

Microservizi REST e architetture SOA

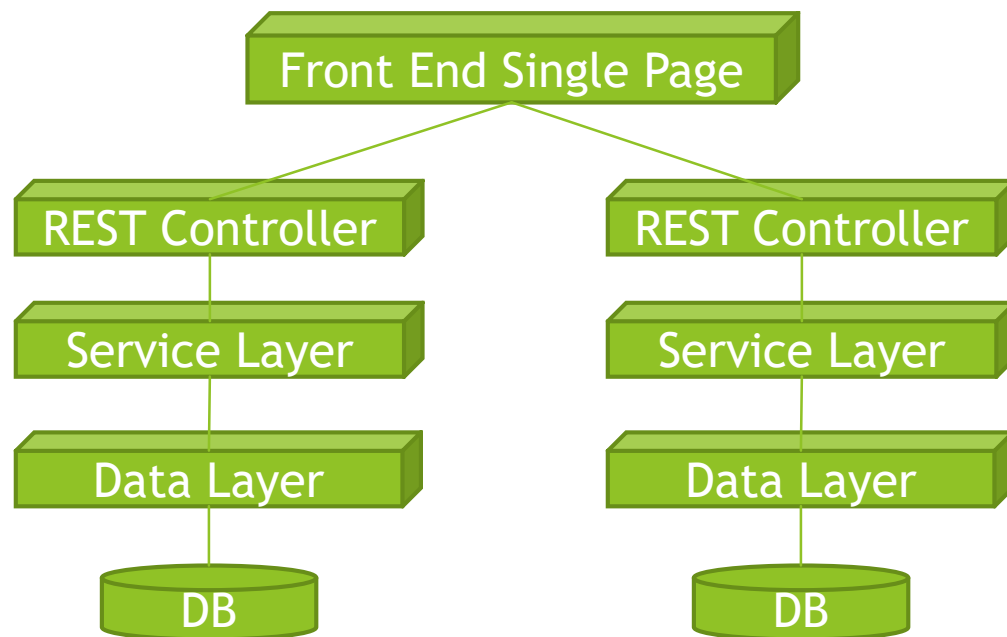
SpringBoot

Argomenti Trattati

- ▶ Micro Servizi
- ▶ Spring Boot
- ▶ JPA
- ▶ JWT e Cookies
- ▶ Servizi REST
- ▶ Pattern MVC
- ▶ Basi delle Single Page Application
- ▶ AJAX
- ▶ JQuery

Obbiettivo

- Realizzare una Web App completa che utilizzi servizi REST, credenziali e persistenza



Microservizi versus Applicazioni monolitiche

- ▶ UNA SVOLTA NELL'APPROCCIO ALLA PROGETTAZIONE DEL SOFTWARE

APPLICAZIONE MONOLITICA

- ▶ Le applicazioni monolitiche svolgono un insieme di funzioni, dette servizi, a vantaggio dell'utente.
- ▶ Tutti i servizi dell'applicazione sono implementati all'interno della stessa applicazione che risulta esserne il contenitore.



Differenze tra app monolitica e micro servizi

- ▶ L'app singola copre diverse responsabilità
- ▶ I microservizi coprono singole responsabilità indipendentemente ad autonomamente
- ▶ La build ed il deploy passano da singoli a multipli, riducendo il rischio di impatti di una modifica sul resto dell'applicazione
- ▶ L'app monolitica viene esposta su una unica porta esposta dall'Application Server che la ospita, come ad esempio Tomcat, Jboss, Wildfly o WebSphere, mentre i microservizi vengono esposti ciascuno su differente porta o addirittura su differenti macchine, facilitando la distribuzione delle funzionalità. Con SpringBoot ad esempio si utilizza un Tomcat od un jetty Embedded.

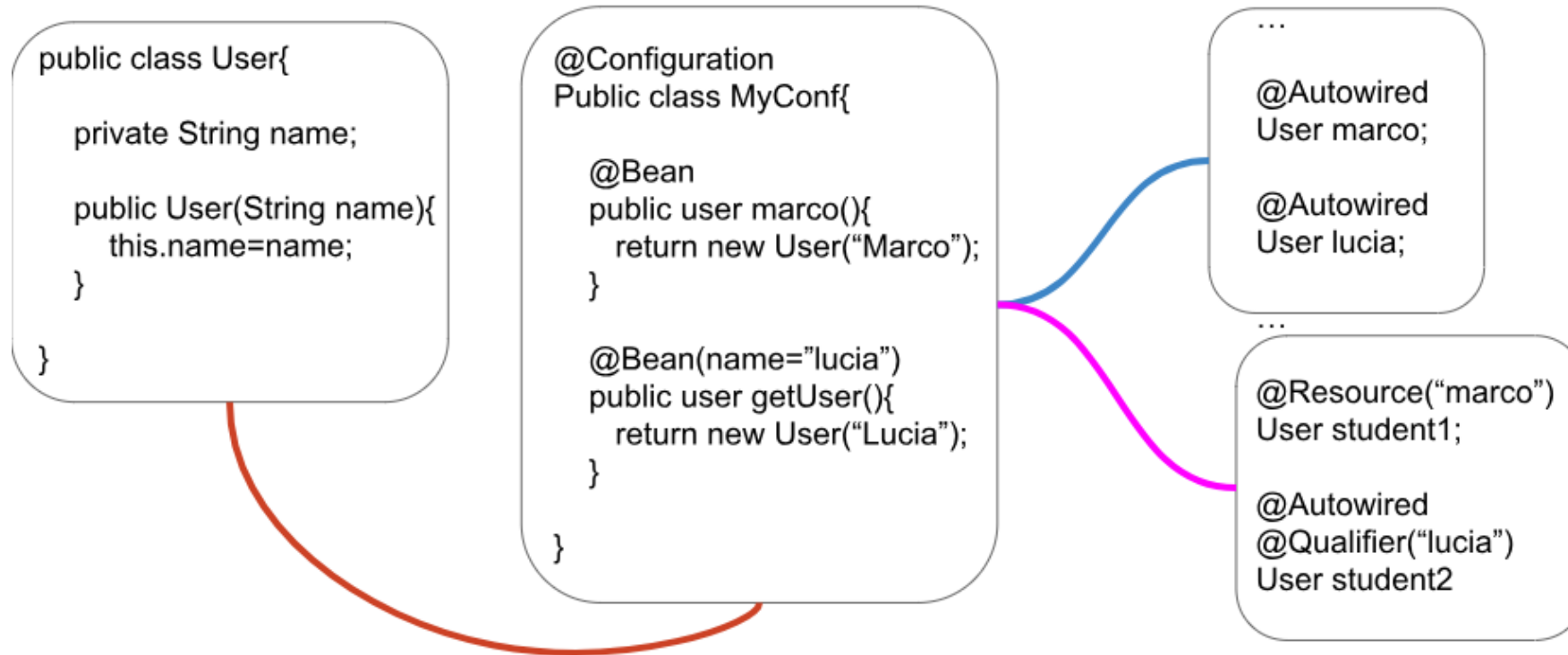
WAR versus FAT JAR

- ▶ I micro servizi hanno ad esempio con SpringBoot un HTTP Listener embedded
- ▶ Le WEB application classiche sono tipicamente impacchettate in un War o EAR, ed inserite all'interno della cartella di deploy dell'Application Server, quando il server verrà avviato gestirà le chiamate http o https e le dirotterà verso l'applicativo spaccettato e istanziato al suo interno.
- ▶ I micro servizi invece vengono impacchettati in un fat Jar o uber jar che viene buildato con maven o gradle utilizzando Spring Boot il quale vi inserisce anche un application server embedded come Tomcat o Jetty
- ▶ Le web application hanno la necessità di avere al loro interno un file di deploy il web.xml mentre i micro servizi no
- ▶ Le web application stanno in ascolto sulla porta dell'application server che le contiene mentre più microservizi stanno in ascolto ciascuno su di una porta diversa

Pillole di Spring

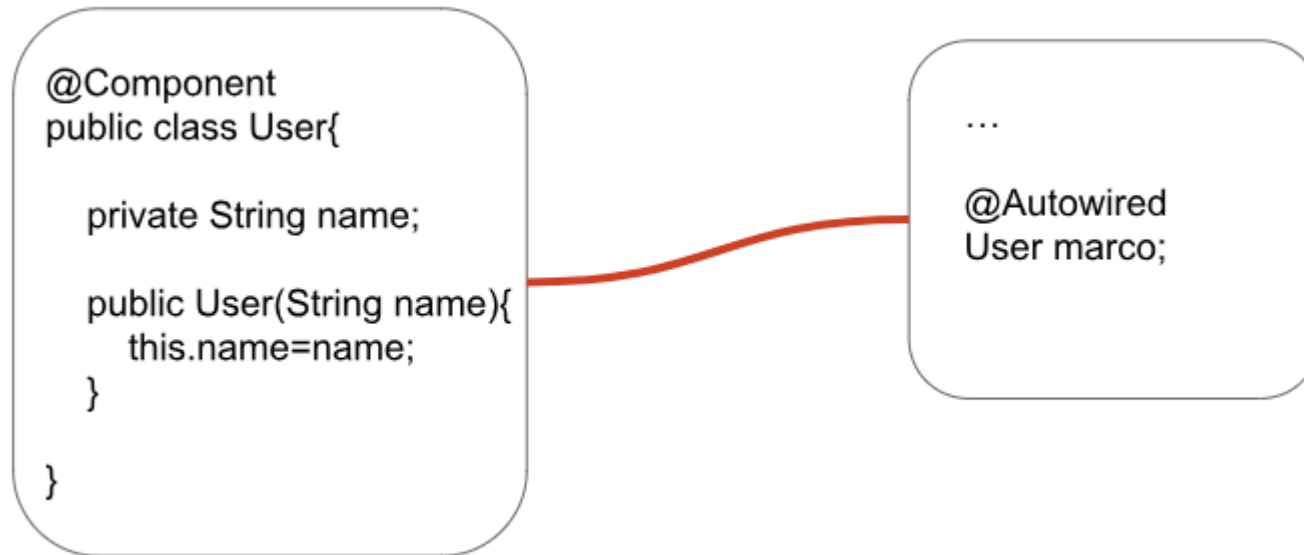
- ▶ In Java per istanziare un oggetto si utilizza l'operatore new che lancia un metodo costruttore della classe dell'oggetto in questione.
- ▶ Buona prassi è separare la creazione degli oggetti da cui una classe dipende dalla classe stessa, seguendo pattern di design come Factory, Builder o come fa Spring la dependency injection o inversion of control.
- ▶ Sarà Spring ad istanziare le classi per noi ed a gestirne il ciclo di vita in modo trasparente, permettendoci di concentrarci meglio sulla logica di business piuttosto che sulla gestione dell'allocazione degli oggetti.
- ▶ Spring lavora con beans o fagioli (POJO), i quali vengono iniettati all'interno delle nostre classi attraverso le classi di configurazione o direttamente nel codice con delle apposite annotation che derivano dalla annotation @Component.

Beans dichiarati nel file di configurazione Configuration



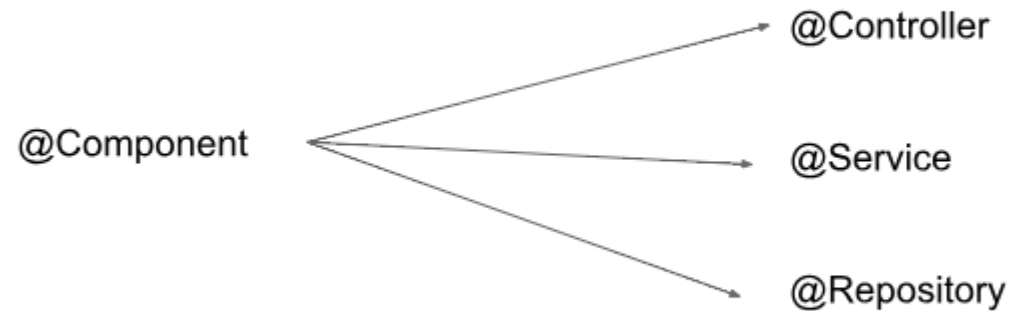
Beans innestati via Component e Autowired

Il metodo più diffuso sfrutta maggiormente Spring, permettendo di scrivere meno codice



Component e le sue specializzazioni

L'annotation `@Component` e le sue estensioni permettono di iniettare quelli che per un micro servizio sono le parti principali:



Pattern MVC

- ▶ MODEL, VIEW, CONTROLLER pattern per strutturare meglio un applicativo
- ▶ Il modello è costituito dalla rappresentazione dei dati, e dalle operazioni CRUD su di essi, nel caso di Spring Boot e di un database relazionale come Oracle, PostgreSQL o MYSQL, tale lo strato è costituito dalle tabelle e dalle Entity che le rappresentano lato back end, mentre le operazioni su di esse tipicamente vengono implementate con dei Repository con Spring Boot che fungono da DAO (Data Access Object).
- ▶ La vista è la rappresentazione dei dati all'utente, come questo li visualizzerà, per cui può essere costituito da pagine JSP, da un front end in React, Angular etc, ma anche, nel caso dei microservizi, dai JSON o XML di risposta alle chiamate REST o SOAP
- ▶ Il controller è invece quella parte del software che si occupa delle logiche di business, come ad esempio la gestione dei JWT o dei cookies per la gestione della sessione utente in un contesto stateless come l'HTTP, nel contesto dei micro servizi e SOA il controller gestisce le richieste e le risposte http restando in ascolto su di una porta ad un dato URL, mentre la logica di business viene implementata dai Service che si appoggeranno al Repository quando necessiteranno di operazioni sul model.

MVC con Spring nei Microservizi o Architetture SOA

- ▶ Il backend estraendo i dati da DB ritorna degli oggetti utilizzando un ORM che si occupa di convertire i dati relazionali in oggetti, ma i servizi dovranno restituire dei dati in formato JSON, per questo Spring utilizza la libreria Jackson che converte un oggetto in formato stringa JSON
- ▶ Per rappresentare i record delle tabelle si utilizzano le entity che rimappano le colonne del DB e sono dei POJO annotati con @Entity
- ▶ Il controller layer di Spring, implementato con il componente Controller si appoggia ai servizi implementati con il componente Service
- ▶ Dove i controller sono delle classi annotate con @Controller e i servizi con @Service

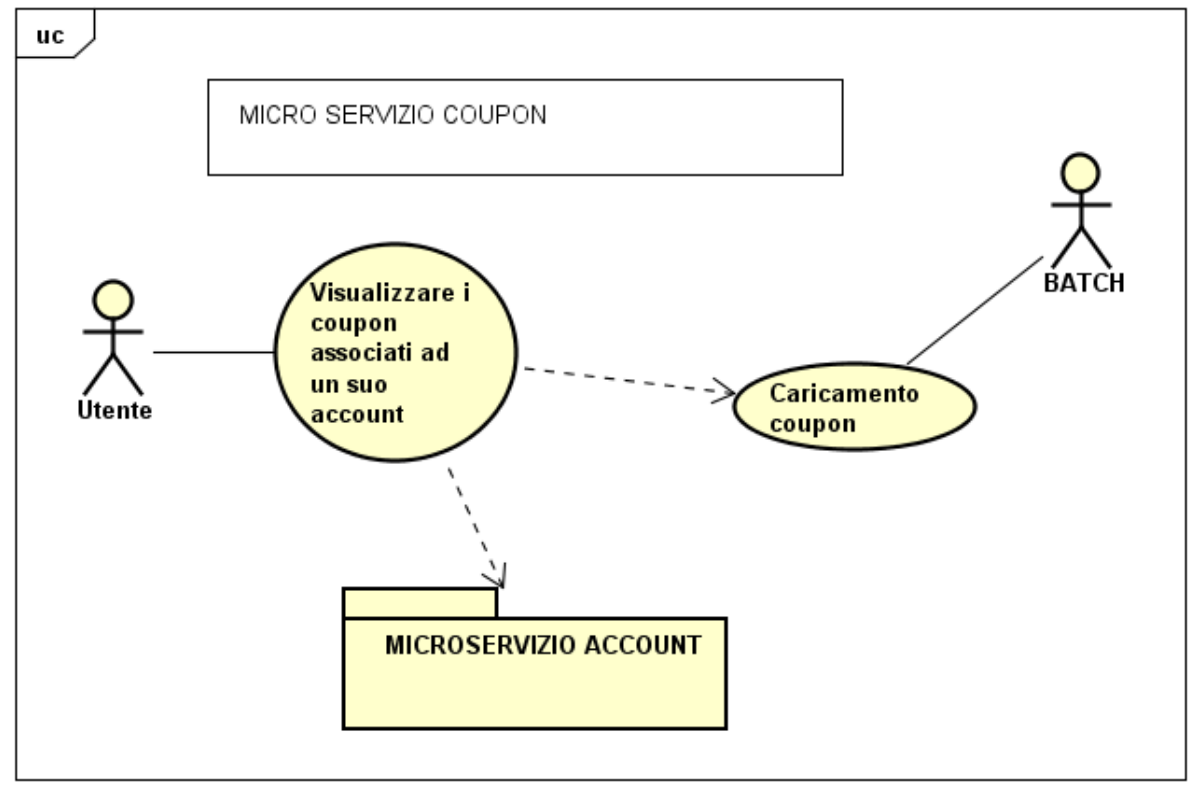
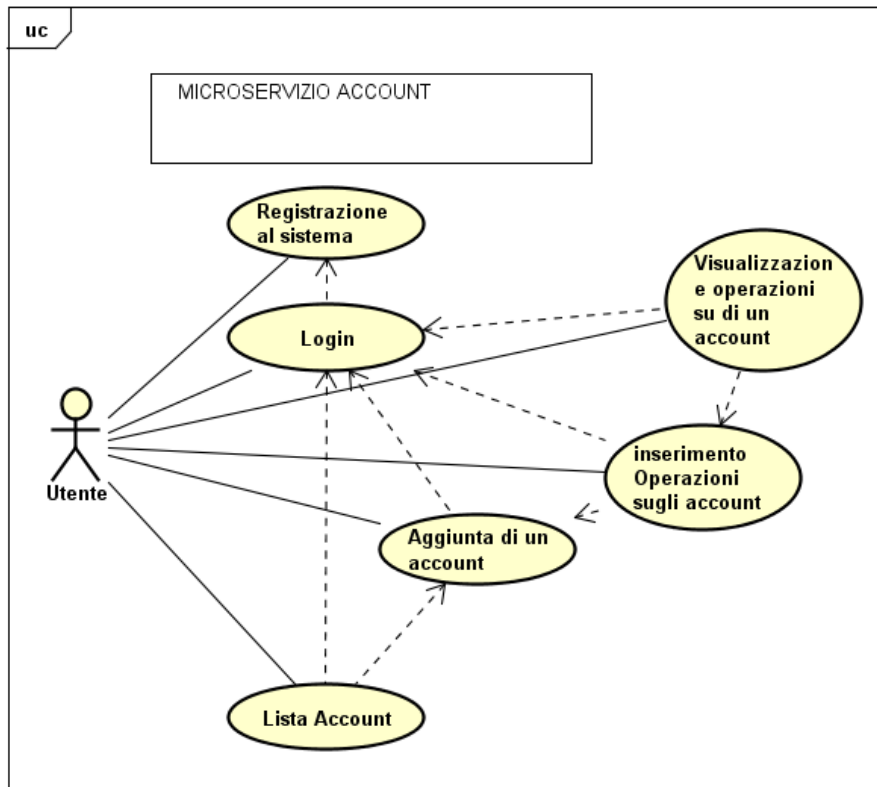
Maven, build tool

- ▶ Un build tool permette di scaricare le librerie da cui dipende il nostro progetto, di compilarlo e impacchettarlo, in un war o ear per un web application SOA o in un fat jar nel caso di micro servizi con Spring Boot, inoltre permette di lanciare anche gli unit test successivamente alla compilazione
- ▶ Il tutto sempre nell'ottica di concentrarsi sulla logica di business piuttosto che sul download manuale delle librerie, sulla compilazione con Javac e sulla creazione del pacchetto, risparmiando tempo e denaro
- ▶ Maven gestisce le dipendenze in un file XML ed è integrato con i principali IDE di sviluppo come Eclipse o IntelliJ.

HTTP, Cookies versus JWT

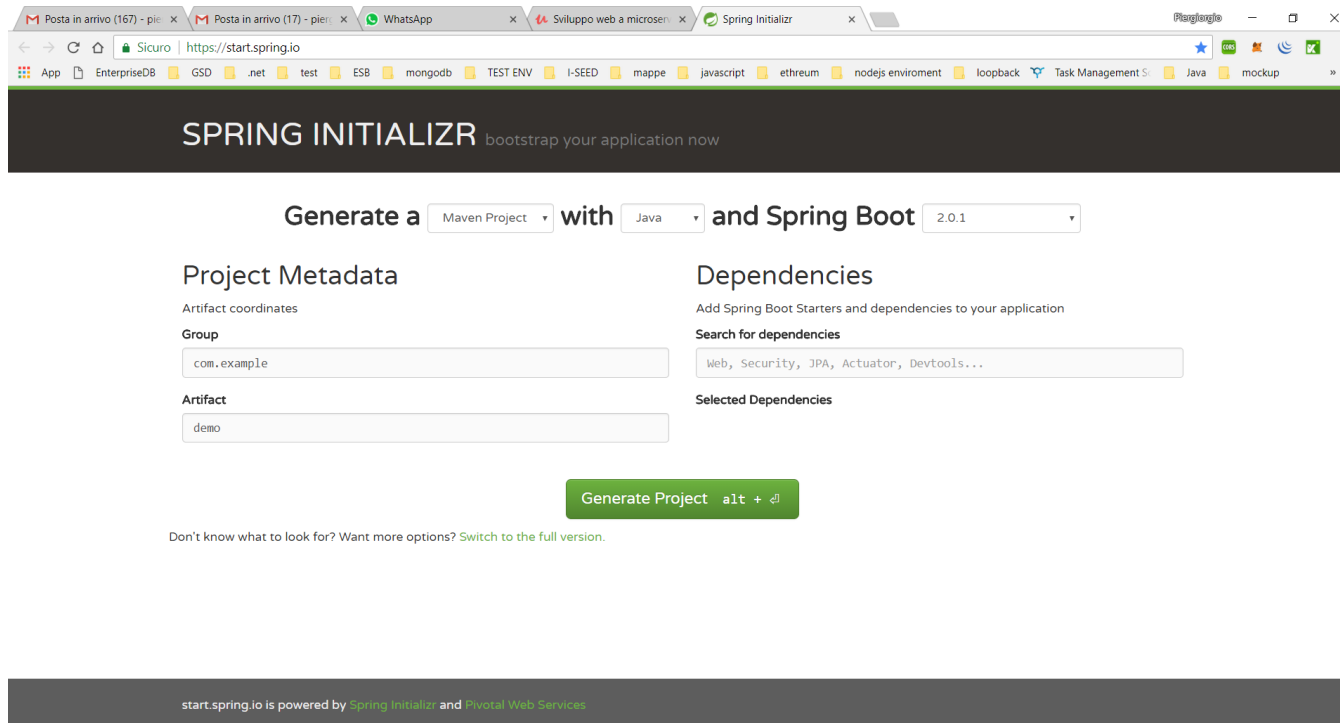
- ▶ L'HTTP è un protocollo stateless (senza stato) cioè ogni chiamata non sa nulla delle chiamate precedenti e non dà nessuna informazione alla chiamata successiva.
- ▶ Per cui per avere uno stato della navigazione utente è necessario utilizzare i cookies o JWT, i quali vengono passati in ogni chiamata e mantengono informazioni di stato.
- ▶ Sia i cookies che i JWT sono generati dal server, inviati alla prima richiesta al client, nel caso di WEB application al browser, e poi vengono reinviati da questo al server ad ogni chiamata successiva.
- ▶ La differenza tra i cookies ed il JWT è che il server memorizza i cookies mentre i JWT non vengono memorizzati dal server, ma li elabora ad ogni chiamata controllandone la validità e decrittando le informazioni con una chiave segreta. Solitamente i cookies memorizzano il sessionId mentre i JWT memorizzano informazioni quali l'utente.

Use Cases



pom.xml with Spring Inizializer

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



The screenshot shows the Spring Initializr web application in a browser. The browser's address bar displays the URL `https://start.spring.io/`. The page features a dark header with the text "SPRING INITIALIZR" and the tagline "bootstrap your application now". Below the header, there is a form to generate a project. The form includes a "Generate a" dropdown menu set to "Maven Project", a "with" dropdown menu set to "Java", and a "Spring Boot" dropdown menu set to "2.0.1". The form is divided into two main sections: "Project Metadata" and "Dependencies". The "Project Metadata" section contains fields for "Group" (set to "com.example") and "Artifact" (set to "demo"). The "Dependencies" section contains a "Search for dependencies" input field with the text "Web, Security, JPA, Actuator, Devtools..." and a "Selected Dependencies" section. A green "Generate Project" button is located at the bottom of the form. At the bottom of the page, a footer states "start.spring.io is powered by Spring Initializr and Pivotal Web Services".

Generate a Maven Project with Java and Spring Boot 2.0.1

Project Metadata
Artifact coordinates

Group

Artifact

Dependencies
Add Spring Boot Starters and dependencies to your application

Search for dependencies

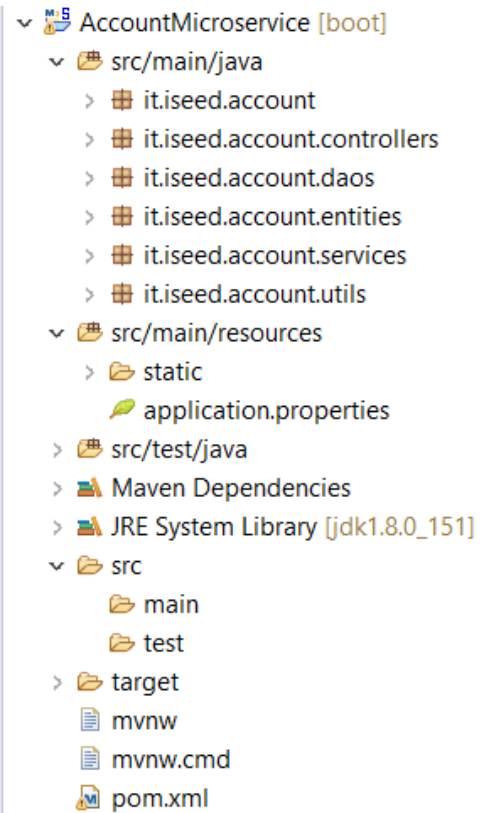
Selected Dependencies

[Generate Project](#) alt + ⌘

Don't know what to look for? Want more options? [Switch to the full version.](#)

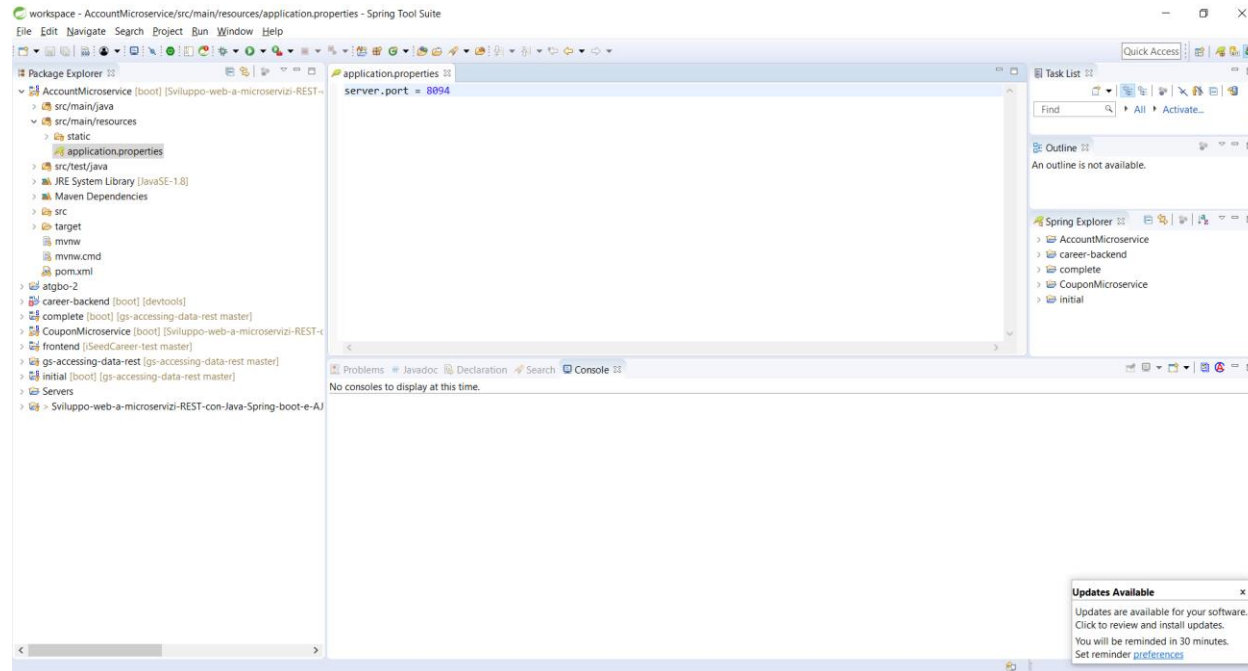
start.spring.io is powered by [Spring Initializr](#) and [Pivotal Web Services](#)

Struttura di un progetto con maven



Configurazione porta di ascolto

`server.port = 8094`



Applicazione lanciata nel main

La classe principale del progetto deve essere annotata con `SpringBootApplication`

```
@SpringBootApplication  
public class AccountMicroserviceApplication
```

Il main utilizzerà il metodo statico di `SpringApplication` `run` per eseguire l'applicazione che istanzierà un http listener con il tomcat embededd sulla porta configurata:

```
public static void main(String[] args){  
    SpringApplication.run(AccountMicroserviceApplication.class, args);  
}
```

Command line runner

Il metodo run della interfaccia CommandLineRunner viene eseguito prima che il server sia in ascolto, può essere utile per prepolare il db di test H2 ad esempio:

```
@SpringBootApplication  
public class AccountMicroserviceApplication implements CommandLineRunner{
```

Qui ad esempio effettuiamo un log dimostrativo che sarà stampato prima che il server embedded sia up and running:

```
@Override  
public void run(String... strings) throws Exception {  
    //...  
    log.info("Hello 1");  
}
```

Hello world Controller

L'annotation `RestController` permette di sfruttare Spring per gestire il mapping tra url e classe di gestione delle request http o https REST

```
@org.springframework.web.bind.annotation.RestController  
public class RestController {
```

```
    @RequestMapping("/hello")  
    public String sayHello(){  
        return "Hello everyone!";  
    }
```

Testiamolo con Postman, la stringa arriverà nel corpo della response

Entity/Pojo/Beans mapping db record

- ▶ Creiamo ad esempio una entity User per rappresentare un utente di una applicazione internet, con properties di tipo String per questo tipo di informazioni:
- ▶ ID, USERNAME, PASSWORD, PERMISISON
- ▶ Il bean con la libreria lombok sarà molto scarno, ridotto alle properties che rappresentano le colonne del db, mentre i metodi getter e setter non sarà necessario implemetarli in quanto li creerà a runtime lombok con le annotation:
 - ▶ @Getter @Setter
- ▶ così come i metodi costruttori con le annotaion:
 - ▶ @AllArgsConstructor @NoArgsConstructor
- ▶ Per la validazione base non sarà necessario utilizzare condizioni esplicite ma si potrà utilizzare la validazione JSR-303 standard:
 - ▶ @NotEmpty @NotBlank @NotNull

```

@AllArgsConstructor @NoArgsConstructor //Lombok annotations
@Entity //JPA defines an Entity
@Table(name = "users") //JPA (if table name in the DB differs from Class Name)
public class User {

    //String ID, String USERNAME, String PASSWORD, String PERMISSION

    @Id //JPA id of the table
    @Column(name="ID") //JPA (if column name is different from variable name)
    @NotEmpty @NotBlank @NotNull //JSR-303 Validation
    @Getter @Setter //Lombok annotations
    private String id;

    @Column(name="USERNAME") //JPA (if column name is different from variable name)
    @NotEmpty @NotBlank @NotNull //JSR-303 Validation
    @Getter @Setter //Lombok annotations
    private String username;

    @Column(name="PASSWORD") //JPA (if column name is different from variable name)
    @NotEmpty @NotBlank @NotNull //JSR-303 Validation
    @Getter @Setter //Lombok annotations
    private String password;

    @Column(name="PERMISSION") //JPA (if column name is different from variable name)
    @NotEmpty @NotBlank @NotNull //JSR-303 Validation
    @Getter @Setter //Lombok annotations
    private String permission;

}

```


Controller, senza validazione

Questo controller se riceverà nella request un id e uno username, come definiti nel bean User, stamperà i valori a log, ma non effettua nessun controllo:

```
/* testare con PostMan con modalità x-www-form-urlencoded */  
  
//if pwd is null it will still return a user  
@RequestMapping("/newuser1")  
public String addUser(User user){  
    return "User added correctly:" + user.getId() + ", " + user.getUsername();  
}
```

Testare con postman, con chiamate senza parametri

Controller, con validazione JSR-303

- ▶ Nell'entity User avevamo inserito le validazioni JSR-303, con le annotations:
 - ▶ @NotEmpty, @NotNull, @NotBlank
- ▶ Se volessimo che il controller le sfrutti dobbiamo usare una nova annotation nel controller: @Valid

```
//if pwd is null it will return a JAVA JSR-303 error message thanks to @Valid
@RequestMapping("/newuser2")
public String addUserValid(@Valid User user){
    return "User added correctly:" + user.getId() + ", " + user.getUsername();
}
```

- ▶ In questo modo tutti le properties del bean dovranno essere valorizzate e non con stringa vuota, si può testare con postman.

Controller, con validazione JSR-303 e Binding Result

- Grazie a Spring si possono intercettare il risultato e gestire la risposta in modo custom:

```
//if pwd is null it will return a JAVA JSR-303 error message thanks to Spring object BindingResult
@RequestMapping("/newuser3")
public String addUserValidPlusBinding(@Valid User user, BindingResult result){
    if(result.hasErrors()){
        return result.toString();
    }
    return "User added correctly:" + user.getId() + ", " + user.getUsername();
}
```

- In questo caso intercettiamo il risultato e invece che dare la response standard in caso di errore in Json, rispondiamo con solamente l'errore nel corpo del messaggio, si può testare con postman.

Validazione complesse 1

Aggiungiamo una validazione più complessa ad esempio sulla lunghezza del campo password:

```
//if pwd is null it will return a SPRING VALIDATOR error message thanks to Spring object BindingResult
@RequestMapping("/newuser4")
public String addUserValidPlusBinding2(User user, BindingResult result){
    /* Spring validation */
    UserValidator userValidator = new UserValidator();
    userValidator.validate(user, result);

    if(result.hasErrors()){
        return result.toString();
    }
    return "User added correctly:" + user.getId() + ", " + user.getUsername();
}
```

Validazione complessa 2

- Creando una inner class di validazione che implementi l'interfaccia Validator:

```
/*-----INNER CLASS-----*/  
//Spring Validator Example  
private class UserValidator implements Validator {  
  
    @Override  
    public boolean supports(Class<?> clazz) {  
        return User.class.equals(clazz);  
    }  
  
    @Override  
    public void validate(Object obj, Errors errors) {  
        User user = (User) obj;  
        if (user.getPassword().length() < 8) {  
            errors.rejectValue("password", "the password must be at least 8 chars long!");  
        }  
    }  
}  
/*-----*/
```



Mapping bean tabella a DB

- ▶ Per quanto riguarda il mapping tabella/colonne e bean/property ci vengono in aiuto le annotation standard di persistenza di JPA:
 - ▶ `@Table(name = "users")`, a livello di classe per mappare il bean su di una tabella
 - ▶ `@Column(name="USERNAME")`, a livello di property per rimappare una property su di un bean
 - ▶ `@Id`, a livello di property per indicare la property che si rimappa sull'identificativo della tabella, la primary key

```

@AllArgsConstructor @NoArgsConstructor //Lombok annotations
@Entity //JPA defines an Entity
@Table(name = "users") //JPA (if table name in the DB differs from Class Name)
public class User {

    //String ID, String USERNAME, String PASSWORD, String PERMISSION

    @Id //JPA id of the table
    @Column(name="ID") //JPA (if column name is different from variable name)
    @NotEmpty @NotBlank @NotNull //JSR-303 Validation
    @Getter @Setter //Lombok annotations
    private String id;

    @Column(name="USERNAME") //JPA (if column name is different from variable name)
    @NotEmpty @NotBlank @NotNull //JSR-303 Validation
    @Getter @Setter //Lombok annotations
    private String username;

    @Column(name="PASSWORD") //JPA (if column name is different from variable name)
    @NotEmpty @NotBlank @NotNull //JSR-303 Validation
    @Getter @Setter //Lombok annotations
    private String password;

    @Column(name="PERMISSION") //JPA (if column name is different from variable name)
    @NotEmpty @NotBlank @NotNull //JSR-303 Validation
    @Getter @Setter //Lombok annotations
    private String permission;

}

```


DAO via Repository

- ▶ Con Spring le CRUD verso il DB vengono realizzate attraverso i Repository
- ▶ Per essere slegati dall'ORM qui useremo il JpaRepository, Spring userà Hibernate per l'implementazione dei metodi definiti nell'interfaccia:

```
▼ JpaRepository.class
  ▼ JpaRepository<T, ID extends Serializable>
    deleteAllInBatch() : void
    deleteInBatch(Iterable<T>) : void
    findAll() : List<T>
    findAll(Iterable<ID>) : List<T>
    findAll(Example<S>) <S extends T> : List<S>
    findAll(Example<S>, Sort) <S extends T> : List<S>
    findAll(Sort) : List<T>
    flush() : void
    getOne(ID) : T
    save(Iterable<S>) <S extends T> : List<S>
    saveAndFlush(S) <S extends T> : S
```

Operazioni CUSTOM su una tabella

- È possibile aggiungere anche operazioni custom verso il DB, ad esempio la ricerca di un utente dalla tabella users per campo ID:

```
package it.iseed.account.daos;  
  
import org.springframework.data.jpa.repository.JpaRepository;  
  
import it.iseed.account.entities.User;  
  
import java.util.Optional;  
  
public interface UserDao extends JpaRepository<User, String>{  
    //custom  
    Optional<User> findById(String id);  
}
```

Servizi

- ▶ I servizi usano, in UML, sono associati, ai DAO, per lavorare con i dati, e implemetano le logiche di business dell'applicazione
 - ▶ nel nostro caso ad esempio il servizio di login, va cercare se l'utente è presente nel DB e recupera le informazioni per verificare la password...

```
@Override
```

```
public Optional<User> getUserFromDbAndVerifyPassword(String id, String password) throws UserNotLoggedExce
```

```
Optional<User> userr = userDao.findById(id);  
if(userr.isPresent()){  
    User user = userr.get();  
    if(encryptionUtils.decrypt(user.getPassword()).equals(password)){  
        Log.info("Username and Password verified");  
    }else{  
        Log.info("Username verified. Password not");  
        throw new UserNotLoggedException("User not correctly logged in");  
    }  
}  
return userr;  
}
```

Classe Optional delle java utils

A container object which may or may not contain a non-null value.

If a value is present, `isPresent()` will return `true` and `get()` will return the value.

- Permette di lavorare in modo elegante con oggetti che possono essere nulli o vuoti

Query native nei DAO

- Nel caso si vogliano lanciare delle query native da un DAO, si possono scrivere direttamente in SQL con la annotation @Query come sotto:

```
public interface AccountDao extends JpaRepository<Account, String>{  
    @Query(value = "SELECT * FROM accounts WHERE FK_USER=:user", nativeQuery = true)  
    List<Account> getAllAccountsPerUser(@Param("user") String user);  
  
    List<Account> findByFkUser(String fkUser);  
}
```

- L'implementazione sarà effettuata automaticamente da Spring che userà il parametro Stringa user e lo passerà alla query.
- Il risultato sarà un List di elementi Account tornati dalla query

JSON Web Token

- Utilizzeremo la libreria jsonwebtoken inserendo la dipendenza nel pom:

```
<dependency>  
  <groupId>io.jsonwebtoken</groupId>  
  <artifactId>jjwt</artifactId>  
  <version>0.7.0</version>  
</dependency>
```

- Come accennato all'inizio lo scopo è gestire «la sessione utente», l'utente loggato potrà rimandare il JWT al server e effettuare delle operazioni successive venendo riconosciuto, in quanto nel JWT salveremo anche l'id dell'utente

JSON Web Token

Utilizzando il metodo statico builder della classe Jwts generiamo un web token con: soggetto, date di scadenza, nome, scope, lo criptiamo e lo comprimiamo prima di inviarlo al browser alla prima chiamata:

```
/**
 * this method generate the Jwt token to be sent to the client
 * @param subject "RGNLSN87H13D761R"
 * @param date    new Date(1300819380)
 * @param name    "Alessandro Argentieri"
 * @param scope   "user"
 * @return String jwt
 * @throws java.io.UnsupportedEncodingException
 */
public static String generateJwt(String subject, Date date, String name, String scope) {

    String jwt = Jwts.builder()
        .setSubject(subject)
        .setExpiration(date)
        .claim("name", name)
        .claim("scope", scope)
        .signWith(
            SignatureAlgorithm.HS256,
            "myPersonalSecretKey12345".getBytes("UTF-8")
        )
        .compact();

    return jwt;
}
```

JWT, gestione sessione utente

- ▶ A questo punto possiamo chiamare il servizio che torna tutti gli account dell'utente loggato con il controller seguente, il quale verifica per prima cosa il JWT:

```
@RequestMapping(value = "/accounts/user", method = POST)
public ResponseEntity<JsonResponseBody> fetchAllAccountsPerUser(HttpServletRequest request){
    //request -> fetch JWT -> recover User Data -> Get user accounts from DB
    try {
        Map<String, Object> userData = loginService.verifyJwtAndGetData(request);
        return ResponseEntity.status(HttpStatus.OK).body(new JsonResponseBody(HttpStatus.OK.value(), operationService.getAllAccountsPerUser((String) userData.get("subject"))));
    }catch(UnsupportedEncodingException e1){
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(new JsonResponseBody(HttpStatus.BAD_REQUEST.value(), "Bad Request: " + e1.toString()));
    }catch(UserNotLoggedInException e2){
        return ResponseEntity.status(HttpStatus.FORBIDDEN).body(new JsonResponseBody(HttpStatus.FORBIDDEN.value(), "User not logged! Login first : " + e2.toString()));
    }catch(ExpiredJwtException e3){
        return ResponseEntity.status(HttpStatus.GATEWAY_TIMEOUT).body(new JsonResponseBody(HttpStatus.GATEWAY_TIMEOUT.value(), "Session Expired!: " + e3.toString()));
    }
}
```

- ▶ In questo modo abbiamo collegato la prima chiamata di login a questa

Chiamate con url parametrico

- ▶ Per effettuare chiamate parametriche si può utilizzare anche un url dinamico:
 - ▶ che contiene un dato, nel nostro caso di filtro per ottenere tutte le operazioni di un dato utente su di un account, e grazie all'annotation `@PathVariable`, convertire l'url automaticamente a parametro del metodo del controller:

```
@RequestMapping("/operations/account/{account}")
public ResponseEntity<JsonResponseBody> fetchAllOperationsPerAccount(HttpServletRequest request, @PathVariable(name = "account") String account){
    //request -> fetch JWT -> check validity -> Get operations from the user account
    try {
        loginService.verifyJwtAndGetData(request);
        //user verified
        return ResponseEntity.status(HttpStatus.OK).body(new JsonResponseBody(HttpStatus.OK.value(), operationService.getAllOperationPerAccount(account)));
    } catch (UnsupportedEncodingException e1){
        return ResponseEntity.status(HttpStatus.FORBIDDEN).body(new JsonResponseBody(HttpStatus.FORBIDDEN.value(), "Unsupported Encoding: " + e1.toString()));
    } catch (UserNotLoggedInException e2) {
        return ResponseEntity.status(HttpStatus.FORBIDDEN).body(new JsonResponseBody(HttpStatus.FORBIDDEN.value(), "User not correctly logged: " + e2.toString()));
    } catch (ExpiredJwtException e3){
        return ResponseEntity.status(HttpStatus.GATEWAY_TIMEOUT).body(new JsonResponseBody(HttpStatus.GATEWAY_TIMEOUT.value(), "Session Expired!: " + e3.toString()));
    }
}
```

Risposte in formato JSON

In tutti i casi abbiamo ottenuto come risposte degli oggetti JSON, grazie alla libreria Jackson che automaticamente converte un oggetto in JSON, rimappando le proprietà di una Classe in coppie nome:valore json. In particolare utilizziamo un oggetto che contiene sia i dati che la risposta http, ad esempio 200 su risposta ok:

```
/**
 * inner class used as the Object tied into the Body of the ResponseEntity.
 * It's important to have this Object because it is composed of server response code and response object.
 * Then, JACKSON LIBRARY automatically convert this JsonResponseBody Object into a JSON response.
 */
@AllArgsConstructor
public class JsonResponseBody{
    @Getter @Setter
    private int server;
    @Getter @Setter
    private Object response;
}
```