# **Lineamientos de Desarrollo ESB**

**Objetivo**

Definir un estándar de diseño y arquitectura que acople a nivel de desarrollo buenas prácticas y patrones de integración para la construcción de servicios en Camel para Jboss Fuse.

**Contenido**

[**Lineamientos de Desarrollo ESB** 1](#_Toc38536947)

[**Estructura del Proyecto** 2](#_Toc38536948)

[**Patrones de Integración y Buenas Practicas** 5](#_Toc38536949)

[Uso de Patrones de Integración 5](#_Toc38536950)

[Buenas Practicas de Escritura de Código 5](#_Toc38536951)

[**Definición de API's Rest** 8](#_Toc38536952)

[Diseño 8](#_Toc38536953)

[Implementación 9](#_Toc38536954)

[Correcto uso del códigos de respuesta HTTP 10](#_Toc38536955)

[**Escritura de Pruebas Unitarias** 14](#_Toc38536956)

[**Validación de código con SonarQube** 17](#_Toc38536957)

[**Manejo de Ramas y Versionamiento** 19](#_Toc38536958)

[**Manejo de Repositorios Archiva** 20](#_Toc38536959)

[Manejo de usuarios y roles de Archiva. 20](#_Toc38536960)

[**Procesos CI & CD** 21](#_Toc38536961)

[Herramientas para CI & CD 21](#_Toc38536962)

[Manejo de profiles 22](#_Toc38536963)

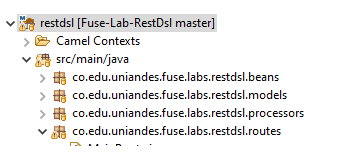
[Manejo de Pipelines en Jenkins 22](#_Toc38536964)

[Definiendo un PipieLine 24](#_Toc38536965)

## **Estructura del Proyecto**

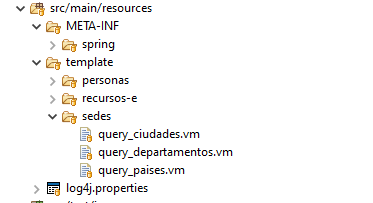
La estructura de directorios y paquetes del proyecto va acorde a la razón de ser del archivo y su contenido, el dominio por defecto para los proyectos: package **(co.edu.uniandes.fuse.{proyect})**

**Lógica de enrutamiento**

* **Beans :** Clases utilitarias que permiten hacer pequeños cambios o abstraer logica de negocio requerida para la ruta, tal como calculos matematicos o el validar especifico objeto para tomar una desicion, generalmente utiliza el exchange message de camel (header/body).
* **Models :** Clases que mapean una estructura de datos de una entidad, generalmente se usa para definir las datos de entrada y salida del servicio, tal como un recurso (persona, producto, etc), este modelo irá ligado al modelo canonico.
* **Processors :** Clases que abstraen logica de procesamiento donde se requeire un esfuerzo más alto a diferencia de un bean, para leer, validar, transformar una petición o una respuesta, generalmente utiliza el exchange message de camel (header/body).
* **Agreggators :** Clases que permiten agregar o enriquecer el contenido de un objeto o mensaje, generalmente utiliza el exchange message de camel (header/body).
* **Routes :** Clases que continen la logica del servicio y estan basadas en camel, ademas de hacer uso de los diferentes tipos de clases previamente mencionados

**Recursos del proyecto**

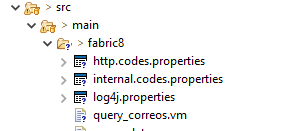
* **Camel Context** : Archivo XML que define el contexto para el arranque del proyecto en jboss fuse.
* **Plantillas (Velocity)** : Archivos que permiten definir una plantilla re-utilizable.



**Recursos fabric8**

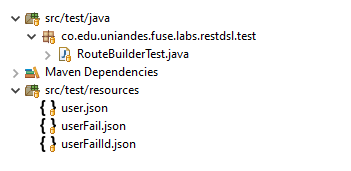
Para el despliegue de archivos que permitan modificaciones desde el contenedor se agregan en el folder /src/main/fabric

* **Archivos de Propiedades (Properties)** : Archivos donde se define una serie parámetros (llave-valor) con el fin de centralizar datos para configuraciones.

****

**Lógica y Recursos de Pruebas Unitarias**

Para el desarrollo y ejecución de pruebas unitarias, las clases java y recursos requeridos se deben alojar en el directorio src/main/test:



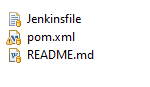
**Proyectos dependientes**

* **Fuse Parent Pom**: Este proyecto **"co.edu.uniandes.fuse.core.parent-pom"** centraliza las dependencias para la creación de API's Rest, de este modo se mantiene un marco de versiones y dependencias estandarizado y desacoplado.
* **Fuse Netty Server**: Este proyecto **"co.edu.uniandes.netty-http-proxy"** centraliza el server y puerto donde se exponen las APi.
* **Fuse Utils**: Este proyecto **"co.edu.uniandes.fuse.core.utils"** centraliza logica general de los servicios en forma de processor y beans, para que sean reutilizados y no escritos repetidamente en cada proyecto.
* **Fuse DataSource**: Este proyecto **"co.edu.uniandes.datasources"** centraliza la conexión al datasource de banner para la consulta hacia la Base de Datos.

**Archivos complementarios**

Ubicados en la raiz del proyecto

* **POM.xml** : Archivo XML con la definición del proyecto maven, dependencias & plugins.
* **README.md** : Archivo con la documentación especifica del proyecto
* **Jenkinsfile** : Archivo que contienen el Pipeline que ejecutará Jenkins o Azure Pipelines.



## **Patrones de Integración y Buenas Practicas**

### Uso de Patrones de Integración

Los EIP de Camel, permiten dar solución a un problema común de una forma estándar y probada por la comunidad. Es aconsejable aplicarlos de manera **obligatoria** dependiendo la necesidad:

Para ver la documentación completa de EIP's de Fuse Dirigirse a: [Apache Camel EIP](https://camel.apache.org/components/latest/eips/enterprise-integration-patterns.html)

### Buenas Practicas de Escritura de Código

* Declaración de clases e interfaces empleando la nomenclatura de camel-case.

public class ClassExample

{

(...)

}

* Declaración de paquetes con todas las letras en minúsculas.

co.edu.uniandes.fuse.labs.restdsl.routes

* Declaración de métodos y variables empleando la nomenclatura camel-case, con la excepción de ser la primera letra de la primera palabra en minúscula. Dichos nombres de clase deben hacer referencia a su funcionalidad.

public void mainClass() throws Exception {

(...)

}

* Declaración de constantes con todas las en mayúsculas separando cada palabra con guiones bajos, procurando dejarlas en una clase aparte.

static final int EJEMPLO\_CONSTANTE= 5;

* Hacer uso de JavaDoc para documentar brevemente los metodos y clases, parametros recibidos y retornados.

@author Nombre autor o autores

@deprecated Indica que el método o clase es obsoleto (propio de versiones anteriores) y que no se recomienda su uso.

**Descripción**

**@param** Definición de un parámetro de un método, es requerido para todos los parámetros del método. Nombre de parámetro y descripción

**@return** Informa de lo que devuelve el método, no se aplica en constructores o métodos "void". Descripción del valor de retorno

**@see** Asocia con otro método o clase. Referencia cruzada referencia (#método(); clase#método(); paquete.clase; paquete.clase#método()).

**@version** Versión del método o clase.

/\*\*

\* Returns an Image object that can then be painted on the screen.

\* The url argument must specify an absolute {@link URL}. The name

\* argument is a specifier that is relative to the url argument.

\* <p>

\* This method always returns immediately, whether or not the

\* image exists. When this applet attempts to draw the image on

\* the screen, the data will be loaded. The graphics primitives

\* that draw the image will incrementally paint on the screen.

\* @author Universidad de Los Andes - Luis Castillo / l.castillo1@uniandes.edu.co

\* @param url an absolute URL giving the base location of the image

\* @param name the location of the image, relative to the url argument

\* @return the image at the specified URL

\* @see Image

\*/

public Image getImage(URL url, String name) {

* La clases y funciones deben de ser pequeñas y hacer una sola cosa, evitar duplicar código.
* Los métodos no debes de consumir paramentos a diestra y siniestra, siempre tratar en enviar objetos que contengan los datos con los cuales va a trabajar el método.
* Uso de una plantilla como formato de codificación, todo el equipo debe tener acceso a dicha plantilla.
* Declarar variables con el menor nivel de acceso posible (solo en dentro del método que trabaja con la variable).

private String sayHello = "Hello World";

* Eliminar código innecesario, obsoleto y/o comentado por completo.
* No usar System.out.println(""); para la "creación" de logs, siempre usar un Framework adecuado para el logback.

org.slf4

* Restringir el acceso a paquetes, clases y miembros de clase cuando sea posible, empleando los modificadores de acceso private, protected, (default), public.

import co.edu.uniandes.fuse.labs.restdsl.routes.MainRoute;

(...)

public void configure() throws Exception {

(...)

}

* Evitar la aparición de nullPointerException dentro del código, devolviendo colecciones vacías o empleando excepciones cuando el resultado sea nulo (null).
* Siempre manejar las excepciones, procurando no hacer impresion de todo el printstacke, simplemente el e->message.

try {

(...)

} catch (Exception e) {

(...)

}

* No presentar en los logs información sensible.
* Liberar recursos siempre que no sean requeridos.
* No mostrar información sensible al momento que ocurra una excepción.

Para mayor información: [*Guía de Estilo de Código*](https://google.github.io/styleguide/javaguide.html)

## **Definición de API's Rest**

### Diseño

El primer paso es analizar el esquema de datos que se desea compartir por medio del API, definiendo una esquema modularizado por recursos y verbos.

**Ejemplo:**

* **Recurso**: Personas (Entidad que contiene datos específicos de una persona y esta mapeada en una serie de tablas en una BD) generalmente se define como plural
* **Verbo:** Acción que identifica el tipo de operación a trabajar sobre el recurso (Consultar [GET], Actualizar [POST], Insertar [PUT], Eliminar [DELETE])

Una vez el recurso este definido se debe armar el path o URI que se expondra definiendo el/los parametros de entrada [REQUEST]

**Ejemplo:**

host.com/personas/{id}

host.com/personas/{id}/telefonos

Una vez definida la URI del recurso se define la estructura del RESPONSE, para ello se debe tener presente de forma obligatoria el uso de los codigos de respuesta HTTP para que viaje en el Header de la respuesta :

* 100 Respuestas informativas
* 200 Respuestas satisfactorias
* 300 Redirecciones
* 400 Errores de cliente
* 500 Errores de servidor

Para mayor información sobre los códigos de respuesta HTTP: [https://developer.mozilla.org/es/docs/Web/HTTP/Status](HTTP Codes - Mozilla)

En el Body contiene el mensaje de respuesta en formato JSON:

{

"primerNombre": "JUAN",

"segundoNombre": "PABLO",

"primerApellido": "PEREZ",

"segundoApellido": "RODRIGUEZ",

"fechaNacimiento": "14/05/1990",

"genero": "MASCULINO",

"nacionalidad": "CO",

"estadoCivil": "SOLTERO"

}

En caso de que la petición genere un error adicional al codigo HTTP correspondiente es buena practica definir tambien un esquema de respuesta de error:

{

"error": {

"codigo": "0000",

"mensaje": "Ha ocirrido un error (...) "

}

Esta definición finalmente debe ser idéntica a la documentación entregada del API en Swagger.

Para Mayor información sobre el diseño de API Rest y sus buenas practicas:[Mejores Practicas en el diseño de API - RedHat](https://www.redhat.com/es/topics/api/what-is-api-design)

### Implementación

Camel por medio de componentes como (servlet, netty, restlet y spark-rest) se permite la construcción de Api's Rest tanto en JAVA DSL como en XML DSL, lo cual se conoce como (REST DSL). Esta tecnología es mucho más amigable y legible para la definición de API's Rest, ya que es orientada a recursos y permite una fácil integración con otros componentes como Swagger.

Dependencias en POM.xml (deben estar instaladas como features en el servidor jboss ya que son dependencias que se requieren en el registry):

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-netty4-http</artifactId>

<version>${camel.version}</version>

</dependency>

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-jackson</artifactId>

<version>${camel.version}</version>

</dependency>

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-http4</artifactId>

<version>${camel.version}</version>

</dependency>

Ejemplo de configuración del servidor para exponer el API Rest, junto con Swagger:

restConfiguration()

.component("netty4-http")

.bindingMode(RestBindingMode.off) //Off para evitar conflictos con los data formats

.endpointProperty("nettySharedHttpServer", "#sharedNettyHttpServer") //#sharedNettyHttpServer Como servicio osgi que comparte el server y puerto de la API

.contextPath("/{{rest.api.name}}/v{{rest.api.version}}")

Ejemplo donde se crea el recurso "example" con acciones get y post a "user"

rest("/example").description("Example rest service")

.get("/user/{type}/{document}")

.to("direct:find")

.post("/user")

.to("direct:update");

### Servidor/Puerto Netty Compartido

Se desplego como servicio OSGI en un esquema blueprint la conexión al server y puerto compartido entre Api's, por detecto se definio el puerto 8186 como el puerto centralizado para exponer las REST API. el server depende del ambiente (tutaza/topaipi/tenerife ....), el proyecto se puede descar en : [Netty-Server-GitHub](https://github.com/UniandesDSIT/Fuse-Transversal-Netty-Http)

Al arrancar el servidor el API deberia estar disponible en: <http://localhost:8186/APIName/example/users>

Para mayor información: <https://camel.apache.org/manual/latest/rest-dsl.html>

### Correcto uso del códigos de respuesta HTTP

Respuesta satisfactorias:

from("direct:find")

.bean(UserService.class, "getUser")

.marshal(jsonDFUser)

.bean(StatusGenerator.class, "statusHttpOk(\*,{{http.code.ok}})")

.log("Body: \n ${body}")

.wireTap("mock:outputGet");

Respuestas no satisfactorias:

onException(Exception.class)

.handled(true)

.choice()

.when(simple("${property[HttpErrorPropertie]} == null"))

.setProperty("HttpErrorCode").simple("{{http.code.internal.server.error}}")

.log("Http-Error-Code: ${property[HttpErrorCode]}")

.otherwise()

.setProperty("HttpErrorCode").simple(" ${properties:${property[HttpErrorPropertie]}}")

.log("Http-Custom-Error-Code: ${property[HttpErrorCode]}")

.end()

.bean(new StatusGenerator(), "statusHttpFail(\*,${property[HttpErrorCode]})")

.setBody().simple("${exception.message}")

.wireTap("mock:outputFail");

**Documentación Automática (Swagger)**

Con fin de automatizar procesos se implementa swagger de la mano con Rest Dsl para generar una documentación de las API Rest directamente basada con el código, esto asegura una documentación actualizada.

Se debe documentar en swagger toda API-Rest o servicio que exponga por medio de HTTP **recursos que permita interactuar con otras aplicaciones por medio de acciones como *(consultar / modificar / crear / eliminar)*** haciendo énfasis en la estructura de petición y respuesta, y el manejo de errores.

Dependencia requerida en el POM

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-swagger-java</artifactId>

<version>${camel.version}</version>

</dependency>

Código Ejemplo:

/\*\*

\* REST & SWAGGER CONFIGURATION

\*/

restConfiguration()

.component("netty4-http")

.bindingMode(RestBindingMode.off) //Off para evitar conflictos con los data formats

.host("{{rest.api.host}}")

.port("{{rest.api.port}}")

.contextPath("/{{rest.api.name}}/v{{rest.api.version}}")

.apiContextPath("/doc")

.apiContextRouteId("doc")

.apiProperty("api.title", "{{rest.api.description}}")

.apiProperty("api.version", "{{rest.api.version}}")

.apiProperty("cors", "{{rest.api.cors}}");

/\*\*

\* REST & SWAGGER COMPONENTSS

\*/

rest("/students").description("Example rest service")

.get("/personal-info/{type}/{document}")

.description("Find a user by type and document")

.consumes("application/json").produces("application/json")

.param().name("type").required(true).description("The type of the user").defaultValue("CC").dataType("string")

.endParam()

.param().name("document").required(true).description("The document of the user").defaultValue("1234567").dataType("integer")

.endParam()

.outType(User.class)

.to("direct:find")

Una vez el proyecto este corriendo en el servidor, se puede acceder a la documentación meadiente la URL compuesta en la configuración, swagger permite generar el contenido tanto en json como en yaml

* Ejemplo Json: <http://localhost:9090/restDSL/v1.0.0/doc/swagger.json>

{

"swagger" : "2.0",

"info" : {

"version" : "1.0.0",

"title" : "User Api Example"

},

"host" : "localhost:9090",

"basePath" : "/restDSL/v1.0.0",

"tags" : [ {

"name" : "students",

"description" : "Example rest service"

}, {

"name" : "teachers",

"description" : "Example rest service"

} ],

"schemes" : [ "http" ],

Ejemplo Yaml: <http://localhost:9090/restDSL/v1.0.0/doc/swagger.yaml>

---

swagger: "2.0"

info:

version: "1.0.0"

title: "User Api Example"

host: "localhost:9090"

basePath: "/restDSL/v1.0.0"

tags:

- name: "students"

description: "Example rest service"

- name: "teachers"

description: "Example rest service"

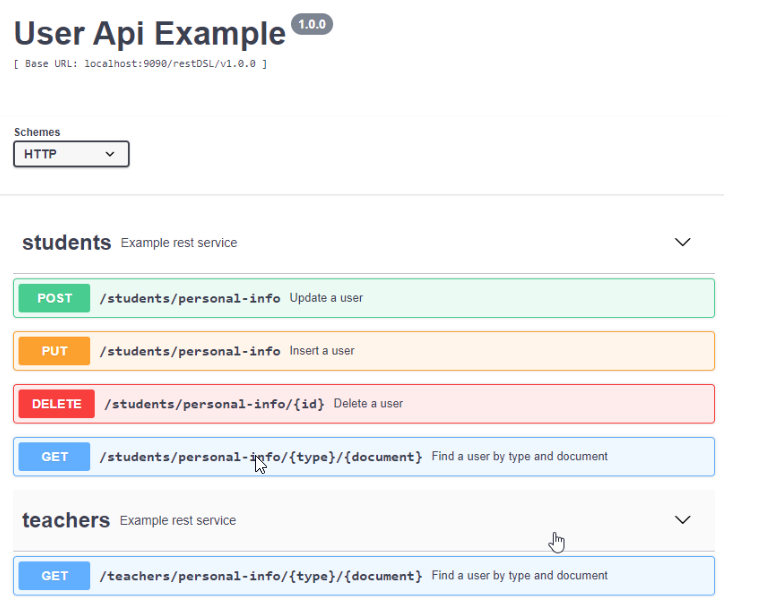
schemes:

- "http"

paths:

/students/personal-info:

Para mayor información: <https://camel.apache.org/components/latest/swagger-java.html>



## **Escritura de Pruebas Unitarias**

El objetivo de las pruebas unitarias va más allá de asegurar la calidad de las rutas expuestas. Es lograr hacer de las pruebas unitarias cultura de trabajo en el desarrollo de software. Al definir una serie de pruebas unitarias a los proyectos camel se puede empezar definir un modelo de CI & CD, ya que al automatizar las pruebas es posible generar una secuencia de trabajos (pipelines) para entornos DevOps.

Cada proyecto creado a partir de esta plantilla **deberá asegurar por lo menos un 80% de cobertura en la ejecución de pruebas en SonarQube**, esto permitirá detectar fallas tempranas previas a despliegues.

Esta cobertura depende de la cantidad de código escrito en la aplicación es decir que si se desea cubrir el 100% de cobertura la prueba unitaria deberá consumir la totalidad de funciones, control de excepciones en try/catch.

Principalmente deben estar enfocadas al core de negocio o razón de ser de la apliación con escensarios y datos reales. Para los proyectos basados en REST-DSL lo ideal es hacer consumo de cada recurso expuesto con datos reales asegurando también los escenarios de fallo en respuestas (300/400/500)

Dependencias requeridas en el POM.xml

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-test</artifactId>

<version>${camel.version}</version>

</dependency>

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-test-spring</artifactId>

<version>${camel.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

Plugins:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.14.1</version>

<configuration>

<includes>

<include>\*\*/\*Test.java</include>

</includes>

</configuration>

</plugin>

Ejemplo de Implementación:

Se utiliza inyección de dependencias por medio de anotaciones:

@Produce -> Plantilla encargada de generar el recurso que se utilizará para al prueba. @EndpointInject-> Punto donde se entra a evaluar el mock donde la prueba finaliza, a esta propiedad en el método de @Test, se define que atributos espera como respuesta, para decir si al prueba fue exitosa o no.

public class RouteBuilderTest extends CamelTestSupport {

@Produce(uri = "http4://localhost:9090/restDSL/v1.0.0/students/personal-info")

protected ProducerTemplate testProducerPost;

@EndpointInject(uri = "mock:outputPost")

protected MockEndpoint resultEndpointPost;

protected RoutesBuilder createRouteBuilder() throws Exception {

return new MainRoute();

}

@Test

public void testPostUser() throws Exception {

resultEndpointPost.expectedHeaderReceived("CamelHttpResponseCode", "200");

resultEndpointPost.setExpectedCount(1);

String contents = new String(Files.readAllBytes(Paths.get("src/test/resources/user.json")));

testProducerPost.sendBodyAndHeader(contents, "CamelHttpMethod", "POST");

resultEndpointPost.assertIsSatisfied();

}

}

Como resultado:

[main] MockEndpoint INFO Asserting: Endpoint[mock://outputPost] is satisfied

[main] RouteBuilderTest INFO \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[main] RouteBuilderTest INFO Testing done: testPostUser(co.edu.uniandes.fuse.labs.restdsl.test.RouteBuilderTest)

[main] RouteBuilderTest INFO Took: 0.008 seconds (8 millis)

[main] RouteBuilderTest INFO \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## **Validación de código con SonarQube**

El objetivo de SonarQube, es ayudarle al equipo de desarrollado a mantener un código limpio que permita detectar bugs, vulnerabilidades y posibles mejoras para asegurar un código legible y mantenible en el tiempo. Adicionalmente si el proyecto cuenta con Pruebas Unitarias, SonarQube permite analizar la covertura (alcance) y éxito de las pruebas del proyecto.

Posterior al análisis de código los aspectos a corregir inmediatamente en el proyecto son los **BUGS** y **Vulnerabilidades**, los Code Smells en la medida de lo posible es aconsejable también depurarlos pero no es la prioridad.

Para ello se requiere la siguiente configuración y dependencias en el POM.xml:

<plugin>

<groupId>org.jacoco</groupId>

<artifactId>jacoco-maven-plugin</artifactId>

<version>0.8.4</version>

<executions>

<execution>

<id>default-prepare-agent</id>

<goals>

<goal>prepare-agent</goal>

</goals>

</execution>

<execution>

<id>default-report</id>

<goals>

<goal>report</goal>

</goals>

</execution>

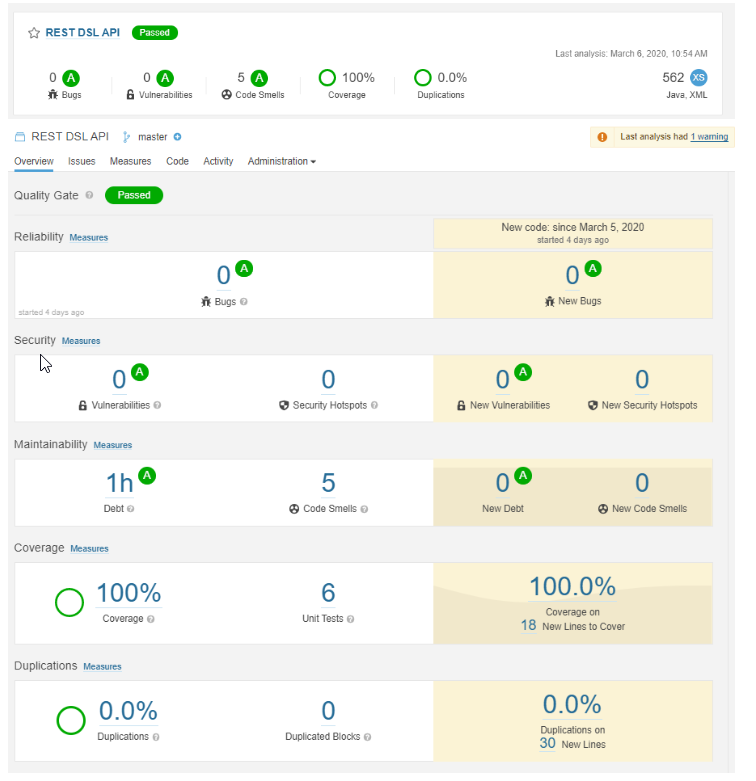
</executions>

</plugin>

Una vez el proyecto se encuentre listo para analizar, se utiliza el siguiente comando:

mvn clean install test sonar:sonar

como resultado desde el panel de SonarQube se visualiza el análisis ejecutado y las sugerencias:

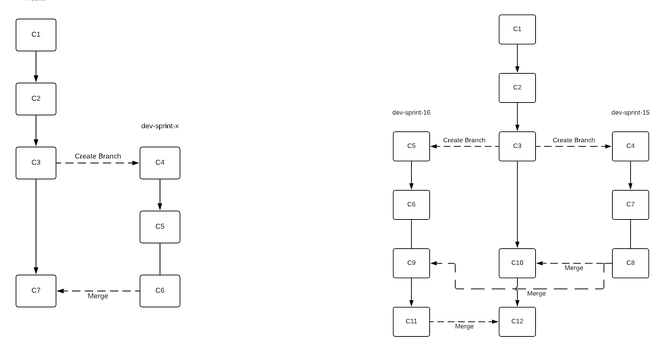


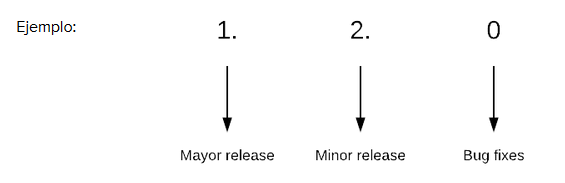
Para mayor información: <https://docs.sonarqube.org/latest/>

## **Manejo de Ramas y Versionamiento**

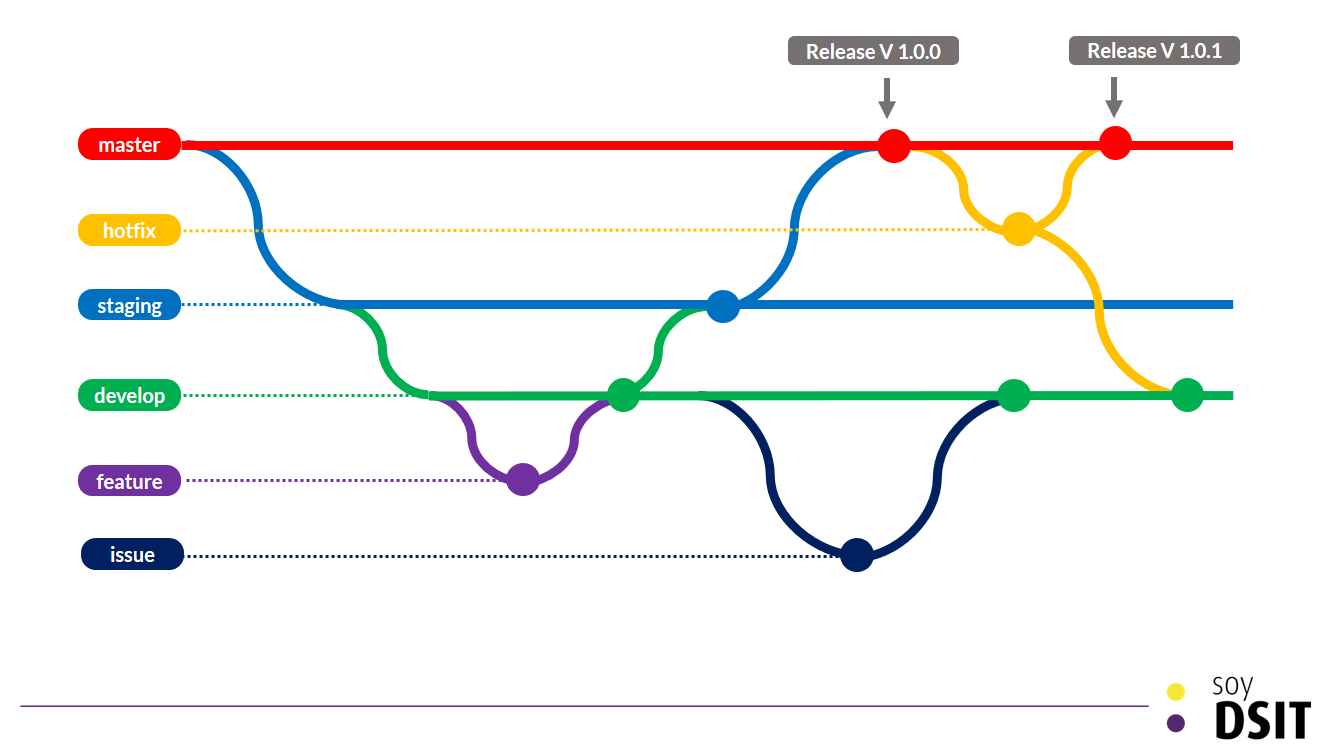
Acorde al lineamiento transversal del CedEx de Desarrollo, se trabaja con el modelo de ramas y versionado semantico definido:

Lo recomendable es manejar N ramas en git con todo el código fuente y los archivos .properties con la siguiente estructura:

* **Rama-Master**: En esta rama estaría el código de las versiones certificadas y entregadas a la universidad, segmentadas por Tags, dependiendo la version.
* **Rama-Staying**: Esta rama deberá contener la versión a certificar para paso a producción, deberá ser similar a master pero si se encuentran issues o bugs en la etapa de certificación las correcciones nacen de esta y no de master directamente.
* **Rama-Develop-Sprint-X**: Apoyados en la metodología SCRUM, en esta rama estaría el código sobre el cual se usaría para la implementación de las nuevas versiones y el manejo de una rama por cada Sprint que se esté desarrollando, se tomaría como base el código de la última versión certificada de la rama master.



Para el caso de los archivos properties, dentro de cada uno de los proyectos se tendrían 4 carpetas que contendrían los archivos .properties de cada ambiente dev, calidad y producción. Para el caso de ejecución de varios Sprint en paralelo que modifiquen algún componente en común se debe realizar un merge manual antes de que se envíe a pruebas.



## **Manejo de Repositorios Archiva**

Se debé manejar 4 repositorios en archiva (uno por ambiente) para almacenar los artefactos generados y tener un versionamiento por ambiente con los properties y el código que se va a desplegar, adicional se debería tener un repositorio de librerías de terceros para la construcción y despliegue de los artefactos.

* <http://archiva.uniandes.edu.co/archiva/esb.libs-repo/> Repositorio de librerías comunes de terceros
* <http://archiva.uniandes.edu.co/archiva/esb.dev-repo/> Repositorio de artefactos snapshot desplegados en desarrollo.
* <http://archiva.uniandes.edu.co/archiva/esb.pru-repo/> Repositorio de artefactos snapshot desplegados en pruebas.
* <http://archiva.uniandes.edu.co/archiva/esb.qa-repo/> Repositorio de artefactos snapshot desplegados en calidad.
* <http://archiva.uniandes.edu.co/archiva/esb.prod-repo/> Repositorio de artefactos release desplegados en Producción.

### Manejo de usuarios y roles de Archiva.

En el archiva deberían existir 3 usuarios principales distribuidos así:

* Admin: Usuario administrador que puede crear repositorios, usuarios y configuración en general del producto.
* Fuse: Usuario para el ESB que permite descargar artefactos de los diferentes repositorios.
* Jenkins: Usuario para el Jenkins que permite subir y descargar artefactos de los diferentes repositorios.

**Permisos:**

* Cicd: read, upload, delete sobre repositorios
* Esb: read sobre repositorios
* Admin: read, upload, delete, manage, annotate sobre repositorios y builds.

**Asociación:**

* Admin - admin
* Fuse – esb
* Jenkins – cicd

## **Procesos CI & CD**

### Herramientas para CI & CD

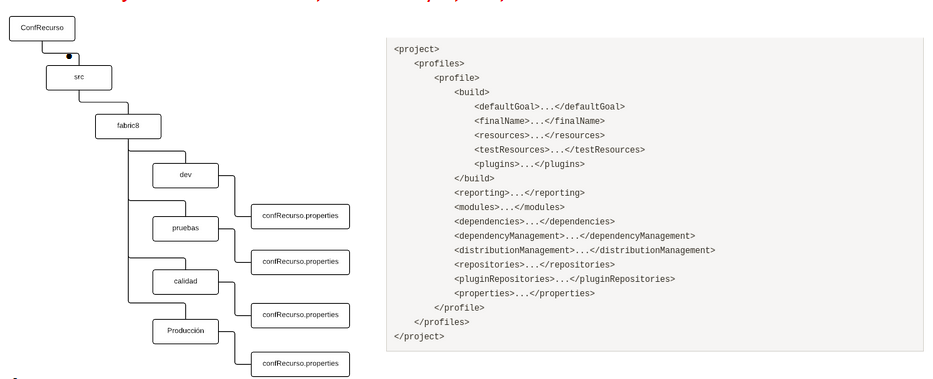
A continuación, se mencionan el conjunto de herramientas que se recomiendan para el proceso de integración continua:

* **Jenkins**: Servidor de integración continua que permite la automatización de tareas, en donde se indica las fases a cubrir en un build.
* **Git**: Software de control de versiones pensado en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente.
* **SonarQube**: Plataforma de código abierto para el análisis de la calidad de código usando herramientas de análisis estático para obtener métricas que pueden ayudar a mejorar la calidad del código de un programa.
* **Archiva**: Programa que sirve para el manejo de repositorios ofreciendo funciones de mantención como proxying de repositorios remotos, seguridad de acceso, almacenamiento de artefactos, suministro revisión, indexación, estadísticas de uso y escaneo de repositorios.
* **Jboos Fuse**: Plataforma de integraciones.

### Manejo de profiles

Dado que el código fuente y en cada rama se van a tener los archivos de cada ambiente al momento de la compilación y generación del artefacto, se debería incluir el archivo respectivo del ambiente para el cual se genera la versión. Esta funcionalidad comúnmente se maneja con los profiles de Maven, junto con las siguientes variables:

* URL de Fuse del ambiente a desplegar
* Ruta del archivo properties que se debe utilizar.



### Manejo de Pipelines en Jenkins

Se debería manejar un pipeline por cada una de las rutas, el cual tiene un Jenkinsfile que importa otro Jenkinsfile dependiendo del ambiente al cual se desea desplegar. Este valor debe ser ingresado de forma manual. En este caso cada ruta debería tener cuatro jenkinsfile (uno por cada ambiente), el cual va a tener los siguientes stages:

* **Jenkinsfile de desarrollo**: Este pipeline se debería ejecutar automáticamente con WebHook cuando se realice un push en el git, debería realizar los siguientes stages:

1. Checkout de la rama dev-sprint-x
2. Clean
3. Copiado de archivos properties de desarrollo (por medio del profile de maven)
4. Package
5. Pruebas unitarias
6. Análisis de SonarQube
7. Subir artefacto a archiva al repositorio de snapshots-dev (por medio de profile de maven)
8. Despliegue en ambiente de desarrollo (por medio de profile de maven)

* **Jenkinsfile de paso a pruebas**: Este pipeline se debería ejecutar manualmente cuando QA realice la certificación, este realiza los siguientes stages:

**Parámetros:**

1. Número de la versión que se va a crear en tag
2. Rama dev-sprint-x

**Stages:**

1. Realiza merge del Branch dev-sprint-x hacia el Branch master
2. Crear un tag de la versión en el Branch master
3. Checkout de la rama master del tag anteriormente creado
4. Clean
5. Copiado de archivos properties
6. Package
7. Pruebas unitarias
8. Análisis de SonarQube
9. Subir artefacto a archiva al repositorio de snapshots-pru
10. Despliegue en ambiente de pruebas

* Jenkinsfile de paso a calidad: Este pipeline se ejecutaría manualmente por el gestor de la configuración de la administración por parte de uniandes y realiza los siguientes stages:

**Parámetros:**

1. Número de la versión del Git a desplegar.

**Stages:**

1. Checkout de la rama master de un tag especifico
2. Clean
3. Copiado de archivos properties de calidad
4. Package
5. Pruebas unitarias
6. Análisis de SonarQube
7. Subir artefacto a archiva al repositorio de snapshots-calidad
8. Despliegue en ambiente de calidad

* **Jenkinsfile de paso a producción**: Este pipeline se ejecutaría manualmente por el gestor de la configuración de la administración por parte de uniandes y realiza los siguientes stages:

**Parámetros:**

1. Número de la versión de Git a desplegar

**Stages:**

1. Checkout de la rama master del tag específico
2. Clean
3. Copiado de archivos properties de producción
4. Package
5. Pruebas unitarias
6. Análisis de SonarQube
7. Subir artefacto a archiva al repositorio de releases
8. Despliegue en ambiente de producción

### Definiendo un PipieLine

Por medio de un Pipeline **(Jenkinsfile)** que todo proyecto camel debe tener se establecerán las tareas de compilación, análisis de código, pruebas unitarias y despliegue en archiva.

pipeline {

agent any

tools {

maven 'maven-local'

}

stages {

stage ('Compile Stage') {

steps {

sh 'mvn clean compile'

}

}

stage ('SonarQube Stage') {

steps {

sh 'mvn clean install'

sh 'mvn test sonar:sonar'

}

}

stage ('Deploy Stage') {

steps {

sh 'mvn deploy -DskipTests'

}

}

}

**Despliegue y generación de release con sus respectivos tags.**

Estos pasos están como plan piloto para una definicion completa de integración y despliegue continuo

* Creación de llaves ssh

ssh-keygen

clip {keyPath}

* Agregar llave privada al agente ssh local de git

eval $(ssh-agent -s)

ssh-add ~/.ssh/<private-key>

//Agregar a los repositorios o al perfil las llave publicas

**Comandos de despliegue**

* Sube el pom y jar a archiva

mvn deploy

* Commit de los cambios realizados en la version

git add .

git commit -m "{description}"

* Prepara al versión del release, genera el tag en git, pushea cambios, actualiza la próxima version snapshot

mvn release:prepare

* Despliega el release final en archiva

mvn release:perform -Darguments="-Dmaven.javadoc.skip=true"

* Despliega en el profile de fabric8

mvn clean -DskipTests=true fabric8:deploy -Dfabric8.jolokiaUrl=http://{user}:{password}@localhost:8181/jolokia -Dfabric8.profileVersion={fabric.container.version}