Crear un método “delete” que recibe un id.

PLANTEAMIENTO 1

Planteamiento

En primer lugar he ido a la clase test “UserServiceTest” y allí he añadido un test así:

```
@Test
public void should_delete_user() {
    //Given
    User userToDelete = new User("4567889098", "Maria", "Lopez Obrador", "01/01/1999", "maria@gmail.com");
    UserService userService = new UserService(mockUserRepository);
    //When
    userService.delete();
    //Then
    Mockito.when(mockUserRepository.findById(userToDelete.getId())).thenReturn(Optional.empty());
}
```

Creo que debemos cerciorarnos de que después de haber borrado si hacemos “findById” nos devuelva un “empty”

Bien, a partir de aquí, creamos el método “delete” en “UserService” el cual sólo va a localizar el usuario en el repositorio y llamará a otro método “delete” que crearemos en “User”.

```
public void delete(String id) {
    User userToDelete = userRepository.findById(id).get();
    userToDelete.delete(userToDelete);
    userRepository.delete(userToDelete);
}
```

Obviamente, ahora deberemos crear el método “delete” en “User”.

```
public void delete(User userToDelete) {
    userToDelete.setName("");
    userToDelete.setLastName("");
    userToDelete.setBirthDate("");
    userToDelete.setEmail("");
}
```

Y luego debemos crear el método “delete” en “UserRepository”:

```
void delete(User user);
```

Y luego debemos implementarlo en “PostgreUserRepository”:

```
public void delete(User user){
    userJPARepository.delete(UserEntity.fromUser(user));
}
```

PLANTEAMIENTO 2

En lugar de que la clase “User” seteé los campos del objeto manualmente dejándolos a cadena vacía, podríamos eliminar esto y directamente en la clase “UserService” se rescata el usuario a eliminar con el método “findById” y luego sea el propio repository quien lo elimina con su método delete pasándole el usuario rescatdo previamente.

Por tanto comento el método de la clase “User”:

```
/*public void delete(User userToDelete) {
    userToDelete.setName("");
    userToDelete.setLastName("");
    userToDelete.setBirthDate("");
    userToDelete.setEmail("");
}*/
```

Y luego modifico el método “delete” en la clase “UserService” comentando la línea que invocaba al método delete de User (que ya hemos comentado):

```
public void delete(String id) {
    User userToDelete = userRepository.findById(id).get();
    //userToDelete.delete(userToDelete);
    userRepository.delete(userToDelete);
}
```

CONTINUACIÓN

Por otra parte, ahora debemos volver a tocar el test, debemos validar varias cosas:

- Que el metodo delete realmente borra, es decir después de usarlo si volvemos a buscar debe salir vacío
- Que antes de usar el método delete el usuario es encontrado.

Por tanto, primero vamos a modificar el test:

```java
@Test
public void should_delete_user() {
    //Given
    User userToDelete = new User("4567889098", "Maria", "Lopez Obrador", "01/01/1999", "maria@gmail.com");
    UserService userService = new UserService(mockUserRepository);
    Mockito.when(mockUserRepository.findById(userToDelete.getId())).thenReturn(Optional.of(userToDelete));
    //When
    userService.delete(userToDelete.getId());
    //Then
    Mockito.verify(mockUserRepository).delete(userToDelete);
    Mockito.when(mockUserRepository.findById(userToDelete.getId())).thenReturn(Optional.empty());
    Assertions.assertThat(mockUserRepository.findById(userToDelete.getId())).isEmpty();
}
```

Obviamente lo anterior nos lleva a preguntarnos si en el método delete de UserService al invocar al método "findById" resulta que estamos intentando borrar un usuario que no existe debería lanzarnos una excepción e informarnos de ello y obviamente deberíamos crear un test para este caso.

```java
@Test
public void should_throw_an_exception_when_delete_a_non_exist_user() {
    //Given
    User userToDelete = new User("3456789", "Jose", "Tomas Barreto", "15/10/2000", "josebarreto@gmail.comm");
    UserService userService = new UserService(mockUserRepository);
    Mockito.when(mockUserRepository.findById(userToDelete.getId())).thenReturn(Optional.empty());
    //When //Then
    Assertions.assertThatThrownBy(() -> userService.delete(userToDelete.getId()))
            .isInstanceOf(UserNotFoundException.class)
            .hasMessage("User not found");
}
```

Obviamente al principio nos sale un error puesto que no hay ninguna excepción que lance el método "delete" de UserService y obviamente la clase "UserNotFoundException" no existe. Por tanto, la creamos e incluimos la excepción el el metodo delete de UserService:

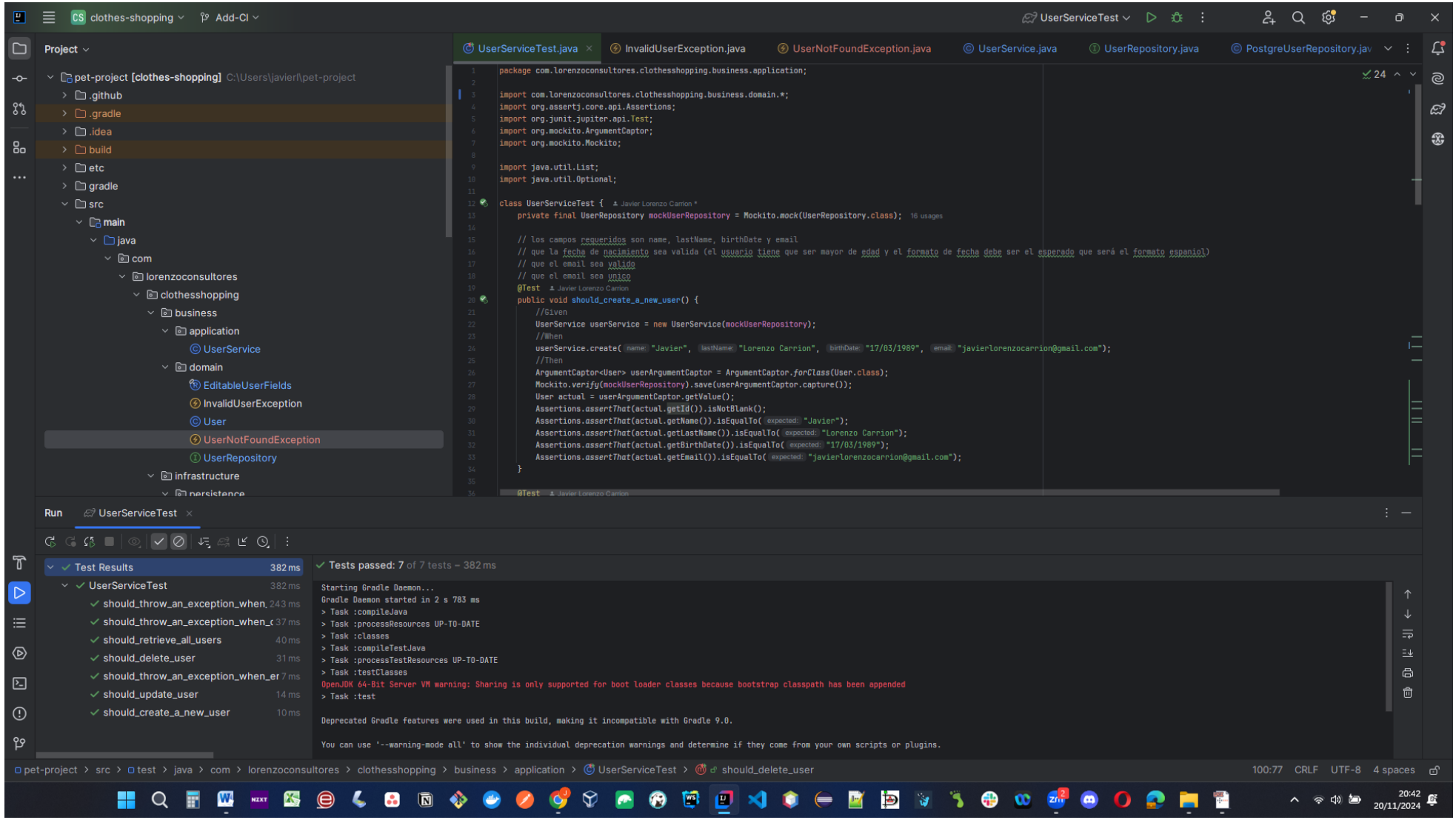
```java
public class UserNotFoundException extends RuntimeException{
    public UserNotFoundException (String message){
        super(message);
    }
}
```

Ahora, vamos a la clase UserService y ponemos la excepción para el método delete:

```java
public void delete(String id) {
    User userToDelete = userRepository.findById(id).orElseThrow(() -> new UserNotFoundException("User not found"));
    //userToDelete.delete(userToDelete);
    userRepository.delete(userToDelete);
}
```

Ahora volvemos a pasar todos los tests y rezamos a la virgen.

Y PASA TODO EN VERDE COJONES, OLE MI POLLAAAA !!!!!!!!!!!!!!!!!!!!!!!!



Bueno, ahora nos toca crear un metodo getUser que devuelva un usuario. Debe recibir un id.

PLANTEAMIENTO 1

Bueno, lo primero es ir a los test. En el servicio (UserService) debe haber un método llamado "getUser" que o bien reciba un "id" o bien reciba un DTO en cuyo interior esté el id. No obstante, como esta vez es solo obtener el usuario y devolverlo y solo precisa el parámetro id (al igual que pasó con el método delete), creo que lo más sensato es no crear ese DTO (pero obviamente no tengo ni puta idea como buen gitano culo que soy).

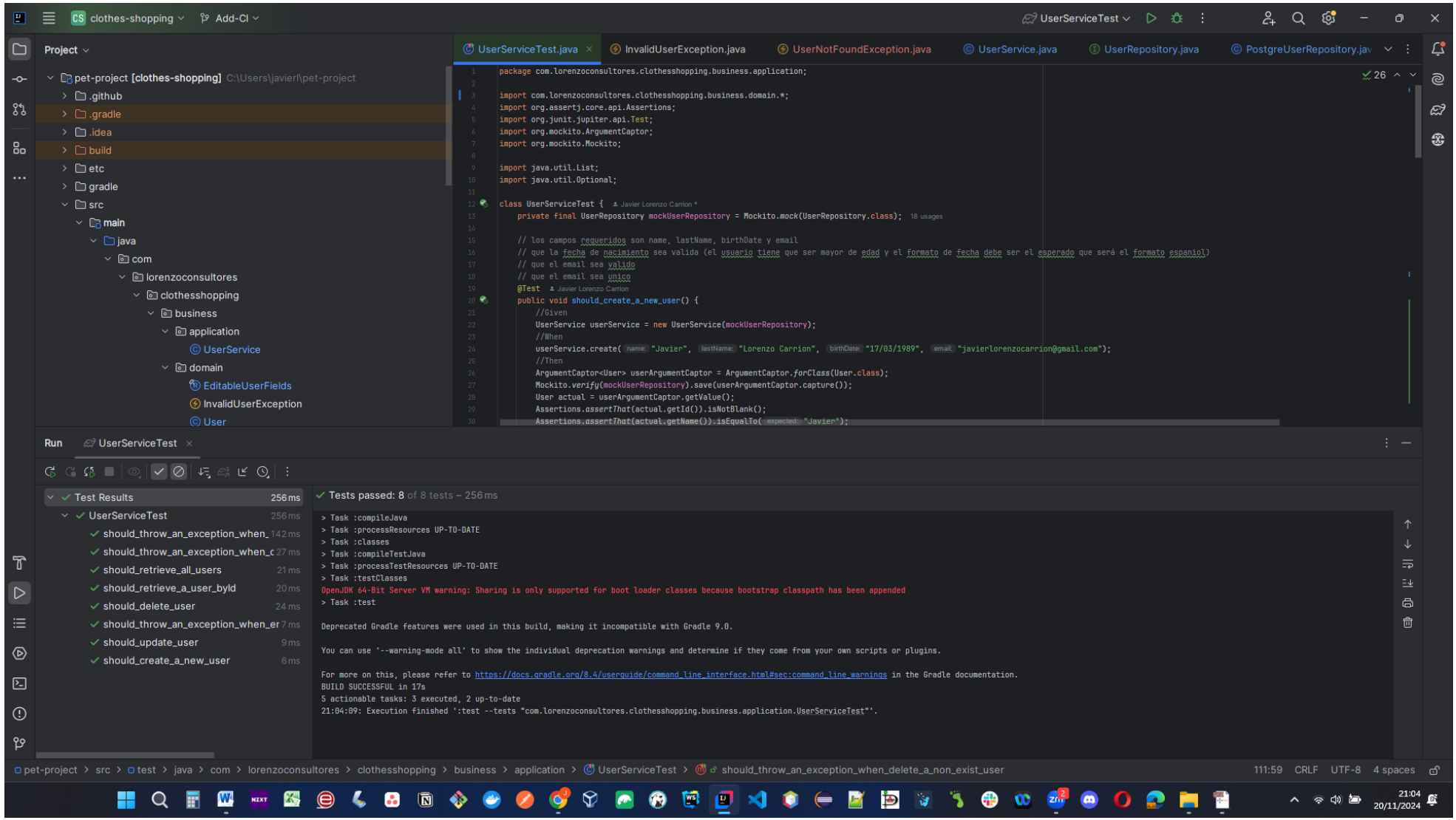
```
@Test
public void should_retrieve_a_user_byId() {
    //Given
    User userToGet = new User("4356789", "Manuel", "Perez Chacon", "10/07/1997", "manuelchacon@gmail.com");
    UserService userService = new UserService(mockUserRepository);
    Mockito.when(mockUserRepository.findById(userToGet.getId())).thenReturn(Optional.of(userToGet));
    //When
    User userGetted = userService.getUser(userToGet.getId());
    //Then
    Assertions.assertThat(userGetted).isEqualTo(userToGet);
    Assertions.assertThat(userGetted.getId()).isEqualTo(userToGet.getId());
    Assertions.assertThat(userGetted.getName()).isEqualTo(userToGet.getName());
    Assertions.assertThat(userGetted.getLastName()).isEqualTo(userToGet.getLastName());
    Assertions.assertThat(userGetted.getBirthDate()).isEqualTo(userToGet.getBirthDate());
    Assertions.assertThat(userGetted.getEmail()).isEqualTo(userToGet.getEmail());
}
```

Obviamente ahora mismo da error pues ese método "getUser" no existe. Así que vamos a UserService a crearlo.

```
public User getUser(String id) {
    User userGetted = userRepository.findById(id).get();
    return userGetted;
}
```

Ahora pasamos los test y volvemos a rezar.

Bueno, parece que ha pasado correctamente.



Al igual que antes nos pasaba con el método “delete”, podría darse el caso de que el método getUser de UserService no devuelva nada pues hemos puesto un id inexistente, por tanto primero hacemos el test:

```java
@Test
public void should_throw_an_exception_when_user_is_not_found() {
    //Given
    User userToGet = new User("435678", "Benito", "Perdomo Perez", "02/02/1975", "benito@gmail.com");
    UserService userService = new UserService(mockUserRepository);
    Mockito.when(mockUserRepository.findById(userToGet.getId())).thenReturn(Optional.empty());
    //When //Then
    Assertions.assertThatThrownBy(() -> userService.getUser(userToGet.getId()))
            .isInstanceOf(UserNotFoundException.class)
            .hasMessage("User not found");
}
```

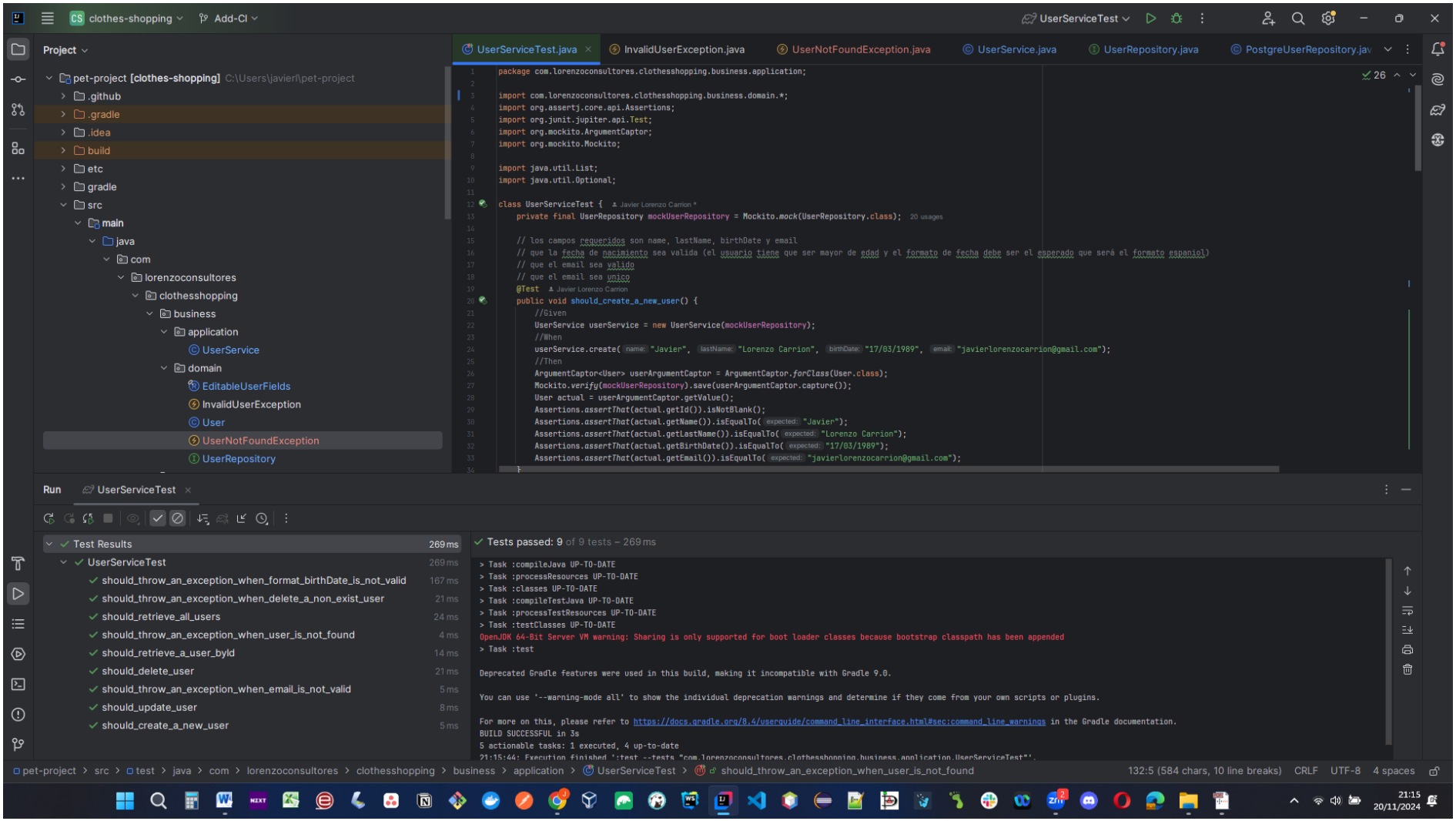
Quizá estoy medio tronado de la cabeza ya pero estoy pensando que se repite mucho código entre el método “delete” y el metodo “getUser”…

En fin, obviamente el test da rojo porque aun no hemos puesto esa excepción en el metodo de UserService.

```java
public User getUser(String id) {
    User userGetted = userRepository.findById(id).orElseThrow(() -> new UserNotFoundException("User not found"));
    return userGetted;
}
```

Y ahora pasamos los test a ver que tal.

Y oleeeeeeeeeeeeeeeeee



Bueno, ahora nos toca refactorizar el método “create” para hacerlo con un DTO (al igual que hicimos en el método “update”).

PLANTEAMIENTO 1

Quizá podría usarse clase DTO ya creada “EditableUserFields” aunque el nombre parece describir algo que solo se usará para cuando queremos editar los campos de un usuario. Quizá podríamos cambiar el nombre de esta clase a “CreateOrEditableUserFields”.

Una vez hecho esto, procedemos a modificar el método “create” de “UserService”. También tenemos un método “create” en la clase “User” que recibe parámetros como el nombre, el apellido, la fecha de nacimiento y el email. Ahora en vez de que reciba estos campos, recibirá un “CreateOrEditableUserFields”.

```
package com.lorenzoconsultores.clothesshopping.business.domain;

public record CreateOrEditableUserFields(String name, String lastName, String birthDate, String email) {

}
```

```
public static User create(CreateOrEditableUserFields fields) {
    String id = UUID.randomUUID().toString();
    User user = new User(id, fields.name(), fields.lastName(), fields.birthDate(), fields.email());
    if (!user.isValidDateFormat()) {
        throw new InvalidUserException("Birth date must have a valid format like \"dd/MM/yyyy\"");
    }
    if (!user.isValidEmailFormat()) {
        throw new InvalidUserException("Email must have a valid format like \"john.doe@example.org\"");
    }
    return user;
}
```

Y ahora en “UserService” pondríamos:

```
public User create(CreateOrEditableUserFields fields) {
    userRepository.findByEmail(fields.email()).ifPresent(existingUser -> {
        throw new InvalidUserException(String.format("Email, %s, is already in use", existingUser.getEmail()));
    });
    User userToCreate = User.create(fields);
    userRepository.save(userToCreate);
    return userToCreate;
}
```

PROBLEMA

Esto nos genera un problema ya que en la clase “UserController” hay un método create en el que se invoca al método create de UserService y este recibía como parámetros los campos (pero ahora con el cambio que he hecho recibe un objeto “CreateOrEditableUserFields” con los campos contenidos en dicho objeto)… Hasta ahora funcionaba porque este método create de UserController recibe como parametro un objeto “CreateUserRequest” y este objeto tiene los mismos campos que un User, por tanto este objeto que entraba como parámetro se pasaba después al create de Userservice…

Pero ahora con mi cambio pues ya no sé que hacer. Porque si pongo aquí un objeto "CreateOrEditableUserFields" estoy rompiendo la arquitectura hexagonal ya que eso es de dominio y esto es de infraestructura…

Por tanto la cosa está así:

```
@PostMapping
public ResponseEntity<Void> create(@RequestBody CreateUserRequest createUserRequest) {
    userService.create(createUserRequest.getName(), createUserRequest.getLastName(), createUserRequest.getBirthDate(), createUserRequest.getEmail());
    return ResponseEntity.status(HttpStatus.CREATED).build();
}
```

Entonces lo único que se me ocurre sería que los parámetros del create fuera un nuevo objeto de tipo CreateOrEditableUserFields y a ese objeto se le pasen como parámetros los campos del objeto createUserRequest (pero eso es como usar el DTO que había creado para nada porque al final lo que estoy usando es el "CreateUserRequest".

Y encima eso es un objeto de dominio y esto es de infraestructura así que estaría ropiendo la infraestructura…

Pero bueno…. El mundo está lleno de cobardes y yo no quiero ser uno de ellos así que voy a hacer esta solución y ya veremos que pasa:

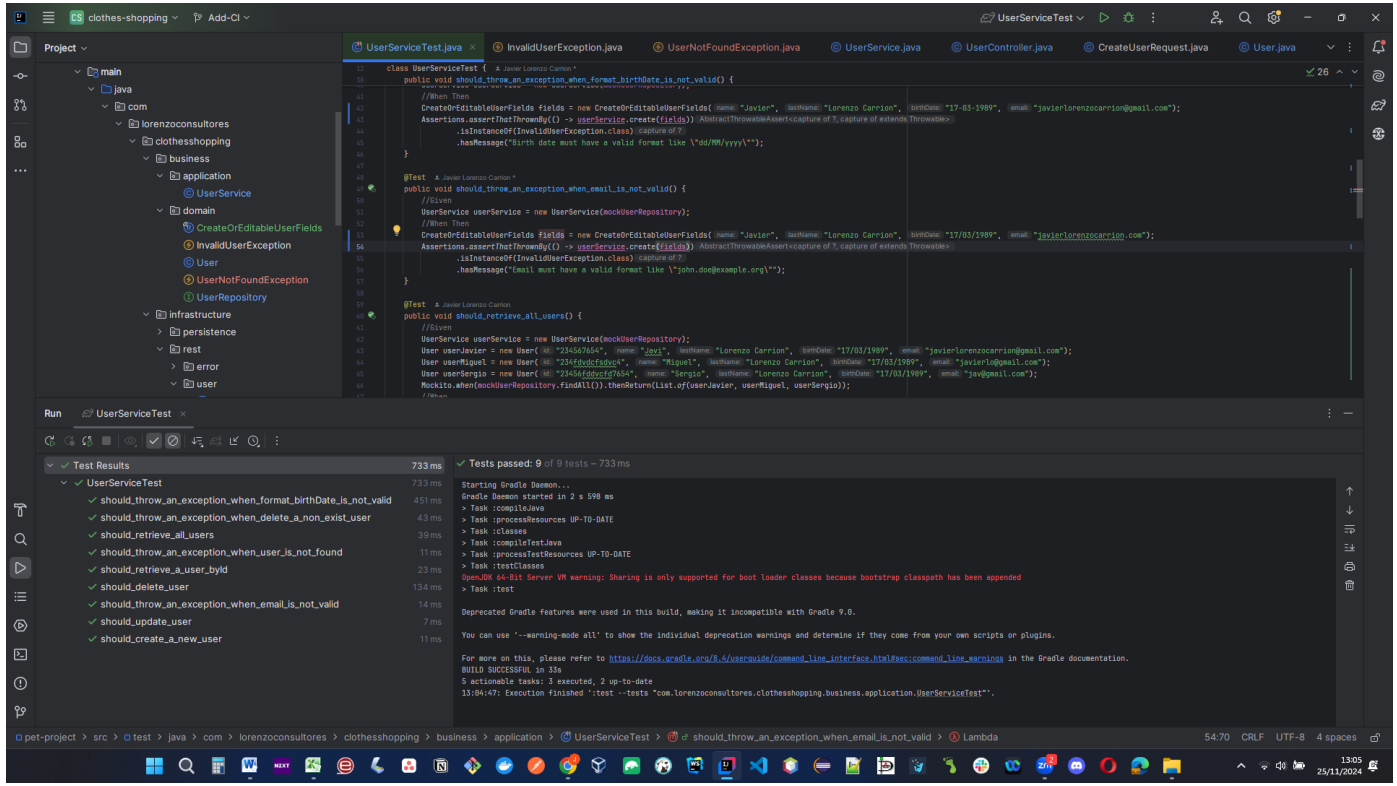
```
@PostMapping
public ResponseEntity<Void> create(@RequestBody CreateUserRequest createUserRequest) {
    userService.create(new CreateOrEditableUserFields(createUserRequest.getName(), createUserRequest.getLastName(), createUserRequest.getBirthDate(),
createUserRequest.getEmail()));
    return ResponseEntity.status(HttpStatus.CREATED).build();
}
```

Obviamente, por todos estos cambios, ahora debemos cambiar los tests también:

```
@Test
public void should_create_a_new_user() {
    //Given
    UserService userService = new UserService(mockUserRepository);
    //When
    CreateOrEditableUserFields fields = new CreateOrEditableUserFields("Javier", "Lorenzo Carrion", "17/03/1989", "javierlorenzocarrion@gmail.com");
    userService.create(fields);
    //Then
    ArgumentCaptor<User> userArgumentCaptor = ArgumentCaptor.forClass(User.class);
    Mockito.verify(mockUserRepository).save(userArgumentCaptor.capture());
    User actual = userArgumentCaptor.getValue();
    Assertions.assertThat(actual.getId()).isNotBlank();
    Assertions.assertThat(actual.getName()).isEqualTo("Javier");
    Assertions.assertThat(actual.getLastName()).isEqualTo("Lorenzo Carrion");
    Assertions.assertThat(actual.getBirthDate()).isEqualTo("17/03/1989");
    Assertions.assertThat(actual.getEmail()).isEqualTo("javierlorenzocarrion@gmail.com");
}
```

```java
@Test
public void should_throw_an_exception_when_format_birthDate_is_not_valid() {
    //Given
    UserService userService = new UserService(mockUserRepository);
    //When Then
    CreateOrEditableUserFields fields = new CreateOrEditableUserFields("Javier", "Lorenzo Carrion", "17-03-1989", "javierlorenzocarrion@gmail.com");
    Assertions.assertThatThrownBy(() -> userService.create(fields))
            .isInstanceOf(InvalidUserException.class)
            .hasMessage("Birth date must have a valid format like \"dd/MM/yyyy\"");
}
```

Bueno, pasan todos los tests en verde:



PERO LO DICHO, CREO QUE HE ROTO LOS PRINCIPIOS DE LA ARQUITECTURA AL PONER EN EL CÓDIGO DE INFRAESTRUCTURA UN OBJETO PROVENIENTE DE DOMINIO.