application Web soit totalement sécurisée : anyRequest().authenticated()
2. Utiliser AntPathRequestMatcher pour préciser que le contexte de sécurité s’en occupe

Dans le cas de notre démonstration, nous voudrions conserver l’URL racine comme URL autorisée par défaut et gérée par notre sécurité. Pour cela ; nous allons mettre en place AntPathRequestMatcher

**package** fr.eni.demo.security.oauth;

**import** org.springframework.context.annotation.\*;

**import** org.springframework.security.config.annotation.web.builders.HttpSecurity;

**import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

**import** org.springframework.security.web.SecurityFilterChain;

**import** org.springframework.security.web.util.matcher.AntPathRequestMatcher;

**import** org.springframework.web.servlet.handler.HandlerMappingIntrospector;

@Configuration

@EnableWebSecurity

**public** **class** SecurityConfig {

/\*\*

\* il faut être authentifié pour accéder à l'API

\*/

@Bean

SecurityFilterChain filterChain(HttpSecurity http, HandlerMappingIntrospector introspector) **throws** Exception {

http

.authorizeHttpRequests(auth -> {

auth

// Permettre l'accès à l'URL racine à tout le monde

//.requestMatchers("/").permitAll()

//Remplacer par la ligne suivante pour que cette URL soit gérée par la sécurité

.requestMatchers(**new** AntPathRequestMatcher("/")).permitAll()

// Il faut être connecté pour toutes autres URLs

.anyRequest().authenticated();

})

// Indiquer à Spring Security l'utilisation de OAuth 2.0

.oauth2Login(oauth -> {

});

**return** http.build();

}

}

# Configuration OAuth 2 en production :

Nous avions créé une configuration sur GitHub pour l’autorisation avec OAuth 2 pour des URLs du type http://localhost:8080

* Maintenant, notre URL racine correspondra à http://localhost:8080/demo-nosql
* Il faut donc recréer une configuration OAuth pour la production
* Sous votre compte GitHub
* Allez sur vos paramètres
* Allez vers le bas sur l’option « Developer settings »
  + Sélectionner OAuth Apps
* Donner un nom à votre application
  + Exemple : Demo-Deploy-VotreNom
  + Et l’url de la page d’accueil
    - Mettre l’URL de « localhost » : http://localhost:8080/demo-nosql
  + L’URL d’autorisation 🡪 permet de gérer le jeton transmis par GitHub sur le contexte du projet (« localhost »)
    - http://localhost:8080/ demo-nosql/login/oauth2/code/github
* Vous obtiendrez une vue récapitulative
  + Où, GitHub aura créé un identifiant Client ID
* Il nous faut générer un « Client Secret »
  + GitHub vous demandera de ressaisir votre mot de passe GitHub pour valider
  + Il nous faut les 2 informations : Client ID et Client Secret pour effectuer toutes les opérations de connexion.
* Modifier les paramètres de OAuth 2.0 dans le fichier de configuration de Spring Boot : application.properties ou application.yml
  + Conseil, mettre en commentaire ceux de DEV

#configuration OAuth2 - GitHub

security:

oauth2:

client:

registration:

github:

#DEV

#client-id: 11a51…

#client-secret: 4f64…

#PROD

client-id: 2ad48…

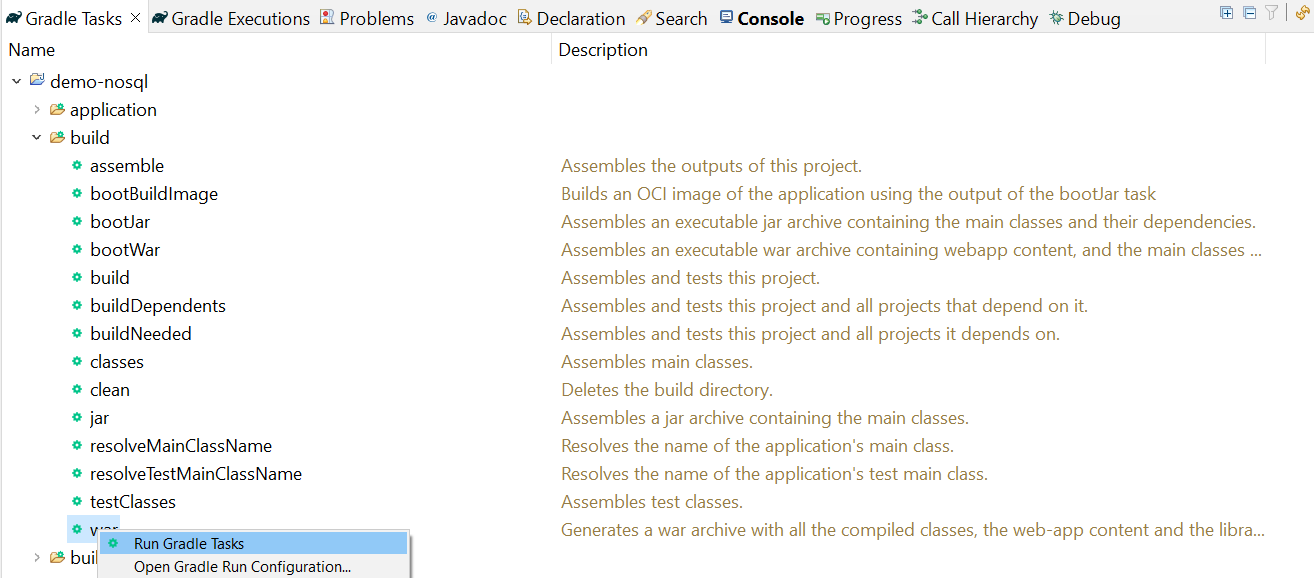
client-secret: 517ba…

Souvent en production, le fichier de configuration Spring Boot serait écrasé dans le WAR pour ce type de paramètre

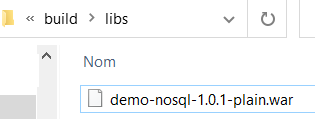
# Création du WAR :

Dans la vue « Gradle Tasks »

* Rechercher votre projet
* Ouvrir : build
* Faire un clic droit sur war 🡪 Run Gradle Tasks



* Dans la vue « Gradle Executions » vous verrez la progression du build
  + Une fois terminée
  + Allez dans un explorer sous votre projet dans votre workspace
  + Dans le répertoire « buid » 🡪 « libs »
  + Le WAR sera là



# Déploiement du WAR dans Tomcat :

* Copier demo-nosql-1.0.1-plain.war dans le répertoire webapps de Tomcat
  + Dans notre exemple : C:\java\_outils\apache-tomcat-10.1.8\webapps
  + Conseil : renommer votre WAR en demo-nosql.war pour que le nom de l’application dans l’URL soit uniquement demo-nosql
* Lancer une fenêtre de commande
  + Placez-vous dans le répertoire bin de votre Tomcat :
  + Exemple de commande pour le répertoire C:\java\_outils\apache-tomcat-10.1.8\bin

cd c:\java\_outils\apache-tomcat-10.1.8\bin

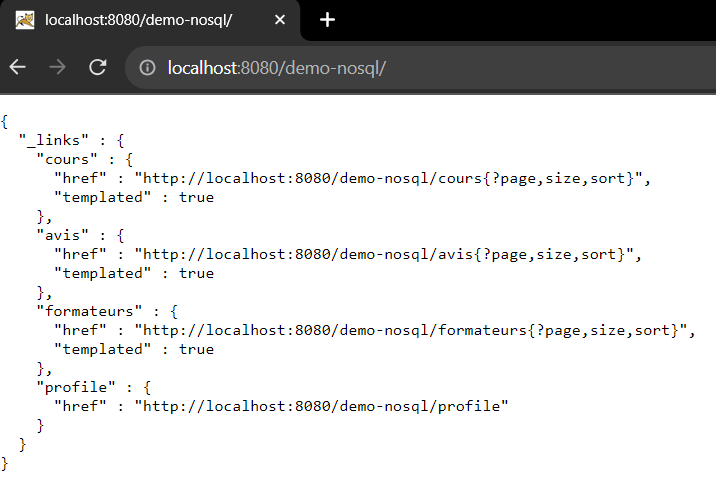
* Démarrer Tomcat en appelant la commande :

startup.bat

* Lorsque le démarrage est terminé
  + Tomcat va décompresser le WAR
  + Et exécuter l’application web
* Dans la console, vous verrez apparaitre la trace de démarrage de Spring Boot :



* Vous pouvez maintenant, vérifier le comportement de votre application déployée
* L’URL racine : http://localhost:8080/demo-nosql/
  + Est accessible par tout le monde :



* L’URL http://localhost:8080/demo-nosql/eni/swagger-ui/index.html
  + N’est accessible que si nous avons les autorisations de GitHub pour OAuth2 :

|  |  |
| --- | --- |
|  |  |