# 

Corso Avanzato Spring Boot

Dispense:

Giuseppe ing. Grosso

[1](#_Toc63148549)

[Introduzione 3](#_Toc63148550)

[Sintesi argomenti trattati: 3](#_Toc63148551)

[Requisiti ingresso 3](#_Toc63148552)

[Startup ambiente. 4](#_Toc63148553)

[Utilizzo di un IDE. 5](#_Toc63148554)

[Annotazioni. 5](#_Toc63148555)

[Spring Boot: struttura del progetto 6](#_Toc63148556)

[Scrittura del primo controller. 7](#_Toc63148557)

[Verbo http. 9](#_Toc63148558)

[Produzione e consumo dei dati. 10](#_Toc63148559)

[Service. 12](#_Toc63148560)

[Dependendy injection. 12](#_Toc63148561)

[Esternalizzazione configurazione. 13](#_Toc63148562)

[Internalization. 14](#_Toc63148563)

[ExceptionHandler. 14](#_Toc63148564)

[Spring Security. 15](#_Toc63148565)

[Basic authentication. 16](#_Toc63148566)

[Definizione di un utente. 18](#_Toc63148567)

[Postman. 20](#_Toc63148568)

[Csrf 22](#_Toc63148569)

[Spring boot MVC: Esempio Applicazione spring boot + thymeLeaf. 22](#_Toc63148570)

[OpenAPI. 22](#_Toc63148571)

[Generazione del catalogo API. 22](#_Toc63148572)

[Pubblicazione del catalogo API. 23](#_Toc63148573)

# **Introduzione**

L’obiettivo del corso è quello di allo studente elementi avanzati di programmazione nel linguaggio java ed in particolare con l’utilizzo d

## Sintesi argomenti trattati:

* startup ambiente
* spring initializer, struttura del progetto
* creazione controller
* architettura del codice
* spring validation
* esercitazioni su REST: controller Base e ereditarietà.
* unit test: scrittura di classi per il test di controller.

## **Requisiti ingresso**

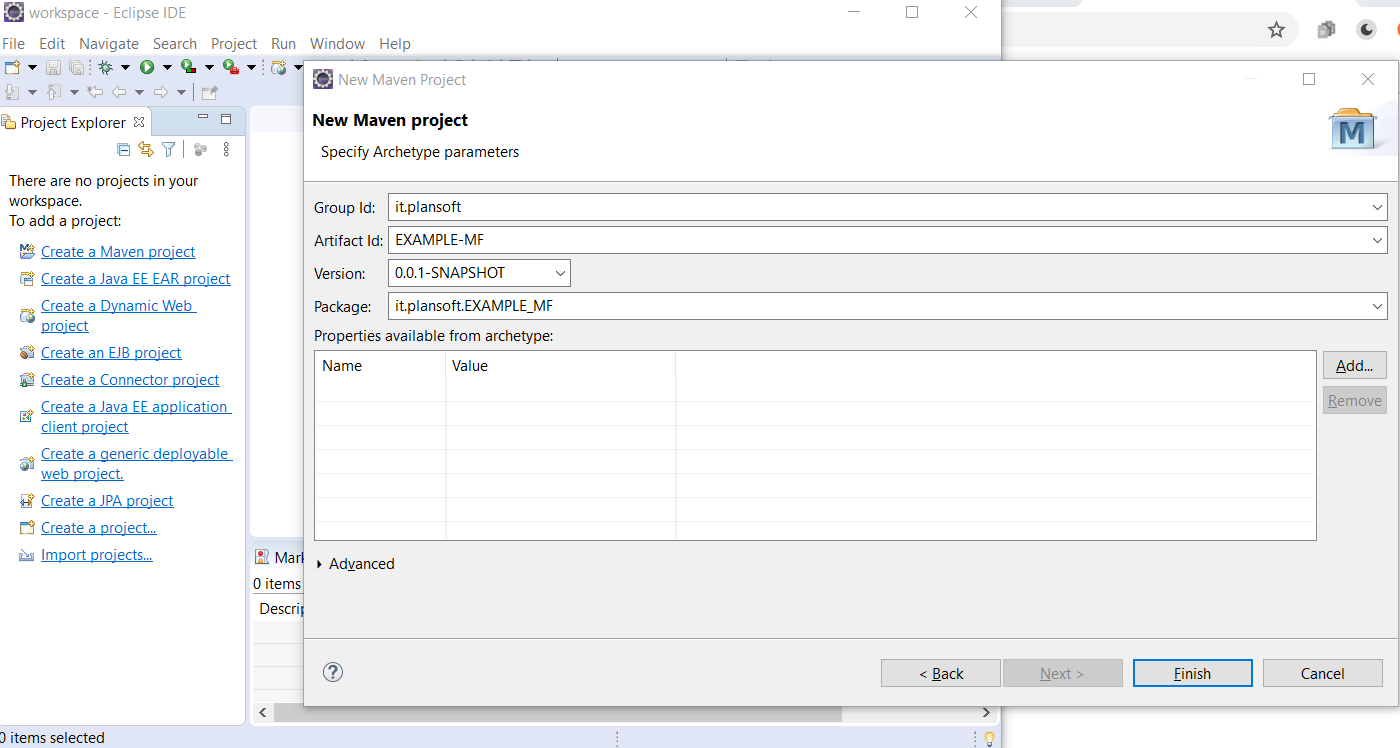
è richiesta la conoscenza java avanzata e l’utilizzo dell’ambiente base di spring boot.

Verranno utilizzate le seguenti tecnologie:

* Spring boot 2.4.0
* JDK 8+ o openJDK 8+
* Maven 3+

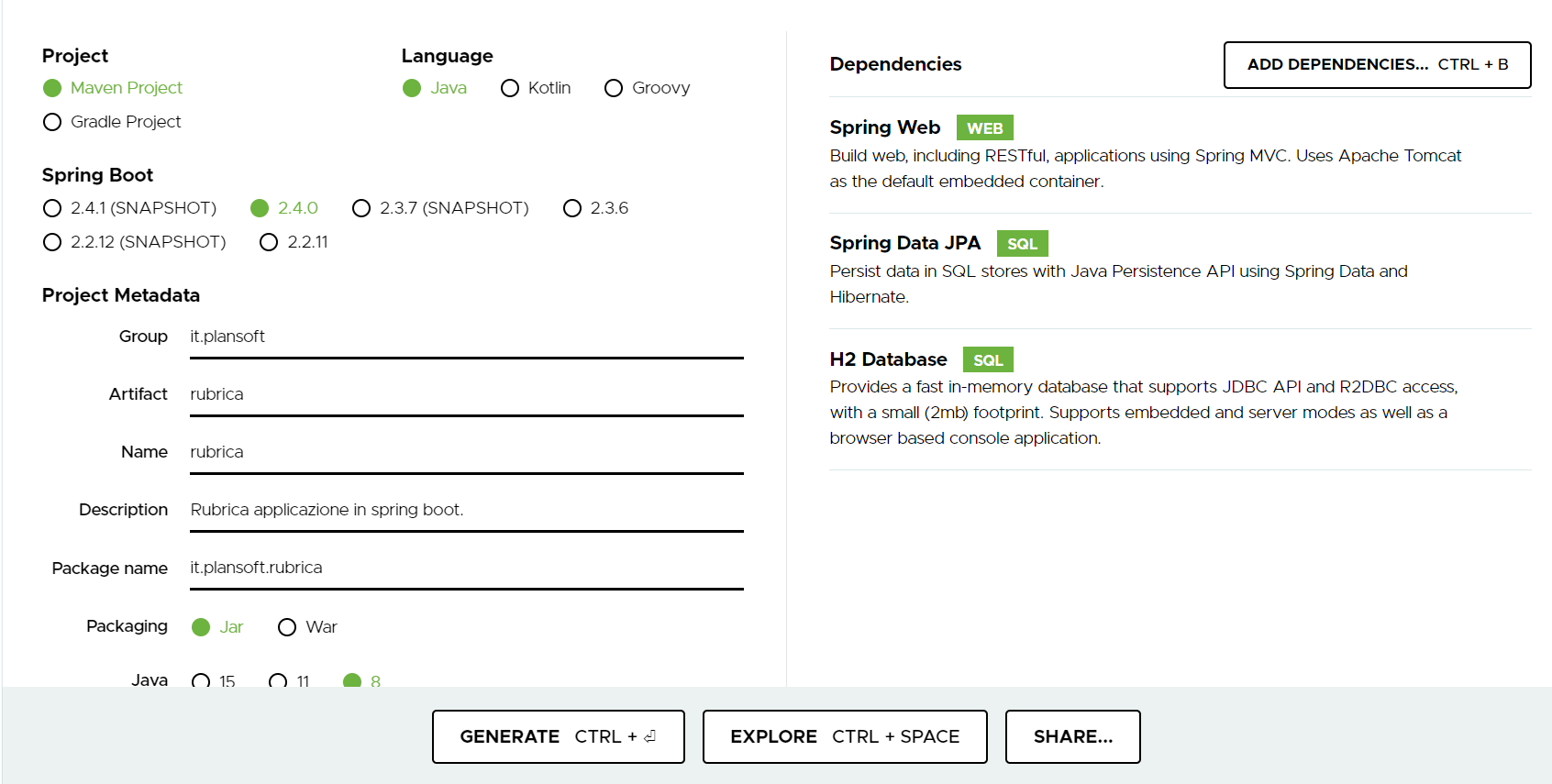
# Startup ambiente.

Per lo start up dell’ambiente fare riferimento al sito ufficiale: <https://spring.io/projects/spring-boot>.



Oppure meglio da spring initializer

<https://start.spring.io/>



Dopo aver impostato i dati sopra riportati:

* progetto
* linguaggio
* versione spring
* package
* versione java
* dependencies: web, jpa, h2

si effettua il download del jar e si importa es. in eclipse come existing maven project, oppure in intellij o visual studio code.

Tramite il comando :

./mvnw clean spring-boot:run

E’ possibile eseguire il progetto da linea di comando o nel terminale di visual studio code.

Oppure da eclipse dalla classe del main eseguire come se fosse un programma stand-alone.

# **Utilizzo di un IDE.**

E’ possibile utilizzare vari IDE per lo sviluppo dei servizi del catalogo API, tra i più importanti:

* eclipse
* visual Studio code
* intellij

Tutti gli ide sopra indicati sono utilizzabili per lo sviluppo sia frontend che backend. Tuttovia si raccomanda di utilizza indifferentemente sia visual studio code che intellij. In entrambi i casi è possibile scrivere codice sia in java che javascript che typescript.

Nel corso verranno utilizzati tutti e tre gli ide di sviluppo, si faccia riferimento ai siti dei produttori.

Es. <https://code.visualstudio.com/docs/java/> per vs code.

# Annotazioni.

In questo esempio andiamo a vedere le annotazioni fondamentali messe a disposizione del framework:

* @Bean – indica che il metodo produce un ‘bean’ per essere utilizzato da spring.
* @Service – indica che la classe è un ‘service’, pertanto una classe contenente la logica applicativa.
* @Repository – la class è un repository, ovvero una astrazione del data access e storage.
* @Configuration – indica che la classe è una configurazione e pertanto contiene la definizione di bean.
* @Controller – si marca per dire che la classe gestisce le richieste web.
* @RequestMapping – mappa la richiesta HTTP con un path, utilizzabile per classe e metodi.
* @Autowired – marca un costruttore, proprietà o un metodo di setter di essere istanziato dalla dependency injection di spring.
* @SpringBootApplication – Abilita l’autoconfigurazione di spring boot ed il componente scanning.

@Component è un generico ‘stereotype’ che viene utilizzato da spring. Permette a spring di trasformare il componente in un bean in fase di auto scan. Le Classi con questo decoratore sono considerate per l’auto-detection in fase di configurazione.

Nota: @Repository, @Service, e @Controller sono specializzazioni di @Component per casi specifici

Ci sono anche annotazioni di tipo @Entity, @Table, @Id, e @GeneratedValue utilizzate per modellizzare le entità.

## Spring Boot: struttura del progetto

Nell’esempio proposto una applicazione Spring Boot application che ritorna datida un H2 database (in memoria) usando Spring Data JPA.

pom.xml

src

├───main

│ ├───java

│ │ └───it

│ │ └───plansoft

│ │ │ Application.java

│ │ ├───controller

│ │ │ RubricaController.java

│ │ ├───model

│ │ │ Rubrica.java

│ │ ├───repository

│ │ │ RubricaRepository.java

│ │ └───service

│ │ RubricaService.java

│ │ IRubricaService.java

│ └───resources

│ │ application.properties

│ │ script.sql (eventuali script sql)

│ ├───static

│ │

│ └───templates

│

│

└───test

└───java

# **Scrittura del primo controller.**

Per la scrittura del primo controller si raccomanda prima di creare una struttura iniziale di progetto, di questo tipo:

* api/controller
* service
* repository
* model

La classe controller che andiamo a scrivere è RubricaController.

Una volta che viene inserita la classe controller è possibile inserire l’annotazione

@RestController

In spring le richieste http sono eseguite da un controller, per questo viene inserita questo tipo di annotazione. Effettuiamo una esposizione di un servizio GET che ritorni una serie di valori, ad esempio per recuperare i dati di rubrica.

Il servizio che andiamo a scrivere ritorna tutti i dati di rubrica senza nessun parametro di ingresso.

Es.

@RestController

**public** **class** RubricaController {

**private** **final** **static** Logger ***log*** = LoggerFactory.*getLogger*(RubricaController.**class**);

@GetMapping("/getAll")

**public** List<String> getAll() {

List<String> messages = Arrays.*asList*("Hello", "World!", "How", "Are", "You");

***log***.info("getAll rubrica {}", messages);

**return** messages;

}

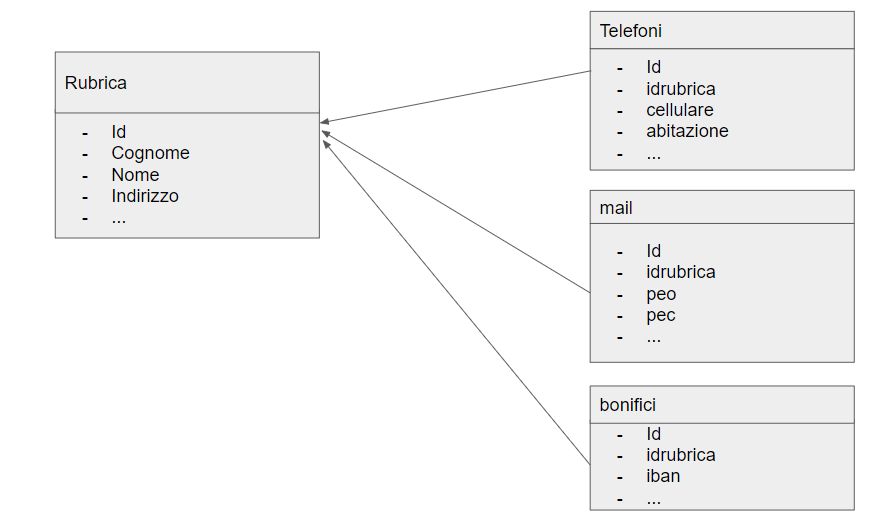
}

Aggiungiamo i seguenti oggetti:

<https://docs.spring.io/spring-boot/docs/2.1.18.RELEASE/reference/html/boot-features-logging.html>

* logging (SLF4j facade per java classi per vari tipi di logging logback, log4j, java logging) http://www.slf4j.org/
* repository (si utilizza un repository con accesso a database tipo H2 in memoria)
* costruttore (per injection del repository o tramite autowired)

Nel frattempo costruiamo una serie di oggetti di entity, per mappare la seguente struttura tabellare:



Es. rubrica

/\*\*

\* entity di rubrica

\*

\* **@author** Giuseppe Grosso

\*

\*/

@Entity

**public** **class** Rubrica {

**private** @Id @GeneratedValue Long id;

**private** String cognome;

**private** String nome;

**private** String indirizzo;

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

**public** String getCognome() {

**return** cognome;

}

**public** **void** setCognome(String cognome) {

**this**.cognome = cognome;

}

**public** String getNome() {

**return** nome;

}

**public** **void** setNome(String nome) {

**this**.nome = nome;

}

**public** String getIndirizzo() {

**return** indirizzo;

}

**public** **void** setIndirizzo(String indirizzo) {

**this**.indirizzo = indirizzo;

}

}

# Verbo http.

All’interno del controller andiamo ad esporre i servizi rispettando la sintassi e i verbi tipici del protocollo http.

In particolare i seguenti:

* GET
* POST
* PUT
* PATCH
* DELETE

Il verbo GET viene utilizzato per effettuare una richiesta che non implica la modifica del dato sul backend, es. lista o singolo oggetto. La richiesta non ha un payload ma viene inserita nella url i parametri di richiesta, può essere di tipo path param o query param es.

Path: <http://server:port/api/rubrica/1>

Query param: <http://server:port/rubrica?id=1&>...

Controindicazione: non è possibile passare un oggetto come payload, ecco perché molte volte si vedono passare anche per le richieste di GET metodi con verbo POST.

Il verbo POST viene utilizzato per effettuare un inserimento di un nuovo oggetto all’interno dei dati, es. un nuovo oggetto.

PUT e PATCH sono sinonimi e servono entrambi per la modifica di dati esistenti e che sono stati recuperati tramite il verbo GET. Il verbo PUT prevede che nel payload venga passato tutto l’oggetto che deve essere aggiornato, mentre nel verbo PATCH solo il dato che è stato modificato.

Es.

PUT /users/1

{

"username": "ggrosso",

"email": "giuseppe.ing.grosso@gmail.com" // nuova email

}

In formato patch:

PATCH /users/1

{

"email": "giuseppe.ing.grosso@gmail.com" // nuova email

}

Come visto la differenza sta nel fatto che nel verbo put si passa tutto l’oggetto anche se non modificato mentre nel verbo patch solo i dati che sono stati modificati e che devono essere aggiornati.

Quando si usa il put si assume che viene inviato l’entità completa di tutte le sue proprietà e che devono essere aggiornate completamente, mentre nel PATCH si aggiorna solo i dati passati per la modifica, occorre pertanto rileggere l’oggetto e aggiornare le sole proprietà passate nel payload.

# Produzione e consumo dei dati.

L’annotazione di @Produces è usata per specificare il MIME media types o rappresenta una risorsa che può essere mandata al client.

Se si imposta l’annotation @produces a livello di classe tutti i metodi della risorsa possono produrre uno specifico MIME type di default. Altrimenti applicato a livello di metodo, sovrascrivono il comportamento a livello di classe. Questo tipo di annotazione si espone a livello di catalogo nell’annotation @RequestMapping

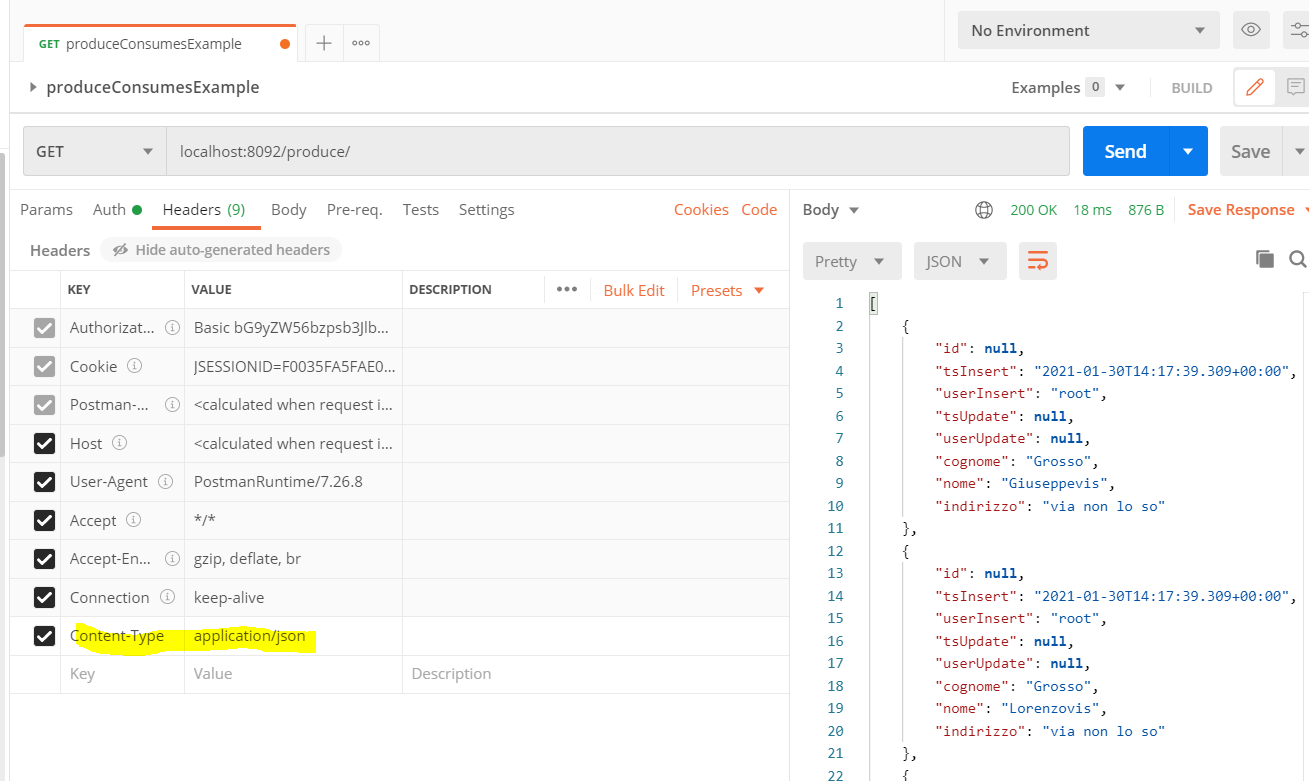
Vedi documentazione ufficiale.

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html>

Esempio:

@RestController  
@RequestMapping(value = "/produce", consumes = {"application/json", "application/xml"}, produces = {"application/json", "application/xml"})  
public class ProducesConsumesController {  
  
 private static final List<Rubrica> *RUBRICA* = Arrays.*asList*(  
 new Rubrica("Grosso", "Giuseppevis", "via non lo so"),  
 new Rubrica("Grosso", "Lorenzovis", "via non lo so"),  
 new Rubrica("Grosso", "Danielevis", "via non lo so")  
 );  
  
  
 @GetMapping(path = "/")  
 public List<Rubrica> getDemo()  
 {  
 return *RUBRICA*;  
 }  
}

Su postman:



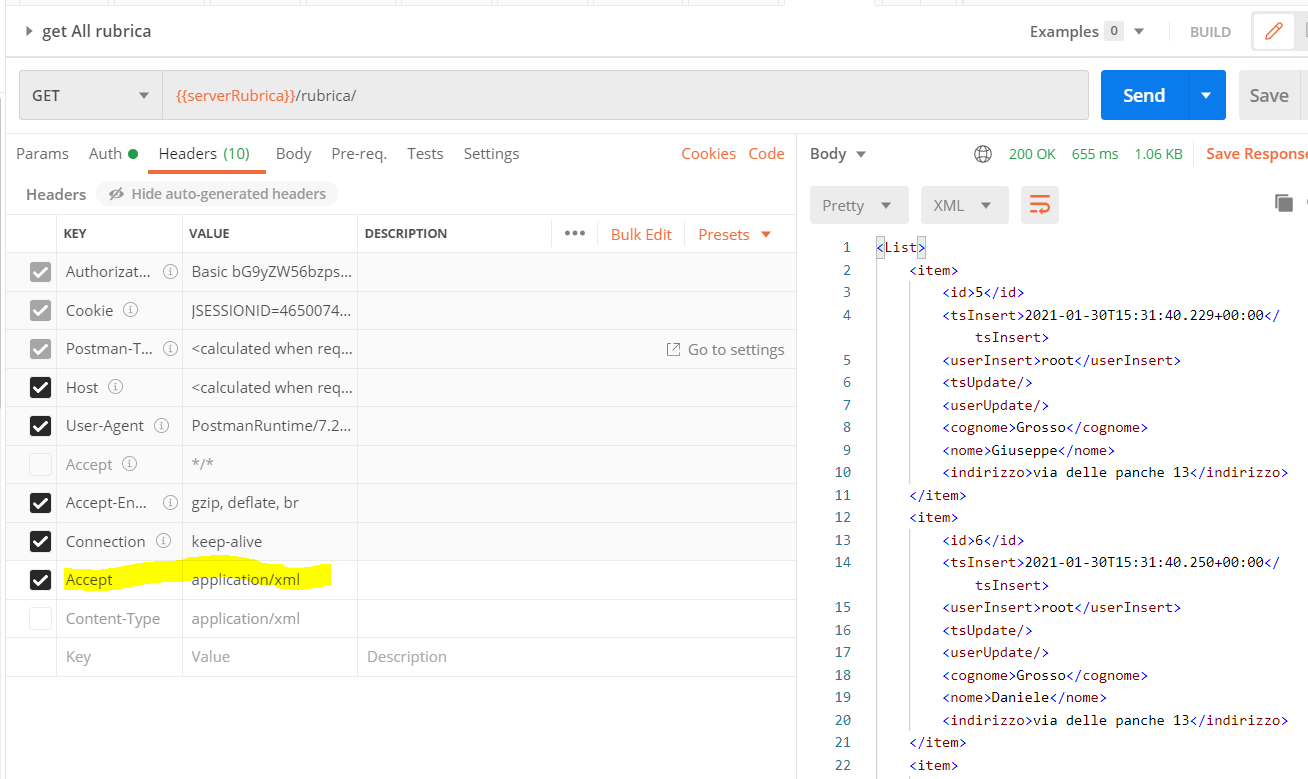
Come libreria inserire nel pom la seguente libreria:

<!-- xml -->  
<dependency>  
 <groupId>com.fasterxml.jackson.dataformat</groupId>  
 <artifactId>jackson-dataformat-xml</artifactId>  
</dependency>

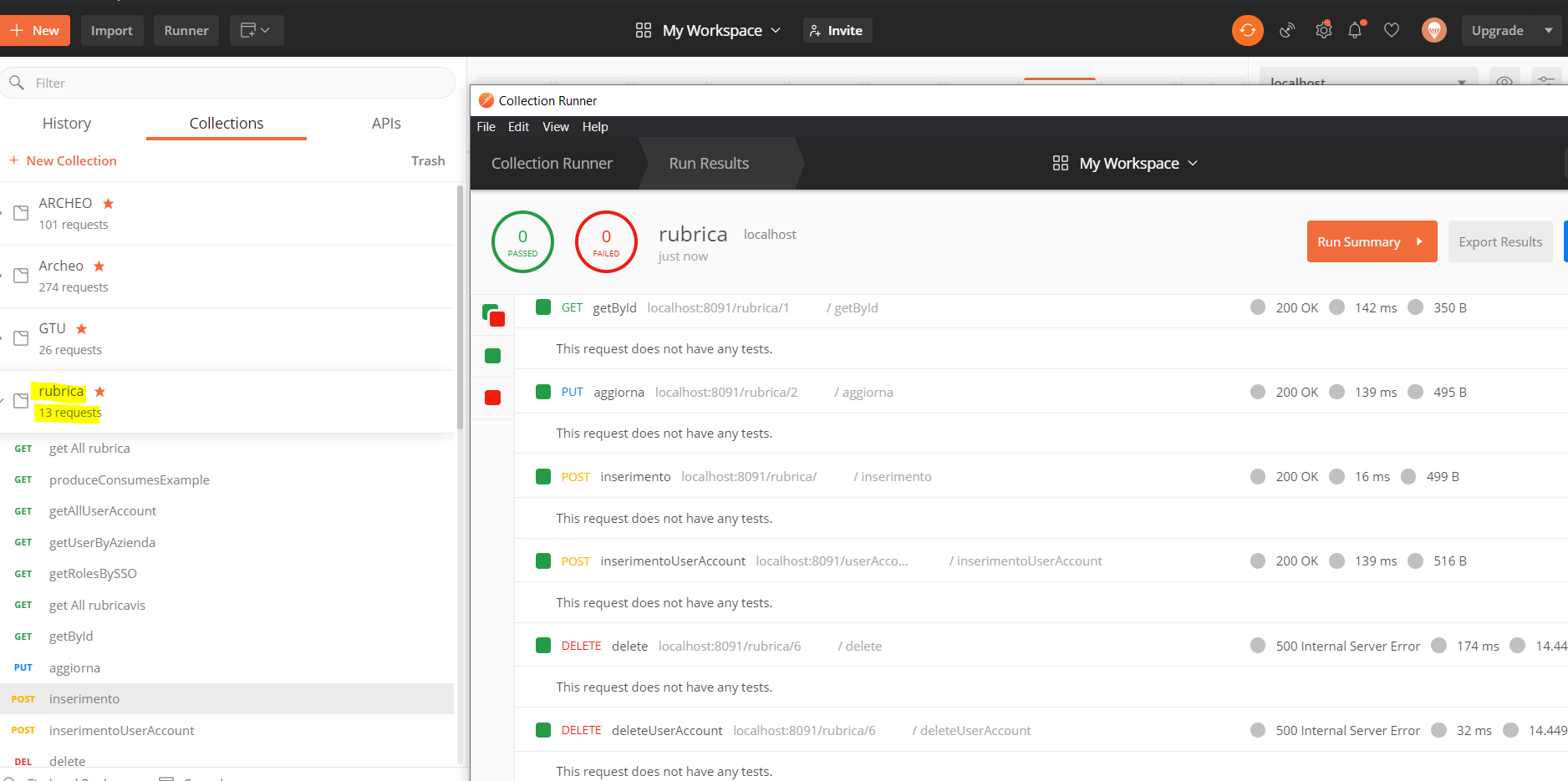
Inserita la libreria si potranno ottenere i dati in formato xml, altrimenti si ottiene un errore di tipo 406 Not acceptable.

Utilizzo

* Content-Type indica il content type per il body di richieste POST e PUT.
* Accept indica che il content type aspettato nella response per le richieste GET.



PostMan esecuzione batteria di test:



# Service.

Lo strato di service è uno strato che permette di poter definire quella che viene chiamata business logic, ovvero lo strato di logica che esula dall’esposizione dei dati attraverso il protocollo http.

All’interno dello strato di service andiamo a inserire tutta una serie di logiche tipiche dell’oggetto che stiamo trattando.

Es. all’interno del service RubricaService andiamo ad inserire i metodi CRUD di recupero dei dati.

Mentre nell’oggetto LoginService andremo ad inserire alcune logiche tipiche della login:

* recupero dell’utente
* recupero dei ruoli/autorities
* creazione del token
* modifica password etc

# Dependendy injection.

E’ possibile utilizzare ogni metodo standard di spring per definire un beans e la loro injection come dipendenza. Es. utilizzando @ComponentScan @Autowired con costruttore, properties o metodo.

Occorre strutturare il codice inserendo in radice l’applicazione con il component scan pertanto ogni componente sia @Component, @Service, @Repository, @Controller etc saranno registrati come bean ed utilizzati mediante dependency injection.

Di seguito un esempio per iniettare un service all’interno del costruttore.

Es.

@Autowired  
public RubricaController(RubricaService service) {  
 super(service);  
}

….

*/\*\*  
 \* Indicates a {****@link*** *Configuration configuration} class that declares one or more  
 \* {****@link*** *Bean @Bean} methods and also triggers {****@link*** *EnableAutoConfiguration  
 \* auto-configuration} and {****@link*** *ComponentScan component scanning}. This is a convenience  
 \* annotation that is equivalent to declaring {****@code*** *@Configuration},  
 \* {****@code*** *@EnableAutoConfiguration} and {****@code*** *@ComponentScan}.  
 \*  
 \** ***@author*** *Phillip Webb  
 \** ***@author*** *Stephane Nicoll  
 \** ***@author*** *Andy Wilkinson  
 \** ***@since*** *1.2.0  
 \*/*@SpringBootApplication  
public class RubricaApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(RubricaApplication.class, args);  
 }  
  
}

Alternativamente è possibile utilizzare la seguente sintassi:

**package** com.example.myapplication;

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

**import** org.springframework.context.annotation.ComponentScan

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

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

**@Configuration(proxyBeanMethods = false)**

**@EnableAutoConfiguration**

**@Import({ MyConfig.class, MyAnotherConfig.class })**

**public** **class** **Application** {

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

SpringApplication.run(Application.class, args);

}

}

In questo esempio l’applicazione è uguale ad altre applicazioni di SpringBoot ad eccezione per il fatto che @component e @ConfigurationProperties non sono automaticamente risolte, pertanto occorre importarle esplicitamente con @import.

# Esternalizzazione configurazione.

<https://docs.spring.io/spring-boot/docs/1.0.0.RC5/reference/html/boot-features-external-config.html>

/\*\*

\* lettura del file di configurazione application.config.properties.

\*

\*/

@Configuration

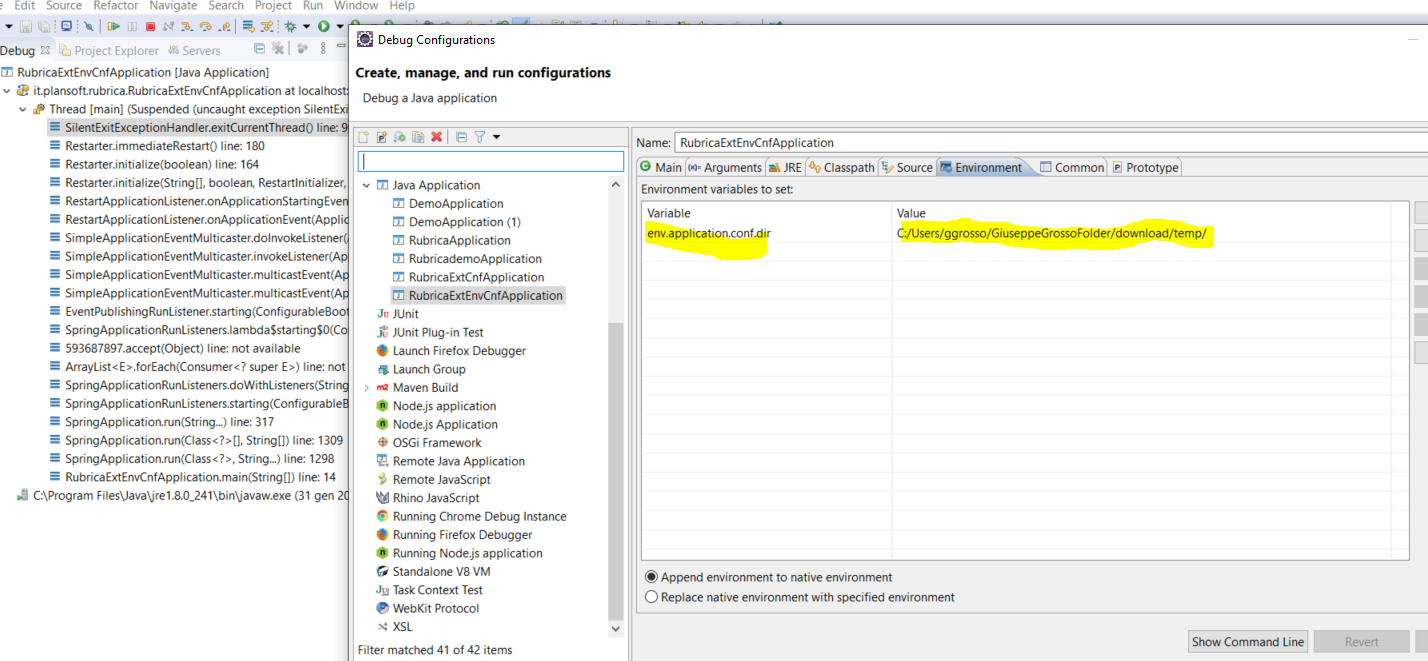
@Configuration(proxyBeanMethods = false)  
@EnableAutoConfiguration // prendo la properties in --spring.config.location=file:/C:/Users/ggrosso/GiuseppeGrossoFolder/download/temp/  
// configurazione custom  
//@PropertySource(ignoreResourceNotFound = true, value = {  
// "file:///C:/Users/ggrosso/GiuseppeGrossoFolder/download/temp/application.ext.properties"//, "file:///${env.application.conf.dir}/application.config.properties"  
// })  
// altre configurazioni.

E’ possibile indicare un file esternalizzato di configurazione che non fa parte del classpath e permette pertanto di poter deployare l’applicazione in contesti diversi es. sit (sviluppo), uat (preproduzione), produzione.

E’ necessario utilizzare una variabile di environment che viene settata es. all’interno dell’ambiente di run di eclipse, intellij etc.

Vedi specifiche su <https://www.eclipse.org/openj9/docs/env_var/>

Es.



# Internalization.

Una volta creata una semplice applicazione spring boot è possibile inserire l’internalizzazione aggiungendo le proprietà necessarie.

Creare la cartella lang all’interno di resource ed inserire al suo interno il file res.proprieties.

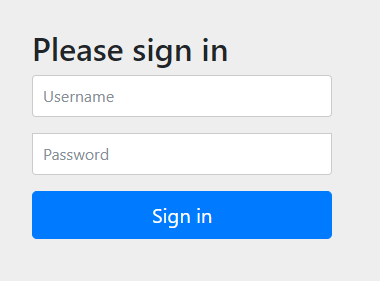
# ExceptionHandler.

# Spring Security.

Spring mette a disposizione un plugin per poter poter pubblicare in maniera sicura il nostro catalogo API, occorre semplicemente inserire all’interno del nostro file di pom la seguente dipendenza:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>

Una volta inserita la dipendenza, rigenerato il progetto con le dipendenze maven è possibile accedere al catalogo, in questo caso una login ci chiederà utente e password.



I riferimenti per utente e password da inserire li troviamo nel log di intellij, eclipse o visual Studio code.

ES: Using generated security password: 8ccc2be1-6a66-48ce-b2f7-d9cb02576597

Utente : user

Password: 8ccc2be1-6a66-48ce-b2f7-d9cb02576597

E successivamente è possibile entrare all’interno del catalogo api.

Se verifichiamo la chiamata http effettuata successivamente vedremo che adesso nella request viene inserito il seguente tag:

**Cookie:**

JSESSIONID=821F39C1FCD81FFAC39898CCE8089617

Che permette al nostro servizio di backend di validare la richiesta controllando che sia stato effettuata la login.

Di seguito il payload effettuato per la login:

1. **username:**

user

1. **password:**

8ccc2be1-6a66-48ce-b2f7-d9cb02576597

1. **\_csrf:**

85750a33-cefb-4b90-96eb-d4c050348b80

Per effettuare il logout andare sull’indirizzo base es.

http://localhost:{port}/logout

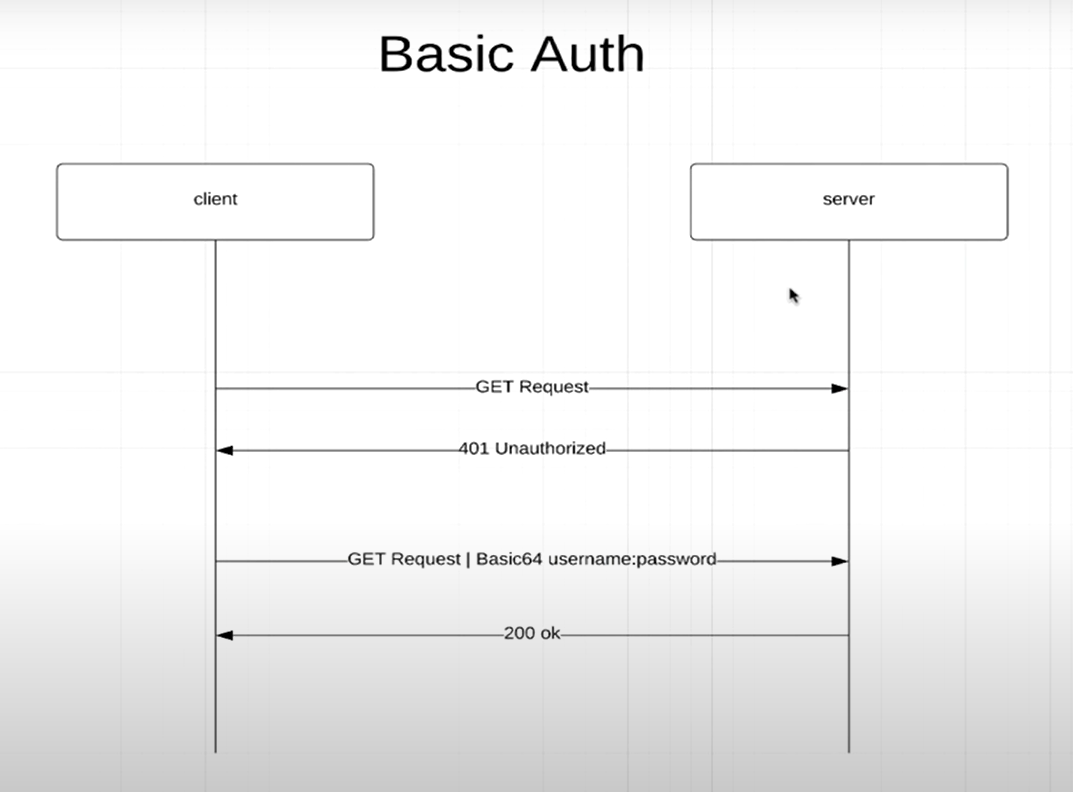
Di seguito vediamo i tipi di autenticazione che in generale esistono:

* Basic authentication
* form based autentication
* jwt
* provider esterno

## Basic authentication.

vediamo i concetti di Basic Authentication in generale.

Di seguito uno schema per la basic authentication:



Quello che vediamo è che se il client effettua una richiesta qualsiasi es. in formato GET riceve in risposta uno stato dal server di tipo 401 (unauthorized), se invece effettuiamo la stessa richiesta http aggiungendo nella header della richiesta con codifica base64 username:password (hard credential) vediamo che il backend ci risponde con stato 200 fornendoci la risposta alla richiesta effettuata.

Pertanto non è necessario effettuare una login, ma occorre passare **ad ogni** richiesta fatta al server la codifica in base64 di user:password all’interno della header della richiesta stessa (GET, POST, PUT, DELETE etc).

Vediamo cosa dobbiamo fare per configurare springSecurity per utilizzare la basic authentication invece che la form base authentication.

Occorre scrivere un componente di configurazione e abilitarlo al webSecurity:

Es.

@Configuration  
@EnableWebSecurity  
public class ApplicationSecurityConfig extends WebSecurityConfigurerAdapter {  
  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http  
 .authorizeRequests()  
 .anyRequest()  
 .authenticated()  
 .and()  
 .httpBasic();  
 }  
}

Nella ui non abbiamo più il form di autenticazione ma ci viene proposto il popup integrato del browser con utente e password (che riprendiamo come in precedenza dal log) e successivamente avremo le richieste tutte autenticate, come evidenziato di seguito:

1. **Authorization:**

Basic dXNlcjpjOTllMmM1NS1mY2E5LTRmZjMtOTNmMS1mZmQ5ZjBkOGMwOWY=

1. **Connection:**

keep-alive

1. **Cookie:**

JSESSIONID=B1F0E727BFEF62B577E75CBFF1638C60

Attenzione non esiste un endpoint per effettuare il logout.

## Definizione di un utente.

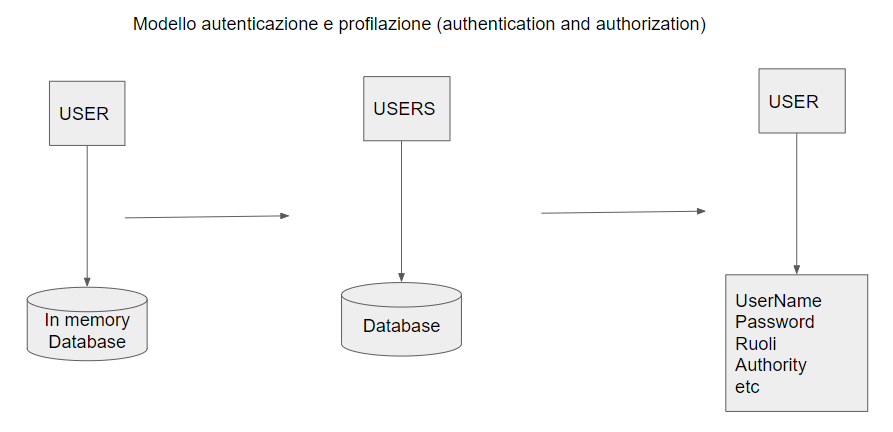
In un’applicazione web solitamente per l’accesso al sito è necessario che gli utenti siano censiti all’interno di un database e successivamente al login è possibile accedere alle risorse.

Simuliamo questo aspetto definendo in spring boot attraverso l’oggetto UserDetailService, che permette di potersi definire gli utenti che possono accedere all’applicazione.

@Override  
@Bean  
protected UserDetailsService userDetailsService() {  
 // recupero l'utente dal database.  
 UserDetails user = User.*builder*()  
 .username("giuseppe")  
 .password("password")  
 .roles("ADMIN") // ROLE\_ADMIN  
 .build();  
  
 return new InMemoryUserDetailsManager(user);  
}

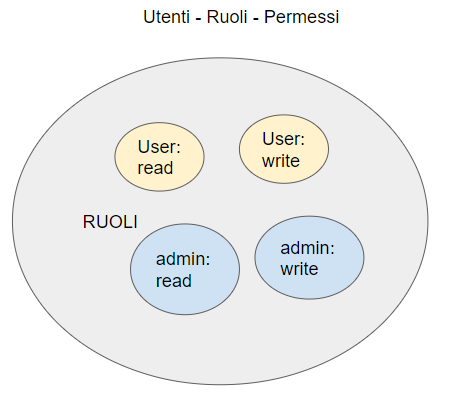
Attenzione la password deve essere cryptata altrimenti il framework ritorna una eccezione, pertanto è obbligatorio utilizzare un tipo di cryptaggio della password.

Utenti, Ruoli e permessi



Utenti - Ruoli - Permessi

RUOLI

* User:write
* User:read
* admin:write
* admin:read

Andiamo ad inserire gli oggetti per la definizione dei ruoli e dei permessi.

public enum ApplicationUserPermission {  
 *WRITE*("write"),  
 *READ*("read"),  
 *ALL*("all");  
  
 private final String permission;  
  
 ApplicationUserPermission(String permission) {  
 this.permission = permission;  
 }  
  
 public String getPermission() {  
 return permission;  
 }  
  
 public static Set<ApplicationUserPermission> getUserPermission()  
 {  
 return new HashSet<ApplicationUserPermission>(Arrays.*asList*(*WRITE*));  
 }  
  
 public static Set<ApplicationUserPermission> getAdminPermission()  
 {  
 return new HashSet<ApplicationUserPermission>(Arrays.*asList*(*WRITE*, *READ*));  
 }  
}

public enum ApplicationUserRole {  
 *USER*(new HashSet<>()), // ApplicationUserPermission.getUserPermission()  
 *ADMIN*(ApplicationUserPermission.*getAdminPermission*());  
  
 private final Set<ApplicationUserPermission> permissions;  
  
 ApplicationUserRole(Set<ApplicationUserPermission> permissions) {  
 this.permissions = permissions;  
 }  
  
 public Set<ApplicationUserPermission> getPermissions() {  
 return permissions;  
 }  
}

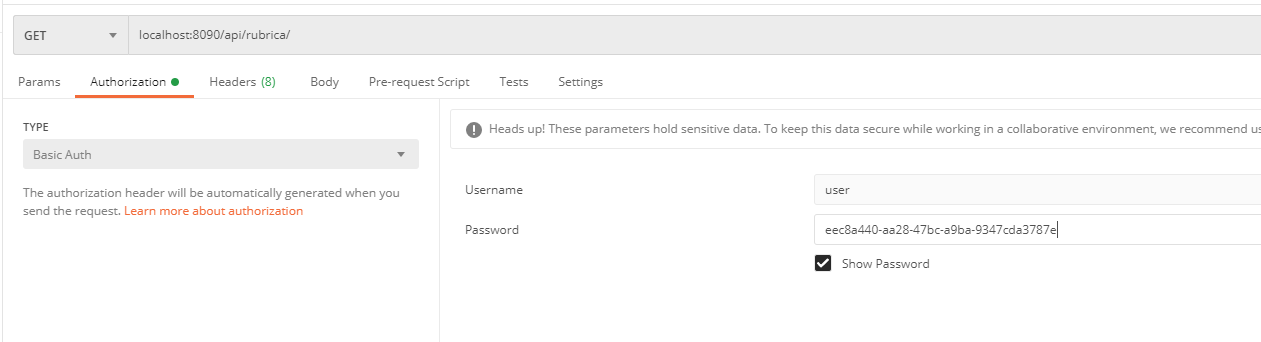
Andiamo adesso a proteggere le risorse definite con i ruoli e i permessi definiti tramite le due classi.

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http  
 .authorizeRequests()  
 // white list  
 .antMatchers("/", "/index", "/css", "/js/\*").permitAll()  
 // protect all  
 .antMatchers("/api/\*\*").hasAnyRole(ApplicationUserRole.*USER*.name(), ApplicationUserRole.*ADMIN*.name())  
 .anyRequest()  
 .authenticated()  
 .and()  
 .httpBasic();  
}

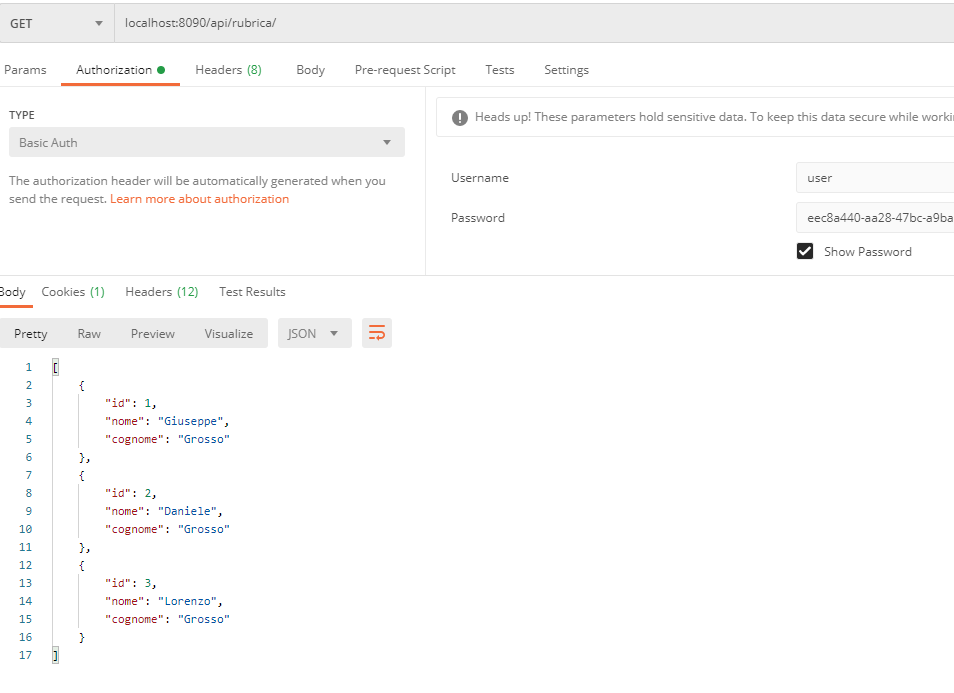
## Postman.

Vediamo come effettuare le richieste http con un client es. postman.

E’ necessario impostare all’interno del tab authorization il tipo di autenticazione, nel nostro caso viene impostato Basic Auth



E successivamente si effettua la chiamata in GET: in questo modo otteniamo i dati richiesti.



Protezione dei dati e white List.

Tramite spring security è possibile inoltre impostare un filtro per permettere di avere alcuni contenuti non protetti, ad esempio la pagina di index, o il catalogo API (vedi catalogo API), occorre pertanto modificare la classe ApplicationSecurityConfig ed aggiungere le eccezioni per permettere l’accesso non protetto.

http  
 .authorizeRequests()  
 // white list  
 .antMatchers("/", "/index", "/css", "/js/\*")  
 .permitAll()  
 // protect all  
 .anyRequest()  
 .authenticated()  
 .and()  
 .httpBasic();

In particolare con antMachers e l’elenco degli url non protetti ed inoltre il comando permittAll.

Inoltre all’interno della cartella resources/static viene aggiunta la pagina html index.html.

Questa risorsa (statica) contiene la pagina di benvenuto del sito web. Es.

<h1>benvenuto in Rubrica: spring Security</h1>

Preauthorize("hasRole, hasAnyRole, hasAuthoriti, hasanyauthority”)

@PreAuthorize("hasAnyRoles('ROLE\_ADMIN','ROLE\_USER')")

inserisco il metodo di preauth direttamente nel metodo.

Per put, delete, insert

@PreAuthorize("hasAuthoritiy('WRITE')")

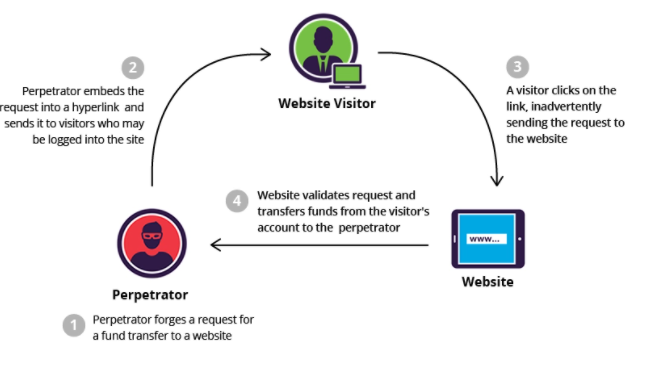
Attenzione: spring boot può essere utilizzato con vari framework di front-end tra cui utilizzando il plugin thymeLeaf avere il pattern MVC e permettere di scrivere applicazioni complete (frontend + backend). Oppure con i più moderni framework utilizzati attualmente es. Angular.io, reactJs, vue.js.

@PreAuthorize possibile inserire il tag direttamente sul metodo invece che nella configurazione del metodo visto in precedenza.

# Csrf: Cross site request Forgery.

Il cross site request forgery, anche conosciuto come XSRF, è un vettore di attacco che permette al browser web di eseguire un’azione non voluta da parte di un utente loggato. E’ un’azione molto grave e può determinare trasferimenti non autorizzati, cambi di password e furto di dati, incluso i dati di sessione.

Tipicamente viene messo in atto usando email o link che attraggono la vittima a inviare richieste ad un server su cui è autenticato in precedenza, così facendo l’attaccante riesce ad ottenere la risposta indirizzata a lui piuttosto che all’utente loginato.



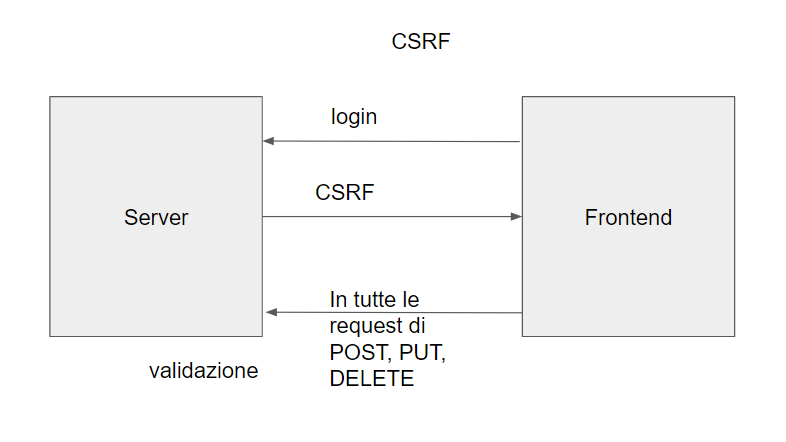
Sito Owas: <https://owasp.org/www-community/attacks/csrf>

Sito spring: <https://docs.spring.io/spring-security/site/docs/3.2.0.CI-SNAPSHOT/reference/html/csrf.html>

Pertanto un attaccante copia la richiesta e simula la richiesta come se proveniente dall’utente loginato.

Per questo motivo su spring security il filtro su csrf() è abilitato di default, questo interviene su tutte le richieste di POST, PUT, DELETE.

Di seguito lo schema delle richieste effettuate con protezione CSRF abilitato in spring boot:



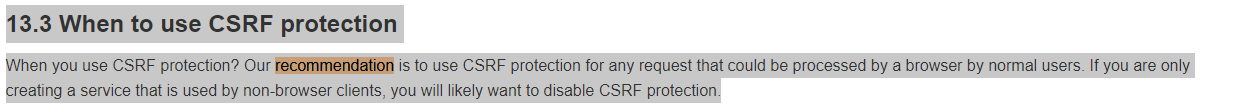
Pertanto se abilitato occorre utilizzare l’anti forgery altrimenti le richieste di POST, PUT, DELETE non viene evasa (vedi raccomandazioni di owasp).

<https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/06-Session_Management_Testing/05-Testing_for_Cross_Site_Request_Forgery>

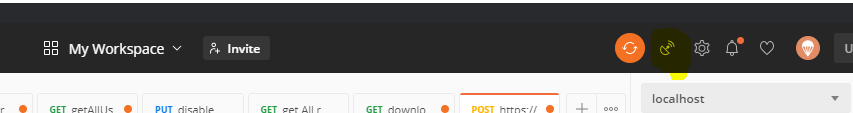
Le raccomandazioni di spring security sono quelle di abilitare il CSRF in caso di utilizzo di browser, nel caso di utilizzo di altro frontend non è necessario Es. in postman non è necessario o su app nativa.

Raccomandazioni di spring security:

<https://docs.spring.io/spring-security/site/docs/3.2.0.CI-SNAPSHOT/reference/html/csrf.html>



Per simularlo con postman occorre installare un plugin (interceptor) che permette di recuperare e visionare il token (cookie: XSRF-TOKEN) e di aggiungerlo manualmente ad ogni richiesta con verbi POST, PUT, DELETE.



Quindi una volta intercettato il token nei cookie XSRF-TOKEN occorre aggiungerlo nella header di ogni request nel formato

X-XSRF-TOKEN: xxxxx se la passo ottengo la risposta e non Forbidden (403).

Se utilizziamo un client angular, vue.js, react etc è possibile recuperare il token in oggetto ed aggiungerlo ad ogni richiesta di POST, PUT, DELETE.

Per la generazione del token in spring security occorre inserire il codice:

.csrf().disable() //per abilitarlo basta commentare il codce.

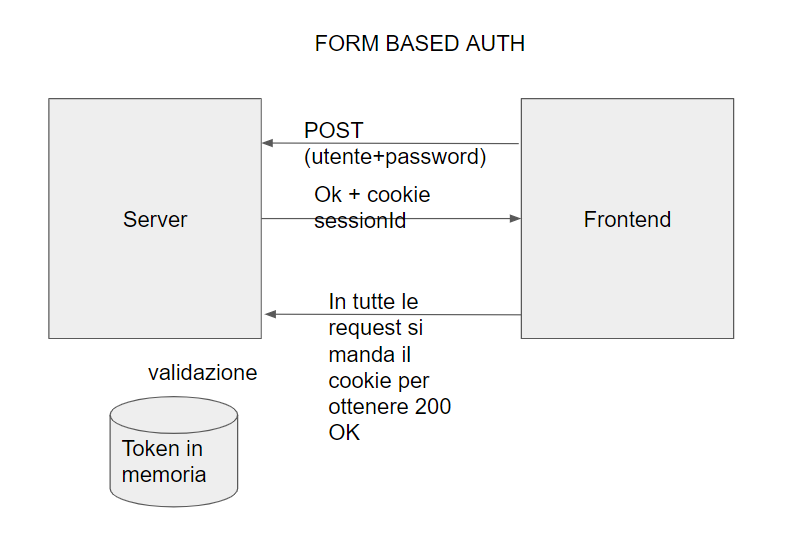
Lo disabilito se utilizzo le api con postman, altrimenti lo abilito.

# FormBasedAuthentication.

In spring security oltre al tipo di autenticazione basic è possibile utilizzare il tipo form based che permette invece di utilizzare la basic auth quella basata su form.

Il vantaggio dell’utilizzo di questo tipo di autenticazione è che è possibile effettuare il logout dall’applicazione.

Mentre con la basic auth, che è più semplice e veloce, non permette di effettuare il logout



Per abilitare la form base Auth occorre sostituire la httpBasic() con la formLogin().

.httpBasic();

sostituito con

.formLogin()

Viene richiesta l’immissione delle credenziali attraverso un form (standard) personalizzabile a valle della richiesta vengono validate le credenziali ed inviato il cookie con il token. Solitamente i browser intercettano e recuperano i token per le successive chiamate. Il cookie di sessioni vale di default 30 minuti.

Lo store dei cookie si fa solitamente in database fisico, nel caso di utilizzo di un database di memoria il session Id e di tipo volatile e pertanto ad ogni riavvio viene rigenerato.

# Custom formogin: integrazione thymeleaf.

Maven:

<!-- thymeleaf ed internazionalizzazione -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-thymeleaf</artifactId>  
 </dependency>

Di seguito una pagina di esempio per la login:

<html lang="en"><head>  
 <meta charset="utf-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">  
 <meta name="description" content="">  
 <meta name="author" content="">  
 <title>Please sign in</title>  
 <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-/Y6pD6FV/Vv2HJnA6t+vslU6fwYXjCFtcEpHbNJ0lyAFsXTsjBbfaDjzALeQsN6M" crossorigin="anonymous">  
 <link href="https://getbootstrap.com/docs/4.0/examples/signin/signin.css" rel="stylesheet" crossorigin="anonymous">  
</head>  
<body>  
<div class="container">  
 <form class="form-signin" method="post" action="/login">  
 <h2 class="form-signin-heading">Custom Login</h2>  
 <p>  
 <label for="username" class="sr-only">Username</label>  
 <input type="text" id="username" name="username" class="form-control" placeholder="Username" required="" autofocus="">  
 </p>  
 <p>  
 <label for="password" class="sr-only">Password</label>  
 <input type="password" id="password" name="password" class="form-control" placeholder="Password" required="">  
 </p>  
 <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>  
 </form>  
</div>  
</body></html>

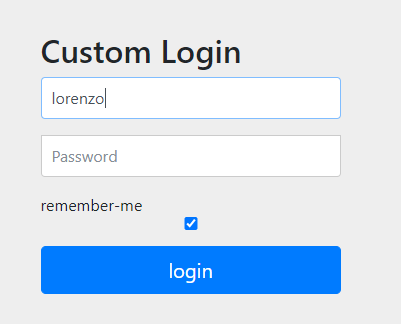
Oppure con la variante al posto dei tag h2 e p una table

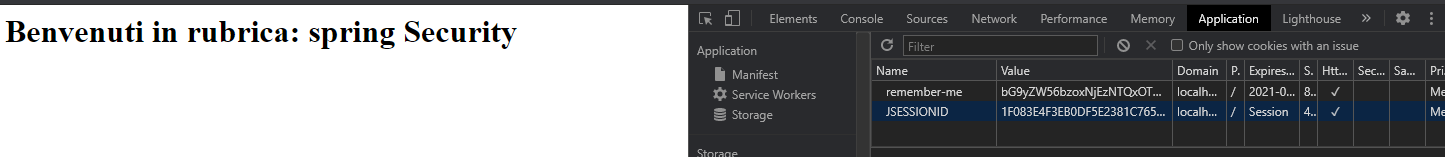
<table>  
 <tr>  
 <td>User:</td>  
 <td><input type='text' name='username' value=''></td>  
 </tr>  
 <tr>  
 <td>Password:</td>  
 <td><input type='password' name='password' /></td>  
 </tr>  
 <tr>  
 <td>Remember Me:</td>  
 <td><input type="checkbox" name="remember-me" /></td>  
 </tr>  
 <tr>  
 <td><input name="submit" type="submit" value="submit" /></td>  
 </tr>  
</table>

Mentre la classe applicationSecurity

// .httpBasic()  
.formLogin().loginPage("/login")  
.permitAll().defaultSuccessUrl("/rubrica", true)  
.and().rememberMe()

Mentre la UI diventa:



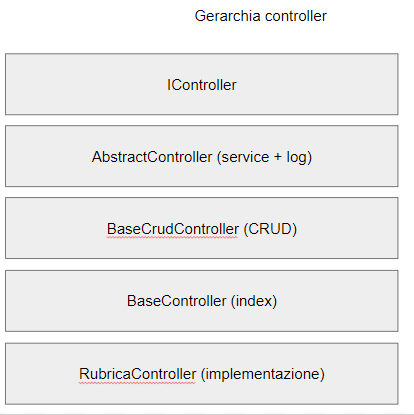


Si nota che adesso tra i cookie ci sono il sessionID ed inoltre il remember-me che mi permette di utilizzare un cookie nominato remember-me per estendere e recuperare il nome utente all’interno del client.

Se si vuole estendere il remember-me a 21 giorni aggiungere:

.formLogin().loginPage("/login")  
.permitAll().defaultSuccessUrl("/rubrica", true)  
.and().rememberMe()  
// variante remeber-me  
.tokenValiditySeconds((int)TimeUnit.*DAYS*.toSeconds(21))  
.key("keySecret")

Per generalizzare la login viene aggiunta la seguente gerarchia:



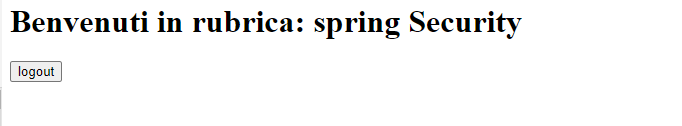
## Logout.

Vogliamo in questo caso pulire la session-ID e il remember-me non appena effettuiamo il logout della sessione.

All’interno della ApplicationSecurityConfig

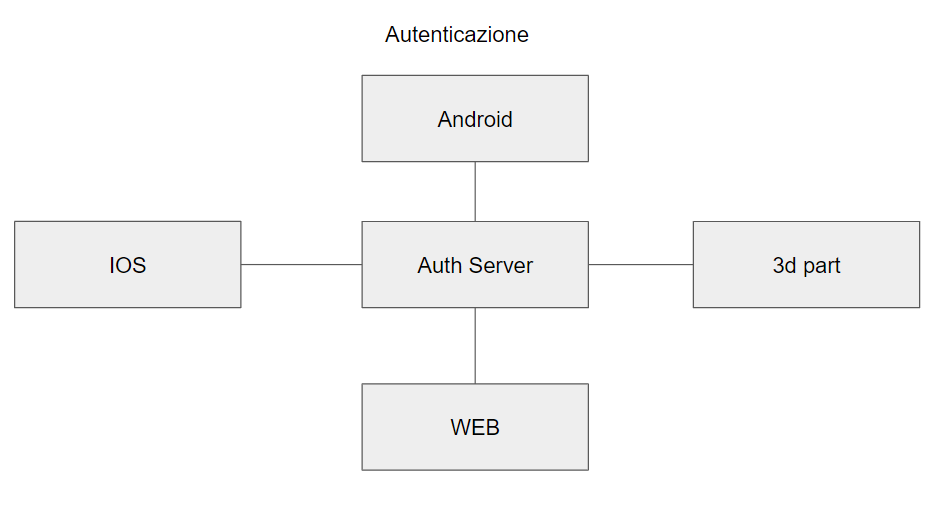
// logout  
.and().logout()  
.logoutUrl("/logout").clearAuthentication(true)  
.invalidateHttpSession(true)  
.deleteCookies("JSESSIONID", "remember-me")  
.logoutSuccessUrl("/login")

Inseriamo inoltre un tasto button per la logout all’interno della pagina vista di rubrica.html



Autenticazione dei sistemi.

Pensiamo al sistema di autenticazione come ad un apparato che debba autenticare più dispositivi eterogenei fra di loro ad esempio:



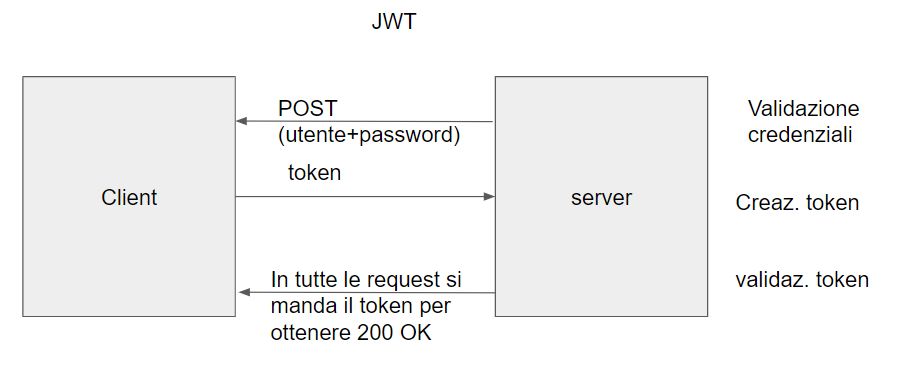
Dispositivi mobili con app custom, web application, e altre applicazioni di terze parti.

Per essere il più possibile astratti dalle applicazioni che chiedono una autenticazione conviene utilizzare un prodotto sul mercato noto con il nome di JWT (Json Web Token) <https://jwt.io/>

Motivazione per utilizzo jwt:

* + fast velocità di esecuzione elevata
* + stateless (non ha bisogno di appoggiarsi ad un database)
* + utilizzabile per più servizi
* - compromette le chiamate
* - non si ha visibilità degli utenti loginati
* - token non può essere rubato.

## Schema funzionamento JWT.



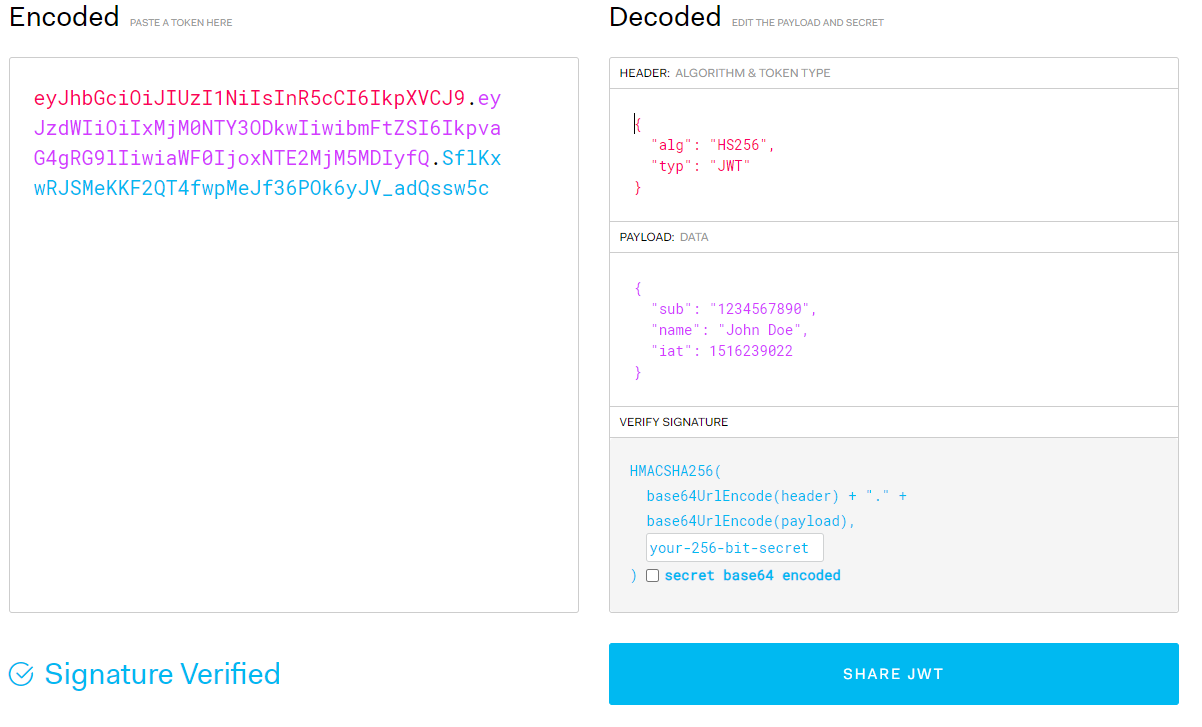
Come si vede il client invia le credenziali di accesso (utente e password = hard credential) il server le valida e genera un token che passa in risposta con stato http = 200 ( eventualmente insieme ai dati dell’utente censito nel db).

Successivamente il client per ogni chiamata (per tutti i verbi) nella header della richiesta allega il token con il formato:

Authorization: Bearer {JWTTOKEN}

## Token encoded

Vediamo adesso come è fatto il token:



Come vediamo il token è formato da tre parti

* Header
* Payload
* Signature

Nella parte di header si ha l’algoritmo ed il tipo, mentre nel payload ci sono i dati su cui è basato il token es. utente, nome etc.

Nell’ultima parte c’e’ la verifica della signature di header e payload fatta con una chiave segreta.

# OpenAPI.

OpenApi è un progetto che permette di descrivere in maniera appropriata un catalogo API.

Con questo strumento è possibile effettuare la progettazione dell’API andando a descrivere il catalogo sia in formato json che in formato yaml.

Il descrittore è possibile scriverlo attraverso il seguente sito online:

<https://editor.swagger.io/>

# Generazione del catalogo API.

Per effettuare la generazione del catalogo delle API è sufficiente aggiungere al pom.xml la seguente libreria:

        <!-- documentazione openapi -->

        <dependency>

            <groupId>org.springdoc</groupId>

            <artifactId>springdoc-openapi-ui</artifactId>

            <version>1.5.1</version>

        </dependency>

Inoltre aggiungendo i seguenti parametri al file application.properties

server.port=8090

# generazoine della documentazione

springdoc.api-docs.path=/api-docs

# pubblicazone della documentazione

springdoc.swagger-ui.path=/swagger-ui.html

E’ possibile cambiare la porta di default al server tomcat (default 8080), cambiare il path del catalogo API di default ed infine la url di consultazione del catalogo swagger.

# Pubblicazione del catalogo API.

Una volta aggiunta la libreria e inserite le proprietà viste in precedenza nel file application.properties è possibile effettuare lo start del progetto e successivamente andare al seguente indirizzo da un browser:

<http://localhost:8090/swagger-ui/index.html?configUrl=/api-docs/swagger-config>

