Opc Server with Milo and Spring Boot

> Documentazione di progetto per la materia Informatica Industriale

Introduzione

La presente documentazione descrive come poter implementare un Server Opc utilizzando come standard SDK per implementare lo stack protocollare Eclipse Milo.

Nell'ambito del corso di Informatica Industriale è stato richiesto di sviluppare un Server OpcUa.

Il server opc deve soddisfare le seguenti caratteristiche:

- Implementare diversi meccanismi di security
- Implementare delle funzionalità interessanti

Al fine di rispettare i requisiti di cui sopra si è scelto di implementare un server con le seguenti caratteristiche:

- Diversi endpoint con il supporto ai certificati
- Implementazione del meccanismo di autenticazione
- Implementazione di un servizio che prenda i dati da OpenWeater

> ℹ️ OpenWeater è un servizio on line che permette previa registrazione di utilizzare API atte a fornire informazioni meteo il servizio è gratuito se si rispettano alcuni limiti, per ulteriori informazioni -> OpenWeater Price

L'idea di questo progetto è quella di implementare il server Opc utilizzando Spring Boot, un framework di sviluppo molto diffuso anche in ambito java enterprise.

Perché Spring Boot

Spring Boot rappresenta un framework che aiuta lo sviluppatore ad implementare automaticamente dei pattern architetturali durante lo sviluppo del progetto.

> ✅ Spring è uno dei framework più utilizzati in abito enterprise. Uno degli obiettivi di questo progetto è fornire una facile implementazione dello Stack OPC UA utilizzando Eclipse Milo, allo scopo di far vedere come chi utilizzi questo Framework possa aggiungere questo stack alla lista dei servizi esposti.

Capo saldo di Spring Boot e più in generale di Spring framework è il concetto dell'IoC (Inversion of control). Il framework definisce quello che si chiama IoC Container, l'IoC Container rappresenta un'area nella quale vengono istanziati gli oggetti del progetto. Gli oggetti che si trovano all'interno del Container vengono istanziati automaticamente utilizzando di default il pattern Singleton evitando così la proliferazione di oggetti nel heap-space di java. Tutti gli oggetti definiti nel Container sono disponibili per essere utilizzati dagli altri oggetti del sistema, in pratica ogni oggetto potrà utilizzare gli altri senza doverli costruire il tutto facilitato dall'annotazione `@Autowire`.

Nel corso di questo documento verranno mostrati quali siano i vantaggi nell'usare questo tipo di framework, tali vantaggi verranno evidenziati dall'icona ✅.

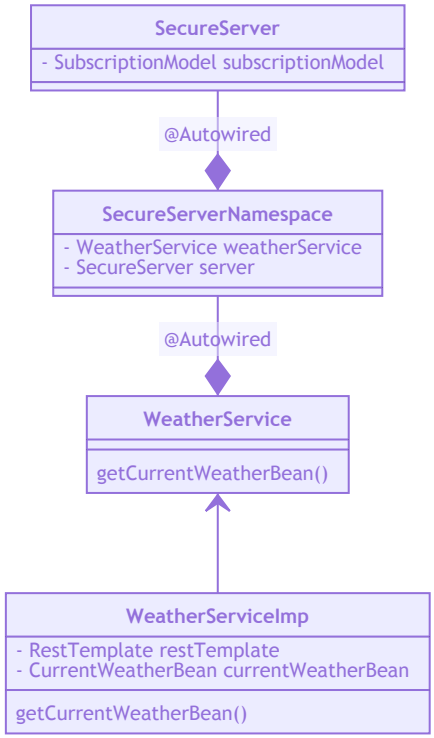
Questa documentazione non ha come obiettivo la descrizione dettagliata del framework, per maggiori informazioni consultare la documentazione ufficiale.

Descrizione del progetto

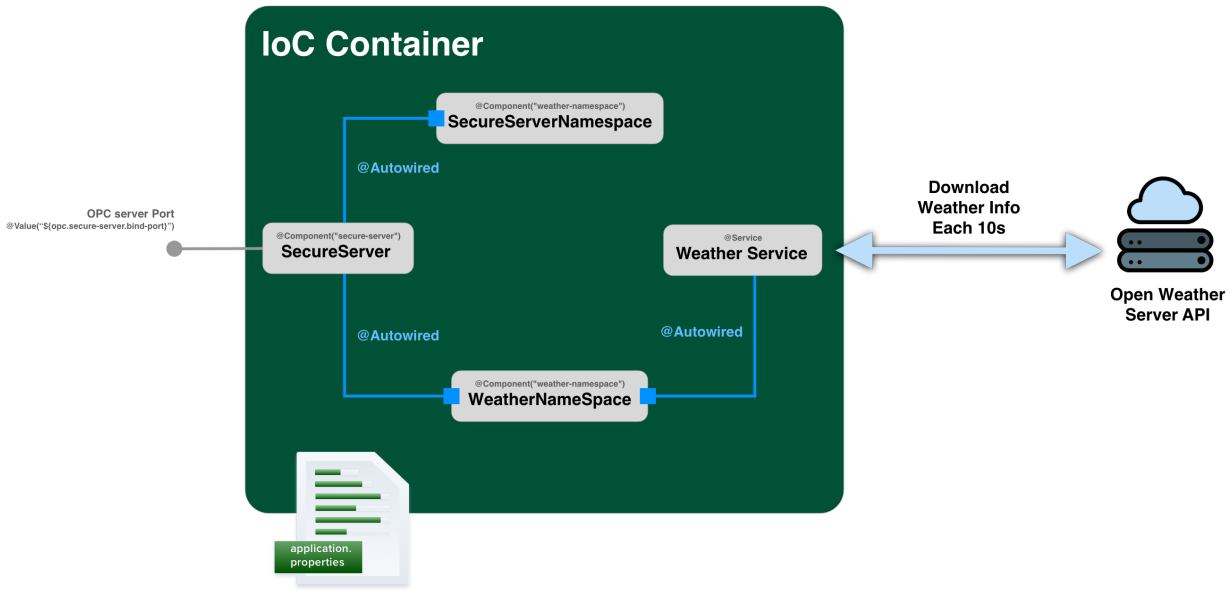
Da un punto di vista logico il progetto si compone da un insieme di oggetti che vengono creati all'interno dell'IoC container di Spring. Gli oggetti che vengono istanziati sono gestiti *automaticamente* da Spring.

> ✅ Spring Istanzia automaticamente tutto gli oggetti utilizzando il pattern **SINGLETON** evitando così la proliferazione incontrollata degli oggetti

I principali aggetti che verranno messi nell'IoC container sono:



Il sistema quindi si compone dai seguenti elementi:



L'immagine sopra mostra la struttura del sistema. Come si vede dall'immagine i componenti atti a gestire il sistema sono

- WeatherService
- SecureServer
- SecureServerNamespace
- WeatherNamespace

Nel seguito dettaglieremo tutti questi componenti.

WeatherService

L'oggetto WeatherService si compone di due elementi un'interfaccia che definisce le azione che vengono fatte dal WeatherService è un'implementazione dell'interfaccia che si chiamo `WeatherServiceImp`.

L'implementazione è annotato con l'annotazione `@Service` che indica a Spring che tale oggetto va istanziato nell'IoC Container.

L'istanziazione del servizio passa attraverso il costruttore al quale iniettiamo dei paramatri che provengono dall'`application.properties` di Spring Boot.

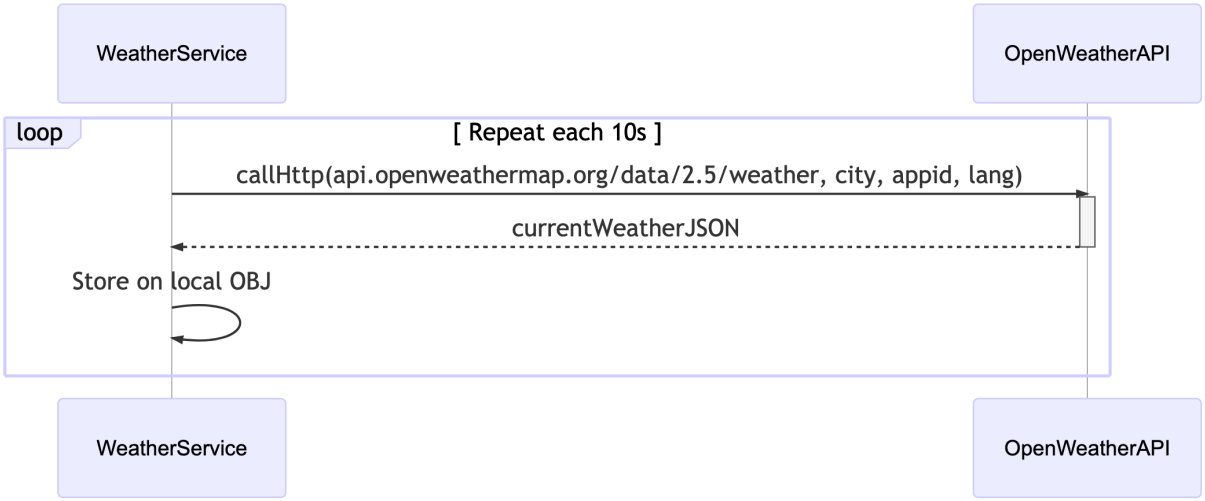
```
@Service
public class WeatherServiceImp implements WeatherService{

    //{... Attributes section}

    public WeatherServiceImp(
            @Value("${weather.apiurl}") String apiurl,
            @Value("${weather.apikey}") String weatherApikey,
```

```
            @Value("${weather.city}") String city,
            @Value("${weather.lang}") String lang
            ) {
        super();

        this.weatherApikey = weatherApikey;
        this.apiUrl = apiurl;
        this.lang = lang;
        this.city = city;

        this.createOrRefreshCompleteUrl();
        //Initialize Rest Template
        this.restTemplate = new RestTemplate();
    }
    // {... Methods section}
}
```

Come vediamo la classe annotata come `@Service` prevede un costruttore con 4 parametri tali parametri vengono presi dal file di properties utilizzando l'annotazione `@Value`.

> ✅ All'interno del costruttore vediamo anche viene inizializzato un attributo `restTemplate`. Il restTemplate è un oggetto Spring che permette di effettuare in maniera semplice le chiamate Http su API esterne.

Le azioni che deve implementare il WaterService sono sintetizzate nel seguente sequence diagram:



I particolare il servizio utilizza l'API http://api.openweathermap.org/data/2.5/weather che prende i seguenti parametri all'interno del query string:

- `q` : query contente la città della quale si vogliono avere le informazioni
- `appid` : parametro nel quale specificare un apikey ottenibile registrandosi al servizio OpenWeather
- `lang` : parametro opzionale che permette di ottenere la frase relativa le previsioni in tutte le lingue speficicado in codice della nazione (e.g. zh_CN per cinese, de per tedesco e così via)

All'interno del servizio la url con i parametri viene creata dal seguente metodo

```
private void createOrRefreshCompleteUrl() {
    //Define the remote weather url to call
    UriComponentsBuilder builder = UriComponentsBuilder
            .fromHttpUrl(this.apiUrl)
            .queryParam("q", city)
            .queryParam("appid", weatherApikey)
            .queryParam("lang", lang);
    this.apiCompleteUrl = builder.toUriString();
}
```

L'`UriComponentsBuilder` utilizza gli attributi di classe per generare una url che srà quindi nel formato

```
http://api.openweathermap.org/data/2.5/weather?q=Catania,it&appid=<your-
API-key>&lang=en
```

Per effettuare le chiamate ogni X secondi viene utilizzato lo **schedulatore** di Spring.

> ✅ Spring fornisce un meccanismo di scheduling che permette facilmente di implementare attività periodiche basate su intervalli fissi (`fixedRate`) ritardo fisso (`fixedDelay`) o cron expression (`cron`) tali valori possono anche essere agganciati dall'application.properties

L'attività schedulata che effettua le chiamate è implementata dal seguente metodo.

```
@Scheduled(fixedRateString = "${weather.callinterval}")
public void refreshCurrentWeatherBean() {
    /**
     * Get the data from URL,
     * mapping the response on class CurrentWeatherBean,
     * store on local attribute currentWeatherBean
     * */
    this.currentWeatherBean =
        this.restTemplate.getForObject(
            this.apiCompleteUrl,
            WeatherBean.class);

    logger.info("\nCall: " + this.apiCompleteUrl  +
    "\nResponse -> " + this.currentWeatherBean);
}
```

Nello snippet di codice sopra si vede l'utilizzo dell'annotazione `@Scheduled` con tale annotazione indichiamo a Spring di richiamare il metodo `refreshCurrentWeatherBean` ogni `weather.callinterval` ms prendendo da file di configurazione il numero di millisecondi che è definito a 10000 come default.

Da sottolineare inoltre l'utilizzo del RestTemplate infatti utilizziamo il metodo `getForObject`che permette di effettuare una chiamata di `GET` alla URL specificata come prim parametro e mappare la risposta JSON nella classe passata come secondo parametro.

Il body della risposta alla chiamata ha il seguente formato:


```
{
    "coord": {
        "lon": 15.09,
        "lat": 37.5
    },
    "weather": [
        {
            "id": 802,
            "main": "Clouds",
            "description": "scattered clouds",
            "icon": "03d"
        }
    ],
    "base": "stations",
    "main": {
        "temp": 301.55,
        "feels_like": 299.89,
        "temp_min": 301.15,
        "temp_max": 302.15,
        "pressure": 1012,
        "humidity": 58
    },
    "visibility": 10000,
    "wind": {
        "speed": 7.2,
        "deg": 80
    },
    "clouds": {
        "all": 40
    },
    "dt": 1593942145,
    "sys": {
        "type": 1,
        "id": 6704,
        "country": "IT",
        "sunrise": 1593920653,
        "sunset": 1593973438
    },
    "timezone": 7200,
    "id": 2525068,
    "name": "Catania",
    "cod": 200
}
```


E viene mappato in un oggetto del tipo:

```
@JsonIgnoreProperties(ignoreUnknown=true)
public class WeatherBean implements Serializable{
```

```
    private static final long serialVersionUID = 1L;

    private CoordElement coord;

    private List<WeatherElement> weather;

    private String base;

    private MainElement main;

    private Integer visibility;

    private WindElement wind;

    private CloudsElement clouds;

    private Long dt;

    private SysElement sys;

    private Long timezone;

    private Integer id;

    private String name;

    private Integer code;

    //{... Getter and Setter}
}
```

L'annotazione `@JsonIgnoreProperties(ignoreUnknown=true)` comunica al deserializzatore JSON presente dentro RestTemplate di ignorare eventuali proprietà aggiuntive ritornate nella response.

SecureServer

Il secure server è l'oggetto che ha il compite di instaurare il canale di comunicazione in accordo allo stack OPC. In particolare definisce:

- Gestisce i certificati sia del cliente che del server
- I meccanismi di connessione
- Definisce gli Endpoint

Il secure server viene creato come un `@Component` come mostrato nello snippet seguente:

```
@Component("secure-server")
public class SecureServer {

    //Internal OPC server from MiloSDK
    private final OpcUaServer server;
```

```
    public SecureServer(
                @Value("${opc.secure-server.bind-address:localhost}")
                String bindAddress,
                @Value("${opc.secure-server.bind-port:4851}")
                Integer bindPort
            ) throws Exception {
            // Costructor stuff
    }

    //Methods omitted
}
```

Con le annotazioni di cui sopra (`@Component` ed `@Value`) diciamo a Spring di istanziare un oggetto di tipo SecureServer

Struttura delle cartelle

Il progetto si struttura in diverse cartelle in accordo alla seguente struttura:

```
spring-camel-milo-server
    ├── dist
    ├── doc
    └── opcserver
```

dove:

- dist: contiene l'eseguibile di questo progetto, dentro la cartella è presente una descrizione per far partire il progetto.
- doc: contiene i sorgenti della documentazione.
- opccamelserver: contiene un server Opc di esempio utilizzando Apache Camel come descritto nell'Annex 2.
- opcserver: contenente il progetto java oggetto della presente documentazione.

Strumenti utilizzati

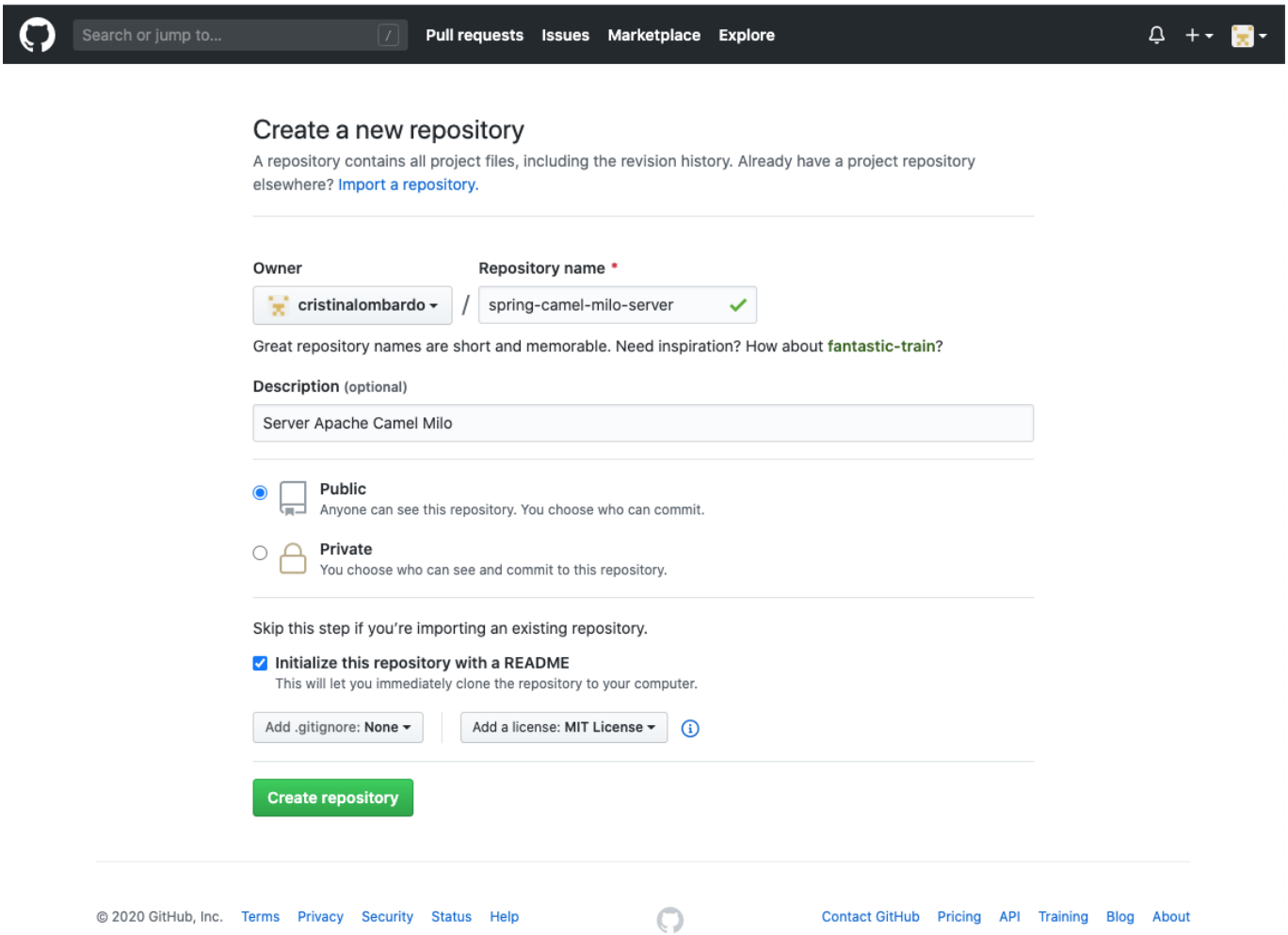
- Java SE 9
- Maven
- Eclipse with Spring tool suite
- Spring initializr

Funzionamento

//TODO

Annex 1: GitHub

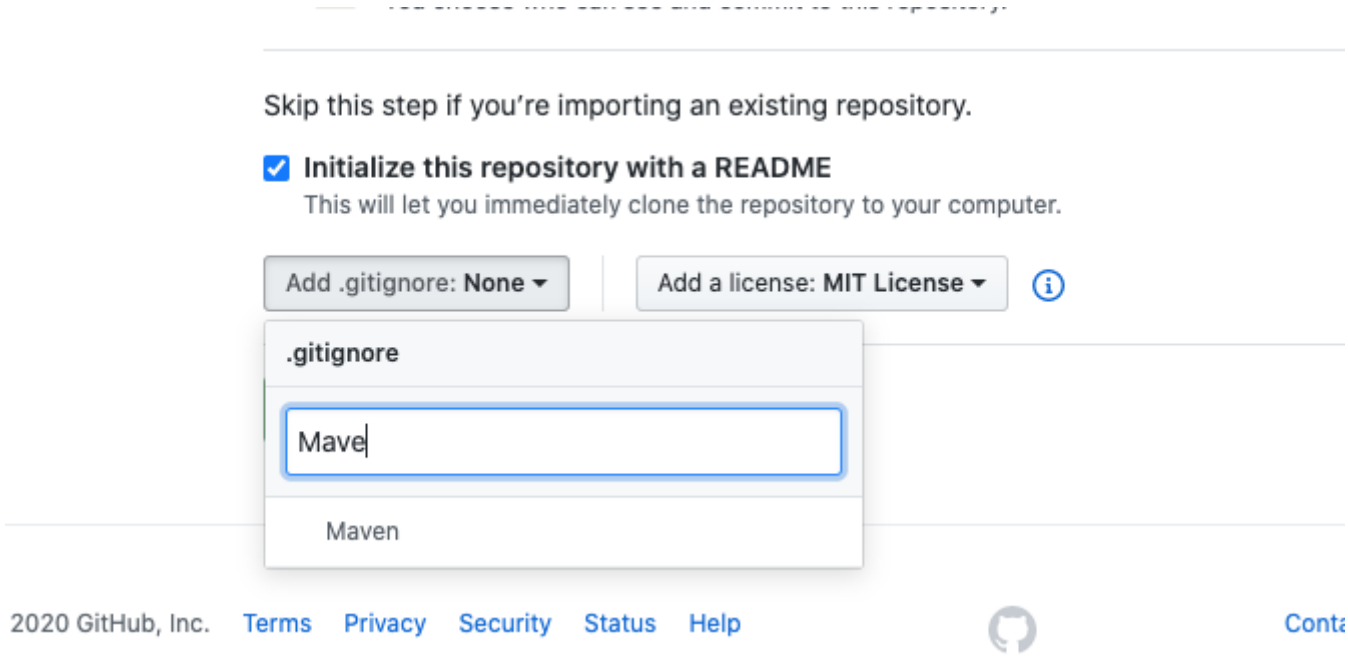
Step1: Creare il progetto su github



Step2: Readme and .gitignore

Aggiungere Readme e .gitignore

Come template del .gitignore è stato scelto Maven.



Step3: Repo Download

Creare il repository e clonarlo in locale con il programma preferito di gestione del git.

Annex 2: Camel Demo with Spring Boot

Prerequisiti:

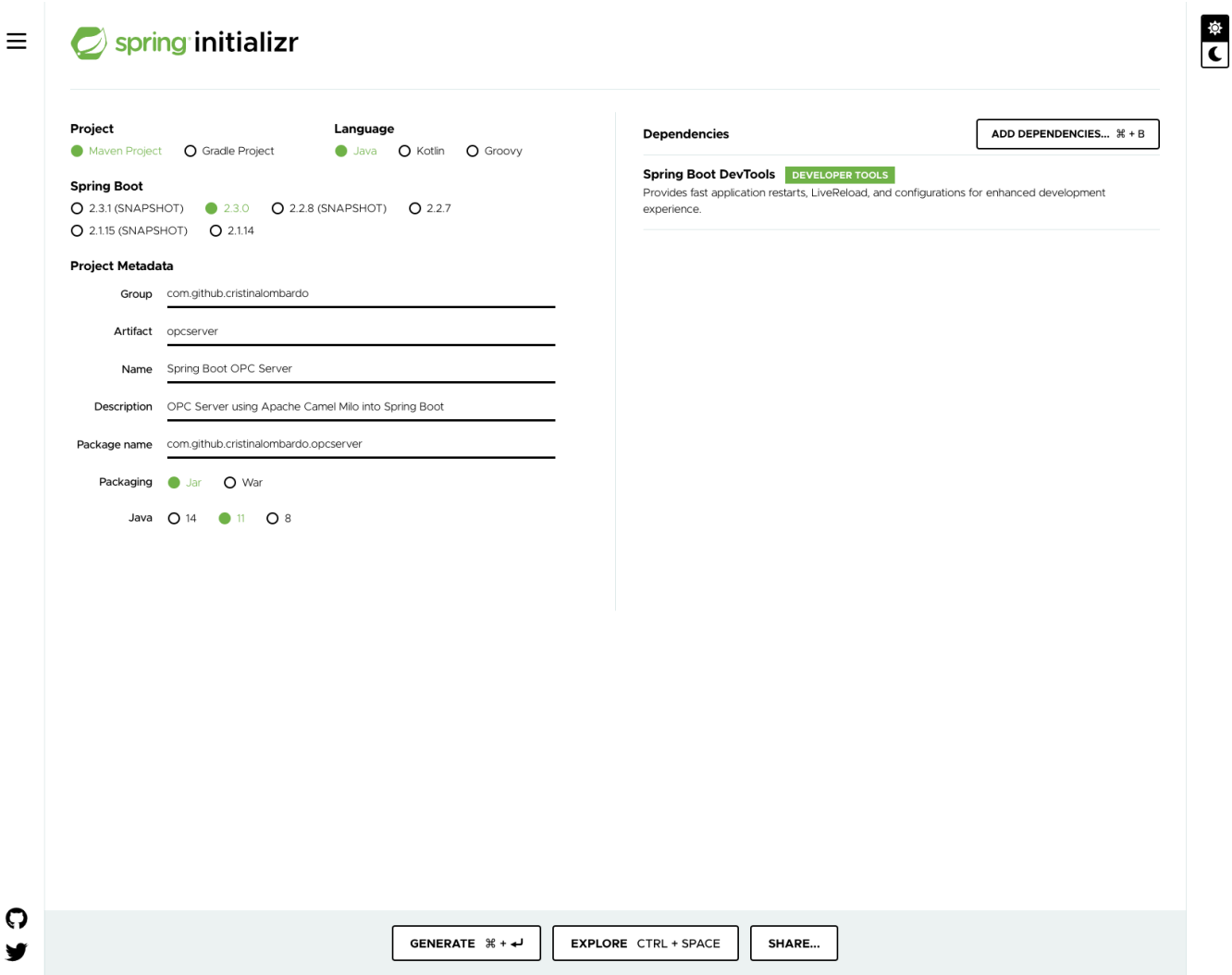
- Java 9+
- Eclipse per j2ee con Spring Tools Suite Plugin installato

Step1: Spring Init

Per inizializzare il progetto utilizzeremo Spring Initializr

Go to Spring initializr

Inizializzare il progetto in accordo alla seguente immagine



Per inizializzare il progetto abbiamo utilizzato le seguenti configurazioni:

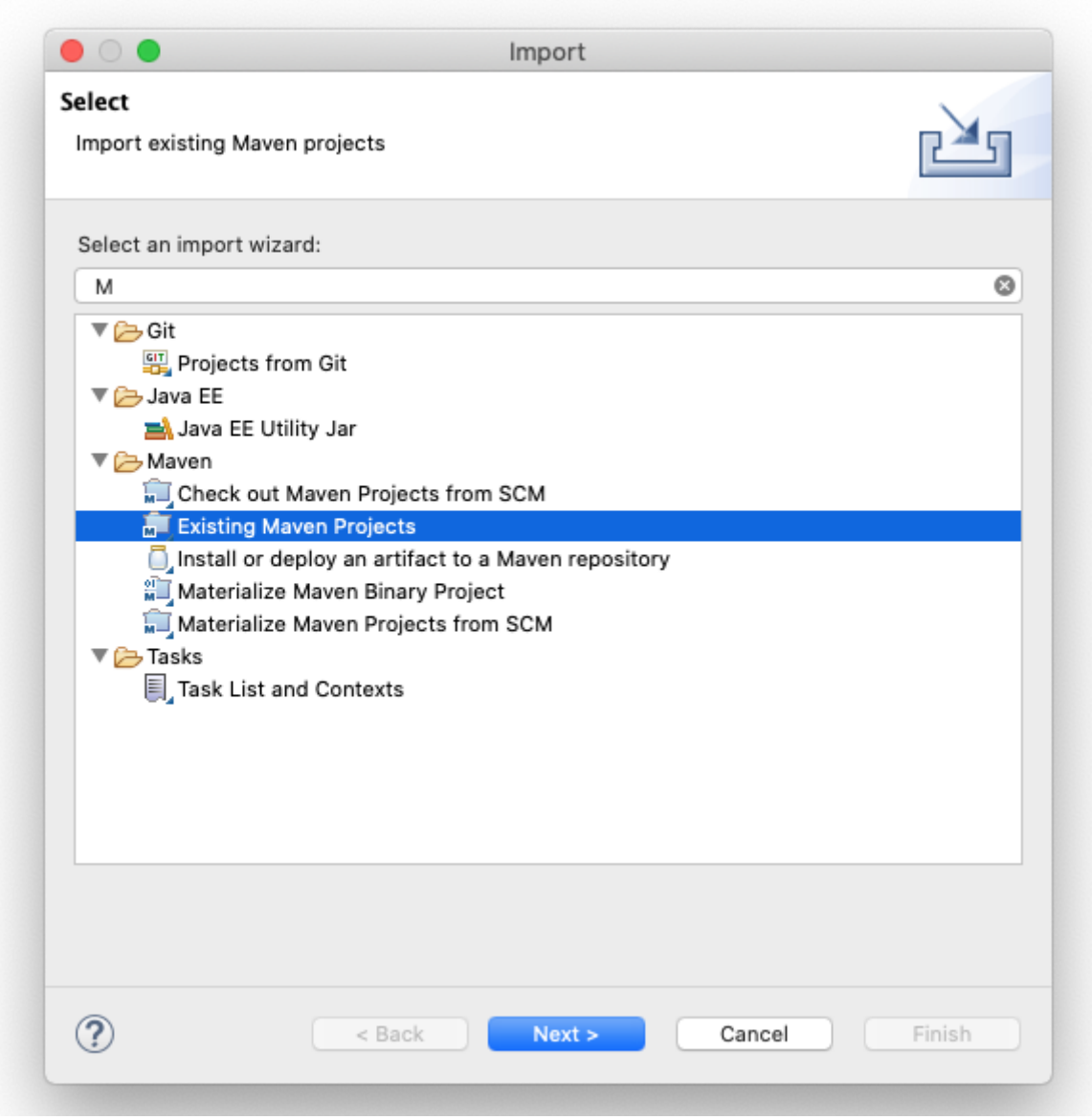
- Maven: Utilizzeremo Maven per compilare il progetto
- Spring Boot 2.3.0: Versione Stable di Spring Boot al momento di questo tutorial
- Packaging Jar: Si vuole realizzare un'applicazione standalone
- Java 11: Apache Camel Milo richiede Java 9+
- Dependencies Spring Boot DevTools: dipendenza utile durante lo sviluppo

A questo punto è possibile scaricare il progetto cliccando sul tasto GENERATE.
Una volta scaricato il progetto è stato copiato nella root del repository e decompattato.

Step2: Importare il progetto Maven dentro Eclipse

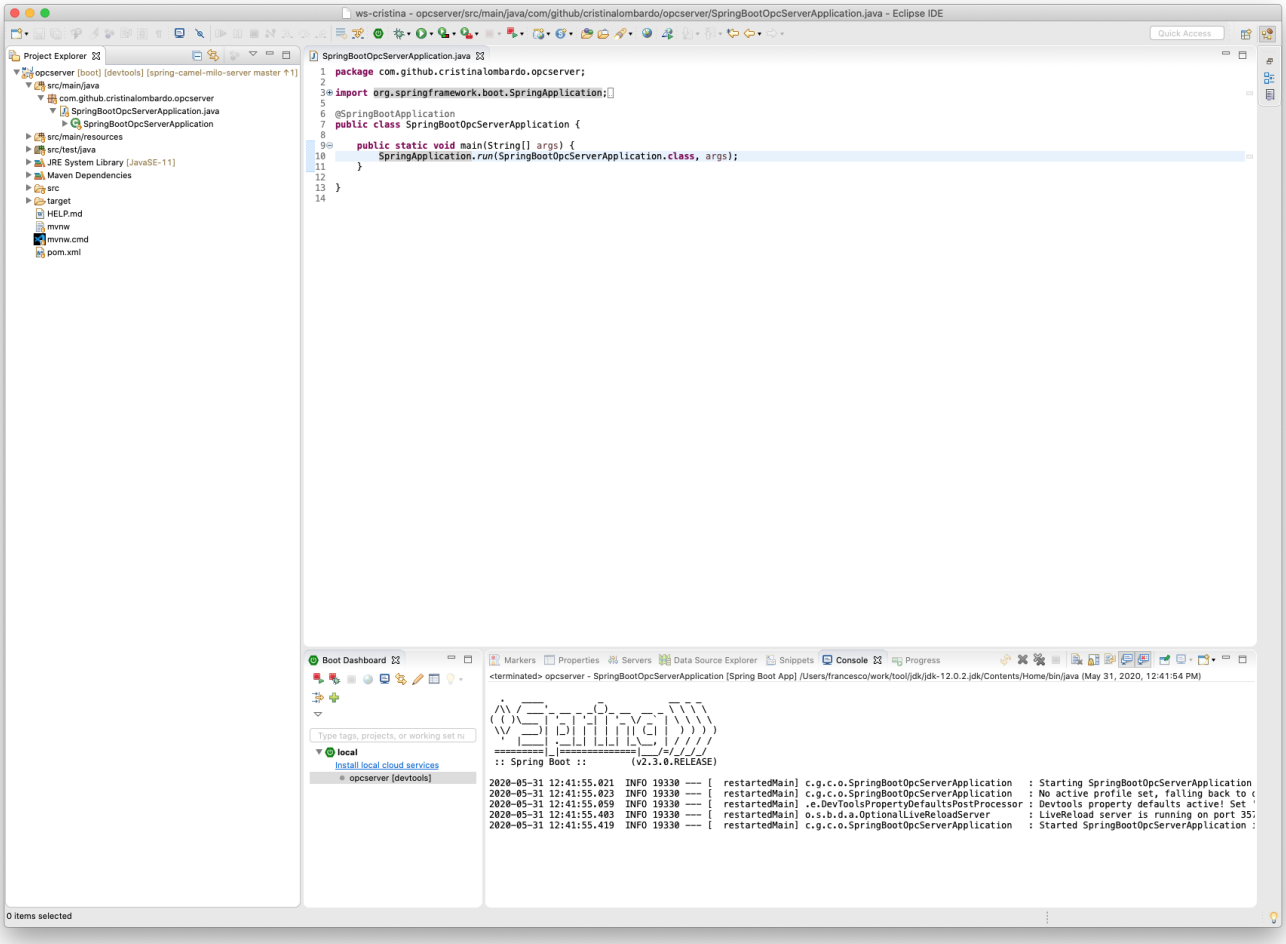
Per importare il progetto i passi sono:

1. Aprire Eclipse
2. Cliccare su File->Import... e selezionare Existing Maven Project



3. Selezionare la cartella dove risiede il progetto e cliccare su Finish

A questo punto il progetto dovrebbe presentarsi nel seguente modo:



Step3: Apache Milo

In questo step aggiungeremo le librerie Apache Camel Milo.

1. Apriamo il file `pom.xml`
2. Aggiungiamo la seguente dipendenza al file

```
<dependency>
    <groupId>org.apache.camel.springboot</groupId>
    <artifactId>camel-milo-starter</artifactId>
    <version>${camel.milo.version}</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

> La versione di Camel viene inserita nella sezione `<properties>` del `pom.xml`
> `<camel.milo.version>3.0.0</camel.milo.version>`
