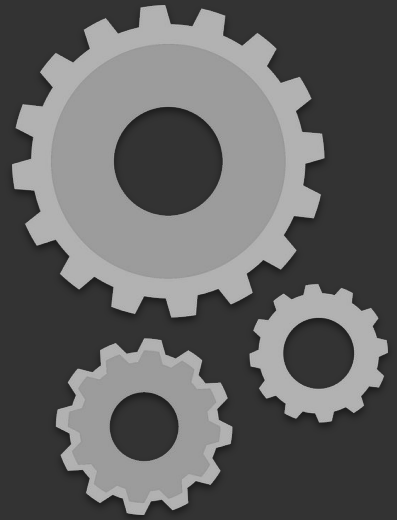
Testing

// Introducción



IT BOARDING

BOOTCAMP

Índice

IT BOARDING BOOTCAMP

TESTING

// La importancia de asegurar calidad

IT BOARDING

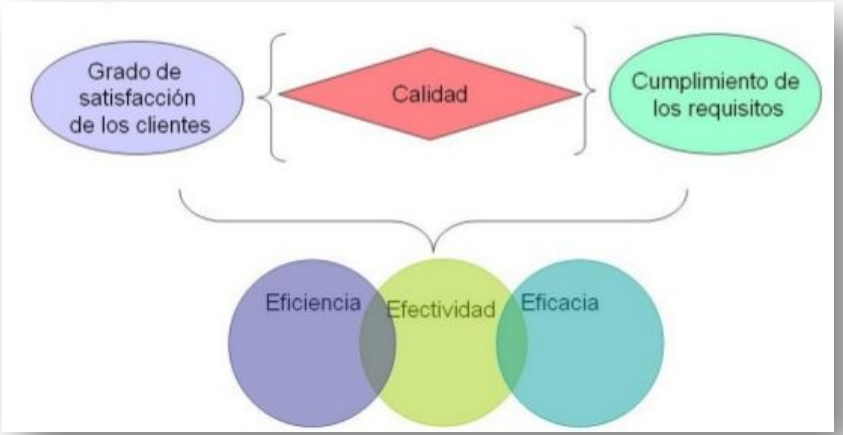
BOOTCAMP

¿Qué es calidad en un software?

Para la IEEE, la **calidad** del software es **«el grado con el que un sistema, componente o proceso cumple los requerimientos especificados y las necesidades o expectativas del cliente o usuario»**

Los requerimientos en un software pueden ser "funcionales" (***QUÉ***) o "no funcionales" (**CÓMO**).

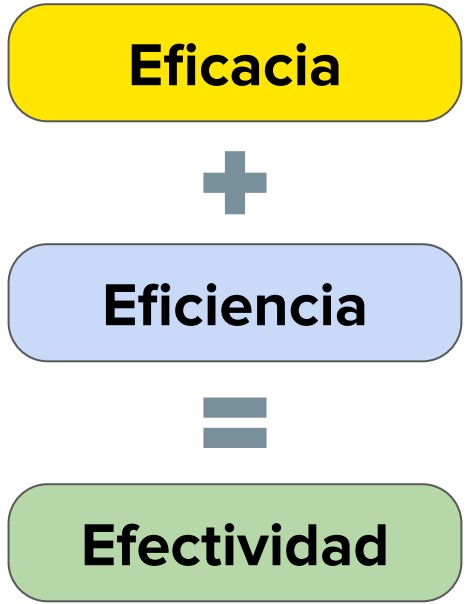
Cumplir con los requerimientos funcionales nos ayuda a saber si desde la perspectiva del cliente el software tiene buena calidad. Por otro lado, cumplir con los no funcionales nos permite comprender la calidad desde una perspectiva más ingenieril. Para poder medir la calidad de un software de una manera más holística es que agrupamos los requerimientos en **«dimensiones de calidad»**.



Que un software cumpla con su funcionalidad no siempre alcanza para decir que su calidad es buena.

Eficacia vs. Eficiencia

Mientras la **Eficacia (Qué)** busca el cumplimiento de un objetivo de la forma más directa posible, centrándose en el resultado y sin tener en cuenta los medios o recursos necesarios para lograrlo, la **Eficiencia (Cómo)**, agrega requisitos al cumplimiento del objetivo. Para que un objetivo se cumpla de forma eficiente, su resolución deberá hacer uso de la cantidad de recursos óptima.

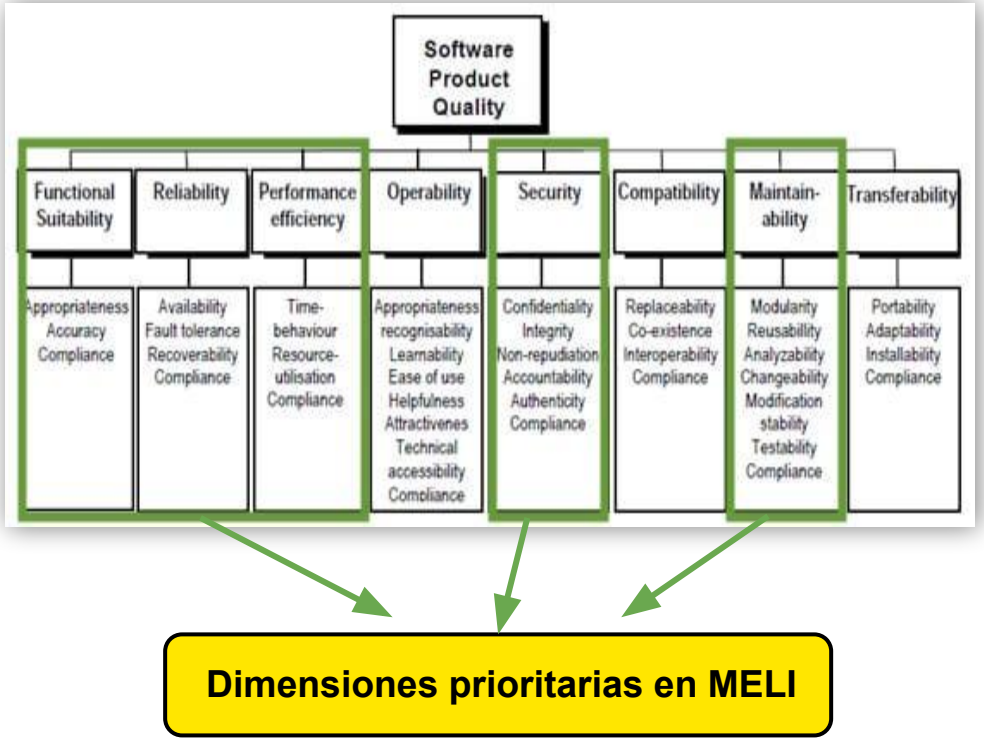


Un software que incorpora Testing va en busca de la Efectividad, es decir Eficacia + Eficiencia.

Dimensiones de calidad

En la industria existen diversos estándares que enumeran las dimensiones a considerar para evaluar la calidad de un software. Por ejemplo el ISO/IEC 25010 que define ocho dimensiones.

- **Mantenibilidad** *(Maintainability)*
- **Portabilidad** *(Portability / Transferability)*
- **Funcionalidad** *(Functionality)*
- **Performance** *(Performance)*
- **Compatibilidad** *(Compatibility)*
- **Usabilidad** *(Usability / Operability)*
- **Confiabilidad** *(Reliability)*
- **Seguridad** *(Security)*



Para que la calidad de un software sea buena no tiene obligatoriamente que cumplir con **TODAS** las dimensiones.

¿Por qué es importante construir software de calidad?

Como desarolladores de software, estamos motivados a construir software de calidad por dos principios básicos:

- **La satisfacción de nuestros usuarios**

Para que puedan realizar las acciones que necesitan de la manera que esperan. Y así impulsar el crecimiento de nuestro negocio.

- **Nuestra felicidad como desarrolladores**

Realizar cambios con la plena confianza de que nuestro software es robusto, mantenible, confiable y performante. Nos permite ser ágiles, eficientes y eficaces.

¿Quién es responsable de la calidad?

Todos nosotros, como desarrolladores de software, somos responsables por entender nuestro software, cuales son sus requerimientos funcionales y no funcionales. Cuales son sus fronteras y que dimensiones de calidad son aplicables en cada una de las fases del ciclo de vida.

Nuestra misión en MercadoLibre (MeLi) no solo es **escribir código** que cumpla su función, sino que ese código **cumpla** los **estándares de calidad** que definimos juntos como **comunidad**.

¿Cómo gestionamos la calidad?

Para gestionar la calidad, en MeLi, existen diversos **servicios provistos por Fury *(serán detallados en módulos de fury)***.

Estos servicios nos permiten **evaluar** cada una de las dimensiones de calidad que, para nosotros como empresa, son importantes a lo largo del ciclo de vida de nuestras aplicaciones, independientemente de la tecnología que usemos o el tipo de problema que estemos resolviendo y nos brindan herramientas para **mejorar** esos issues.

TESTING

// Código ~~mantenible~~ sustentable

IT BOARDING

BOOTCAMP

Mantenibilidad / Sustentabilidad

Es una dimensión que observa la **calidad** desde la perspectiva del **código fuente** del software.

Para entender el estado de esta dimensión es necesario tener en cuenta métricas como la **complejidad ciclomática** de nuestro código (cyclomatic complexity), la **cobertura de código** (code coverage), cantidad de **código duplicado**, la adhesión a **buenas prácticas** del lenguaje, etc.

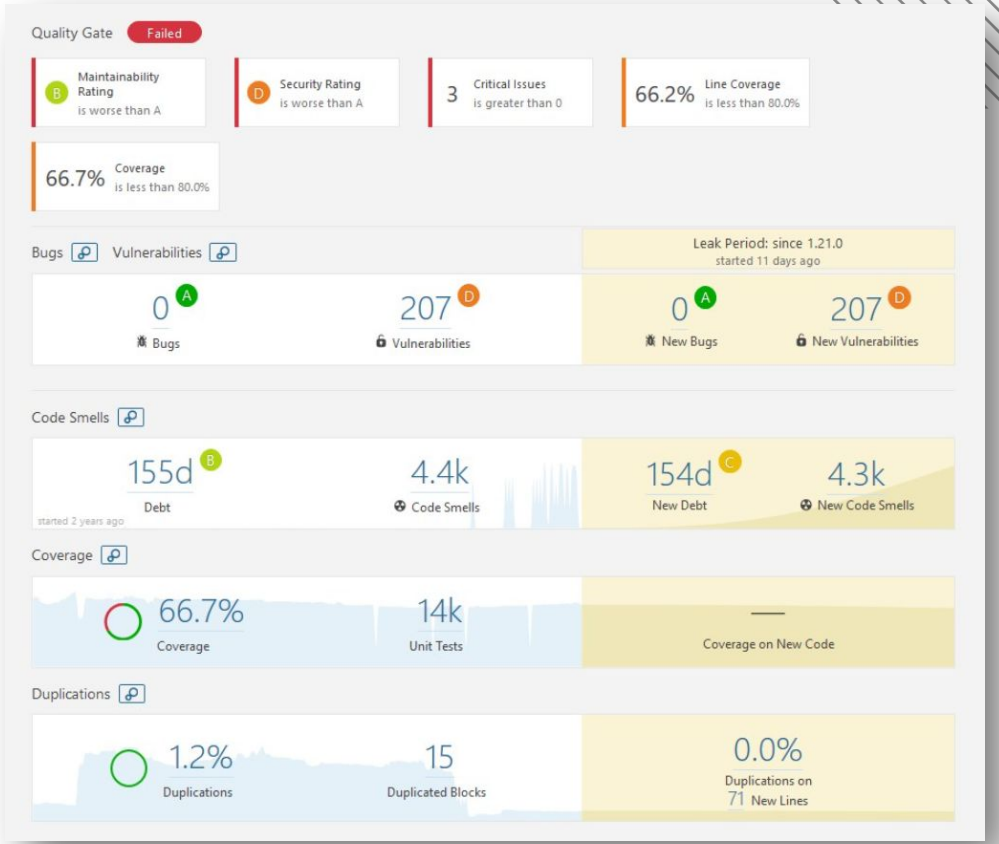
En la industria existen herramientas especializadas para ayudarnos en este análisis, una de las más conocidas es, por ejemplo, **SonarQube**.

Herramientas: SonarQube

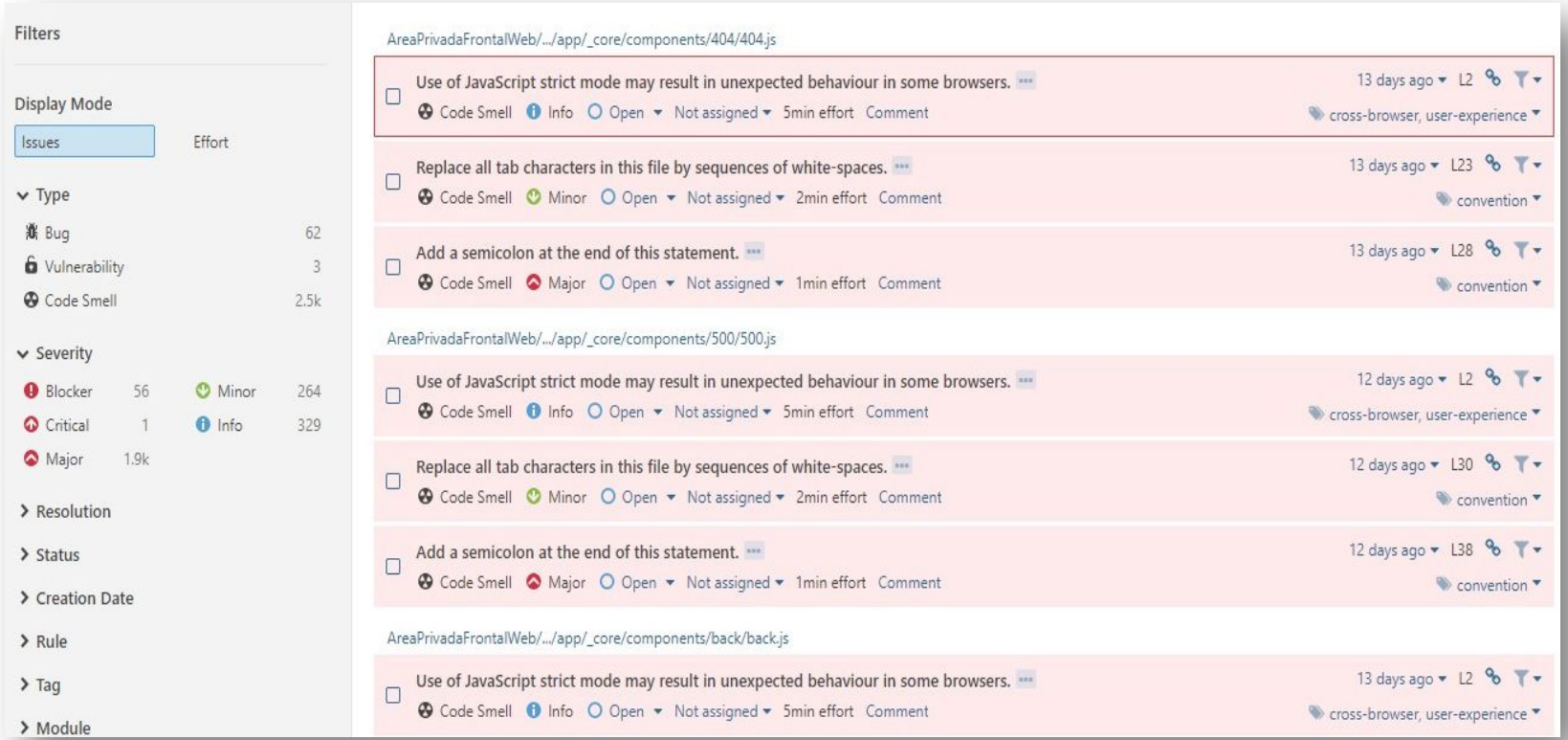
- Es una plataforma de software libre para evaluar la calidad de nuestro código, realizando un análisis estático.
- El **análisis estático** se trata de evaluar el software sin ejecutarlo.
- Obtiene métricas que ayudan a mejorar la calidad.
- El **Quality Gate** es el conjunto de condiciones que el proyecto debe cumplir antes de que pueda ser lanzado a producción.

Funcionalidades:

- Detección de código duplicado.
- Detección de código muerto, que no es utilizado.
- Estándares y buenas prácticas.
- Detección de posibles bugs.
- Análisis de complejidad ciclomática (cálculo del número de caminos independientes que tiene nuestro código).
- Comentarios.
- Coverage en test unitarios.



Dashboard de SonarQube, contabiliza bugs, vulnerabilidades, coverage, código duplicado y code smells.



La **vista de issues** permite visualizar y obtener más detalles de los mismos. También sugiere la manera de solucionarlos.

SonarQube Features: Estándar Base

Que las **reglas** estén **claras** todo el tiempo nos ayuda a lograr una mejor experiencia. Por eso para cada una de las tecnologías del servicio creamos un **estándar base** que aplica **para todos**, **sin** necesidad de **configuración extra.**

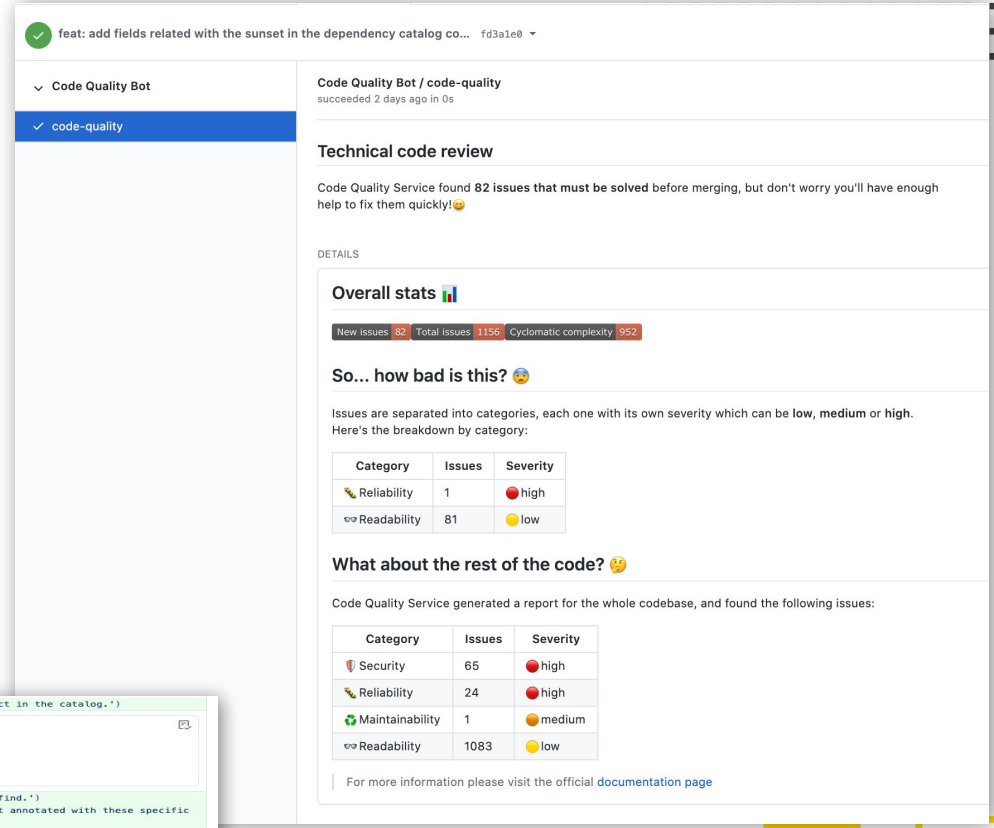
Definimos cual es el estilo de código y buenas prácticas queremos en MeLi y qué issues queremos evitar o eliminar en un lapso de tiempo.

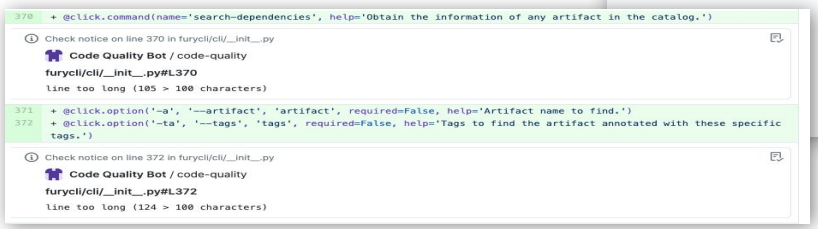
En Java podemos encontrar el siguiente standard:
https://google.github.io/styleguide/javaguide.html

El estándar base podrá extenderse haciéndose más exigente si algún equipo así lo quisiera para una/s aplicación/es.

SonarQube Features: Análisis de Pull Requests

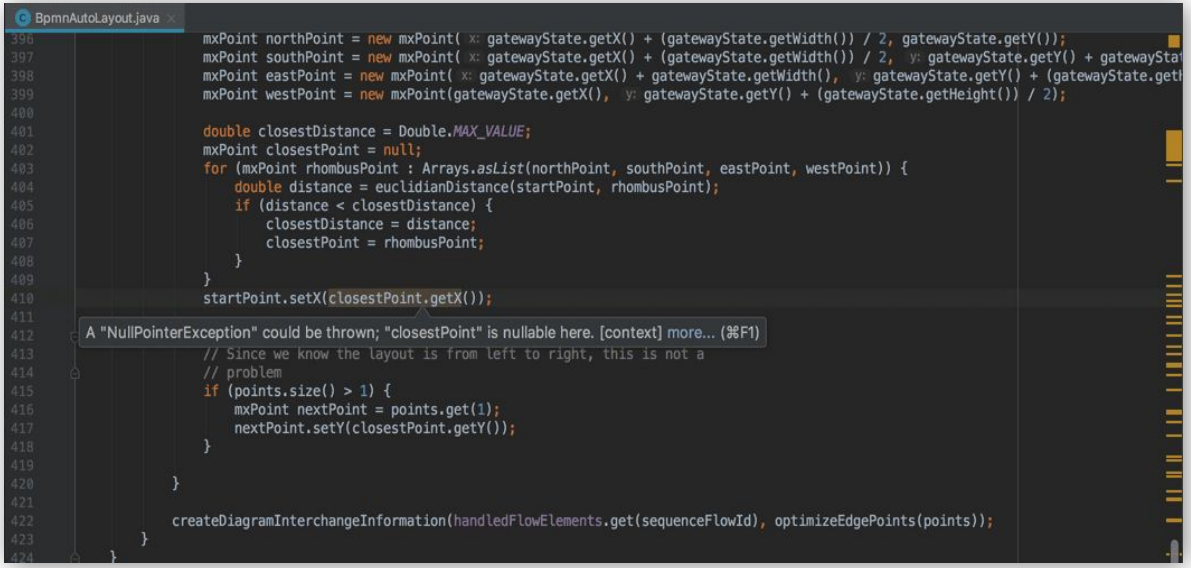
Prevenir es mejor que curar, incluso cuando hablamos de código fuente. Por eso el servicio nos provee un análisis basado en los cambios que vamos introduciendo o modificando en cada Pull Request.
De esta manera podemos **evitar** que nuestro **código** siga siendo **poco sustentable** o incluso lograr código sustentable desde el minuto uno si nuestra aplicación es nueva.





SonarQube Features: Análisis de código local (WIP)

En la misma línea con el feature de análisis de Pull Request, el servicio nos permitirá realizar el análisis de manera manual en entornos locales, maximizando así la agilidad a la hora de desarrollar libre de issues.



Sustentabilidad = Agilidad

¿Cuándo? ¿En qué casos?

- **Para entender el código** cuando cualquier miembro, tanto interno como externo, del equipo o nosotros mismos necesitemos recurrir al código para entender, revisar o proponer algún cambio.
- **Para incorporar nueva funcionalidad** cuando la urgencia del negocio requiere ser preciso y efectivo para lograr subir ese nuevo feature que va a incrementar las métricas.
- **Reparar bugs** cuando nuestros usuarios no están viviendo una buena experiencia y es necesario tomar acciones rápidas y certeras para que vuelvan a estar satisfechos.
- **Incorporar nuevos integrantes al equipo** cuando el equipo crece y es necesario explicarle a un nuevo integrante todo nuestro negocio, nuestras aplicaciones, el código y cada una de las decisiones que tomamos en él.

TESTING

// Código Seguro

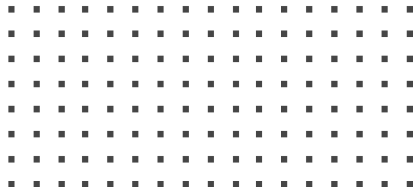
IT BOARDING

BOOTCAMP

Código seguro

El código seguro es un atributo más de la calidad del producto que construimos y como tal es una responsabilidad **de todos**.

Secure Coding como práctica de programación tiene como fin anticiparse a todos los posibles puntos de fallas que podrían ser aprovechados por un atacante.



Código seguro

- Una buena práctica de código seguro es **validar** todos los **inputs** del usuario.

- Mantener actualizado, **testeando compatibilidad**, las librerías usadas. Esto nos permite reducir fricciones al intentar actualizar versiones deprecadas

- **Evitar dependencias (librerías) con vulnerabilidades conocidas:**
  Según estudios el 70% de nuestro código es de terceras partes, en su mayoría librerías y frameworks open source.

TESTING

// Validaciones

IT BOARDING

BOOTCAMP

Validaciones

Para activar las validaciones se utilizarán las siguientes dependencias. Agregar al pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.1.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.el</artifactId>
    <version>3.0.0</version>
</dependency>
```

Validar Beans (DTOs)

- Para indicarle a Spring que valide un elemento, se utilizará la anotación **@Valid.**
- Se podrá emplear tanto desde los parámetros de un método, como dentro de un atributo, para indicar la validación de otro bean (DTO) anidado.
- Esta anotación activa la validación **TOTAL** del elemento, es decir, validará en cascada, todos los atributos del objeto señalado.

```
public class UserDto {

    //...

    @Valid
    private UserAddressDto;

    //...
}
```

```
@RestController
public class UserController {

    @PostMapping("/users")
    ResponseEntity<String> addUser(@Valid @RequestBody UserDto user) {
        return ResponseEntity.ok("User is valid");
    }

}
```

Validar Beans (DTOs)

- Para validar los tipos de datos nativos dentro de un Bean (DTO) se utilzarán una serie de anotaciones, que se podrán colocar sobre cada atriburo a validar, asi como tambíen en los parámetros recibidos o de retorno de un método.

- Algunas anotaciones aceptan diferentes atributos, pero el atributo **message** es común a todas. Representa el mensaje que va mostrar normalmente cuando la respectiva validación falle.

  (Este mensaje se podrá personalizar, más adelante en esta clase...)

Validar Beans (DTOs)

- **@NotNull** valida que el atributo no sea null.
- **@AssertTrue** valida que el atributo de tipo boolean sea true.
- **@Size** valida que el atributo tenga un valor entre **min** y **max**. Se puede aplicar a String, Collection, Map, y Array.
- **@Min** valida que el valor del atriburo sea **mayor** que lo especificado en **value**.
- **@Max** valida que el valor del atriburo sea **menor** que lo especificado en **value**.
- **@Email** valida que el atribto tenga formato válido de e-mail.

Anotaciones adicionales de JSR:

- **@NotEmpty** valida que el atributo no sea null o vacío. Se puede aplicar a String, Collection, Map o Array.
- **@NotBlank** solo se puede aplicar a valores de texto. Valida que el atributo no sea null o espacios vaciós.
- **@Positive** y **@PositiveOrZero** aplica a valores numéricos. Valida que sean estrictamente positivos (incluído el 0).
- **@Negative** y **@NegativeOrZero** aplica a valores numéricos. Valida que sean estrictamente negativos (incluído el 0).
- **@Past** y **@PastOrPresent** valida que un atributo de tipo fecha esté en el pasado o pasado incluído presente.
- **@Future** and **@FutureOrPresent** valida que un atributo de tipo fecha esté en el futuro o futuro incluído presente.

Estas validaciones también pueden aplicarse a las colecciones:

```
List<@NotBlank String> preferences;
```

Validar Beans (DTOs)

```
public class UserDto {

    @NotNull(message = "Name cannot be null")
    private String name;

    @AssertTrue
    private boolean working;

    @Size(min = 10, max = 200, message
      = "About Me must be between 10 and 200 characters")
    private String aboutMe;

    @Min(value = 18, message = "Age should not be less than 18")
    @Max(value = 150, message = "Age should not be greater than 150")
    private int age;

    @Email(message = "Email should be valid")
    private String email;

    // standard setters and getters
}
```

Validar Beans (DTOs)

Personalizar mensaje de validaciones:

Capturando las excepciones **`MethodArgumentNotValidException`** y **`HttpMessageNotReadableException`** desde el controlador base para el manejo de excepciones (@ControllerAdvice, @ExceptionHandler) se podrá manejar y personalizar los mensajes devueltos por la falla de las validaciones de los beans (DTOs).

```
@ControllerAdvice
public class CalculadoraM2ExceptionController  extends ResponseEntityExceptionHandler {

   @ExceptionHandler(MethodArgumentNotValidException.class)
   protected ResponseEntity<ErrorDTO>  handleValidationExceptions (MethodArgumentNotValid e)
{
        return new  ResponseEntity<>(e.getError() , e.getReturnHTTPStatus()) ;
   }

   @ExceptionHandler(HttpMessageNotReadableException.class)
   protected ResponseEntity<ErrorDTO>  handleValidationExceptions (HttpMessageNotReadable e)
{
        return new  ResponseEntity<>(e.getError() , e.getReturnHTTPStatus()) ;
   }
}
```

Jackson para manejo de JSON

También **fasterxml.jackson** puede sernos muy útil para la serialización de objetos java en JSON y viceversa.

- **@JsonPropertyOrder** especifica el orden de las propiedades de un objeto de manera que el json sea deserealizado en ese orden.
- **@JsonSerialize** indica qué clase será utilizada para deserealizar una propiedad determinada.
- **@JsonIgnoreProperties** sirve para enumerar a nivel de clase una lista de propiedades que Jackson va a ignorar.
- **@JsonIgnore** es usado para marcar una propiedad específica al momento de serializar.
- **@JsonInclude** permite excluir propiedades con valores vacíos, null, o por defecto.
- **@JsonFormat** permite formatear propiedades. Útil para tipos Date, Decimal, Currency..

Gracias.

IT BOARDING

BOOTCAMP