

Práctica 2: Aplicación ToDoList

1. Objetivos

En práctica 2 vamos a trabajar sobre la aplicación inicial [domingogallardo/mads-todolist-inicial](#).

Esta parte tendrá una duración de cuatro semanas. Deberás realizarla de forma individual, siguiendo las indicaciones que encontrarás en este documento. Tendrás que desarrollar código y trabajar en GitHub desarrollando *issues*, *pull requests*, *releases* y actualizando los tableros del proyecto (en Trello y en GitHub).

Al igual que en la práctica 1 usaremos GitHub Classroom para crear un repositorio individual con el que realizarás la práctica. El proyecto base será la aplicación inicial [domingogallardo/mads-todolist-inicial](#). En este repositorio se ha seguido una metodología similar a la que vamos a utilizar en este práctica y puedes examinarlo para ver distintos elementos:

- [Issues cerrados](#)
- [Pull Requests mezclados](#)
- [Tablero de issues](#)
- [Tablero Trello de historias de usuario](#)

Debes leer la [introducción a Spring Boot](#) para entender los conceptos fundamentales del framework.

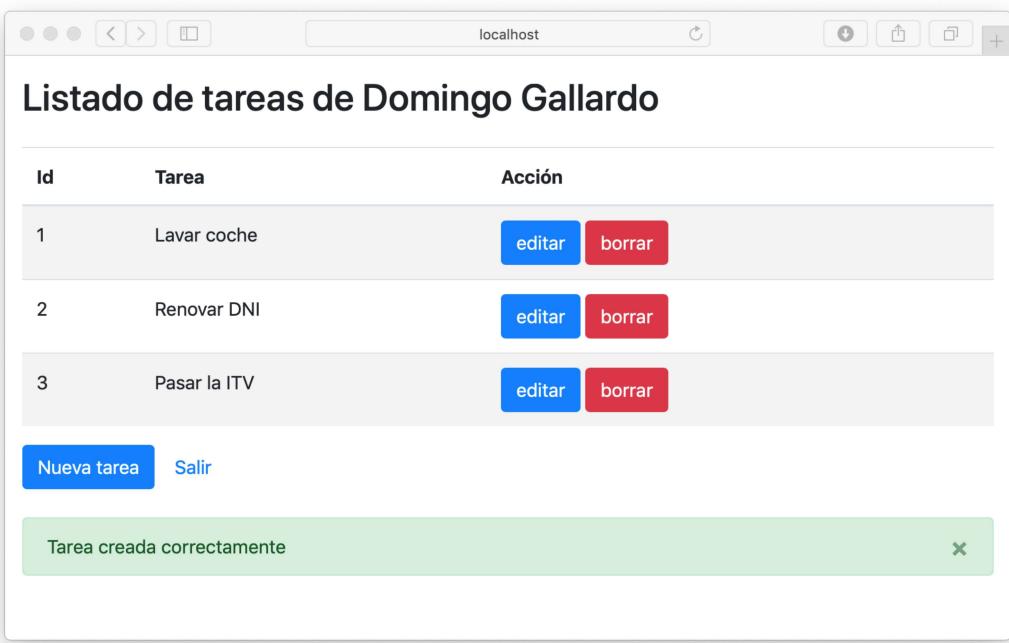
2. Aplicación inicial

La aplicación con la que vamos a trabajar es una típica aplicación ToDo que sirve para gestionar tareas pendientes. Se pueden registrar y logear usuarios y los usuarios registrados pueden añadir, modificar y borrar tareas pendientes de hacer.

A continuación puedes ver dos de las pantallas de la aplicación.



Pantalla de login



Pantalla con listado de tareas

Iremos desarrollando características adicionales de la aplicación a lo largo de las prácticas. El nombre de la aplicación es **mads-todolist**.

3. La aplicación ToDoList

La aplicación [mads-todolist-inicial](#) es la versión inicial de la aplicación que se va a desarrollar durante toda la asignatura.

Es una aplicación bastante más compleja que la vista en la práctica 1. Entre otros, tiene los siguientes elementos:

- Distintos comandos HTTP: GET, POST, DELETE.
- Recogida de datos en formularios HTML y validación de los datos.
- Base de datos gestionada con JPA (*Java Persistence API*), un ORM (*Object Relational Mapping*) implementado por la librería Hibernate. Se utiliza una capa de persistencia basada en clases *repository*.
- *Capa de servicio* que proporciona la **lógica de negocio** a los controllers.
- Las *clases controller* sólo se encargan de hacer de interfaz de la capa de servicio:
 - Recoger datos de la petición HTTP,
 - tratar y validar estas entradas,
 - llamar a la clase de servicio para que se realice la acción requerida, y
 - convertir la respuesta obtenida de la aplicación en una vista que se devuelve como respuesta de la petición.
- En las plantillas se incluye *Bootstrap* y scripts JavaScript.
- Las clases de servicio y de *repository* se obtienen por inyección de dependencias.
- Gran número de tests que prueban la capa de servicios y la de presentación.

Vamos a ver con un poco más de detalle dónde puedes encontrar en el código todos estos elementos.

Configuración de la aplicación

Los distintos parámetros de la aplicación Spring Boot se configuran un fichero de propiedades. El fichero de propiedades por defecto es `application.properties`.

Fichero `/src/main/resources/application.properties`:

```
spring.application.name = mads-todolist
spring.datasource.url=jdbc:h2:mem:dev
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
logging.level.org.hibernate.SQL=debug
logging.level.madstodolist=debug
spring.sql.init.mode=always
# cargar los datos despues de que Hibernate inicialice
# los esquemas de datos
```

```
# https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.5-Release-Notes#sql-script-datasource-initialization
spring.jpa.defer-datasource-initialization=true
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
# Deshabilitamos Open EntityManager in View
# https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/html/data.html#data.sql.jpa-and-spring-data.open-entity-manager-in-view
# Ver tambien https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/orm/hibernate5/support/OpenSessionInViewInterceptor.html

# y https://www.baeldung.com/spring-open-session-in-view
spring.jpa.open-in-view=false
```

Entre otras cosas, se define las características de la fuente de datos con la que trabaja la aplicación (la base de datos en memoria H2):

- El parámetro `spring.jpa.hibernate.ddl-auto=update` hace que Hibernate actualice automáticamente los esquemas de la base de datos, construyéndolos a partir de las clases `Entity`. En un entorno de producción el valor de esta propiedad deberá ser `validate` para no modificar la base de datos de producción.
- El parámetro `spring.sql.init.mode=always` hace que se carguen datos iniciales en la base de datos al arrancar la aplicación. El parámetro `spring.jpa.defer-datasource-initialization=true` hace que los datos se carguen después de que Hibernate haya actualizado el esquema de datos.

Los datos iniciales de la aplicación se encuentran en el fichero `data.sql`:

Fichero `/src/main/resources/data.sql`:

```
/* Populate tables */
INSERT INTO usuarios (id, email, nombre, password, fecha_nacimiento)
VALUES('1', 'user@ua', 'Usuario Ejemplo', '123', '2001-02-10');
INSERT INTO tareas (id, titulo, usuario_id) VALUES('1', 'Lavar coche', '1');
INSERT INTO tareas (id, titulo, usuario_id) VALUES('2', 'Renovar DNI', '1');
```

Base de datos H2 en memoria en desarrollo

En esta práctica vamos a trabajar únicamente con la base de datos en memoria. Esto significa que los datos que introduzcamos van a estar presentes mientras que la aplicación esté funcionando. Cuando matemos la aplicación y la volvamos a reiniciar sólo estarán los datos iniciales, los datos que se cargan del fichero `data.sql`.

En la práctica 3 utilizaremos una base de datos real, que deberemos gestionar también en producción. En concreto, se tratará de una base de datos PostgreSQL.

Por último, el parámetro `spring.jpa.open-in-view=false` deshabilita una característica de Spring denominada *open in view* que mantiene abierta la conexión de la base de datos de forma automática en cada petición HTTP. Se trata de una característica que facilita el trabajo con las relaciones *lazy* entre entidades, pero puede introducir errores no deseados al permitir acceder a la base de datos en la capa *controller*. Al deshabilitar esta característica tendremos que gestionar manualmente las relaciones *lazy*, recuperándolas en la capa de servicio, en donde sí que mantenemos abierta la conexión con la base de datos. Lo veremos más adelante con más detalle.

Otras configuraciones

Es posible definir otras configuraciones e indicar en el comando de ejecución de la aplicación Spring Boot qué fichero de configuración usar. Lo veremos en la práctica 3.

En esta práctica se define otra configuración en el directorio de test, que es la que se carga cuando se lanzan los tests:

Fichero `src/test/resources/application.properties`:

```
spring.datasource.url=jdbc:h2:mem:test
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=create
logging.level.org.hibernate.SQL=debug
# Es necesario definir el sql.init.mode a never para evitar
# que se carguen los datos de src/main/resources/data.sql
spring.sql.init.mode=never
# obligamos a que Hibernate inicialice los esquemas de datos
# https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.5-Release-
Notes#sql-script-datasource-initialization
spring.jpa.defer-datasource-initialization=true
# Deshabilitamos Open EntityManager in View
# https://docs.spring.io/spring-boot/docs/current-
SNAPSHOT/reference/html/data.html#data.sql.jpa-and-spring-data.open-entity-
manager-in-view
# Ver tambien https://docs.spring.io/spring-framework/docs/current/javadoc-
api/org/springframework/orm/hibernate5/support/OpenSessionInViewInterceptor.html

# y https://www.baeldung.com/spring-open-session-in-view
spring.jpa.open-in-view=false
```

Una diferencia con el fichero de configuración de desarrollo es el nombre de la fuente de datos, el modo del `spring.jpa.hibernate.ddl-auto`, que es `create` y el fichero de datos iniciales que se carga al ejecutar los tests.

La otra diferencia importante es que evitamos que se carguen los datos poniendo el parámetro `spring.sql.init.mode` a `never`. Cargaremos los datos de prueba manualmente en los tests.

Gestión de persistencia con JPA

Para la gestión de la persistencia de los datos en la aplicación Spring Boot usaremos [Spring Data JPA](#). Se trata de un API de Spring Boot que se construye sobre JPA (*Java Persistence API*), el ORM (*Object Relational Mapping*) estándar de Java. La implementación de JPA que utiliza Spring Boot es [Hibernate](#).

Spring Data JPA usa todos los conceptos de JPA y añade algunos adicionales que facilitan aún más su utilización, como es la definición de interfaces `Repository` con métodos CRUD estándar para las entidades.

Definición del modelo de datos

El framework JPA permite definir el esquema de la base de datos usando anotaciones en las clases denominadas de entidad. Para cada clase de entidad se define una tabla en la base de datos, con columnas que se mapean con sus atributos.

Por ejemplo, la clase `Usuario` que se lista a continuación define la tabla `usuarios` en la base de datos. Los distintos atributos (`login`, `email`, ...) se corresponden con las columnas de la tabla.

El atributo `id` se corresponde con la clave primaria de la tabla. JPA define varias estrategias para obtener esa clave primera, y se ha escogido la estrategia `@GeneratedValue(strategy = GenerationType.IDENTITY)` que define una columna que se autoincrementa en cada operación de inserción de un nuevo registro en la tabla.

Además de los atributos, en la clase se define un constructor con los atributos obligatorios para definir un usuario (en nuestro caso el correo electrónico), los *getters* y *setters* de todas las propiedades (necesario para JPA) y los métodos `equals` y `hashCode` para comparar usuarios.

Los métodos `equals` y `hashCode` son necesarios para buscar instancias de la entidad en colecciones y JPA los usa para no incluir instancias repetidas en los resultados de las queries. El método `equals` proporcionado no es el que genera IntelliJ por defecto, sino que hay que considerar si la instancia ha sido ya vinculada a la base de datos o no. En el caso en que la instancia ya esté vinculada a la base de datos, tendrá una clave primaria asignada y ésta será la que se usará para comparar. En el caso en que la instancia no esté vinculada (se acaba de crear o la estamos usando para alguna parte de la lógica de negocio y no se va a persistir) se comparan los atributos obligatorios (en este caso el correo electrónico).

Fichero `src/main/java/madstodolist/model/Usuario.java`:

```
package madstodolist.model;

import javax.persistence.*;
import javax.validation.constraints.NotNull;
import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
```

```
import java.util.Objects;
import java.util.Set;

@Entity
@Table(name = "usuarios")
public class Usuario implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @NotNull
    private String email;
    private String nombre;
    private String password;
    @Column(name = "fecha_nacimiento")
    @Temporal(TemporalType.DATE)
    private Date fechaNacimiento;

    // Definimos el tipo de fetch como EAGER para que
    // cualquier consulta que devuelve un usuario rellene automáticamente
    // toda su lista de tareas
    // CUIDADO!! No es recomendable hacerlo en aquellos casos en los
    // que la relación pueda traer a memoria una gran cantidad de entidades
    @OneToMany(mappedBy = "usuario", fetch = FetchType.EAGER)
    Set<Tarea> tareas = new HashSet<>();

    // Constructor vacío necesario para JPA/Hibernate.
    // No debe usarse desde la aplicación.
    public Usuario() {}

    // Constructor público con los atributos obligatorios. En este caso el
    correo electrónico.
    public Usuario(String email) {
        this.email = email;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getNombre() {
        return nombre;
    }
}
```

```

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

public Date getFechaNacimiento() {
    return fechaNacimiento;
}

public void setFechaNacimiento(Date fechaNacimiento) {
    this.fechaNacimiento = fechaNacimiento;
}

public Set<Tarea> getTareas() {
    return tareas;
}

public void setTareas(Set<Tarea> tareas) {
    this.tareas = tareas;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Usuario usuario = (Usuario) o;
    if (id != null && usuario.id != null)
        // Si tenemos los ID, comparamos por ID
        return Objects.equals(id, usuario.id);
    // sino comparamos por campos obligatorios
    return email.equals(usuario.email);
}

@Override
public int hashCode() {
    // Generamos un hash basado en los campos obligatorios
    return Objects.hash(email);
}
}

```

En la definición de la entidad también se incluyen relaciones con otras entidades. En este caso un `Usuario` tiene muchas `Tarea`s (una relación uno-a-muchos).

La relación uno-a-muchos se representa en la base de datos con una clave ajena. El atributo `mappedBy` indica que la clave ajena se va a guardar en la columna correspondiente con el atributo `usuario` de la entidad `Tarea`.

La definición de `Tarea` es la siguiente:

Fichero `src/main/java/madstodolist/model/Tarea.java`:

```
package madstodolist.model;

import javax.persistence.*;
import javax.validation.constraints.NotNull;
import java.io.Serializable;
import java.util.Objects;

@Entity
@Table(name = "tareas")
public class Tarea implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @NotNull
    private String titulo;

    @NotNull
    // Relación muchos-a-uno entre tareas y usuario
    @ManyToOne
    // Nombre de la columna en la BD que guarda físicamente
    // el ID del usuario con el que está asociado una tarea
    @JoinColumn(name = "usuario_id")
    private Usuario usuario;

    // Constructor vacío necesario para JPA/Hibernate.
    // No debe usarse desde la aplicación.
    public Tarea() {}

    // Al crear una tarea la asociamos automáticamente a un
    // usuario. Actualizamos por tanto la lista de tareas del
    // usuario.
    public Tarea(Usuario usuario, String titulo) {
        this.usuario = usuario;
        this.titulo = titulo;
        usuario.getTareas().add(this);
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getTitulo() {
        return titulo;
    }
}
```

```

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public Usuario getUsuario() {
        return usuario;
    }

    public void setUsuario(Usuario usuario) {
        this.usuario = usuario;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Tarea tarea = (Tarea) o;
        if (id != null && tarea.id != null)
            // Si tenemos los ID, comparamos por ID
            return Objects.equals(id, tarea.id);
        // sino comparamos por campos obligatorios
        return titulo.equals(tarea.titulo) &&
            usuario.equals(tarea.usuario);
    }

    @Override
    public int hashCode() {
        return Objects.hash(titulo, usuario);
    }
}

```

Recuperación *eager* y *lazy* de las colecciones

Como hemos visto anteriormente, en la aplicación se define la relación *uno-a-muchos* entre usuarios y tareas: un usuario tiene muchas tareas.

Por defecto, todas las relaciones *a-muchos* en JPA se definen de tipo `LAZY`.

La característica de los atributos marcados como *lazy* en JPA es que no se traen a memoria cuando se recupera la entidad, sino cuando se consultan explícitamente accediendo al atributo. Para que se traigan a memoria **la conexión con la base de datos debe estar abierta**. Una forma de hacerlo, la que usamos en la aplicación, es marcando el método que accede a las entidades con la anotación `@Transactional`. Las entidades que usemos en los métodos con esa anotación estarán conectadas a la base de datos y podremos recuperar sus relaciones *lazy*. Una vez terminada la transacción, por ejemplo fuera del método anotado con `@Transactional`, si intentamos acceder a una relación *lazy* sin haberla inicializado se producirá un error.

En la práctica 3 definiremos una relación *lazy* y explicaremos cómo gestionarla en la capa de servicios. En esta práctica, sin embargo, usaremos relaciones de tipo *EAGER*.

En el caso de definir un relación entre entidades de tipo *EAGER*, JPA traerá siempre a memoria todos los elementos cuando se recupere cualquier entidad. Hemos definido de esta forma la relación entre un usuario y sus tareas. Por ejemplo, cuando se realice una búsqueda y se recupere un usuario, se recuperarán también automáticamente todas sus tareas y se inicializará en memoria la colección de tareas del usuario.

En general, no es conveniente definir una relación como *eager* porque puede provocar problemas de rendimiento en el caso en que haya muchos elementos relacionados. Pero si no hay muchos datos en la relación y los vamos a usar con frecuencia, sí que es aconsejable usar el tipo *EAGER* para facilitar el manejo de la entidad.

El código queda como hemos visto anteriormente:

```
@Entity
public class Usuario {
    ...
    // Definimos el tipo de fetch como EAGER para que
    // cualquier consulta que devuelve un usuario rellene automáticamente
    // toda su lista de tareas
    // CUIDADO!! No es recomendable hacerlo en aquellos casos en los
    // que la relación pueda traer a memoria una gran cantidad de entidades
    @OneToMany(mappedBy = "usuario", fetch = FetchType.EAGER)
    List<Tarea> tareas = new ArrayList<Tarea>();
    ...
}

@Entity
public class Tarea {
    ...
    // Relación muchos-a-uno entre tareas y usuario
    @ManyToOne
    // Nombre de la columna en la BD que guarda físicamente
    // el ID del usuario con el que está asociado una tarea
    @JoinColumn(name = "usuario_id")
    private Usuario usuario;
    ...
}
```

Note

La diferencia entre recuperación *lazy* y recuperación *eager* de las relaciones es uno de los conceptos que causan más problemas al trabajar con Hibernate. Te recomiendo que lo estudies bien. Un ejercicio que te recomiendo es cambiar el atributo anterior a *LAZY* y comprobar qué errores se producen en el código y en los tests.

Clases Repository

Spring define la clase genérica `CrudRepository` que contienen métodos por defecto para actualizar las entidades y realizar *queries* sobre ellas. Para dejar abierta la posibilidad de

cambiar la implementación, se definen con interfaces.

```
public interface CrudRepository<T, ID extends Serializable> extends
Repository<T, ID> {
    <S extends T> S save(S entity);
    Optional<T> findById(ID primaryKey);
    Iterable<T> findAll();
    long count();
    void delete(T entity);
    boolean existsById(ID primaryKey);
    // ... more functionality omitted.
}
```

Para usar estos métodos con nuestras entidades basta con definir interfaces que extienden esta clase genérica. Por ejemplo, la interfaz `TareaRepository`:

Fichero `src/main/java/madstodolist/model/TareaRepository.java`:

```
package madstodolist.model;

import org.springframework.data.repository.CrudRepository;

public interface TareaRepository extends CrudRepository<Tarea, Long> {}
```

Una vez definida la interfaz, ya podemos inyectar una instancia de `repository` y usarla en las clases de servicio. Por ejemplo, mostramos el método de servicio que modifica el título de una tarea:

```
@Service
public class TareaService {

    @Autowired
    private UsuarioRepository usuarioRepository;
    @Autowired
    private TareaRepository tareaRepository;

    ...

    @Transactional
    public Tarea modificaTarea(Long idTarea, String nuevoTitulo) {
        Tarea tarea = tareaRepository.findById(idTarea).orElse(null);
        if (tarea == null) {
            throw new TareaServiceException("No existe tarea con id " +
idTarea);
        }
        tarea.setTitulo(nuevoTitulo);
        tareaRepository.save(tarea);
        return tarea;
    }

    ...
}
```

En el cuerpo del método se llama al método `findById` del repositorio que realiza una búsqueda en la base de datos y al método `save` que actualiza el valor de la entidad.

La anotación `@Transactional` hace dos cosas. En primer lugar, abre una conexión con la base de datos y hace que todas las llamadas a las clases repository se realicen usando esa conexión. Las entidades están conectadas a la base de datos durante todas las sentencias que se ejecutan dentro del método anotado y cualquier cambio en ellas se propaga a la base de datos (en este caso el título de la tarea). Además, y muy importante, las relaciones *lazy* pueden recuperarse sin problemas accediendo a los atributos correspondientes de las entidades.

En segundo lugar, la anotación `@Transactional`, como su nombre indica, hace que las acciones sobre la base de datos se ejecuten de forma transaccional. Se abre la transacción al del método y se cierra al final. Si sucede alguna excepción durante su ejecución la transacción se deshace.

La interfaz `UsuarioRepository` es similar.

Fichero `src/main/java/madstodolist/model/UsuarioRepository.java`:

```
package madstodolist.model;

import org.springframework.data.repository.CrudRepository;

import java.util.Optional;

public interface UsuarioRepository extends CrudRepository<Usuario, Long> {
    Optional<Usuario> findByEmail(String s);
}
```

La diferencia es que se añade un método `findByEmail` que hace que Spring construya automáticamente una consulta sobre la base de datos. Al usar como nombre del método el nombre de la propiedad de la entidad (`email`), Spring puede generar automáticamente la consulta.

Puedes consultar una lista completa de las traducciones de nombres de métodos a consultas a la base de datos en [este enlace](#) de la documentación de Spring Boot.

También es posible definir explícitamente en el Repository la consulta a realizar a la base de datos utilizando la anotación `@Query`. Puedes encontrar varios ejemplos en [este enlace](#).

Servicios

La capa de servicios es la capa intermedia entre la capa de *controllers* y la de *repository*. Es la capa que implementa toda la lógica de negocio de la aplicación.

La responsabilidad principal de la capa de servicios es crear, obtener o modificar los objetos entidad necesarios para cada funcionalidad a partir de los datos que manda la capa *controller*, trabajar con ellos en memoria y hacer persistentes los cambios utilizando la capa *repository*.

La capa de servicios también gestionará errores y lanzará excepciones cuando no se pueda realizar alguna funcionalidad.

Los servicios obtienen instancias de *Repository* usando inyección de dependencias.

Como hemos comentado anteriormente, los métodos de la capa de servicios estarán anotados con `@Transactional` para garantizar la transaccionalidad.

Por ejemplo, la clase `UsuarioService` se define como se muestra a continuación.

Fichero `src/main/java/madstodolist/service/UsuarioService.java`:

```
package madstodolist.service;

import madstodolist.model.Usuario;
import madstodolist.model.UsuarioRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.Optional;

@Service
public class UsuarioService {

    public enum LoginStatus {LOGIN_OK, USER_NOT_FOUND, ERROR_PASSWORD}

    @Autowired
    private UsuarioRepository usuarioRepository;

    @Transactional(readOnly = true)
    public LoginStatus login(String eMail, String password) {
        Optional<Usuario> usuario = usuarioRepository.findByEmail(eMail);
        if (!usuario.isPresent()) {
            return LoginStatus.USER_NOT_FOUND;
        } else if (!usuario.get().getPassword().equals(password)) {
            return LoginStatus.ERROR_PASSWORD;
        } else {
            return LoginStatus.LOGIN_OK;
        }
    }

    // Se añade un usuario en la aplicación.
    // El email y password del usuario deben ser distinto de null
    // El email no debe estar registrado en la base de datos
    @Transactional
    public Usuario registrar(Usuario usuario) {
```

```

        Optional<Usuario> usuarioBD =
usuarioRepository.findByEmail(usuario.getEmail());
    if (usuarioBD.isPresent())
        throw new UsuarioServiceException("El usuario " +
usuario.getEmail() + " ya está registrado");
    else if (usuario.getEmail() == null)
        throw new UsuarioServiceException("El usuario no tiene email");
    else if (usuario.getPassword() == null)
        throw new UsuarioServiceException("El usuario no tiene password");
    else return usuarioRepository.save(usuario);
}

@Transactional(readOnly = true)
public Usuario findByEmail(String email) {
    return usuarioRepository.findByEmail(email).orElse(null);
}

@Transactional(readOnly = true)
public Usuario findById(Long usuarioId) {
    return usuarioRepository.findById(usuarioId).orElse(null);
}
}

```

Fichero src/main/java/madstodolist/service/UsuarioServiceException.java :

```

package madstodolist.service;

public class UsuarioServiceException extends RuntimeException {

    public UsuarioServiceException(String message) {
        super(message);
    }
}

```

Ventajas de utilizar una capa de servicios

Al utilizar clases de servicios podemos aislar la lógica de negocio de la aplicación usando métodos y objetos Java, sin preocuparnos de cómo obtener los datos de la interfaz de usuario ni de cómo mostrar los resultados. De esto ya se ocuparán las clases *controller*. De esta forma, si se necesita modificar la interfaz de usuario de la aplicación, o convertirla en un servicio REST que devuelva JSON en lugar de HTML sólo tendremos que tocar las clases *controller*, no las de servicio.

Además, al no tener ninguna dependencia con la interfaz de usuario, estas clases de servicios también podrán ser fácilmente testeadas. La mayoría de los tests automáticos los haremos sobre ellas.

Controllers

Las clases *controllers* son las que gestionan la interfaz de usuario de la aplicación. Se encargan de recibir los datos de las peticiones HTTP, llamar a la clase de servicio necesaria

para procesar la lógica de negocio y mostrar la vista con la información resultante proporcionada por la clase de servicio.

En la aplicación se definen dos clases controller:

- `LoginController`: con métodos para registrar y logear usuarios.
- `TareasController`: con métodos para crear, borrar y modificar tareas de un usuario.

Los controllers usan clases auxiliares en las que se guardan los datos introducidos en los formularios. Por ejemplo, la clase `LoginController` usa las clases `LoginData` y `RegistroData`.

Fichero `src/main/java/madstodolist/controller/LoginController.java`:

```
package madstodolist.controller;

import madstodolist.authentication.ManagerUserSession;
import madstodolist.model.Usuario;
import madstodolist.service.TareaService;
import madstodolist.service.UsuarioService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;

import javax.servlet.http.HttpSession;
import javax.validation.Valid;

@Controller
public class LoginController {

    @Autowired
    UsuarioService usuarioService;

    @Autowired
    ManagerUserSession managerUserSession;

    @GetMapping("/")
    public String home(Model model) {
        return "redirect:/login";
    }

    @GetMapping("/login")
    public String loginForm(Model model) {
        model.addAttribute("loginData", new LoginData());
        return "formLogin";
    }

    @PostMapping("/login")
    public String loginSubmit(@ModelAttribute LoginData loginData, Model
model, HttpSession session) {
```

```

        // Llamada al servicio para comprobar si el login es correcto
        UsuarioService.LoginStatus loginStatus =
    usuarioService.login(loginData.geteMail(), loginData.getPassword());

        if (loginStatus == UsuarioService.LoginStatus.LOGIN_OK) {
            Usuario usuario =
    usuarioService.findByEmail(loginData.geteMail());

            managerUserSession.logearUsuario(usuario.getId());

            return "redirect:/usuarios/" + usuario.getId() + "/tareas";
        } else if (loginStatus == UsuarioService.LoginStatus.USER_NOT_FOUND) {
            model.addAttribute("error", "No existe usuario");
            return "formLogin";
        } else if (loginStatus == UsuarioService.LoginStatus.ERROR_PASSWORD) {
            model.addAttribute("error", "Contraseña incorrecta");
            return "formLogin";
        }
        return "formLogin";
    }

    @GetMapping("/registro")
    public String registroForm(Model model) {
        model.addAttribute("registroData", new RegistroData());
        return "formRegistro";
    }

    @PostMapping("/registro")
    public String registroSubmit(@Valid RegistroData registroData,
BindingResult result, Model model) {

        if (result.hasErrors()) {
            return "formRegistro";
        }

        if (usuarioService.findByEmail(registroData.geteMail()) != null) {
            model.addAttribute("registroData", registroData);
            model.addAttribute("error", "El usuario " +
registroData.geteMail() + " ya existe");
            return "formRegistro";
        }

        Usuario usuario = new Usuario(registroData.geteMail());
        usuario.setPassword(registroData.getPassword());
        usuario.setFechaNacimiento(registroData.getFechaNacimiento());
        usuario.setNombre(registroData.getNombre());

        usuarioService.registrar(usuario);
        return "redirect:/login";
    }

    @GetMapping("/logout")
    public String logout(HttpSession session) {
        managerUserSession.logout();
        return "redirect:/login";
    }

```

```

    }
}
```

Fichero src/main/java/madstodolist/controller/LoginData.java:

```

package madstodolist.controller;

public class LoginData {
    private String eMail;
    private String password;

    public String geteMail() {
        return eMail;
    }

    public void seteMail(String eMail) {
        this.eMail = eMail;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

Peticiones y rutas

Las rutas (*endpoints*) que se definen en los controllers para realizar las acciones de la aplicación son:

LoginController

- GET /login: devuelve el formulario de login
- POST /login: realiza el login
- GET /registro: devuelve el formulario de registro
- POST /registro: realiza el registro
- GET /logout: realiza la salida del usuario de la aplicación

TareaController

- GET /usuarios/{id}/tareas/nueva: devuelve el formulario para añadir una tarea al usuario con identificador {id}
- POST /usuarios/{id}/tareas/nueva: añade una tarea nueva a un usuario
- GET /usuarios/{id}/tareas: devuelve el listado de tareas de un usuario
- GET /tareas/{id}/editar": devuelve el formulario para editar una tarea

- `POST /tareas/{id}/editar`: añade una tarea modificada
- `DELETE /tareas/{id}`: realiza el borrado de una tarea

Vistas

Todas las vistas de la aplicación comparten la misma cabecera y pie de página. Para centralizar estos elementos se usa la característica de fragmentos de Thymeleaf. Los fragmentos comunes se definen en el fichero `fragments.html`.

Fichero `src/main/resources/templates/fragments.html`:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<head th:fragment="head (titulo)">
    <meta charset="UTF-8"/>
    <title th:text="${titulo}"></title>
    <link rel="stylesheet" th:href="@{/css/bootstrap.min.css}"/>
</head>

<div th:fragment="javascript">
    <script th:src="@{/js/jquery.min.js}"></script>
    <script th:src="@{/js/popper.min.js}"></script>
    <script th:src="@{/js/bootstrap.min.js}"></script>
</div>
</html>
```

Vemos que las vistas usan el framework CSS *Bootstrap* (en concreto, la versión [Bootstrap 4.6](#)) y varias librerías JavaScript. Ambos se encuentran en el directorio `src/main/resources/static/`, el directorio por defecto en el que se guardan los recursos estáticos de una aplicación Spring Boot.

La vista principal de la aplicación es el listado de tareas que vemos a continuación.

Fichero `src/main/resources/templates/listaTareas.html`:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<head th:replace="fragments :: head (titulo='Login')"/></head>

<body>
<div class="container-fluid">

    <div class="row mt-3">
        <div class="col">
            <h2 th:text="'Listado de tareas de ' + ${usuario.nombre}"></h2>
        </div>
    </div>

    <div class="row mt-3">
```

```

<div class="col">
    <table class="table table-striped">
        <thead>
            <tr>
                <th>Id</th>
                <th>Tarea</th>
                <th>Acción</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="tarea: ${tareas}">
                <td th:text="${tarea.id}"></td>
                <td th:text="${tarea.titulo}"></td>
                <td><a class="btn btn-primary btn-xs"
th:href="@{/tareas/{id}/editar(id=${tarea.id})}">editar</a>
                    <a class="btn btn-danger btn-xs" href="#" onmouseover="" style="cursor: pointer;" th:onclick="'del('/tareas/' + ${tarea.id} + '\')'">borrar</a>
                </td>
            </tr>
        </tbody>
    </table>
    <p><a class="btn btn-primary"
th:href="@{/usuarios/{id}/tareas/nueva(id=${usuario.id})}"> Nueva tarea</a>
        <a class="btn btn-link" href="/logout">Salir</a></p>
    </div>
</div>
<div class="row mt-2">
    <div class="col">
        <div class="alert alert-success alert-dismissible fade show" role="alert" th:if="${!#strings.isEmpty(mensaje)}">
            <span th:text="${mensaje}"></span>
            <button type="button" class="close" data-dismiss="alert" aria-label="Close">
                <span aria-hidden="true">&times;</span>
            </button>
        </div>
    </div>
</div>
</div>

</div>

<div th:replace="fragments::javascript"/>

<!-- Ejemplo de uso de Ajax para lanzar una petición DELETE y borrar una tarea
-->

<script type="text/javascript">
    function del(urlBorrar) {
        if (confirm('¿Estás seguro/a de que quieres borrar la tarea?')) {
            $.ajax({
                url: urlBorrar,
                type: 'DELETE',

```

```

        success: function (results) {
            //refresh the page
            location.reload();
        }
    });
}
</script>

</body>
</html>

```

- La plantilla recibe una lista de tareas, un usuario y un mensaje (consultar en el controller `TareasController` cómo se obtienen esos datos).
- Define un script JavaScript en el que se realiza una petición `DELETE` a la URL que se le pasa como parámetro (se utilizará para lanzar la acción de borrar una tarea).
- Utiliza una instrucción de iteración sobre la lista de tareas para construir los elementos de la tabla de tareas.
- En las acciones de añadir y editar tareas se construyen las URLs a las que hacer la petición usando el identificador de la tarea.

Autenticación y control de acceso

En la aplicación se realiza una autenticación y un control de acceso muy sencillo usando la sesión HTTP (clase `HttpSession`). Esta sesión se implementa en Spring Boot con una cookie que se pasa desde el navegador hasta el servidor en cada petición.

El manejo de la clase `HttpSession` es muy sencillo: es un diccionario en el que podemos añadir datos. En el servidor podemos obtener los datos de la sesión consultando el diccionario.

La implementación de la autenticación y del control de acceso se realiza con en la clase `ManagerUserSesion`:

Fichero `src/main/java/madstodolist/authentication/ManagerUserSesion.java`:

```

package madstodolist.authentication;

import madstodolist.controller.exception.UsuarioNoLogeadoException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import javax.servlet.http.HttpSession;

@Component
public class ManagerUserSession {

    @Autowired
    HttpSession session;
}

```

```
// Añadimos el id de usuario en la sesión HTTP para hacer
// una autorización sencilla. En los métodos de controllers
// comprobamos si el id del usuario logeado coincide con el obtenido
// desde la URL
public void logearUsuario(Long idUsuario) {
    session.setAttribute("idUsuarioLogeado", idUsuario);
}

public Long usuarioLogeado() {
    return (Long) session.getAttribute("idUsuarioLogeado");
}

public void logout() {
    session.setAttribute("idUsuarioLogeado", null);
}
}
```

Se implementa como un componente Spring con la anotación `@Component`. La referencia a la clase `HttpSession` se obtiene por inyección de dependencias con la anotación de spring Boot `@Autowired`.

La anotación `@Component` permite injectar un `ManagerUserSession` en los controllers para gestionar allí el usuario que está logeado y mockearlo en los tests.

Pruebas manuales y automáticas

Durante el desarrollo de la práctica será necesario realizar **pruebas manuales** de la aplicación, introducir datos en sus pantallas y comprobar que los cambios que vamos introduciendo funcionan correctamente.

Para estas pruebas manuales recomendamos utilizar la configuración de ejecución trabajando sobre una base de datos con valores iniciales. Estos valores iniciales se cargan en la aplicación al comenzar.

Fichero `src/main/resources/data.sql`:

```
/* Populate tables */
INSERT INTO usuarios (id, email, nombre, password, fecha_nacimiento)
VALUES('1', 'user@ua', 'Usuario Ejemplo', '123', '2001-02-10');
INSERT INTO tareas (id, titulo, usuario_id) VALUES('1', 'Lavar coche', '1');
INSERT INTO tareas (id, titulo, usuario_id) VALUES('2', 'Renovar DNI', '1');
```

En los tests automáticos se cargan los datos de prueba al comienzo de cada test y, usando la anotación `@Sql`, se limpian las tablas con el script `clean-db.sql`.

```
@Sql/scripts = "/clean-db.sql", executionPhase = AFTER_TEST_METHOD)
public class TareaTest {
    ...
}
```

Fichero `src/test/resources/clean-db.sql`:

```
DELETE FROM tareas;
DELETE FROM usuarios;
```

Tests de las entidades y de la capa repository

Se realizan tests automáticos sobre las entidades y repository:

- `TareaTest.java`
- `UsuarioTest.java`:

Veamos, por ejemplo, el fichero `TareaTest.java`:

Fichero `src/test/java/madstodolist/TareaTest.java`:

```
package madstodolist;

import madstodolist.model.Tarea;
import madstodolist.model.TareaRepository;
import madstodolist.model.Usuario;
import madstodolist.model.UsuarioRepository;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.jdbc.Sql;
import org.springframework.transaction.annotation.Transactional;

import java.util.Set;

import static org.assertj.core.api.Assertions.assertThat;
import static org.springframework.test.context.jdbc.Sql.ExecutionPhase.AFTER_TEST_METHOD;

@SpringBootTest
@Sql(scripts = "/clean-db.sql", executionPhase = AFTER_TEST_METHOD)
public class TareaTest {

    @Autowired
    UsuarioRepository usuarioRepository;

    @Autowired
    TareaRepository tareaRepository;

    //
    // Tests modelo Tarea en memoria, sin la conexión con la BD
    //

    @Test
    public void crearTarea() {
        // GIVEN
        // Un usuario nuevo creado en memoria, sin conexión con la BD,
        Usuario usuario = new Usuario("juan.gutierrez@gmail.com");
    }
}
```

```
// WHEN
// se crea una nueva tarea con ese usuario,
Tarea tarea = new Tarea(usuario, "Práctica 1 de MADS");

// THEN
// el título y el usuario de la tarea son los correctos.

assertThat(tarea.getTitulo()).isEqualTo("Práctica 1 de MADS");
assertThat(tarea.getUsuario()).isEqualTo(usuario);
}

@Test
public void laListaDeTareasDeUnUsuarioSeActualizaEnMemoriaConUnaNuevaTarea() {
    // GIVEN
    // Un usuario nuevo creado en memoria, sin conexión con la BD,
    Usuario usuario = new Usuario("juan.gutierrez@gmail.com");

    // WHEN
    // se crea una tarea de ese usuario,

    Set<Tarea> tareas = usuario.getTareas();
    Tarea tarea = new Tarea(usuario, "Práctica 1 de MADS");

    // THEN
    // la tarea creada se ha añadido a la lista de tareas del usuario.

    assertThat(usuario.getTareas()).contains(tarea);
    assertThat(tareas).contains(tarea);
}

@Test
public void comprobarIgualdadTareasSinId() {
    // GIVEN
    // Creadas tres tareas sin identificador, y dos de ellas con
    // la misma descripción

    Usuario usuario = new Usuario("juan.gutierrez@gmail.com");
    Tarea tarea1 = new Tarea(usuario, "Práctica 1 de MADS");
    Tarea tarea2 = new Tarea(usuario, "Práctica 1 de MADS");
    Tarea tarea3 = new Tarea(usuario, "Pagar el alquiler");

    // THEN
    // son iguales (Equal) las tareas que tienen la misma descripción.

    assertThat(tarea1).isEqualTo(tarea2);
    assertThat(tarea1).isNotEqualTo(tarea3);
}

@Test
public void comprobarIgualdadTareasConId() {
    // GIVEN
    // Creadas tres tareas con distintas descripciones y dos de ellas
    // con el mismo identificador,
```

```
Usuario usuario = new Usuario("juan.gutierrez@gmail.com");
Tarea tarea1 = new Tarea(usuario, "Práctica 1 de MADS");
Tarea tarea2 = new Tarea(usuario, "Lavar la ropa");
Tarea tarea3 = new Tarea(usuario, "Pagar el alquiler");
tarea1.setId(1L);
tarea2.setId(2L);
tarea3.setId(1L);

// THEN
// son iguales (Equal) las tareas que tienen el mismo identificador.

assertThat(tarea1).isEqualTo(tarea3);
assertThat(tarea1).isNotEqualTo(tarea2);
}

// Tests TareaRepository.
// El código que trabaja con repositorios debe
// estar en un entorno transactional, para que todas las peticiones
// estén en la misma conexión a la base de datos, las entidades estén
// conectadas y sea posible acceder a colecciones LAZY.
//

@Test
@Transactional
public void guardarTareaEnBaseDatos() {
    // GIVEN
    // Un usuario en la base de datos.

    Usuario usuario = new Usuario("user@ua");
    usuarioRepository.save(usuario);

    Tarea tarea = new Tarea(usuario, "Práctica 1 de MADS");

    // WHEN
    // salvamos la tarea en la BD,
    tareaRepository.save(tarea);

    // THEN
    // se actualiza el id de la tarea,

    assertThat(tarea.getId()).isNotNull();

    // y con ese identificador se recupera de la base de datos la tarea
    // con los valores correctos de las propiedades y la relación con
    // el usuario actualizado también correctamente (la relación entre
    tarea
    // y usuario es EAGER).

    Tarea tareaBD = tareaRepository.findById(tarea.getId()).orElse(null);
    assertThat(tareaBD.getTitulo()).isEqualTo(tarea.getTitulo());
    assertThat(tareaBD.getUsuario()).isEqualTo(usuario);
}

@Test
```

```
@Transactional
public void salvarTareaEnBaseDatosConUsuarioNoBDLanzaExpcion() {
    // GIVEN
    // Un usuario nuevo que no está en la BD
    // y una tarea asociada a ese usuario,
    Usuario usuario = new Usuario("juan.gutierrez@gmail.com");
    Tarea tarea = new Tarea(usuario, "Práctica 1 de MADS");

    // WHEN // THEN
    // se lanza una excepción al intentar salvar la tarea en la BD

    Assertions.assertThrows(Exception.class, () -> {
        tareaRepository.save(tarea);
    });
}

@Test
@Transactional
public void unUsuarioTieneUnaListaDeTareas() {
    // GIVEN
    // Un usuario con 2 tareas en la base de datos
    Usuario usuario = new Usuario("user@ua");
    usuarioRepository.save(usuario);
    Long usuarioId = usuario.getId();

    Tarea tarea1 = new Tarea(usuario, "Práctica 1 de MADS");
    Tarea tarea2 = new Tarea(usuario, "Renovar el DNI");
    tareaRepository.save(tarea1);
    tareaRepository.save(tarea2);

    // WHEN
    // recuperamos el ususario de la base de datos,
    Usuario usuarioRecuperado =
    usuarioRepository.findById(usuarioId).orElse(null);

    // THEN
    // su lista de tareas también se recupera, porque se ha
    // definido la relación de usuario y tareas como EAGER.

    assertThat(usuarioRecuperado.getTareas()).hasSize(2);
}

@Test
@Transactional
public void añadirUnaTareaAUnUsuarioEnBD() {
    // GIVEN
    // Un usuario en la base de datos
    Usuario usuario = new Usuario("user@ua");
    usuarioRepository.save(usuario);
    Long usuarioId = usuario.getId();

    // WHEN
    // Creamos una nueva tarea con el usuario recuperado de la BD
    // y la salvamos,
```

```

        Usuario usuarioBD =
usuarioRepository.findById(usuarioId).orElse(null);
Tarea tarea = new Tarea(usuarioBD, "Práctica 1 de MADS");
tareaRepository.save(tarea);
Long tareaId = tarea.getId();

// THEN
// la tarea queda guardada en la BD asociada al usuario

Tarea tareaBD = tareaRepository.findById(tareaId).orElse(null);
assertThat(tareaBD).isEqualTo(tarea);
assertThat(tarea.getUsuario()).isEqualTo(usuarioBD);

// y si recuperamos el usuario se obtiene la nueva tarea
usuarioBD = usuarioRepository.findById(usuarioId).orElse(null);
assertThat(usuarioBD.getTareas()).contains(tareaBD);
}

@Test
@Transactional
public void cambioEnLaEntidadEnTransactionalModificaLaBD() {
    // GIVEN
    // Un usuario y una tarea en la base de datos
    Usuario usuario = new Usuario("user@ua");
    usuarioRepository.save(usuario);
    Tarea tarea = new Tarea(usuario, "Práctica 1 de MADS");
    tareaRepository.save(tarea);

    // Recuperamos la tarea
    Long tareaId = tarea.getId();
    tarea = tareaRepository.findById(tareaId).orElse(null);

    // WHEN
    // modificamos la descripción de la tarea

    tarea.setTitulo("Esto es una prueba");

    // THEN
    // la descripción queda actualizada en la BD.

    Tarea tareaBD = tareaRepository.findById(tareaId).orElse(null);
    assertThat(tareaBD.getTitulo()).isEqualTo(tarea.getTitulo());
}
}

```

Te recomiendo que leas con cuidado los tests y sus comentarios. Son muy útiles para entender el funcionamiento de la aplicación (en este caso de las entidades y de la capa *repository*).

Utilizamos el formato *GIVEN*, *WHEN*, *THEN* para estructurar el test. En la parte *GIVEN* se preparan los datos, en la parte *WHEN* se lanza el método o métodos que se quieren probar y en la parte *THEN* se comprueban los resultados.

Se realizan distintos tipos de tests dentro de la misma clase:

- Pruebas sobre las entidades por si solas, sin conexión con la base de datos. Son lo que se denomina tests del modelo.
- Pruebas sobre la capa *repository*, en las que se comprueban que las operaciones de búsqueda y actualización funcionan correctamente sobre la base de datos. En muchas de estas pruebas se necesita realizar más de una sentencia con la misma conexión a la base de datos o acceder a atributos *lazy*. Para esto es necesario usar la anotación `@Transactional`.

Tests de la capa de servicios

También se realizan tests sobre la capa de servicio:

- `TareaServiceTest.java`
- `UsuarioServiceTest.java`

Estos tests comprueban que los métodos de servicio funcionan correctamente y modifican la base de datos tal y como se pretende en cada operación.

Veamos, por ejemplo, el fichero `TareaServiceTest.java`:

Fichero `src/test/java/madstodolist/TareaServiceTest.java`:

```
package madstodolist;

import madstodolist.model.Usuario;
import madstodolist.service.UsuarioService;
import madstodolist.service.UsuarioServiceException;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.jdbc.Sql;

import static org.assertj.core.api.Assertions.assertThat;
import static org.springframework.test.context.jdbc.Sql.ExecutionPhase.AFTER_TEST_METHOD;

@SpringBootTest
@Sql(scripts = "/clean-db.sql", executionPhase = AFTER_TEST_METHOD)
public class UsuarioServiceTest {

    @Autowired
    private UsuarioService usuarioService;

    // Método para inicializar los datos de prueba en la BD
    // Devuelve el identificador del usuario de la BD
    Long addUsuarioBD() {
        Usuario usuario = new Usuario("user@ua");
        usuario.setNombre("Usuario Ejemplo");
        usuario.setPassword("123");
        usuario = usuarioService.registrar(usuario);
        return usuario.getId();
    }
}
```

```
}

@Test
public void servicioLoginUsuario() {
    // GIVEN
    // Un usuario en la BD

    addUsuarioBD();

    // WHEN
    // intentamos logear un usuario y contraseña correctos
    UsuarioService.LoginStatus loginStatus1 =
    usuarioService.login("user@ua", "123");

    // intentamos logear un usuario correcto, con una contraseña
    // incorrecta
    UsuarioService.LoginStatus loginStatus2 =
    usuarioService.login("user@ua", "000");

    // intentamos logear un usuario que no existe,
    UsuarioService.LoginStatus loginStatus3 =
    usuarioService.login("pepito.perez@gmail.com", "12345678");

    // THEN

    // el valor devuelto por el primer login es LOGIN_OK,
    assertEquals(loginStatus1, UsuarioService.LoginStatus.LOGIN_OK);

    // el valor devuelto por el segundo login es ERROR_PASSWORD,
    assertEquals(loginStatus2, UsuarioService.LoginStatus.ERROR_PASSWORD);

    // y el valor devuelto por el tercer login es USER_NOT_FOUND.

    assertEquals(loginStatus3, UsuarioService.LoginStatus.USER_NOT_FOUND);
}

@Test
public void servicioRegistroUsuario() {
    // GIVEN
    // Creado un usuario nuevo, con una contraseña

    Usuario usuario = new Usuario("usuario.prueba2@gmail.com");
    usuario.setPassword("12345678");

    // WHEN
    // registramos el usuario,

    usuarioService.registrar(usuario);

    // THEN
    // el usuario se añade correctamente al sistema.

    Usuario usuarioBaseDatos =
    usuarioService.findByEmail("usuario.prueba2@gmail.com");
    assertEquals(usuarioBaseDatos, usuario);
}
```

```
assertThat(usuarioBaseDatos.getPassword()).isEqualTo(usuario.getPassword());
}

@Test
public void servicioRegistroUsuarioExpcionConNullPassword() {
    // GIVEN
    // Un usuario creado sin contraseña,
    Usuario usuario = new Usuario("usuario.prueba@gmail.com");

    // WHEN, THEN
    // intentamos registrarlo, se produce una excepción de tipo
    UsuarioServiceException
    Assertions.assertThrows(UsuarioServiceException.class, () -> {
        usuarioService.registrar(usuario);
    });
}

@Test
public void servicioRegistroUsuarioExpcionConEmailRepetido() {
    // GIVEN
    // Un usuario en la BD
    addUsuarioBD();

    // WHEN
    // Creamos un usuario con un e-mail ya existente en la base de datos,
    Usuario usuario = new Usuario("user@ua");
    usuario.setPassword("12345678");

    // THEN
    // si lo registramos, se produce una excepción de tipo
    UsuarioServiceException
    Assertions.assertThrows(UsuarioServiceException.class, () -> {
        usuarioService.registrar(usuario);
    });
}

@Test
public void servicioRegistroUsuarioDevuelveUsuarioConId() {
    // GIVEN
    // Dado un usuario con contraseña nuevo y sin identificador,
    Usuario usuario = new Usuario("usuario.prueba@gmail.com");
    usuario.setPassword("12345678");

    // WHEN
    // lo registramos en el sistema,
    usuarioService.registrar(usuario);

    // THEN
    // se actualiza el identificador del usuario

    assertThat(usuario.getId()).isNotNull();
}
```

```

    // con el identificador que se ha guardado en la BD.

    Usuario usuarioBD = usuarioService.findById(usuario.getId());
    assertThat(usuarioBD).isEqualTo(usuario);
}

@Test
public void servicioConsultaUsuarioDevuelveUsuario() {
    // GIVEN
    // Un usuario en la BD

    Long usuarioId = addUsuarioBD();

    // WHEN
    // recuperamos un usuario usando su e-mail,

    Usuario usuario = usuarioService.findByEmail("user@ua");

    // THEN
    // el usuario obtenido es el correcto.

    assertThat(usuario.getId()).isEqualTo(usuarioId);
    assertThat(usuario.getEmail()).isEqualTo("user@ua");
    assertThat(usuario.getNombre()).isEqualTo("Usuario Ejemplo");
}
}

```

Para conseguir que los tests sean independientes y evitar que datos introducidos o modificados en un test afecten a otros tests, limpiamos las tablas de la base de datos al final de cada test usando la anotación `@Sql`:

```

@Sql/scripts = "/clean-db.sql", executionPhase = AFTER_TEST_METHOD)
public class TareaServiceTest {

```

Los datos de prueba se introducen al principio de cada test.

Tests de la capa controller

Por último, también realizamos tests sobre los controllers:

- `UsuarioWebTest.java`
- `TareaWebTest.java`

En estos tests se comprueba que el resultado de realizar un `GET` o un `POST` sobre los endpoints correspondientes devuelven un HTML que contiene alguna cadena que coincide con lo esperado.

Existen dos enfoques a la hora de definir estos tests.

- Podemos, al igual que hemos hecho en los tests de servicio, introducir los datos de prueba al comienzo de cada test.

- Podemos *mockear* los servicios para que devuelvan los datos que nos interesan.

Utilizamos ambos enfoques para que aprendas a trabajar con los dos. En la clase

`TareaWebTest` se utilizan los datos de prueba de la base de datos y en la clase `UsuarioWebTest` se mockean los servicios.

La utilización de *mocks* es muy útil también para poder testear los métodos que tienen un acceso restringido al usuario que hace la operación. Por ejemplo, la consulta o modificación de una tarea. Mockeamos el `managerUserSession` para simular que el usuario está logeado.

Mostramos a continuación los ficheros de test de controllers.

Fichero `src/test/java/madstodolist/TareaWebTest.java`:

```
package madstodolist;

import madstodolist.authentication.ManagerUserSession;
import madstodolist.model.Tarea;
import madstodolist.model.Usuario;
import madstodolist.service.TareaService;
import madstodolist.service.UsuarioService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.jdbc.Sql;
import org.springframework.test.web.servlet.MockMvc;

import static org.hamcrest.Matchers.*;
import static org.mockito.Mockito.when;
import static
org.springframework.test.context.jdbc.Sql.ExecutionPhase.AFTER_TEST_METHOD;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@SpringBootTest
@AutoConfigureMockMvc
@Sql(scripts = "/clean-db.sql", executionPhase = AFTER_TEST_METHOD)
public class TareaWebTest {

    @Autowired
    private MockMvc mockMvc;

    // Declaramos los servicios como Autowired
    @Autowired
    private TareaService tareaService;

    @Autowired
    private UsuarioService usuarioService;
```

```

// Moqueamos el managerUserSession para poder moquear el usuario logeado
@MockBean
private ManagerUserSession managerUserSession;

class DosIds {
    Long usuarioId;
    Long tareaId;
    public DosIds(Long usuarioId, Long tareaId) {
        this.usuarioId = usuarioId;
        this.tareaId = tareaId;
    }
}

// Método para inicializar los datos de prueba en la BD
// Devuelve una pareja de identificadores del usuario y la primera tarea
añadida
DosIds addUsuarioTareasBD() {
    Usuario usuario = new Usuario("user@ua");
    usuario.setPassword("123");
    usuario = usuarioService.registrar(usuario);
    Tarea tarea1 = tareaService.nuevaTareaUsuario(usuario.getId(), "Lavar
coche");
    tareaService.nuevaTareaUsuario(usuario.getId(), "Renovar DNI");
    return new DosIds(usuario.getId(), tarea1.getId());
}

@Test
public void listaTareas() throws Exception {
    // GIVEN
    // Un usuario con dos tareas en la BD
    Long usuarioId = addUsuarioTareasBD().usuarioId;

    // Moqueamos el método usuarioLogeado para que devuelva el usuario 1L,
    // el mismo que se está usando en la petición. De esta forma evitamos
    // que salte la excepción de que el usuario que está haciendo la
    // petición no está logeado.
    when(managerUserSession.usuarioLogeado()).thenReturn(usuarioId);

    // WHEN, THEN
    // se realiza la petición GET al listado de tareas del usuario,
    // el HTML devuelto contiene las descripciones de sus tareas.

    String url = "/usuarios/" + usuarioId.toString() + "/tareas";

    this.mockMvc.perform(get(url))
        .andExpect((content()).string(allOf(
            containsString("Lavar coche"),
            containsString("Renovar DNI")
        )));
}

@Test
public void getNuevaTareaDevuelveForm() throws Exception {
    // GIVEN
    // Un usuario con dos tareas en la BD
    Long usuarioId = addUsuarioTareasBD().usuarioId;
}

```

```

    // Ver el comentario en el primer test
    when(managerUserSession.usuarioLogeado()).thenReturn(usuarioId);

    // WHEN, THEN
    // si ejecutamos una petición GET para crear una nueva tarea de un
    usuario,
    // el HTML resultante contiene un formulario y la ruta con
    // la acción para crear la nueva tarea.

    String urlPeticion = "/usuarios/" + usuarioId.toString() +
    "/tareas/nueva";
    String urlAction = "action=\"/usuarios/" + usuarioId.toString() +
    "/tareas/nueva\"";

    this.mockMvc.perform(get(urlPeticion)
        .andExpect((content()).string(allOf(
            containsString("form method=\"post\""),
            containsString(urlAction)
        ))));
}

@Test
public void postNuevaTareaDevuelveRedirectYAñadeTarea() throws Exception {
    // GIVEN
    // Un usuario con dos tareas en la BD
    Long usuarioId = addUsuarioTareasBD().usuarioId;

    // Ver el comentario en el primer test
    when(managerUserSession.usuarioLogeado()).thenReturn(usuarioId);

    // WHEN, THEN
    // realizamos la petición POST para añadir una nueva tarea,
    // el estado HTTP que se devuelve es un REDIRECT al listado
    // de tareas.

    String urlPost = "/usuarios/" + usuarioId.toString() +
    "/tareas/nueva";
    String urlRedirect = "/usuarios/" + usuarioId.toString() + "/tareas";

    this.mockMvc.perform(post(urlPost)
        .param("titulo", "Estudiar examen MADS"))
        .andExpect(status().is3xxRedirection())
        .andExpect(redirectedUrl(urlRedirect));

    // y si después consultamos el listado de tareas con una petición
    // GET el HTML contiene la tarea añadida.

    this.mockMvc.perform(get(urlRedirect))
        .andExpect((content()).string(containsString("Estudiar examen
MADS"))));
}

@Test
public void deleteTareaDevuelveOKyBorraTarea() throws Exception {
    // GIVEN
    // Un usuario con dos tareas en la BD
    DosIds dosIds = addUsuarioTareasBD();
}

```

```

        Long usuarioId = dosIds.usuarioId;
        Long tareaLavarCocheId = dosIds.tareaId;

        // Ver el comentario en el primer test
        when(managerUserSession.usuarioLogeado()).thenReturn(usuarioId);

        // WHEN, THEN
        // realizamos la petición DELETE para borrar una tarea,
        // se devuelve el estado HTTP que se devuelve es OK,

        String urlDelete = "/tareas/" + tareaLavarCocheId.toString();

        this.mockMvc.perform(delete(urlDelete)
            .andExpect(status().isOk()));

        // y cuando se pide un listado de tareas del usuario, la tarea borrada
        ya no aparece.

        String urlListado = "/usuarios/" + usuarioId + "/tareas";

        this.mockMvc.perform(get(urlListado)
            .andExpect(content().string(
                allOf(not(containsString("Lavar coche")),
                      containsString("Renovar DNI"))));
    }

    @Test
    public void editarTareaActualizaLaTarea() throws Exception {
        // GIVEN
        // Un usuario con dos tareas en la BD
        DosIds dosIds = addUsuarioTareasBD();
        Long usuarioId = dosIds.usuarioId;
        Long tareaLavarCocheId = dosIds.tareaId;

        // Ver el comentario en el primer test
        when(managerUserSession.usuarioLogeado()).thenReturn(usuarioId);

        // WHEN, THEN
        // realizamos una petición POST al endpoint para editar una tarea

        String urlEditar = "/tareas/" + tareaLavarCocheId + "/editar";
        String urlRedirect = "/usuarios/" + usuarioId + "/tareas";

        this.mockMvc.perform(post(urlEditar)
            .param("titulo", "Limpiar cristales coche"))
            .andExpect(status().is3xxRedirection())
            .andExpect(redirectedUrl(urlRedirect));

        // Y si realizamos un listado de las tareas del usuario
        // ha cambiado el título de la tarea modificada

        String urlListado = "/usuarios/" + usuarioId + "/tareas";

        this.mockMvc.perform(get(urlListado))
            .andExpect(content().string(containsString("Limpiar cristales
coche")));
    }
}

```

```

    }
}
```

Fichero src/test/java/madstodolist/UsuarioWebTest.java:

```

package madstodolist;

import madstodolist.model.Usuario;
import madstodolist.service.UsuarioService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.web.servlet.MockMvc;

import static org.hamcrest.Matchers.containsString;
import static org.mockito.Mockito.when;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@SpringBootTest
@AutoConfigureMockMvc
//
// A diferencia de los tests web de tarea, donde usábamos los datos
// de prueba de la base de datos, aquí vamos a practicar otro enfoque:
// moquear el usuarioService.
public class UsuarioWebTest {

    @Autowired
    private MockMvc mockMvc;

    // Moqueamos el usuarioService.
    // En los tests deberemos proporcionar el valor devuelto por las llamadas
    // a los métodos de usuarioService que se van a ejecutar cuando se
realicen
    // las peticiones a los endpoint.
    @MockBean
    private UsuarioService usuarioService;

    @Test
    public void servicioLoginUsuarioOK() throws Exception {
        // GIVEN
        // Moqueamos la llamada a usuarioService.login para que
        // devuelva un LOGIN_OK y la llamada a usuarioService.findByEmail
        // para que devuelva un usuario determinado.

        Usuario anaGarcia = new Usuario("ana.garcia@gmail.com");
        anaGarcia.setId(1L);

        when(usuarioService.login("ana.garcia@gmail.com", "12345678"))
            .thenReturn(UsuarioService.LoginStatus.LOGIN_OK);
        when(usuarioService.findByEmail("ana.garcia@gmail.com"))
```

```

        .thenReturn(anaGarcia);

    // WHEN, THEN
    // Realizamos una petición POST al login pasando los datos
    // esperados en el mock, la petición devolverá una redirección a la
    // URL con las tareas del usuario

    this.mockMvc.perform(post("/login")
        .param("eMail", "ana.garcia@gmail.com")
        .param("password", "12345678"))
        .andExpect(status().is3xxRedirection())
        .andExpect(redirectedUrl("/usuarios/1/tareas"));
    }

    @Test
    public void servicioLoginUsuarioNotFound() throws Exception {
        // GIVEN
        // Moqueamos el método userService.login para que devuelva
        // USER_NOT_FOUND
        when(userService.login("pepito.perez@gmail.com", "12345678"))
            .thenReturn(UserService.LoginStatus.USER_NOT_FOUND);

        // WHEN, THEN
        // Realizamos una petición POST con los datos del usuario mockeado y
        // se debe devolver una página que contenga el mensaje "No existe
        // usuario"
        this.mockMvc.perform(post("/login")
            .param("eMail", "pepito.perez@gmail.com")
            .param("password", "12345678"))
            .andExpect(content().string(containsString("No existe
usuario")));
    }

    @Test
    public void servicioLoginUsuarioErrorPassword() throws Exception {
        // GIVEN
        // Moqueamos el método userService.login para que devuelva
        // ERROR_PASSWORD
        when(userService.login("ana.garcia@gmail.com", "000"))
            .thenReturn(UserService.LoginStatus.ERROR_PASSWORD);

        // WHEN, THEN
        // Realizamos una petición POST con los datos del usuario mockeado y
        // se debe devolver una página que contenga el mensaje "Contraseña
incorrecta"
        this.mockMvc.perform(post("/login")
            .param("eMail", "ana.garcia@gmail.com")
            .param("password", "000"))
            .andExpect(content().string(containsString("Contraseña
incorrecta")));
    }
}

```

4. Metodología de desarrollo

En cuanto a la metodología de desarrollo, en esta práctica repasaremos e introduciremos el uso de:

- [Git](#) como sistema de control de versiones que nos permitirá registrar paso a paso los cambios realizados en el desarrollo, realizando e integrando ramas de *features* en las que desarrollaremos pequeños incrementos que añadirán poco a poco las funcionalidades necesarias en la aplicación.
- [GitHub](#) como servicio en el que publicaremos los cambios e integraremos las ramas usando pull requests (PRs). Utilizaremos un gran número de características de GitHub para realizar el seguimiento del desarrollo del proyecto: issues, labels, milestones, etc.
- JUnit y las [características de testing de Spring Boot](#) para realizar continuamente pruebas unitarias que validen el desarrollo.

El objetivo es desarrollar software de una forma similar a cómo se hace en cientos de proyectos punteros de desarrollo *open source*.

Existen muchos proyectos que tienen un desarrollo abierto, transparente, en GitHub. Podemos aprender de sus metodologías estudiándolos. A continuación listamos ejemplos de repositorios en GitHub interesantes, en los que podemos estudiar los procesos de *pull requests*, issues, tableros, etc. y las dinámicas de comunicación que utilizan.

- [CartoDB](#). Software español para representación visual de datos geográficos.
- [Vapor](#). Framework web en Swift.
- [Guice](#). Framework de inyección de dependencias en Java.
- [swift-nio](#). Framework asíncrono de entrada-salida en Swift.
- [Spring Boot](#). Framework web en Java.

Git

Git es el sistema de control de versiones más utilizado en la actualidad. Es muy flexible, distribuido, adaptable a múltiples flujos de trabajo e ideal para una metodología de desarrollo en equipo. Suponemos que ya tienes cierta experiencia con su uso. Puedes usar los siguientes enlaces para repasar su funcionamiento.

- [Resumen de comandos de Git](#): Resumen de comandos principales para empezar a trabajar con Git.
- [Atlassian Git Tutorials](#): Tutoriales muy orientados al uso de Git con gran cantidad de ejemplos. Es recomendable repasar los tutoriales básicos [Getting Started](#) y los tutoriales [Syncing](#) y [Using Branches](#) en el apartado *Collaborating*.
- [Libro de Scott Chacon](#): Completo manual con todos los detalles de todos los comandos de Git.

Cuando utilicemos git es muy importante realizar unos mensajes de *commit* claros. Un mensaje de *commit* es la forma de comunicar a los compañeros del equipo qué cambios se han introducido en la aplicación y ponerlos en contexto (explicar por qué se han hecho, dar algún detalle de implementación, etc.). El post [How to Write a Git Commit Message](#) explica muy bien esto.

Flujo de trabajo

Desarrollaremos la aplicación de forma iterativa, utilizando inicialmente un flujo de trabajo Git denominado *feature branch* (consultar la [guía de GitHub](#)) en el que cada característica nueva se implementa en una rama separada que después se mezcla con la rama principal de desarrollo. Más adelante veremos otros flujos de trabajo. Puedes ver una introducción a distintos flujos de trabajo básicos con Git en este [documento de Atlassian](#).

Para implementar este flujo de trabajo utilizaremos los siguientes instrumentos de GitHub que facilitan la comunicación entre los miembros del equipo:

- **Issues (incidencias):** GitHub permite abrir issues (incidencias o tareas), asignarlos a personas, realizar comentarios, asignar etiquetas y cerrarlos cuando la implementación ha terminado. Consultar [Mastering Issues](#).

	Author	Labels	Project
<input type="checkbox"/> ⓘ 0 Open ✓ 22 Closed			
<input type="checkbox"/> ⓘ Autorización sencilla 005 Autorización	#41 by domingogallardo	was closed yesterday	
<input type="checkbox"/> ⓘ Redirección login y logout enhancement	#39 by domingogallardo	was closed 2 days ago	
<input type="checkbox"/> ⓘ Controller y vista para borrar tarea 004 Editar y borrar tareas	#37 by domingogallardo	was closed 2 days ago	
<input type="checkbox"/> ⓘ Controller y vista para modificar una tarea 004 Editar y borrar tareas	#35 by domingogallardo	was closed 2 days ago	
<input type="checkbox"/> ⓘ Métodos de servicio para borrar y editar tareas 004 Editar y borrar tareas	#33 by domingogallardo	was closed 2 days ago	
<input type="checkbox"/> ⓘ Listado de tareas de un usuario 003 Listar tareas	#31 by domingogallardo	was closed 4 days ago	
<input type="checkbox"/> ⓘ Añadir @Transactional en todos los métodos de servicio bug	#27 by domingogallardo	was closed 4 days ago	

Definiremos distintos tipos de issues en función de su propósito: *bug*, *technical*, *enhancement*. Los issues que implementan una historia de usuario los etiquetaremos con el código de la historia de usuario. Puede haber más de un issue asociado con una historia de usuario y de esta forma podemos agruparlos.

The screenshot shows a GitHub Issues page with the following structure:

- Header navigation: Code, Issues (0), Pull requests (0), Projects (1), Wiki.
- Section tabs: Labels (selected), Milestones, Search all labels.
- Label count: 8 labels.
- Issues listed:
 - 001 Login de usuarios (yellow)
 - 002 Crear tareas
 - 003 Listar tareas
 - 004 Editar y borrar tareas
 - 005 Autorización (blue)
- Label categories:
 - bug: Something isn't working
 - enhancement: New feature or request
 - technical

Cada issue se desarrollará en una rama de Git y se integrará en la rama `main` haciendo un pull request.

- **Pull Requests:** Un pull request permite avisar al equipo de que se va a integrar en la rama principal una rama con un desarrollo nuevo. Cuando creamos un PR, GitHub crea una página en la que se pueden realizar comentarios, revisiones de código o definir políticas de aceptación del PR. Consultar [About pull requests](#).

Implementaremos cada issue en una rama separada de git y la integraremos en la rama `main` haciendo un *pull request*. Cuando se mezcle el PR en `main` el issue se cerrará.

Añadida página 'Acerca de' #3

Merged domingogallardo merged 2 commits into master from acerca-de 3 days ago

Conversation 0 Commits 2 Checks 0 Files changed 4 +14 -2

domingogallardo commented 3 days ago • edited Closes #2

domingogallardo added some commits 3 days ago

- Cambiado nombre del proyecto
- Añadida página 'acerca de'

domingogallardo merged commit ae1abe5 into master 3 days ago Revert

domingogallardo deleted the acerca-de branch 3 days ago Restore branch

domingogallardo added this to the 1.0.0 milestone 2 days ago

Reviewers: No reviews

Assignees: No one—assign yourself

Labels: feature

Projects: None yet

Milestone: 1.0.0

Más adelante añadiremos otra rama de largo recorrido releases para incluir en ella las releases del proyecto.

- **Milestones y Releases:** Etiquetaremos cada issue con el *milestone* en el que queremos que se lance. Para identificar el milestone usaremos el **versionado semántico**: MAJOR.MINOR.PATCH.

Issues 5 Pull requests 0 Projects 1 Wiki Insights Settings

Milestones

New milestone

2 Open 1 Closed Sort ▾

1.0.0

Closed 2 days ago Last updated 2 days ago Versión inicial

100% complete 0 open 2 closed

Edit Reopen Delete

Usaremos la funcionalidad de GitHub *Releases* para etiquetar los commits en los que queramos marcar una versión nueva del proyecto. Podemos añadir información sobre las novedades de la versión (normalmente serán enlaces a los issues ese milestone).

The screenshot shows the GitHub Releases page for the 'ToDoList 1.0.0' release. It includes the latest release information (v1.0.0, commit 4f06462), the release author (domingogallardo), and a brief description of the initial version. Below the description, there is a section for 'Assets' containing two items: 'Source code (zip)' and 'Source code (tar.gz)'. There is also an 'Edit release' button.

- **Tablero de proyecto:** Un tablero de proyecto en GitHub nos ayudará a hacer un seguimiento de en qué estado se encuentra cada issue: cuáles han sido implementados, cuáles faltan por asignar, implementar, probar, etc. Vamos a utilizar la funcionalidad propia de GitHub llamada *Projects*. Consultar [Quickstart for Projects](#)

The screenshot shows a GitHub Projects board titled 'Mads ToDoList'. It features four columns: 'To do', 'In progress', 'In Pull Request', and 'Done'. Each column contains several cards representing issues or pull requests. For example, the 'To do' column has a card for 'Autorización sencilla' (issue #41), and the 'In Pull Request' column has a card for 'Refactorización' (pull request #43). The 'Done' column contains several completed tasks related to login, controllers, and service methods.

Cuando se crea un pull request que resuelve un issue enlazaremos el issue con el pull request. Podremos ver en el tablero que bajo el issue aparece su PR enlazado y podremos desplegarlo en la propia tarjeta (funcionalidad nueva de GitHub).

This screenshot illustrates the linking of an issue to its pull request. On the left, a card for issue '#49' (Actualizar la versión de Spring Boot) shows a link to a pull request '#49' (Actualizada versión a 2.1.16). A blue arrow points from the issue card to the linked pull request card on the right, which is highlighted with a blue border.

También utilizaremos un panel de [Trello](#) para representar las historias de usuario que se van implementando en el proyecto.



Cada historia de usuario tendrá un código numérico y podrá implementarse con uno o más issues. En GitHub crearemos una etiqueta por cada historia de usuario y se la asignaremos a los issues que se usen para implementarla.

🔥 Importante

Puede parecer redundante el uso de dos tableros, uno para las historias de usuario y otro para los issues y PR. La justificación es que los objetivos de ambos tableros son distintos (y los contenidos también). El tablero Trello es un **tablero de funcionalidades de usuario**, que es gestionado por el *product owner*, usado por el equipo de desarrollo y puede ser compartido también con clientes y gerencia. En la terminología de Scrum será el *product backlog*. Mientras que el tablero de GitHub será un **tablero técnico** gestionado por el equipo de desarrollo. En terminología de Scrum será el *sprint backlog*.

La documentación en Trello y en GitHub (en los issues, en los PRs y en el propio README .md del proyecto) hay que escribirla en **Markdown**, un lenguaje de marcado muy popular y sencillo de dominar. Si no has trabajado todavía con él puedes leer estas [guías de GitHub](#).

✍ Nota

Existen herramientas y servicios más avanzados para gestionar todos estos elementos del desarrollo. Por ejemplo [Jira](#), [YouTrack](#) o [Confluence](#). Pero la combinación de GitHub + Trello es suficiente para lo que vamos a realizar en la asignatura y para aprender los objetivos y el funcionamiento de estos tipos de sistemas basados en incidencias.

5. Antes de empezar la práctica

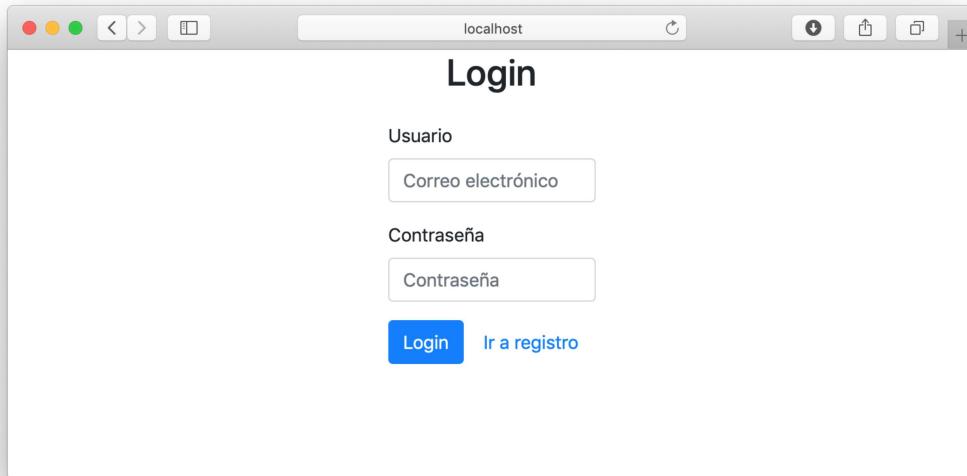
1. Una vez logeado en GitHub, copia el enlace con una invitación que compartiremos en el foro de Moodle. Con esa invitación se creará automáticamente tu repositorio `mads-todolist-<usuario>` en la organización `mads-ua`. Al igual que el repositorio de la primera parte de la práctica es un repositorio privado al que tienes acceso tú y el profesor. Contiene el código inicial de un proyecto base (es una copia del repositorio [domingogallardo/mads-todolist-inicial](https://github.com/domingogallardo/mads-todolist-inicial)) en la que se han comprimido todos los commits en uno.

Es importante que tengas en cuenta que el repositorio recién creado no reside en tu cuenta, sino en la organización `mads-ua-22-23`. Puedes acceder a él desde el *dashboard* de GitHub que aparece cuando te logeas.

2. Descarga el proyecto y comprueba que se compila y ejecuta correctamente:

```
$ git clone https://github.com/mads-ua/mads-todolist-<usuario>.git  
$ cd mads-todolist-<usuario>  
$ ./mvnw spring-boot:run
```

Comprueba que la aplicación está funcionando en <http://localhost:8080/login> en la máquina host.

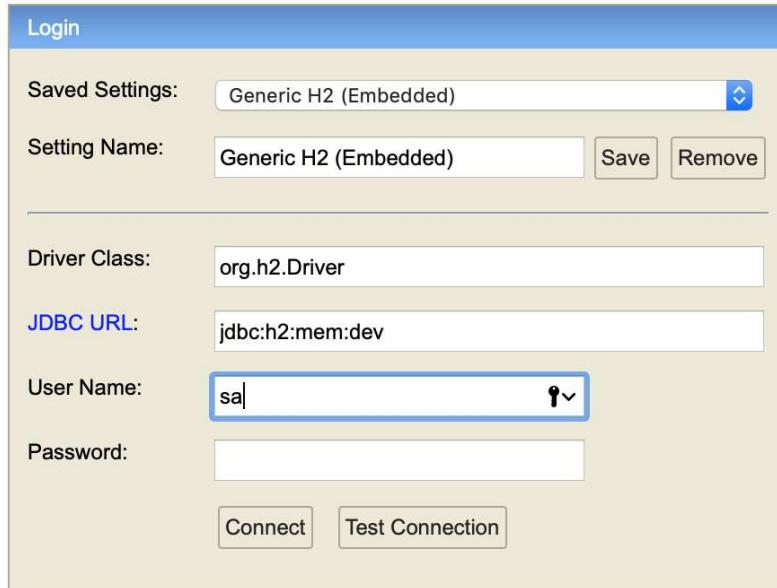


Para la aplicación haciendo CTR+C en el terminal.

3. Importa el proyecto en IntelliJ para trabajar, ejecutar los tests y lanzar la aplicación desde este entorno.
4. Es posible examinar el esquema de la base de datos y los datos accediendo a la base de datos H2 en memoria añadiendo las siguientes preferencias:

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Una vez lanzada la aplicación, podemos acceder a <http://localhost:8080/h2-console> introduciendo como JDBC URL la dirección de la fuente de datos `jdbc:h2:mem:dev` y como User name la cadena `sa`



Y examinar tablas en concreto:

ID	TITULO	USUARIO_ID
1	Lavar coche	1
2	Renovar DNI	1

6. Desarrollo de la práctica

En esta primera práctica vamos a desarrollar las siguientes historias de usuario o features:

1. Página Acerca de
2. Barra de menú

3. Página listado de usuarios
4. Página descripción de usuario
5. Usuario administrador (opcional)
6. Protección del listado y descripción de usuarios (opcional)
7. Bloqueo de usuarios por el usuario administrador (opcional)

La práctica va a consistir en la realización en tu proyecto de todos los elementos necesarios para implementar estas features : tablero Trello, issues, pull requests (con sus *commits* en los que se desarrolla paso a paso cada issue) y tablero del proyecto.

Haremos paso a paso la historia de usuario 1, creando la primera versión 1.0.1 de la aplicación. Las siguientes características las deberás desarrollar tu mismo y entregar la versión 1.1.0.

Versión 1.0.1

La versión 1.0.1 será la versión inicial de la aplicación. Desarrollaremos en esta versión la primera característica: **Página Acerca de**.

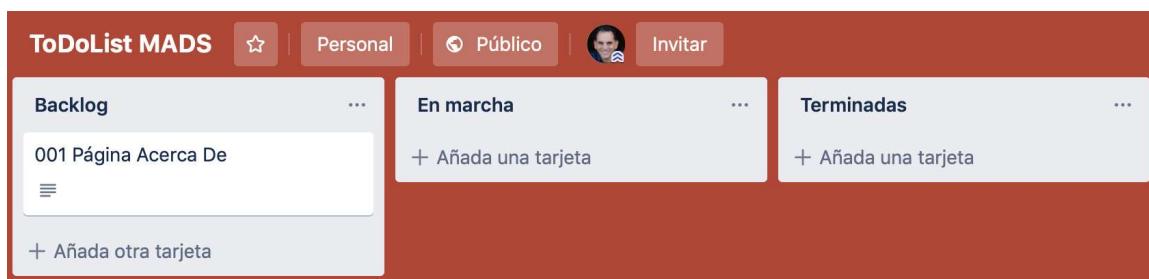
Tablero Trello

Crea un en Trello un **tablero público** llamado `ToDoList MADS`. Va a servir como *backlog* de las historias de usuario que debes realizar en la práctica. Añade en él 3 columnas, tal y se explica en el apartado anterior de metodología de desarrollo.

Añade el enlace en la descripción del repositorio GitHub, para que el profesor pueda acceder a consultar el estado del proyecto.

Un ejemplo de tablero es el [Trello del proyecto mads-todolist-inicial](#).

Utilizaremos el tablero Trello para documentar las características a desarrollar en la aplicación. Deberá haber una tarjeta para cada característica. Cada característica deberá tener un número y un título.



Añade la descripción de la característica **Página Acerca de**:

001 Página Acerca De

en la lista Backlog

Descripción Editar

Historia de usuario

La aplicación tendrá una página Acerca De en la que se incluirá:

- Equipo de desarrollo
- Número y fecha de la versión de la aplicación

Detalles

- Pondremos inicialmente el enlace a la página *acerca de* en la página de login.
- Cuando se actualice la versión de la aplicación también se actualizará en esta página.
- La fecha de la versión se corresponderá con la fecha en la que se crea la nueva versión en GitHub.

COS (Condiciones de satisfacción)

- Acceder a la página de login y pinchar en el enlace para acceder al Acerca de. Deberá aparecer la página con los autores del proyecto y el número y la fecha de versión que se está ejecutando.

Actividad Mostrar detalles

Escriba un comentario...

AÑADIR A LA TARJETA

- Miembros
- Etiquetas
- Checklist
- Vencimiento
- Adjunto
- Portada

POWER-UPS

- Conseguir Power-Ups

ACCIONES

- Mover
- Copiar
- Seguir
- Archivar

Cuando empecemos a trabajar en la historia de usuario moveremos la tarjeta a *En marcha* y cuando la hayamos terminado de testear e integrar en la rama principal la moveremos a *Terminadas*.

Tablero de GitHub

GitHub ha cambiado recientemente la forma de gestionar visualmente los *issues* para hacerla mucho más flexible y potente.

En la versión actual, la funcionalidad se denomina *Proyectos*. Un proyecto está asociado a un usuario de GitHub y puede contener issues de más de un repositorio. Un usuario puede crear los proyectos que considere necesarios. También se pueden crear proyectos asociados a organizaciones.

En cuanto a la forma de visualizar los issues, podemos seleccionar dos formas: como un tablero o como una hoja de cálculo. La primera forma es más sencilla y la segunda más potente. En la asignatura usaremos la primera.

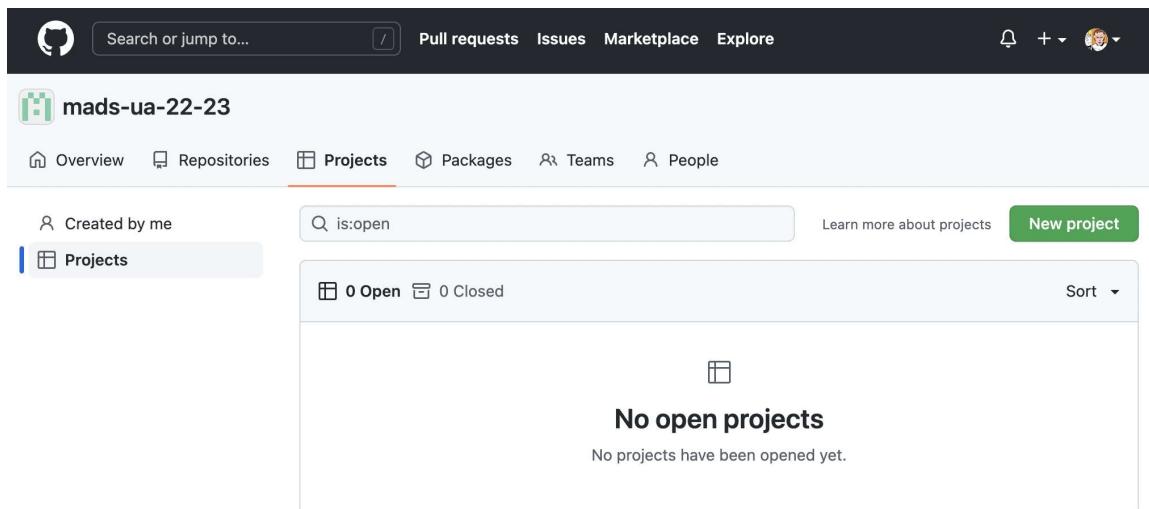
Puedes encontrar más información sobre los GitHub Projects en [este enlace](#).

En la asignatura vamos a usar los proyectos de GitHub para mostrar, en forma de tablero, los issues del repositorio de la práctica. También podremos acceder a los pull requests desde cada uno de los issues (enlazaremos los issues a su pull request).

Aviso

Aunque se han añadido en los apuntes las nuevas imágenes sobre el proceso de creación de un proyecto, no se han actualizado todas las imágenes en las que aparece el tablero de proyectos, por lo que puede que alguna imagen no represente fielmente la apariencia real que tiene en la actualidad.

Lo primero que debes hacer es crear un proyecto desde el enlace [Projects](#) en la organización [mads-ua-22-23](#).



The screenshot shows the GitHub organization page for 'mads-ua-22-23'. The 'Projects' tab is selected. The interface includes a search bar with 'is:open', a 'New project' button, and a message indicating there are no open projects yet.

Selecciona la opción **Board** y ponle como nombre **tu usuario de GitHub**:

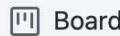
Select a template

X

[Start from scratch](#)



[Table](#)



[Board](#)

[Project templates](#)



New board

Start with a fast and simple task board. Easily switch to a table layout at any time.

[Create](#)

Define una columna adicional *In Pull Request*, entre *In Progress* y *Done*:

En las columnas colocaremos los issues del proyecto (y los PRs estarán enlazados en ellos). GitHub permite automatizar el movimiento de las tarjetas de una columna a otra.

Activa dos flujos de trabajo. Uno para que cuando un issue nuevo se añada al proyecto se coloque en la columna *To Do*:

The screenshot shows the GitHub Workflows interface. On the left, under 'Default workflows', there is a list of items: 'Item added to project' (selected), 'Item reopened', 'Item closed', 'Code changes requested', 'Code review approved', and 'Pull request merged'. The main area displays a workflow named 'Item added to project' which is enabled. The workflow consists of two steps: 'When issue is added' and 'Set Status:Todo'.

```

graph TD
    A[When issue is added] --> B[Set Status:Todo]

```

Y otro para que cuando se cierre un issue se mueva a la columna de *Done*:

The screenshot shows the GitHub Workflows interface. On the left, under 'Default workflows', there is a list of items: 'Item added to project' (selected), 'Item reopened', 'Item closed', 'Code changes requested', 'Code review approved', and 'Pull request merged'. The main area displays a workflow named 'Item closed' which is enabled. The workflow consists of two steps: 'When issue is:closed' and 'Set Status:Done'.

```

graph TD
    A[When issue is:closed] --> B[Set Status:Done]

```

El resto de cambios de los issues los tendrás que hacer manualmente. Por ejemplo, cuando crees el *pull request* asociado a un issue tendrás que mover el issue a la columna de *In Pull Request*.

En resumen, las condiciones de las fichas que habrá en cada columna son las siguientes:

- **Columna To do :** Nuevos issues añadidos al proyecto. Cuando añadimos el proyecto al issue (en la página del issue) GitHub lo coloca automáticamente en esta columna.
- **Columna In progress :** issues que se han comenzado a implementar (se ha creado una rama su desarrollo). Manual.
- **Columna In pull request :** moveremos a esta columna el issue abriremos un PR y lo enlazemos con el issue. Manual. GitHub lo coloca automáticamente en esta columna. implementado por el pull request manualmente.
- **Columna Done :** pull requests cerrados. GitHub lo detecta automáticamente.

Por último, en la opción *Settings > Manage access* comparte el tablero con mi usuario de GitHub `domingogallardo`, para que pueda revisarlo:

Who has access

Private project
Only those with access to this project can view it.

Manage

Invite collaborators

Search by username: domingogallardo Role: Read Invite

Y cambia el *base role* a modo *No access* para solo tengan acceso al tablero las personas colaboradoras.

Who has access

Private project
Only those with access to this project can view it.

Base role
Only those with direct access and **owners** can see this project. Owners are also admins of this project.

No access ✓ Changes saved

Admin
Can see, make changes to, and add new collaborators to this project.

Write
Can see and make changes to this project.

Read
Can see this project.

✓ No access
Organization members will only be able to see this project if it's public. To give an organization member additional access, they can be added as part of a team or as a collaborator.

Role: Write Invite

Por último, desde la página *Projects* del repositorio, añade el proyecto al repositorio:

mads-ua-22-23 / mads-todolist-domingogallardo2 Private

generated from domingogallardo/mads-todolist-initial

Code Issues 1 Pull requests Actions Projects Wiki Security Insights Settings

Projects

Learn more about projects Add project

Quick access 0

Search projects

domingogallardo2
mads-ua-22-23

Go to the mads-ua-22-23 organization to create a new project

Quick access to ...

Add projects from your organization that you'd like to view and access from this repository.

Issues

Añade en el proyecto las etiquetas que vamos a usar inicialmente.

4 labels

001 Acerca de

- bug** Something isn't working
- enhancement** New feature or request
- technical** Technical issue

Crea el primer issue, correspondiente a la *feature* a desarrollar **Página Acerca de**.

Añadir página 'Acerca de' #1

Open domingogallardo2 opened this issue 5 minutes ago · 0 comments

domingogallardo2 commented 5 minutes ago • edited

- Añadir página 'Acerca de' con:
 - Nombre del autor del proyecto
 - Número y fecha de versión inicial
- Modificar el nombre del proyecto (`mads-todolist-dgallardo`) en el fichero `pom.xml`

Crea el milestone 1.0.1. Y, desde la página del issue, añade el milestone y el proyecto. Automáticamente se añadirá en la columna **To Do**.

Añadir página 'Acerca de' #1

Open domingogallardo2 opened this issue 2 hours ago · 0 comments

domingogallardo2 commented 2 hours ago

- Añadir página 'Acerca de' con:
 - Nombre del autor del proyecto
 - Número y fecha de versión
- Modificar el nombre del proyecto en fichero pom.xml

domingogallardo2 added the 001 Acerca de label 2 hours ago

domingogallardo2 added this to the 1.0.1 milestone 2 hours ago

Assignees
No one—assign yourself

Labels
001 Acerca de

Projects
Recent Repository Organization

Projects
Filter projects
Recent Repository Organization

✓ domingogallardo2
mads-ua-22-23

Create a branch for this issue or link a pull request.

En el listado de issues del repositorio debe aparecer este recién creado:

Code Issues 1 Pull requests 0 Projects 1 Wiki

Filters ▾ is:issue is:open Labels 4

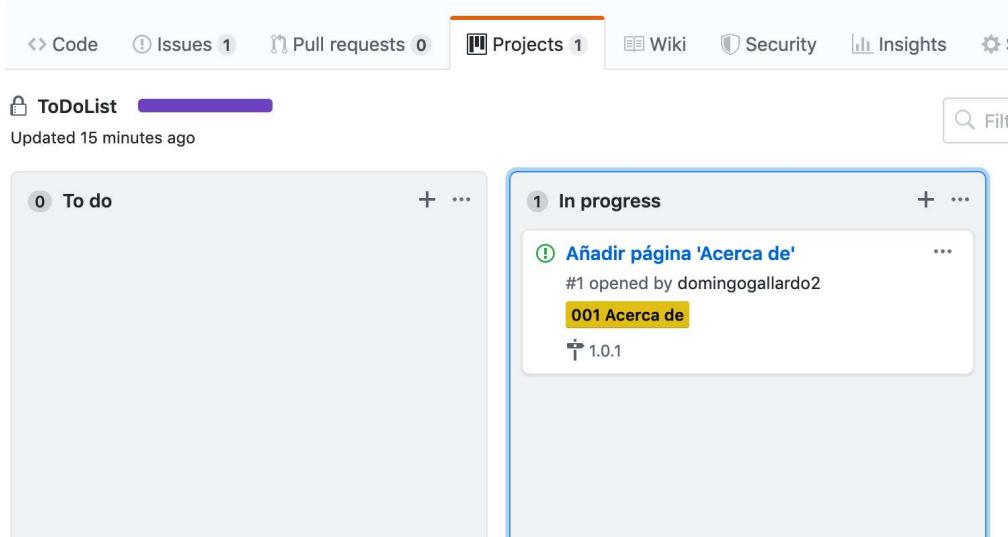
ⓘ 1 Open ✓ 0 Closed Author ▾ La

ⓘ Añadir página 'Acerca de' 001 Acerca de
#1 opened now by domingogallardo2 1.0.1

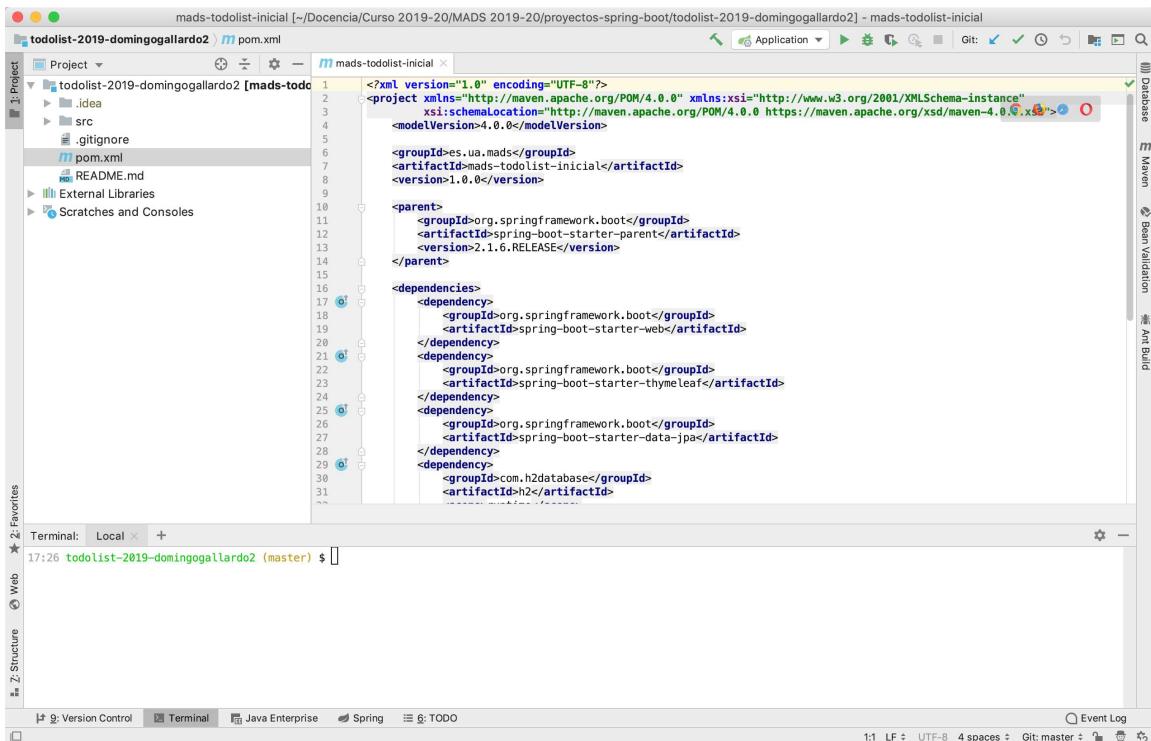
Desarrollo

Para desarrollar el issue abriremos una rama en Git, realizaremos commits sobre ella hasta estar terminado y después crearemos un *pull request* en GitHub para realizar la integración con la rama `main`.

Mueve en el tablero la tarjeta con el issue a la columna `In progress`.



Empezamos el desarrollo importando el proyecto en IntelliJ y abriendo un terminal para trabajar con Git:



En el terminal escribimos los comandos para crear la rama en la que desarrollaremos la feature y subirla:

```
(main) $ git checkout -b acerca-de
(acerca-de) $ git push -u origin acerca-de
```

PRIMER COMMIT

Hacemos un primer commit.

Cambia en `pom.xml` el nombre del proyecto (`artifactId`) a `mads-todolist-<tu-nombre>` y la versión a `1.0.1-SNAPSHOT`. El sufijo `SNAPSHOT` indica *en desarrollo*. Cuando hagamos el release de la versión 1.0.1 eliminaremos el sufijo.

Realiza el commit y súbelo a GitHub:

```
(acerca-de) $ git status (comprobamos los ficheros que han cambiado)
On branch acerca-de
Your branch is up to date with 'origin/acerca-de'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md
    modified:   pom.xml

no changes added to commit (use "git add" and/or "git commit -a")
(acerca-de) $ git add .
(acerca-de) $ git status (comprobamos que está listo para añadirse en el
commit)
(acerca-de) $ git commit -m "Cambiado el nombre del proyecto y empezamos
versión 1.0.1"
On branch acerca-de
Your branch is up to date with 'origin/acerca-de'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README.md
    modified:   pom.xml
(acerca-de) $ git push
```

Consulta en GitHub que el *commit* se ha subido en GitHub:

acerca-de

- o Commits on Sep 5, 2020
 - Cambiamos el nombre del proyecto y empezamos la versión 1.0.1**
domingogallardo2 committed 1 minute ago
- o Commits on Sep 4, 2020
 - Initial commit**
domingogallardo committed yesterday

De esta forma habrás comprobado que tienes permiso de escritura en el repositorio y que ya puedes comenzar a realizar la práctica.

SEGUNDO COMMIT

En el segundo commit incluiremos el desarrollo de los elementos necesarios para la página *acerca de*:

- Acción en controller
- Vista

Añade los siguientes ficheros:

Controller main/java/madstodolist/controller/HomeController.java

```
package madstodolist.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {

    @GetMapping("/about")
    public String about(Model model) {
        return "about";
    }
}
```

Vista main/resources/templates/about.html:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

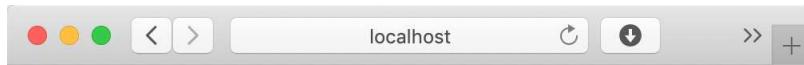
<head th:replace="fragments :: head (titulo='Acerca de')"></head>

<body>
<div class="container-fluid">
    <div class="container-fluid">
        <h1>ToDoList</h1>
        <ul>
            <li>Desarrollada por TU NOMBRE </li>
            <li>Versión 1.0.1 (en desarrollo)</li>
            <li>Fecha de release: pendiente de release</li>
        </ul>
    </div>
</div>

<div th:replace="fragments::javascript"/>

</body>
</html>
```

Prueba la página accediendo a la url <http://localhost:8080/about>.



ToDo List

- Desarrollada por Domingo Gallardo
- Versión 1.0.1 (en desarrollo)
- Fecha de release: pendiente de release

Añade un test que automatiza la comprobación de que la URL `/about` debe devolver el nombre de la aplicación.

Test test/java/madstodolist/AcercaDeWebTest.java:

```
package madstodolist;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;

import static org.hamcrest.Matchers.containsString;
import static
```

```

org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;

@SpringBootTest
@AutoConfigureMockMvc
public class AcercaDeWebTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void getAboutDevuelveNombreAplicacion() throws Exception {
        this.mockMvc.perform(get("/about"))
            .andExpect(content().string(containsString("ToDoList")));
    }
}

```

Puedes lanzar el test pulsando en IntelliJ con el botón derecho en el fichero (en el panel del proyecto) y seleccionando la opción *Run AcercaDeWebTest*.

Puedes lanzar también todos los tests en el terminal para comprobar que no se ha roto nada.

```

(acerca-de) $ ./mvnw test
...
[INFO]
[INFO] Tests run: 34, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
-
[INFO] BUILD SUCCESS
[INFO] -----
-
[INFO] Total time: 21.879 s

```

Por último, confirma el commit en la rama y súbelo a GitHub. En el panel Git:

```

(acerca-de) $ git add .
(acerca-de) $ git status (comprueba que se han añadido los ficheros)
(acerca-de) $ git commit -m "Añadida vista y controller 'about'"
(acerca-de) $ git push

```

TERCER COMMIT

En el tercer commit pondremos un enlace a la página *acerca de* en la página de login de la aplicación.

Realiza el siguiente cambio:

Fichero formLogin.html:

```

<a class="btn btn-link" href="/registro">Ir a
registro</a>

```

```
+           <a class="btn btn-link" href="/about">Acerca de</a>
      </div>
    </form>
```

Prueba que funciona correctamente, prueba los tests, haz el commit y súbelo a GitHub:

```
(acerca-de) $ git status
(acerca-de) $ git add .
(acerca-de) $ git commit -m "Añadido enlace a página 'about' en página 'login'"
(acerca-de) $ git push
```

Pull request

Una vez terminada la implementación de la feature en la rama, creamos un pull request en GitHub para indicar que estamos listos para mezclar la rama con la feature con la rama principal de desarrollo (*main*).

CREACIÓN DEL PULL REQUEST

Accede en GitHub a la rama `acerca-de` y comprueba que se han subido todos los cambios pulsando `Compare`.

The screenshot shows a GitHub repository page for 'mads-ua-20-21 / todolist-domingogallardo2'. The 'acerca-de' branch is selected. The main content area shows a message: 'acerca-de had recent pushes 6 minutes ago'. Below it, there's a summary: 'This branch is 3 commits ahead of main.' To the right, there are buttons for 'Compare & pull request', 'Go to file', 'Add file', and a green 'Code' button. A red arrow points from the bottom right towards the 'Compare' button. At the bottom, a list of commits is shown:

- domingogallardo2** Añadido enlace a página 'about' en página 'login' (8aee033, 7 minutes ago) 4 commits
- .mvn/wrapper Initial commit (yesterday)
- src Añadido enlace a página 'about' en página 'login' (7 minutes ago)

Aparecerá la siguiente página, con la información de los cambios que introducen todos los commits de la rama:

base: main ▾ compare: acerca-de ▾ ✓ Able to merge. These branches can be automatically merged.

Discuss and review the changes in this comparison with others. [Learn about pull requests](#)

Create pull request

3 commits 5 files changed 0 comments 1 contributor

Commits on Sep 05, 2020

- Cambiamos el nombre del proyecto y empezamos la versión 1.0.1 8a296f1
- Añadida vista y controller 'about' a03ec7f
- Añadido enlace a página 'about' en página 'login' 8aee033

Showing 5 changed files with 41 additions and 3 deletions.

Unified Split

```

v 2 README.md 
... @@ -21,7 +21,7 @@ También puedes generar un `jar` y ejecutarlo:
21 21
22 22
23 23 $ ./mvnw package
24 - $ java -jar target/mads-todolist-inicial-0.0.1-SNAPSHOT.jar
24 + $ java -jar target/mads-todolist-dgallardo-1.0.1-SNAPSHOT.jar
25 25
26 26

```

Pulsa después el botón *Create pull request* para crear el pull request.

Escribe como título del PR: Añadida página 'Acerca de' y en el comentario escribe:

Closes #1

Verás que al escribir `#1` aparecerá el nombre del issue. Si escribes sólo `#` verás una lista de los últimos issues.

De esta forma estamos enlazando el PR con el issue. Cuando se cierre el pull request se cerrará automáticamente el issue. También podremos acceder desde el issue al PR enlazado.

Pulsa en el botón para crear el pull request. Debe quedar la siguiente pantalla en la que informa del PR recién creado:

Añadida página 'Acerca de' #2

domingogallardo2 wants to merge 3 commits into `main` from `acerca-de`

Conversation 0 **Commits** 3 **Checks** 0 **Files changed** 5 +41 -3

Reviewers: **domingogallardo** Request
Suggestions
Assignees: No one—assign yourself
Labels: None yet

En el proyecto mueve la tarjeta con el issue a la columna `In Pull Request`. Verás que se ha añadido en la parte inferior de la tarjeta un desplegable con la información sobre el PR enlazado.

En este momento los compañeros del equipo podrían revisar el pull request y el código que se va a introducir. En la propia página del pull request es posible conversar y realizar comentarios que puede aclarar el autor del PR. Y también es posible subir nuevos commits con modificaciones o ampliaciones correspondientes a las sugerencias indicadas.

Haremos esto en futuras prácticas.

Podemos ver que GitHub informa de que no hay conflictos con la rama `main` y que es posible hacer el merge en GitHub.

Antes de pulsar el botón para realizar el merge, lanzamos los tests (estando en la rama) para comprobar que no se ha roto nada y que los tests que se han añadido pasan correctamente

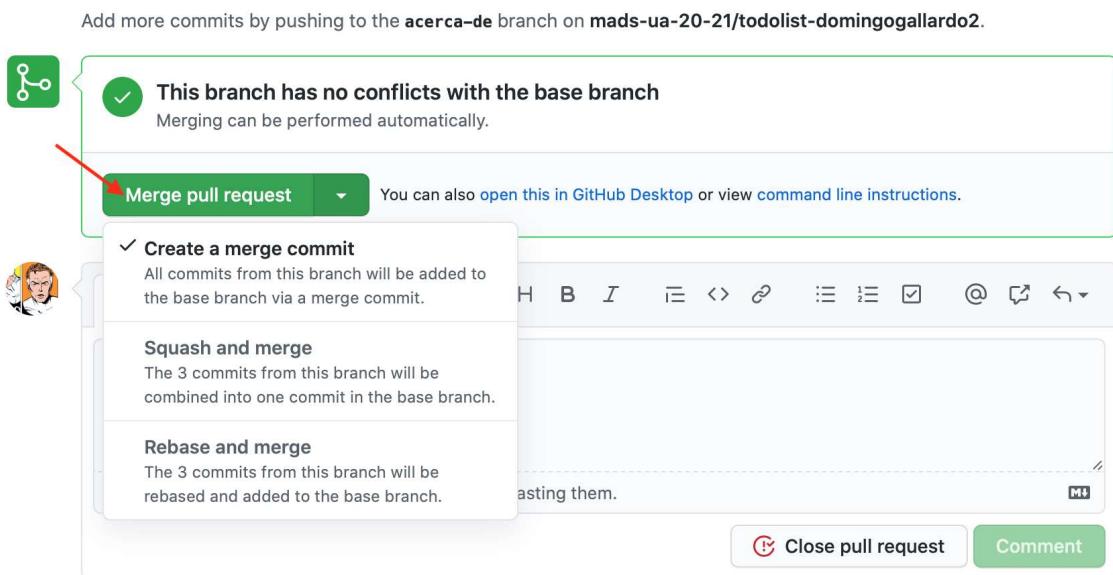
(en este caso no hemos añadido ninguno).

```
(acerca-de) $ ./mvnw test
...
[INFO]
[INFO] Tests run: 34, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
-
[INFO] BUILD SUCCESS
[INFO] -----
-
[INFO] Total time: 21.879 s
```

Veremos en la próxima práctica cómo configurar GitHub para que esta comprobación se haga de forma automática en GitHub.

Aunque deberíamos también comprobar que los tests pasan correctamente después de mezclar la rama con `main`, dejamos de hacer esta comprobación porque a partir de la próxima práctica lo haremos también de forma automática en GitHub.

Pulsa el botón de `Merge pull request` (con la opción por defecto `Create a merge commit`) y confírmalo.



Borra la rama en GitHub, pulsando el botón correspondiente.

The screenshot shows a GitHub pull request merge history. At the top, a message from 'domingogallardo2' indicates they added the pull request to the 'ToDoList' repository via automation 30 minutes ago. Below that, another message from 'domingogallardo2' shows the commit '9527ae2' merged into 'master' 2 minutes ago, with a 'Revert' button. A third message from 'ToDoList' automation moves the pull request to 'Done' 2 minutes ago. A large callout box highlights the merge message: 'Pull request successfully merged and closed' and 'You're all set—the acerca-de branch can be safely deleted.' A red arrow points to a 'Delete branch' button.

Este *merge* lo has hecho en GitHub. Debes por último integrarlo en tu repositorio local. En el terminal:

```
(acerca-de) $ git checkout main
(main) $ git pull (bajamos los cambios)
(main) $ git branch -d acerca-de (borramos la rama)
(main) $ git remote prune origin (borramos referencias a rama remota)
(main) $ git log --oneline --graph --all
*   9527ae2 (HEAD --> main, origin/main, origin/HEAD) Merge pull request #2
from mads-ua-18/acerca-de
  \\
  | *
  | * 672c28f Añadido enlace a página 'about' en página 'login'
  | * 3fdfb83 Añadida ruta, vista y controller 'about'
  | * a332017 Cambiado el nombre del proyecto y empezamos versión 1.0.0
  |
* 6767016 Commit inicial
```

Comprobamos también la historia de *commits* en GitHub. Aparecerá el *commit* de *merge* introducido por el pull request.

The screenshot shows a GitHub repository interface. At the top, there are navigation links: Code, Issues, Pull requests, Actions, Projects (1), and Wiki. Below this, a dropdown menu shows 'main'. Under the 'Commits' section, there are two entries:

- Merge pull request #2 from mads-ua-20-21/acerca-de** ...
domingogallardo2 committed 3 minutes ago
- Añadido enlace a página 'about' en página 'login'**
domingogallardo2 committed 9 hours ago

Below these, under 'Commits on Sep 5, 2020', are three more commits:

- Añadida vista y controller 'about'**
domingogallardo2 committed 9 hours ago
- Cambiamos el nombre del proyecto y empezamos la versión 1.0.1**
domingogallardo2 committed 9 hours ago

Under 'Commits on Sep 4, 2020', there is one commit:

- Initial commit**
domingogallardo committed yesterday

De esta forma hemos cerrado el PR e integrado su código en la rama principal de desarrollo. El issue ligado al PR se habrá cerrado automáticamente y en el tablero de proyecto debe haber cambiado la tarjeta a la columna `Done`.

Actualizamos tablero Trello

Actualizamos el tablero Trello moviendo la historia de usuario a la columna *Terminadas*.

The screenshot shows a Trello board titled 'ToDoList MADS'. The board has three columns: 'Backlog', 'En marcha', and 'Terminadas'. The 'Backlog' column has a button '+ Añada una tarjeta'. The 'En marcha' column also has a button '+ Añada una tarjeta'. The 'Terminadas' column contains one card with the title '001 Página Acerca De'. There is a button '+ Añada otra tarjeta' at the bottom of this column.

Release 1.0.1

Vamos a ver por último cómo crear un *release* y poner en producción la aplicación. Lo vamos a hacer ahora como ejemplo, creando el release 1.0.1 y tendrás que hacerlo otra vez más al final de la práctica, creando el release 1.1.0.

Para hacer el release haremos un commit directamente sobre la rama `main` (más adelante explicaremos una forma más elaborada de hacer un release, cuando expliquemos el flujo de trabajo de GitFlow).

Crea un commit con la confirmación del número de versión y fecha en los ficheros `pom.xml` y `about.html`

Fichero pom.xml:

```

<groupId>es.ua.mads</groupId>
<artifactId>mads-todolist-dgallardo</artifactId>
- <version>1.0.1-SNAPSHOT</version>
+ <version>1.0.1</version>

```

Fichero about.html:

```

<h1>ToDo List</h1>
<ul>
    <h1>ToDo List</h1>
    <ul>
        <li>Desarrollada por Domingo Gallardo</li>
- <li>Versión 1.0.1 (en desarrollo)</li>
- <li>Fecha de release: pendiente de release</li>
+ <li>Versión 1.0.1</li>
+ <li>Fecha de release: 17/9/2018</li>
    </ul>
}

```

Añadimos el commit y lo subimos a GitHub

```

(main) $ git add .
(main) $ git commit -m "Cambio de versión a 1.0.1"
(main) $ git push

```

Y creamos la versión 1.0.1 en GitHub pulsando en el enlace `Create a new release` en la página principal:

The screenshot shows a GitHub repository page for 'mads-ua-20-21 / todolist-domingogallardo2'. The 'About' section displays the repository was created by 'todolist-domingogallardo2' via GitHub Classroom. It includes a 'Readme' link and a 'Releases' section which states 'No releases published'. A red arrow points to the 'Create a new release' link.

Un release en GitHub se guarda como una etiqueta Git, junto con información asociada. Se suelen indicar las nuevas features añadidas en el release mediante enlaces a los pull requests añadidos.

The screenshot shows a GitHub repository interface. At the top, there are navigation links: Code, Issues (0), Pull requests (0), Projects (1), Wiki, Security, Insights, and a gear icon. Below these, there are two tabs: Releases (which is selected) and Tags. A release card for 'v1.0.1' is displayed, labeled as an 'Existing tag'. The title of the release is 'ToDoList 1.0.1'. There are two buttons at the bottom of the card: 'Write' and 'Preview'. The main body of the card contains the note: '* Añadida página _Acerca de_'. At the bottom, there is a placeholder for attachments with the text 'Attach files by dragging & dropping, selecting or pasting them.' and a file attachment icon.

El resultado será:

The screenshot shows a GitHub repository page for 'ToDoList 1.0.1'. At the top, there are navigation links for Code, Issues (0), Pull requests (0), Projects (1), Wiki, and a shield icon. Below these, there are tabs for Releases (selected) and Tags. A sidebar on the left indicates it's the 'Latest release'. The main content area displays the release title 'ToDoList 1.0.1' and the author 'domingogallardo2' who released it 1 minute ago. A note says 'Añadida página Acerca de'. Below this, there's a section for 'Assets' with two items: 'Source code (zip)' and 'Source code (tar.gz)'.

Puesta en producción

Debes por último poner en producción la nueva versión, igual que hicimos en la práctica 1, creando una imagen Docker, subiéndola a Docker Hub y poniéndola en ejecución en el servidor de la asignatura.

Para crear la imagen Docker:

```
$ docker build -t <usuario-docker>/mads-todolist .
```

Sube la máquina a Docker Hub (automáticamente se etiquetara como `latest`). Y etiqueta la máquina docker con la versión `1.0.1` y súbela también.

```
$ docker push <usuario-docker>/mads-todolist
Using default tag: latest
$ docker tag <usuario-docker>/mads-todolist <usuario-docker>/mads-
todolist:1.0.1
$ docker push <usuario-docker>/mads-todolist:1.0.1
```

Conéctate al servidor de la asignatura, descarga en él la máquina Docker y pon en producción la aplicación. Comprueba que todo funciona correctamente y después para y borra el contenedor y la imagen.

En clase de prácticas deberás hacer lo mismo y el profesor revisará que la aplicación en producción funciona correctamente.

Resto de la práctica (versión 1.1.0)

El resto de la práctica consistirá en desarrollar la versión 1.1.0, usando la misma metodología vista anteriormente.

Deberás desarrollar tres características nuevas obligatorias y 3 opcionales:

- (Obligatoria) Barra de menú
- (Obligatoria) Página de listado de usuarios
- (Obligatoria) Página de descripción de un usuario
- (Opcional) Usuario administrador
- (Opcional) Protección listado y descripción de usuario
- (Opcional) Administrador puede bloquear el acceso a usuarios

Deberás implementar cada característica siguiendo la metodología que hemos usado anteriormente. En la implementación, deberás añadir el código necesario en cada una de las capas de la aplicación:

- Capa de presentación (vista)
- Nuevo método en la capa de controller
- Métodos necesarios en la capa de servicio y de repository

En cada característica deberás también incluir **tests** que prueben los nuevos métodos añadidos en la capa de servicio, así como los nuevos controllers y vistas añadidos.

Barra de menú

- La aplicación deberá tener una barra de menú común a todas sus páginas, menos en las páginas de login y registro.
- La barra de menú estará situada en la parte superior de la página y será un **Navbar** de Bootstrap.
- La barra de menú tendrá como mínimo los siguientes elementos (de izquierda a derecha):
 - `ToDoList`: enlace a la página *acerca de*.
 - `Tareas`: enlace a la página de tareas, con la lista de tareas pendientes del usuario.
- *Nombre usuario*: A la derecha de la página. Desplegable con las opciones:
 - `Cuenta`: Futura página para gestionar la cuenta
 - `Cerrar sesión <nombre usuario>`: cierra la sesión y lleva a la página de login.
- En la página *acerca de* se debe cambiar la barra de menú dependiendo de si el usuario está o no logeado. Si está logeado será la barra común con el resto de las páginas. Si el usuario no está logeado, aparecerán enlaces a las páginas de login y registro.

Listado de usuarios

- Si se introduce la URL `/registrados` aparecerá un listado de los usuarios registrados (identificador y correo electrónico).

Descripción de usuario

- En la lista de usuarios habrá un enlace para acceder a su descripción.
- En la descripción de un usuario aparecerán todos sus datos, menos la contraseña.
- La ruta para obtener la descripción de un usuario registrado será `/registrados/:id`.

Usuario administrador (opcional)

Al realizar el registro será posible darse de alta como usuario administrador.

- Para darse de alta como administrador se deberá activar un *check box* en la página de registro.
- Sólo puede haber un administrador. Si ya existe un administrador, no debe aparecer el *check box* en la página de registro.
- El usuario administrador accederá directamente a la lista de usuarios.

Protección de listado de usuario y descripción de usuario (opcional)

- Proteger las páginas con el listado de usuarios y la descripción de usuario para que sólo las pueda consultar el administrador. En el caso en que un usuario no administrador intente acceder a esas páginas, devolver un código de error HTTP "No autorizado" y un mensaje indicando que no se tiene suficiente permiso (de forma similar a como se gestionan los accesos a las páginas de tareas sin estar logeado).

Bloqueo de usuarios por usuario administrador (opcional)

- Añadir en el listado de usuarios un botón para que el administrador pueda bloquear o habilitar el acceso a cada uno de los usuarios.
- Si el usuario tiene bloqueado el acceso cuando intente logearse aparecerá un mensaje de error indicándoselo.

7. Documentación, entrega y evaluación

Deberás añadir una página documentación `/doc/practica2.md` en la que debes realizar una breve **documentación técnica** de entre 500 y 800 palabras.

Debes suponer que estás trabajando con un equipo de desarrollo y que debes dejar una breve documentación para que el resto del equipo sepa cómo ha evolucionado la

implementación de la aplicación. **No debe ser una manual de usuario, no es una documentación para el cliente.**

Por ejemplo, la documentación podría contener:

- Listado de nuevas clases y métodos implementados.
- Listado de plantillas thymeleaf añadidas.
- Explicación de los tests implementados.
- Explicación de código fuente relevante de las nuevas funcionalidades implementadas.

Obligatoriamente debes incluir en la documentación algún ejemplo de código fuente que has añadido y que consideres interesante y su explicación.

Deberás escribir esta documentación en Markdown. Tienes disponible en GitHub una breve pero útil [introducción a Markdown](#).

- La práctica tiene una duración de 4 semanas y debe estar terminada el martes 18 de octubre. El profesor comprobará en clase de prácticas el funcionamiento de la práctica en producción.
- La parte obligatoria puntuá sobre 6 y la opcional sobre 4 puntos.
- La calificación de la práctica tiene un peso de un 25% en la nota final de prácticas.
- Para realizar la entrega se debe subir a Moodle un ZIP que contenga todo el proyecto, incluyendo el directorio `.git` que contiene la historia Git. Para ello comprime tu directorio local del proyecto **después de haber hecho un `./mvnw clean`** para eliminar el directorio `target` que contiene los binarios compilados. Debes dejar también en Moodle la URL del repositorio en GitHub.

Para la evaluación se tendrá en cuenta:

- Desarrollo continuo (los *commits* deben realizarse a lo largo de las 4 semanas y no dejar todo para la última semana).
- Correcto desarrollo de la metodología.
- Diseño e implementación del código y de los tests de las características desarrolladas. Correcto funcionamiento.
- Documentación.