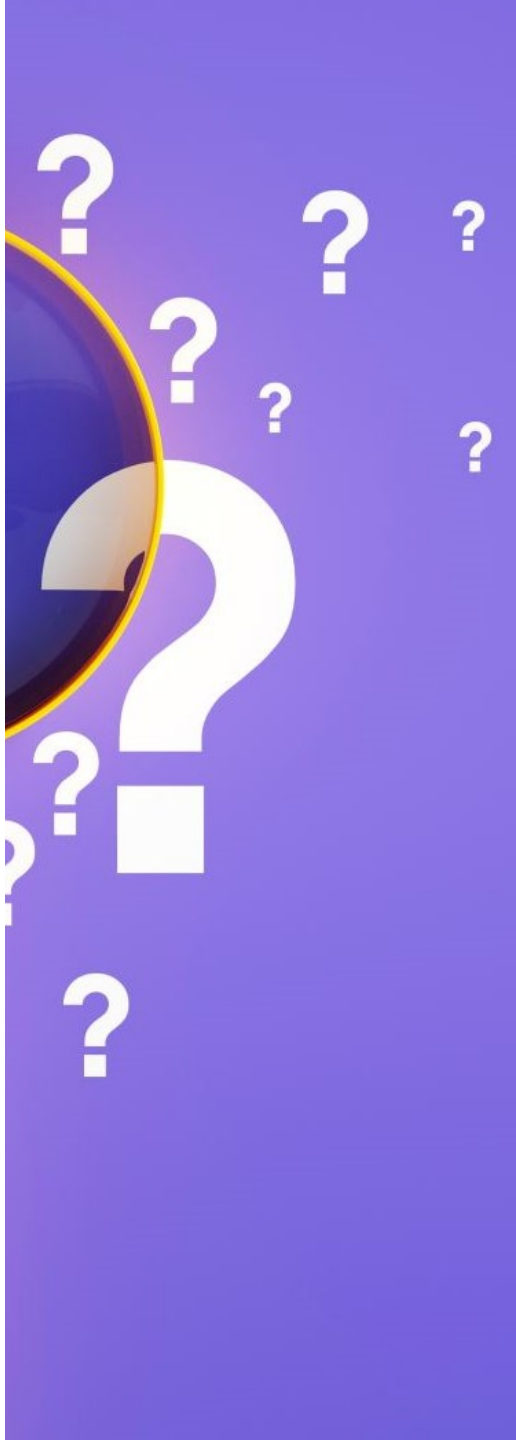
Java Advanced Programming

Summer Workshop

Foreword

Why do we attend this course?



Real world application

- We will learn how to create real world Web application in Java with Spring

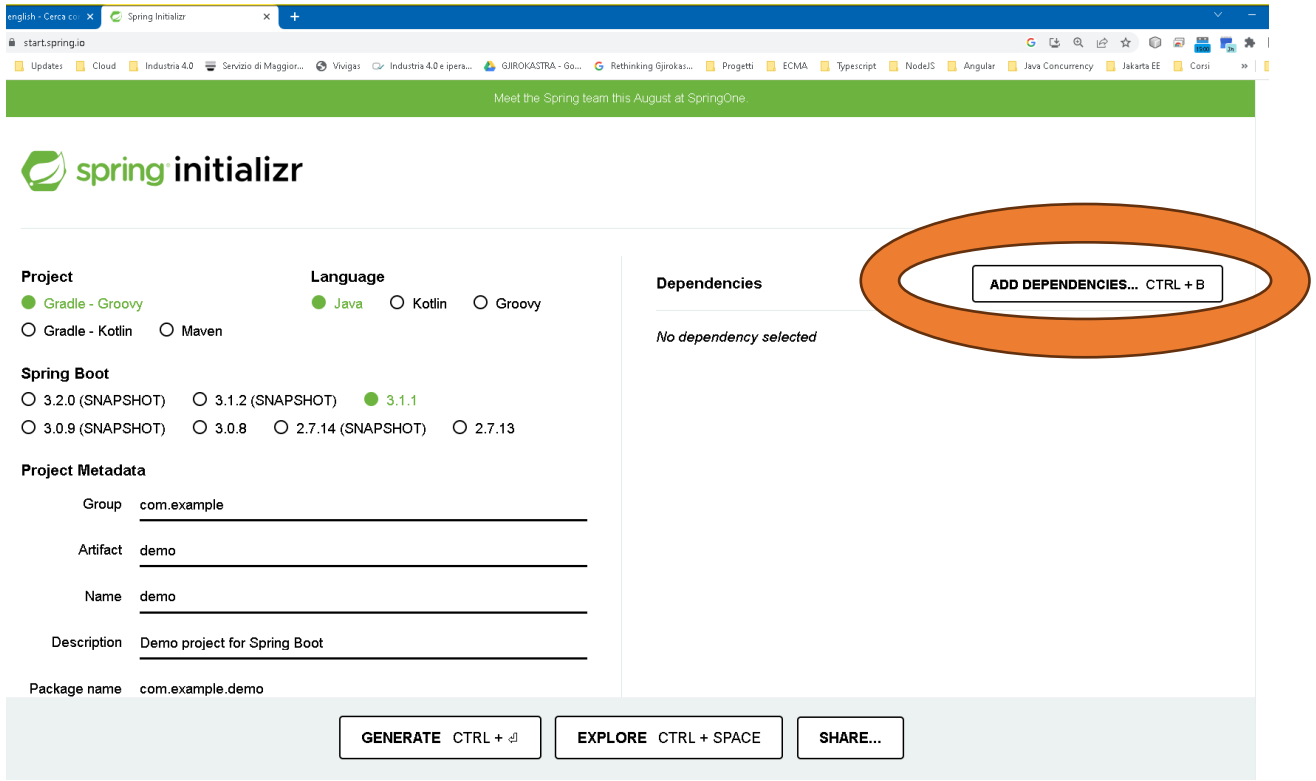
Job Interview

- This course prepare you, if you will study, to apply for a Job Interview as Junior Java Programmer for Albania, Italy, Germany and more

Spring Basics

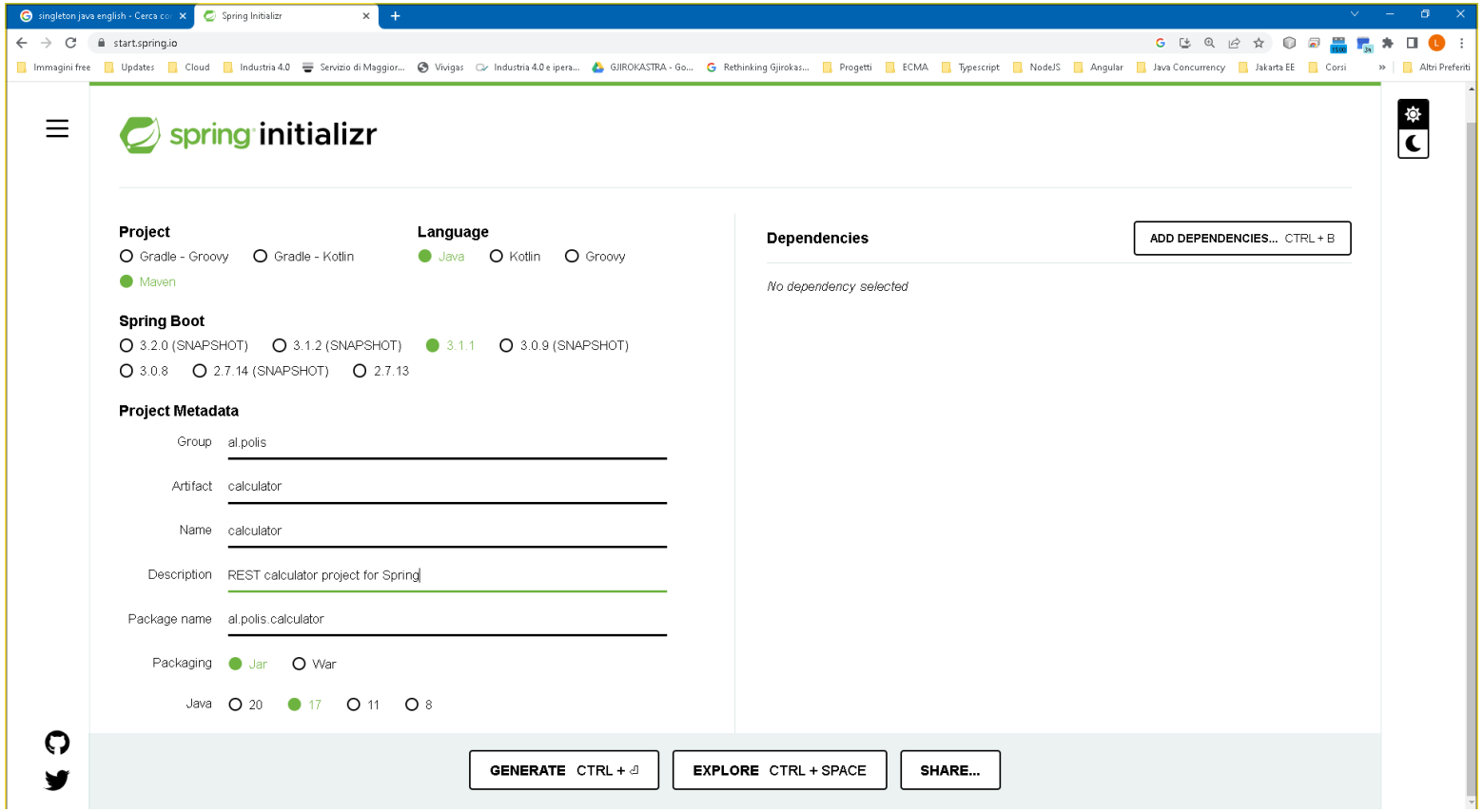
Architecture and starting a project

Creating a new Spring Application

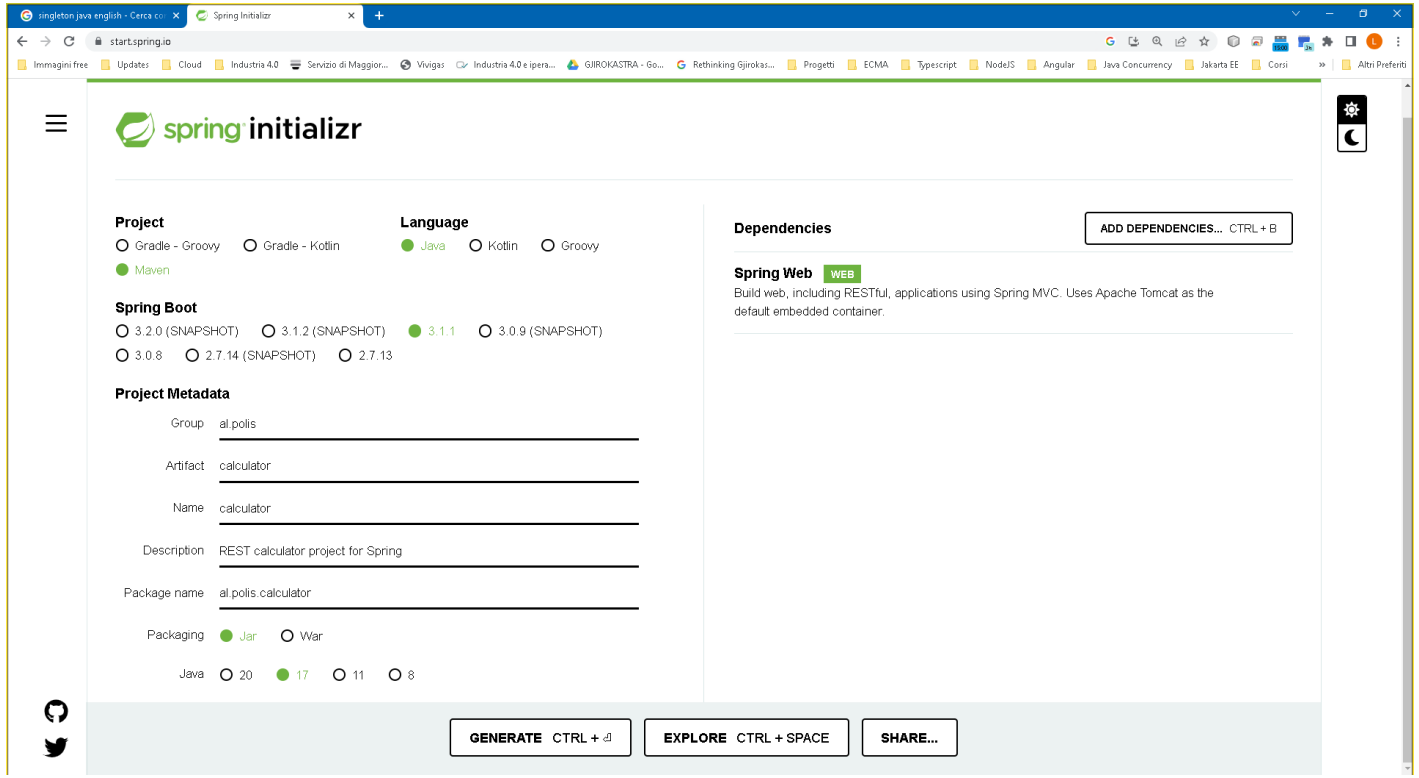


- Using initializr we can create the skeleton of an empty spring application.
- We must only provide the «libraries» (called dependencies) that the framework has to use
- To select dependencies we use ADD DEPENDENCIES button

Basic configuration



Web application only



Creating an app: 1) Controller

- Create the controller package (inside root application package!)
- Create the controller class

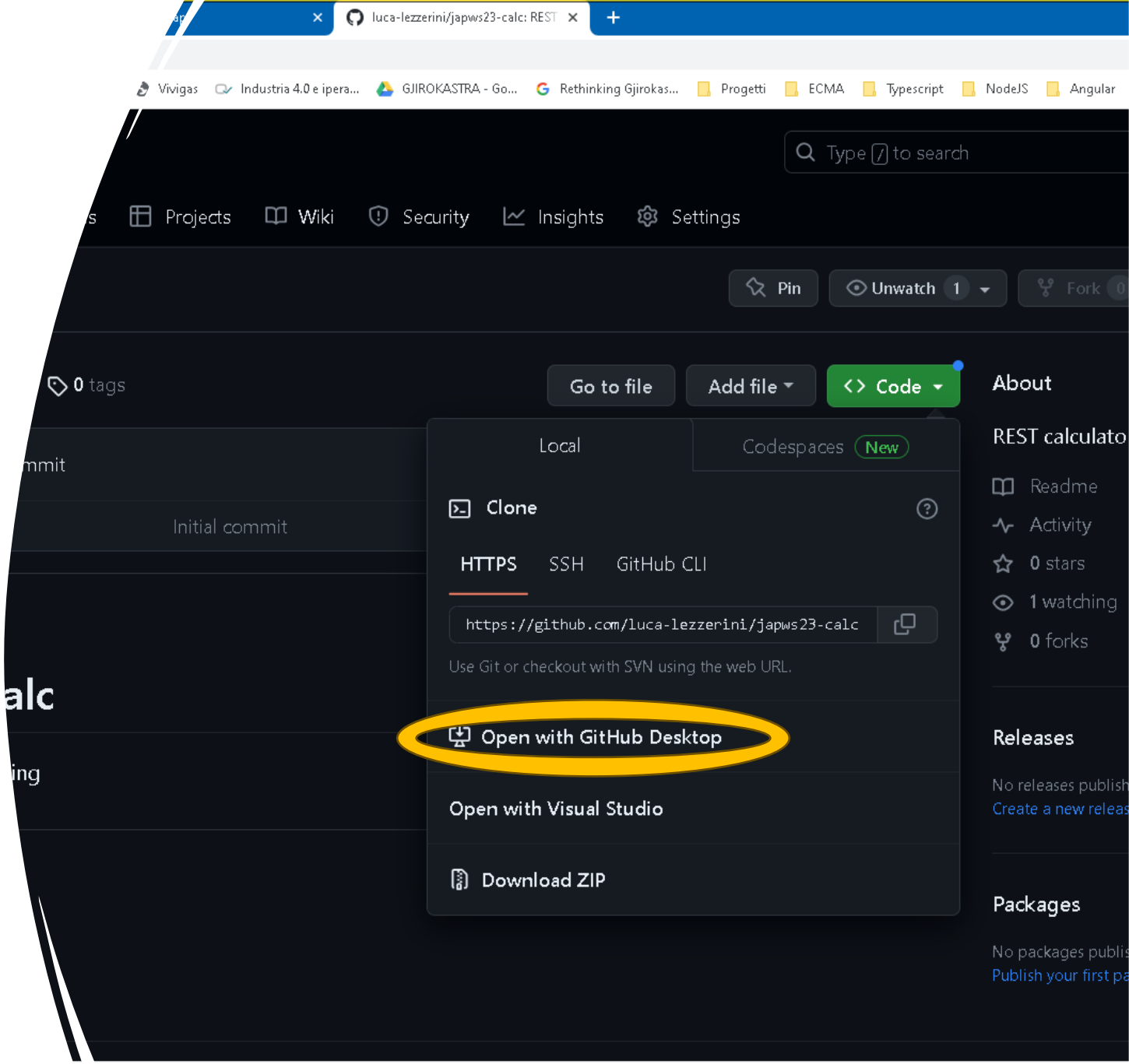
GitHub Basics

GitHub is a Source Control Software

- Remembers all the changes on the code
- Avoids conflicts
- Enable cooperation of remote programmers like if they were in the same office
- Allows managing issues (i.e. bugs and changes), milestones and more
- Keeps colorful statistics

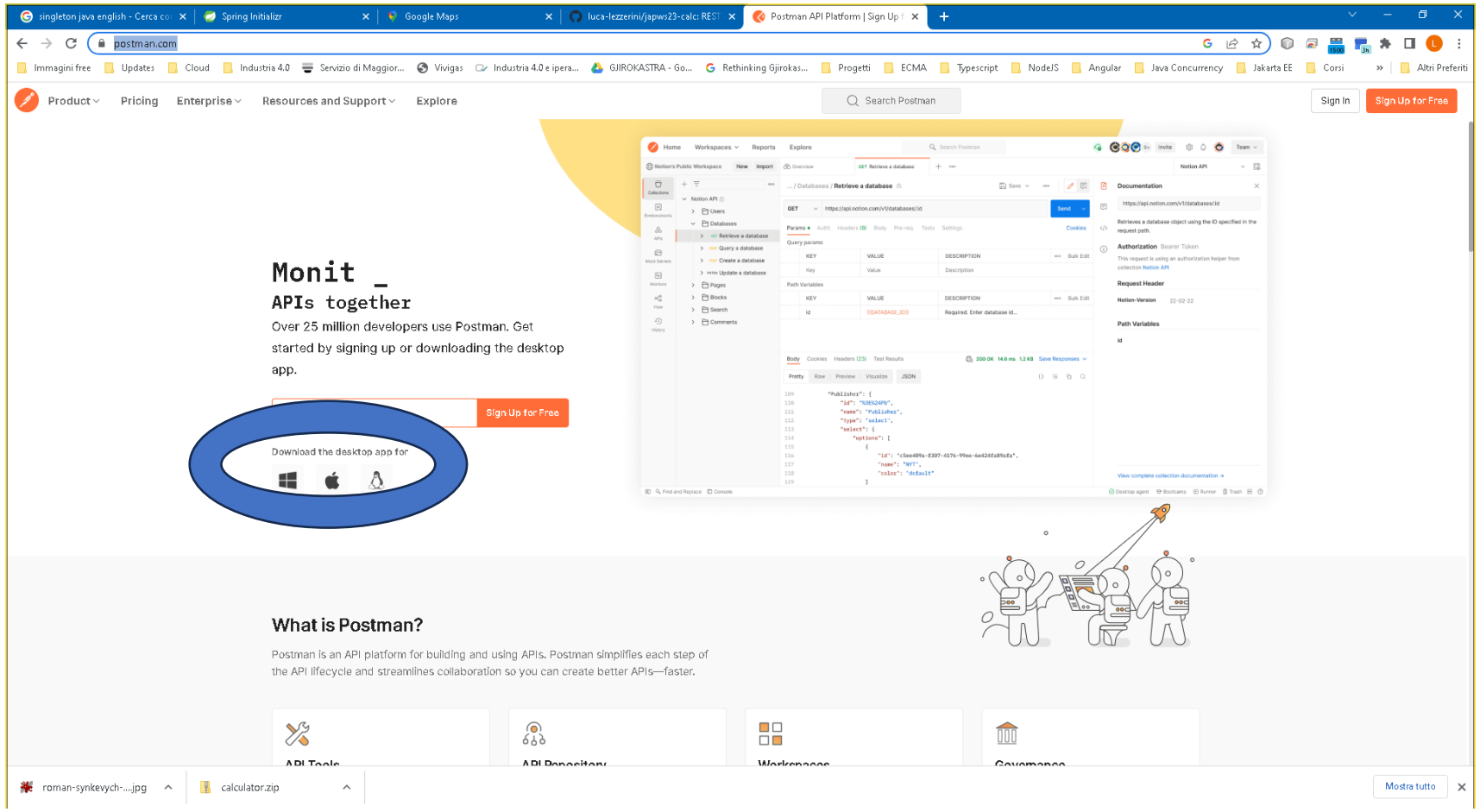
Create a local copy of repository

It is called «cloning»

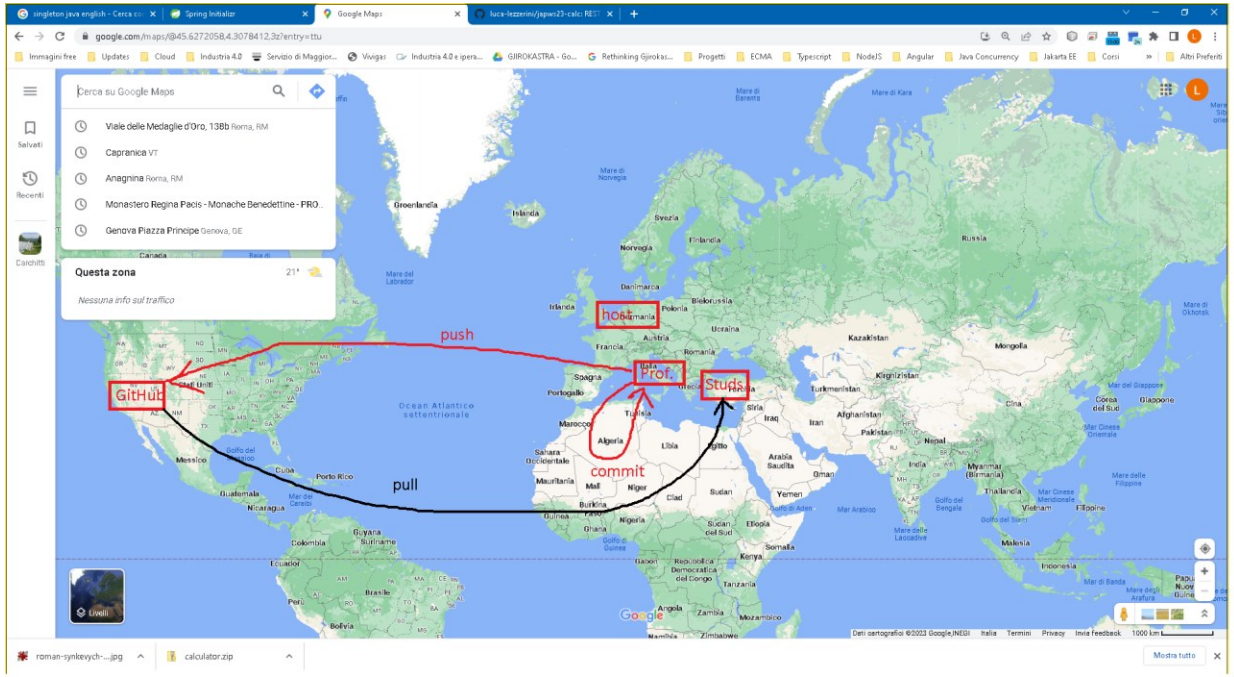


Postman

- Is an HTTP client
- Is used for testing and debugging
- Very powerful and easy to use



GithHub& hosting



Create a REST service in Spring

Steps to create a REST service

1. Define the http request
2. Define the request's DTO
3. Define the repsonse's DTO
4. Create a service to execute required processing
    1. Create a service interface
    2. Create a service implementation class
5. Create a controller's method with GET or POST mapping

Define the http request

- http request has the following pattern:
  - <protocol> <domain>:<port> **<request>**

  - E.g.
    - <protocol> is http:// or https://
    - <domain> is polisapp.al or 84.123.78.99
    - <port> is 8080
    - <request> is something like:
      - /sum
      - /listAll
      - /delete-all
      - …

Define the request's DTO

- For a /login request a typical DTO will be:

```java
package al.polis.calculator.dto;

import lombok.Data;

@Data
public class LoginReqDto {

    private String username;
    private String password;
}
```

Define the repsonse's DTO

- The response DTO for a /login request will be:

```java
package al.polis.calculator.dto;

import lombok.Data;

@Data
public class LoginRespDto {
    private String token;
}
```

Create a service interface

```java
package al.polis.calculator.service;

public interface SecurityService {

    /**
     * This method authenticate the user polis with password isthebest
     * @param username the username to be authenticated
     * @param password the password to authenticate
     * @return the OAUTH2 token (Bearer wehf4q8fhafjdfhwu...) if authenticated or empty string if not
     */
    String login(String username, String password);
}
```

Create a service implementation class

```java
@Service
public class SecurityServiceImpl implements SecurityService {

    // random number generator
    private Random rnd = new Random();

    @Override
    public String login(String username, String password) {
        if (username == null || password == null) {
            return ""; // no username or password -> no token!
        }

        if (username.equalsIgnoreCase("polis") && password.equalsIgnoreCase("isthebest")) {
            long tknumber = rnd.nextLong();
            String token = "Bearer " + Long.toHexString(tknumber);
            return token;
        } else {
            return "";
        }
    }

}
```

Create a controller's method with GET or POST mapping

- The controller method structure is:
  - @PostMapping o @GetMapping annotation with request (@PostMapping(«/login»)
  - Method declaration:
    - @ResponseBody public LoginRespDto login(@RequestBody LoginReqDto req)
  - Inject the service needed
  - Method body (simply calls the service's method after injection)

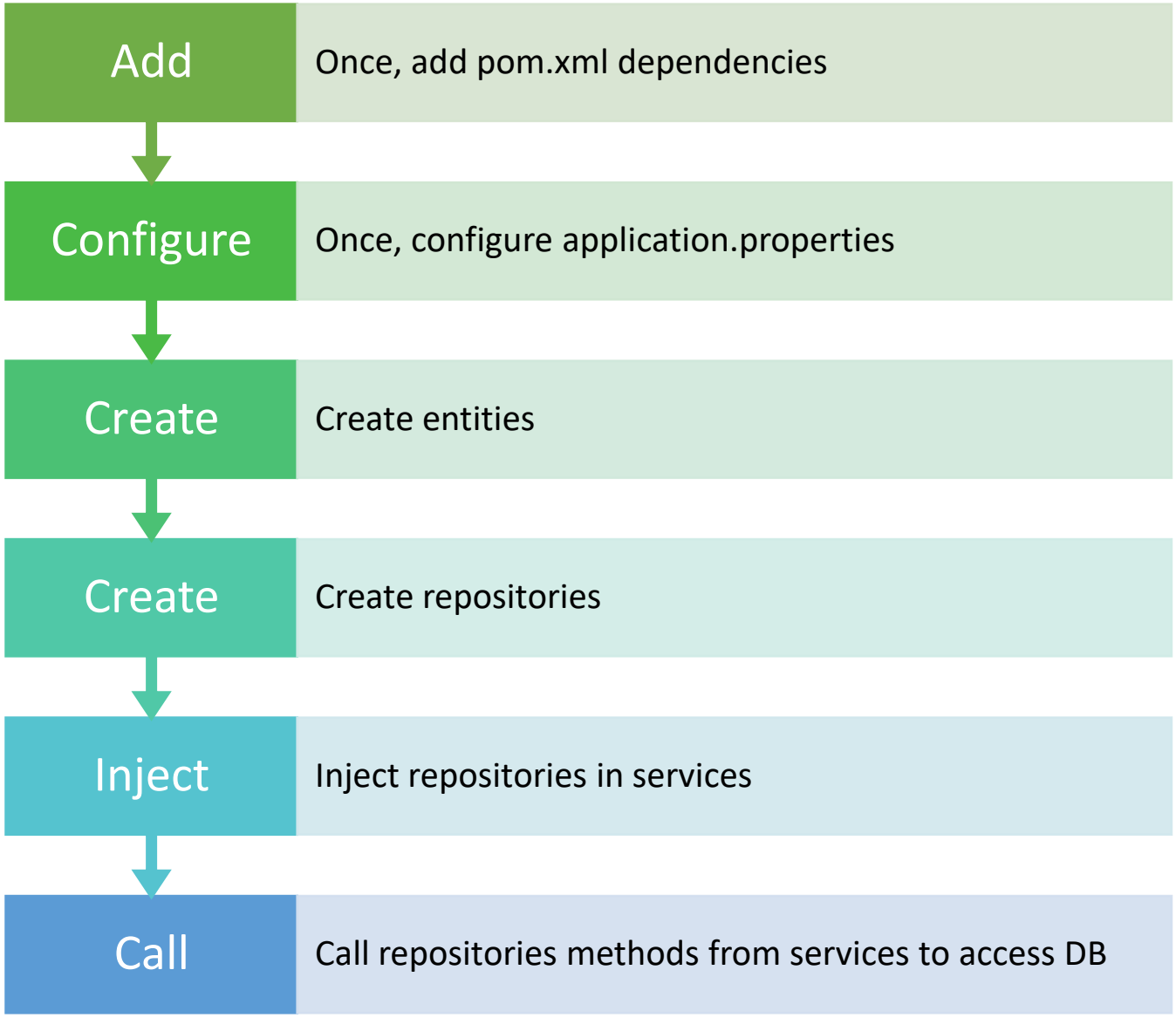
```java
/**
 * This method will login with userame and password and return an OAUTH2
 * token
 */
@PostMapping("/login")
@ResponseBody
public LoginRespDto login(@RequestBody LoginReqDto req) {
    String token = securityService.login(req.getUsername(), req.getPassword());
    LoginRespDto resp = new LoginRespDto();
    resp.setToken(token);
    return resp;
}
```

```java
@RestController
public class CalculatorController {

    @Autowired
    private CalculatorService calculatorService;
    @Autowired
    private SecurityService securityService;
```

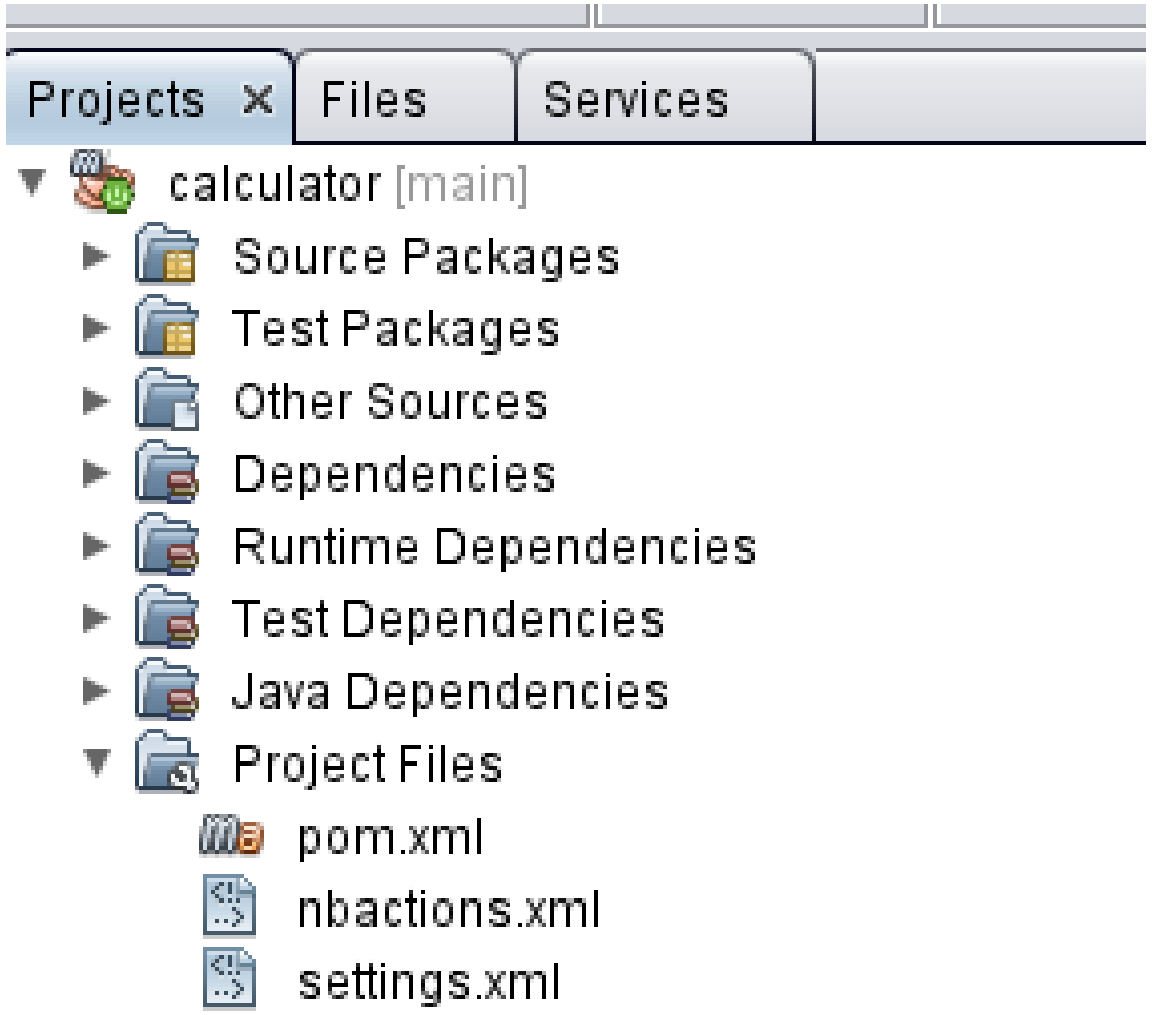
RDBMS Access using JPA

Steps (basic version)



Once, add pom.xml dependencies

- In pom.xml add the following dependencies:
- <dependency>
- <groupId>com.mysql</groupId>
- <artifactId>mysql-connector-j</artifactId>
- <scope>runtime</scope>
- </dependency>
- <dependency>
- <groupId>org.springframework.boot</groupId>
- <artifactId>spring-boot-starter-data-jpa</artifactId>
- </dependency>



Once, configure application.properties

- # configuration data source (MySQL)
- spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
- spring.datasource.url=jdbc:mysql://localhost:3306/**calculatordb**?createDatabaseIfNotExist=true&autoReconnect=true
- spring.datasource.username=**dbuser**
- spring.datasource.password=**dbuser**

- spring.jpa.properties.hibernate.id.new_generator_mappings=true
- # non cambiare il dialetto altrimenti smette di aggiornare la struttura del DB
- spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
- spring.jpa.properties.hibernate.dialect.storage_engine=innodb
- spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect

- # Hibernate debugging
- logging.level.org.hibernate.SQL=DEBUG
- logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
- spring.jpa.properties.hibernate.show_sql=true
- spring.jpa.properties.hibernate.use_sql_comments=true
- spring.jpa.properties.hibernate.format_sql=true
- spring.jpa.properties.hibernate.type=trace

- spring.jpa.generate-ddl=true
- spring.jpa.hibernate.ddl-auto=update

Create entities

```
1   package al.polis.calculator.model;
2
3   import jakarta.persistence.Column;
4   import jakarta.persistence.Entity;
5   import jakarta.persistence.GeneratedValue;
6   import jakarta.persistence.Id;
7   import lombok.Data;
8
9   @Entity
10  @Data
11  public class CalculatorRow {
```

all entities are in model package

Entities are annotated with @Entity

to avoid writing getters and setters we use lombok with @Data annotation

```
    @Id
    @GeneratedValue
    private Long id;
    @Column
    private double firstNumber;
    @Column
    private double secondNumber;
    @Column
    private double result;
    @Column
    private String operator;


}
```

If the primary key must be autogenerated by JPA the @GeneratedValue must be used

a primary key MUST eb defined with @Id

each field is a class property (@Column is optional)

Create repositories/1

```
@Repository
public interface CalculatorRowRepository extends
JpaRepository<CalculatorRow, Long>{
}
```

Repository is an **INTERFACE** annotated as @Repository

It **must extend JpaRespository** that enables many methods generating their code

JpaRepository is a **generic type** with two parameters …

Create repositories/2

Extending JpaRespository means passing two type parameters:

1st parameter is the type of the entity that is accessed by the repository (e.g. CalculatorRow)

```
@Repository
public interface CalculatorRowRepository extends JpaRepository<CalculatorRow, Long>{

}
```

```
@Entity
@Data
public class CalculatorRow {

    @Id
    @GeneratedValue
    private Long id;
    @Column
```

2nd parameter is the type of the primary key of the entity (e.g. Long)

JpaRepository Javadoc

Extending JpaReosiory gives the following ready-made methods that can be used without need of writing code: simply inject (@Autowired) the repository and call them

All Methods | Instance Methods | Abstract Methods | Default Methods | Deprecated Methods

Modifier and Type	Method	Description
void	deleteAllByIdInBatch(Iterable <ID> ids)	Deletes the entities identified by the given ids using a single query.
void	deleteAllInBatch()	Deletes all entities in a batch call.
void	deleteAllInBatch(Iterable <T> entities)	Deletes the given entities in a batch which means it will create a single query.
default void	deleteInBatch(Iterable <T> entities)	**Deprecated.** Use deleteAllInBatch(Iterable) instead.
<S extends T> List <S>	findAll(Example <S> example)	
<S extends T> List <S>	findAll(Example <S> example, Sort sort)	
void	flush()	Flushes all pending changes to the database.
T	getById(ID id)	**Deprecated.** use getReferenceById(ID) instead.
T	getOne(ID id)	**Deprecated.** use getReferenceById(ID) instead.
T	getReferenceById(ID id)	Returns a reference to the entity with the given identifier.
<S extends T> List <S>	saveAllAndFlush(Iterable <S> entities)	Saves all entities and flushes changes instantly.
<S extends T> S	saveAndFlush(S entity)	Saves an entity and flushes changes instantly.

Methods inherited from interface org.springframework.data.repository.CrudRepository

count , delete , deleteAll , deleteAll , deleteAllById , deleteById , existsById , findById , save

Methods inherited from interface org.springframework.data.repository.ListCrudRepository

findAll , findAllById , saveAll

Methods inherited from interface org.springframework.data.repository.ListPagingAndSortingRepository

findAll

Methods inherited from interface org.springframework.data.repository.PagingAndSortingRepository

findAll

Methods inherited from interface org.springframework.data.repository.query.QueryByExampleExecutor

count , exists , findAll , findBy , findOne

Inject repositories in services

- You inject (green)
- Then you can call its methos (light blue)

```
@Service
public class CalculatorServiceImpl implements CalculatorService {

    @Autowired
    CalculatorRowRepository calculatorRowRepository;

    @Override
    public double sum(double a, double b) {
        String operator = "+";
        CalculatorRow cr = new CalculatorRow();
        cr.setFirstNumber(a);
        cr.setSecondNumber(b);
        final double result = a + b;
        cr.setResult(result);
        cr.setOperator(operator);
        calculatorRowRepository.save(cr);

        return result;

    }
```

repository's method invocation

Repository injection

```
@Service
public class CalculatorServiceImpl implements CalculatorService {

    @Autowired
    CalculatorRowRepository calculatorRowRepository;

    @Override
    public double sum(double a, double b) {
        String operator = "+";
        CalculatorRow cr = new CalculatorRow();
        cr.setFirstNumber(a);
```

Call repositories methods from services to access DB

- Using two repositories:
  - Inject all needed repositories
  - Invoke their methods

```java
@Service
public class RegisterServiceImpl implements RegisterService {

    // FIXME: definire la scadenza e la politica di scadenza
    public static final int TOKEN_EXPIRATION_TIME_MINUTES = 60 * 24 * 365 * 100;

//    @Value("${uel.server.host}")
//    private String urlBase;
    @Autowired
    private UelEnvironment uelEnvironment;
    @Autowired
    private SecurityConfig securityConfig;
    @Autowired
    private PasswordEncoder passwordEncoder;
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private AccessTokenRepository accessTokenRepository;
    @Autowired
    private MembroRepository membroRepository;
    @Autowired
    private LoginProfileRepository loginProfileRepository;
```

```java
ux.setPassword(passwordEncoder.encode(dto.getPassword()));
ux = userRepository.save(ux);

var membro = new Membro();
membro.setName(ux.getUsername());
membro.setPrivacyObbligatorio(dto.getPrivacyObbligatorio());
membro.setUtente(ux);
membro = membroRepository.save(membro);
ux.setMembro(membro);
userRepository.save(ux);
```

Next steps

Use different JPA default methods (findById, findAll, …)

Create our methods to query with conditions (e.g. select * from customer where customer.name = «ledio»)

Use JPQL to create complex queries

Use different JPA default methods (findById, findAll, …)