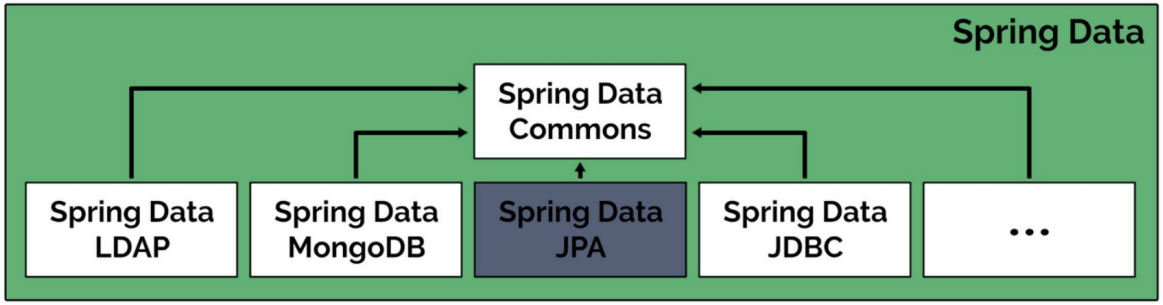
# Spring Data

- Spring Data
  - Conceptos principales
  - Repositorios
    - Por nombre de método
    - Genéricos
    - Customizados
- Integración Spring MVC & Spring Data
- Spring Boot

# Spring Data
## Conceptos principales

- Spring Data es el nombre de un módulo de Spring cuyo objetivo es facilitar el acceso y uso de datos en aplicaciones basadas en Spring.
- Los datos pueden provenir de orígenes tan dispares como bases de datos relacionales a través de JPA o Spring Data JDBC, bases de datos NoSQL como MongoDB o Neo4J, o directorios LDAP.




- Spring Data JPA usa JPA (Java Persistence API) y simplifica las operaciones comunes de acceso a datos, al mismo tiempo que mejora el soporte de JPA con funcionalidades nuevas y potentes. Facilita la implementación de repositorios basados en JPA.
- El objetivo de la abstracción de Spring Data es reducir la cantidad de código repetitivo necesario para implementar una capa de acceso a datos para diferentes repositorios.

# Spring Data
## Conceptos principales

Con Spring Data -> No mas implementaciones DAO.

Con Spring Data JPA, podemos crear una interfaz que incluya diferentes métodos que sigan un **patrón de nombres**, y el framework desarrollará la implementación automáticamente por nosotros.

Para aprovechar el modelo de programación Spring Data con JPA, la interfaz a crear debe extender la interfaz del repositorio específica de JPA, `JpaRepository`. Esto permitirá que Spring Data cree automáticamente una implementación para ella.

El gran objetivo es reducir al máximo la escritura de código.

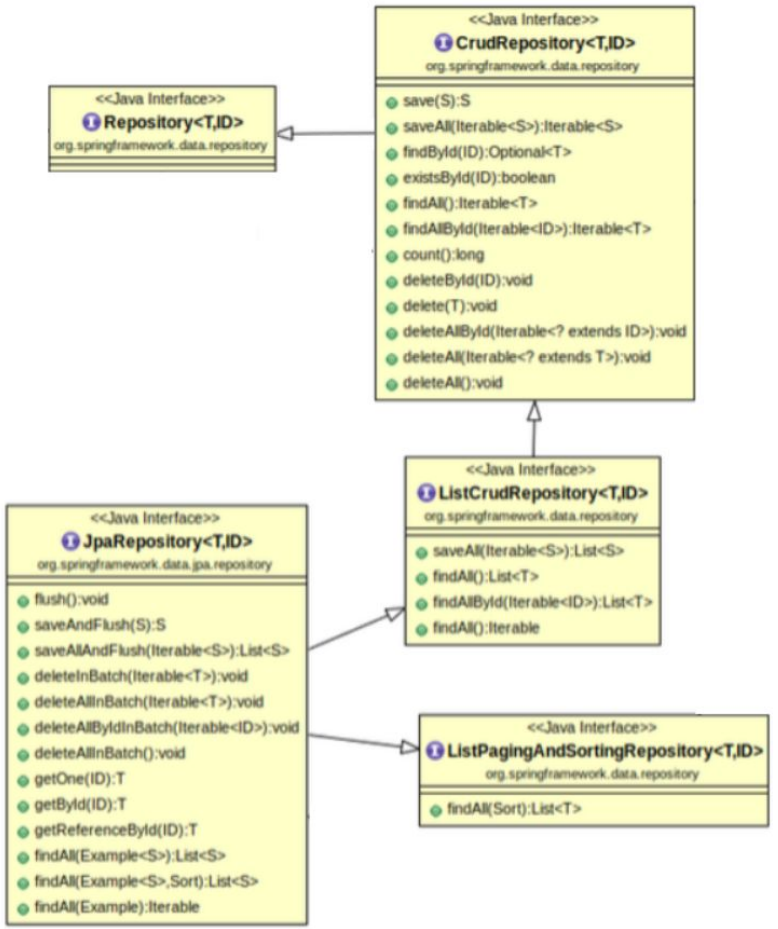
# Spring Data

## Conceptos principales

Spring Data se basa en un conjunto de interfaces. La interfaz central en Spring Data es `Repository`, la cual tiene dos parámetros: el tipo de la clase y el tipo del ID de la clase y ningún método. La interfaz `CrudRepository` provee la funcionalidad de un CRUD sofisticado para la clase (entity) que maneja.

```
public interface Repository<T, ID> {

}
```

```
public interface CrudRepository<T,ID>
                                extends Repository<T,ID> {

  <S extends T> S save(S entity);

  Iterable<S> saveAll(Iterable<S>);

  Optional<T> findById(ID primaryKey);

  Iterable<T> findAll();

  long count();

  void delete(T entity);

  boolean existsById(ID primaryKey);

  boolean deleteById(ID primaryKey);

  // ... mas funcionalidades

}
```



# Spring Data
## Conceptos principales

Una vez definido el modelo de objetos y decorado con las anotaciones de JPA, se puede comenzar a definir los repositorios utilizando Spring Data JPA. Para cada entidad JPA con la que queramos trabajar, en lugar del tradicional DAO, se creará un "repositorio" de Spring Data que no es más que una interfaz que especializa `JpaRepository`. Esta interfaz proporciona las operaciones CRUD más habituales y que por lo tanto no tendremos que implementar.

Extendiendo la interfaz JpaRepository obtenemos la implementación de los métodos CRUD más relevantes para acceder de manera estándar a los datos.

Para definir métodos de acceso más específicos, Spring JPA admite opciones como:

- definir nuevos métodos en la interfaz usando palabras claves
- proporcionar una consulta JPQL utilizando la anotación @Query
- definir queries personalizadas a través de @NamedQuery

# Spring Data
## Repositorio Básico

Para cada entidad JPA se debe crear un "repositorio" de Spring Data que es una subinterface especializada de la interfaz **`JpaRepository`**. Esta interfaz proporciona las operaciones CRUD habituales, las cuales no se deben implementar.

```
@Entity
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private String dni;
    . . .
    public String getName(){
     return name;
    }
    public void setName(String name){
     this.name=name;
    }
}
```

La convención a seguir para nombrar el repositorio es utilizar el nombre de la entidad con el sufijo "Repository". Asimismo, hay que indicar la entidad para la que se crea el repositorio y el tipo de su clave primaria.

```
public interface UserRepository extends JpaRepository<User, Long> {
}
```

Con esta simple definición, Spring se encarga de la implementación de las operaciones.

# Spring Data

## Repositorio Básico

La interface **UserRepository** extiende de **`JpaRepository`** entonces hereda varios métodos para recuperar, guardar, actualizar y borrar entidades.

```
public interface UserRepository extends JpaRepository<User, Long> {
}
```

Algunos métodos heredados cuando se extiende JpaRepository que permiten las operaciones CRUD son:

```
Optional<T> findById(ID id);
```

El método **findById** recupera una entidad por id y devuelve un contenedor Opcional, que indica que la entidad puede existir o no, se incluirá si existe sino estará vacío.

```
T getById(ID id);
```

El método **getById** devuelve la entidad directamente. Si la entidad no existe en la base de datos, se generará una **EntityNotFoundException**.

```
T getReferenceById(ID id);
```

El método **getReferenceById** devuelve un objeto proxy que representa a la entidad y que solo contiene su identificador. No busca en la DB, no se sabe si existe o no, pero esa entidad responde a todos los getters.

```
T save(T entity);
```

El método **save** guarda o actualiza una entidad en la base de datos. Internamente invocará al método persist o bien al método merge.

```
T delete(T entity);
```

El método **delete** elimina la entidad recibida por parámetro.

# JPA Spring Data

## Especialización de repositorios - Queries Automáticas

La primera forma para extender el repositorio sin codificar es agregar métodos en la interfaz, usando convenciones en los nombres de los métodos. Existe un amplio conjunto de palabras claves para utilizar. Cuando Spring crea una implementación de `Repository`, analiza los métodos definidos en la interfaz y trata de generar las queries automáticamente desde los nombres de métodos.

```
public interface UserRepository extends JpaRepository<User, Long> {
    ????
}
```

```
@Entity
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private String dni;
    . . .
    public String getName(){
     return name;
    }
    public void setName(String name){
     this.name=name;
    }
}
```

# JPA Spring Data

## Especialización de repositorios - Queries automáticas

**findBy** es uno de los prefijos más utilizados para crear métodos que deriven automáticamente en queries de búsquedas.

El verbo **"find"** le dice a Spring Data que genere una consulta de tipo SELECT. Por otro lado, la palabra clave **"By"** actúa como cláusula WHERE ya que filtra el resultado devuelto.

A continuación, agreguemos un método de consulta a nuestro **UserRepository** para que obtenga a un User por su nombre en nuestro UserRepository. Supongamos que insertamos estos datos para prueba.

```
INSERT INTO User (id, first_name, last_name, dni) VALUES('Juan', 'Perez', '23450876');
INSERT INTO User (id, first_name, last_name, dni) VALUES('Sol', 'Rocha', '19087444');
INSERT INTO User (id, first_name, last_name, dni) VALUES('Lucia', 'Rocha', '23087444');
```

```
public interface UserRepository extends JpaRepository<User, Long> {
    User findByFirstName(String firstName);
}
```

```
@Test
void givenFirstName_whenCallingFindByFirstName_ThenReturnOneUser() {
    User user = userRepository.findByFirstName("Sol");
    assertNotNull(user);
    assertEquals("Sol", user.getFirstName());
}
```

# JPA Spring Data

## Especialización de repositorios - Queries automáticas

Ahora que hemos visto cómo usar **findBy** para crear un método de consulta que devuelve un único objeto, veamos si podemos usarlo para obtener una lista de objetos.

Para hacerlo, agregaremos otro método de consulta a nuestro **UserRepository** para que nos devuelva una lista de Users que coinciden con un lastName particular:

```
public interface UserRepository extends JpaRepository<User, Long> {
     List<User> findByLastName(String lastName);
}
```

De manera similar, podemos hacer un test:

```
@Test
void givenLastName_whenCallingFindByLastName_ThenReturnUserList() {
    List<User> users = userRepository.findByLastName("Rocha");
    assertNotNull(users);
    assertEquals(2, users.size());
}
```

Como vimos, podemos usar **findBy** para obtener un objeto o una colección de objetos. Lo que marca la diferencia aquí es el tipo de retorno de los métodos de consulta.

# JPA Spring Data

## Especialización de repositorios - Queries automáticas

El nombre debe tener un prefijo (findBy, exist, getBy, etc.) que establece el tipo de consulta, seguido de los criterios de selección de los resultados y, con carácter opcional, de los criterios de ordenación (and, or, after, etc.) admitidas por el mecanismo de derivación de consultas del repositorio Spring Data.

*Table 1. Supported keywords inside method names*

| Keyword | Sample | JPQL snippet |
|---|---|---|
| `Distinct` | `findDistinctByLastnameAndFirstname` | `select distinct … where x.lastname = ?1 and x.firstname = ?2` |
| `And` | `findByLastnameAndFirstname` | `… where x.lastname = ?1 and x.firstname = ?2` |
| `Or` | `findByLastnameOrFirstname` | `… where x.lastname = ?1 or x.firstname = ?2` |

Documentación https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

Spring parsea los métodos y genera automáticamente las queries respectivas.

```
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByFirstName(String firstName);
    Optional<User> findByFirstNameAndLastName(String firstName, String lastName);
    boolean existsByDni(String dni);
    List<User> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

# JPA Spring Data

## Especialización de repositorios - Queries personalizada JPQL

Otra manera de extender la interfaz es proporcionar la consulta JPQL utilizando la anotación `@Query`. Los parámetros de la consulta serán los parámetros que reciba el método.

a. Los parámetros que recibe el método son referenciados en la consulta según el orden en el que aparecen en la firma del método.

```
public interface UserRepository extends JpaRepository<User, Long> {
     @Query("SELECT u FROM User u WHERE LOWER(u.firstname)=LOWER(?1)")
     Optional<User> retrieveByFirstName(@Param("name") String name);
}
```

b. Los parámetros se pueden poner por nombre, utilizando un alias para los mismos y la anotación `@Param`:

```
public interface UserRepository extends JpaRepository<User, Long> {
     @Query("SELECT u FROM User u WHERE LOWER(u.firstname)=LOWER(:name)")
     Optional<User> retrieveByFirstName(@Param("name") String name);
}
```

# JPA Spring Data

## Especialización de repositorios - Queries @NamedQuery

Otra manera de extender la interfaz es utilizar una `NamedQuery` definida en una entidad. En lugar de escribir la consulta directamente en el código del repositorio (como con @Query), se define en la clase de entidad y luego se puede invocar por su nombre.

```
@Entity
@Table(name="User")
@NamedQuery(name = "User.byDniNamedQuery", query = "SELECT u FROM User WHERE dni=?1")
public class User {
     private String firstName;
     private String lastName;
     private String dni;
     ...
}
```

debe tener como prefijo el nombre de la entidad seguido de un punto.

El nombre del método que la ejecuta tiene que coincidir con el de la `NamedQuery`.

```
public interface UserRepository extends JpaRepository<User, Long> {
     Optional<User> byDniNamedQuery(String dni);
}
```

También podemos obviar esta convención y simplemente indicar el nombre de la consulta.

```
public interface UserRepository extends JpaRepository<User, Long> {
     @Query(name = "User.byDniNamedQuery")
     Optional<User> cualquierNombre(String dni);
}
```

# JPA Spring Data

## Especialización de repositorios - Queries @NamedQuery

¿Cómo sabe Spring Data que el método `byDniNamedQuery(String dni)` corresponde a una *NamedQuery* y no intenta generar una *consulta derivada automática*?

debe tener como prefijo el nombre de la entidad seguido de un punto.

```
@Entity
@Table(name="User")
@NamedQuery(name = "User.byDniNamedQuery", query = "SELECT u FROM User WHERE dni=?1")
public class User {
     private String firstName;
     private String lastName;
     private String dni;
      ...
}
```

**Esta consulta se "registra" automáticamente cuando arranca JPA (al inicializar la entidad User).**

**Orden de resolución de consultas en Spring Data JPA**

1. **Busca si existe una** `@Query` **anotada** → si la encuentra en un método, la usa directamente.
2. **Busca una** `@NamedQuery` **registrada con el nombre correspondiente.**
3. Si no hay ninguna, intenta **derivar la consulta automáticamente del nombre del método** (por ejemplo, `findByEmail`, `existsById`, etc.).

**Qué pasaría en nuestro caso si no tenemos anotaciones?** Como `byDniNamedQuery` no sigue el patrón `findBy`, `getBy`, `existsBy`, etc., lanzará una excepción al iniciar la app:

```
InvalidDataAccessApiUsageException: Could not create query for public abstract
java.util.Optional ...
No property byDniNamedQuery found for type User!
```

# JPA Spring Data

## Integración con Spring Rest - Inyección del repo en el Controller

```
@RestController
@RequestMapping("/user")
public class UserController {
 @Autowired
 private UserRepository userRepository;

 @PostMapping
 public User create(@RequestBody User user) {
      //validaciones
      return userRepository.save(user);
 }

 @PutMapping(value = "/{id}")
 public User update(@RequestBody User user, @PathVariable("id") Long userId) {
 User userDB = userRepository.findById(userId)
  .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Usuario no encontrado"));
    // Actualizamos solo los campos necesarios
    userDB.setFirstName(user.getFirstName());
    userDB.setLastName(user.getLastName());
    userDB.setDni(user.getDni());
    return userRepository.save(userDB);
  }
  ...
}
```

UserRepository hereda las funcionalidades de JpaRepository:
`save()`, `findById()`, `deleteById(id)`, ...

# JPA Spring Data

## Integración con Spring Rest - Inyección del repo en el Controller

```java
@RestController
@RequestMapping("/user")
public class UserController {
   @Autowired
   private UserRepository userRepository;

   @PostMapping
   public User create(@RequestBody User user) {
      //validaciones
      return userRepository.save(user);
   }

 @PutMapping(value = "/{id}")
 public User update(@RequestBody User user, @PathVariable("id") Long userId) {
 User userDB = userRepository.findByIdById(userId)
  .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Usuario no encontrado"));
    // Actualizamos solo los campos necesarios
    userDB.setFirstName(user.getFirstName());
    userDB.setLastName(user.getLastName());
    userDB.setDni(user.getDni()); // si aplica
    return userRepository.save(userDB);
  }
  ...
}
```

qué hace el save() se hace persist o merge?

findById(userId) devuelve un Optional<User>, porque se guarda en User?

qué hace exactamente save(userDB)?

**Nada adicional en este caso.**
Como el objeto ya está dentro del *persistence context*, `save()` **no invoca** `persist()` **ni** `merge()` — el `EntityManager` simplemente mantiene el objeto y lo actualizará al final de la transacción cuando ocurre el flush().

# JPA Spring Data

## Integración con Spring Rest - Inyección del repo con una capa de Servicio

```java
@RestController
@RequestMapping("/user")
public class UserController {
   @Autowired
   private UserService userService;

   @PostMapping
   public User create(@RequestBody User user) {
      //validaciones
      return this.userService.create(user);
   }

  @PutMapping(value = "/{id}")
  public User update(@RequestBody User user,
              @PathVariable("id") Long userId) {
      return this.userService.update(user, id);
  }
  …

}
```

```java
@Service
@Transactional
public class UserService {
 @Autowired
 private UserRepository userRepository;

 public boolean existsByDNI(String dni) {
      return userRepository.existsByDni(dni);
 }

 public User create(User user) {
      // validaciones
      return userRepository.save(user);
 }

 public void setNewPassword(User user,String newPass) {
    user.setPassword(bCryptPasswordEncoder.encode(newPass));
    userRepository.save(user);
 }
  ...
}
```

```java
public interface UserRepository extends JpaRepository<User, Long> {
      User findByFirstName(String firstName);
      @Query("SELECT u FROM User u WHERE LOWER(u.firstname)=LOWER(?1)")
      User retrieveByFirstName(@Param("name") String name);
      . . .
 }
```

Hereda las funcionalidades de JpaRepository:
`save()`,
`deleteById(id)`, . . .

# Spring Boot

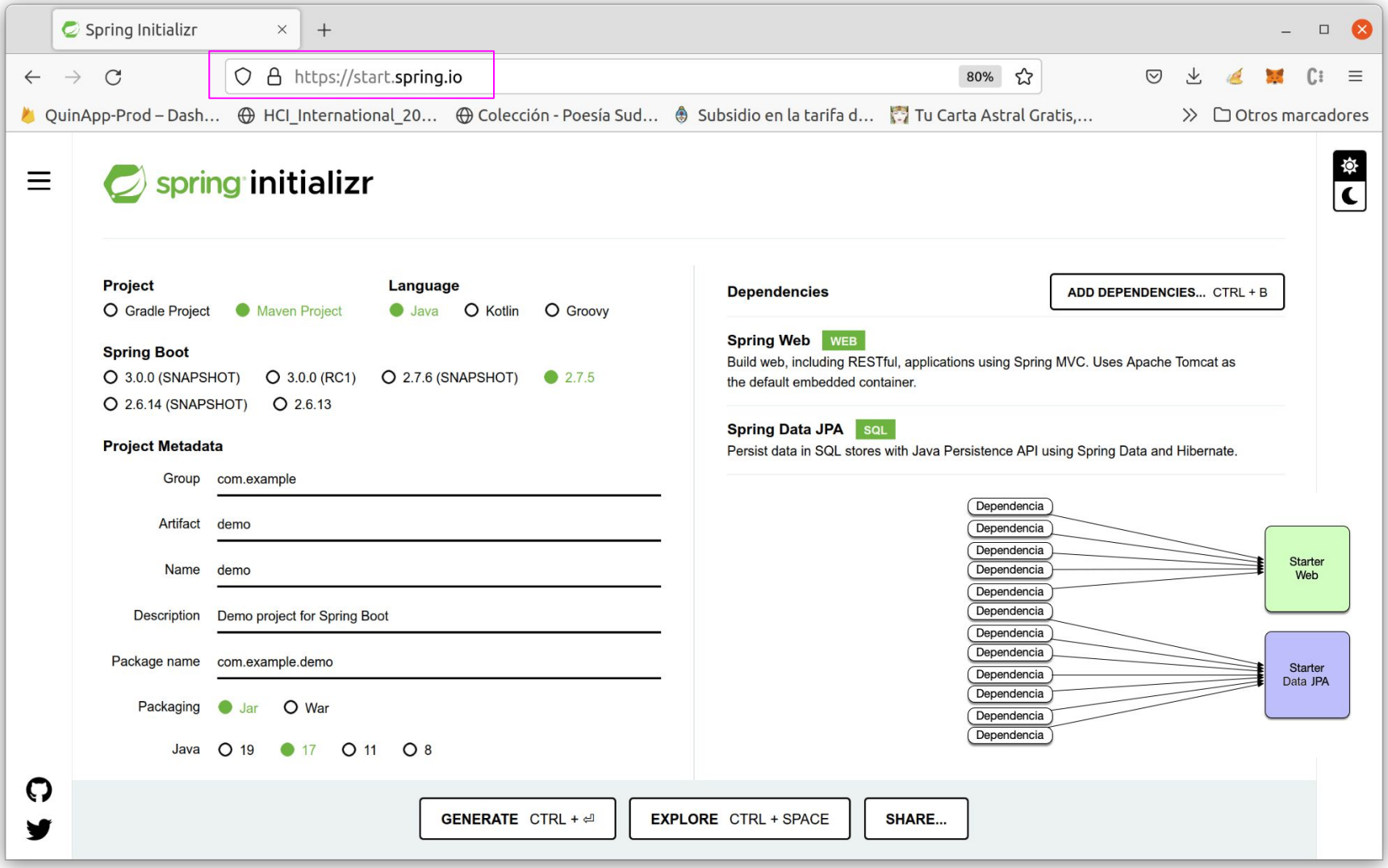
Spring Boot proporciona una manera rápida de construir aplicaciones. Se utiliza para simplificar y acelerar el desarrollo de aplicaciones basadas en Spring gracias a una gestión automatizada de la infraestructura de software que necesitamos en nuestra aplicación. Se centra en la configuración e integración de las dependencias.

---

## Características principales

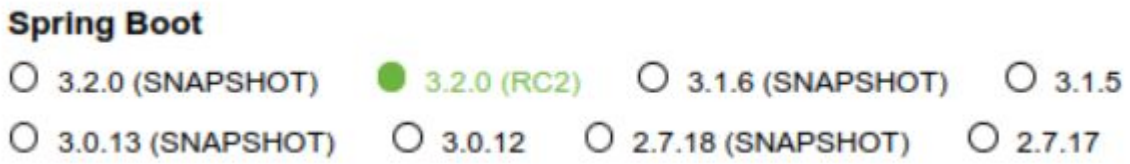
- Template inicial configurable via Web ( https://start.spring.io )
- Integra Tomcat, Jetty o Undertow directamente (sin necesidad de desplegar archivos WAR)
- Proporciona dependencias para simplificar la configuración de la compilación
- Configura automáticamente Spring y las bibliotecas de terceros siempre que sea posible
- No se genera absolutamente ningún código y no se requiere ninguna configuración XML

# Spring Boot


Spring Initializr
https://start.spring.io
80%
QuinApp-Prod – Dash...
HCI_International_20...
Colección - Poesía Sud...
Subsidio en la tarifa d...
Tu Carta Astral Gratis,...
Otros marcadores
spring initializr
Project
Gradle Project
Maven Project
Language
Java
Kotlin
Groovy
Spring Boot
3.0.0 (SNAPSHOT)
3.0.0 (RC1)
2.7.6 (SNAPSHOT)
2.7.5
2.6.14 (SNAPSHOT)
2.6.13
Project Metadata
Group
com.example
Artifact
demo
Name
demo
Description
Demo project for Spring Boot
Package name
com.example.demo
Packaging
Jar
War
Java
19
17
11
8
Dependencies
ADD DEPENDENCIES... CTRL + B
Spring Web
WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
Spring Data JPA
SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
Dependencia
Starter Web
Starter Data JPA
GENERATE CTRL + ⏎
EXPLORE CTRL + SPACE
SHARE...

# Versiones de Spring Boot & Java

En el contexto del desarrollo de software, las versiones Snapshot y RC (Release Candidate) indican diferentes etapas en el ciclo de vida del software antes de la liberación de una versión estable.




- **Versiones sin aclaración entre paréntesis** son versiones estables.
- **SNAPSHOT** se refiere a una versión en desarrollo y no estable de Spring Boot. Puede contener las últimas características, pero no está garantizada la estabilidad.
- **RC2** indica que la versión es una Release Candidate y que se está acercando al lanzamiento de la versión estable. Aunque se supone que es más estable que las versiones Snapshot, aún se recomienda utilizarla con precaución, especialmente en entornos de producción.

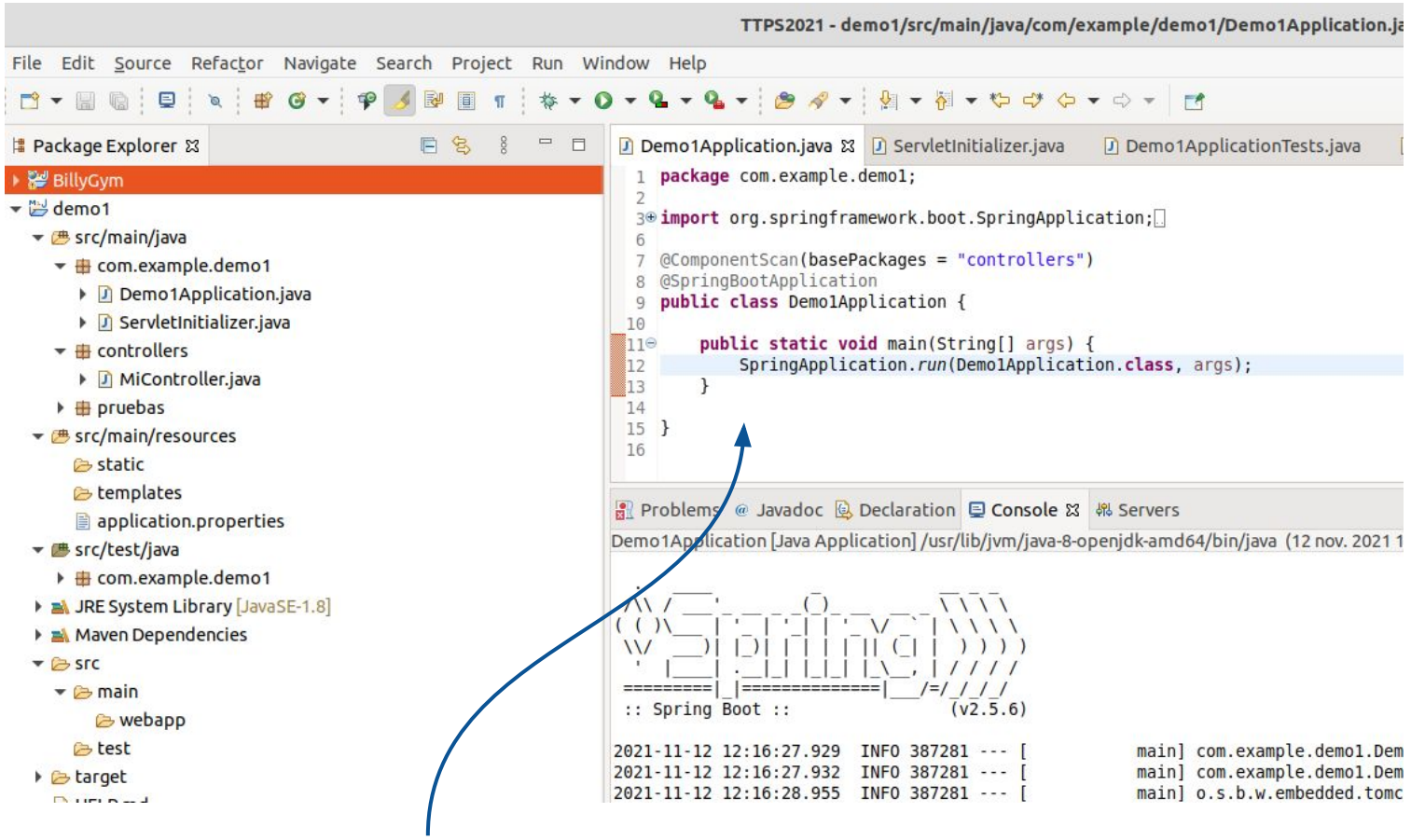
---

**Spring & Java**

- **Spring Boot 1.x:** Compatible con Java 6, 7 y 8.
- **Spring Boot 2.x:** Mayormente compatible con Java 8 y 11. A partir de la versión 2.5, Java 16 también es soportado.
- **Spring Boot 3.x:** Compatible con Java 17+ y el espacio de nombres *jakarta.* para JPA/servlets.

# Spring Boot

Luego de seleccionar los starters deseados (Web/Data JPA, en la diapo anterior), presionar GENERATE proyecto, lo que descargará un proyecto Maven en formato zip. Al descomprimir el proyecto, este será su contenido.




Esta clase es la encargada de arrancar la aplicación de Spring. A diferencia de un enfoque clásico no hace falta desplegarla en un servidor web ya que Spring Boot provee de uno.