Java Spring Data:

Spring Data is a part of the larger Spring Framework that simplifies data access and persistence. It provides a higher-level, more convenient way to work with data sources like relational databases, NoSQL databases, and more. Spring Data includes various subprojects like Spring Data JPA, Spring Data MongoDB, Spring Data Redis, and others.

Let's focus on an example using **Spring Data JPA**, which is a subproject designed for working with relational databases and JPA (Java Persistence API).

Example 1: Spring Data JPA

Suppose you have an entity class `User` representing a user in your database, and you want to perform CRUD operations on it using Spring Data JPA.

```java
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class User {
    @Id
    private Long id;
    private String username;
    private String email;
    // Getters and setters
}
```

Now, create a repository interface that extends `JpaRepository` to inherit common CRUD methods without writing any implementation code:

```java
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    // You can add custom query methods here if needed
}
```

With this repository, you can easily perform operations like saving, updating, deleting, and querying `User` entities in your database without writing SQL queries.

Spring Security:

Spring Security is a powerful authentication and access-control framework for securing web applications. It provides various features like authentication, authorization, protection against common web application security vulnerabilities, and more.

Example 2: Spring Security

Suppose you want to secure a RESTful API using Spring Security. Here's an example:

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .inMemoryAuthentication()
            .withUser("user").password(passwordEncoder().encode("password")).roles("USER");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/public/**").permitAll()
                .antMatchers("/private/**").hasRole("USER")
                .and()
            .httpBasic()
                .and()
            .csrf().disable();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

In this example:

- We configure an `AuthenticationManager` to have a single in-memory user.
- We configure HTTP security to allow unauthenticated access to URLs under `/public/` and require the "USER" role for URLs under `/private/`.
- We enable HTTP Basic Authentication for user login.
- We disable CSRF protection for simplicity.

This is a basic example. In a real application, you might use a database-backed user store and more advanced security features.

These examples provide a high-level overview of Spring Data and Spring Security. They are powerful tools for data access and security in Java-based applications, allowing you to focus on application logic while Spring takes care of the boilerplate code for data access and security.