Validación de dos propiedades que dependen entre sí a nivel de dominio.

La validación de estas dos propiedades se ha hecho a través de la creación de una anotación personalizada que comprueba si el momento final es posterior al momento inicial de nuestra clase de Activity.

Para realizar esta anotación no hay ningún requisito previo en particular, ya que en la plantilla del proyecto ya se incluye la librería *Hibernate Validator 4.3.1.* Librería que soporta la creación de restricciones personalizadas.

```
@Constraint(validatedBy = MomentsValidator.class)
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Moments {

      String message() default "{error.Moments}";
      Class<?>[] groups() default {};
      Class<? extends Payload>[] payload() default {};
}
```

Esta sería la definición de nuestra anotación. Hay varias cosas a tener en cuenta:

- @Contraint: Esta etiqueta lo que define es qué clase va a implementar nuestra interfaz de validación
- @Target: Esta etiqueta define cuál es el alcance de nuestra restricción. Puede recibir distintos elementos de tipo ElementType, los cuales van desde campos, a métodos e incluso a tipos enteros, siendo este último el que se ha escogido, ya que vamos a validar dos campos de una clase en concreto en conjunto.
- @Retention: Esta informa al compilador dónde mantener la información de la anotación. Puede ser “RUNTIME”, “CLASS” y “SOURCE”. Se ha usado porque la evaluación de la validez de los datos se realiza en tiempo de ejecución.
- String message(): Aquí es donde se define el mensaje de error cuando la validación falla. En este caso es un mensaje de internacionalización que una vez llegado a la vista sería: “El momento inicial debe ser antes que el momento final” o su traducción al inglés.
- Class<?>[] groups(): Esta es una propiedad requerida para la comprobación en tiempo de ejecución en la que se define el tipo de dato que va a recibir el validador. Se podría haber especificado por tanto como Date.Class[], pero se ha dejado como se encontraría por defecto ya que no cambia la funcionalidad.

- Class<? Extends Payload>[] payload(): Esta es otra propiedad requerida para la comprobación en tiempo de ejecución. Según el javadoc de la intefaz, sirven para transmitir información sobre metadatos para que sean consumidos por el cliente de validación.

Una vez hecha la anotación, debemos definir una clase que la implemente:

```
public class MomentsValidator implements
ConstraintValidator<Moments, domain.Activity> {

	@Override

	public void initialize(final Moments constraintAnnotation) {

	}

	@Override

	public boolean isValid(final domain.Activity value, final
ConstraintValidatorContext context) {

		if (value.getStartMoment().after(value.getEndMoment()))

			return false;

		return true;

	}

}
```

Esta clase debe implementar la interfaz ConstraintValidator, la cual es una interfaz que requiere dos parámetros genéricos, siendo el primero la anotación que hemos definido, Moments, y el segundo el tipo o clase que se valida, en este caso domain.Activity.

Al implementar la interfaz se nos rquiere que implementemos dos métodos:

- *void initialize(A constraintAnnotation):* sirve para inicializar cualquier información interna que el validador pueda necesitar. Es requerida, pero se puede dejar vacía si no es necesaria, como en este caso.
- *boolean isValid(T value, ConstraintValidatorContext context):* Es el método que efectúa la validación, si devuelve falso nos devolverá una ConstraintViolationException con el mensaje que hemos definido previamente en la anotación, y si devuelve verdadero continuará la operación que estemos realizando.

Por último, para aplicar la restricción a nuestro dominio debemos añadir la anotación en la definición de la clase, de la siguiente forma:

```
@Moments
@Entity
@Access(AccessType.PROPERTY)
public class Activity extends DomainEntity {
..
}
```

Este sería el resultado final tras añadir una etiqueta <form:errors path="" /> en la vista cuando el validador ha encontrado el fallo en la transacción. El path debe estar vacío ya que el fallo es a nivel de objeto al completo.