Universidad ORT Uruguay
Facultad de Ingeniería

Ruski Oil International (ROI)
Arquitectura de Software - Obligatorio

Entregado como requisito para la obtención del crédito Arquitectura de Software

Veronica Seoane - 172103
Alejandro Rama - 167015

Grupo N7A
Docentes: Andrés Calviño, Mathias Fonseca

2017

Índice

1. Introducción

El siguiente documento de arquitectura presenta una visión general de la arquitectura del sistema, representado a través de una serie de diferentes vistas, cada una de las cuales muestran un aspecto particular del software desarrollado.

2. Propósito

En el presente documento se da una visión global y comprensible de la arquitectura del sistema de ROI (Ruski Oil International). En él se va encontrar detalles de implementación, las principales decisiones de diseño tomadas y sus correspondientes justificaciones.

3. Antecedentes

3.1. Propósito del sistema

El sistema de ROI tiene como propósito ser una solución que permita gestionar la distribución y comercialización de gas natural y petróleo crudo.
Además permite administrar y configurar la programación del suministro en cada actuador de la ruta asignada. Establece tramos óptimos de conexión cada vez que el plan es generado.

3.2. Objetivos de Arquitectura

3.2.1. Requerimientos Funcionales

- **RF1 – Registrar Orden de abastecimiento**
  Permitir crear, modificar, eliminar y consultar Ordenes de Abastecimiento en **roi-supplying.**

- **RF2 – Comunicar las Ordenes 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.

- **RF3** – 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 (implica conectarse a la API de roi-pipeline-calc para obtener la ruta del 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 marca el Plan de Suministro como “anulado”

- **RF4 – Habilitar un nuevo actuador para enviarle comandos**
  Para un nuevo actuador, facilitar una forma de registrar el controlador correspondiente en **Goliath**

- **RF5 – Gestión de errores y fallas**
  El sistema debe proveer suficiente información, que permita conocer el detalle de las tareas que se realizan. En particular en el caso de ocurrir alguna falla o cualquier tipo de error.

3.2.2. Requerimientos No Funcionales

RNF1 – El impacto de agregar/cambiar un controlador debe ser mínimo
Al agregar o actualizar un controlador, debe 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.

RNF2 – Posibilidad de cambio de herramienta/app de Logging con poco impacto
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 re utilizar esta solución en otras aplicaciones, con el menor impacto posible en el código.

RNF3 – Asegurar la legitimidad de los request al backend
Los request al backend deben ser legítimos (de aplicaciones permitidas para la comunicación).

3.2.3. Restricciones

- La implementación del backend debe desarrollarse en JEE
- Tecnologías (Web service, EJB, JMS, JPA)
- Podrían utilizarse otras tecnologías Java(logging, mail, formatos de datos -xml, json)
- Para calcular el tramo más óptimo se debe usar la API de roi-pipeline-calc que está publicada en 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.
- Fecha de entrega: 29-11-2017

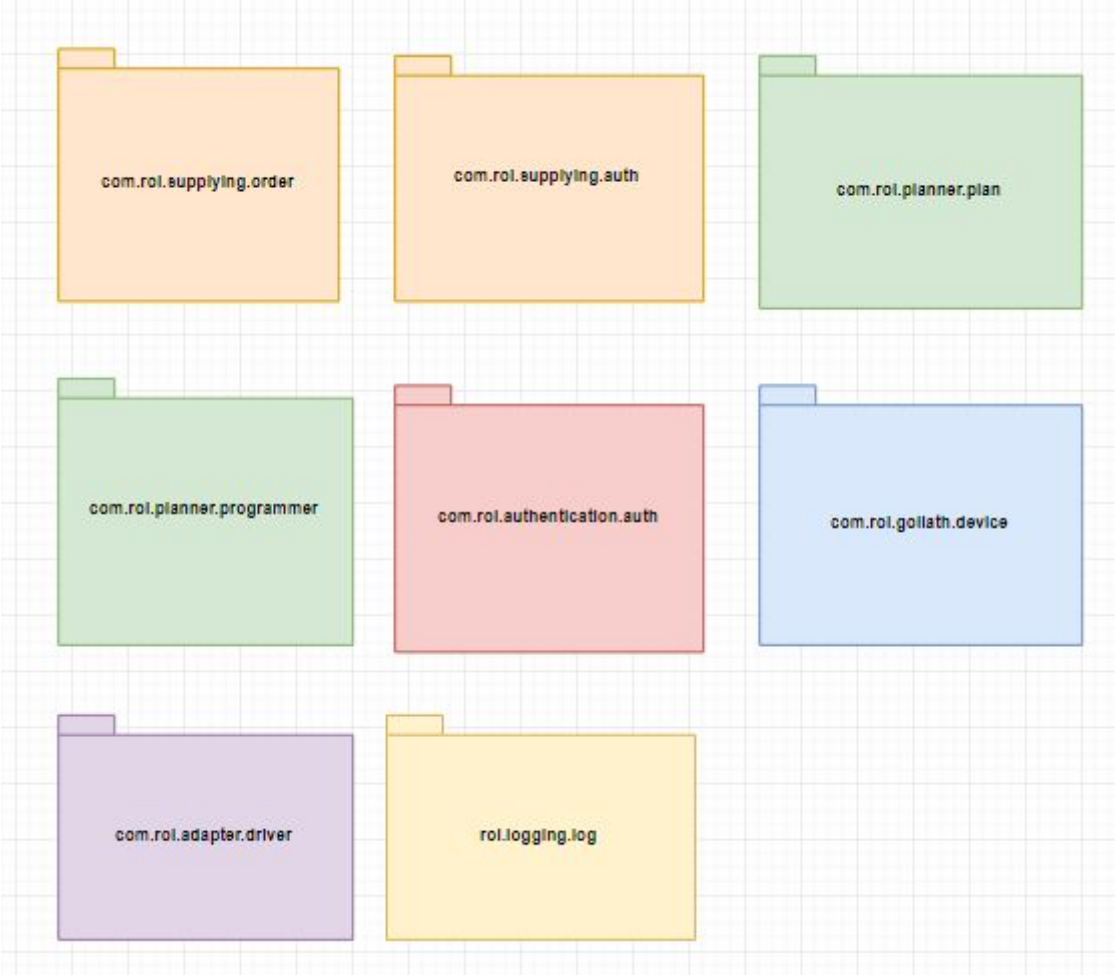
4. Documentación de Arquitectura

A continuación se describe la arquitectura del sistema de ROI desde distintos puntos de vista a través de componentes y conectores, módulos y asignación.

Vista de Módulos del sistema:

Se separó por color aquellos módulos pertenecientes al mismo subsistema.

Presentación primaria:



Catálogo de elementos:

Elemento	Responsabilidad
com.supplying.orders	Este módulo contiene todo para poder registrar una orden y que luego sea enviada para generar el plan. Permitiendo crear, modificar o eliminar lógicamente una orden.
com.supplying.Authentication	Este módulo permite el acceso a la aplicación roiAuthentication.
com.roi.planner.planes	Este módulo contiene todo lo necesario para generar un plan a partir de una orden recibida.
com.roi.planner.programmer	Aquí se encuentran todas las clases relacionadas a programar a través de comandos ya definidos a los actuadores de un plan. Además desde aquí se establece la comunicación con Goliath, donde le envía el plan con los actuadores y los comandos programados.
com.roi.authentication.auth	En este módulo se especifica o controla qué aplicación tiene acceso a qué otra aplicación.
com.roi.goliath.device	Se encarga de ejecutar los comandos sobre los dispositivos de la red de suministro.
com.roi.logging.log	Se encarga de loguear en un archivo los logs del sistema.
com.roi.adapter.device	Simula ser un adapter de un dispositivo.

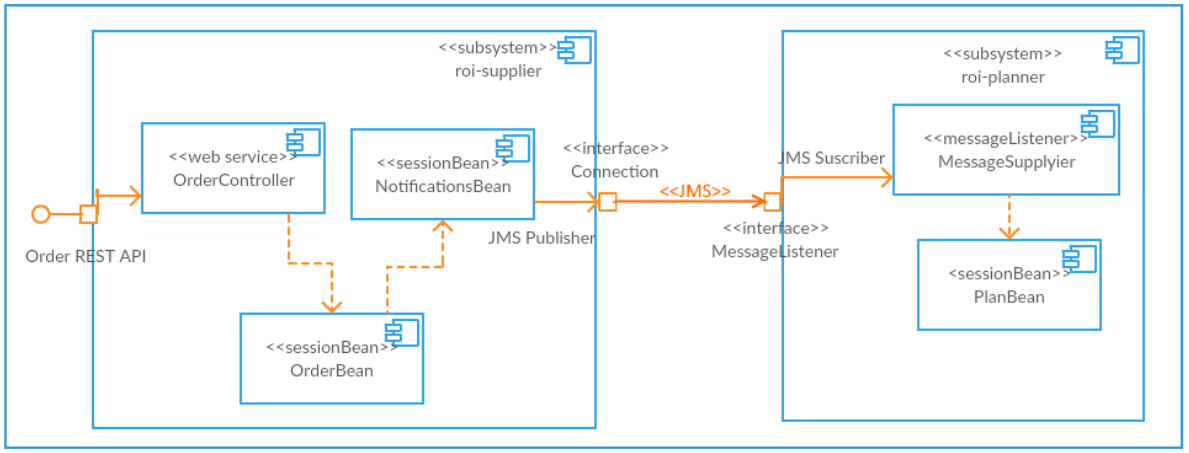
Razón:

Nuestro proyecto está subdividido en una extensa red de paquetes interrelacionados, en este caso, se eligió arquitectura basada en módulos para favorecer el reúso común, apuntando a la cohesión de paquetes, por lo tanto, decidimos que aquella clase que usa otra clase, estas deben ir juntas en el mismo paquete.
La forma de trabajo fue, se utilizó un único paquete al principio y luego clases fueron “abandonando” el paquete si ellos no eran usados por el resto de las clases, pasándolas a paquetes más específicos, basándonos en el principio de reúso común.

RF2 – La comunicación entre roi-supplyer y roi-planner debe ser inmediata

Vista de Componentes y Conectores

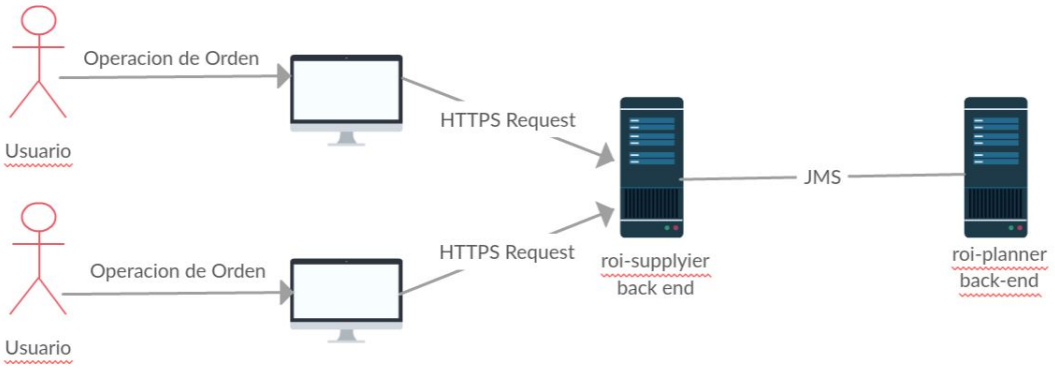
Presentación primaria



Catálogo de elementos

	Descripcion
JMS	tipo:Queue nombre de la cola : SupplyingQueue JAVA JMS Docs

Diagrama de contexto



Razón

Ante la necesidad del cliente de que la comunicación de las operaciones de órdenes sean comunicadas al instante para que éstas sean procesadas se toma la siguiente decisión teniendo en cuenta el AC (Atributo de Calidad) *Availability* aplicando las tácticas correspondientes

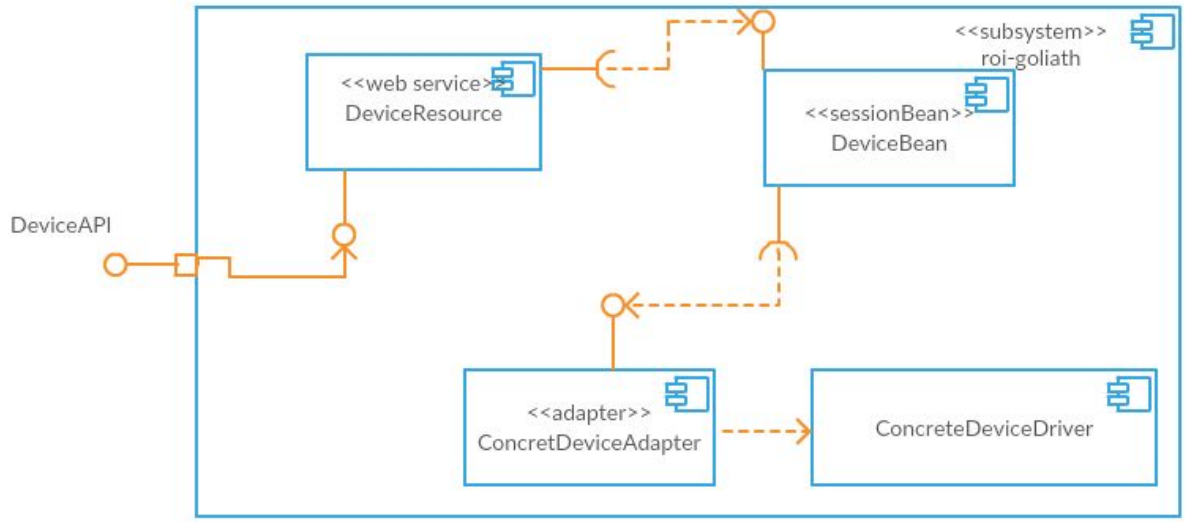
Por un lado se busca que la comunicación sea instantánea y por otro que si roi-planner se cae , esto sea invisible para el usuario que está ingresando las operaciones de órdenes en roi-supplying, para cumplir con ambas, se aplica la táctica *Retry* que se logra mediante la implementación de la comunicación mediante una JMS Queue , logrando así que todas las operaciones de orden se encolen para ser enviadas y procesadas por roi-planner, con lo cual en caso de que roi-planner se caiga , estas permanecerán en la cola hasta que el sistema vuelva a funcionar

Las otras opciones que fueron evaluadas pero descartadas son:

- Con web services (exponiendo un endpoint en roi-planner) : si bien esta opción cumple con la parte de “inmediatez”, fue descartada ante la complicación que significa utilizar Retry con ella, ya que de una u otra forma se podría lograr aplicar la táctica con esta opción pero no nos convenció, por ejemplo se podría tener una propiedad en las órdenes que diga “Enviado” true o false, y que si roi planner está offline y el request falla, entonces se le pone false , y luego con un servicio programado (scheduled job / scheduled service) , cada cierto tiempo se volverían a enviar todas las órdenes que no fueron enviadas. Pero esto agrega complejidad y validaciones extras que hay que hacer, por ejemplo que pasa si roi supplyier se cae justo en el momento luego que envía una orden, la cual es procesada satisfactoriamente por planner pero no seteada como Enviada = true en supplyier, en este caso se procesaría dos veces. Entonces, básicamente ayuda a resolver el problema por un lado pero lo complica por el otro
- JMS Topic : Esta fue descartada ya que no es posible detectar si un mensaje llegó o no a roi planner para ser procesado, el JMS Topic notifica a quienes estan suscriptos pero si estos estan offline el mensaje se pierde
- Servidor de Archivos + JMS Queue : Poner la orden en un archivo como ya se hacía pero avisar al instante a roi-planner sobre la misma para que vaya, la tome del servidor y la procese. Esto se descarto ya que a pedido del cliente se quería deshacer del intermediario lo cual de todas formas es bueno y que disminuye la latencia, pero con esta opción se hubiese podido aplicar Retry sin problemas también

RNF1 - el impacto de agregar/modificar un controlador debe ser mínimo y si es posible cero a nivel codigo

Representación primaria



Catálogo de elementos

Elemento	Notas
DeviceAPI	Expone el endpoint /device POST por el cual se recibe una lista de actuadores con los comandos a ejecutar por JSON. La request debe ser legítima ya que valida contra el servidor de autenticación central
ConcretDeviceAdapter	Es un adapter (Patrón Adapter) – en el cual se mapea los comandos genéricos de los controladores a los específicos de cada uno
ConcreteDeviceDriver	Es el controlador concreto para un device/actuador específico
DeviceBean	Es el encargado de levantar a ConcretDeviceAdapter por Reflection

Razón

Para esta decisión tuvimos en cuenta el AC Modificabilidad , ya que se pedía que se pudiera agregar o cambiar controladores con el menor impacto de código posible.

Por un lado, anticipamos los cambios y utilizamos la táctica de Reduce Coupling – Encapsulate , creando adapters para los controladores, con lo cual encapsulamos el “mapeo” de los comandos genéricos a los comandos específicos del controlador , en

el cual todos los adapters van a implementar una interfaz IDriverAdapter asegurando que todos cumplan los mismos métodos. Estos adapters se crean como una aplicación separada que mediante composición-delegación mapea y delega los comandos al driver concreto.

Con el patrón Adapter nos aseguramos de marcar un límite en cuanto al impacto de código que pueden producir un cambio/modificación en un driver concreto

Y para disminuir más aún el impacto del cambio decidimos evaluar la táctica de "Registro en tiempo de ejecución" que nos permitiría agregar/cambiar controladores (adapters) en tiempo de ejecución, es con esto que llegamos a que tendríamos que utilizar Reflection que justamente es eso lo que hace.

Puntos a tener en cuenta para el sistema real que no se implementaron en el prototipo

- Ubicación de los adaptadores a cargar por reflection : la ubicación, está hardcodeada a D:/temp en el prototipo, pero para el sistema real se pondrá en un archivo de configuración
- Validación de comandos : De momento agregamos una mínima validación en cuanto a la semántica de los comandos como soporte para la demostración del prototipo pero que no representa las intenciones reales para el sistema. Lo ideal sería que además cada adapter proporciona un método que valide los comandos (algunos controladores podrían aceptar valores mayores o menores que otros por ejemplo)

Algo que no hicimos, pero creemos que es muy importante, teniendo en cuenta el AC Availability para goliath con el objetivo de que los defectos no se transformen en fallas o enmascarar estos,

1. La ejecución de los comandos de cada actuador debe ser encolada , porque está la chance de que por una u otra razón , la conexión al dispositivo falle y los comandos queden en la nada, por eso aplicando la táctica Retry llegamos a que aplicando una cola local (que no sea JMS ya que se necesitaría JMS en el receptor también, que no está bajo nuestro control) , la ejecución de comandos se volvería a intentar hasta que sea satisfactoria , esto habría que hacerlo de una forma inteligente para que si el primero en la cola sigue fallando, no entorpezca a los demás, por lo que se podría llegar a tener varias colas en paralelo y quizás hasta con prioridad/niveles basados en la cantidad de fallos que tuvo cada ejecución de comandos
2. En el instante que un un plan de ejecución de comandos llega a goliath, esta debería ser encolado utilizando alguna tecnología de background processing con queues como Quartz Scheduler para :

a. Por un lado, enmascarar los posibles defectos en cuanto al procesado de la misma, siendo así invisible a roi-planner
b. Asegurar el procesado del request en el caso de que falle por alguna razón, con una alarma/log que nos va a avisar en el caso de que falle 3 veces ya que en ese caso es probable que se precise la intervención de personal

Por otro lado, por un tema de Seguridad, habría que asegurar que la ejecución de comandos sea completa o ninguna. Basándonos en la táctica de Roll back es que llegamos a que si un comando falla se debería poder hacer “roll back” es decir, ejecutar los comandos inversos sobre el dispositivo para restablecer su estado. Esto, no fue implementado pero deberá ser tenido en cuenta en el sistema real.

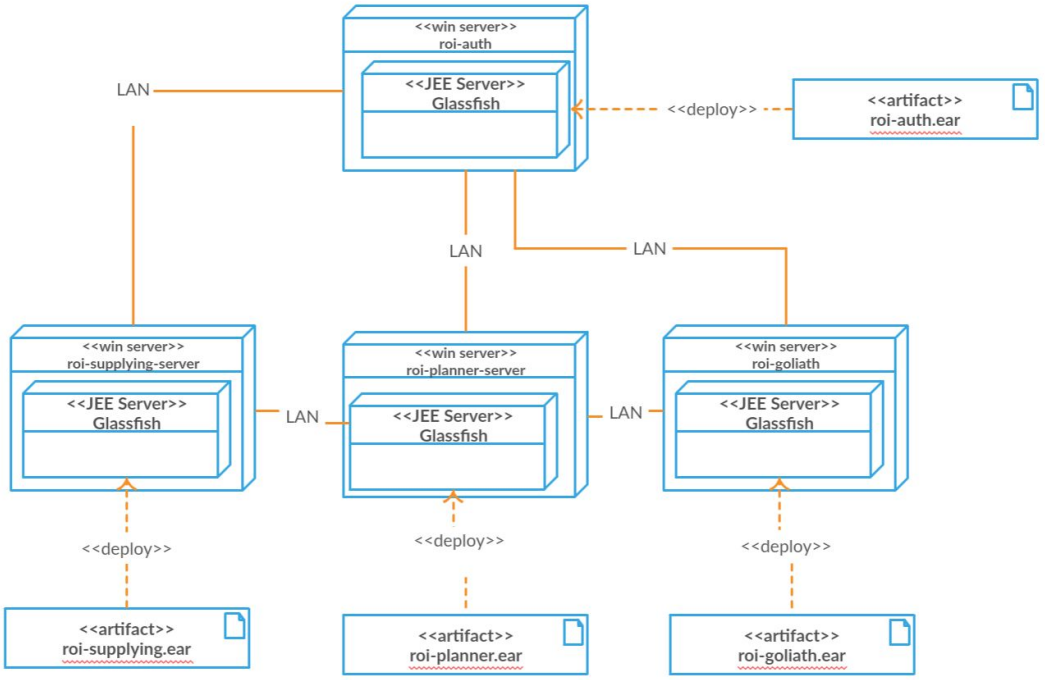
Este subsistema (considerando la solución entera como un Sistema de sistemas) va a tener especial foco en cuanto a testing, ya que se quieren probar controladores nuevos que van saliendo al mercado, entonces aplicamos tacticas de AC Testeabilidad.

- Limit structural complexity : Se tuvo en cuenta desde el primer momento que este subsistema debia tener una complejidad reducida, si bien no iba a ser muy complejo siempre se tuvo esto en mente siguiendo ciertas pautas como limitar las herencias, niveles de jerarquía y endpoints/comunicaciones con el exterior. Por ejemplo, para este prototipo tenemos una base de datos con Actuadores puramente para demostrar cómo funciona el prototipo pero esta sería una base de datos la cual no sería responsabilidad de goliath mantenerla, ya que queremos que la única función de goliath sea la ejecución/procesamiento de un plan específico sobre los controladores de los actuadores de este
- Specialized interfaces : Agregamos metodos especificos para colaborar con el logging/monitoreo del procesado de un plan, por ejemplo un método que retorna el estado completo del objeto
- Record / Play back : Con la ayuda de los métodos agregados en el punto anterior es que centralizamos el logeo del estado del objeto en el momento antes de cruzar una interfaz para que en caso de error esta pueda ser recreada
- Sandbox (no aplicada) :Se despliega una instancia de roi-goliath pura y exclusivamente con fines de testeo , sin que esté ligada con otras partes del sistema logrando así probar roi-goliath sin la preocuparse por las consecuencias. Esta tactica la evaluamos pero **NO** la aplicamos ni pensamos aplicarla en un futuro ya que de todas formas esto no es tan asi, porque si habria que preocuparse si utilizando el sandbox libero 50000m3 de gas lo cual sería un gran problema, así que no es el caso para su aplicación.

RNF3 - Asegurar legitimidad de los requests

Vista de Asignación

Presentación primaria



Catálogo de elementos

Glassfish	Versión 4.1.1
roi-suppling-server	Windows Server 2012
roi-planner-server	Windows Server 2012
roi-goliath	Windows Server 2012
roi-auth	Windows Server 2012 , 128 gb Ram, CPU 3.6 Ghz Octa core, 128 gb ssd

Razón

Se pedía que los requests a los backends sean autenticadas , o sea validar que sea una request legítima para la app que la recibe.

En un principio, para la toma de decisión, tuvimos en cuenta el AC Seguridad y en base al problema a resolver aplicamos la táctica “Identificar actores” con la cual llegamos a la decisión de que cada aplicación tendría un CÓDIGO el cual la identifique y además cada aplicación tendría una lista de aplicaciones (sus códigos) las cuales tienen permiso para accederla/enviarle requests.

O sea que cada app tendría un archivo de configuración en donde tendría
appCode = [código]
appWithPermissions = [codigoDeApp1],[codigoDeApp2]

Entonces, si el App#1 hace un request a App#2, va a enviar su código en el header para que App#2 sepa quien es y si puede acceder.

Para simplificar, en este prototipo se hardcodeó el valor de esa clave que es enviada para validar contra un servicio encargado de definir si tiene acceso tal aplicación a otra.

Si bien esto soluciona, si evaluamos la solución en cuanto al AC Modificabilidad, nos damos cuenta que si agregamos una nueva app/subsistema , o si necesitamos cambiar el código de alguna de las existentes por alguna razón, entonces el impacto es bastante grande ya que tendríamos que ir modificar la configuración de todas las apps necesarias

Entonces es que se nos ocurrió hacer la autenticación centralizada en un servidor específico, el cual va a manejar las aplicaciones, sus códigos y los permisos entre estas. De esta manera cada vez que una app recibe un request, va a validar el mismo contra el servidor de autenticación roi-auth

Esto agrega otro AC a tener en cuenta, *Availability* de roi-auth el cual va a ser utilizado por las demás apps siendo clave en el funcionamiento de todas. Por eso, es que siguiendo con las tácticas de Availability - Active Redundancy es que decidimos tener 3 nodos procesando las autenticaciones en paralelo por si uno falla , otro se encargue de procesar el request.

Además, es importante estar al tanto de todo lo que pase en roi-auth así como su salud, por eso guiándonos por la táctica de Availability – Detect faults – Monitor es que vamos a loguear y monitorear todo lo que suceda en el mismo

Por otro lado, sabemos que esto es algo que va a impactar en todo el sistema y que si la autenticación demora 1 segundo entonces cada request que se haga entre aplicaciones va a demorar un segundo más, y si bien esto no es crítico nos pareció importante aplicar una táctica de AC Performance – Increase Resources para la gestión de recursos y así es que decidimos ciertas especificaciones del servidor para garantizar esto.

- A modo de demostración, solo fue implementado contra el backend de roi-supplying
- Las tokens de cada subsistema deberían estar en un archivo de configuración pero para el prototipo a modo de simplificar las dejamos hardcodeadas
- No deberíamos precisar encriptar los datos ya que estamos dentro de una red privada pero consideramos hacerlo para protegernos ante un eventual intruso en la misma

- Siguiendo con el punto anterior, llevamos un log de los intentos fallidos de autenticación para detectar y anticipar cualquier anormalidad

RNF2 - Posibilidad de cambio de herramienta de logging con bajo impacto y reutilización de la misma en las demás apps

Es un tema que toca Modificabilidad , así que primero basándonos en la primera llegamos a que teníamos que Encapsular y anticipar el cambio , luego aplicando tácticas de Defer Binding de Configuration at deployment time es que llegamos a que teníamos que tener una interfaz de Logger común a la cual pudiéramos setear una implementación de logger, pero esta tendría que estar embebida/encapsulada en un jar, ya que esto tiene que ser reutilizable.

Pensamos una solución posible pero buscando llegamos a algo pre-hecho en github Simple logging Facade for Java que nos ofrece exactamente lo que queríamos , la posibilidad de configurar un third party logger , esto lo hicimos en un proyecto aparte, que compilamos en un .jar y reutilizamos en nuestros proyectos , una vez agregado al proyecto en el cual se precisa loggear, la clase proveída por Simple logging Facade for Java, para levantar la implementación de logger de terceros que seteamos , que esto lo hace buscando la dependencia con java.util.ServiceLoader class y utilizando reflection.

El REQ no pedía que se aplicará a todas las apps, sino que solo pedía una arquitectura que soporte el requerimiento así que solo lo aplicamos en goliath a modo de ejemplo para poder mostrarlo

5. Decisiones de diseño

Uso de EJBs - Enterprise JavaBean.

Se decide utilizar este API en principio porque forman parte del estándar de construcción de aplicaciones JEE, por otro lado, permite modularizar de forma eficiente las responsabilidades, encapsular la lógica y manejar de forma eficiente la seguridad del sistema permitiéndonos abstraer los problemas generales de una aplicación como concurrencia, transacciones, persistencia entre otros para enfocar en el desarrollo de la lógica de negocio en sí.
Nosotros utilizamos los tres tipos de *EJB, Entity EJB* para las entidades a persistir, los *Session EJB* para el manejo de la lógica del sistema y relación con el *EntityManager.*
Y por último hemos usado además *Message-driven EJB* para el manejo de JMS (*Java Messaging System*) que se activan al recibir un mensaje dirigido a una cola, facilitando su uso.

Uso de gson

Para el intercambio de información de los diferentes sistemas se usó JSON (JavaScript Object Notation) es un formato de intercambio de información que está basado en estructuras de pares clave - valor.
Para ello necesitamos serializar y deserializar nuestro JSON, y lo hicimos a través de la librería Gson.
Esta librería nos permite la conversión entre objetos Java y JSON.

6. Aseguramiento de Calidad

Para mejorar la mantenibilidad del sistema se usaron varias técnicas que facilitan la introducción de cambios y disminuyen los defectos del sistema.
Se utilizó las buenas prácticas de Clean Code para tener un código mantenible y reusable.
Por otro lado se mejoró el código con el uso de CheckStyle.
Se siguieron las buenas prácticas de restful recomendadas en la siguiente URL:

https://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/

7. Anexo

7.1. End Points

GET http://localhost:8080/roiSupplying-war/orders
GET http://localhost:8080/roiSupplying-war/orders/1
POST http://localhost:8080/roiSupplying-war/orders
DELETE http://localhost:8080/roiSupplying-war/orders/1
PUT http://localhost:8080/roiSupplying-war/orders/1
GET hhttp://localhost:8080/roiPlanner-war/plans
PUT http://localhost:8080/roiPlanner-war/plans/firstApproval/1
PUT http://localhost:8080/roiPlanner-war/plans/secondApproval/1
PUT http://localhost:8080/roiPlanner-war/actuator/setCommands

7.2. Manual de Configuración

Para comenzar, se deberá ejecutar los siguientes comandos en la consola de *GlassFish*. Para la creación de las bases de datos y la queue para poder comunicar los proyectos *roi-Supplying* y *roi-Planner*

7.2.1. Creación de Base de datos

ROI-SUPPLYING DB

```
create-jdbc-connection-pool --datasourceclassname
org.apache.derby.jdbc.ClientDataSource --restype javax.sql.DataSource --property
portNumber=1527:password=root:user=root:serverName=localhost:databaseName=roi-supp
lying-DB:connectionAttributes=\;create\\=true roi-supplyingPool

create-jdbc-resource --connectionpoolid roi-supplyingPool jdbc/roi-supplyingPool
```

ROI-PLANNER DB

```
create-jdbc-connection-pool --datasourceclassname
org.apache.derby.jdbc.ClientDataSource --restype javax.sql.DataSource --property
portNumber=1527:password=root:user=root:serverName=localhost:databaseName=roi-plan
ner-DB:connectionAttributes=\;create\\=true roi-plannerDBPool

create-jdbc-resource --connectionpoolid roi-plannerDBPool jdbc/roi-plannerDBPool
```

AUTHENTICATION DB

```
create-jdbc-connection-pool --datasourceclassname
org.apache.derby.jdbc.ClientDataSource --restype javax.sql.DataSource --property
portNumber=1527:password=root:user=root:serverName=localhost:databaseName=authenti
cation:connectionAttributes=\;create\\=true authenticationPool

create-jdbc-resource --connectionpoolid authenticationPool jdbc/authenticationPool
```

GOLIATH DB

create-jdbc-connection-pool --datasourceclassname
org.apache.derby.jdbc.ClientDataSource --restype javax.sql.DataSource --property
portNumber=1527:password=root:user=root:serverName=localhost:databaseName=roi-golia
th-bd:connectionAttributes=\;create\\=true roi-goliath-pool

create-jdbc-resource --connectionpoolid roi-goliath-pool jdbc/roi-goliath-pool

7.2.2. Creación de Queue

SUPPLYING QUEUE
create-jms-resource --restype javax.jms.ConnectionFactory jms/ConnectionFactory

create-jms-resource --restype javax.jms.Queue --property Name=SupplyingQueue
jms/SupplyingQueue

7.2.3. Scripts en la Base de Datos

Luego se necesita crear los comandos del sistema en la base de datos para eso se debe de realizar las siguientes consultas SQL:

```
INSERT INTO COMMAND (ID, "NAME", TYPEVALUECOMMAND)
        VALUES (1, 'TURN_ON', '');
INSERT INTO COMMAND (ID, "NAME", TYPEVALUECOMMAND)
        VALUES (2, 'ENABLE', 'datetime');
INSERT INTO COMMAND (ID, "NAME", TYPEVALUECOMMAND)
        VALUES (3, 'DISABLE', 'datetime');
INSERT INTO COMMAND (ID, "NAME", TYPEVALUECOMMAND)
        VALUES (4, 'TURN_OFF', '');
INSERT INTO COMMAND (ID, "NAME", TYPEVALUECOMMAND)
        VALUES (5, 'SET_LEVEL', 'double');
INSERT INTO COMMAND (ID, "NAME", TYPEVALUECOMMAND)
        VALUES (6, 'SET_TIMER', 'long');
```

Para que el sistema de autenticación funcione se deben registrar las siguientes aplicaciones con sus respectivos *tokens* o claves mediante SQL:

```
INSERT INTO APPAUTH (ID, "NAME", TOKEN)
        VALUES (1, 'roiPlanner', 'ffb9e7b3-a2f4-4c5d-8ea2-e0c038e07319');
INSERT INTO APPAUTH (ID, "NAME", TOKEN)
        VALUES (2, 'roiSupply', 'bd1315c8-8a5c-4854-82d7-3caeaf0a90de');
INSERT INTO APPAUTH (ID, "NAME", TOKEN)
        VALUES (3, 'Goliath', '6917cbe4-a511-4a5f-a336-d3a28b442bff');
INSERT INTO APPAUTH (ID, "NAME", TOKEN)
```

```
        VALUES (4, 'FrontEnd roiPlanner', '8f9ddb11-85bd-45b9-882a-dc321077a4ef');
INSERT INTO APPAUTH (ID, "NAME", TOKEN)
        VALUES (5, 'FrontEnd roiSupply', '7535e79e-5e94-4b7f-9386-634f01657658');

INSERT INTO APPLICATION_PERMISSIONS (APPAUTH_ID, HASACCESSTO_ID)
        VALUES (2,1);
INSERT INTO APPLICATION_PERMISSIONS (APPAUTH_ID, HASACCESSTO_ID)
        VALUES (1,3);
INSERT INTO APPLICATION_PERMISSIONS (APPAUTH_ID, HASACCESSTO_ID)
        VALUES (5,2);
```