Universidad ORT Uruguay
Facultad de Ingeniería

Ruski Oil International
Arquitectura de Software - Obligatorio

Entregado como requisito para la obtención del crédito Arquitectura de Software

Rodrigo Stratta - 142630
Luis Introini -181108

Grupo N7A
Docentes: Andrés Calviño, Mathias Fonseca

2017

Abstract

Ruski Oil International (ROI) es una empresa multinacional que se dedica a la explotación, distribución y comercialización de gas natural y petróleo crudo. Como proveedor mayorista, abastece a refinerías y empresas que producen derivados a través de redes de oleoductos/gasoductos y buques tanque en todo el mundo.

Para ello ROI necesita piezas de Software que le permita llevar a cabo esta tarea, y es por ello que en el presente documento, se especifican las diferentes decisiones arquitectónicas que hacen al conjunto de sistemas.

Índice

1. Introducción

En el presente documento se detallarán las principales decisiones arquitectónica de las aplicaciones, pretendiendo dar información sobre cómo fue construido el sistema, en base a los requerimientos funcionales y no funcionales.

La documentación que se presenta a continuación sigue el modelo “Views and Beyond”, una forma de analizar, diseñar y documentar la arquitectura de sistemas descrita en el libro “Software Architecture in Practice (Third edition)”[1]

1.1.Propósito

El principal propósito de este documento es proveer una especificación de la arquitectura de las aplicaciones que utiliza *Ruski Oil International*, para llevar a cabo su negocio. Se entiende por esto que el sistema permitirá registrar las órdenes, generar los planes de suministro y enviar comandos a los actuadores para que se haga efectivo el plan.

Este documento puede ser utilizado por otros arquitectos, desarrolladores, o cualquier Stackeholders que desee obtener información y justificaciones acerca de las decisiones tomadas.

2.Antecedentes

2.1.Propósito del sistema

El propósito del sistema es el manejo de todo lo que refiere al trabajo sobre órdenes de suministro, y todos los pasos que se tienen que dar para que se haga efectiva la entrega del servicio. Para explicar de mejor manera esta secuencia de pasos, en el próximo punto se especifica el flujo normal de la utilización del sistema.

[1] https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30264

2.2. Flujo del sistema

El sistema está compuesto por aplicaciones de Front-End, las cuales no modificamos y Back-End, que se sustituyeron viejas aplicaciones con tecnologías obsoletas.

El flujo final del sistema resultó ser el siguiente:

1. Se ingresa una nueva orden de suministro, ingresando los datos necesarios para que se pueda crear un plan de suministro.
2. Se crea automáticamente un plan de suministro.
3. Se calcula automáticamente la ruta para llegar al punto de suministro.
4. Se aprueba el plan de suministro y se programa
5. Se ejecutan los comandos programados
6. Se recolectan datos sobre actuadores.

2.3. Resumen de requerimientos

A continuación se presenta el resumen de los requerimientos desarrollados en el sistema.

Id. de Requerimiento	Descripción
REQ-1	Registrar Orden de Abastecimiento: Permitir crear, modificar, eliminar y consultar Órdenes de Abastecimiento en **roi-supplying**. Una Orden de Abastecimiento contiene la siguiente información: ● Nro de orden ● Nro de cliente ● Fecha de inicio del suministro ● Volumen contratado ● Identificador de punto de servicio (nro. del contador donde se conecta el cliente) ● Día de cierre para facturación (por ej. 25 de cada mes)
REQ-2	Comunicar las Órdenes de Abastecimiento a roi-planner: Cada vez que se crea, modifica o elimina una Orden de Abastecimiento en roi-supplying se debe comunicar inmediatamente a roi-planner indicando la operación realizada.
REQ-3	Registrar Plan de Suministro: Al recibir una Orden de Abastecimiento desde roi-supplying, se procede a registrar el

	Plan de Suministro correspondiente según la operación indicada: ● Si la operación informada con la Orden es “creación”, se debe crear un nuevo Plan de Suministro. ● Si la operación informada con la Orden es “modificación”, se deben reflejar los cambios que correspondan en el Plan de Suministro. ● Si la operación informada con la Orden es “eliminación”: se debe marcar el Plan de Suministro como “anulado”. La creación del Plan de Suministro implica además conectarse a la API de **roi-pipeline-calc** para obtener la ruta del suministro (tramos de red). Un Plan de Suministro contiene la siguiente información: ● Nro identificador de Plan ● Lista de “tramos de red”. Cada tramo contiene: ○ Identificador de punto de suministro (parte física de la red por la que sale el gas, por ejemplo una válvula) ○ Identificador de actuador (que permite accionar sobre el punto de suministro) ○ Identificador de tramo (piense en el nro que identifica el tramo de caño que lleva el gas hasta el siguiente punto de suministro)
REQ-4	Habilitar un nuevo actuador para enviarle comandos: Para un nuevo actuador, facilitar una forma de registrar el controlador correspondiente en Goliath de forma que se le puedan enviar comandos desde roi-planner tratando de afectar con los cambios la menor cantidad total de elementos, y si fuera posible sin que haya necesidad de modificar el código fuente de roi-planner.
REQ-6	Gestión de errores y fallas: El sistema debe proveer suficiente información, de alguna forma, que permita conocer el detalle de las tareas que se realizan. En particular, en el caso de ocurrir una falla o cualquier tipo de error, es imprescindible que el sistema provea toda la información necesaria que permita a los administradores hacer un diagnóstico rápido y preciso sobre las causas. Se espera que la solución contemple la posibilidad de poder cambiar las herramientas o librerías concretas que se utilicen para producir esta información, así como reutilizar esta solución en otras aplicaciones, con el menor impacto posible en el código.

REQ-7	Autenticación de operaciones: Actualmente, cualquier usuario con acceso a la red informática donde están conectados los servidores de aplicaciones puede invocar libremente cualquier operación expuesta por las API de backend de cualquier aplicación (imagine un usuario ejecutando llamadas http usando Postman o cualquier otra herramienta). Se requiere que todas las invocaciones a cualquier operación de backend en cualquiera de las aplicaciones que vaya a construir sean debidamente autenticadas, es decir, que sea una invocación legítima para la aplicación que la recibe.

2.4. Atributos de calidad

En base a los requerimientos anteriormente planteados se realiza una tabla con los atributos de calidad que se infirieron de las diferentes interacciones que se han tenido con el cliente y del conocimiento adquirido del negocio.

Id. de requerimiento	Atributo de calidad
REQ-4	Modificabilidad
REQ-6	Modificabilidad Disponibilidad Seguridad
REQ-7	Seguridad

2.5. Restricciones

- La implementación del backend debe desarrollarse en JEE utilizando las tecnologías vistas en el curso (web services, EJB, JMS, JPA). De ser necesario, podrán utilizarse otras tecnologías Java (logging, mail, formatos de datos -xml, json-, etc.)

- La API de **roi-pipeline-calc** está publicada en la siguiente URL: http://docs.pipelinecalculatorapi.apiary.io/#

- Las API que exponen los backend de **roi-supplying** y **roi-planner** a sus respectivos front-end **debe ser REST**. Para las comunicaciones entre aplicaciones de backend tiene libertad de seleccionar el mecanismo.

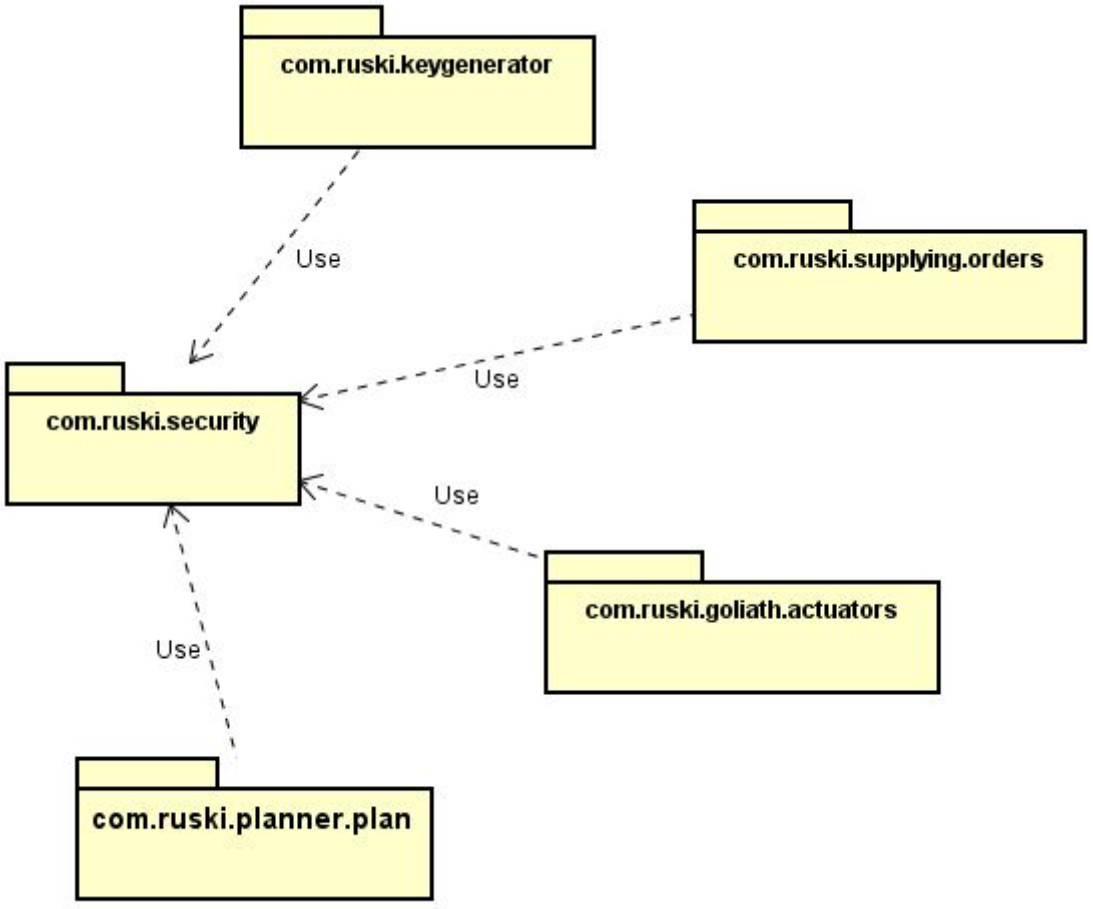
- Todo el código fuente, documentación, archivos de configuración o cualquier otro artefacto referido al desarrollo de los prototipos debe gestionarse en el repositorio Git asignado.

3.Documentación de la arquitectura

3.1. Vistas de módulos

3.1.1.Diagrama de usos de módulos más relevantes en cuanto a seguridad

3.1.1.1. Representación Primaria



3.1.1.2. Catálogo de elementos

Elemento	Responsabilidades
com.ruski.goliath. actuators	Realizar el manejo de los actuadores: en este elemento se dan de alta los actuadores configurados para ejecutar sus propias librerías de drivers. También permite la ejecución de comandos a los propios actuadores.
com.ruski.keygenerator	Generar las llaves públicas y privadas de cualquier aplicación que quiera ejecutar otra aplicación.
com.ruski.planner.plan	Realizar el manejo de los planes y de sus rutas de abastecimiento
com.ruski.security	Manejar todo lo que tenga que ver con la seguridad de aplicaciones (encriptación, llaves, etc). Por este módulo pasan todas las aplicaciones que necesiten interactuar con otras aplicaciones.
com.ruski.supplying.orders	Realizar el manejo de las órdenes de abastecimiento

3.1.1.3. Decisiones de diseño

A la hora de descomponer la solución en módulos, se buscó lograr una alta cohesión y bajo acoplamiento entre los mismos con el objetivo de favorecer la modificabilidad. Es por este motivo, que los módulos están separados en primer lugar por “objeto de negocio” (*order*, *plan* y *actuators*), de modo de tener una agrupación cohesiva de clases.

Para satisfacer el requerimiento REQ-7, se implementó un mecanismo de seguridad que además de permitir la autenticación entre aplicaciones, también nos garantiza privacía, integridad y no repudio. Esto se logra aplicando diferentes tácticas que se detallan a continuación:

VERIFY MESSAGE INTEGRATY: Cualquier tipo de comunicación ya sea a los endpoints de servicios (POST y PUT), como a los mensajes enviados por cola o tópicos contienen un hash de validación para que el receptor pueda por sus propios medios validar la integridad del mensaje enviado.

AUTHENTICATE ACTORS: Por medio del uso de claves públicas y privadas, logramos poder identificar que el emisor del mensaje es quien dice ser y no un intruso

ENCRYPT DATA: Cualquier tipo de comunicación ya sea a los endpoints antes nombrados como a los mensajes enviados por cola y tópicos son encriptados en el emisor y desencriptados en el receptor.

Para mantener la separación de responsabilidades (ya que la seguridad no tiene que ver con ninguna lógica de negocio), se creó el módulo *security* que se encarga meramente de la seguridad y *keygenerator*, que solo genera las llaves públicas y privadas que se utilizan. Se decidió separar estos dos módulos ya que son dos partes independientes dentro de lo que es la seguridad de la aplicación. La forma de generar llaves es independiente de como luego se vaya a aplicar para encriptar y desencriptar la información.

El módulo com.ruski.keygenerator provee dos aplicaciones, utilizadas para realizar las pruebas con POSTMAN:

- Provee una aplicación de consola que permite crear las llaves públicas y privadas para cada aplicación que se le indique. Ejemplo (supplying, planner, pipeline, goliath)
- Provee una aplicación de consola, que nos retorna el mensaje encriptado a enviar por POSTMAN a la hora de realizar invocaciones a los endpoints POST y PUT. Para su invocación es necesario ingresar los siguientes datos:
  - Aplicación de origen
  - Aplicación de destino
  - Json a enviar

Sobre la encriptación, los pasos que se llevan a cabo son:
Emisor

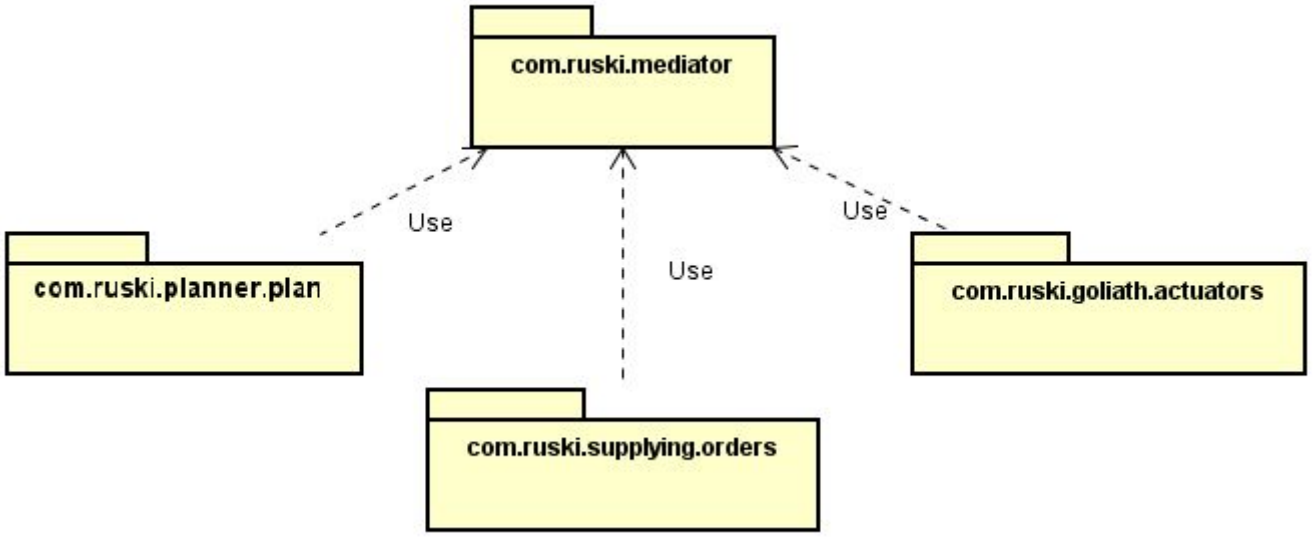
- Se calcula el hash del json a enviar.
- Se encripta el hash calculado con la clave privada del emisor (aplicación de origen)
- Se encripta el hash y el Json a enviar con la clave pública del receptor
- Se envían datos.

Receptor

- Desencripta el mensaje recibido con su clave privada
- Desencripta el hash recibido con la clave pública del emisor
- Calcula el hash del Json obtenido en el primer paso
- Compara el hash recibido con el calculado en el punto anterior.

3.1.2. Diagrama de usos de módulos más relevantes en el uso de Mediador de módulos

3.1.2.1. Representación primaria



3.1.2.2. Catálogo de elementos

Elemento	Responsabilidades
com.ruski.mediator	Exponer objetos compartidos entre las aplicaciones.
com.ruski.goliath.actuators	Realizar el manejo de los actuadores: en este elemento se dan de alta los actuadores configurados para ejecutar sus propias librerías de drivers. También permite la ejecución de comandos a los propios actuadores.
com.ruski.planner.plan	Realizar el manejo de los planes y de sus rutas de abastecimiento
com.ruski.supplying.orders	Realizar el manejo de las órdenes de abastecimiento

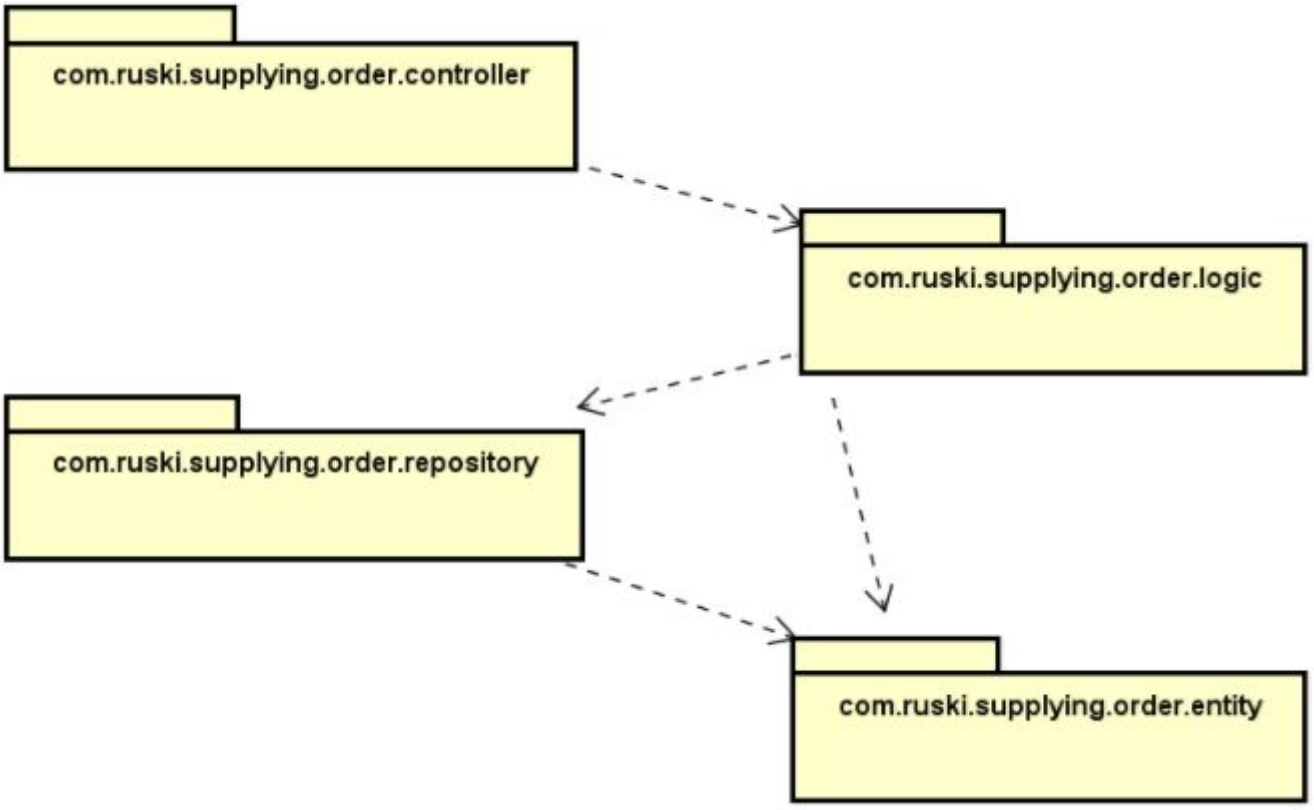
3.1.2.3. Justificaciones de diseño

Para favorecer la modificabilidad se creó el módulo *mediator*, el cual contiene la información que es común a todas las aplicaciones. De esta manera cuando una aplicación se quiere comunicar con otra, encontrará el módulo *mediator,* los objetos que comparten. Por ejemplo cuando la aplicación *Supplying* crea la orden de suministro, debe comunicarse con la aplicación *Planner, que* maneja los planes de suministro; para ello intercambian un objeto del tipo OrderDto, el cual se encuentra en el módulo *mediator.* Esto favorece la modificabilidad ya que si se modifica información en el objeto que se intercambia, no deberán modificarse clases que definen la estructura a intercambiar en las dos aplicaciones.

Lo antes descrito es la aplicación de la táctica de reducir acoplamiento y aumentar cohesión.

3.1.3. Diagrama de descomposición y uso de com.rusi.supplying.order

3.1.3.1. Representación primaria



3.1.3.2. Catálogo de elementos

Elemento	Responsabilidades
com.ruski.supplying.order.controler	Recibir la peticiones http como se detalla en el apartado 3.1.3.3 y retornar una respuesta.
com.ruski.supplying.order.logic	Realizar toda la lógica de negocio de orders.
com.ruski.supplying.order.repository	Persistir los datos de orders
com.ruski.supplying.order.entity	Contener las entidades asociadas a *orders*

3.1.3.3. Interfaces

EndPoint	POST - Orders
Sintaxis	{"clientId":458,"volume":"50","servicePointId":"10","billingCloseday":"25"}
Postcondiciones	Se crea la orden recibida y se encola la creación del plan.
Errores http	● 201: La orden se creó correctamente ● 500: Error al realizar validaciones o guardar.
Excepciones	● OrderValidationException ● OrderOperationException ● RepositoryException ● JMSException ● HashGenerationException

EndPoint	GET - Orders
Sintaxis	Orders
Postcondiciones	Se retorna las órdenes que contiene el sistema.
Errores http	● 200: Órdenes retornadas ● 500: Error al obtener las órdenes.
Excepciones	● OrderOperationException ● RepositoryException

EndPoint	GET - Orders/{id}
Sintaxis	Orders/{id}
Postcondiciones	Se retorna la orden correspondiente a ese id.
Errores http	● 200: Se retorna correctamente ● 500: Error al obtener la orden. ● 400: No se encuentra la orden

Excepciones	• OrderOperationException • RepositoryException

EndPoint	PUT - Orders
Sintaxis	{"clientId":123,"volume":"10","servicePointId":"11","billingCloseday":"20"}
Postcondiciones	Se modifica la orden.
Errores http	• 201: La orden se modificó correctamente • 500: Error al realizar validaciones o guardar. • 400: No se encuentra la orden
Excepciones	• OrderValidationException • OrderOperationException • RepositoryException • JMSException • HashGenerationException

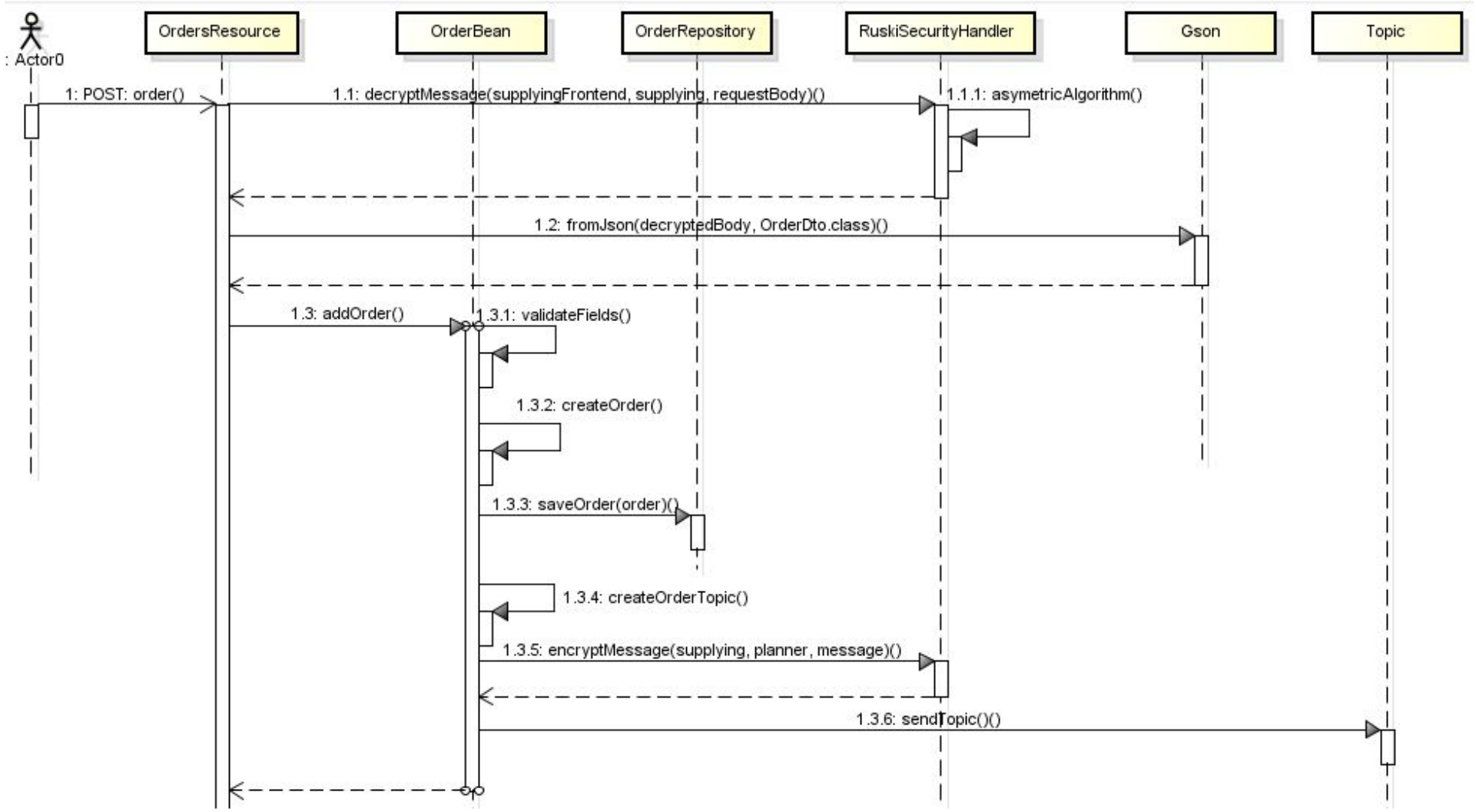
EndPoint	DELETE - Orders
Sintaxis	Orders/{id}
Postcondiciones	Se elimina la orden con ese id.
Errores http	• 201: La orden se eliminó correctamente • 500: Error al realizar validaciones o eliminar. • 400: No se encuentra la orden
Excepciones	• OrderValidationException • OrderOperationException • RepositoryException • JMSException • HashGenerationException

3.1.3.4. Justificaciones de diseño

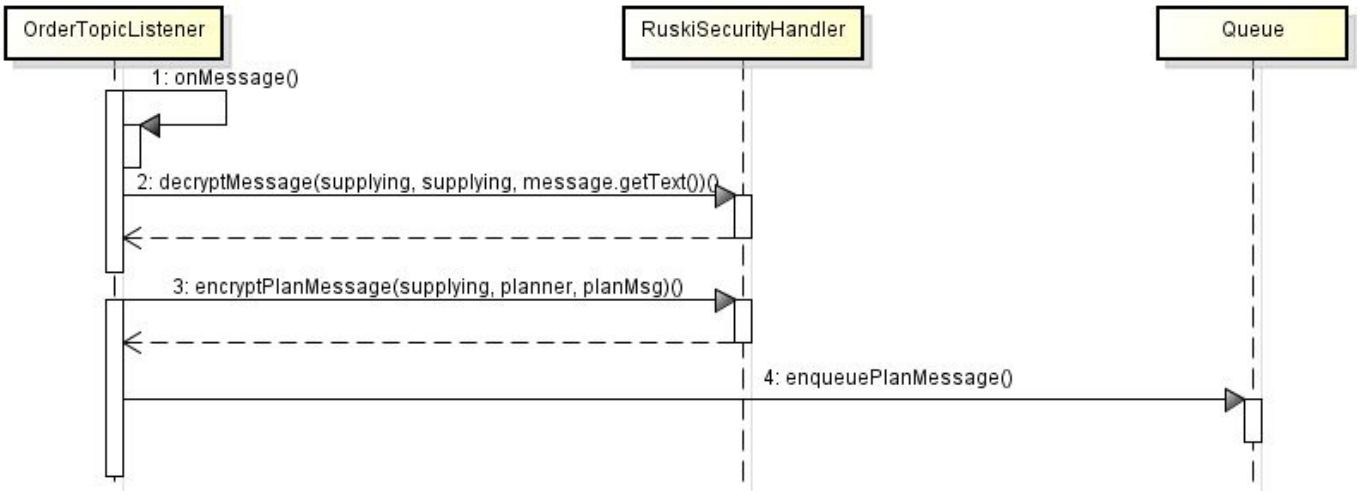
Las órdenes una vez creadas y almacenadas en la base de datos, se guardan en un tópico, que luego notifica a una cola de mensajes que la orden fue creada. De esta manera nos aseguramos que si la aplicación Planner, está desconectada o no está operativa, le es transparente a la aplicación Supplying, y mucho más al usuario final. Mediante este mecanismo aseguramos la disponibilidad del Supplying y el enmascaramiento de cualquier error que pueda darse en el Planner. Además de eso, al hacer asincrónica la llamada a la aplicación Planner, tenemos menor latencia en la respuesta del mismo.

En primera instancia se utiliza tópico porque decidimos dejar extensible el sistema a tener que notificar a otras aplicaciones sobre la creación de una orden. Si en otro momento luego de trabajar con una orden, hay que notificar a otra aplicación, basta con agregar la aplicación al tópico y la aplicación es notificada de la creación, modificación o eliminación de la orden.

En el siguiente diagrama se expone cómo se realiza la creación de una orden y como se guarda la orden creada en el tópico:


: Actor0
OrdersResource
OrderBean
OrderRepository
RuskiSecurityHandler
Gson
Topic
1: POST: order()
1.1: decryptMessage(supplyingFrontend, supplying, requestBody)()
1.1.1: asymetricAlgorithm()
1.2: fromJson(decryptedBody, OrderDto.class)()
1.3: addOrder()
1.3.1: validateFields()
1.3.2: createOrder()
1.3.3: saveOrder(order)()
1.3.4: createOrderTopic()
1.3.5: encryptMessage(supplying, planner, message)()
1.3.6: sendTopic()()

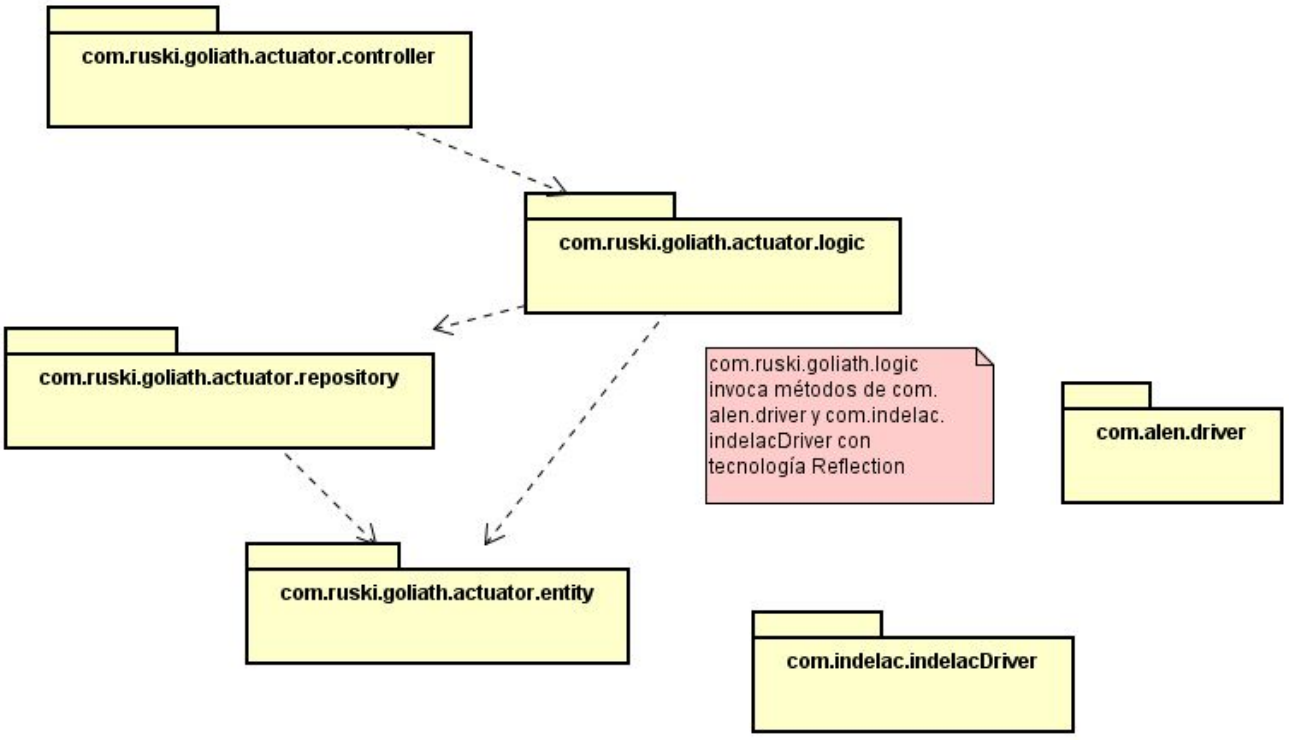
Una vez agregada al tópico, existe un OrderTopicListener, el cual desencripta el mensaje obtenido, encripta el mensaje para enviarle al Planer y lo encola. En el siguiente diagrama se detalla este comportamiento:



Una vez encolado, la aplicación Planner, desencola el mensaje, desencripta y procede a la creación del plan, encolando también la ejecución de la aplicación Calc, asegurándonos la disponibilidad del Planner y haciendo la llamada de forma asincrónica para mejorar tiempos de respuesta.

3.1.4.Diagrama de descomposición y uso de com.ruski.goliath

3.1.4.1. Representación primaria



3.1.4.2. Catálogo de elementos

Elemento	Responsabilidades
com.ruski.goliath.actuator.controler	Recibir la peticiones http como se detalla en el apartado 3.1.4.3 y retornar una respuesta.
com.ruski.goliath.actuator.logic	Realizar toda la lógica de negocio de actuadores.
com.ruski.goliath.actuator.repository	Persistir los datos de actuadores
com.ruski.goliath.actuator.entity	Contener las entidades asociadas a *actuadores*
com.indelac.indelacDriver	Librería de tercero que contiene implementaciones de funciones que ejecutan comandos en el actuador
com.alen.driver	Librería de tercero que contiene implementaciones de funciones que ejecutan comandos en el actuador

3.1.4.3. Interfaces

EndPoint	POST - Actuators
Sintaxis	{"id":"","name":"Valvula Alen","commands":[{"command":"TURN_ON", "jarToExecute" : "c:/temp/Alen.jar", "classToExecute":"com.alen.driver", "methodToExecute":"prender" }, {"command":"TURN_OFF", "jarToExecute" : "c:/temp/Alen.jar", "classToExecute":"com.alen.driver", "methodToExecute":"apagar" }, {"command":"SET_TIMER", "jarToExecute" : "c:/temp/Alen.jar", "classToExecute":"com.alen.driver", "methodToExecute":"cambiarTiempo", "methodParam" : "3" }, {"command":"SET_LEVEL", "jarToExecute" : "c:/temp/Alen.jar", "classToExecute":"com.alen.driver", "methodToExecute":"cambiarNivel", "methodParam" : "1,123" }, {"command":"ENABLE", "jarToExecute" : "c:/temp/Alen.jar", "classToExecute":"com.alen.driver", "methodToExecute":"habilitado",

	"methodParam" : "2018-04-23T18:25:43.511" }, {"command":"DISABLE", "jarToExecute" : "c:/temp/Alen.jar", "classToExecute":"com.alen.driver", "methodToExecute":"deshabilitado", "methodParam" : "2018-04-23T18:25:43.511" }]}
Postcondiciones	Se crea el actuador con la configuración de donde está ubicado cada método que se le puede ejecutar.
Errores http	• 201: El actuador se creó correctamente • 500: Error al realizar validaciones o guardar.
Excepciones	• ActuatorOperationException • ActuatorValidationException • RepositoryException • NullPointerException • HashGenerationException

EndPoint	POST - Actuators/program
Sintaxis	{"commands":[{"ActuatorId":"081092e3-e0e4-432c-bcf3-7ad2b9e5fde3","commandasToExecute":["DISABLE"]}]}
Postcondiciones	Se ejecuta librerías de terceros que aplican los comandos a los actuadores
Errores http	• 201: El programa se ejecutó correctamente • 500: Error al realizar validaciones.
Excepciones	• ActuatorOperationException • ActuatorValidationException • RepositoryException

	• HashGenerationException

3.1.4.4. Justificaciones de diseño

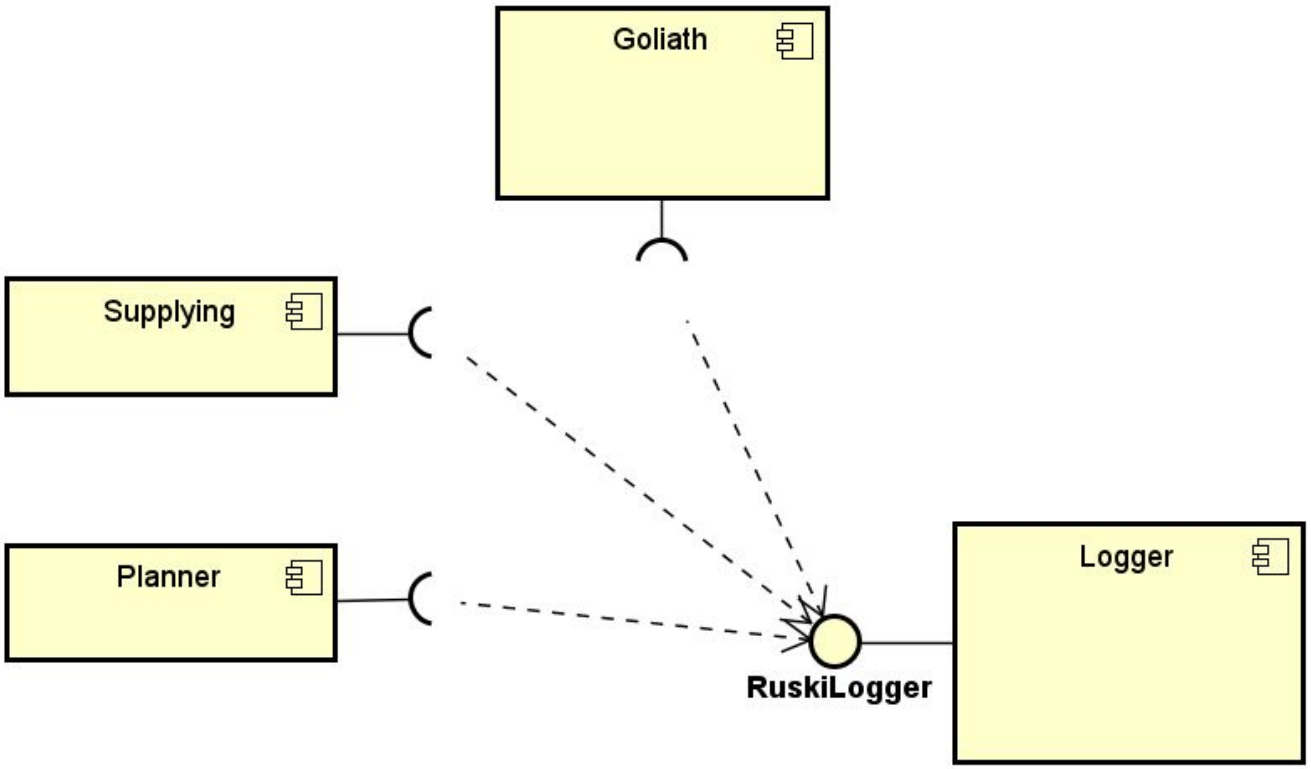
Favoreciendo el atributo de calidad modificabilidad, se implementa mediante Reflection la ejecución de librerías de terceros, las cuales ejecutan ciertos comandos en los actuadores.
Cuando se crea un nuevo actuador, se requiere que se le configuren todos los comandos, indicando la ubicación del JAR a ejecutar, el nombre de la clase, el método y los parámetros que se les pasa para ejecutarlos. Mediante este mecanismo, cada vez que se crea un nuevo actuador, basta con darlo de alta en Goliath y pasarle la configuración de los comandos concretos. Luego, cuando desde el Front-End se quiero programar un plan, o ejecutar algún comando concreto, se irán a buscar las librerías configuradas cuando se creó el actuador.

Se proveen dos ejemplos de librerías de terceros, las cuales retornan “true”, para saber que efectivamente se ejecutó el comando en el actuador correspondiente.

3.2. Vistas de componentes y conectores

3.2.1. Vista de componentes y conectores para Log

3.2.2.Representación primaria



3.2.3.Catálogo de elementos

Elemento	Responsabilidades
Supplying	Mantener las órdenes de abastecimiento
Planner	Mantener los planes de abastecimiento
Goliath	Mantener los actuadores y comandos
Logger	Realizar el log, tanto a nivel de errores como información de pasos que da el sistema

3.2.4.Justificaciones de diseño

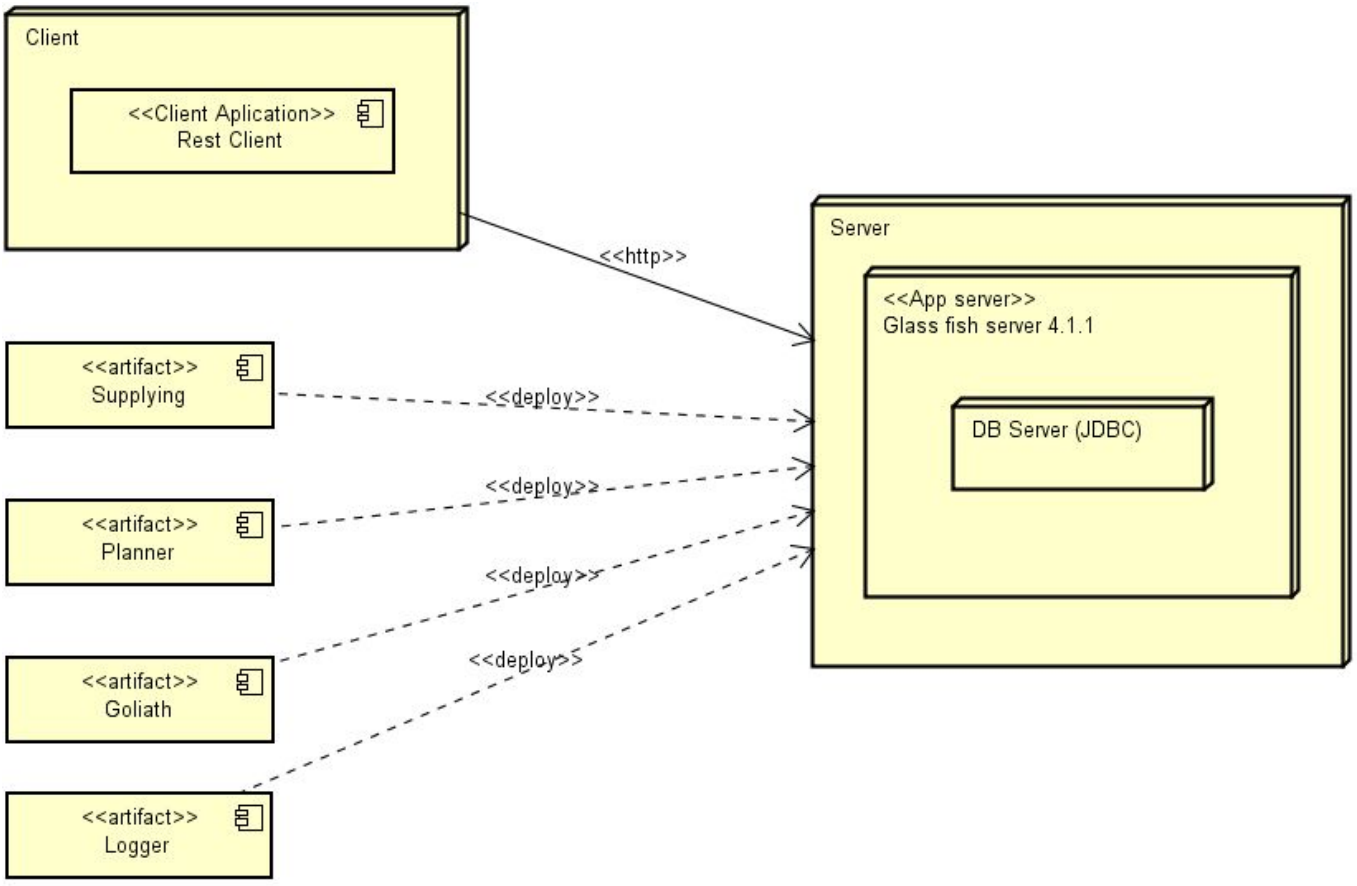
Buscando satisfacer el requerimiento REQ-6 se implementó el componente Logger, el cual es encargado de registrar los principales acontecimientos del sistema, ya sea para tener auditoría como para diagnosticar causas de errores de manera rápida. En este sentido se priorizó la auditoría de eventos para que exista un detalle de las operaciones realizadas en todos las aplicaciones. Para ello se creó un artefacto ruskiLogger, el cual provee una interfaz de logger que podrá ser implementada con cualquier tecnología según como se necesite en estas a aplicaciones o en futuras.
Si se desea cambiar la forma de loguear, basta con desplegar el nuevo componente de logueo que implemente la interface ruskiLogger, no teniendo que modificar ningún otro componente. La implementación de esta interfaz fue realizada utilizando el patrón Adapter delegando por medio de este, las tareas de logger al framework log4j.
Por otra parte, si se desea reutilizar el componente Logger en otra aplicación, basta con desplegar y utilizarlo, siempre respetando la interface ruskiLogger.

Además de favorecer la modificabilidad y la reutilización, por este punto también se favorece la seguridad, ya que se tiene auditoría de todo lo ocurrido en el sistema, aplicando la táctica *MAINTAIN AUDIT TRAIL*.

3.3. Vistas de *allocation*

3.3.1. Vista de despliegue de ROI

3.3.1.1. Representación primaria



3.3.1.2. Catálogo de elementos

Elemento	Responsabilidades
Client	Nodo donde el usuario ejecuta el Rest Client.
Rest Client	Ejecutar las aplicaciones desplegadas en el server y brindarle una interfaz gráfica al usuario final.
Server	Nodo donde se encuentra el servidor de aplicaciones.
Glass fish server 4.1.1.1	Ejecutar las aplicaciones desplegadas para ser ejecutada de los distintos clientes.

DB Server (JDBC)	Persistir los datos de las aplicaciones
Supplying	Manejar lecturas, escrituras y modificaciones de órdenes
Planner	Manejar lecturas, escrituras y modificaciones de planes
Goliath	Manejar lecturas, escrituras y modificaciones de actuadores
Logger	Registrar auditoría y errores en el sistema

3.3.1.3. Justificaciones de diseño

Todas las aplicaciones debían desplegarse y ejecutarse de forma independiente, para esto se tomó la decisión de utilizar varios artefactos. La independencia es tal, que podrían ejecutarse cada uno de estos componentes en diferentes nodos. Esto permite poder intercambiar componentes en tiempo real, por ejemplo el logger, como fue detallado en apartado 3.2.4.
Se puede apreciar en el diagrama la utilización del patrón cliente-servidor que permite a las aplicaciones cliente, en este caso Rest Client (Postman), comunicarse con el servidor a través del protocolo HTTP.