Universidad ORT Uruguay
Facultad de Ingeniería
Bernard Wand Polak

Arquitectura de Software

Obligatorio

"ROI System"

Pablo Morales – 177769
Sebastián Uriarte – 194973

Docentes: Gastón Mousqués y Mathías Fonseca.

2018

Índice

1. Introducción

En este obligatorio, entregado como parte de la currícula de la materia Arquitectura de software, de la Universidad ORT Uruguay, se buscará lograr aplicar los distintos conocimientos y estrategias vistos a lo largo del curso respecto de la estructuración a nivel arquitectónico de un sistema de *software*, con el fin de no sólo cumplir con los requerimientos del cliente a nivel funcional sino también potenciar ciertos atributos de calidad buscados.

En particular, la propuesta trataría del desarrollo de un prototipo de sistema que permitiera y facilitara el manejo de órdenes y planes de suministro, así como su ejecución automatizada, vinculado a la actividad económica de una empresa dedicada a la explotación, distribución y comercialización de gas natural y petróleo crudo, *Ruski Oil International (ROI).* Se haría, en cuanto a esto, especial énfasis en la necesidad de permitir la fácil extensibilidad y mantenibilidad al sistema, introduciendo un componente (Kremlin) que mediara en la comunicación entre las restantes aplicaciones vinculadas al sistema, que se habría de desarrollar enteramente en lenguaje Java, utilizando la tecnología *Java Enterprise Edition.* El propósito del presente documento es pues proveer una especificación lo más completa posible de la arquitectura del sistema desarrollado como parte de este obligatorio.

2. Requerimientos

El conjunto de requerimientos a ser abordados en este obligatorio fue entregado como parte de la propuesta inicial, disponible en formato *online.*[1] Se incluye aquí en las páginas siguientes el listado original del mismo:

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, evitando el intercambio de archivos reduciendo así los tiempos de procesamiento.

REQ-3: Registrar Plan de Suministro

Para cada Orden de Abastecimiento recibida de forma automática desde **roi-supplying**, **roi-planner** registra 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. No se puede crear un Plan de otra forma (por ejemplo, manualmente).
- 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”. No se pueden modificar Planes anulados, ni se pueden eliminar directamente.

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), la que puede ser modificada por el Agente de Planificación.

[1]: https://docs.google.com/document/d/1K1siHtUzbsl0avB6HEkL_hn652pvaZqCVF4lHfvUmBQ

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: Aprobación de Plan de Suministro

El Agente de Planificación aprueba un Plan de Suministro en **roi-planner** y este es enviado automáticamente a **Goliath** para que ejecute las acciones sobre los dispositivos de la red. Por tratarse de un prototipo, alcanza con que **Goliath** deje una traza visible del Plan de Suministro que le fue enviado como evidencia de su procesamiento.

REQ-5: Mecanismo de integración de aplicaciones (Kremlin)

Desarrollar la aplicación **Kremlin** para resolver la integración e interoperabilidad de las aplicaciones actuales y futuras. Toda aplicación que se registra en Kremlin puede actuar como prestador de servicios y/o consumidor de servicios.

Para cada prestador de servicios se debe poder:

- Registrar una “interfaz de servicio” que permita acceder a su funcionalidad.
- Especificar propiedades sobre la forma de usar cada interfaz (por ejemplo, el formato de las estructuras de datos que se intercambian por la interfaz, o el tipo de comunicación).

Para cualquier consumidor de servicios de debe poder:

- Localizar una interfaz de un servicio registrado
- Facilitar la invocación de las funcionalidades del servicio a consumir

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

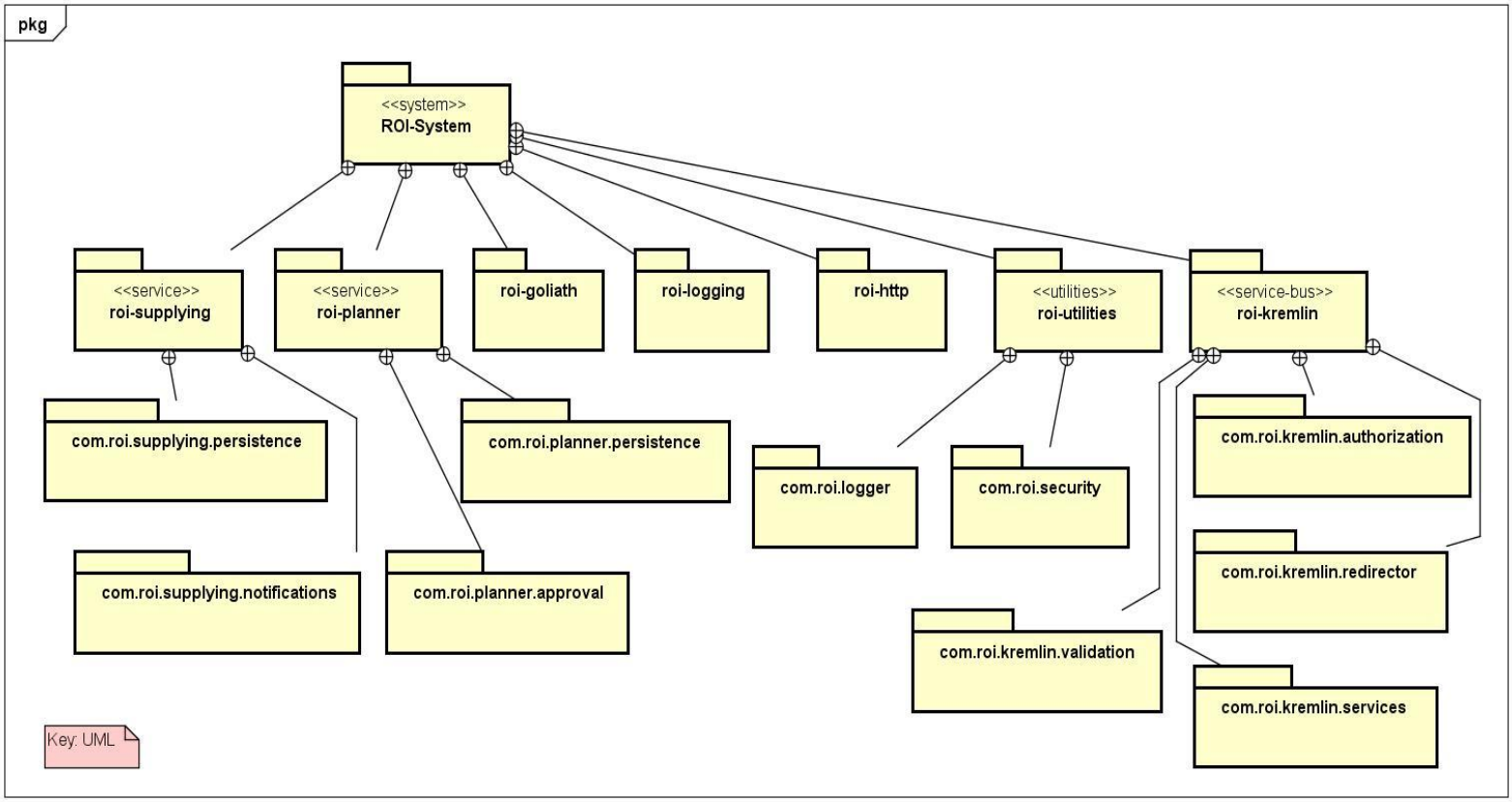
Actualmente, cualquier usuario con acceso a la red informática donde están conectados los servidores de aplicaciones puede invocar libremente cualquier operación expuesta por las API de backend de cualquier aplicación (imagine un usuario ejecutando llamadas http usando Postman o cualquier otra herramienta).

Para toda aplicación “prestadora de servicios” que se registre en **Kremlin**, se requiere que todas sus operaciones publicadas deben ser invocadas únicamente a través de **Kremlin**. Es decir que toda invocación a una operación de un servicio registrado que se realice por fuera de **Kremlin** se debe considerar como no legítima.
De igual forma, toda invocación a **Kremlin** debe provenir de una aplicación “consumidora de servicios” legítima. Por ejemplo, realizar una invocación http por Postman a Kremlin para consumir un servicio debe considerarse como un acceso no legítimo.

3. Vistas de módulos

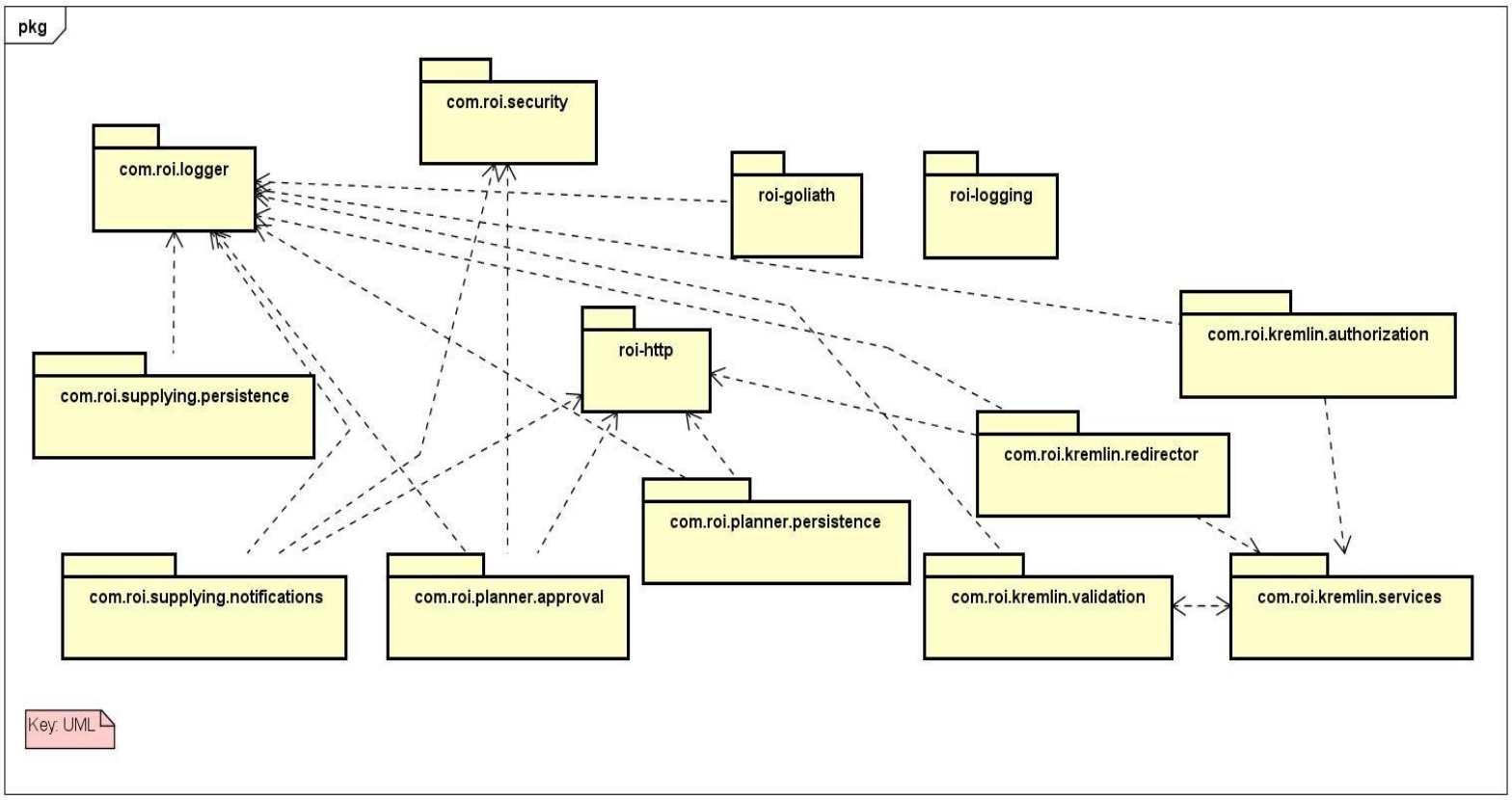
3.1. Vista de descomposición

3.1.1. Representación primaria



3.2. Vista de usos

3.2.1. Representación primaria



3.3. Catálogo de elementos

Elemento	Responsabilidades
ROI-System	Entidad asociada al sistema en su totalidad, como contenedor de las restantes aplicaciones y de la funcionalidad completa a proveerse.
roi-kremlin	Aplicación que actúa, a efectos del sistema, como intermediario entre las restantes. Provee mecanismos de validación de credenciales, y de comprobación y transformación de datos, permitiendo mayor independencia y flexibilidad respecto de la comunicación entre servicios.
roi-supplying	Aplicación encargada de la creación, persistencia, modificación y eliminado de planes de suministro del sistema, así como de notificar variaciones en éstos de corresponder.
roi-planner	Servicio vinculado al almacenamiento de planes de suministro, proveyendo funcionalidades de alta, baja y modificación sobre los mismos.
roi-goliath	Servicio vinculado a la ejecución de un plan de suministro, una vez aprobado este mediante la aplicación *roi-planner*. Ésta, un prototipo inicial, se limita a dejar traza visible de tal situación mediante *logging*.
roi-logging	Módulo encargado de efectuar *logging* de la información reportada por los restantes servicios, con finalidades de auditoría y gestión de errores y fallas.
roi-http	Módulo contenedor de las distintas clases asociadas a la funcionalidad del envío de peticiones HTTP configurables hacia un destino y con un cuerpo y verbo dados.
roi-utilities	Entidad que contiene distintos servicios comunes a las restantes aplicaciones abstraídos en él como forma de generar reúso, en particular vinculados al envío de mensajes de *loggeo*, y el cargado y manejo en memoria de *tokens* de autenticación respecto de *Kremlin*.
roi-logging	Módulo que contiene como principal funcionalidad asociada al procesamiento y registro de mensajes de eventos ocurridos en el sistema, en un archivo de *logging* externo.

com.roi.supplying.persistance	Módulo vinculado a la creación y persistencia de entidades de órdenes de suministro, con operaciones de alta, baja y modificación de las mismas.
com.roi.supplying.notifications	Módulo asociado al envío de notificaciones de la ejecución de una acción sobre una orden de suministro a *Kremlin* en dirección a *roi-planner,* para actualizar su plan asociado.
com.roi.planner.persistance	Módulo vinculado a la creación y persistencia de entidades de planes de suministro, con operaciones de alta, baja y modificación.
com.roi.planner.approval	Vinculado a la funcionalidad de aprobación de plan, marca al mismo como aprobado y lo actualiza, y notifica a la aplicación *Goliath* a través de *Kremlin* para iniciar su ejecución.
com.roi.logger	Módulo contenedor de la funcionalidad asociada al envío de mensajes de *loggeo* a la cola de mensajes correspondiente.
com.roi.security	Módulo encargado de proveer funcionalidad de manejo de *tokens* de sesión de cada aplicación haca *Kremlin.*
com.roi.kremlin.authorization	Módulo vinculado a la validación de un *token* recibido en una determinada petición. Verifica que el mismo sea uno correcto, y que en virtud del mismo se tenga permisos para acceder a la funcionalidad deseada.
com.roi.kremlin.validation	Se encarga de verificar que los datos recibidos en el cuerpo de determinada petición sean correctos; que se hayan recibido la totalidad de los requeridos y que éstos sean de tipo correcto, así como de efectuar las transformaciones de éstos si correspondiera.
com.roi.kremlin.services	Contiene servicios auxiliares vinculados a *Kremlin*, en particular respecto de la inicialización de distintas variables del mismo en base a archivos de configuración al desplegar, y de consultas sobre ellas.
com.roi.kremlin.redirection	Asociado a la redirección a su destino final (el servicio proveedor que correspondiera) de las distintas peticiones intermedias recibidas por *Kremlin*.

3.4. Decisiones de diseño

En cuanto a los diagramas anteriormente presentados, tenemos, en primer lugar, que, a grandes rasgos varios aspectos del diseño presentado en el mismo surgen de restricciones o decisiones tomadas por la empresa en base a su sistema anterior, y al flujo de negocio asociado a la misma. En este sentido, cobran sentido las aplicaciones o servicios *roi-supplying, roi-planner* y *roi-goliath* presentados, además de *roi-kremlin* cuya introducción como forma de introducir indirección en las llamadas entre estas mediante un intermediario, reduciendo el acoplamiento y restringiendo dependencias, es el principal objetivo del presente proyecto.

Analizando el problema un poco más finamente, es posible observar que, dentro de estos grandes servicios de alto nivel, se ven involucradas distintas funcionalidades identificables y separables. En este sentido, se intentó subdividir lo más posible a las mismas en diferentes entidades, aplicando tácticas de incrementar la coherencia semántica y abstraer servicios comunes como forma de mejorar la mantenibilidad del sistema, un atributo de calidad considerado clave a efectos de este obligatorio, que está detrás también de la introducción de *roi-kremlin.* Esa modularización se extendió hasta un nivel de *Enterprise Java Beans* individuales, que intentan encapsular una funcionalidad concreta cada uno de ellos, para luego componerse mediante la inyección de dependencias que ofrece la tecnología para interactuar y lograr así implementar la funcionalidad deseada.

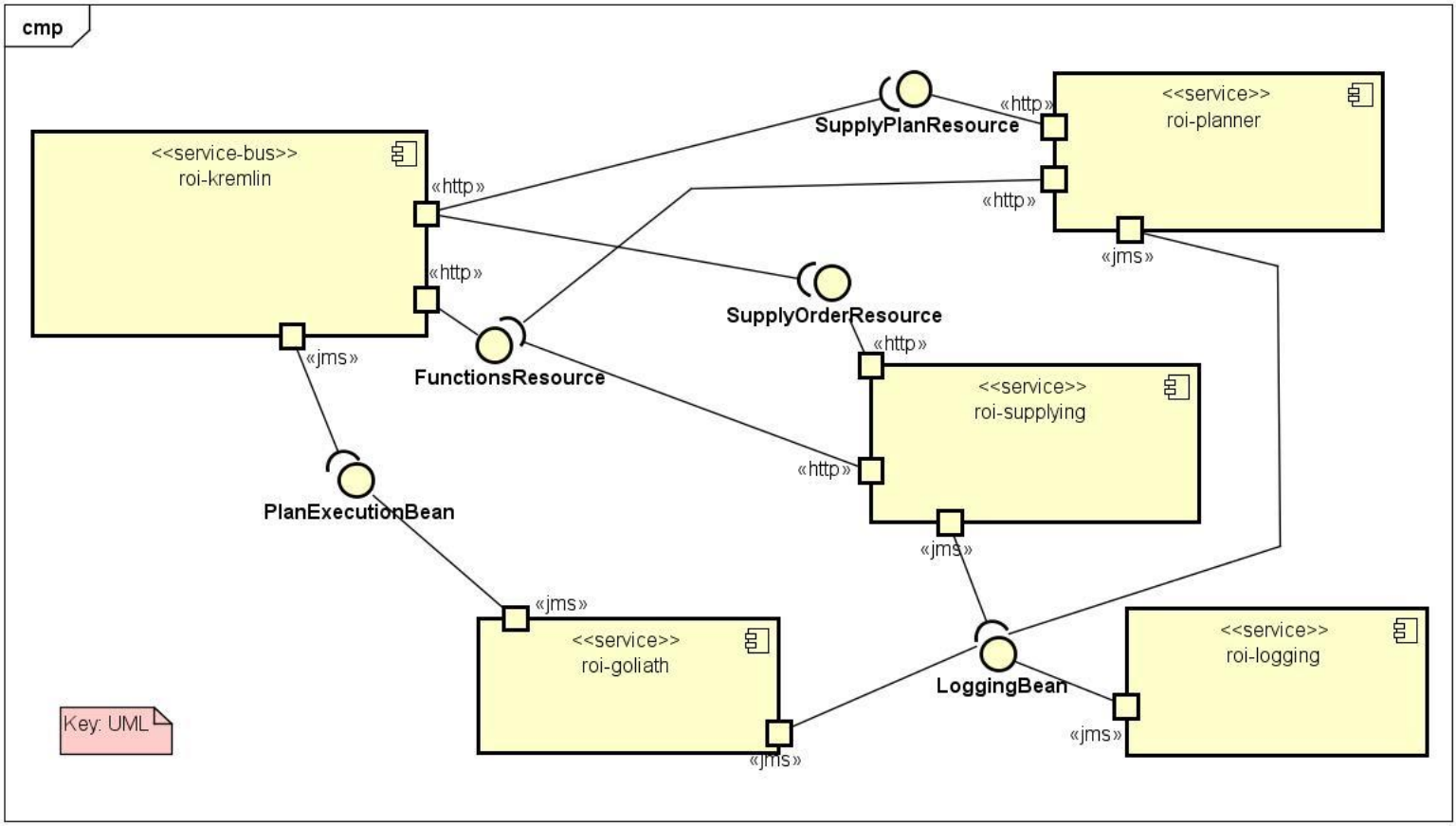
En cuanto a la comunicación entre distintos módulos, se utilizaría más allá de lo anteriormente mencionado, dependiendo de a que aplicación pertenecieran los mismos, mecanismos de comunicación mediante HTTP/REST (formato universal agnóstico del lenguaje e independiente de la tecnología) o mensajes mediante JMS, en particular esto respecto de los servicios *roi-goliath* y *roi-logging*, que como forma de extraer servicios comunes de las aplicaciones y a su vez centralizar en un único punto asincrónicamente la funcionalidad de *loggeo*, se extrajo a una aplicación independiente del resto. Esto se cree mejora la alternativa anterior de utilizar archivos de texto entre aplicaciones en cuanto a *performance,* respecto de los requerimientos REQ-2 y REQ-5.

4. Vista de componentes y conectores

En esta sección se intentará transmitir una visión del sistema en tiempo de ejecución; en particular, los componentes, formas de conexión y la interacción entre componentes para explicar la implementación de funcionalidades o de mecanismos claves. Para ello, se presenta en primer lugar un diagrama de componentes y conectores general al sistema, dividido en distintos servicios independientes por *features* que a cada uno correspondiera, y luego más adelante distintos diagramas que detallan la estructura interna de cada uno de ellos.

4.1. Contexto

4.1.1. Representación primaria



4.1.2. Catálogo de elementos

Componente/Conector	Tipo	Descripción
roi-kremlin	Servicio/Aplicación.	Aplicación que actúa, a efectos del sistema, como intermediario entre las restantes. Provee mecanismos de validación de credenciales, y de comprobación y transformación de datos, permitiendo mayor independencia y flexibilidad respecto de la comunicación entre servicios.
roi-supplying	Servicio/Aplicación.	Aplicación encargada de la creación, persistencia, modificación y eliminado de planes de suministro del sistema, así como de notificar variaciones en éstos de corresponder.
roi-planner	Servicio/Aplicación.	Servicio vinculado al almacenamiento de planes de suministro, proveyendo funcionalidades de alta, baja y modificación sobre los mismos.
roi-goliath	Servicio/Aplicación.	Servicio vinculado a la ejecución de un plan de suministro, una vez aprobado este mediante la aplicación *roi-planner.* Ésta, un prototipo inicial, se limita a dejar traza visible de tal situación mediante *logging*.
roi-logging	Servicio/Aplicación.	Servicio encargado de efectuar *logging* de la información reportada por los restantes servicios, con finalidades de auditoría y gestión de errores y fallas.

4.1.3. Decisiones de diseño

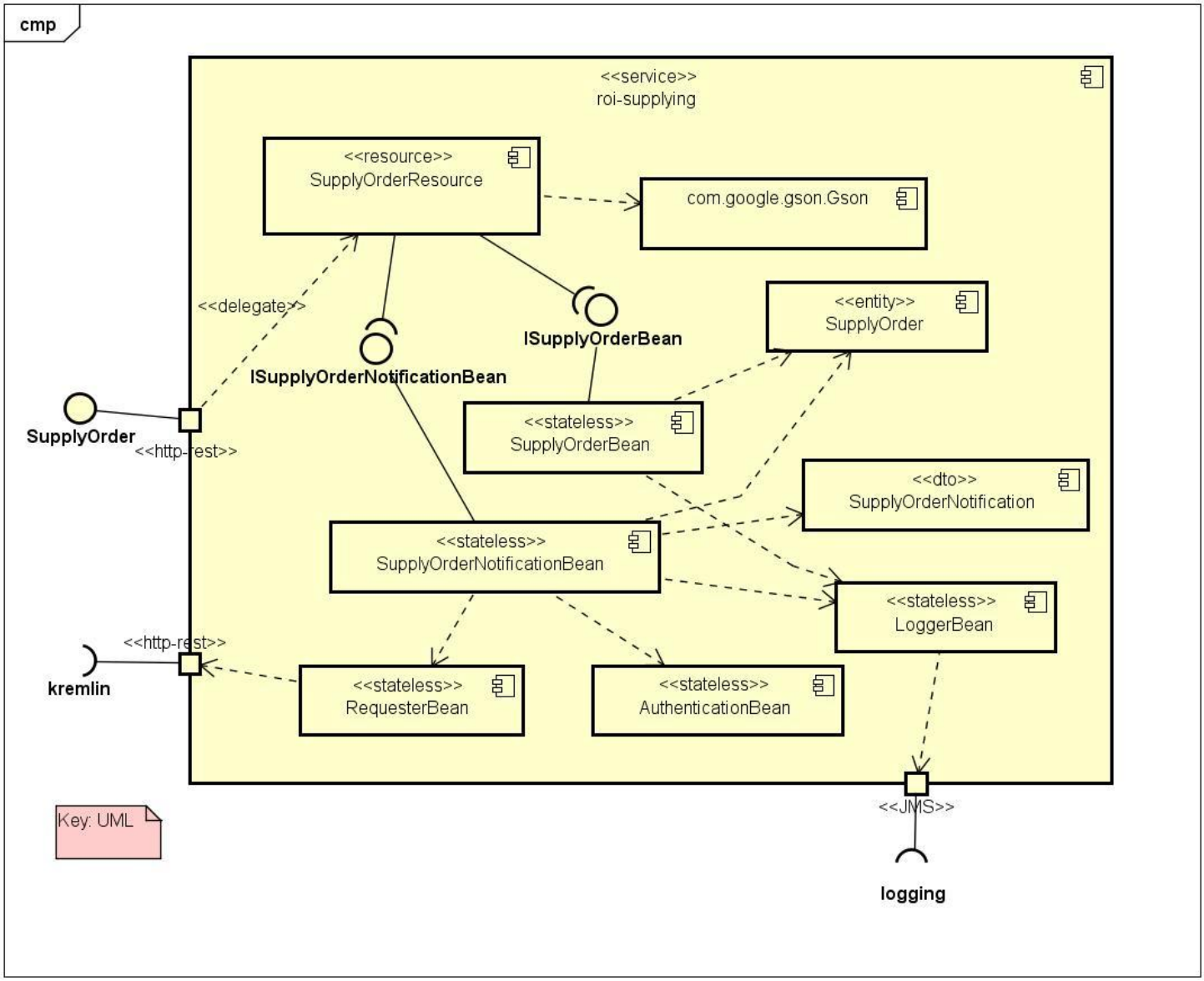
En este apartado en concreto, no se considera exija una justificación demasiado extensa, puesto que la arquitectura a alto nivel presentada anteriormente era ya parte de la propuesta inicial de la empresa, que contaba actualmente con las aplicaciones *roi-supplying, roi-planner,* y *goliath,* y propuso la construcción de una cuarta aplicación *Kremlin* para lograr una mejor, más flexible y segura comunicación entre ellas.

En cuanto a los mecanismos de comunicación elegidos, por otro lado, se decidió que las invocaciones a *Kremlin* se dieran exclusivamente a través de una interfaz http REST. Este es un mecanismo que, de los tres razonablemente disponibles a ser implementados por el equipo de desarrollo (los otros siendo una cola de mensajes mediante *JMS* e interfaces remotas de *JEE*), era el que mayor flexibilidad ofrecía respecto de la comunicación con aplicaciones que en un futuro puedan querer incorporarse al sistema, dado que es este un protocolo estandarizado en la industria que no depende del lenguaje de implementación. Asimismo, permite obtener fácilmente una respuesta frente a una petición, de manera de comunicar al usuario el resultado obtenido frente a determinada acción, cuando la naturaleza asíncrona de *JMS* por ejemplo dificultaría esto, generar una vía de comunicación entre ambas partes en las que se identifique claramente el origen del mensaje.

Con respecto a la aplicación *roi-logging*, esta fue incorporada por varias razones. En primer lugar, era un requerimiento el ofrecer flexibilidad a futuro respecto del mecanismo o librería, *framework*, etc., concreto de *logging* a utilizarse, con lo cual, encapsulándolo en ésta se considera se cumple de forma correcta satisfactoriamente. Esto es, las restantes aplicaciones no efectúan ningún tipo de *loggeo* más allá de escribir en una cola de mensajes, un recurso nativo de la tecnología utilizada (*Java Enterprise Edition*). De esta forma, la aplicación encargada de ello luego lee asíncronamente estos mensajes, dado que en tal funcionalidad no es necesaria ningún tipo de respuesta, no importando por tanto el momento de procesamiento propiamente dicho de estos mensajes. Además, se consideró podría buscarse o ser conveniente esto de ejecutarse ésta en algún tipo de servidor administrativo externo a las restantes aplicaciones, donde se concentren los *logs* de todas ellas, lo que se cree esta solución favorece.

4.2. Vista de componentes y conectores: *roi-supplying*

4.2.1. Representación primaria



4.2.2. Catálogo de elementos

Componente/Conector	Tipo	Descripción
SupplyOrderResource	*Resource*	Contiene los *endpoints* expuestos por la interfaz del componente hacia el exterior, y es el encargado de delegar llamadas hacia componentes encargados de realizarlas.
com.google.gson.Gson	*Class*	Clase auxiliar importada de la librería *gson.jar,* provista por Google. Brinda funcionalidades de parseo automático a y de formato JSON respecto de tipos de objetos.
SupplyOrder	*Entity class*	Clase cuyo esquema de atributos es utilizado para modelar la entidad de órdenes de suministro, vinculada al dominio del problema. Son estas las entidades persistidas por la aplicación en una base de datos relacional para almacenar tales datos.
SupplyOrderBean	*Stateless session bean*	*Bean* responsable del manejo de la lógica asociada a la persistencia en base de datos de las entidades de órdenes de suministro: su creación, borrado y modificación.
SupplyOrderNotification	*Data transfer object class*	Clase modelo utilizada para la transferencia de datos involucrados en la notificación enviada hacia *roi-planner* de la realización de una acción sobre una orden de suministro, para poder ejecutar la acción correspondiente sobre su plan asociado.
SupplyOrderNotificationBean	*Stateless session bean*	*Bean* encargado de la creación y envío mediante una petición HTTP hacia Kremlin de un objeto *SupplyOrderNotification*, generado en base a una orden de suministro válida, conteniendo los datos a ser procesados por *roi-planner.*
RequesterBean	*Stateless session bean*	Entidad en la que, por razones de mantenibilidad asociadas a la

		repetición de código similar en distintas clases, se extrajo la funcionalidad asociada a el envío de peticiones HTTP hacia un destino dado, pudiendo parametrizar las mismas según corresponda.
LoggerBean	*Stateless session bean*	*Bean* donde se extrajo la repetida funcionalidad asociada a la escritura de un mensaje dado hacia la cola utilizada para el *logging* de determinado evento, a ser procesado por la aplicación *roi-logging*.

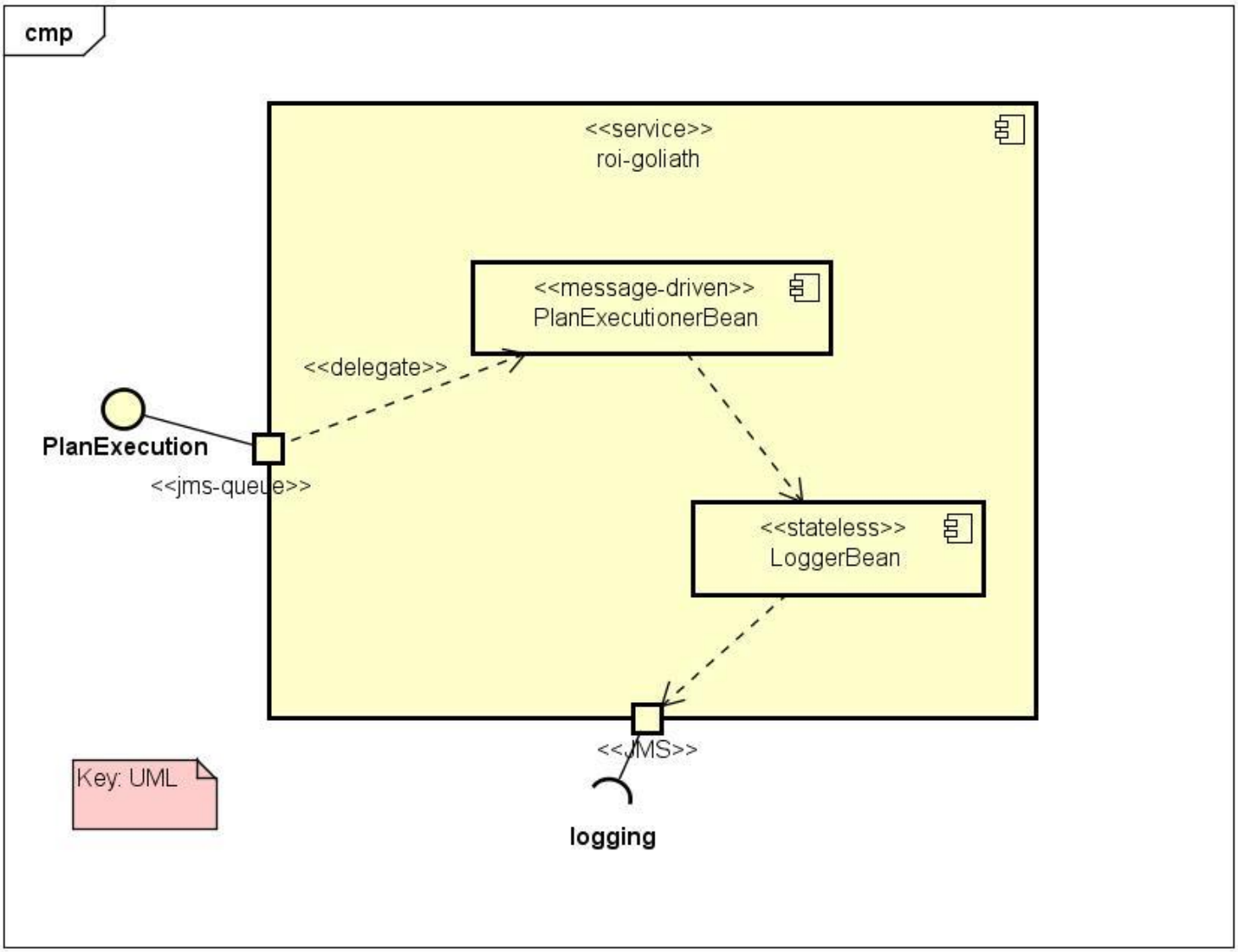
4.2.3. Decisiones de diseño

En cuanto al diseño interno de este componente, este de las distintas aplicaciones desarrolladas, más allá de Kremlin, es una de las más complejas. Tenemos que, a nivel general, las responsabilidades asociadas a la misma son a grandes rasgos dos: el implementar y dar soporte a las operaciones de alta, baja y modificación sobre entidades de planes de suministro, y el notificar de la realización de las mismas a *roi-planner*.

Éstas, como forma de aumentar la cohesión entre clases, y respetar el principio de responsabilidad única, se colocaron en distintos *Stateless session beans*, aumentando entonces la mantenibilidad de la solución. De la misma forma, funcionalidades comunes a distintas partes del sistema, se extraerían a componentes cuya implementación sería compartida por los distintos servicios, teniendo como ejemplos de esto en este caso a *LoggerBean* y a *RequesterBean* (abstraer servicios comunes). Un único *resource* encapsularía los puntos de acceso de las funcionalidades expuestas por el servicio hacia los restantes. Finalmente, se utilizaría a su vez una clase de modelo *data transfer object* como forma de hacer uso del parseo automático que ofrece la librería *Gson*, provista por Google; de un *string* en formato JSON hacia un objeto, y viceversa. De esta forma, la correctitud del *string* generado es asegurada con mínimo esfuerzo.

4.3. Vista de componentes y conectores: *roi-goliath*

4.3.1. Representación primaria



4.3.2. Catálogo de elementos

Componente/Conector	Tipo	Descripción
PlanExecutionerBean	*Message-driven session bean*	*Bean* encargado de efectuar la ejecución de un plan de suministro registrado mediante *roi-planner*, mediante actuadores, válvulas y sensores que correspondan. La versión entregada es un prototipo que se limita a *loggear* el recibimiento de un plan.
LoggerBean	*Stateless session bean*	*Bean* donde se extrajo la repetida funcionalidad asociada a la escritura de un mensaje dado hacia la cola utilizada para el *logging* de determinado evento, a ser procesado por la aplicación *roi-logging*.

4.3.3. Decisiones de diseño

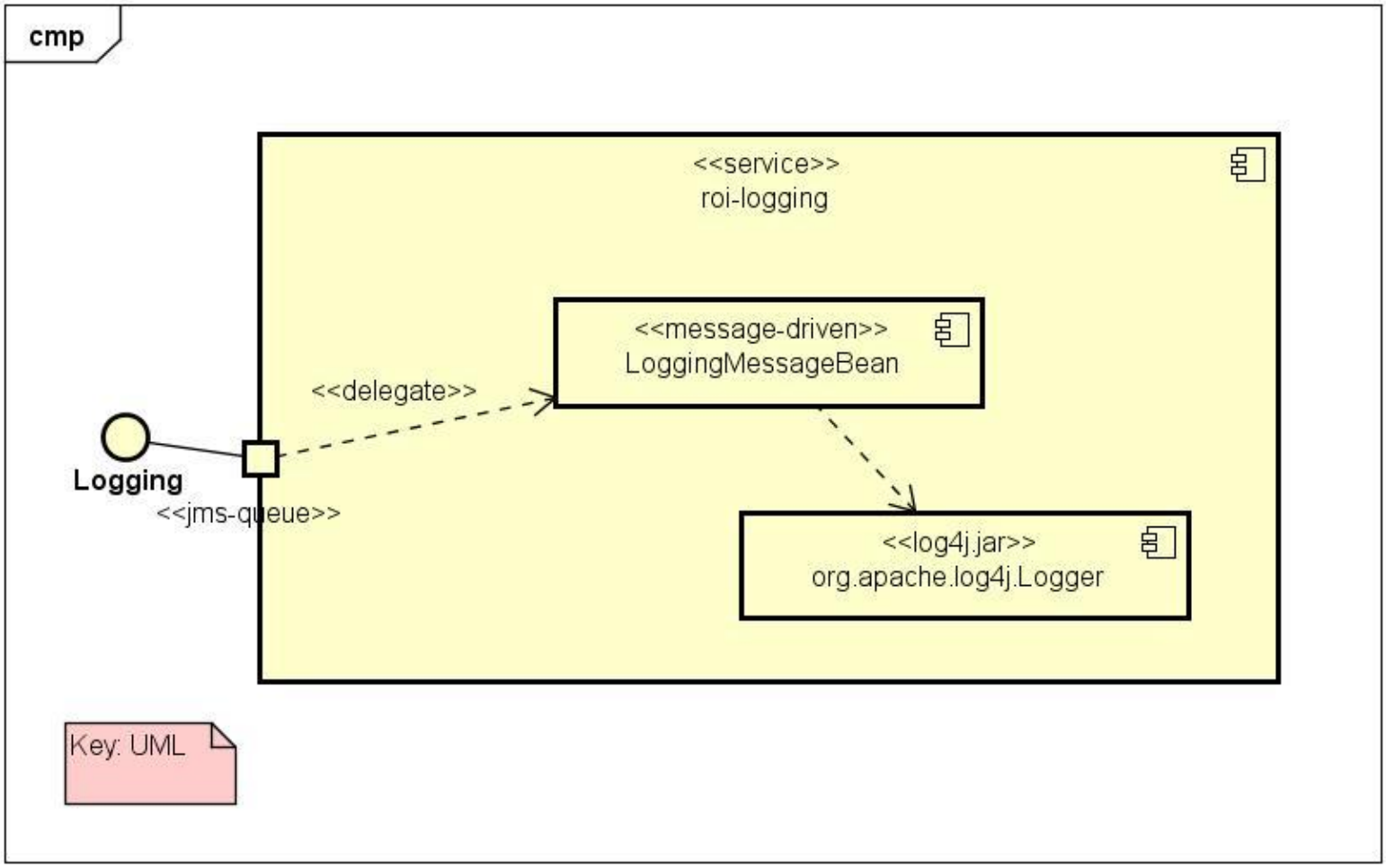
En cuanto a aclaraciones correspondientes al diseño interno de este componente, en primer lugar, ya en apartados anteriores se ha aclarado los motivos detrás de la generación de la clase *LoggerBean,* como forma de extraer y reutilizar código asociado a la funcionalidad común del *loggeo* de determinados eventos del sistema. Asimismo, se utilizaría en esta clase a la librería *Gson,* una vez más, como forma de simplificar el parseo desde un *string* en formato JSON, utilizándose para ello clases modelo *SupplyPlan* y *NetworkFrame,* habiéndose compuesto en una colección la segunda dentro de la primera, parseándose la primera al haberse reconocido el formato JSON esperado correctamente por la librería.

Quizás el punto interesante radica, en este caso también, en la elección del mecanismo de comunicación elegido para este servicio, utilizándose JMS en lugar de una API REST, en esta ocasión. Esto se debe a dos motivos, principalmente. En primer lugar, era una restricción del proyecto de desarrollo el incorporar dos mecanismos de comunicación distintos en las aplicaciones a implementarse, lo cual se logró de esta forma. Asimismo, fue elegida esta, de entre las tres, y JMS en oposición a interfaces remotas, por ejemplo, debido a que se consideró quizás el funcionamiento interno de la aplicación *Goliath* real, en oposición al prototipo sumamente básico generado, podría llegar a ser algo muy complejo, debiéndose activar una compleja red de actuadores, sensores, válvulas, etc.

Así, el introducir un componente asincrónico, como lo son las colas de mensajes por naturaleza, permitiría desacoplar una llamada realizada al servicio de su realización y procesamiento concreto, pudiéndose realizar a futuro de ser algo muy costoso en tiempo de ejecución, o estar sobrecargado el sistema, etc., sin esperar una llamada o retorno del mismo. Claro que esto podría haber sido hecho en forma interna a *Goliath*, pero como se contaba con la restricción anteriormente detallada, se movió a la comunicación entre *Kremlin* y *Goliath.*

4.4. Vista de componentes y conectores: *roi-logging*

4.4.1. Representación primaria



4.4.2. Catálogo de elementos

Componente/Conector	Tipo	Descripción
LoggingMessageBean	*Message-driven session bean*	*Bean* encargado del procesamiento de los distintos mensajes para *loggeo* de eventos del sistema enviados por las restantes aplicaciones. Se limita simplemente a leerlos, reconstruir la información en ellos contenida, y llamar a la librería utilizada con este fin.
org.apache.log4j.Logger	*Class*	Clase externa utilizada para hacer efectivo el registro de eventos del sistema. En este caso, fue configurada mediante un archivo *.properties* para que el mismo se realice en un archivo externo en formato HTML, para lo cual brinda funcionalidades.

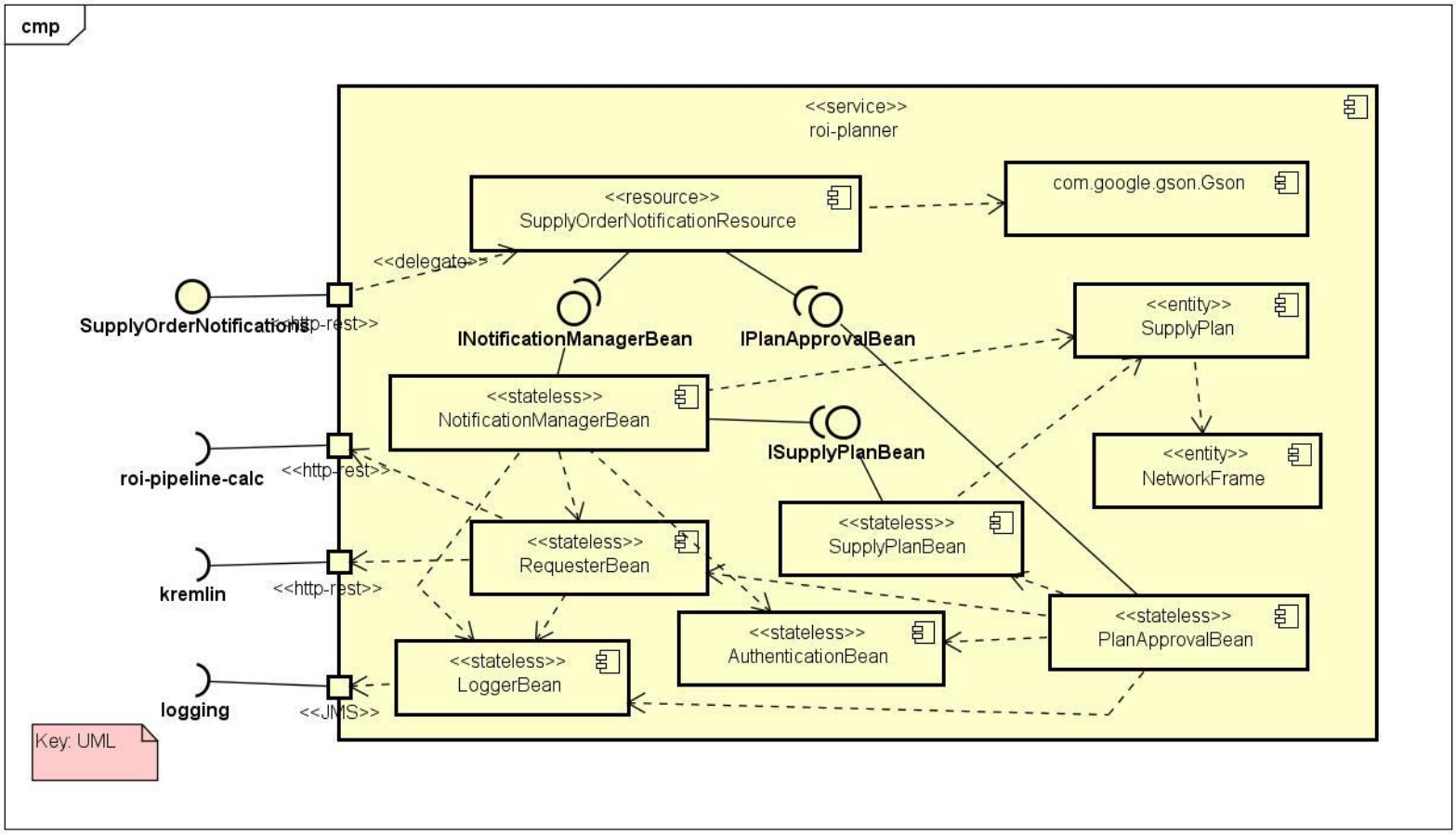
4.4.3. Decisiones de diseño

El diseño de este componente pasa, en primer lugar, por la decisión anteriormente mencionada de intentar desacoplar la funcionalidad concreta de *loggeo* del resto del sistema, como forma de centralizar mensajes que pudieran originarse en distintos equipos de despliegue, introducir asincronismo en funciones que lo permiten además de evitar funcionalidad y código repetido, a raíz de lo cual esta aplicación fue concebida. Para esto, se utilizó una cola de mensajes, necesitándose entonces, naturalmente, un componente que lea de la misma y procese los mensajes que en ella se encuentra.

Este procesamiento se daría simplemente llamando a la librería utilizada, que no sólo provee ampliamente las funcionalidades requeridas para esto, sino que además esta es configurable a futuro mediante un archivo *.properties*, brindando mayor flexibilidad en caso de desearse cambiar el formato o mecanismo de *loggeo* concreto a futuro. A su vez, dado que esta es llamada únicamente desde el *bean* anteriormente descrito, *LoggingMessageBean*, el costo o impacto de cambio respecto del cambio de la librería en sí a futuro se minimiza, siendo posible entonces desarrollar otro *bean* o utilizar otra aplicación que lea de la cola de mensajes utilizada, lo cual era un requerimiento del sistema (REQ-6).

4.5. Vista de componentes y conectores: *roi-planner*

4.5.1. Representación primaria



4.5.2. Catálogo de elementos

Componente/Conector	Tipo	Descripción
SupplyOrderNotificationResource	*Resource*	Contiene los *endpoints* expuestos por la interfaz del componente hacia el exterior, y es el encargado de delegar llamadas hacia componentes encargados de realizarlas.
com.google.gson.Gson	*Class*	Clase auxiliar importada de la librería *gson.jar,* provista por Google. Brinda funcionalidades de parseo automático a y de formato JSON respecto de tipos de objetos.
NotificationManagerBean	*Stateless session bean*	*Session bean* encargado del procesamiento concreto de notificaciones enviadas desde *roi-supplying* sobre la realización de una operación sobre una orden de suministro, para ejecutar la acción correspondiente sobre el plan asociado.
SupplyPlan	*Entity class*	Clase modelo asociada a la entidad del dominio del problema “plan de suministro”, a ser manejada y almacenada por el sistema.
NetworkFrame	*Entity class*	Entidad vinculada a un plan de suministro, respecto de la ruta de actuadores, etc., asociada al mismo, guardándose una colección de objetos de la misma.
SupplyPlanBean	*Stateless session bean*	*Bean* responsable del manejo de la lógica asociada a la persistencia en base de datos de las entidades de planes de suministro: su creación, borrado y modificación.
RequesterBean	*Stateless session bean*	Entidad en la que, por razones de mantenibilidad asociadas a la repetición de código similar

		en distintas clases, se extrajo la funcionalidad asociada a el envío de peticiones HTTP hacia un destino dado, pudiendo parametrizar las mismas según corresponda.
AuthenticationBean	*Stateless session bean*	*Bean* asociado al manejo de *tokens* de autenticación por parte de las distintas aplicaciones que deben a su vez consumir servicios de *Kremlin*. Estos son mediante esta clase leídos de archivos de configuración.
PlanApprovalBean	*Stateless session bean*	*Session bean* vinculado a la funcionalidad de la aprobación de un plan de suministro, marcándolo como tal y enviando al mismo a *Kremlin* en dirección a *Goliath* para su ejecución.
LoggerBean	*Stateless session bean*	*Bean* donde se extrajo la repetida funcionalidad asociada a la escritura de un mensaje dado hacia la cola utilizada para el *logging* de determinado evento, a ser procesado por la aplicación *roi-logging*.

4.5.3. Decisiones de diseño

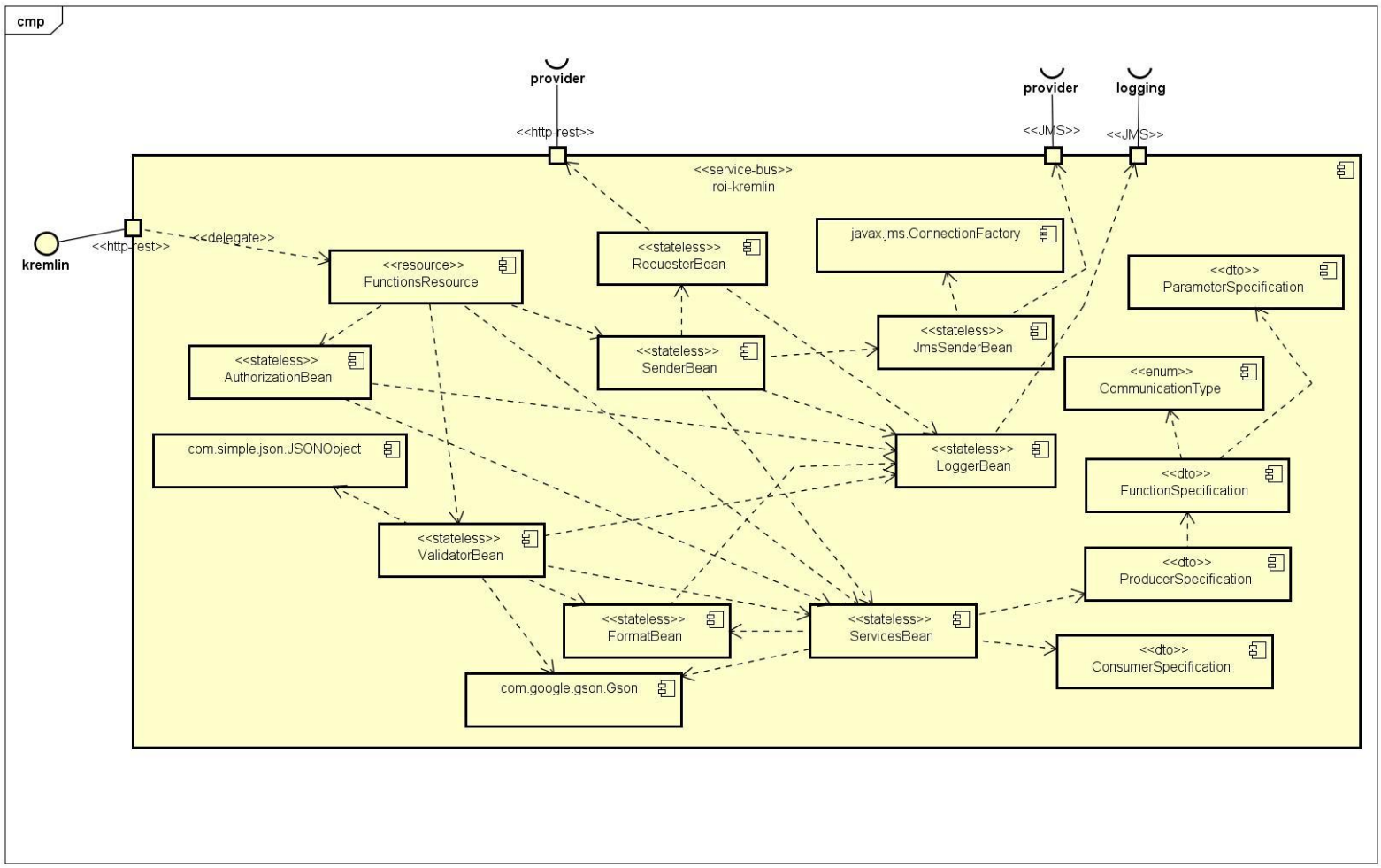
Con respecto a esta aplicación proveedora de servicios, al igual que ocurría en *roi-supplying*, esta debe soportar a grandes rasgos dos conjuntos de funcionalidades: el manejo de las distintas operaciones de alta baja y modificación de entidades de planes de suministro, y la aprobación de los mismos, que implica una complejidad adicional en la comunicación con *roi-goliath* a través de *Kremlin*. Se intentó, al igual que en el caso anteriormente mencionado, separar este conjunto de responsabilidades en la forma más unitaria posible de forma de aumentar la coherencia semántica y disminuir el acoplamiento, favoreciendo la modificabilidad a futuro. Se generaron entonces dos *session beans, SupplyPlanBean* y *PlanApprovalBean,* encargados de manejar esta lógica.

Nuevamente, al igual que en *RequesterBean* y *LoggerBean*, en el caso de *AuthenticationBean* se intentó abstraer servicios comunes a las distintas aplicaciones y así favorecer la mantenibilidad y modificabilidad a futuro eliminando código asociado a una misma funcionalidad repetida. Se recurriría también a la librería *Gson* para efectuar el parseo de y hacia el formato JSON por la sencillez que esta ofrece, como se ha mencionado anteriormente, a las clases de entidades manejadas.

El punto de interés que ofrece este diagrama, particularmente, es quizás la aparición de la interfaz externa requerida *roi-pipeline-calc.* Se decidió, en cuanto a esto, accederla directamente desde el servicio *roi-planner*, en lugar de pasar por *Kremlin*. El objetivo de éste es permitir flexibilidad a futuro en cuanto al manejo de las operaciones del sistema, extendiéndolas o facilitando el cambio de servicio proveedor, desacoplando las llamadas internas entre los mismos al actuar de intermediario. Sin embargo, se consideró que esta en particular, al ser una API externa a lo que es el sistema, vinculada a una operación en concreto que es hoy por hoy responsabilidad única de *roi-planner* como lo es la creación y registro de planes de suministro, esta podía ser accedida directamente por ella, puesto que de desearse a futuro modificarse la forma en la que los planes son creados correspondería este impacte en esta aplicación y no en *Kremlin*, cuyas responsabilidades no pasan estrictamente por ahí.

4.6. Vista de componentes y conectores: *roi-kremlin*

4.6.1. Representación primaria



4.6.2. Catálogo de elementos

Componente/Conector	Tipo	Descripción
FunctionsResource	*Resource*	Contiene el *endpoint* por el que se accede a las distintas funciones de los proveedores de servicios.
com.google.gson.Gson	*Class*	Clase auxiliar importada de la librería *gson.jar,* provista por Google. Brinda funcionalidades de parseo automático a y de formato JSON respecto de tipos de objetos.
javax.jms.ConnectionFactory	*Class*	Permite obtener una conexión con recursos (*Queues*, particularmente) mediante instancias de clases de JMS.
ParameterSpecification	*Data transfer object*	Clase modelo asociada a un parámetro de determinada función.
CommunicationType	*Enum*	Entidad que modela los distintos tipos de conexión soportados por Kremlin; en este caso, JMS y REST.
SenderBean	*Stateless session bean*	Reenvía solicitudes recibidas por *Kremlin* hacia su destino final.
RequesterBean	*Stateless session bean*	Entidad en la que, por razones de mantenibilidad asociadas a la repetición de código similar en distintas clases, se extrajo la funcionalidad asociada a el envío de peticiones HTTP hacia un destino dado, pudiendo parametrizar las mismas según corresponda.
AuthorizationBean	*Stateless session bean*	*Bean* encargado de la validación de las credenciales en una solicitud recibidas, así como de las mismas cuenten con autorización para acceder a ella.

FunctionSpecification	*Data transfer object*	Modelo de la declaración de una función provista por un servicio.
LoggerBean	*Stateless session bean*	*Bean* donde se extrajo la repetida funcionalidad asociada a la escritura de un mensaje dado hacia la cola utilizada para el *logging* de determinado evento, a ser procesado por la aplicación *roi-logging*.
ServicesBean	*Stateless session bean*	Vinculado a la obtención de la especificación de los distintos servicios manejados de un archivo de configuración.
FormatBean	*Stateless session bean*	Permite realizar cierta transformación de datos, hoy por hoy principalmente vinculadas al formato de fechas.
ValidationBean	*Stateless session bean*	Permite la validación de la correctitud los parámetros potencialmente requeridos por determinada función al ser invocada.
com.simple.json.JSONObject	*Class*	Clase de una librería auxiliar utilizada que permite el manejo de entidades JSON genéricas.
ProducerSpecification	*Data transfer object*	Modelo asociado a los datos de un proveedor de servicios.
ConsumerSpecification	*Data transfer object*	Modelo asociado a los datos de un consumidor de servicios.
JmsSenderBean	*Stateless session bean*	*Bean* auxiliar que permite el envío de mensajes mediante JMS hacia proveedores de servicios que utilicen este tipo de comunicación.

4.6.3. Decisiones de diseño

Se buscó separar las responsabilidades lo más posible . Por ello mismo, se tomó cómo guía las funcionalidades principales de buscadas para un ESB, comúnmente abreviado VETO, Validar, Enriquecer, Transformar y Operar.
Al igual que en las restantes aplicaciones, el principio guía sería la creación de varios *beans* con responsabilidades bien definidas, que intentan abordar cada una de estas en forma unitaria, de acuerdo con el principio de responsabilidad única. Es en este sentido que las responsabilidades de la aplicación *Kremlin* se encuentran distribuidas en una serie de *session beans,* lo cual favorece la mantenibilidad del servicio al haberse incrementado la cohesión entre los componentes internos al mismo, creándose un flujo de trabajo entre estas distintas clases.

Como punto de interés en cuanto a este componente, por otro lado, tenemos el que el cargado de los diferentes proveedores de servicios que maneja, así como el procesamiento de la información que en invocaciones a funciones le llega, etc., se da enteramente en formato JSON. Formato bastante estandarizado a nivel de la industria, se observa, en esta instancia de contemplación luego de haber desarrollado el sistema, que varias partes del mismo se encuentran fuertemente acopladas al mismo, además de a la librería *Gson* que se utiliza en varias partes del mismo.

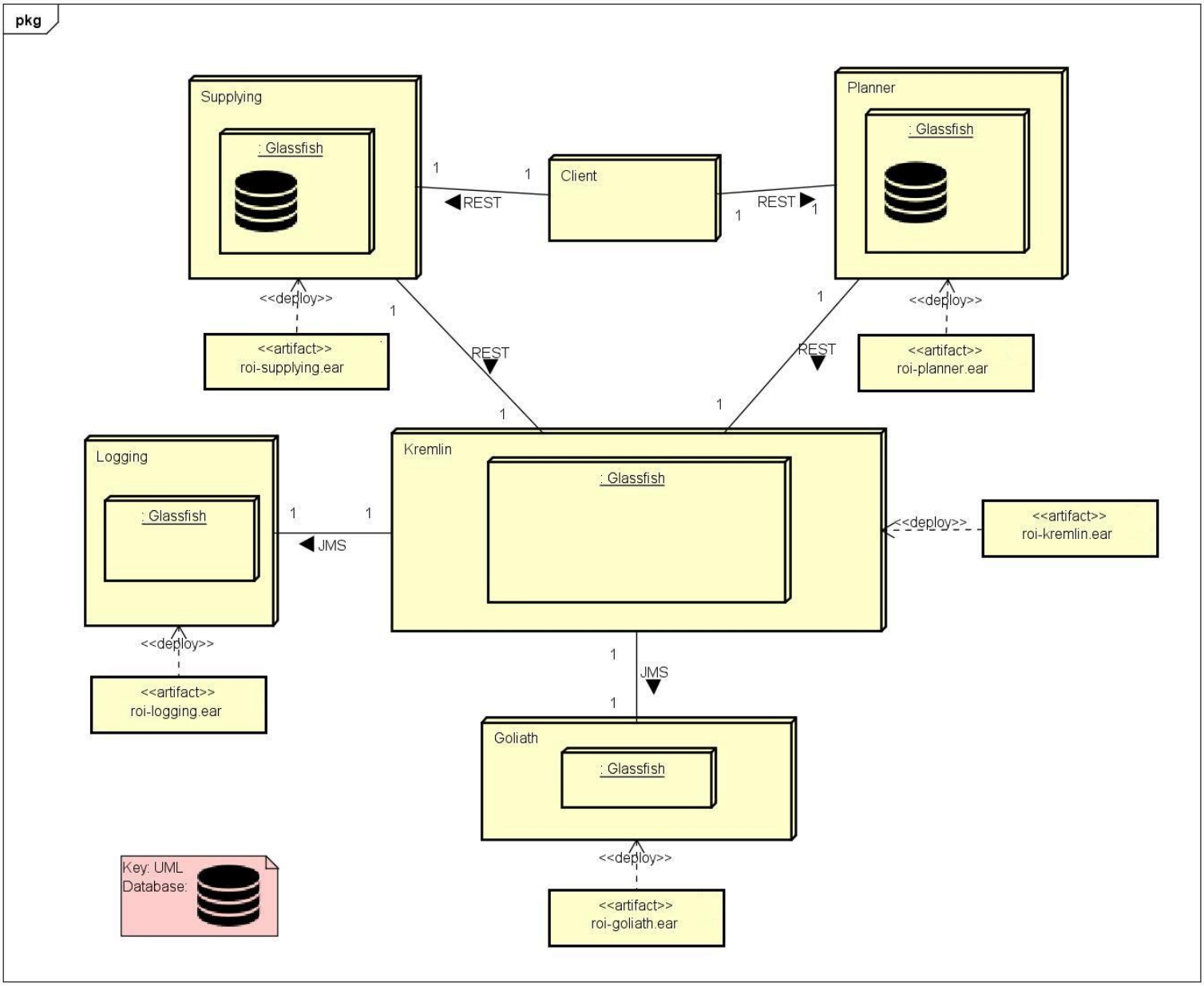
La decisión de utilizar archivos de configuración en esta aplicación fue examinada con detenimiento, y será analizada más adelante. En cuanto al formato del mismo, como se mencionó anteriormente, se realizó en base a la simplicidad y la familiaridad que se tenía con la mencionada librería, pero podría ser un punto futuro para mejorar el aceptar parámetros de entrada con otros formatos (XML, etc.), más allá del formato de los archivos de configuración internos que no se verían afectados por el formato en el que envíen peticiones clientes. Sería necesario en tal caso introducir alguna especie de adaptador de distintos tipos de entradas para poder manejar estos datos, así como tal vez un componente que encapsule a la librería *Gson* para limitar las dependencias de la misma.

A pesar de ello, se considera que la funcionalidad requerida por el requerimiento REQ-5, en cuanto a la introducción de un intermediario como forma de desacoplar llamadas internas entre distintos servicios, se encuentra implementado de forma razonablemente exitosa, aunque con ciertas limitaciones que serán analizadas más adelante.

5. Vistas de asignación

5.1. Vista de despliegue

5.1.1. Representación primaria



5.1.2. Catálogo de elementos

Nodo	Descripción
Client	Cliente externo al sistema que busque acceder a las funcionalidades expuestas por el mismo, mediante algún tipo de cliente HTTP (como podría ser *Postman*, por ejemplo).
Supplying	Nodo de ejecución de la aplicación/servicio *roi-supplying* mediante un servidor de aplicaciones *Glassfish*, con su base de datos asociada.
Planner	Nodo de ejecución de la aplicación/servicio *roi-planner* mediante un servidor de aplicaciones *Glassfish,* con su base de datos asociada.
Goliath	Nodo de ejecución de la aplicación/servicio *roi-goliath* mediante un servidor de aplicaciones *Glassfish.*
Kremlin	Nodo de ejecución de la aplicación/servicio *roi-kremlin* mediante un servidor de aplicaciones *Glassfish.*
Logging	Nodo de ejecución de la aplicación/servicio *roi-logging* mediante un servidor de aplicaciones *Glassfish.*

Artefactos	Descripción
roi-supplying.ear	Artefacto de instalación o despliegue de la aplicación/servicio *roi-supplying*.
roi-planner.ear	Artefacto de instalación o despliegue de la aplicación/servicio *roi-planner*.
roi-goliath.ear	Artefacto de instalación o despliegue de la aplicación/servicio *roi-goliath*.
roi-kremlin.ear	Artefacto de instalación o despliegue de la aplicación/servicio *roi-kremlin*.
roi-logging.ear	Artefacto de instalación o despliegue de la aplicación/servicio *roi-logging*.

5.1.3. Decisiones de diseño

En cuanto al diagrama anteriormente presentado, se tiene que, en primer lugar, la empresa actualmente cuenta con un sistema que se ejecuta en un servidor centralizado, buscando la misma cambiar tal arquitectura por una que incorpore un intermediario entre las llamadas a distintos servicios, el *service bus Kremlin.* Así, se buscó adoptar para este proyecto algo que intenta aproximar a una arquitectura orientada a servicios, en la que distintas aplicaciones proveedoras de servicios se publican en una aplicación central (*Kremlin*), con la que interactúan distintos clientes que busquen acceder a determinada funcionalidad, redirigiéndolo este entonces al proveedor apropiado.
Esta permite que cada servicio se ejecute en un nodo independiente, comunicándose estos mediante canales apropiados, que es lo que se encuentra representado en el diagrama anterior. A efectos del proceso de desarrollo y de la posterior defensa de esta entrega, sin embargo, tal ejecución se limitó a un único nodo, el servidor *Glassfish* embebido que provee el entorno de desarrollo.

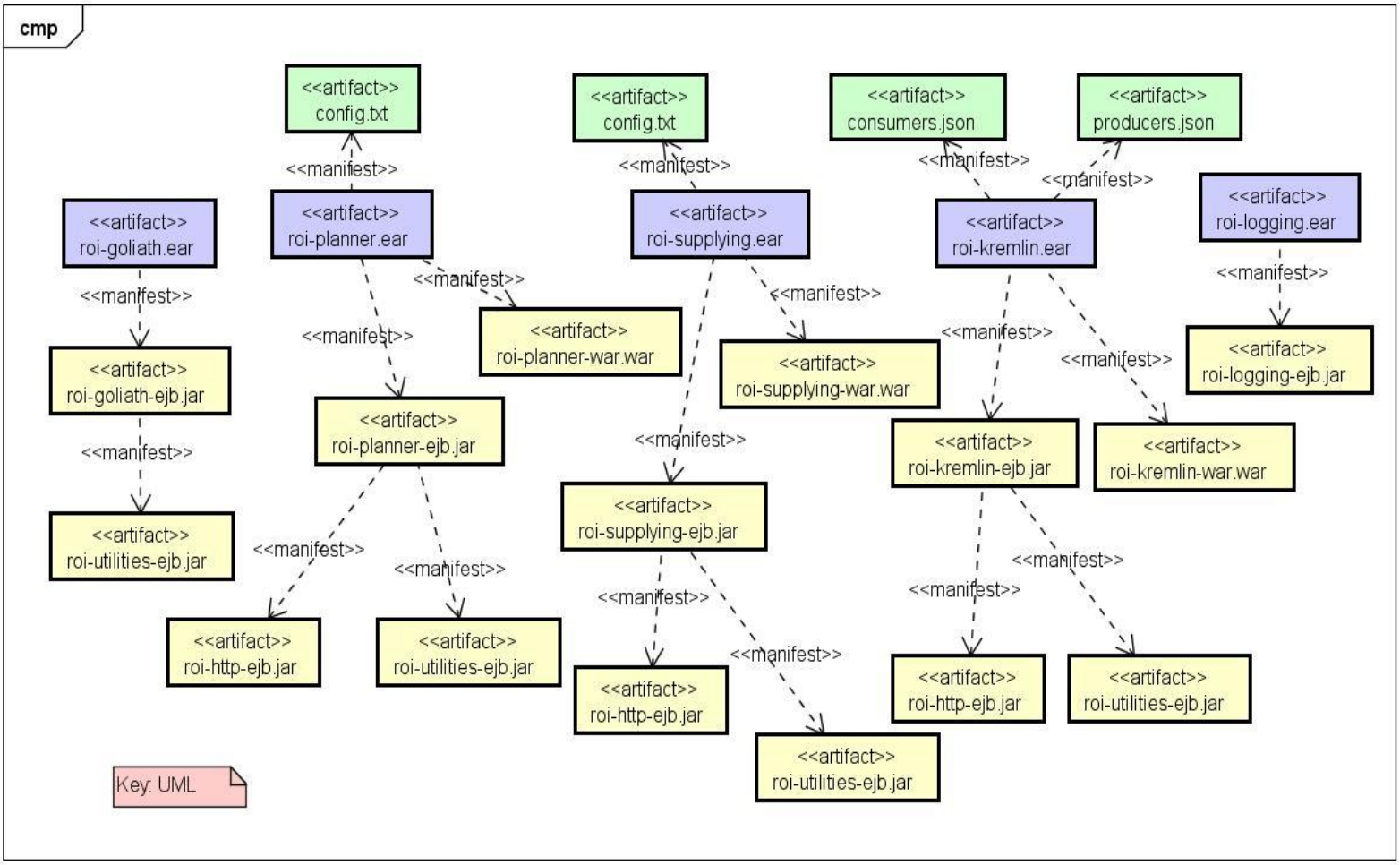
Un punto interesante de discusión en este respecto es la colocación de los puntos de acceso que tienen clientes externos al sistema en el diagrama anterior. Como se observa, en el mismo estos acceden a través de *endpoints* que se encuentran presentes en las aplicaciones *roi-supplying* y *roi-planner.* Esto es debido a que Kremlin, concretamente, expone un *endpoint* genérico común a las distintas acciones del sistema, que puede resultar algo confuso acceder a él desde un cliente externo, y que recibe a su vez un conjunto de información que resultaría innecesaria en tal caso, cuando menos, puesto que sería necesario ignorar cierta lógica de validación de *tokens* en ciertos casos que hoy por hoy se hace automáticamente en todos. En este caso, asimismo, no se estaría respetando el estándar REST al exponer las funcionalidades del sistema.

Otra opción que se evaluó fue la de generar una aplicación cliente que expusiera los *endpoints* visibles desde el exterior, mediando ella entonces con *Kremlin.* Esto finalmente se decidió no hacer, además de porque agregaba cierta complejidad tal vez innecesaria al momento del desarrollo, debido a que tal aplicación debiera ser modificada cada vez que se busque desplegar un nuevo servicio en *Kremlin* para poder acceder a él externamente, aumentando por tanto el impacto de cambio del mismo y atentando contra el objetivo de aumentar la modificabilidad a futuro del sistema en este sentido, por tanto.

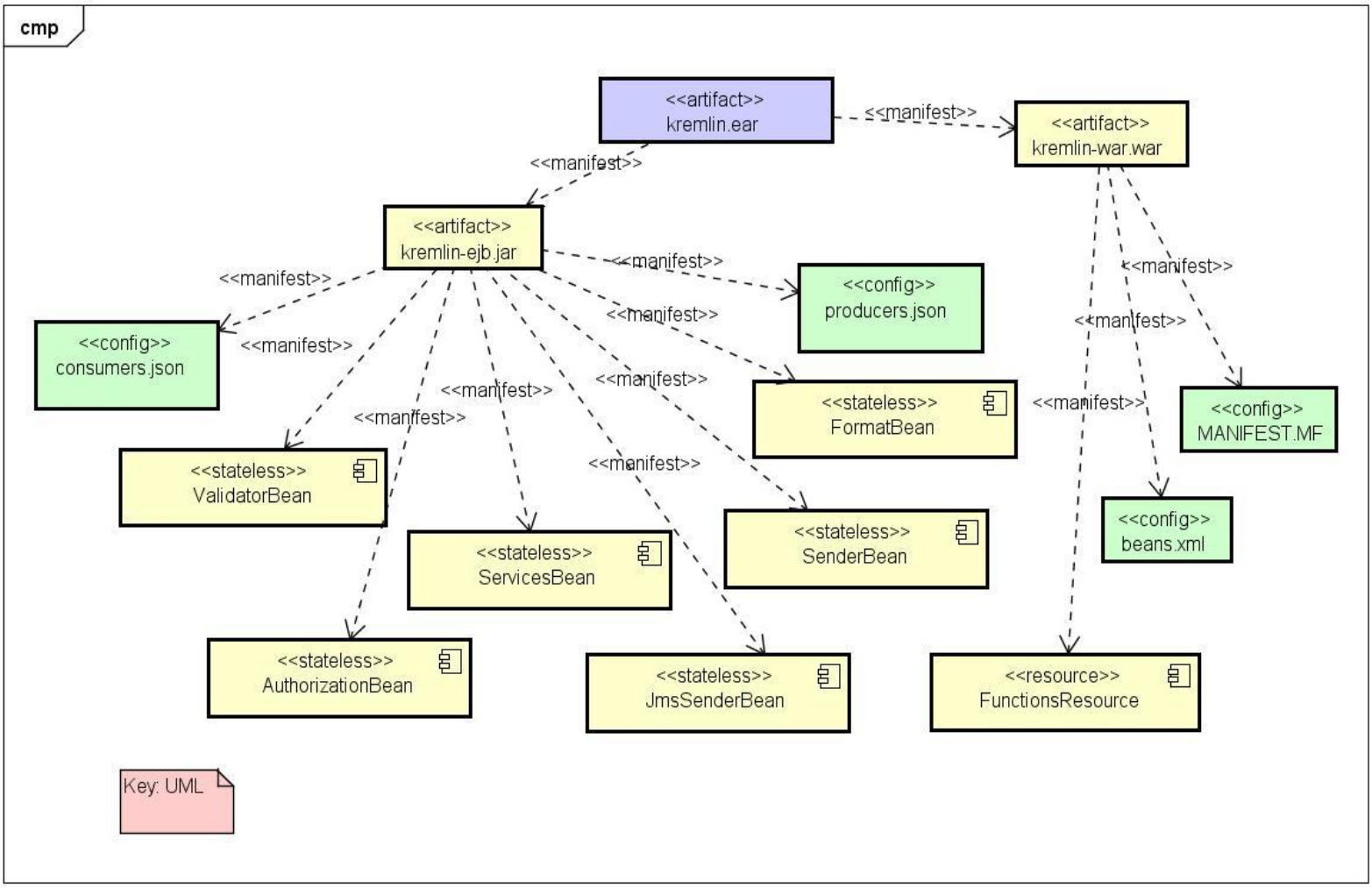
5.2. Vista de despliegue

5.2.1. Representación primaria

5.2.1.1. General



5.2.1.2. *roi-kremlin*



5.2.2. Catálogo de elementos

Artefactos	Descripción
**.ear*	Artefacto de instalación o despliegue de la aplicación/servicio correspondiente.
**-ejb.jar*	Artefacto contenedor de la implementación de las distintas clases y *session beans* que logran llevar a cabo la funcionalidad propia del servicio, así como librerías de las que se haga uso.
**.war*	Artefacto donde se encuentran principalmente los distintos *resources* que exponen *endpoints* vinculados a una aplicación que se comunique mediante HTTP/REST.
MANIFEST.MF	Archivo de configuración interno a la tecnología utilizada.
beans.xml	Archivo de configuración necesario para poder acceder a *session beans* desde un proyecto *war* en *Java Enterprise Edition.*
**Bean*	*Session bean* vinculada a determinada funcionalidad concreta en el servicio/aplicación correspondiente.
config.txt	Archivo de configuración contenedor del *token* de acceso de la aplicación correspondiente a ser validado en cada consulta enviada hacia *Kremlin*, así como la URL de este.
producers.json	Archivo de configuración que especifica los distintos proveedores de servicios presentes en el sistema, junto con las funciones que estos ofrecen, las características de los datos que reciben, su mecanismo de comunicación y las aplicaciones consumidoras autorizadas para consumirlas.
consumers.json	Archivo en formato JSON que detalla la información asociada a cada aplicación consumidora de servicios presente en el sistema: su *token* de acceso, y un identificador para la misma.

5.2.3. Decisiones de diseño

Como puntos a aclarar respecto de los diagramas anteriores, en primer lugar, vemos que, en líneas con lo anteriormente mencionado de intentar subdividir al dominio del problema en aplicaciones/servicios independientes entre sí, según lo manifestaba como requerimiento la propuesta de este, se observa el objetivo sería desplegar cada una de ellas en artefactos separados. Esto es, como se vio, en caso de buscarse a futuro desplegar cada una de ellas en nodos de ejecución independientes, o necesitar a futuro modificarse una de ellas sin perder funcionalidad en las restantes, o reusarse según lo plantea el principio de equivalencia reúso-liberación. Podría permitir esto también extender horizontalmente los recursos del sistema, desplegando varias instancias de una misma aplicación de detectarse problemas de *performance* sin mayores inconvenientes.

Con respecto al formato de los artefactos en sí, vinculados a un proyecto del tipo *Java Enterprise Application* en *NetBeans*, originalmente el proceso de desarrollo se dio trabajando en módulos EJB (*EJB Modules*) y aplicaciones Web (*Web applications*) independientemente, pero esto trajo consigo ciertos problemas de dependencias entre los mismos, debiéndose incorporar esta en ambos, que fue resuelta de esta manera, teniéndose como única unidad de liberación entonces un archivo *.ear.* Esto se estima correcto, dado que resulta razonable desplegar en un nodo de ejecución ambas en conjunto, los resources que atienden peticiones, así como los *session beans* que implementan la funcionalidad de los mismos.

Finalmente, como comentario breve respecto los archivos de configuración en formato JSON anteriormente mencionados, se exploraron diversas opciones respecto al mecanismo concreto con el cual la funcionalidad de *Kremlin* sería implementada, decantándose el equipo de desarrollo por esta. Estos detallan la información necesaria para efectuar la autenticación de un cliente, la validación de la correctitud de los datos que pudieran recibirse, y la redirección hacia el proveedor del servicio correspondiente, encontrándose esta centralizada en *Kremlin* completamente, debiéndose modificar estos archivos de buscar agregarse nuevos servicios, etc. Las implicancias de esto serán analizadas en mayor detalle en secciones posteriores.

6. Consideraciones adicionales

6.1. Modo de registrar proveedores y consumidores

La decisión fue bastante meditada dado el impacto que tiene en la construcción del prototipo. Se manejaron cuatro opciones, cada una con sus ventajas y desventajas.

Mecanismo	Ventajas	Desventajas	Decisión
Endpoint para registrar consumidores y proveedores. Un administrador (con credenciales) crea las especificaciones “a mano” y las envía.	• No se debe reiniciar Kremlin para incorporar funciones, consumidores o cambios de especificación.	• Se tendría que implementar junto con un mecanismo de autenticación que impida que cualquiera pueda registrar consumidores o productores y asignar los permisos.	✗
Generar endpoints dinámicamente con alguna herramienta (ej.: *Swagger*).	• Provee más mantenibilidad. • Menos trabajo para incorporar nuevos proveedores.	• Solución demasiado compleja para el *scope* del prototipo. • Puede que no cubra todos los tipos de comunicación utilizados.	✗
Archivo de configuración en *Kremlin* que especifique las aplicaciones consumidoras y productoras.	• Si se tiene familiaridad con el formato se puede trabajar más rápido. • Ofrece más flexibilidad a la hora de diseñar la especificación. • Fácil de implementar.	• Se debe conocer el formato de especificación para trabajar con el archivo correctamente. • Se es más propenso a cometer errores al editar el archivo. • Se debe reiniciar Kremlin cada vez que se actualiza el archivo de configuración o éste debe ser capaz de escuchar los cambios en el archivo. • Cambios en especificación de las	✓

		aplicaciones proveedoras me impactan en Kremlin (debo editar; el impacto de cambio es mayor porque tengo que ir a cambiar Kremlin porque pueda haber cambiado alguna de las otras aplicaciones).	
Endpoint para registrar consumidores y productores. Donde cada aplicación proveedora tiene un archivo de configuración con su especificación. Al cambiar esta especificación, envía nueva especificación al endpoint. Kremlin actualiza la especificación para dicho proveedor.	• Cambios en las aplicaciones proveedoras no afectan a Kremlin.	• Aplicaciones precisan contar con alguna clave para demostrar que están autorizadas a realizar cambios de especificación en Kremlin. • Se debe escuchar cambios de archivo en cada aplicación.	✗

6.2. Seguridad

Sistema de permisos:

Permiso para consumir un servicio registrado

No por ser una aplicación consumidora habilitada para hablar con *Kremlin*, es decir con un token válido, significa que soy capaz de consumir todas las funciones de los servicios registrados. A modo de ejemplo: la aplicación Planner no puede hacer uso de las funcionalidades que Goliath expone.

Se consideraron dos posibles mecanismos para lograr esto:

Mecanismo	Consideraciones	Decisión
En archivo que se listan aplicaciones consumidoras(“*consumers.json*”) se asignan permisos sobre quien tienen permitido consumir.	Es más fácil de agregar permisos a nuevos consumidores. Ofrece menos detalle al elegir permisos.	✗
En cada función que un proveedor específico se indica que aplicaciones consumidoras tienen permiso para llamarla.	Ofrece más granularidad. Cada función explícitamente dice quién puede consumirla. Es así que proveedor puede elegir no tener su totalidad de funciones expuesta a un consumidor. Facilita agregar nuevas funciones.	✓

6.3. Implementación de configuración

Se meditaron varias opciones a la hora de decidir cómo se iban a guardar datos de configuración en el sistema:

Nombre	Características a tomar en cuenta	Decisión
Registro de Windows	• No es portable ni demasiado simple de usar.	✗
Apache Commons Configuration	• Implica nuevas dependencias. • Acepta varios formatos de persistencia (XML, *documents*, JNDI, JDBC) • Soporta *configuration*	✗

	listeners para notar cambios en configuración.	
Preferences API	• Portable: usa registro de windows en windows y ad-hoc *file-based scheme* en Unix. • Ofrece un mínimo de *type* safety: se puede usar tipos numéricos y booleanos. • Difícil de editar con editor/herramienta externa.	✗
Archivo XML	• *Xpath* está por defecto en el JDK. • *XML Schema*: para validación de tipos y estructuras de datos. • No tan legible	✗
Archivo JSON	• Más legible y menos “cargado” que XML • Requiere agregar dependencias.	✓
Java.util.Properties	• Es fácil de instanciar y usar. • Se puede editar con cualquier editor de texto fácilmente. • No hay type safety (solo acepta *strings*) por lo que si se requiere convertir a otro tipo	✓

	se debe hacer dentro de la aplicación.	

En nuestro sistema contamos con archivos de configuración que tienen distintos fines.

Para los archivos simples de configuración ("config.txt") se utilizaron las Properties de Java, dado que al no haber cosas complejas para guardar nos resultaba conveniente y simple.

Para los archivos de especificación ("*producers.json", "consumers.json"*) se utilizó formato JSON ya que requiere guardar tipos algo complejos. Para parsear los archivos se hizo uso de la reconocida librería Gson de Google.

Para más información acerca de esta decisión se referencia un artículo que detalla algunas de las opciones.[2]

6.4. Diseño de configuración

6.4.1. Configuración de consumidores

Para los consumidores se pensó en un archivo en formato JSON *("consumers.json"*) ubicado dentro de Kremlin con el siguiente formato:

```
[
    {
        "name": "planner",
        "token": "a5ea3b8c-620c-432a-a442-98fde0990182"
    },
    {
        "name": "supplying",
        "token": "bb9cc201-247a-422a-b515-c097b4eb3853"
    }
]
```

Aquí están las aplicaciones consumidoras. Se les llama consumidoras porque que llaman a kremlin por una funcionalidad de las que antes fue registrada por alguna aplicación proveedora.

[2]: http://www.injoit.org/index.php/j1/article/viewFile/33/24

Las aplicaciones consumidoras al comunicarse con kremlin siempre deben proveer su token en el *header* "*Authentication*" de forma de identificarse y autenticarse.

El proceso para registrar una aplicación consumidora nueva es el siguiente:

1. Se genera un UUID como token.
2. Se agrega el nombre de la aplicación y su token al archivo "consumers.json" de kremlin.
3. La nueva aplicación guarda su token en un archivo de configuración("config.txt") para que sea fácilmente accesible al realizar peticiones y modificable en caso de que se desee.

6.4.2. Configuración de proveedores

Para los proveedores se pensó en un archivo en formato json ("producers.json") ubicado dentro de Kremlin con el siguiente formato:

```
[
    {
        "name": "planner",
        "functionSpecifications": [
            {
                "name": "createdSupplyOrder",
                "location": "POST%http://planner/web-resources/supply-order/{id}",
                "type": "WEB_REST_API",
                "parameters": [
                    {
                        "name": "id",
                        "type": "Long"
                    },
                    {
                        "name": "orderNumber",
                        "type": "Long"
                    },
                    {
                        "name": "servicePointId",
                        "type": "Long"
                    }
                ],
                "accessibleBy": [
                    "supplying"
                ]
            },
            {...},
            {...},
            {...}
        ]
    },
    {...}
]
```

En este archivo se detallan las distintas aplicaciones proveedoras. Se les llama proveedoras porque proveen funciones a ser llamadas por las aplicaciones consumidoras.

A continuación, explicamos cómo cada una especifica sus funciones.
Una función se compone por:

Nombre(“*name*”)
Para luego ser capaz de llamarla.

Tipo de conexión(“*type*”)
Indica cómo quiere que nos comuniquemos con ella. El día de hoy se soporta JAVA_MESSAGE_QUEUES y WEB_REST_API.

Ubicación(“*location*”)
Detalla según el tipo especificado los datos de conexión.

Si se trata de JAVA_MESSAGE_QUEUES:
Indica el nombre de la cola de mensajes. Ejemplo: “jms/roiGoliathQueue”

Si se trata de WEB_REST_API:
Indica el método HTTP y la *uri*. De la siguiente forma “METODO%URL”
Ejemplo: “PUT%http://planner/web-resources/supply-order”.
En caso de que la hayan *path params* se especifican entre corchetes({PARAM_NAME}).
Ejemplo: “POST%http://planner/web-resources/supply-order/{id}”.

Consumidores permitidos(“*accessibleBy*”)
Se utiliza para la validación de permisos, se listan los nombres de las aplicaciones permitidas para consumir.

Parámetros que recibe(“*parameters*”)
Detalla los distintos parámetros que espera.
Para cada parámetro se detalla: nombre, tipo y formato(si amerita).
Ejemplo:

```
{
    "name": "supplyStart",
    "type": "Date",
    "format": "yyyy/mm/dd"
},
```

No se validan los tipos anidados que son *Collections* ni tipos anidados que Kremlin no conozca (como es de esperar). Esto no sería difícil de implementar en un futuro siempre que se especifique bien el tipo en la descripción del mismo.

Notar que no se soportan parámetros opcionales, todos los especificados son obligatorios. Si se desea esto en un futuro, basta con agregar un campo

booleano “*optional*” que pueda ser *true* o *false*. De tal forma, si un parámetro es opcional Kremlin (*ValidatorBean*) ignoraría que no sea provisto.

Para registrar una aplicación proveedora nueva simplemente se debe agregar su especificación a *“producers.json”.*

6.4.3. Configuración de aplicaciones consumidoras

Las aplicaciones consumidoras precisan de información vital para llamar a las funciones de los proveedores. Ellos son:

- Token: para identificarse y autenticarse con Kremlin.
- *URL* de Kremlin: para poder llamar al endpoint de Kremlin quien va a llamar a la funcionalidad deseada.

Ejemplo:

```
1  token=a5ea3b8c-620c-432a-a442-98fde0990182
2  kremlinUrl=http://kremlin.com/web-services/functions/
```

6.4.4. Endpoint de servicios expuestos en Kremlin

En los ESB se estila que haya un servicio donde se devuelvan las funciones registradas en el mismo.
En un sistema cambiante sería deseable que la respuesta sea de tal completitud que las aplicaciones consumidoras podrían dinámicamente crear sus llamadas.
En nuestro sistema, al ser tan pequeño como es basta con que este servicio devuelva las funciones registradas por los proveedores de forma que sea útil para el desarrollador de una aplicación consumidora en tiempo de desarrollo.

6.5. Decisiones particulares

Planner recibe todas las notificaciones posibles y decide si recalcular o no plan:

El dominio requiere que, al crear, modificar o eliminar una orden de suministro roi-planner sea notificado y realice las acciones correspondientes. Puede ser necesario que recalcule el plan de suministro o no, dependiendo de que fue modificado de la orden.

Por el momento roi-planner solo le importan los cambios de orden que cambien el identificador de punto de servicio, ya que en base a eso calcula la ruta del plan (es el único parámetro que le pasa al pipeline-calculator).
Considerando que en un futuro el pipeline necesite de más parámetros para calcular una ruta, como puede ser el volumen contratado, se necesitara cambiar el plan en más casos. Por eso se decidió colocar la lógica de recalcular el plan en planner, y recibir todas las notificaciones sobre una orden, aunque no se cambie el identificador de punto de servicio. Es así que planner decide si es necesario recalcular o no. Lo que tiene sentido siendo el encargado de los planes.

A continuación, a modo de aclaración, aunque debe resultar evidente para un arquitecto experimentado se explica porque puede haber clases de dominio o DTO repetidos en aplicaciones distintas.
Ejemplificamos con el caso de supplying y planner. Ambos cuentan con una clase SupplyOrderNotification. En Supplying, esta clase se utiliza para notificar al Planner los cambios en órdenes de abastecimiento. Mientras que en Planner se utiliza para recibir la información de notificación. Bajo ningún concepto se puede incluir SupplyOrderNotification en una biblioteca común a ambas aplicaciones. Esto va en contra a la idea del ESB. SupplyOrderNotification (sea de planner o supplying) tiene total libertad de cambiar, Kremlin sería quien ajusta los datos a lo esperado por el proveedor.

7. Mejoras a futuro

Además de implementar opciones más complejas que sean adecuadas para una aplicación en producción (no un prototipo) hay ciertas mejoras que podrían implementarse.

Por ejemplo, antes de pedir un cálculo de plan al *pipeline-calculator* se podría verificar que esté levantado, dado que tiene un *endpoint* para ello.

Así como las aplicaciones consumidoras se registran en Kremlin y acuerdan un token. Lo análogo debería pasar con Kremlin y las aplicaciones proveedoras. Es decir, siempre que Kremlin se comunique con los proveedores debe incluir un token que antes fuera acordado. De esta manera nos aseguramos de que solo Kremlin se comunique con los proveedores.