

Aditi for Developers

The programming reference
for the Aditi platform v.2.0.0



Inspired by

nature,

developed

by

Mithikel

discover more: mithikel.com

Introducing Aditi to Business

©2020 Information Critical Solutions All rights reserved

This is reference material of the Aditi platform. Its purpose is to explain and disseminate its operation and applications. It cannot be used for purposes other than the above, distributed or modified without prior authorization from Mithikel Technology. For any inquiries, please contact us at official@mithikel.



mithikel.com

official@mithikel.com

Este es el material de referencia para desarrollar sistemas críticos de información en versión **2.0.0** plataforma Aditi.

Nomenclatura usada en el documento:



En estos recuadros presentaremos **conceptos clave** de la plataforma Aditi.



En estos recuadros presentaremos **recomendaciones para el diseño y desarrollo**.



En estos recuadros presentaremos **advertencias** a cuestiones confusas o susceptibles de equivocaciones.



En estos recuadros presentaremos **definiciones**.



Contenido

Parte I: Introducción

1. ¿Qué es Aditi?	01
2. ¿Cómo se desarrolla un sistema Aditi?	06
3. ¿Cómo funciona la Aditi Cell?	07
4. Protocolos de Comunicación	09
5. Protocolo de Autenticación	13
6. Protocolo de Sincronización	16
7. Protocolo de Autorrecuperación	18
8. Esquemas de Seguridad	21
9. ¿Qué son los Artefactos?	22

Parte II: Desarrollo

1. ¿Cómo crear un Recurso?	23
○ <i>Ejemplo 1:</i> Recurso Hola Mundo	26
○ <i>Ejemplo 2:</i> Recurso Ventana JFrame	29
○ <i>Ejemplo 3:</i> Recurso AditiLogger	33
2. ¿Cómo crear un Servicio?	38
○ <i>Ejemplo 4:</i> Servicio Hola Mundo	41
3. ¿Cómo acceder a Recursos desde Servicios?	44
○ <i>Ejemplo 5:</i> Servicio Hola Mundo con Recurso	45
○ <i>Ejemplo 6:</i> Servicio Hola Mundo con Aditi Logger	48



Contenido

Parte II: Desarrollo

4. ¿Cómo solicitar Servicios?	51
○ Ejemplo 7: Servicio Properties (NTx)	54
○ Ejemplo 8: Servicio Properties (Tx)	62
5. ¿Cómo acceder a los Atributos de Entrada?	69
○ Ejemplo 9: Servicio PingPong	70
○ Ejemplo 10: Servicio PrintProperties	73

Parte III: Implantación y Operación

1. ¿Cómo funciona el Aditi Cellman?	76
2. ¿Cómo usar el Aditi Manager?	77
3. ¿Cómo añadir Artefactos?	78
4. ¿Cómo crear Ambientes?	79
5. ¿Cómo modelar Aditi Cells?	81

Parte IV: Demos Completas .

1. Calculadora Aditi	86
○ Análisis	86
○ Diseño	88
○ Desarrollo	89
○ Implementación	99



PARTE I:

INTRODUCCIÓN



¿Qué es Aditi?



- Aditi es una plataforma para **diseñar, administrar y operar sistemas de información críticos**; es decir, aquellos sistemas que al no estar disponibles por actualizaciones, fallas de software o de hardware o cualquier otra razón, generar un gran impacto económico, social o de reputación.
- La plataforma Aditi está integrada por los siguientes elementos:



Aditi Cell

Es el componente de software que proporciona funcionalidad de microservicio en aplicaciones desarrolladas con tecnología ADITI.



Aditi Net

Es una red de alta disponibilidad formada por nodos. Permite el envío de mensajes entre las Aditi Cell.



Aditi Manager

Es el componente que permite administrar la plataforma ADITI y los sistemas desarrollados con Aditi.



Aditi Cell Add-ons

Son componentes de software que permiten la integración de diferentes tecnologías, como bases de datos y servicios web, con la tecnología ADITI.

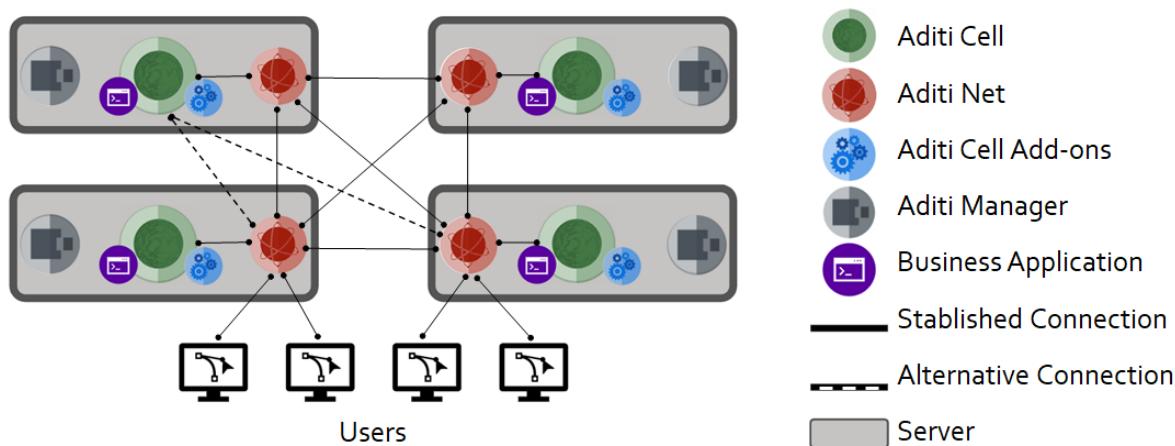


La plataforma Aditi tiene dos características principales:

- Es **autónoma** porque sus componentes se **controlan y coordinan** por sí mismos.
- Es **descentralizada** porque **elimina** los **puntos únicos de falla**.

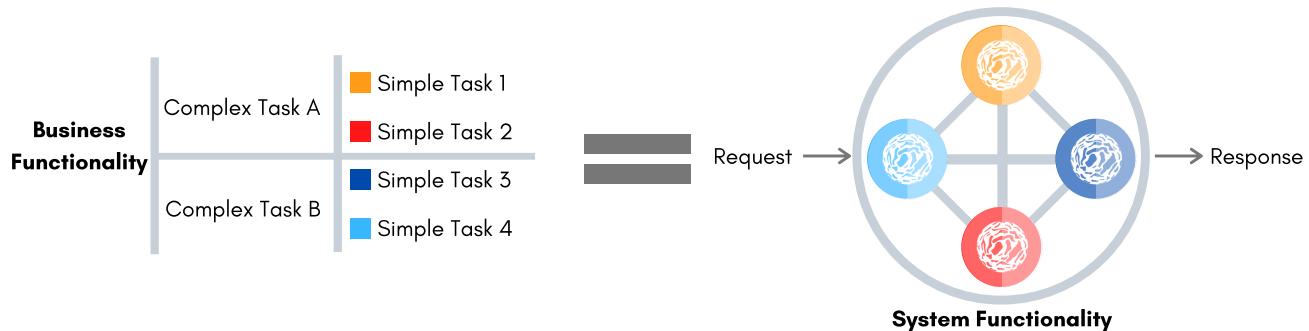
- Como podrás notarlo en los nombres de los elementos, Aditi está **inspirada** en el **comportamiento celular** de los seres vivos.
- Cada tipo de célula presente en un organismo vivo tiene una forma, tamaño y distribución acorde a la tarea específica que realiza. Conforme más complejo sea el organismo, tendrá mayor diversidad celular y tendrá mayor número de funciones.
- De manera similar, en la plataforma Aditi los sistemas críticos de información son integrados por *Aditi Cells*, aplicativos parecidos a pequeños servidores de aplicaciones distribuidos que replican las funcionalidades de las células de los organismos vivos.

Modelo conceptual de la plataforma Aditi



- Las Aditi Cell se conectan a una instancia de Aditi Net para intercambiar solicitudes de servicio y respuestas con el resto del sistema. En caso de que se pierda esa conexión, Aditi Cell tiene un protocolo para buscar una conexión disponible con alguna otra instancia de Aditi Net.
- Los Aditi Cell Add-ons se pueden incorporar a las Aditi Cells para realizar tareas más especializadas, como conectarse a bases de datos o utilizar interfaces gráficas de usuario.
- Además, las Aditi Cell están conectadas al Aditi Manager, que permite monitorear su estado y modificar los servicios y los Add-ons que contienen.

- El diseño de los sistemas de información en Aditi parte de dos principios. El primero de ellos es la **división funcional**: la funcionalidad general de negocio se divide sucesivamente en funciones más sencillas hasta obtener servicios especializados que serán ofrecidos y solicitados por las células.



- La división consta de cuatro etapas. De la primera se obtienen los **negocios** del sistema. En la segunda etapa, los negocios se dividen en **sistemas**. A continuación, los sistemas se dividen en **entidades** que equivalen a los tipos de células que conformarán al sistema. Por último, se determina qué servicios en concreto debe ofrecer cada Aditi Cell.
- Cada negocio, sistema y entidad debe tener un valor numérico propio que lo distinga del resto. Si en sistema crítico consta de dos negocios, uno debe ser identificado con el número 1 y otro con el 2. Si a su vez el negocio 1 se divide en dos sistemas, uno tendrá el identificador 1 y otro el 2; si el negocio 2 consta únicamente de un sistema, este tendrá el identificador 1. Lo mismo aplica para las entidades de cada sistema.



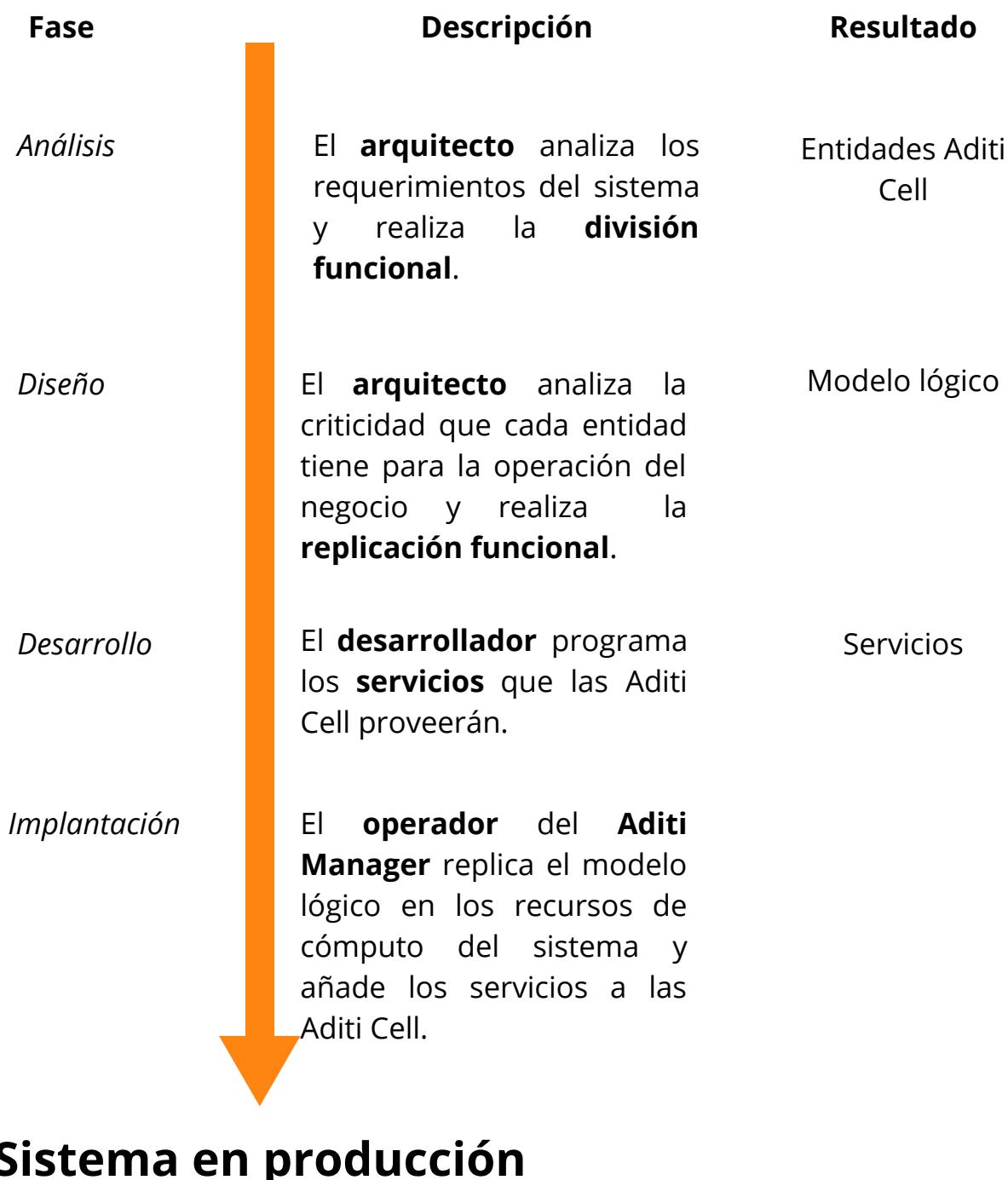
Tip:

Para obtener las células del sistema también es válido aplicar un enfoque *bottom-top*: comenzar conociendo los servicios del sistema y agruparlos en entidades que hagan funciones similares; después agrupar las entidades similares en sistemas orientados a determinado tipo de actividad; finalmente, agrupar los sistemas en negocios.

- Para entender este concepto, vamos a analizar un restaurante como si fuera un sistema Aditi. Para comenzar, podemos identificar que hay tres tipos de actividades que se realizan en los procesos del restaurante: la atención al cliente, la administración y la preparación de alimentos y bebidas. Estas grandes actividades las definimos como los **negocios** del sistema y les asignamos un identificador numérico:
 - Negocio 1: atención al cliente.
 - Negocio 2: administración.
 - Negocio 3: preparación de alimentos y bebidas.
- Ahora, de cada negocio obtendremos los **sistemas** que lo conforman, identificando las actividades que se realizan en cada uno de ellos:
 - Negocio 1: atención al cliente.
 - Sistema 1: recepción de clientes.
 - Sistema 2: atención en la mesa.
 - Negocio 2: administración.
 - Sistema 1: monitoreo y control de recursos.
 - Sistema 2: cobranza.
 - Negocio 3: preparación de alimentos y bebidas.
 - Sistema 1: preparación de alimentos en cocina.
 - Sistema 2: preparación de bebidas en barra.
 - Sistema 3: lavado de utensilios.
- Por último, de cada sistema se obtienen las **entidades** que realizan dichas actividades:
 - Negocio 1: atención al cliente.
 - Sistema 1: recepción de clientes:
 - Entidad 1: host.
 - Sistema 2: atención en la mesa:
 - Entidad 1: mesero.

- Negocio 2: administración.
 - Sistema 1: monitoreo y control de recursos.
 - Entidad 1: gerente.
 - Sistema 2: cobranza.
 - Entidad 1: cajero.
- Negocio 3: preparación de alimentos y bebidas.
 - Sistema 1: preparación de alimentos en cocina.
 - Entidad 1: cocinero.
 - Entidad 2: ayudante.
 - Sistema 2: preparación de bebidas en barra.
 - Entidad 1: barrista.
 - Sistema 3: lavado de utensilios.
 - Entidad 1: lava platos.
- Por su parte, el principio de **replicación funcional** consiste en determinar cuántas células de cada tipo deben existir en el sistema, acorde a la criticidad que cada una tenga en la operación del negocio, esto con el fin de garantizar la alta disponibilidad del sistema.
- Si la *Aditi Cell "A"* presta servicios más demandados y más importantes para el sistema en comparación con la *Aditi Cell "B"*, entonces el sistema deberá tener siempre más instancias de la *Aditi Cell "A"*, por si alguna de ellas falla o no se encuentre disponible, existan otras que puedan seguir ofreciendo la funcionalidad.
- Siguiendo con el ejemplo del restaurante, en la replicación funcional se define cuántos meseros requiere el restaurante para atender la demanda, cuántos cocineros, cuántos lavaplatos, etc.

¿Cómo se desarrolla un sistema en Aditi?



¿Cómo funciona la Aditi Cell?



- Aditi Cell es la materia prima para la construcción de sistemas y componente principal de la Plataforma Aditi.
- Se trata de un aplicativo que contiene protocolos y tecnologías para **imitar** el comportamiento de las **células vivas** y **proveer** las funcionalidades del sistema mediante **microservicios**.
- Garantiza tres **características online**:
 - *Expansión en línea*: se pueden **añadir** nuevas **células** o nuevos **microservicios** a las ya existentes **en cualquier momento** que el negocio lo requiera, **sin necesidad de detener** la operación total del sistema o de reiniciarlo.
 - *Mantenimiento en línea*: es posible **dar mantenimiento** a una célula o servicio en específico **sin afectar la operación** del resto del sistema.
 - *Tolerancia a fallas*: si una **célula o servicio falla**, el resto del sistema **continuará operando**. Las células cuentan con una tecnología de **autorrecuperación** que se ejecuta en escenarios críticos para estabilizar la operación del sistema.

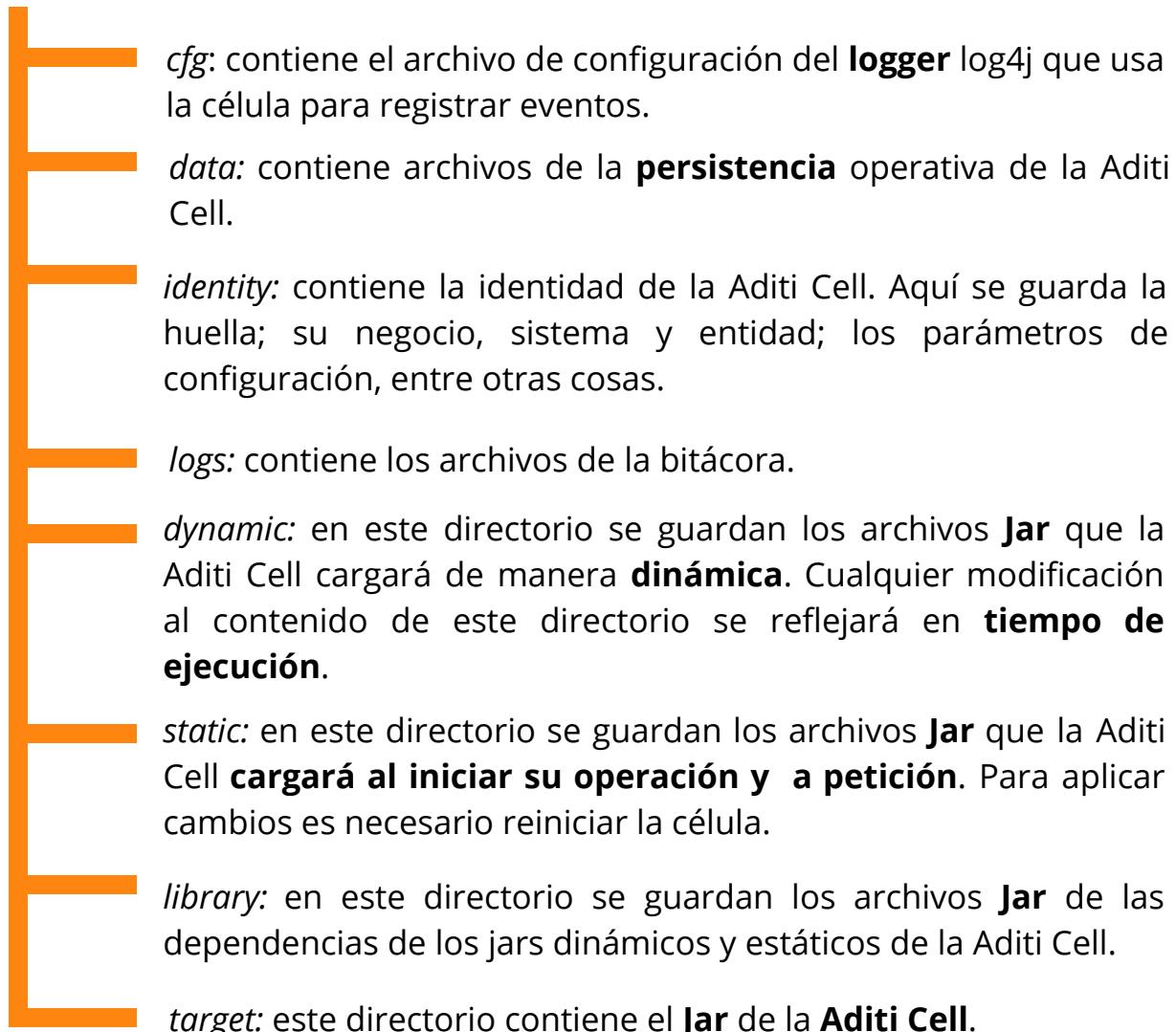


Cada Aditi Cell cuenta con **dos identificadores**:

- **Huella**: cadena de caracteres que es única para cada Aditi Cell y que definen la individualidad de los componentes dentro de la plataforma Aditi.
- **Identidad**: es el identificador grupal en la plataforma. Todas las Aditi Cell que son del mismo tipo lo comparten. Está conformado por los valores numéricos asignados al negocio, sistema y entidad al que pertenece la célula. Se representa con el formato *negocio.sistema.entidad*.
- Por ejemplo, las Aditi Cells que pertenezcan al negocio 3, al sistema 2 y sean la entidad 1, tendrán la identidad 3.2.1 en común, pero cada una de ellas contará con su propia huella.

- El Aditi Cell tiene la siguiente estructura de directorios:

Nombre de la Célula



El Aditi Manager crea todos estos directorios por ti cuando instala una célula nueva. Es importante no manipularlos directamente, sino mediante el Aditi Manager para garantizar el correcto funcionamiento de la Aditi Cell.

Protocolos de Comunicación



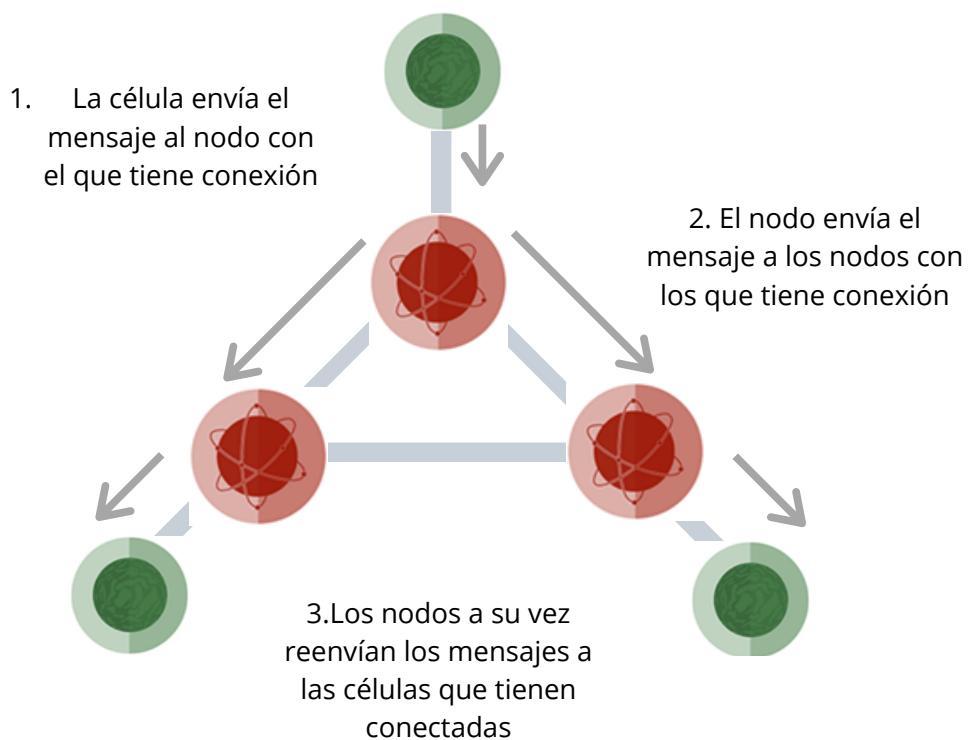
- La comunicación entre células es la base de la interacción en el sistema. Así como en los organismos vivos las células liberan moléculas que viajan por la sangre para realizar procesos vitales, en la plataforma Aditi las Aditi Cell liberan mensajes en un **bloodstream** del Aditi Net para que sean distribuidos en el resto del sistema y se lleven a cabo sus funciones.



Bloodstream:

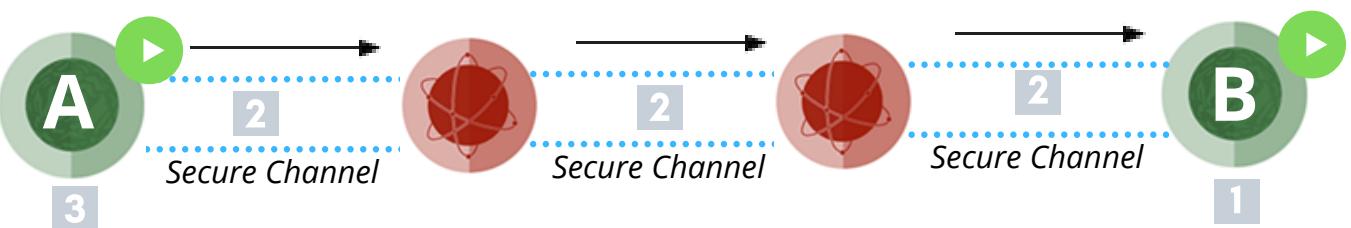
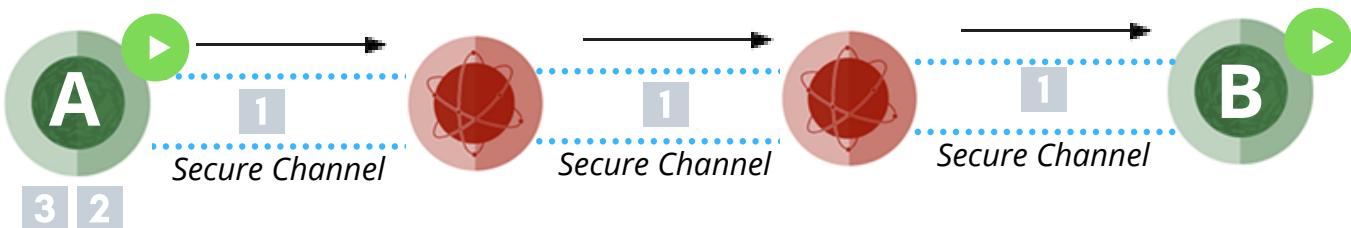
Conjunto de nodos de Aditi Net conectados entre sí a manera de malla, que tiene los mismo permisos de acceso para las mismas Aditi Cells.

- Las Aditi Cells se conectan a un nodo del bloodstream a la vez. A este nodo es al que envían los mensajes con peticiones para otras células y de él reciben los mensajes que envían otras células. Los nodos a su vez reenvían los mensajes que reciben de sus células a los demás nodos y envían a las células los mensajes que reciben de los nodos.



- Esta forma de comunicación hace flexible a los servicios y a la infraestructura pues evita que las células deban conocer en todo momento las direcciones de o que exista un componente en el sistema que guarde esa información global. Además, aumenta la supervivencia de los sistemas pues elimina puntos únicos de falla.
- Tanto la célula como el Aditi-Net están preparados para manejar la expansión de la red, si un nuevo equipo de cómputo se agrega a los recursos del sistema, los ya existentes lo incluirán de manera inmediata dentro de la red de comunicación.
- Si alguno de estos nodos de la red llega a fallar, las células que se encuentran conectadas a él iniciarán el **protocolo de reconexión** para a otro punto disponible de la red. En caso de no encontrar ninguno disponible, la célula se termina para no consumir recursos.
- Las Aditi Cells tienen dos formas de intercambiar mensajes. La primera es mediante el **Protocolo Transaccional**, en donde si la célula A envía un mensaje a la célula B, se distribuye por broadcast a todas las células B en el sistema y éstas deben enviar un acuse de recibo a A.
- Al ser distribuidos por todo el Aditi Net, los mensajes tienen dos identificadores para tener control sobre ellos:
 - Secuencial: número continuo de mensaje transaccional que el solicitante envía al destinatario.
 - Evento: identificador de la solicitud transaccional. Una vez atendida la solicitud, se marca como atendido dicho evento. Si un nuevo mensaje llega con un evento ya atendido, la solicitud se ignora.

- Al diseñar el sistema, se define el **nivel de servicio** de cada célula, es decir, cuál es el número mínimo de instancias existentes en el sistema para cada célula. Si A no recibe el mismo número de acuses que el nivel de servicio, volverá a enviar el mismo mensaje (si tiene otros en cola para la célula B, permanecerán en espera) hasta obtener respuesta o hasta agotar el límite de intentos.
- En caso de que ocurra este último escenario, se ejecutarán el protocolo de autorrecuperación de la plataforma, para generar el número de células faltantes y así preservar el nivel de servicio.



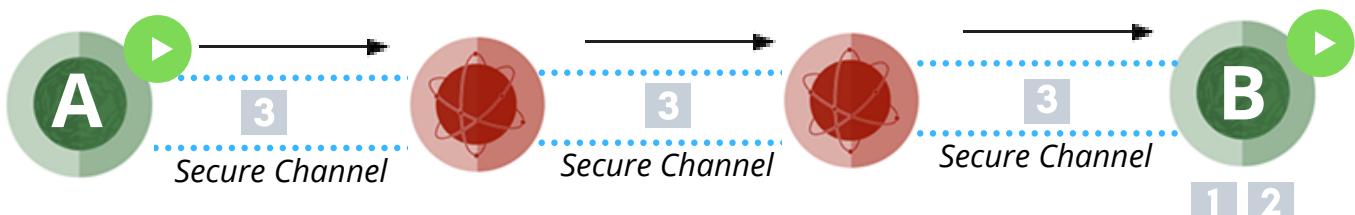
- El segundo protocolo de comunicación es el **No Transaccional**, donde el intercambio de mensajes se realiza de manera simple (A envía a B cuantos mensajes requiera sin necesidad de esperar respuesta), puesto que el emisor no requiere que el destinatario envíe un acuse.
- Este protocolo no se garantiza que el mensaje haya sido recibido ni se toma en cuenta el nivel de servicio de cada célula. No obstante, la no transaccionalidad permite enviar **mensajes dirigidos** a una célula en específico.



Paso 1: "A" envía el mensaje 1 a "B" mediante el Aditi Net



Paso 2: "A" puede enviar inmediatamente el mensaje 2 a "B"



Paso 3: "A" puede enviar inmediatamente el mensaje 3 a "B"

Protocolo de Autenticación

- Este es el mecanismo de seguridad más elemental de la Plataforma Aditi. Para que la Aditi Cell establezca cualquier tipo de comunicación con el Aditi Net, primero debe autenticarse mediante password o usando certificados digitales, con el nodo del Aditi Net al que se conecta y también viceversa, el nodo debe autenticarse con la Aditi Cell.
- De igual manera, los nodos del Aditi Net deben estar autenticados entre sí para compartir mensajes.
- Explicaremos cómo ocurre el protocolo para la autenticación entre nodo y célula:
 - En el instante 0, La célula conoce la dirección y el puerto donde se encuentra el nodo y abre un socket para enviar un mensaje de petición de autenticación por uno de los dos mecanismos.



- En el instante 1A, El nodo recibe el mensaje y busca en su base de permisos si la célula N.S.E. tiene permisos de autenticación por ese mecanismo. En caso afirmativo, contesta un mensaje con un reto.



- En el instante 1B, el nodo inicia su autenticación con la célula, mandando un mensaje de petición de autenticación.



- En el instante 2A, la célula recibe el reto, genera la solución utilizando su mecanismo de autenticación y manda la respuesta al nodo.



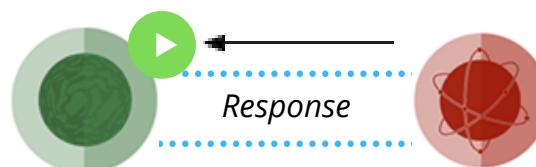
- En el instante 2B, recibe el mensaje de petición de autenticación del nodo y manda un reto.



- En el instante 3A, el nodo recibe la respuesta de la célula. Si es correcta, la célula se ha autenticado con el nodo.



- En el instante 3B, el nodo recibe el reto de la célula, genera la solución empleando su mecanismo de autenticación y la envía a la célula.



- En el instante 4, la célula recibe la respuesta. Si es correcta la solución, el nodo queda autenticado y la comunicación entre la célula y el Aditi net puede iniciar. Este mismo protocolo aplica para las conexiones entre nodos.

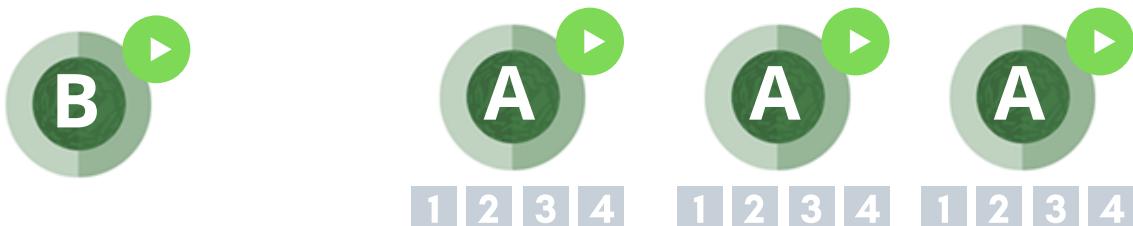


- En caso de que se logre la autenticación en los dos sentidos, inicia el **protocolo de expansión**. En este, el nodo envía a la célula todos los nodos que conoce para que la célula registre a los que tenga faltantes. Igualmente, la célula envía al nodo todos sus nodos conocidos; si alguno de ellos no lo conociera, lo registra y lo manda a los nodos que tiene conectados para que sea conocido por el resto del bloodstream.
- La autenticación puede fallar porque la célula no tiene permisos con el nodo y viceversa, porque el reto no fue correcto o porque se cumplió el tiempo para terminar la autenticación. La célula tiene un **límite de intentos** para autenticarse con los nodos del bloodstream. Si no logra autenticarse con un determinado nodo, intentará con algún otro de los que conozca, sucesivamente hasta agotar todos sus intentos posibles. Si no logra establecer una conexión doblemente autenticada con el bloodstream, la célula se termina para no consumir recursos. Del otro lado, los nodos detectan los intentos fallidos de conexión y registran la dirección origen de las peticiones. Si se alcanza el límite de autenticaciones fallidas, la dirección es registrada en la **lista negra** y bloqueada para contener **ataques de denegación de servicio**.

Protocolo de Sincronización



- El protocolo de sincronización tiene como finalidad mantener a las células que conforman el sistema en el mismo estado válido.
- Por su paradigma descentralizado y autónomo, las células pueden continuar operando en ausencia de otras. Dentro de este escenario, es posible que células que dejaron de estar disponibles se reincorporen a la operación o que nuevas células sean añadidas al sistema. En ambos casos, dichas células carecerán de los mensajes transaccionales que se generaron en el lapso que estuvieron fuera de operación.
- Supongamos que en el instante 0 existen tres células "A" que han atendido 4 mensajes transaccionales de las células "B".



- En el instante 1, la célula "A-3" pierde la conexión con su nodo.



- En el instante 2, una célula "B" envía el siguiente mensaje transaccional y las dos células restantes lo procesan.



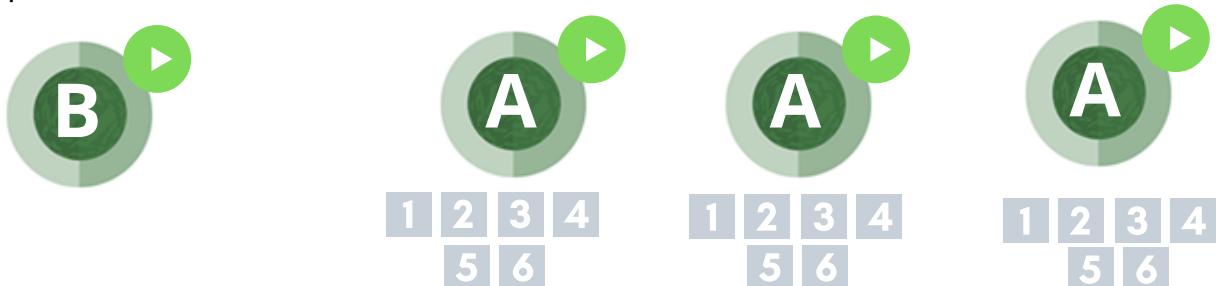
- Para el instante 3, la célula perdida recupera la conexión y la célula "B" envía el mensaje transaccional número 6. Entonces, la célula A-3 identifica que el secuencial recibido es distinto al secuencial siguiente de los mensajes que tiene registrados y por ende no lo puede procesar. Es entonces que la célula "A-3" inicia el protocolo de sincronización para recuperar los mensajes faltantes.



- En el instante 4, "A-3" envía una petición de sincronización a las otras células A. "A-1" y "A-2" envían el mensaje faltante.



- Posteriormente, "A-3" lo registra como un mensaje atendido y puede seguir operando.



- No obstante, las células tienen un límite de mensajes que pueden sincronizar. Por ejemplo, Si el límite son 100 mensajes, entonces las células guardarán los últimos 100 mensajes transaccionales que reciban. En caso de que una célula necesite más mensajes de los que se tienen en la tolerancia, la célula quedará en un estado inoperable.

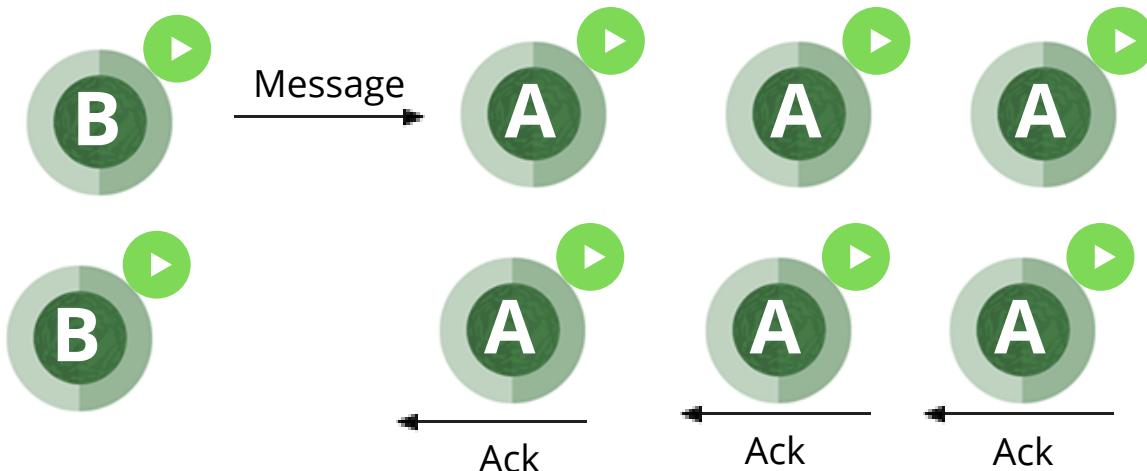
Protocolo de Autorrecuperación



- El protocolo de autorecuperación tiene como objetivo regenerar las células que han quedado en un estado inoperante o que se han perdido por alguna falla.
- Para que el sistema se autorrecupere, las células funcionales copian toda la información que contienen en una nueva instancia que se inicia de manera independiente.
- Este protocolo utiliza los mensajes transaccionales y el nivel de servicio de las células para determinar el momento en que se requiere y cuántas células deben clonarse.
- Supongamos que en un sistema, la célula "A" tiene un nivel de servicio 3, por lo que deben existir 3 instancias funcionales de ese tipo simultáneamente.



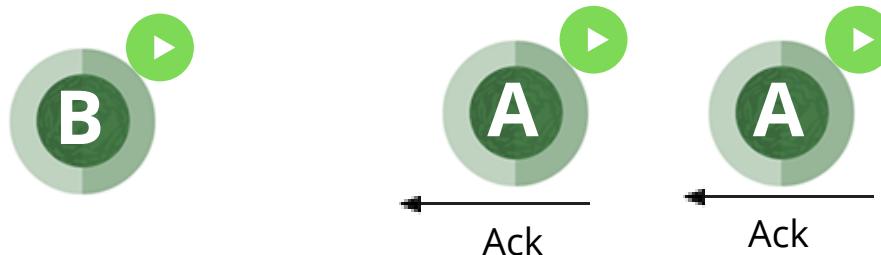
- En el instante 0, una célula "B" envía un mensaje transaccional a las células A. Las 3 instancias de A reciben el mensaje y lo acusan a "B".



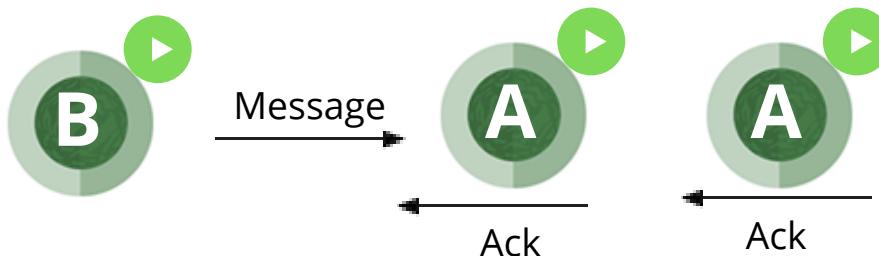
- En el instante 1, "B" recibe los tres acuses y manda el siguiente mensaje transaccional. Simultáneamente, la célula "A-3" deja de funcionar.



- En el instante 2, las dos células "A" que quedan reciben el mensaje y lo acusan a "B".



- En el instante 3, "B" recibe únicamente 2 acuses. Esperará un tiempo para recibir el acuse faltante. Si se agota el tiempo, reenviará el mensaje hasta agotar la tolerancia.



- En el instante 4, al no completar el número de acuses, "B" envía una solicitud de clonado a las células "A" disponibles.



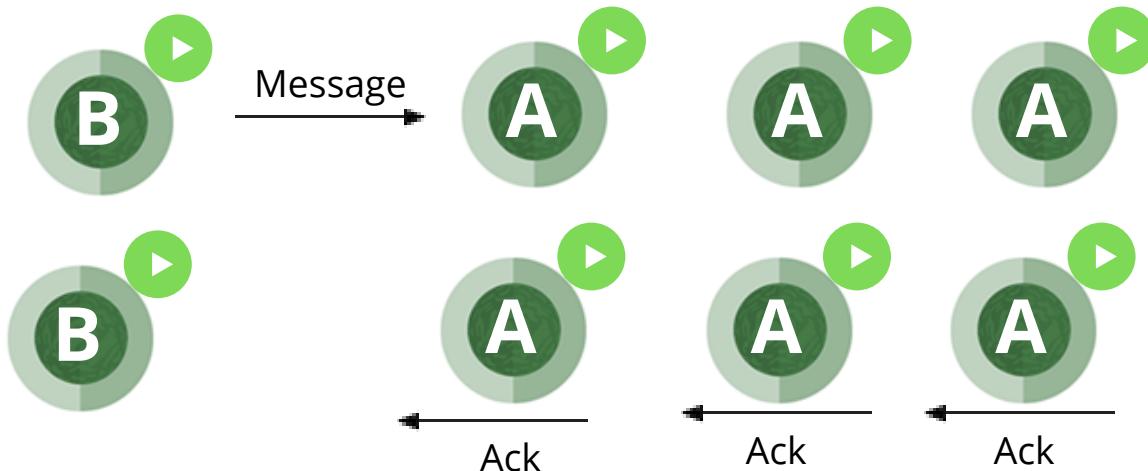
- En el instante 5, las células "A" reciben la solicitud y determinan cuál de las dos se clona.



- En el instante 6, la célula "A" elegida copia todo su contenido e inicia un nuevo aplicativo Aditi Cell. La nueva célula se autentica con el bloodstream y comienza su operación.



- En el instante 7, "B" reenvía el mensaje transaccional y las tres células A lo acusan. El sistema quedó estabilizado.



Esquema de Seguridad Aditi

- El esquema de seguridad de la plataforma Aditi está dividido por capas. En la capa de más bajo nivel es donde se ejecuta el protocolo de autenticación, por el cual se crean canales seguros para la transmisión de información.
- Las capas de más alto nivel son totalmente definibles para satisfacer las necesidades de seguridad de cada sistema. La plataforma Aditi provee una interfaz para definir mecanismos de cifrado adaptables a cada uno de los servicios de la célula.
- Éstos son implementados extendiendo de la clase *SecurityService* que define dos métodos:
 - *pack*: donde se define la manera en que se cifrará los parámetros del mensaje cuando se solicite un servicio.
 - *verifyUnpack*: donde se define la manera en que se descifrarán los parámetros cuando se reciba una petición de servicio.
- Aditi provee tres clases *SecurityService* por defecto:
 - *DefaultEncryptSecurityService*: para cifrado AES usando el certificado de autenticación de la célula.
 - *DefaultSignSecurityService*: para firmar la información mediante SHA256withRSA.
 - *DefaultEncryptSignSecurityService*: para enviar la información en un sobre digital usando AES para encriptar y SHA256withRSA para firmar.
- Para hacer uso de estos mecanismos, todas las células del sistema deben tener registrado en su identidad qué clase *SecurityService* usa qué número de servicio para qué célula, además de los certificados de cada una de ellas.

¿Qué son los Artefactos?



Artefacto:

Archivos Jar que contiene clases que la Aditi Cell puede interpretar.

- Los artefactos representan el ADN de las Aditi Cells, pues definen el comportamiento de negocio que pueden realizar.
- Los artefactos se clasifican dependiendo de las funcionalidades que aporten a la Aditi Cell:
 - Recurso: Si las clases contenidas en el Jar representan un **elementos auxiliares** o **microservicios** para la ejecución de las tareas de negocio del sistema y extienden de la superclase *Resource* de la API de Aditi.
 - Servicio: Si las clases contenidas en el Jar definen las **tareas de negocio** o **microservicios** de la célula y extienden de la superclase *Service* de la API de Aditi.



Tip:

Instala los servicios en el directorio *dynamic* para administrarlos en línea. Instala los recursos en el directorio *static*.

- Librería: Como su nombre lo indica, son las dependencias que los servicios y recursos requieren para ejecutar sus funciones. Estos artefactos deben ser instalados en el directorio *Library* de la Aditi Cell para que sean cargadas en la máquina virtual y estén disponibles cuando se requieran.



En caso de que un artefacto *librería* no contenga todas sus dependencias, éstas deberán ser agregadas también en el directorio *Library*.



PARTE II:

Desarrollo



¿Cómo crear un Recurso?



Recurso:

Artefacto que define un elemento auxiliar para la ejecución de las tareas de negocio del sistema.

- Los recursos son objetos cargados en la máquina virtual y que pueden ser accedidos por la célula.
- Estos objetos se “encapsulan” en un objeto de la clase **Resource** de la API de Aditi, para que puedan ser manejados por la célula.
- Dicha clase contiene cuatro funciones que definen el **ciclo de vida** del recurso:
 - *make*: en esta función se define cómo crear el recurso. Regresa el objeto que será manipulado por la célula.
 - *isValid*: en esta función se define una condición de validez que será evaluada cada vez que el recurso sea solicitado. Si el recurso deja de ser válido, la célula lo destruye y crea uno nuevo.
 - *destroy*: en esta función se definen los pasos a seguir para destruir el objeto encapsulado en el recurso.
 - *exception*: en esta función se define el manejo de las excepciones. Se ejecuta al ocurrir una excepción en alguna de las tres funciones anteriores.

- Adicionalmente, es necesario agregar la anotación **ResourceInformation** a la clase del Recurso para que la célula pueda interpretarla como tal. Esta anotación tiene los siguientes parámetros:
 - name*: es el nombre con el que la célula identificará al recurso.
 - shared*: un booleano que indica si el recurso será compartido por más de un servicio a la vez. Tiene valor verdadero por defecto.
 - init*: un booleano que indica si el recurso debe crearse cuando la célula inicia o se crea hasta que es solicitado por primera vez. Tiene valor negativo por defecto.



En caso de que dos Recursos en una misma Aditi Cell tengan el mismo nombre, solamente será válido el último que haya sido cargado.

- El valor del parámetro *init* determina el momento en que la célula ejecuta la función *make* para crear el recurso. Si el recurso es solicitado una vez que ya ha sido creado, la célula ejecuta la función *isValid*: si el recurso aún es válido lo devuelve; si no, lo destruye mediante la función *destroy* y crea uno nuevo.



Los recursos son accedidos desde los servicios (más adelante veremos cómo). Se puede especificar dos parámetros más:

- Si la célula debe destruir y volver a crear el recurso antes de que sea usado por el servicio.
- Si la célula debe destruir el recurso una vez que termine de ejecutar el servicio.

- Así es la estructura de un recurso Aditi:

```

1  @ResourceInformation(name = "myResource", init = true, shared = false)
2  v public class myResource extends Resource<Object> {
3
4      @Override
5      public Object make() throws Exception {
6          /*
7              Your code to make the object
8          */
9      }
10     @Override
11     public void destroy() throws Exception {
12         /*
13             Your code to destroy the object
14         */
15     }
16     @Override
17     public void exception(Throwable throwable) {
18         /*
19             Your code to handle exceptions
20         */
21     }
22     @Override
23     public Boolean isValid() throws Exception {
24         /*
25             Your code to validate the object
26         */
27     }
28 }
```

*Annotation with the **Resource Information***

Resource superclass inheritance

Overrided Functions that define the life cycle of the resource

Ejemplo 1:

Recurso Hola Mundo

Dificultad 

- El primer recurso que construiremos será un típico Hola Mundo. Para ello, creamos una clase Java llamada ResourceHelloWorld e incluimos como dependencia la API de Aditi.



Tip:

Como buena práctica de programación, inicia el nombre de tus clases con *Resource*, para que puedas distinguirlas de los Servicios y Librerías.

Ejemplo: *ResourceFoo*, *ResourceBar*.

- El primer paso es agregar la anotación *ResourceInformation* a la clase. Para este ejemplo, solo indicaremos que el parámetro *name* toma el valor de "*ResourceHelloWorld*"; los otros dos parámetros (*init* y *share*) los dejaremos con sus valores por defecto.

```
8 @ResourceInformation(name = "ResourceHelloWorld")
```



Tip:

Utiliza el nombre de la clases como nombre del recurso en la etiqueta.

- Indicar que nuestra clase *ResourceHelloWorld* extiende de la superclase *Resource* de la API Aditi. En este punto se declara el tipo de recurso, que para nuestro ejemplo es un String.

```
9 public class ResourceHelloWorld extends Resource<String> {
```

- La Aditi Cell ocupa log4j para manejar sus bitácoras. Para poder utilizar esta funcionalidad, agregamos el Logger.

```
14     private static final Logger LOGGER = Logger.getLogger(ResourceHelloWorld.class);
```

- Declaramos el objeto String que será encapsulado como el Recurso Hola Mundo.

```
18     String helloWorld;
```

- En la función make añadimos el código que asigne el valor "Hello World" al atributo helloWorld y lo regrese.

```
20     @Override
21     public String make() throws Exception {
22         if (Objects.isNull(helloWorld)) {
23             helloWorld = "Hello World";
24         }
25         return helloWorld;
}
```

- En la función destroy, declaramos que la manera de destruir el recurso es asignando un valor nulo al objeto helloWorld.

```
28     @Override
29     public void destroy() throws Exception {
30         helloWorld = null;
31     }
```

- En la función isValid, declaramos que el recurso es válido si el objeto es distinto a nulo.

```
33     @Override
34     public Boolean isValid() throws Exception {
35         return Objects.nonNull(helloWorld);
36     }
```

- En la función exception agregamos una entrada al log de la célula indicando la excepción que fue arrojada.

```
38     @Override
39     public void exception(Throwable throwable) {
40         LOGGER.error("Hello world", throwable);
41     }
```

- El código completo del Recurso Hola Mundo:

```
1 package org.mithikel.aditi.demo.resourceelloworld;
2
3 import java.util.Objects;
4 import org.apache.log4j.Logger;
5 import org.mithikel.aditi.cell.rs.resource.Resource;
6 import org.mithikel.aditi.cell.rs.resource.ResourceInformation;
7
8 @ResourceInformation(name = "ResourceHelloWorld")
9 public class ResourceHelloWorld extends Resource<String> {
10
11     /**
12      * Log
13      */
14     private static final Logger LOGGER = Logger.getLogger(ResourceHelloWorld.class);
15
16     * String object
17     */
18     String helloWorld;
19
20     @Override
21     public String make() throws Exception {
22         if (Objects.isNull(helloWorld)) {
23             helloWorld = "Hello World";
24         }
25         return helloWorld;
26     }
27
28     @Override
29     public void destroy() throws Exception {
30         helloWorld = null;
31     }
32
33     @Override
34     public Boolean isValid() throws Exception {
35         return Objects.nonNull(helloWorld);
36     }
37
38     @Override
39     public void exception(Throwable throwable) {
40         LOGGER.error("Hello world", throwable);
41     }
42
43 }
```

Ejemplo 2: Ventana JFrame

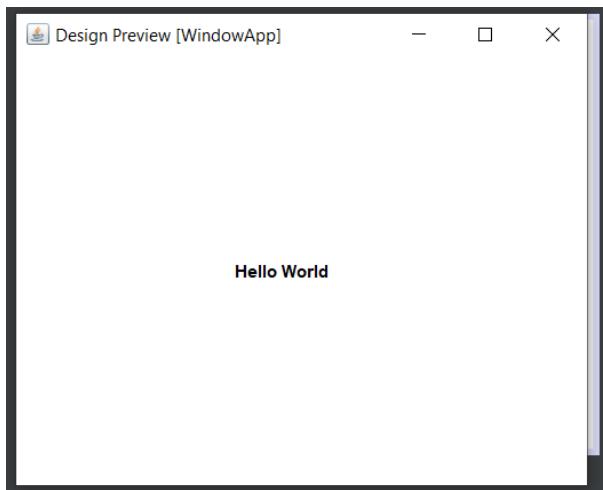
Dificultad 

- El segundo ejemplo que presentamos es un JFrame que se desplegará cuando la Aditi Cell se inicie. Para ello, crearemos dos clase, una será la clase del JFrame que será usada en la clase del Recurso Aditi.



La clase del Recurso tendrá como dependencia la clase del JFrame. En este ejemplo manejaremos las clases en dos artefactos diferentes, por lo que el artefacto que contenga el JFrame deberá ser agregado como **Librería** en la Aditi Cell.

- Comenzamos creando un proyecto que contenga una clase JFrame que nombraremos *WindowApp*. En este ejemplo únicamente añadiremos un label que diga "Hello World" para hacerlo lo más simple posible; no obstante, la ventana puede ser lo más compleja que requieras, eso no afecta al Recurso Aditi.



- Compilamos el proyecto *WindowApp* para poder utilizarlo en la clase del Recurso.

- En un proyecto distinto, creamos una clase llamada *ResourceWindowApp*. Añadimos la anotación *ResourceInformation* con los siguientes parámetros:
 - *name = ResourceWindowApp*
 - *init = True*, para que cuando la Aditi Cell inicie, cree el recurso y despliegue la ventana.
 - *shared = False*, para que la instancia de la ventana sea única.

```
10 @ResourceInformation(name = "window", init = true, shared = false)
```

- El siguiente paso es extender de la superclase *Resource* indicando que el recurso es de tipo *WindowApp* (la clase del JFrame).

```
11 public class ResourceWindowApp extends Resource<WindowApp> {
```

- Como atributos de la clase *ResourceWindowApp*, declaramos el logger de log4j para escribir en la bitácora de la célula y la instancia de *WindowApp* que será manejada como recurso.

```
13 /**
14  * Logger
15 */
16 private static final Logger LOGGER = Logger.getLogger(ResourceWindowApp.class);
17 /**
18  * JFrame
19 */
20 private WindowApp windowApp;
```

- En la función *make* creamos la instancia de *WindowApp* y la desplegamos mediante el *EventQueue*.

```
21
22     @Override
23     public WindowApp make() throws Exception {
24         if (Objects.isNull(windowApp)) {
25             windowApp = new WindowApp();
26             EventQueue.invokeLater(() -> {
27                 windowApp.setVisible(true);
28             });
29         }
30         LOGGER.info("win value (" + windowApp + ")");
31         return windowApp;
32     }
```

- En la función *destroy* creamos un evento de cierre en el objeto de la ventana.

```
33
34     @Override
35     public void destroy() throws Exception {
36         windowApp.dispatchEvent(new WindowEvent(windowApp, WindowEvent.WINDOW_CLOSING));
37     }
```

- En la función *isValid* definimos que el recurso de la ventana es válido mientras la instancia de *WindowApp* no sea nulo.

```
44     @Override
45     public Boolean isValid() throws Exception {
46         return Objects.nonNull(windowApp);
47     }
48 }
49 }
```

- Para el manejo de excepciones añadimos una entrada a la bitácora indicando la excepción reportada.

```
39     @Override
40     public void exception(Throwable throwable) {
41         LOGGER.error("WindowApp", throwable);
42     }
```

- El código completo del Recurso WindowApp:

```
1 org.mithikel.aditi.demo.resourcewindowapp;
2
3 import java.awt.EventQueue;
4 import java.awt.event.WindowEvent;
5 import java.util.Objects;
6 import org.apache.log4j.Logger;
7 import org.mithikel.aditi.cell.rs.resource.Resource;
8 import org.mithikel.aditi.cell.rs.resource.ResourceInformation;
9
10 @ResourceInformation(name = "window", init = true, shared = false)
11 public class ResourceWindowApp extends Resource<WindowApp> {
12
13     /**
14      * Logger
15     */
16     private static final Logger LOGGER = Logger.getLogger(ResourceWindowApp.class);
17
18     /**
19      * JFrame
20     */
21     private WindowApp windowApp;
22
23     @Override
24     public WindowApp make() throws Exception {
25         if (Objects.isNull(windowApp)) {
26             windowApp = new WindowApp();
27             EventQueue.invokeLater(() -> {
28                 windowApp.setVisible(true);
29             });
30             LOGGER.info("win value (" + windowApp + ")");
31             return windowApp;
32         }
33     }
34
35     @Override
36     public void destroy() throws Exception {
37         windowApp.dispatchEvent(new WindowEvent(windowApp, WindowEvent.WINDOW_CLOSING));
38     }
39
40     @Override
41     public void exception(Throwable throwable) {
42         LOGGER.error("WindowApp", throwable);
43     }
44
45     @Override
46     public Boolean isValid() throws Exception {
47         return Objects.nonNull(windowApp);
48     }
49 }
```

Ejemplo 3:

Aditi Logger

Dificultad 

- En este ejemplo crearemos una clase para que escriba bitácoras en un archivo de texto y la usaremos para crear un recurso. Al igual que en el ejemplo pasado, las dos clases estarán en artefactos distintos.
- Como primer paso, en un proyecto nuevo creamos una clase llamada AditiLogger. Declaramos como atributos de clase el archivo en el que se escribirá la bitácora, el nombre del archivo y un objeto Date para obtener la fecha y hora. A todos ellos los inicializamos en el constructor de la clase.

```

11  /**
12  * Log file
13  */
14 private File file;
15 /**
16 * File name for the log
17 */
18 private final String LOGNAME;
19 /**
20 * Date object to obtain the current date and time
21 */
22 private Date date;
23
24 public AditiLogger() {
25     LOGNAME = "aditi.log";
26     file = new File(LOGNAME);
27     date = new Date();
28 }
```

- Los métodos del AditiLogger son los siguientes:
 - *open*: para crear el archivo del log si no existe y para registrar la hora en que se abrió.
 - *write*: para escribir en el log una cadena de texto y la hora en que se ha registrado.
 - *close*: para registrar en el log la hora en que se cerró.

```

29
30     public void open() throws Exception {
31         if (!file.exists()) {
32             file.createNewFile();
33         }
34         write("THE LOG IS OPENED");
35     }
36
37     public void write(String info) throws Exception {
38         try ( FileWriter fw = new FileWriter(file, true); BufferedWriter bw = new BufferedWriter(fw); PrintWriter out = new PrintWriter(bw)) {
39             out.println(date.toString() + ": " + info);
40         }
41     }
42
43     public void close() throws Exception {
44         write("THE LOG IS CLOSED. GOODBYE!");
45     }
46 }
```

- En un proyecto aparte, creamos la clase ResourceAditiLogger y le agregamos la anotación ResourceInformation con los siguientes parámetros:
 - *name*: utilizamos el mismo nombre de la clase *ResourceAditiLogger*.
 - *init*: cambiamos el valor a *True* para que se quede registrado en el log el momento en que la célula creó el recurso.
 - *shared*: dejamos el valor *True* por defecto.

```
9  @ResourceInformation(name = "ResourceAditiLogger", init = true)
```

- Extendemos esta clase de la superclase Resource de la API de Aditi e indicamos que el recurso será del tipo *AditiLogger*.

```
10 ~ public class ResourceAditiLogger extends Resource<AditiLogger> {
```

- Como atributos de la clase ResourceAditiLogger, declaramos el logger de log4j para escribir en la bitácora de la célula y la instancia de AditiLogger que será manejada como recurso.

```
12 ~     /**
13      * Logger
14      */
15     private static final Logger LOGGER = Logger.getLogger(ResourceAditiLogger.class);
16 ~     /**
17      * Aditi Logger
18      */
19     private AditiLogger aditiLogger;
```

- En la función *make* creamos la instancia de AditiLogger y ejecutamos su método *open*.

```
21     @Override
22 ~     public AditiLogger make() throws Exception {
23 ~         if (Objects.isNull(aditiLogger)) {
24 ~             aditiLogger = new AditiLogger();
25 ~             aditiLogger.open();
26 ~         }
27 ~         return aditiLogger;
28     }
```

- En la función `destroy` declaramos que se debe ejecutar el método `close` del logger.

```
30     @Override
31     public void destroy() throws Exception {
32         aditiLogger.close();
33     }
```

- En la función `isValid` declaramos que el recurso es válido mientras el objeto `aditiLogger` sea distinto a nulo.

```
35     @Override
36     public Boolean isValid() throws Exception {
37         return Objects.nonNull(aditiLogger);
38     }
```

- Para el manejo de las excepciones únicamente reportamos en el log de la célula.

```
40     @Override
41     public void exception(Throwable throwable) {
42         LOGGER.error("Resource Aditi Logger", throwable);
43     }
```

- El código completo de la clase Aditi Logger:

```
1 package org.mithikel.aditi.demo.aditilogger;
2
3 import java.io.BufferedWriter;
4 import java.io.File;
5 import java.io.FileWriter;
6 import java.io.PrintWriter;
7 import java.util.Date;
8
9 v public class Aditilogger {
10
11 v     /**
12      * Log file
13      */
14     private File file;
15 v     /**
16      * File name for the Log
17      */
18     private final String LOGNAME;
19 v     /**
20      * Date object to obtain the current date and time
21      */
22     private Date date;
23
24 v     public Aditilogger() {
25         LOGNAME = "aditi.log";
26         file = new File(LOGNAME);
27         date = new Date();
28     }
29
30     public void open() throws Exception {
31         if (!file.exists()) {
32             file.createNewFile();
33         }
34         write("THE LOG IS OPENED");
35     }
36
37     public void write(String info) throws Exception {
38         try ( FileWriter fw = new FileWriter(file, true);
39              BufferedWriter bw = new BufferedWriter(fw);
40              PrintWriter out = new PrintWriter(bw)) {
41             out.println(date.toString() + ": " + info);
42         }
43     }
44
45     public void close() throws Exception {
46         write("THE LOG IS CLOSED. GOODBYE!");
47     }
48 }
```

- El código completo del Recurso Aditi Logger:

```
1 package org.mithikel.aditi.demo;
2
3 import java.util.Objects;
4 import org.apache.log4j.Logger;
5 import org.mithikel.aditi.cell.rs.resource.Resource;
6 import org.mithikel.aditi.cell.rs.resource.ResourceInformation;
7 import org.mithikel.aditi.demo.aditilogger.AditiLogger;
8
9 @ResourceInformation(name = "ResourceAditiLogger", init = true)
10 public class ResourceAditiLogger extends Resource<AditiLogger> {
11
12     /**
13      * Logger
14      */
15     private static final Logger LOGGER = Logger.getLogger(ResourceAditiLogger.class);
16
17     /**
18      * Aditi Logger
19      */
20     private AditiLogger aditiLogger;
21
22     @Override
23     public AditiLogger make() throws Exception {
24         if (Objects.isNull(aditiLogger)) {
25             aditiLogger = new AditiLogger();
26             aditiLogger.open();
27         }
28         return aditiLogger;
29     }
30
31     @Override
32     public void destroy() throws Exception {
33         aditiLogger.close();
34     }
35
36     @Override
37     public Boolean isValid() throws Exception {
38         return Objects.nonNull(aditiLogger);
39     }
40
41     @Override
42     public void exception(Throwable throwable) {
43         LOGGER.error("Resource Aditi Logger", throwable);
44     }
45 }
```

¿Cómo crear un Servicio?



Servicio:

Artefacto que define una funcionalidad específica de negocio que la célula puede ejecutar.

- La API de Aditi contiene una superclase *Service* de la que deben heredar las clases que representen servicios. Ésta contiene el siguiente par de métodos:
 - *run*: función que es invocada por la Aditi Cell cuando se solicita el servicio. En esta función se deben escribir la funcionalidad principal del servicio.
 - *exception*: función que debe contener las acciones a realizar en caso de que se arroje una excepción al ejecutar la función *run*.
- Adicionalmente, la clase contiene los siguientes atributos:
 - La **huella** de la Aditi Cell que solicita el servicio, de tipo *Integer*.
 - El número de **Negocio** de la Aditi Cell solicitante, de tipo *Short*.
 - El número de **Sistema** de la Aditi Cell solicitante, de tipo *Short*.
 - El número de **Entidad** de la Aditi Cell solicitante, de tipo *Short*.
 - El identificador del mensaje de solicitud del servicio
 - El **secuencial** del mensaje de solicitud del servicio



Para garantizar las características de la plataforma Aditi, los **servicios** de las Aditi Cells deben ser **sin estado**.

- Todas las clases que hereden de Service deben tener la anotación ServiceInformation de la API de Aditi para que la Aditi Cell pueda distinguirlos. En ella se deben indicar tres parámetros:
 - *Protocol*: Qué tipo de mensajes pueden ser solicitudes a este servicio. El parámetro debe ser de tipo Protocol (una enumeración contenida en la API): TRANSACTIONAL, NON_TRANSACTIONAL; ALL.
 - *Service Number*: El identificador numérico del servicio. El valor del parámetro debe ser de tipo Short.
 - *Service Name*: El identificador dinámico del servicio. El parámetro debe ser de tipo String con el nombre que se le da asigna al servicio.
- Así es la estructura de un servicio Aditi:

*Annotation with the **Service Information***



```

1  @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "MyService")
2  public class MyService extends Service {
3
4      @Override
5      public void run() throws Exception {
6          /**
7             * Your code with service implementation
8             */
9      }
10
11     @Override
12     public void exception(Throwable throwable) throws Exception {
13         /**
14             * Your code to handle exceptions
15             */
16     }
17
18 }

```

Service superclass inheritance

- Al igual que los recursos, los servicios también tienen un ciclo de vida definido, solamente que es más corto y sencillo. A continuación lo explicamos:

1. La Aditi Cell recibe un mensaje de **petición de servicio** de parte de otra célula. En este mensaje está indicado o el **número de servicio** o el **nombre del servicio** que se solicita. Además, puede incluir parámetros de entrada para la ejecución del servicio.



Los mensajes, las solicitudes de servicio y el envío de parámetros los explicaremos más adelante, por el momento basta con que sepas de su existencia.

2. La Aditi Cell busca en sus clases cargadas el servicio que coincida con el parámetro de la solicitud. En caso de no encontrarlo, arrojará una excepción.



En caso de que dos Servicios en una misma Aditi Cell tengan el mismo nombre o número de servicio, solamente será válido el último que haya sido cargado.

3. Si el servicio fue encontrado, la célula crea una nueva instancia de él, asignando valor a los atributos de la clase; a los parámetros de entrada, en caso de estar especificados; y obteniendo los recursos que sean necesitados.

4. Por último, la célula ejecuta el método *run* del servicio y maneja las excepciones que se presenten mediante el método *exception*.

Ejemplo 4 :

Servicio Hola Mundo

Dificultad     

- El primer recurso que construiremos será un Hola Mundo para mostrar los conceptos básicos en acción.
- El primer paso es crear en un nuevo proyecto, una clase llamada *ServiceHelloWorld* que anotamos con *ServiceInformation* y los siguientes parámetros:
 - *serviceNumber*: declaramos que este servicio tendrá el número 10 como identificador.



Como se trata de un ejemplo muy simple, el número de servicio es poco relevante en términos prácticos, así que se puede ser cualquier valor *short*. En implementaciones más complejas donde existan más servicios e interactúen entre sí, debes prestar más atención a la numeración de los servicios para que no haya conflictos al momento de poner en producción el sistema.

- *protocol*: utilizamos *Protocol.ALL* para que el servicio pueda ser solicitado por cualquier tipo de mensaje. En una sección más adelante explicaremos los tipos de mensajes de la plataforma Aditi.
- *name*: al igual que con los recursos, seguiremos la convención de nombrar a los servicios igual que las clases que los contienen.

```
7 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServiceHelloWorld")
```

- Extendemos esta clase de la superclase *Service* de la API de Aditi para acceder a los métodos *run* y *exception*.

```
8 v public class ServiceHelloWorld extends Service {
```

- Agregamos el Logger de log4j para escribir en la bitácora de la Aditi Cell.

```
10 v      /**
11      * Logger
12      */
13      private static final Logger LOGGER = Logger.getLogger(ServiceHelloWorld.class);
```

- En el método *run* simplemente vamos a escribir que cuando la célula ejecute el servicio se agregue un "Hello World" en la bitácora.

```
15      @Override
16 v    public void run() throws Exception {
17      LOGGER.info("Hello World");
18 }
```

- En el método *exception* solo agregamos una entrada en la bitácora para reportar qué excepción ocurrió.

```
20      @Override
21 v    public void exception(Throwable throwable) throws Exception {
22      LOGGER.error("", throwable);
23 }
```

- El código completo del Servicio Hello World:

```
1 import org.apache.log4j.Logger;
2 import org.mithikel.aditi.cell.rs.service.Protocol;
3 import org.mithikel.aditi.cell.rs.service.Service;
4 import org.mithikel.aditi.cell.rs.service.ServiceInformation;
5
6
7 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServiceHelloWorld")
8 public class ServiceHelloWorld extends Service {
9
10 /**
11 * Logger
12 */
13 private static final Logger LOGGER = Logger.getLogger(ServiceHelloWorld.class);
14
15 @Override
16 public void run() throws Exception {
17     LOGGER.info("Hello World");
18 }
19
20 @Override
21 public void exception(Throwable throwable) throws Exception {
22     LOGGER.error("", throwable);
23 }
24 }
```

¿Cómo acceder a recursos desde un servicio?



- Para hacer uso de cualquier recurso de la célula desde un servicio se emplea la anotación *InputResource* en la declaración de una variable de clase que sea del mismo tipo que el recurso. La anotación tiene tres parámetros:
 - resource*: es el nombre con el que fue creado el recurso. Con este identificador la célula busca el recurso en la máquina virtual y lo asigna a la variable. El parámetro es de tipo *String*.
 - renew*: un *Booleano* que indica si se debe renovar el recurso antes de asignarlo al atributo de la clase service. En otras palabras, si antes de asignar el valor, ejecuta el método *destroy* del recurso y genera uno nuevo. Tiene valor falso por defecto.
 - destroy*: un *Booleano* que indica si se debe destruir el recurso cuando el servicio termine de ejecutarse. Tiene valor falso por defecto.



En caso de que el valor del parámetro *resource* no coincida con ningún Recurso contenido en la Célula, la variable de clase permanecerá nula en la ejecución del servicio.

- Así es la estructura de un servicio que accede a un recurso:

```
1 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "MyService")
2 public class MyService extends Service {
3     /**
4      * MyResource object
5      */
6     @InputResource(resource = "ResourceMyResource")
7     Object myResource;
8
9     @Override
10    public void run() throws Exception {
11        /**
12         * Your code using the resource
13         */
14    }
15
16    @Override
17    public void exception(Throwable throwable) throws Exception {
18        /**
19         * Your code to handle exceptions
20         */
21    }
22
23 }
```

Annotación con la información del Recurso solicitado

Variable de clase a la que se asignará el recurso

Ejemplo 5: Hola Mundo con recursos

Dificultad 

- Este ejemplo es muy parecido al anterior, la única diferencia es que en lugar de crear un String en el método *run* para escribir en la bitácora, utilizaremos el recurso *ResourceHelloWorld* del primer ejemplo.

```
9  @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServiceHelloWorld")
10 v public class ServiceHelloWorldWithResource extends Service {
```

- La estructura de este ejemplo es prácticamente la misma del ejemplo anterior, puedes reutilizar el código, pero la recomendación es seguir paso a paso las indicaciones de este.
- Como primer paso creamos una clase llamada *ServiceHelloWorldWithResource* que extienda de la superclase *Service* de la API de Aditi y la anotamos con *ServiceInformation* y los siguientes parámetros:
 - serviceNumber: *short* 10.
 - protocol: *Protocol.ALL*, para que pueda ser solicitado por cualquier mensaje.
 - name: *ServiceHelloWorldWithResource*, para continuar con la convención de nombres usada hasta el momento.

```
9  @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServiceHelloWorldWithResource")
10 v public class ServiceHelloWorldWithResource extends Service {
```

- Agregamos el Logger de log4j para escribir en la bitácora de la Aditi Cell.

```
11 v /**
12  * Logger
13  */
14  private static final Logger LOGGER = Logger.getLogger(ServiceHelloWorldWithResource.class);
```

- Declaramos una variable de clase de tipo String llamada *helloWorld* y la anotamos con *InputResource* para que la célula le asigne el valor del recurso. El único parámetro que usaremos en la anotación es *resource* con valor *ResourceHelloWorld*, pues ese es el nombre que asignamos en la clase del Recurso.

```
14     /**
15      * Resource HelloWorld
16      */
17     @InputResource(resource = "ResourceHelloWorld")
18     String helloWorld;
```

- Ahora, a diferencia del ejemplo anterior, para escribir en la bitácora utilizamos la variable *helloWorld*. Obtenremos el mismo resultado que en ServiceHelloWorld, pero aquí exemplificamos el uso de Recursos.

```
20     @Override
21     public void run() throws Exception {
22         LOGGER.info(helloWorld);
23     }
```

- En el método exception solo agregamos una entrada en la bitácora para reportar qué excepción ocurrió.

```
25     @Override
26     public void exception(Throwable throwable) throws Exception {
27         LOGGER.error("", throwable);
28     }
```

- El código completo del Servicio Hello World con recursos:

```
1 package org.mithikel.aditi.demo;
2
3 import org.apache.log4j.Logger;
4 import org.mithikel.aditi.cell.rs.service.InputResource;
5 import org.mithikel.aditi.cell.rs.service.Protocol;
6 import org.mithikel.aditi.cell.rs.service.Service;
7 import org.mithikel.aditi.cell.rs.service.ServiceInformation;
8
9 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServiceHelloWorldWithResource")
10 public class ServiceHelloWorldWithResource extends Service {
11     /**
12      * Logger
13     */
14     private static final Logger LOGGER = Logger.getLogger(ServiceHelloWorldWithResource.class);
15     /**
16      * Resource HelloWorld
17     */
18     @InputResource(resource = "ResourceHelloWorld")
19     String helloWorld;
20
21     @Override
22     public void run() throws Exception {
23         LOGGER.info(helloWorld);
24     }
25
26     @Override
27     public void exception(Throwable throwable) throws Exception {
28         LOGGER.error("", throwable);
29     }
30 }
```

Ejemplo 6: Hola Mundo y AditiLogger

Dificultad 

- Para reforzar el uso de recursos desde los servicios, implementaremos un tercer Hola Mundo empleando los recursos del ejemplo 1 (*ResourceHelloWorld*) y del ejemplo 3 (*ResourceAditiLogger*).
- De nueva cuenta, la estructura de este ejemplo es prácticamente la misma del ejemplo anterior, puedes reutilizar el código, pero la recomendación es seguir paso a paso las indicaciones de este.
- Como primer paso creamos una clase llamada *ServiceHelloWorldWithAditiLogger* que extienda de la superclase *Service* de la API de Aditi y la anotamos con *ServiceInformation* y los siguientes parámetros:
 - *serviceNumber*: short 10.
 - *protocol*: *Protocol.ALL*, para que pueda ser solicitado por cualquier mensaje.
 - *name*: *ServiceHelloWorldWithAditiLogger*, para continuar con la convención de nombres usada hasta el momento.

```
10  @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServiceHelloWorldWithAditiLogger")
11  public class ServiceHelloWorldWithAditiLogger extends Service {
```

- Agregamos el Logger de log4j para escribir en la bitácora de la Aditi Cell.

```
13  /**
14  * Logger
15  */
16  private static final Logger LOGGER = Logger.getLogger(ServiceHelloWorldWithAditiLogger.class);
```

- Declaramos una variable de clase de tipo String llamada *helloWorld* y la anotamos con *InputResource* para que la célula le asigne el valor del recurso. El único parámetro que usaremos en la anotación es *resource* con valor *ResourceHelloWorld*, pues ese es el nombre que asignamos en la clase del Recurso.

```

14     /**
15      * Resource HelloWorld
16     */
17     @InputResource(resource = "ResourceHelloWorld")
18     String helloWorld;

```

- Declaramos una variable de clase de tipo AditiLogger llamada *aditiLogger* y la anotamos con *InputResource* para que la célula le asigne el valor del recurso. El único parámetro que usaremos en la anotación es *resource* con valor *ResourceAditiLogger*, pues ese es el nombre que asignamos en la clase del Recurso.

```

17 v    /**
18      * Resource HelloWorld
19     */
20     @InputResource(resource = "ResourceHelloWorld")
21     private String helloWorld;

```

- En el método *run* accedemos al método *write* de *aditiLogger* para escribir el valor del recurso *helloWorld*.

```

28     @Override
29     public void run() throws Exception {
30         aditiLogger.write(helloWorld);
31     }

```

- En el método *exception* solo agregamos una entrada en la bitácora para reportar qué excepción ocurrió.

```

33     @Override
34     public void exception(Throwable throwable) throws Exception {
35         LOGGER.error("ServiceHelloWorldWithAditiLogger", throwable);
36     }

```

- El código completo del Servicio Hello World con AditiLogger:

```
1 package org.mithikel.aditi.demo;
2
3 import org.apache.log4j.Logger;
4 import org.mithikel.aditi.cell.rs.service.InputResource;
5 import org.mithikel.aditi.cell.rs.service.Protocol;
6 import org.mithikel.aditi.cell.rs.service.Service;
7 import org.mithikel.aditi.cell.rs.service.ServiceInformation;
8 import org.mithikel.aditi.demo.aditilogger.AditiLogger;
9
10 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServiceHelloWorldWithAditiLogger")
11 public class ServiceHelloWorldWithAditiLogger extends Service {
12
13     /**
14      * Logger
15      */
16     private static final Logger LOGGER = Logger.getLogger(ServiceHelloWorldWithAditiLogger.class);
17
18     /**
19      * Resource HelloWorld
20      */
21     @InputResource(resource = "ResourceHelloWorld")
22     private String helloWorld;
23
24     /**
25      * Resource AditiLogger
26      */
27     @InputResource(resource = "ResourceAditiLogger")
28     private AditiLogger aditiLogger;
29
30     @Override
31     public void run() throws Exception {
32         aditiLogger.write(helloWorld);
33     }
34
35     @Override
36     public void exception(Throwable throwable) throws Exception {
37         LOGGER.error("ServiceHelloWorldWithAditiLogger", throwable);
38     }
39 }
```

¿Cómo solicitar Servicios?



- La API de Aditi contiene un singleton llamado **MessageFactory** que construye los mensajes y accede a la capa de comunicación de la célula para mandarlos al Aditi Net.
- El MessageFactory tiene un método para mensajes transaccionales y otro para mensajes no transaccio, ambos están sobrecargados para poder solicitar servicios por identificador numérico y por nombre.
- Se recomienda emplear las solicitudes de servicio por nombre para mayor flexibilidad en el sistema. Como podrás observar, este tipo de solicitud no requiere especificar en el código el negocio, sistema y entidad de la célula a quien se dirige el mensaje, por lo que los servicios pueden ser intercambiados dinámicamente entre células sin necesidad de volver a generar los artefactos.



Package Message:

Wrapper que empaqueta informacion mediante un mecanismo llave-valor para ser enviada en un mensaje.

- Estos son los parámetros de las sobrecargas del método **sendTransactional** solicitando por **nombre** de servicio:
 - *transaction*: transacción de la base de datos MattDB embebida en la Aditi Cell que permite formar el mensaje para su envío al Aditi Net.
 - *serviceName*: String con el mismo valor con el atributo *name* de la anotación *ServiceInformation* presente en el servicio solicitado.
 - *bloodstream*: String nombre del Aditi Net al que se enviará el mensaje.
 - *event*: String con el identificador del mensaje.
 - *packageMessage/ data*: una sobrecarga del método admite adjuntar un objeto Package Message; otra sobrecarga adjunta la información mediante un arreglo de bytes.

- Estos son los parámetros de las sobrecargas del método **sendTransactional** solicitando por **número** de servicio:
 - *transaction*: transacción de la base de datos MattDB embebida en la Aditi Cell que permite formar el mensaje para su envío al Aditi Net.
 - *business*: Short con el negocio de la célula destinataria del mensaje.
 - *system*: Short con el sistema de la célula destinataria del mensaje.
 - *entity*: Short con la entidad de la célula destinataria del mensaje.
 - *serviceNumber*: String con el mismo valor con el atributo *name* de la anotación *ServiceInformation* presente en el servicio solicitado.
 - *bloodstream*: String nombre del Aditi Net al que se enviará el mensaje.
event: String con el identificador del mensaje.
 - *packageMessage/data*: una sobrecarga del método admite adjuntar un objeto Package Message; otra sobrecarga adjunta la información mediante un arreglo de bytes.
- Estos son los parámetros de las sobrecargas del método **sendNonTransactional** solicitando por **nombre** de servicio:
 - *transaction*: transacción de la base de datos MattDB embebida en la Aditi Cell que permite formar el mensaje para su envío al Aditi Net.
 - *serviceName*: String con el mismo valor con el atributo *name* de la anotación *ServiceInformation* presente en el servicio solicitado.
 - *footprint*: Integer con la huella de la célula a la que se le envía el mensaje. Si el mensaje es un **broadcast**, el valor debe ser *Integer.MAX_VALUE* o *ConstantMessage.BROADCAST_FOOTPRINT*. Si el mensaje se ocupa en el esquema de **balanceo de trabajo**, el valor debe ser *Integer.MIN_VALUE* o *ConstantMessage.LOAD_BALANCER_FOOTPRINT*.
 - *bloodstream*: String nombre del Aditi Net al que se enviará el mensaje.
 - *seed*: String con el identificador del mensaje. Una sobrecarga permite enviar el mensaje sin este parámetro.
 - *packageMessage/data*: una sobrecarga del método admite adjuntar un objeto Package Message; otra sobrecarga adjunta la información mediante un arreglo de bytes.

- Estos son los parámetros de las sobrecargas del método **sendNonTransactional** solicitando por **número** de servicio:
 - *transaction*: transacción de la base de datos MattDB embebida en la Aditi Cell que permite formar el mensaje para su envío al Aditi Net.
 - *business*: Short con el negocio de la célula destinataria del mensaje.
 - *system*: Short con el sistema de la célula destinataria del mensaje.
 - *entity*: Short con la entidad de la célula destinataria del mensaje.
 - *footprint*: Integer con la huella de la célula a la que se le envía el mensaje.
 - *serviceNumber*: String con el mismo valor con el atributo name de la anotación ServiceInformation presente en el servicio solicitado.
 - *bloodstream*: String nombre del Aditi Net al que se enviará el mensaje.
 - *seed*: String con el identificador del mensaje. Una sobrecarga permite enviar el mensaje sin este parámetro.
 - *packageMessage/data*: una sobrecarga del método admite adjuntar un objeto Package Message; otra sobrecarga adjunta la información mediante un arreglo de bytes.



El parámetro *seed* es análogo al evento de los mensajes transaccionales: sirve para que la célula identifique solicitudes de servicio repetidas y garantice una única ejecución.

Ejemplo 7:



Servicio Propiedades (NTx)

Dificultad

- Este ejemplo consiste en crear un servicio que obtenga algunas propiedades del sistema, las guarde en un Package Message y las envíe a otro servicio mediante un mensaje No Transaccional.
- Mostraremos cómo solicitar un servicio mediante nombre y también mediante número de servicio. Recuerda que el MessageFactory tiene varias sobrecargas para el método sendNonTransactional, aquí mostraremos solamente dos de ellas que te servirán de guía para dominar por completo el envío de este tipo de mensajes.
- La parte más importante que exemplificaremos es el uso de la transacción de MattDB que requieren todos los métodos *sendNonTransactional*. Si deseas conocer más acerca de este Manejador de Base de Datos creado por Mithikel ingresa a <http://mithikel.com/matt-db--english.html>.
- Para el ejemplo del mensaje **No Transaccional** con **nombre de servicio**, creamos una clase llamada *ServicePropertiesNTx* que extienda de la superclase *Service* de la API de Aditi y la anotamos con *ServiceInformation* y los siguientes parámetros:
 - serviceNumber: short 10.
 - protocol: Protocol.ALL, para que pueda ser solicitado por cualquier mensaje.
 - name: *ServicePropertiesNTx*, para continuar con la convención de nombres usada hasta el momento.

```
14  @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServicePropertiesNTx")
15  public class ServicePropertiesNTx extends Service {
```

- Agregamos el Logger de log4j para escribir en la bitácora de la Aditi Cell.

```

17     /**
18      * Logger
19     */
20     private static final Logger LOGGER = Logger.getLogger(ServicePropertiesNTx.class);

```

- Declaramos una variable de tipo Package Message que utilizaremos para almacenar los atributos para el servicio que solicitemos en el mensaje.

```

21     /**
22      * Package Message
23     */
24     PackageMessage packageMessage;

```

- La explicación del método *run* la vamos a dividir en tres partes. En esta primera simplemente asignamos valor a la variable packageMessage y obtenemos algunas propiedades del sistema mediante System.getProperty() que almacenamos en variables String.

```

26     @Override
27     public void run() throws Exception {
28         /**
29          * Create a Package Message
30         */
31         packageMessage = new PackageMessage();
32         /**
33          * Get system properties
34         */
35         String username = System.getProperty("user.name");
36         String home = System.getProperty("user.home");
37         String jdkVersion = System.getProperty("java.version");
38         String os = System.getProperty("os.name");
39         String osVersion = System.getProperty("os.version");

```

- Posteriormente, guardamos las variables de las propiedades del sistema que obtuvimos en el Package Message usando el método *putData*. A cada una de las propiedades que guardamos les asignamos una llave String para que puedan ser obtenidas en el servicio de destino.

```

40     /**
41      * Store the system properties into Package Message instance
42     */
43     packageMessage.putData("username", username);
44     packageMessage.putData("home", home);
45     packageMessage.putData("jdkVersion", jdkVersion);
46     packageMessage.putData("os", os);
47     packageMessage.putData("osVersion", osVersion);

```

- La última parte es la más importante. Creamos una instancia de *SecureSenderTransaction* para poder acceder a una transacción del mecanismo de envío de mensajes de la célula. Esta clase tiene dos métodos:
 - *businessSender*: donde se encuentra disponible la transacción y donde codificaremos el envío del mensaje mediante el *MessageFactory*.
 - *lockTimeOutException*: donde se maneja la excepción arrojada por la base de datos de la célula cuando el enviador consume el tiempo para ejecutar la transacción.

```

48  /*
49   * Create the MattDB transaction
50   */
51  SecureSenderTransaction<Void> sendMessage = new SecureSenderTransaction<Void>() {

```

- Dentro de la función *businessSender* obtenemos el bloodstream al que está conectada la célula mediante el singleton Bloodstream y la función que se especifica en la línea 57 del ejemplo.
- También generamos el identificador *seed* del mensaje utilizando el timestamp del instante en el que se ejecuta el servicio.
- Declaramos que el valor del footprint será *integer.MAX_VALUE* para que el mensaje sea enviado por broadcast a todas las células que ofrezcan el servicio *ServicePrintProperties* (en un ejemplo siguiente lo construiremos para explicar el acceso a los atributos del package message).

```

54  /**
55   * Message parameters of Send Non Transactional
56   */
57  String bloodstream = BloodStream.getInstance().getBloodStreamActiveNetworkSockets().keySet().iterator().next();
58  String seed = System.currentTimeMillis() + "";
59  Integer footprint = Integer.MAX_VALUE;

```

- Para que la transacción sea aplicada invocamos el método run de la instancia del SecureSenderTransaction.
- El código de toda esta explicación del sender de la célula es el siguiente:

```
48 v      /**
49   * Create the MattDB transaction
50   */
51 v     SecureSenderTransaction<Void> sendMessage = new SecureSenderTransaction<Void>() {
52       @Override
53 v         protected Void businessSender(Transaction transaction) throws Exception {
54 v           /**
55             * Message parameters of Send Non Transactional
56             */
57             String bloodstream = BloodStream.getInstance().getBloodStreamActiveNetworkSockets().keySet().iterator().next();
58             String seed = System.currentTimeMillis() + "";
59             Integer footprint = Integer.MAX_VALUE;
60 v           /**
61             * Send the message
62             */
63 v             MessageFactory.getInstance().sendNonTransactional(transaction, "ServicePrintProperties", footprint,
64               bloodstream, seed, packageMessage);
65             return null;
66         }
67
68         @Override
69         protected void lockTimeOutException(LockTimeOutException ltoe) {
70             LOGGER.error("Sending the message", ltoe);
71         }
72     };
73     sendMessage.run();
74 }
75 }
```

- El código completo del Servicio Properties que manda un mensaje *No Transaccional* con *nombre* de Servicio.

```

1 package org.mithikel.aditi.demo;
2
3 import org.apache.log4j.Logger;
4 import org.mithikel.aditi.cell.bloodstream.BloodStream;
5 import org.mithikel.aditi.cell.factory.MessageFactory;
6 import org.mithikel.aditi.cell.rs.service.Protocol;
7 import org.mithikel.aditi.cell.rs.service.Service;
8 import org.mithikel.aditi.cell.rs.service.ServiceInformation;
9 import org.mithikel.aditi.cell.sender.SecureSenderTransaction;
10 import org.mithikel.aditi.messages.PackageMessage;
11 import org.mithikel.mattdb.control.Transaction;
12 import org.mithikel.mattdb.exception.LockTimeOutException;
13
14 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServicePropertiesNTx")
15 public class ServicePropertiesNTx extends Service {
16
17     /**
18      * Logger
19      */
20     private static final Logger LOGGER = Logger.getLogger(ServicePropertiesNTx.class);
21
22     /**
23      * Package Message
24      */
25     PackageMessage packageMessage;
26
27     @Override
28     public void run() throws Exception {
29         /**
30          * Create a Package Message
31          */
32         packageMessage = new PackageMessage();
33
34         /**
35          * Get system properties
36          */
37         String username = System.getProperty("user.name");
38         String home = System.getProperty("user.home");
39         String jdkVersion = System.getProperty("java.version");
40         String os = System.getProperty("os.name");
41         String osVersion = System.getProperty("os.version");
42
43         /**
44          * Store the system properties into Package Message instance
45          */
46         packageMessage.putData("username", username);
47         packageMessage.putData("home", home);
48         packageMessage.putData("jdkVersion", jdkVersion);
49         packageMessage.putData("os", os);
50         packageMessage.putData("osVersion", osVersion);
51
52         /**
53          * Create the MattDB transaction
54          */
55         SecureSenderTransaction<Void> sendMessage = new SecureSenderTransaction<Void>() {
56             @Override
57             protected Void businessSender(Transaction transaction) throws Exception {
58                 /**
59                  * Message parameters of Send Non Transactional
60                  */
61                 String bloodstream = BloodStream.getInstance().getBloodStreamActiveNetworkSockets().keySet().iterator().next();
62                 String seed = System.currentTimeMillis() + "";
63                 Integer footprint = Integer.MAX_VALUE;
64
65                 /**
66                  * Send the message
67                  */
68                 MessageFactory.getInstance().sendNonTransactional(transaction, "ServicePrintProperties", footprint,
69                             bloodstream, seed, packageMessage);
70                 return null;
71             }
72
73             @Override
74             protected void lockTimeOutException(LockTimeOutException ltoe) {
75                 LOGGER.error("Sending the message", ltoe);
76             }
77         };
78         sendMessage.run();
79     }
80
81     @Override
82     public void exception(Throwable throwable) throws Exception {
83         LOGGER.error("Service Properties NTx", throwable);
84     }
85 }
```

- Para el ejemplo del mensaje **No Transaccional** con **número de servicio**, creamos una clase llamada *ServicePropertiesNTxServiceNumber* que extienda de la superclase *Service* de la API de Aditi y la anotamos con *ServiceInformation* y los siguientes parámetros:
 - serviceNumber*: short 10.
 - protocol*: Protocol.ALL, para que pueda ser solicitado por cualquier mensaje.
 - name*: *ServicePropertiesNTxServiceNumber*, para continuar con la convención de nombres usada hasta el momento.

```
14 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServicePropertiesNTxServiceNumber")
15 public class ServicePropertiesNTxServiceNumber extends Service {
```

- Al igual que en el ejemplo anterior, las variables de clase serán el logger de la célula y el package message.

```
17 /**
18  * Logger
19 */
20 private static final Logger LOGGER = Logger.getLogger(ServicePropertiesNTx.class);
21 /**
22  * Package Message
23 */
24 PackageMessage packageMessage;
25
```

- En la primera parte del método *run* obtenemos las propiedades del sistema y las guardamos en el package message.

```
26 @Override
27 public void run() throws Exception {
28 /**
29  * Create a Package Message
30 */
31 packageMessage = new PackageMessage();
32 /**
33  * Get system properties
34 */
35 String username = System.getProperty("user.name");
36 String home = System.getProperty("user.home");
37 String jdkVersion = System.getProperty("java.version");
38 String os = System.getProperty("os.name");
39 String osVersion = System.getProperty("os.version");
40 /**
41  * Store the system properties into Package Message instance
42 */
43 packageMessage.putData("username", username);
44 packageMessage.putData("home", home);
45 packageMessage.putData("jdkVersion", jdkVersion);
46 packageMessage.putData("os", os);
47 packageMessage.putData("osVersion", osVersion);
```

- En la segunda parte del método *run* creamos la instancia de *SecureSenderTransaction* justo como en el ejemplo anterior.
- Obtenemos los mismos parámetros para el mensaje que el ejemplo del mensaje con nombre de servicio.
- Lo único diferente que haremos para esta la sobrecarga de *sendNonTransactional* es especificar el N.S.E. de la célula a quien va dirigida. Para nuestro ejemplo utilizaremos los atributos de la clase del servicio que indican la identidad de la célula que solicitó el servicio que se está ejecutando. En otras palabras, vamos a mandar como respuesta las propiedades del sistema a todas las células del tipo que solicitó el servicio *Properties*.
- Por último, para que la transacción sea aplicada invocamos el método *run* de la instancia del *SecureSenderTransaction*.

```

48     /**
49      * Create the MattDB transaction
50     */
51     SecureSenderTransaction<Void> sendMessage = new SecureSenderTransaction<Void>() {
52         @Override
53         protected Void businessSender(Transaction transaction) throws Exception {
54             /**
55              * Message parameters of Send Non Transactional
56             */
57             String bloodstream = BloodStream.getInstance().getBloodStreamActiveNetworkSockets().keySet().iterator().next();
58             String seed = System.currentTimeMillis() + "";
59             Integer footprint = Integer.MAX_VALUE;
60             /**
61              * Send the message
62             */
63             MessageFactory.getInstance().sendNonTransactional(transaction, requesterBusiness, requesterSystem, requesterEntity,
64                     footprint, (short) 11, bloodstream, seed, packageMessage);
65             return null;
66         }
67
68         @Override
69         protected void lockTimeOutException(LockTimeOutException ltoe) {
70             LOGGER.error("Sending the message", ltoe);
71         }
72     };
73     sendMessage.run();
74 }
```

- El código completo del Servicio Properties que manda un mensaje *No Transaccional* con *número de Servicio*.

```

1 package org.mithikel.aditi.demo;
2
3 import org.apache.log4j.Logger;
4 import org.mithikel.aditi.cell.bloodstream.BloodStream;
5 import org.mithikel.aditi.cell.factory.MessageFactory;
6 import org.mithikel.aditi.cell.rs.service.Protocol;
7 import org.mithikel.aditi.cell.rs.service.Service;
8 import org.mithikel.aditi.cell.rs.service.ServiceInformation;
9 import org.mithikel.aditi.cell.sender.SecureSenderTransaction;
10 import org.mithikel.aditi.messages.PackageMessage;
11 import org.mithikel.mattdb.control.Transaction;
12 import org.mithikel.mattdb.exception.LockTimeOutException;
13
14 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServicePropertiesNTxServiceNumber")
15 public class ServicePropertiesNTxServiceNumber extends Service {
16
17     /**
18      * Logger
19      */
20     private static final Logger LOGGER = Logger.getLogger(ServicePropertiesNTx.class);
21
22     /**
23      * Package Message
24      */
25     PackageMessage packageMessage;
26
27     @Override
28     public void run() throws Exception {
29
30         /**
31          * Create a Package Message
32          */
33         packageMessage = new PackageMessage();
34
35         /**
36          * Get system properties
37          */
38         String username = System.getProperty("user.name");
39         String home = System.getProperty("user.home");
40         String jdkVersion = System.getProperty("java.version");
41         String os = System.getProperty("os.name");
42         String osVersion = System.getProperty("os.version");
43
44         /**
45          * Store the system properties into Package Message instance
46          */
47         packageMessage.putData("username", username);
48         packageMessage.putData("home", home);
49         packageMessage.putData("jdkVersion", jdkVersion);
50         packageMessage.putData("os", os);
51         packageMessage.putData("osVersion", osVersion);
52
53         /**
54          * Create the MattDB transaction
55          */
56         SecureSenderTransaction<Void> sendMessage = new SecureSenderTransaction<Void>() {
57             @Override
58             protected Void businessSender(Transaction transaction) throws Exception {
59
60                 /**
61                  * Message parameters of Send Non Transactional
62                  */
63                 String bloodstream = BloodStream.getInstance().getBloodStreamActiveNetworkSockets().keySet().iterator().next();
64                 String seed = System.currentTimeMillis() + "";
65                 Integer footprint = Integer.MAX_VALUE;
66
67                 /**
68                  * Send the message
69                  */
70                 MessageFactory.getInstance().sendNonTransactional(transaction, requesterBusiness, requesterSystem, requesterEntity,
71                             footprint, (short) 11, bloodstream, seed, packageMessage);
72                 return null;
73             }
74
75             @Override
76             protected void lockTimeOutException(LockTimeOutException ltoe) {
77                 LOGGER.error("Sending the message", ltoe);
78             }
79         };
80         sendMessage.run();
81     }
82
83     @Override
84     public void exception(Throwable throwable) throws Exception {
85         LOGGER.error("Service Properties NTx", throwable);
86     }
87 }
```

Ejemplo 8:

Servicio Propiedades (Tx)

Dificultad 

- Este ejemplo consiste en crear un servicio que obtenga algunas propiedades del sistema, las guarde en un Package Message y las envíe a otro servicio mediante un mensaje No Transaccional.
- Mostraremos cómo solicitar un servicio mediante nombre y también mediante número de servicio. Recuerda que el MessageFactory tiene varias sobrecargas para el método sendNonTransactional, aquí mostraremos solamente dos de ellas que te servirán de guía para dominar por completo el envío de este tipo de mensajes.
- La parte más importante que exemplificaremos es el uso de la transacción de MattDB que requieren el todos los métodos *sendNonTransactional*. Si deseas conocer más acerca de este Manejador de Base de Datos creado por Mithikel ingresa a <http://mithikel.com/matt-db--english.html>.
- Para el ejemplo del mensaje **Transaccional** con **nombre de servicio**, creamos una clase llamada *ServicePropertiesNTx* que extienda de la superclase *Service* de la API de Aditi y la anotamos con *ServiceInformation* y los siguientes parámetros:
 - serviceNumber: short 10.
 - protocol: Protocol.ALL, para que pueda ser solicitado por cualquier mensaje.
 - name: *ServicePropertiesNTx*, para continuar con la convención de nombres usada hasta el momento.

```
14 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServicePropertiesTx")
15 public class ServicePropertiesTx extends Service {
```

- Agregamos el Logger de log4j para escribir en la bitácora de la Aditi Cell.

```

17 /**
18 * Logger
19 */
20 private static final Logger LOGGER = Logger.getLogger(ServicePropertiesNTx.class);

```

- Declaramos una variable de tipo Package Message que utilizaremos para almacenar los atributos para el servicio que solicitemos en el mensaje.

```

21 /**
22 * Package Message
23 */
24 PackageMessage packageMessage;

```

- La explicación del método *run* la vamos a dividir en tres partes. En esta primera simplemente asignamos valor a la variable packageMessage y obtenemos algunas propiedades del sistema mediante System.getProperty() que almacenamos en variables String.

```

26 @Override
27 public void run() throws Exception {
28 /**
29 * Create a Package Message
30 */
31 packageMessage = new PackageMessage();
32 /**
33 * Get system properties
34 */
35 String username = System.getProperty("user.name");
36 String home = System.getProperty("user.home");
37 String jdkVersion = System.getProperty("java.version");
38 String os = System.getProperty("os.name");
39 String osVersion = System.getProperty("os.version");

```

- Posteriormente, guardamos las variables de las propiedades del sistema que obtuvimos en el Package Message usando el método *putData*. A cada una de las propiedades que guardamos les asignamos una llave String para que puedan ser obtenidas en el servicio de destino.

```

40 /**
41 * Store the system properties into Package Message instance
42 */
43 packageMessage.putData("username", username);
44 packageMessage.putData("home", home);
45 packageMessage.putData("jdkVersion", jdkVersion);
46 packageMessage.putData("os", os);
47 packageMessage.putData("osVersion", osVersion);

```

- La última parte es la más importante. Creamos una instancia de *SecureSenderTransaction* para poder acceder a una transacción del mecanismo de envío de mensajes de la célula. Esta clase tiene dos métodos:
 - *businessSender*: donde se encuentra disponible la transacción y donde codificaremos el envío del mensaje mediante el *MessageFactory*.
 - *lockTimeOutException*: donde se maneja la excepción arrojada por la base de datos de la célula cuando el enviador consume el tiempo para ejecutar la transacción.
- Dentro de la función *businessSender* obtenemos el bloodstream al que está conectada la célula mediante el singleton Bloodstream y la función que se especifica en la línea 57 del ejemplo. También generamos el identificador *seed* del mensaje utilizando el timestamp del instante en el que se ejecuta el servicio.
- Por último invocamos a la función *sendTransactional* del *MessageFactory*, empleando los parámetros explicados en el punto anterior, solicitando el servicio con el nombre *ServicePrintProperties*.
- Para que la transacción sea aplicada invocamos el método *run* de la instancia del *SecureSenderTransaction*.

```

48     /**
49      * Create the MattDB transaction
50      */
51     SecureSenderTransaction<Void> sendMessage = new SecureSenderTransaction<Void>() {
52         @Override
53         protected Void businessSender(Transaction transaction) throws Exception {
54             /**
55              * Message parameters of Send Non Transactional
56              */
57             String bloodstream = BloodStream.getInstance().getBloodStreamActiveNetworkSockets().keySet().iterator().next();
58             String seed = System.currentTimeMillis() + "";
59             /**
60              * Send the message
61              */
62             MessageFactory.getInstance().sendTransactional(transaction, "ServicePrintProperties",
63                 bloodstream, seed, packageMessage);
64             return null;
65         }
66
67         @Override
68         protected void lockTimeOutException(LockTimeOutException ltoe) {
69             LOGGER.error("Sending the message", ltoe);
70         }
71     };
72     sendMessage.run();
73 }
```

- El código completo del Servicio Properties que manda un mensaje *Transaccional con nombre* de Servicio.

```

1 package org.mithikel.aditi.demo;
2
3 import org.apache.log4j.Logger;
4 import org.mithikel.aditi.cell.bloodstream.BloodStream;
5 import org.mithikel.aditi.cell.factory.MessageFactory;
6 import org.mithikel.aditi.cell.rs.service.Protocol;
7 import org.mithikel.aditi.cell.rs.service.Service;
8 import org.mithikel.aditi.cell.rs.service.ServiceInformation;
9 import org.mithikel.aditi.cell.sender.SecureSenderTransaction;
10 import org.mithikel.aditi.messages.PackageMessage;
11 import org.mithikel.mattdb.control.Transaction;
12 import org.mithikel.mattdb.exception.LockTimeOutException;
13
14 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServicePropertiesTx")
15 public class ServicePropertiesTx extends Service {
16
17     /**
18      * Logger
19      */
20     private static final Logger LOGGER = Logger.getLogger(ServicePropertiesNTx.class);
21
22     /**
23      * Package Message
24      */
25     PackageMessage packageMessage;
26
27     @Override
28     public void run() throws Exception {
29         /**
30          * Create a Package Message
31          */
32         packageMessage = new PackageMessage();
33
34         /**
35          * Get system properties
36          */
37         String username = System.getProperty("user.name");
38         String home = System.getProperty("user.home");
39         String jdkVersion = System.getProperty("java.version");
40         String os = System.getProperty("os.name");
41         String osVersion = System.getProperty("os.version");
42
43         /**
44          * Store the system properties into Package Message instance
45          */
46         packageMessage.putData("username", username);
47         packageMessage.putData("home", home);
48         packageMessage.putData("jdkVersion", jdkVersion);
49         packageMessage.putData("os", os);
50         packageMessage.putData("osVersion", osVersion);
51
52         /**
53          * Create the MattDB transaction
54          */
55         SecureSenderTransaction<Void> sendMessage = new SecureSenderTransaction<Void>() {
56             @Override
57             protected Void businessSender(Transaction transaction) throws Exception {
58                 /**
59                  * Message parameters of Send Non Transactional
60                  */
61                 String bloodstream = BloodStream.getInstance().getBloodStreamActiveNetworkSockets().keySet().iterator().next();
62                 String seed = System.currentTimeMillis() + "";
63
64                 /**
65                  * Send the message
66                  */
67                 MessageFactory.getInstance().sendTransactional(transaction, "ServicePrintProperties",
68                     bloodstream, seed, packageMessage);
69                 return null;
70             }
71
72             @Override
73             protected void lockTimeOutException(LockTimeOutException ltoe) {
74                 LOGGER.error("Sending the message", ltoe);
75             }
76         };
77         sendMessage.run();
78     }
79
80     @Override
81     public void exception(Throwable throwable) throws Exception {
82         LOGGER.error("Service Properties NTx", throwable);
83     }
84 }
```

- Para el ejemplo del mensaje **Transaccional con número de servicio**, creamos una clase llamada *ServicePropertiesTxServiceNumber* que extienda de la superclase *Service* de la API de Aditi y la anotamos con *ServiceInformation* y los siguientes parámetros:
 - serviceNumber*: short 10.
 - protocol*: Protocol.ALL, para que pueda ser solicitado por cualquier mensaje.
 - name*: *ServicePropertiesTxServiceNumber*, para continuar con la convención de nombres usada hasta el momento.

```
14 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServicePropertiesTxServiceNumber")
15 public class ServicePropertiesTxServiceNumber extends Service {
```

- Al igual que en el ejemplo anterior, las variables de clase serán el logger de la célula y el package message.

```
17 /**
18 * Logger
19 */
20 private static final Logger LOGGER = Logger.getLogger(ServicePropertiesNTx.class);
21 /**
22 * Package Message
23 */
24 PackageMessage packageMessage;
25
```

- En la primera parte del método *run* obtenemos las propiedades del sistema y las guardamos en el package message.

```
26 @Override
27 public void run() throws Exception {
28 /**
29 * Create a Package Message
30 */
31 packageMessage = new PackageMessage();
32 /**
33 * Get system properties
34 */
35 String username = System.getProperty("user.name");
36 String home = System.getProperty("user.home");
37 String jdkVersion = System.getProperty("java.version");
38 String os = System.getProperty("os.name");
39 String osVersion = System.getProperty("os.version");
40 /**
41 * Store the system properties into Package Message instance
42 */
43 packageMessage.putData("username", username);
44 packageMessage.putData("home", home);
45 packageMessage.putData("jdkVersion", jdkVersion);
46 packageMessage.putData("os", os);
47 packageMessage.putData("osVersion", osVersion);
```

- En la segunda parte del método *run* creamos la instancia de *SecureSenderTransaction* justo como en el ejemplo anterior.
- Obtenemos los mismos parámetros para el mensaje que el ejemplo del mensaje con nombre de servicio.
- Lo diferente que haremos para esta la sobrecarga de *sendTransactional* es especificar el N.S.E. de la célula a quien va dirigida y el número de servicio.
- Para nuestro ejemplo utilizaremos el número de servicio 11 y los atributos de la clase del servicio que indican la identidad de la célula que solicitó el servicio que se está ejecutando. En otras palabras, vamos a mandar como respuesta las propiedades del sistema a todas las células del tipo que solicitó el servicio *Properties*.
- Por último, para que la transacción sea aplicada invocamos el método *run* de la instancia del *SecureSenderTransaction*.

```

48     /**
49      * Create the MattDB transaction
50     */
51     SecureSenderTransaction<Void> sendMessage = new SecureSenderTransaction<Void>() {
52         @Override
53         protected Void businessSender(Transaction transaction) throws Exception {
54             /**
55              * Message parameters of Send Non Transactional
56             */
57             String bloodstream = BloodStream.getInstance().getBloodStreamActiveNetworkSockets().keySet().iterator().next();
58             String seed = System.currentTimeMillis() + "";
59             /**
60              * Send the message
61             */
62             MessageFactory.getInstance().sendTransactional(transaction, requesterBusiness, requesterSystem, requesterEntity,
63                 (short) 11, bloodstream, seed, packageMessage);
64             return null;
65         }
66
67         @Override
68         protected void lockTimeOutException(LockTimeOutException ltoe) {
69             LOGGER.error("Sending the message", ltoe);
70         }
71     };
72     sendMessage.run();
73 }
```

- El código completo del Servicio Properties que manda un mensaje *Transaccional* con *número de Servicio*.

```

1 package org.mithikel.aditi.demo.servicepropertiestx;
2
3 import org.apache.log4j.Logger;
4 import org.mithikel.aditi.cell.bloodstream.BloodStream;
5 import org.mithikel.aditi.cell.factory.MessageFactory;
6 import org.mithikel.aditi.cell.rs.service.Protocol;
7 import org.mithikel.aditi.cell.rs.service.Service;
8 import org.mithikel.aditi.cell.rs.service.ServiceInformation;
9 import org.mithikel.aditi.cell.sender.SecureSenderTransaction;
10 import org.mithikel.aditi.messages.PackageMessage;
11 import org.mithikel.mattdb.control.Transaction;
12 import org.mithikel.mattdb.exception.LockTimeOutException;
13
14 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServicePropertiesTxServiceNumber")
15 public class ServicePropertiesTxServiceNumber extends Service {
16
17     /**
18      * Logger
19      */
20     private static final Logger LOGGER = Logger.getLogger(ServicePropertiesTx.class);
21
22     /**
23      * Package Message
24      */
25     PackageMessage packageMessage;
26
27     @Override
28     public void run() throws Exception {
29
30         /**
31          * Create a Package Message
32          */
33         packageMessage = new PackageMessage();
34
35         /**
36          * Get system properties
37          */
38         String username = System.getProperty("user.name");
39         String home = System.getProperty("user.home");
40         String jdkVersion = System.getProperty("java.version");
41         String os = System.getProperty("os.name");
42         String osVersion = System.getProperty("os.version");
43
44         /**
45          * Store the system properties into Package Message instance
46          */
47         packageMessage.putData("username", username);
48         packageMessage.putData("home", home);
49         packageMessage.putData("jdkVersion", jdkVersion);
50         packageMessage.putData("os", os);
51         packageMessage.putData("osVersion", osVersion);
52
53         /**
54          * Create the MattDB transaction
55          */
56         SecureSenderTransaction<Void> sendMessage = new SecureSenderTransaction<Void>() {
57             @Override
58             protected Void businessSender(Transaction transaction) throws Exception {
59
60                 /**
61                  * Message parameters of Send Non Transactional
62                  */
63                 String bloodstream = BloodStream.getInstance().getBloodStreamActiveNetworkSockets().keySet().iterator().next();
64                 String seed = System.currentTimeMillis() + "";
65
66                 /**
67                  * Send the message
68                  */
69                 MessageFactory.getInstance().sendTransactional(transaction, requesterBusiness, requesterSystem, requesterEntity,
70                     (short) 11, bloodstream, seed, packageMessage);
71                 return null;
72             }
73
74             @Override
75             protected void lockTimeOutException(LockTimeOutException ltoe) {
76                 LOGGER.error("Sending the message", ltoe);
77             }
78         };
79         sendMessage.run();
80     }
81
82     @Override
83     public void exception(Throwable throwable) throws Exception {
84         LOGGER.error("Service Properties NTx", throwable);
85     }
86 }
```

¿Cómo acceder a los atributos de entrada?



- Para hacer uso de los atributos recibidos en el *Package Message* del mensaje con el que se solicitó el servicio a ejecutar se emplea la anotación **InputResource** en la declaración de una variable de clase que sea del mismo tipo que el atributo. La anotación tiene un solo parámetro:
 - key: es el *String* de la llave con la que se guardó el atributo en el *PackageMessage*.



En caso de que el parámetro *key* no coincida con ningún atributo del *Package Message*, el valor la variable de clase permanecerá nula en la ejecución del servicio.



La variable de clase que reciba el atributo de entrada debe ser del mismo tipo que el objeto que fue guardado en el *Package Message*.

- Así es la estructura de un servicio que accede a un atributo de entrada:

```
1  @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "MyService")
2  public class ServiceClock extends Service {
3  /**
4   * Input Attribute Object
5   */
6  @InputAttribute(key = "key")—————→ Anotación con la información del Atributo solicitado
7  private Object attribute;
8  _____|_____ Variable de clase a la que se asignará el Atributo
9  @Override
10 public void run() throws Exception {
11 /**
12   * Your code using the input attribute
13   */
14 }
15
16 @Override
17 public void exception(Throwable throwable) throws Exception {
18 /**
19   * Your code to handle exceptions
20   */
21 }
22
23 }
```

Ejemplo 9: Servicio Ping Pong

Dificultad 

- Este es un ejemplo muy sencillo para mostrar cómo usar los atributos recibidos en el package message del mensaje que solicitó el servicio.
- Este servicio evalúa el valor del atributo de entrada y genera una respuesta que es enviada a la célula que solicitó el servicio. Como su nombre lo sugiere, si el atributo de entrada es un "Ping", la respuesta será un "Pong" y viceversa.
- El primer paso es crear una clase llamada *ServicePingPong* que extienda de la superclase *Service* de la API de Aditi y que esté anotada con *ServiceInformation*. El número de servicio será el 11, aceptará solicitudes de ambos protocolos (*protocol=Protocol.ALL*) y el nombre será el mismo que el de la clase (*name=ServicePingPong*).

```
16 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServicePingPong")
17 public class ServicePingPong extends Service {
```

- El primer atributo de la clase que declaramos es el Logger de la Aditi Cell.

```
19 /**
20  * Logger
21 */
22 private static final Logger LOGGER = Logger.getLogger(ServicePingPong.class);
```

- Declaramos un atributo package message para enviar la respuesta.

```
28 /**
29  * Package message
30 */
31 private PackageMessage packageMessage;
```

- Para acceder al atributos declaramos una variable String de nombre *pingPong*, con la anotación *InputAttribute* usando y la llave "*pingpong*". De esta manera, cuando la célula procese el mensaje de petición de servicio y construya la instancia del servicio, desempaquetará el atributo solicitado y asignará su valor a la variable.

```

23  /**
24  * Pingpong Attribute
25  */
26 @InputAttribute(key = "pingpong")
27 private String pingpong;

```

- En el método *run* realizamos una comparación para determinar cuál será la respuesta que enviará este servicio y guardamos el valor en el atributo package message con la llave "*pingpong*".

```

-- 
33     @Override
34     public void run() throws Exception {
35         if (objects.equals(pingpong, "Ping")) {
36             packageMessage.putData("pingpong", "Pong");
37         } else {
38             packageMessage.putData("pingpong", "Ping");
39         }

```

- Una vez empaquetada la respuesta, creamos la instancia de *SecureSenderTransaction* para formar el mensaje se salida en la cola de comunicaciones de la célula.
- En este ejemplo mandaremos la respuesta mediante un mensaje transaccional dirigido específicamente a la célula que solicitó el servicio. Para ello usamos el atributo *requesterFootprint* de la superclase.

```

40 /**
41 * Create the MattDB transaction
42 */
43 SecureSenderTransaction<Void> sendMessage = new SecureSenderTransaction<Void>() {
44     @Override
45     protected Void businessSender(Transaction transaction) throws Exception {
46         /**
47          * Message parameters of Send Non Transactional
48          */
49         String bloodstream = BloodStream.getInstance().getBloodStreamActiveNetworkSockets().keySet().iterator().next();
50         /**
51          * Send the message
52          */
53         MessageFactory.getInstance().sendNonTransactional(transaction, "ServicePingPongClient", requesterFootprint,
54                 bloodstream, packageMessage);
55         return null;
56     }
57
58     @Override
59     protected void lockTimeOutException(LockTimeOutException ltoe) {
60         LOGGER.error("Sending the message", ltoe);
61     }
62 };
63 sendMessage.run();
64 }

```

- El código completo del Servicio Ping Pong:

```

1 package org.mithikel.aditi.demo.servicepingpong;
2
3 import java.util.Objects;
4 import org.apache.log4j.Logger;
5 import org.mithikel.aditi.cell.bloodstream.BloodStream;
6 import org.mithikel.aditi.cell.factory.MessageFactory;
7 import org.mithikel.aditi.cell.rs.service.InputAttribute;
8 import org.mithikel.aditi.cell.rs.service.Protocol;
9 import org.mithikel.aditi.cell.rs.service.Service;
10 import org.mithikel.aditi.cell.rs.service.ServiceInformation;
11 import org.mithikel.aditi.cell.sender.SecureSenderTransaction;
12 import org.mithikel.aditi.messages.PackageMessage;
13 import org.mithikel.mattdb.control.Transaction;
14 import org.mithikel.mattdb.exception.LockTimeOutException;
15
16 @ServiceInformation(serviceNumber = 10, protocol = Protocol.ALL, name = "ServicePingPong")
17 public class ServicePingPong extends Service {
18
19     /**
20      * Logger
21      */
22     private static final Logger LOGGER = Logger.getLogger(ServicePingPong.class);
23
24     /**
25      * Pingpong Attribute
26      */
27     @InputAttribute(key = "pingpong")
28     private String pingpong;
29
30     /**
31      * Package message
32      */
33     private PackageMessage packageMessage;
34
35     @Override
36     public void run() throws Exception {
37         if (Objects.equals(pingpong, "Ping")) {
38             packageMessage.putData("pingpong", "Pong");
39         } else {
40             packageMessage.putData("pingpong", "Ping");
41         }
42         /**
43          * Create the MattDB transaction
44          */
45         SecureSenderTransaction<Void> sendMessage = new SecureSenderTransaction<Void>() {
46             @Override
47             protected Void businessSender(Transaction transaction) throws Exception {
48                 /**
49                  * Message parameters of Send Non Transactional
50                  */
51                 String bloodstream = BloodStream.getInstance().getBloodStreamActiveNetworkSockets().keySet().iterator().next();
52
53                 /**
54                  * Send the message
55                  */
56                 MessageFactory.getInstance().sendNonTransactional(transaction, "ServicePingPongClient", requesterFootprint,
57                             bloodstream, packageMessage);
58                 return null;
59             }
60
61             @Override
62             protected void lockTimeOutException(LockTimeOutException ltoe) {
63                 LOGGER.error("Sending the message", ltoe);
64             }
65         };
66         sendMessage.run();
67     }
68
69     @Override
70     public void exception(Throwable throwable) throws Exception {
71         LOGGER.error("ServicePrintProperties", throwable);
72     }
73 }
```

Ejemplo 10: Servicio Print Properties

Dificultad 

- En este ejemplo construiremos el complemento de *ServiceProperties*. La funcionalidad de este servicio será obtener los atributos que fueron enviados en el package message que se envió desde el *ServiceProperties*.
- Comenzamos como siempre, creando una clase llamada *ServicePrintProperties* que extienda de la superclase *Service* de la API de Aditi y que esté anotada con *ServiceInformation* y los siguientes parámetros:
 - *serviceNumber*: este servicio será el número 11 para que se corresponda con el código que escribimos en *ServiceProperties*.
 - *protocol*: *Protocol.ALL*, para que cualquier tipo de mensaje pueda solicitar este servicio.
 - *name*: ocupamos el nombre de la clase para continuar con la convención que hemos seguido en todos los ejemplos.

```
11  @ServiceInformation(serviceNumber = 11, protocol = Protocol.ALL, name = "ServicePrintProperties")
12  public class ServicePrintProperties extends Service {
```

- El primer atributo de la clase que declaramos es el Logger de la Aditi Cell.

```
14  /**
15  * Logger
16  */
17  private static final Logger LOGGER = Logger.getLogger(ServicePrintProperties.class);
```

- Solicitamos el recurso *ResourceAditiLogger* para emplearlo en la ejecución del servicio.

```
18  /**
19  * Aditi Logger Resource
20  */
21  @InputResource(resource = "ResourceAditiLogger")
22  AditiLogger aditiLogger;
```

- Para acceder a los atributos declaramos las variables String de los elementos a obtener y los anotamos con *InputAttribute* usando la llave con la que fueron guardados en el package message.

```

28  /**
29   * Attribute "home"
30   */
31 @InputAttribute(key = "home")
32 String home;
33 /**
34  * Attribute "jdkVersion"
35  */
36 @InputAttribute(key = "jdkVersion")
37 String jdkVersion;
38 /**
39  * Attribute "os"
40  */
41 @InputAttribute(key = "os")
42 String os;
43 /**
44  * Attribute "osVersion"
45  */
46 @InputAttribute(key = "osVersion")
47 String osVersion;
48

```

- En el método *run* invocamos el método *write* del Aditi Logger para registrar los valores de los atributos obtenidos.

```

--> 49 @Override
50 public void run() throws Exception {
51     aditiLogger.write("username: " + username);
52     aditiLogger.write("home: " + home);
53     aditiLogger.write("jdkVersion: " + jdkVersion);
54     aditiLogger.write("os: " + os);
55     aditiLogger.write("osVersion: " + osVersion);
56 }

```

- Para manejar excepciones únicamente las reportamos en la bitácora de la célula.

```

58 @Override
59 public void exception(Throwable throwable) throws Exception {
60     LOGGER.error("ServicePrintProperties", throwable);
61 }

```

- El código completo del Servicio Print Properties:

```
1 package org.mithikel.aditi.demo.serviceprintproperties;
2
3 import org.apache.log4j.Logger;
4 import org.mithikel.aditi.cell.rs.service.InputAttribute;
5 import org.mithikel.aditi.cell.rs.service.InputResource;
6 import org.mithikel.aditi.cell.rs.service.Protocol;
7 import org.mithikel.aditi.cell.rs.service.Service;
8 import org.mithikel.aditi.cell.rs.service.ServiceInformation;
9 import org.mithikel.aditi.demo.aditilogger.AditiLogger;
10
11 @ServiceInformation(serviceNumber = 11, protocol = Protocol.ALL, name = "ServicePrintProperties")
12 public class ServicePrintProperties extends Service {
13
14     /**
15      * Logger
16      */
17     private static final Logger LOGGER = Logger.getLogger(ServicePrintProperties.class);
18
19     * Aditi Logger Resource
20     */
21     @InputResource(resource = "ResourceAditiLogger")
22     AditiLogger aditiLogger;
23
24     * Attribute "username"
25     */
26     @InputAttribute(key = "username")
27     String username;
28
29     * Attribute "home"
30     */
31     @InputAttribute(key = "home")
32     String home;
33
34     * Attribute "jdkVersion"
35     */
36     @InputAttribute(key = "jdkVersion")
37     String jdkVersion;
38
39     * Attribute "os"
40     */
41     @InputAttribute(key = "os")
42     String os;
43
44     * Attribute "osVersion"
45     */
46     @InputAttribute(key = "osVersion")
47     String osVersion;
48
49
50     @Override
51     public void run() throws Exception {
52         aditiLogger.write("username: " + username);
53         aditiLogger.write("home: " + home);
54         aditiLogger.write("jdkVersion: " + jdkVersion);
55         aditiLogger.write("os: " + os);
56         aditiLogger.write("osVersion: " + osVersion);
57     }
58
59     @Override
60     public void exception(Throwable throwable) throws Exception {
61         LOGGER.error("ServicePrintProperties", throwable);
62     }
63 }
```

PARTE III:

Implantación y Operación



¿Cómo funciona el Aditi Manager?



- El Aditi Manager es el componente de la plataforma Aditi para administrar, monitorear y desplegar sistemas críticos de información. Permite controlar a los sistemas de la plataforma a nivel de software y a nivel de hardware.
- El Aditi Manager es en realidad un **sistema Aditi** integrado por **tres tipos de células**:
 - *Agent*: sirve para realizar instalaciones y agregar computadoras a la infraestructura.
 - *Server*: es el elemento que almacena la información de los sistemas controlados por el Aditi Manager (ambientes, artefactos, células, etc.).
 - *Client*: es la interfaz gráfica de usuario donde se monitorea y se ordenan las operaciones del Aditi Manager. Tiene tres ventanas: *Architecture*, *Environments* y *Artifacts*.
- Estas Aditi Cells pueden estar conectadas a más de un Bloodstreams del Aditi Net, pues pertenecen al **bloodstream** del **Aditi Manager** y a los **bloodstreams** de cada uno de los **ambientes** que controlan.

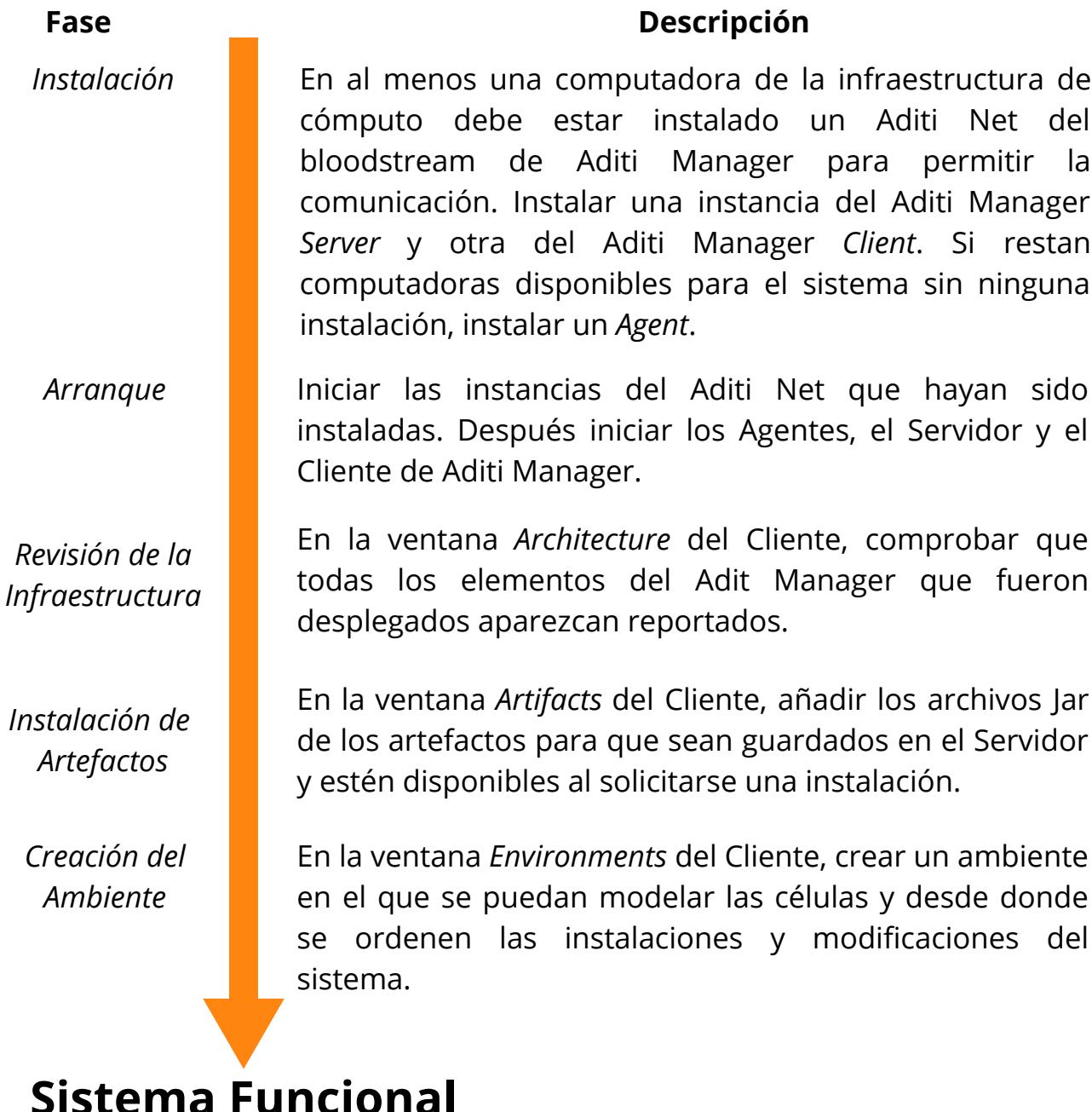


Ambiente:

Conjunto de Aditi Cells que implementan un sistema crítico de información, con una configuración en común y un bloodstream del Aditi Net.

- Tiene tres parámetros para adaptar la plataforma Aditi a sus necesidades:
 - *Selfrecovery cycles*: la cantidad de ciclos de intentos para obtener los acuses de los mensajes transaccionales antes de iniciar el mecanismo de **autorrecuperación**.
 - *Acknowledgment limit time*: el **tiempo límite** para obtener los acuses de los **mensajes transaccionales**.
 - *Tx message synchronization*: el número máximo de mensajes transaccionales que la célula guardará para realizar **sincronizaciones** con otras células.

¿Cómo se usa Aditi Manager?



¿Qué hace la ventana de Arquitectura?



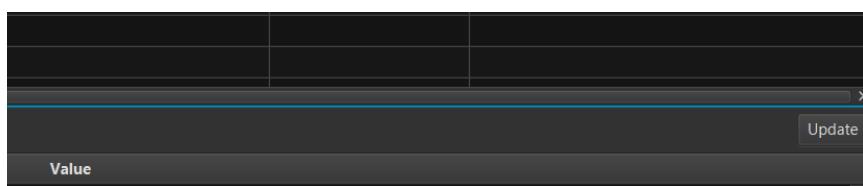
- La función de la ventana *Architecture* es mostrar en **tiempo real** a los componentes desplegados del Aditi Manager. La importancia de esto es que cualquier computadora donde se ejecute un componente del Manager forma parte de la **infraestructura** de la plataforma Aditi y puede ser utilizada para instalar las Aditi Cells de los ambientes.
- La ventana se despliega al dar click en este ícono del contenedor de ventanas del cliente.



- Esta es la ventana más sencilla del Cliente. Solo tiene dos funcionalidades además de mostrar información:
 - Listar las propiedades del equipo donde se encuentra instalado cada componente, al dar click en su nombre.

Property	Value
-	C:\Program Files\Java\jdk-14/bin/java
=::	::\
ACLOCAL_PATH	C:\Program Files\Git\mingw64\share\aclocal;C:\Program Files\Git\usr\share\aclocal
ALLUSERSPROFILE	C:\ProgramData
APPDATA	C:\Users\Gabriel\AppData\Roaming
COMMONPROGRAMFILES	C:\Program Files\Common Files
CommonProgramFiles(x86)	C:\Program Files (x86)\Common Files
CommonProgramW6432	C:\Program Files\Common Files
COMPUTERNAME	LAPTOP-8DKIVF0D
COMSPEC	C:\WINDOWS\system32\cmd.exe
CONFIG_SITE	C:/Program Files/Git/mingw64/etc/config.site
DISPLAY	needs-to-be-defined
DriverData	C:\Windows\System32\Drivers\DriverData
EXEPATH	C:\Program Files\Git

- Pedir que los componentes reenvíen toda su información asociada, con el botón *Update*.



botón *Update*

¿Qué hace la ventana de Artefactos?



- La ventana *Artifacts* sirve para guardar en el *Server* los artefactos que serán agregados a las Aditi Cells. También permite visualizar las dependencias y metadatos de los que ya se han almacenado, además de tener la función de descargar su archivo Jar.
- La ventana se despliega al dar click en este ícono del contenedor de ventanas del cliente.



- En este panel se despliegan todos los artefactos registrados en el *Server*:

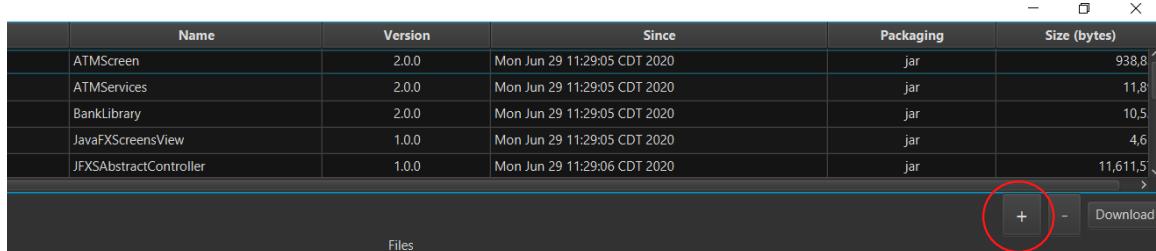
File Name	Group	Name	Version	Since	Packaging	Size (bytes)
ATMScreen-2.0.jar	org.mithikel.aditi.demo	ATMScreen	2.0.0	Mon Jun 29 11:29:05 CDT 2020	jar	938,8
ATMServices-2.0.0.jar	org.mithikel.aditi.demo	ATMServices	2.0.0	Mon Jun 29 11:29:05 CDT 2020	jar	11,8
BankLibrary-2.0.jar	org.mithikel.aditi.demo	BankLibrary	2.0.0	Mon Jun 29 11:29:05 CDT 2020	jar	10,5
JavaFXScreensView-1.0.0.jar	org.mithikel.aditi.addons	JavaFXScreensView	1.0.0	Mon Jun 29 11:29:05 CDT 2020	jar	4,6
JFXScreens-1.0.0-jar-with-dependencies.jar	org.mithikel	JFXSAbstractController	1.0.0	Mon Jun 29 11:29:06 CDT 2020	jar	11,611,5

- Al dar *click* en uno de ellos, su información se mostrará en el panel inferior:

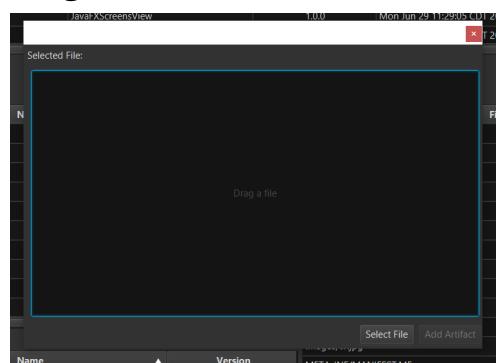
Dependencies			Files		
Group	Name	Version	File	File	Sign
\$project.groupId	BankLibrary	\$project.version	fxml/ATM.fxml	aad931b117fc222f51d416829f39fd68	
org.mithikel	JFXScreens	1.0.0	images/368402.png	16c8565b7cb6c86ecf81f64d45ce04	
			images/aditi-ajolote.png	88228022a7ccff81616bac925da7512b	
			images/atm.ico	a27ad0431b97e9967252efbdcc09fe01	
			images/atm.jpg	02f94248f66f510ca797c841a0a3d65	
			images/atm.png	3a76addc1a53f15d9e352a6810946ac9	
			images/atm2.ico	a27adb431b97e9967252efbdcc09fe01	
			images/bank.jpg	3b70580d18f2744880a1dfa9a850a5b	
			images/engrane.html	35c648eadd8b944f4c2c0b1193e69ad4	
			images/engrane.jpg	547cd8694bc710ab82c73d329e19557	
			images/Thumbs.db	68c2363ae36f78f2ee5e15d995a132dc	
			images/tv.jpg	80cb027f588e4ffa261b0d45ca8f6	
			META-INF/MANIFEST.MF	205f31be556c1fa875f4272783186556	
			META-INF/maven/org.mithikel.aditi.demo/ATMScreen/pom.properties	cc2fc1fdab2b0f1a4c6e89f50cd2a28	
			META-INF/maven/org.mithikel.aditi.demo/ATMScreen/pom.xml	fc5cb3cd8dbfe4220802caa633ef122	
			org/mithikel/aditi/demo/matrix/ATMController.class	b063ee073bcd8fc8306500c8a2b1a58	
			org/mithikel/aditi/demo/matrix/ATMController\$1.class	b5212d91c14eed8ebf5be0c99df20	
			org/mithikel/aditi/demo/matrix/ATMController\$2.class	1b93b64641e37f161b4b71267cef4b4	
			org/mithikel/aditi/demo/matrix/ATMController\$3.class	8a0fb2ea3d893d1b592d52b8cf3f0644	
			styles/dark_theme.css	26f9eaef15757f25be42876df28fe04d	

- Pasos para agregar un artefacto:

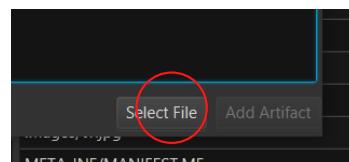
1. Dar *click* en el botón "+", ubicado en la parte inferior derecha del panel de artefactos.



2. Se abrirá la siguiente ventana emergente para que selecciones el archivo Jar del Artefacto a registrar.



3. Hay dos opciones para seleccionar el archivo. Puedes arrastrarlo directamente sobre la ventana o puedes abrir un selector de archivos presionando el botón Select File.



4. Una vez que termine la carga del archivo y estés seguro de subir el Artefacto, pulsa el botón Add Artifact para confirmar. Dependiendo de la latencia de la red, la operación podría tardar en reflejarse en la interfaz.



Para eliminar un artefacto del *Server* basta con seleccionarlo y dar click en el botón "-" que se encuentra del lado derecho del botón "+".

¿Qué hace la ventana de Ambientes?



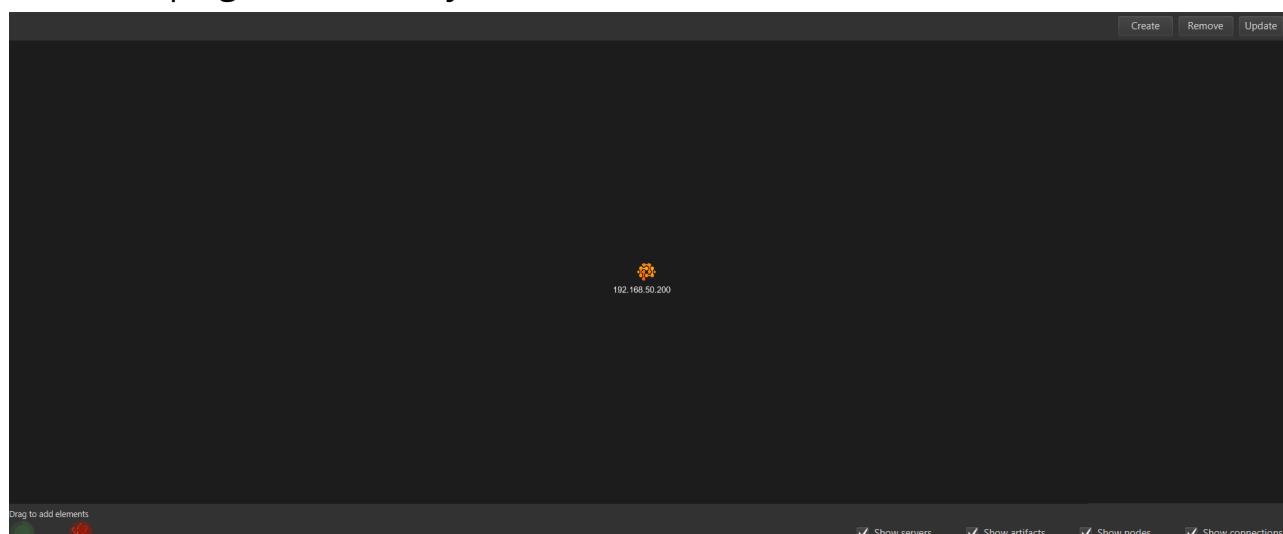
- La ventana *Environments* es la que ofrece las funcionalidades más importantes del Aditi Manager. Desde aquí se crean y configuran los ambientes; se ordena la instalación de instancias de las Aditi Cells y Aditi Net; se modela la funcionalidad de las células del sistema usando los artefactos.
- La ventana se despliega al dar click en este ícono del contenedor de ventanas del cliente.



- La estructura de la ventana se divide en dos secciones:
 - Panel de ambientes: muestra los ambientes creados y su configuración.

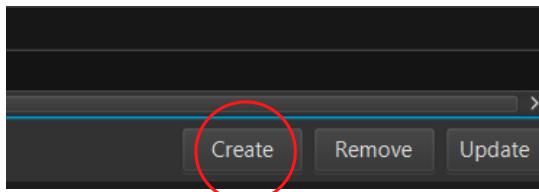
Environment	Self Recovery Cycles	Acknowledgment Limit Time (millis)	Tx Message SYNC	Port (Default)
Bank	4	5000	1000	31011

- Panel de creación: muestra los recursos de cómputo disponibles de la plataforma y la arquitectura del ambiente mediante un grafo. Permite desplegar Aditi Cells y Aditi Net de forma sencilla.

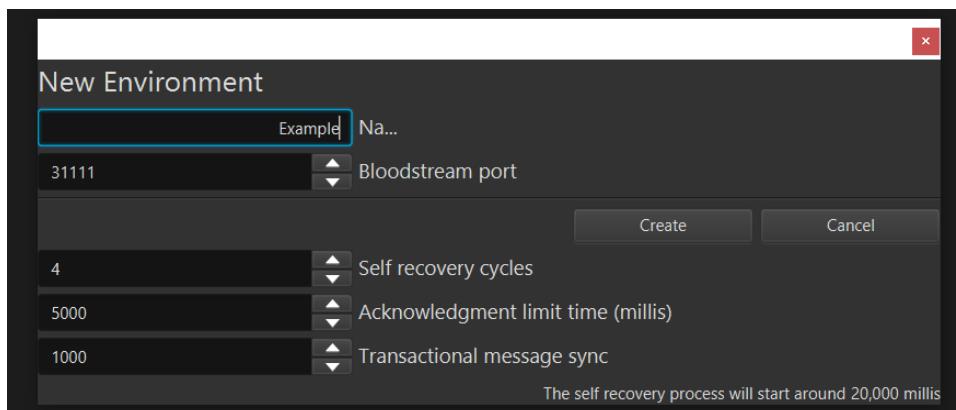


- Pasos para crear un ambiente:

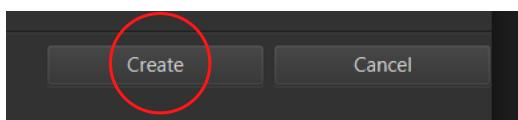
1. Dar click en el botón "Create".



2. Se desplegará una ventana como la siguiente para introducir la configuración del ambiente. Hay que introducir el nombre del ambiente y el número de puerto que usará el bloodstream del Aditi Net para comunicarse con las Aditi Cell. Cerciórate de que el puerto que introduzcas esté disponible. También se deben especificar los parámetros del ambiente (*Selfrecovery cycles, Acknowledgment limit time, Tx message synchronization*).



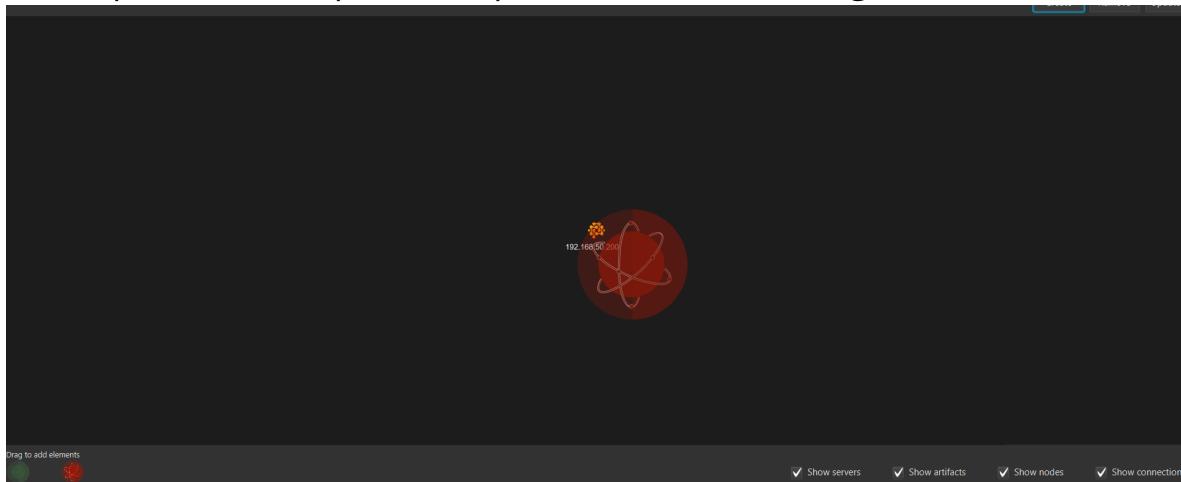
3. Dar click en el botón "Create" cuando todos los parámetros hayan sido correctamente introducidos.



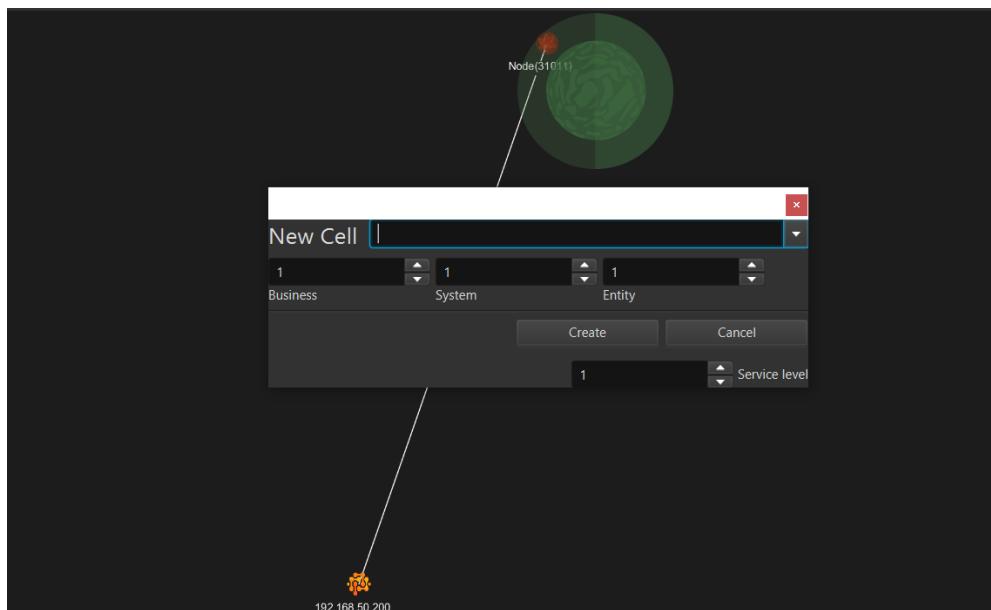
4. Cuando se haya registrado el ambiente nuevo en el *Server* del Aditi Manager, el panel de creación se habilitará para manipular los elementos del ambiente.

- Pasos para desplegar un ambiente:

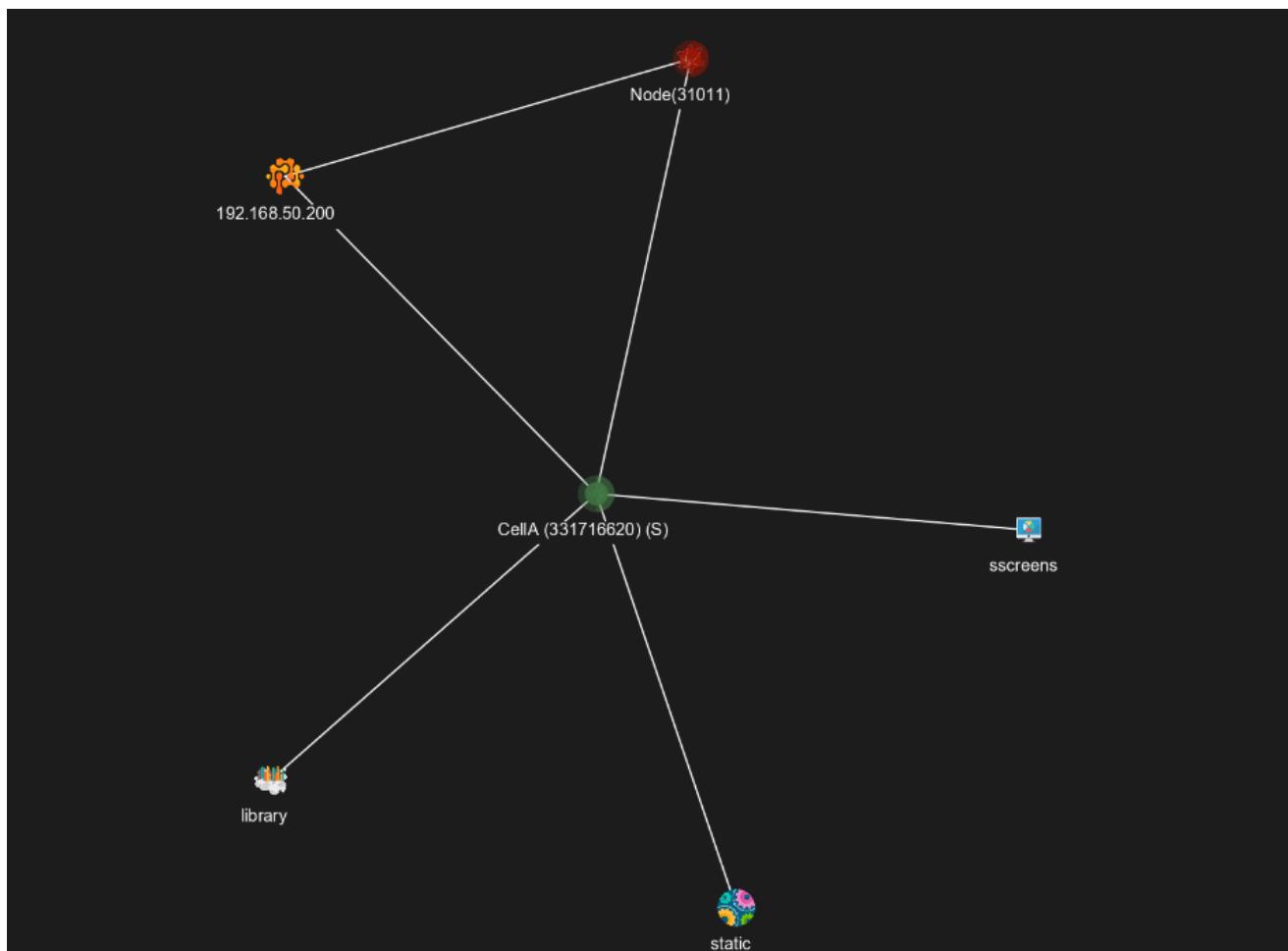
1. El primer requisito es tener al menos una instancia del Aditi Net funcionando. Para agregarla, arrastramos el ícono del Aditi Net ubicado en la esquina inferior izquierda del panel de creación y la soltamos sobre una de las computadoras disponibles que se muestran en el grafo.



2. Una vez disponible el bloodstream, se pueden instalar las Aditi Cell. Arrastramos el ícono del Aditi Cell hacia alguna instancia del Aditi Net. Se desplegará la siguiente ventana emergente para introducir la identidad de la nueva célula y su nivel de servicio.



3. Cuando la célula haya sido instalada e iniciada correctamente, se visualizará en el grafo de la siguiente manera, mostrando los directorios de almacenamiento de artefactos.



4. Al dar click sobre el icono de la célula del grafo, se mostrará una ventana con toda su información (identidad, huella, estadísticas, artefactos). Desde aquí se realizan las siguientes operaciones:

- Agregar los artefactos de la célula, arrastrando los que estén disponibles en la lista de la esquina inferior derecha hacia los paneles de cada uno de los directorios. Para guardar cambios dar click en el botón Update.
- Para eliminar artefactos, se debe seleccionar el artefacto y arrastrarlo fuera del panel del directorio. Para guardar cambios dar click en el botón Update.
- Obtener las bitácoras de la célula.
- Terminar la célula o reinstalarla.
- Consultar estadísticas.

CellA (331716620) (S) - 1.1.1

Start time: Fri Jul 17 20:40:41 CDT 2020
 Location: LAPTOP-8DKIVFOD - 192.168.50.200

SLAVE

Statistics

Messages	Received	Processed	Successes	Failures
11				
10				
9				
8				
7				
6				
5				
4				
3				
2				
1				
0				

Artifacts and services

 Library
 Static
 Screens
 Available artifacts

ATMScreen-2.0.0.jar
 ATMServices-2.0.0.jar
 BankLibrary-2.0.0.jar
 JFXScreens-1.0.0-jar-with-dependencies.jar
 JavaFXScreensView-1.0.0.jar
 ServerScreen-2.0.0.jar
 ServerServices-2.0.0.jar
 TVPScreen-2.0.0.jar
 TVPServices-2.0.0.jar
 UpdaterScreen-2.0.0.jar

↔

Update **Re-install** **Clean**



PARTE IV:

Demos Completas



Demo 1:

Calculadora Aditi

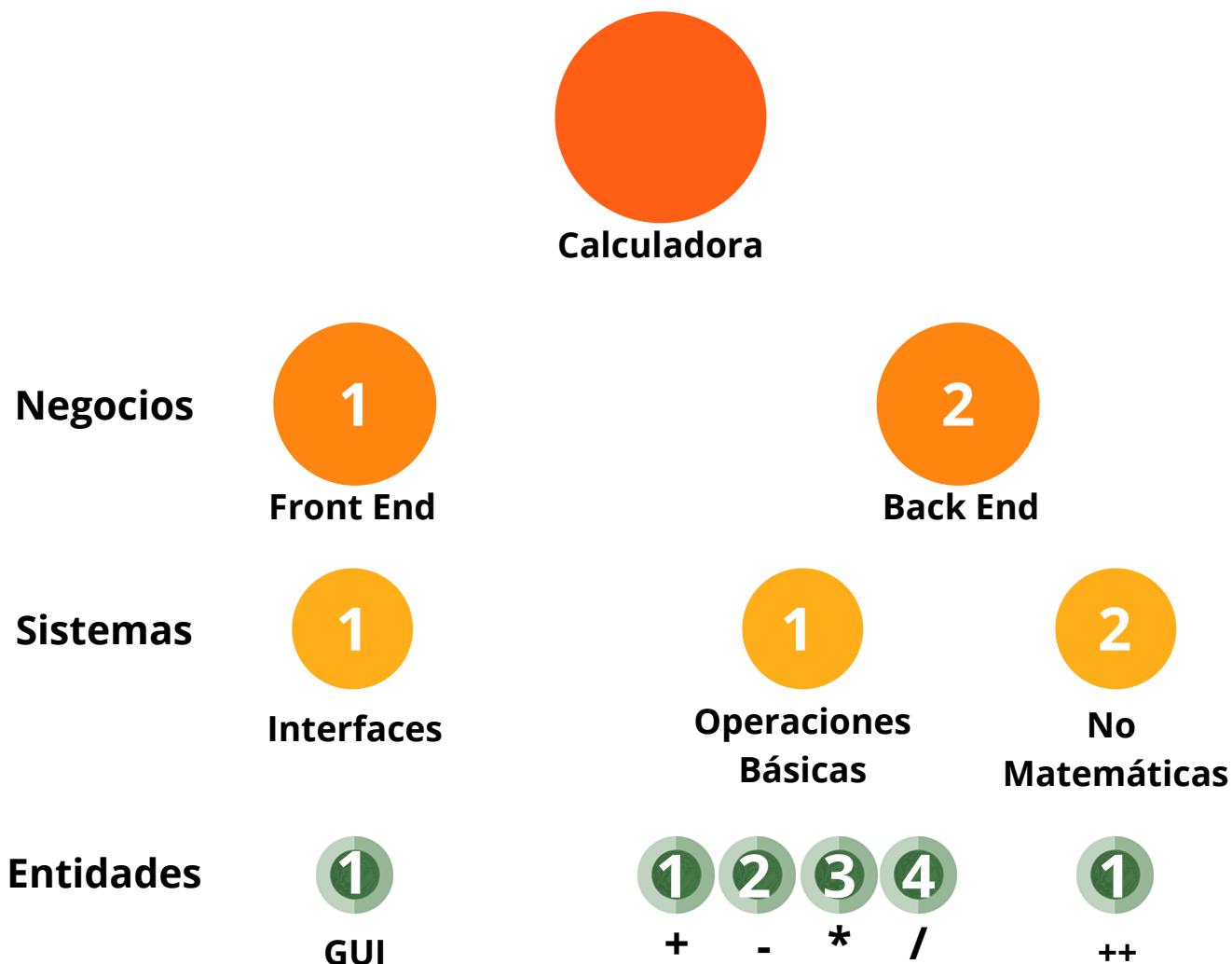
Dificultad 

Para integrar todos los temas presentados en esta documentación, construiremos desde cero una **calculadora** bajo el paradigma **Aditi**. Mostraremos cómo obtener las **células**, el **modelo lógico**, el código de los **servicios** y por último cómo implantarlo mediante el **Aditi Cellman**.

Fase 1: Análisis

- El objetivo de esta fase es determinar **qué células** conforman el sistema que estamos construyendo.
- El punto de partida es determinar las necesidades de negocio que serán resueltas por el sistema. En este caso, se requiere que la calculadora ofrezca los servicios de suma, resta, multiplicación, división y un contador, que puedan ser solicitados mediante una interfaz gráfica de usuario.
- Con esta información podemos dividir los servicios a ofrecer en dos **negocios** distintos:
 - El **negocio 1**, especializado en el front end.
 - El **negocio 2**, especializado en el back end.
- El siguiente paso es obtener los **sistemas** que integran los negocios:
 - Del **negocio 1**:
 - Un **sistema 1** encargado del manejo de las interfaces de usuario.
 - Del negocio 2:
 - Un **sistema 1** orientado a las operaciones matemáticas básicas.
 - Un **sistema 2** orientado a las operaciones no matemáticas.

- A continuación, se dividen los sistemas para obtener las **entidades**:
 - Del **negocio 1**:
 - Del **sistema 1**:
 - La **entidad 1**: La célula que ofrecerá el servicio de GUI.
 - Del **sistema 2**:
 - La **entidad 1**: La célula que ofrecerá el servicio de contador.
 - Del **negocio 2**:
 - Del **sistema 1**:
 - La **entidad 1**: La célula que ofrecerá el servicio de suma.
 - La **entidad 2**: La célula que ofrecerá el servicio de resta.
 - La **entidad 3**: La célula que ofrecerá el servicio de multiplicación.
 - La **entidad 4**: La célula que ofrecerá el servicio de división.



Fase 2: Análisis

- El objetivo de esta fase es determinar el **nivel de servicio** de las células obtenidas en la fase anterior. En otras palabras, hay que definir cuántas instancias de cada célula deben existir simultáneamente en el sistema. Si ocurriera alguna falla y el número se redujera, la plataforma Aditi lo detectaría e iniciaría el procedimiento de **autorrecuperación** para regenerar las instancias faltantes.
- El nivel de servicio de las células debe ir acorde a la criticidad de los servicios que ofrecen en el sistema. Para nuestra calculadora, supondremos que el servicio de suma es el más importante para el funcionamiento del sistema y que el GUI es el menos crítico de todos. Entonces, la escala del nivel de servicio irá del valor asignado a la suma a al valor que se asigne al contador.
- Vamos a listar todas nuestras células, indicando su criticidad y el nivel de servicio que asignamos a cada una:
 - Sumadora (2.1.1): Muy crítica, **nivel de servicio = 4**
 - Restadora (2.1.2): Crítica, **nivel de servicio = 3**
 - Multiplicadora (2.1.3): Crítica, **nivel de servicio = 3**
 - Divisora (2.1.4): Crítica, **nivel de servicio = 3**
 - Contadora (2.2.1): Moderada, **nivel de servicio = 2**
 - GUI (1.1.1): No crítica, **nivel de servicio = 1**
- Como resultado obtenemos que el **modelo lógico** de la calculadora Aditi está conformado por 4 células Sumadoras (2.1.1), 3 Restadoras (2.1.2), 3 Multiplicadoras (2.1.3), 3 Divisoras (2.1.4), 2 Contadoras (2.2.1) y 1 GUI (1.1.1).



La escala para definir la criticidad y asignar el nivel de servicio es a criterio del Arquitecto del sistema. No hay una regla universal para determinar los parámetros, pues depende principalmente de las necesidades de negocio y los recursos de cómputo disponibles.

Fase 3: Desarrollo

- El objetivo de esta fase es construir los **artefactos** que utilizarán las Aditi Cell para prestar y solicitar servicios.
- Ocuparemos todas lo presentado en la *Parte II* de este documento. Iremos célula por célula presentando el código de sus artefactos. Como recomendación, cualquier duda en la construcción del código, consulta la referencia antes de proseguir.

Célula 1.1.1

- Para esta célula construiremos dos artefactos:
 - La interfaz gráfica de usuario con el teclado de la calculadora, el display de los resultados y el botón "=" que enviará la solicitud de servicio para la operación solicitada con los valores introducidos. Para esta interfaz haremos uso del Add-on del contenedor de ventanas JFX.

```

1  @Screen(name = "Calculator", fxml = "/fxml/Calculator.fxml", start = Start.SHOW)
2  public class CalculatorScreen extends JFXScreenController {
3
4      /**
5       * Bitacora de la clase
6       */
7      private static final Logger LOGGER = Logger.getLogger(CalculatorScreen.class);
8
9      /* Label para mostrar la operación
10     */
11    @FXML
12    private TextField operationTextField;
13
14    /* Label para mostrar el resultado
15     */
16    @FXML
17    private TextField resultTextField;
18
19    /* Checkbox para activar el contador
20     */
21    @FXML
22    private CheckBox check_counter;
23
24    /* Label para mostrar el contador
25     */
26    @FXML
27    private TextField counterTextField;
28
29    private String num1 = null;
30    private String num2 = null;
31    private String operand = null;
32    private Integer operation = null;
33    private String name;
34    private String result = null;
35    private Boolean acceptnum1 = Boolean.TRUE;
36    private final String RESOURCE = "continueCounter";
37    private final String NUM1 = "num1";
38    private final String NUM2 = "num2";
39    private Double counter = null;
40
41    @Override
42    public void updateScreen(Map<? extends Object, Boolean> arg0, String arg1, String arg2) throws Exception {
43    }
44
45    @Override
46    public void initialize(URL url, ResourceBundle resourceBundle) {
47        Resources.getInstance().putResource(CalculatorScreen.class.getCanonicalName(), this);
48        check_counter.selectedProperty().addListener((observable, oldValue, newValue) -> {
49            if (oldValue && newValue) {
50                LOGGER.info("INICIAR CONTADOR");
51                Resources.getInstance().putResource(RESOURCE, Boolean.TRUE);
52                LOGGER.info("1");
53                counter = 0;
54                LOGGER.info("2");
55                try {
56                    LOGGER.info("3");
57                    requestCounter(counter);
58                    LOGGER.info("4");
59                } catch (Exception ex) {
60                    LOGGER.error("Requesting counter operation for (" + counter + ")");
61                }
62                LOGGER.info("5");
63            }
64        } else if (oldValue && !newValue) {
65            LOGGER.info("DETENER CONTADOR");
66            Resources.getInstance().putResource(RESOURCE, Boolean.FALSE);
67            counter = null;
68        }
69    });
70}

```

```

72 /**
73 * Métodos del teclado numérico
74 */
75 @FXML
76 void select0(ActionEvent event) {
77     pushNumber("0");
78     printOperation();
79 }
80
81 @FXML
82 void select1(ActionEvent event) {
83     pushNumber("1");
84     printOperation();
85 }
86
87 @FXML
88 void select2(ActionEvent event) {
89     pushNumber("2");
90     printOperation();
91 }
92
93 @FXML
94 void select3(ActionEvent event) {
95     pushNumber("3");
96     printOperation();
97 }
98
99 @FXML
100 void select4(ActionEvent event) {
101     pushNumber("4");
102     printOperation();
103 }
104
105 @FXML
106 void select5(ActionEvent event) {
107     pushNumber("5");
108     printOperation();
109 }
110
111 @FXML
112 void select6(ActionEvent event) {
113     pushNumber("6");
114     printOperation();
115 }
116
117 @FXML
118 void select7(ActionEvent event) {
119     pushNumber("7");
120     printOperation();
121 }
122
123 @FXML
124 void select8(ActionEvent event) {
125     pushNumber("8");
126     printOperation();
127 }
128
129 @FXML
130 void select9(ActionEvent event) {
131     pushNumber("9");
132     printOperation();
133 }
134
135 /**
136 * Métodos de las operaciones
137 */
138 @FXML
139 void addition(ActionEvent event) {
140     pushOperand("+");
141 }
142
143 @FXML
144 void subtraction(ActionEvent event) {
145     pushOperand("-");
146 }
147
148 @FXML
149 void multiplication(ActionEvent event) {
150     pushOperand("*");
151 }
152
153 @FXML
154 void division(ActionEvent event) {
155     pushOperand("/");
156 }
157
158 /**
159 * Métodos para la funcionalidad de la calculadora
160 */
161 @FXML
162 void equals(ActionEvent event) throws Exception {
163     if (operation != null) {
164         PackageMessage pm = new PackageMessage();
165         Boolean canSend = Boolean.FALSE;
166         String numbers = "";
167         switch (operation) {
168             case 121:
169                 if (num1 != null) {
170                     pm.putData(NUM1, Double.parseDouble(num1));
171                     canSend = Boolean.TRUE;
172                     numbers = num1;
173                 }
174                 break;
175             case 122:
176             case 123:
177             case 124:
178             case 125:
179                 if (num1 != null && num2 != null) {
180                     pm.putData(NUM1, Double.parseDouble(num1));
181                     pm.putData(NUM2, Double.parseDouble(num2));
182                     canSend = Boolean.TRUE;
183                     numbers = num1 + num2;
184                 }
185                 break;
186             default:
187                 LOGGER.error("Cannot request operation");
188                 break;
189         }
190     }
191 }
192

```

```

193 ~     if (canSend) {
194 ~         String seed = getSeed(operation, numbers);
195 ~         pm.putData("seed", seed);
196 ~         LOGGER.info("Operation (" + operation.shortValue() + ") Name (" + name + ") Seed (" + seed + ") PM (" + pm + ")");
197 ~         requestOperation(pm, seed);
198 ~     }
199 ~ } else {
200 ~     LOGGER.error("Cannot request operation");
201 ~ }
202 ~ }
203 ~
204 ~ @FXML
205 ~ void selectAc(ActionEvent event) {
206 ~ /**
207 ~ * Clear operation
208 ~ */
209 ~ operationTextField.textProperty().setValue("");
210 ~ num1 = null;
211 ~ num2 = null;
212 ~ operand = null;
213 ~ operation = null;
214 ~ name = null;
215 ~ /**
216 ~ * Clear result
217 ~ */
218 ~ resultTextField.textProperty().setValue("0");
219 ~ }
220 ~
221 ~ @FXML
222 ~ void select_point(ActionEvent event) {
223 ~     LOGGER.info("Select point");
224 ~ }
225 ~
226 ~ /**
227 ~ * Métodos del controlador
228 ~ */
229 ~ private void pushNumber(String number) {
230 ~     if (num1 == null) {
231 ~         num1 = number;
232 ~     } else if (operand == null) {
233 ~         num1 += number;
234 ~     } else if (acceptNum2) {
235 ~         if (num2 == null) {
236 ~             num2 = number;
237 ~         } else {
238 ~             num2 += number;
239 ~         }
240 ~     }
241 ~ }
242 ~
243 ~ private void pushOperand(String operand) {
244 ~     switch (operand) {
245 ~         case "+":
246 ~         case "-":
247 ~         case "*":
248 ~         case "/":
249 ~             if (!Objects.isNull(result) && Objects.isNull(num1)) {
250 ~                 num1 = result;
251 ~             }
252 ~             if (num1 != null) {
253 ~                 operation = getOperation(operand);
254 ~                 this.operand = operand;
255 ~             }
256 ~             printOperation();
257 ~             break;
258 ~         default:
259 ~             break;
260 ~     }
261 ~ }
262 ~
263 ~ private Integer getOperation(String operand) {
264 ~     switch (operand) {
265 ~         case "+":
266 ~             name = "ADD";
267 ~             acceptNum2 = Boolean.TRUE;
268 ~             return 12;
269 ~         case "-":
270 ~             name = "SUBS";
271 ~             acceptNum2 = Boolean.TRUE;
272 ~             return 13;
273 ~         case "*":
274 ~             name = "MULT";
275 ~             acceptNum2 = Boolean.TRUE;
276 ~             return 14;
277 ~         case "/":
278 ~             name = "DIV";
279 ~             acceptNum2 = Boolean.TRUE;
280 ~             return 15;
281 ~         default:
282 ~             return null;
283 ~     }
284 ~ }
285 ~
286 ~ private void printOperation() {
287 ~     String s = "";
288 ~     if (num1 != null) {
289 ~         s += num1;
290 ~     }
291 ~     if (operand != null) {
292 ~         s += " " + operand;
293 ~     }
294 ~     if (num2 != null) {
295 ~         s += " " + num2;
296 ~     }
297 ~     operationTextField.textProperty().setValue(s);
298 ~ }
299 ~
300 ~ /**
301 ~ * Métodos para solicitar operaciones
302 ~ */
303 ~
304 ~ private void requestOperation(PackageMessage packageMessage, String seed) throws Exception {
305 ~     sendMessage(name, seed, packageMessage);
306 ~ }
307 ~

```

```

388     private void requestCounter(Double number) throws Exception {
389
390         SecureSender<Transaction<Void>> send = new SecureSender<Transaction<Void>>() {
391             @Override
392             protected Void businessSender(Transaction transaction) throws Exception {
393
394                 Map<String, NetworkSocket> bloodStreamActiveNetworkSockets = BloodStream.getInstance().getBloodStreamActiveNetworkSockets();
395                 if (bloodStreamActiveNetworkSockets.isEmpty()) {
396                     return null;
397                 }
398
399                 String seed = getSeed(121, number + "");
400                 PackageMessage packageMessage = new PackageMessage();
401                 packageMessage.putData(NUM1, number);
402                 packageMessage.putData("seed", seed);
403                 LOGGER.info("Requesting counter for (" + number + ")");
404                 String selectedBloodstream = bloodStreamActiveNetworkSockets.keySet().iterator().next();
405                 MessageFactory.getInstance().sendTransactional(transaction, "COUNT", selectedBloodstream, seed, packageMessage);
406                 return null;
407             }
408
409             @Override
410             protected void lockTimeOutException(LockTimeOutException ltoe) {
411                 LOGGER.error("Lock requesting operation", ltoe);
412             }
413         };
414         send.run();
415     }
416
417     private String getSeed(Integer operation, String numbers) throws Exception {
418         return Utilities.getMD5(operation + "-" + numbers + "-" + System.nanoTime());
419     }
420
421     private void sendMessage(String name, String seed, PackageMessage packageMessage) throws Exception {
422         SecureSender<Transaction<Void>> send = new SecureSender<Transaction<Void>>() {
423             @Override
424             protected Void businessSender(Transaction transaction) throws Exception {
425
426                 Map<String, NetworkSocket> bloodStreamActiveNetworkSockets = BloodStream.getInstance().getBloodStreamActiveNetworkSockets();
427                 if (bloodStreamActiveNetworkSockets.isEmpty()) {
428                     return null;
429                 }
430                 String selectedBloodstream = bloodStreamActiveNetworkSockets.keySet().iterator().next();
431                 MessageFactory.getInstance().sendTransactional(transaction, name, selectedBloodstream, seed, packageMessage);
432                 return null;
433             }
434
435             @Override
436             protected void lockTimeOutException(LockTimeOutException ltoe) {
437                 LOGGER.error("Lock requesting operation", ltoe);
438             }
439         };
440         send.run();
441     }
442
443     /**
444      * OTROS
445      */
446     public void showResult(Double result, Short operation) throws Exception {
447         switch (operation) {
448             case (short) 121:
449                 processCounterResult(result);
450                 break;
451             default:
452                 LOGGER.info("Result (" + result + ") Operation (" + operation + ")");
453                 resultTextField.setTextProperty().setValue(NumberFormat.getNumberInstance(java.util.Locale.US).format(result));
454                 this.result = NumberFormat.getNumberInstance(java.util.Locale.US).format(result);
455                 operand = null;
456                 num1 = null;
457                 num2 = null;
458                 break;
459         }
460     }
461
462     private void processCounterResult(Double result) throws Exception {
463         LOGGER.info("Counter result (" + result + ")");
464         counterTextField.setTextProperty().setValue(NumberFormat.getNumberInstance(java.util.Locale.US).format(result));
465         if (!((Boolean) Resources.getInstance().getResource(RESOURCE))) {
466             return;
467         }
468         try {
469             Thread.sleep(SOL);
470             counter = result;
471             requestCounter(counter);
472         } catch (InterruptedException ex) {
473             LOGGER.error(ex);
474         }
475     }
476 }
477
478 }
479

```

- El servicio "PRINT" que recibirá como entrada los resultados de las operaciones y los mostrará en la interfaz de usuario.

```

1 √ /**
2  * Update GUI
3 *
4  * @author Gabriel Escobedo Ramírez
5  * @since January 2020
6 */
7 @ServiceInformation(serviceNumber = 111, protocol = Protocol.ALL, name = "PRINT")
8 public class UpdateGUI extends Service {
9
10 √ /**
11  * Bitacora de La clase
12  */
13 private static final Logger LOGGER = Logger.getLogger(UpdateGUI.class);
14 /**
15  * Result
16  */
17 @InputAttribute(key = "result")
18 Double result;
19 /**
20  * Operation solved
21  */
22 @InputAttribute(key = "operation")
23 Short operation;
24
25 @Override
26 public void run() throws Exception {
27     LOGGER.info("Result>> " + result);
28     CalculatorScreen controller = (CalculatorScreen) Resources.getInstance().getResource(CalculatorScreen.class.getCanonicalName());
29     Platform.runLater(() -> {
30         try {
31             controller.showResult(result, operation);
32         } catch (Exception ex) {
33             LOGGER.error("Updating screen", ex);
34         }
35     });
36 }
37
38 @Override
39 public void exception(Throwable throwable) throws Exception {
40     LOGGER.error("In the updaters", throwable);
41 }
42
43 }
```

Célula 2.1.1

- Para esta célula construiremos un artefactos:
 - El servicio "ADD" que recibirá como entrada dos números, los sumará y enviará el resultado al servicio "PRINT".

```

25 v /**
26  * Add two numbers
27 *
28 * @author Gabriel Escobedo Ramirez
29 * @since January 2020
30 */
31 @ServiceInformation(serviceNumber = 122, protocol = Protocol.ALL, name = "ADD")
32 v public class Addition extends Service {
33
34 v     /**
35      * Bitacora de La clase
36      */
37     private static final Logger LOGGER = Logger.getLogger(Addition.class);
38 v     /**
39      * First number
40      */
41     @InputAttribute(key = "num1")
42     Double num1;
43 v     /**
44      * Second number
45      */
46     @InputAttribute(key = "num2")
47     Double num2;
48 v     /**
49      * Seed
50      */
51     @InputAttribute(key = "seed")
52     String seed;
53
54     @Override
55     public void run() throws Exception {
56         /**
57          * Get the result
58          */
59         Double result = num1 + num2;
60         LOGGER.info("Addition>> " + num1 + " + " + num2 + " = " + result);
61         /**
62          * Send
63          */
64         PackageMessage pm = new PackageMessage();
65         pm.putData("result", result);
66         pm.putData("operation", (short) 122);
67         sendMessage(pm, Utilities.getMD5(seed));
68     }
69
70     @Override
71     public void exception(Throwable throwable) throws Exception {
72         LOGGER.error("In the addition", throwable);
73     }
74
75     private void sendMessage(PackageMessage packageMessage, String seed) throws Exception {
76         SecureSenderTransaction<Void> send = new SecureSenderTransaction<Void>() {
77             @Override
78             protected Void businessSender(Transaction arg0) throws Exception {
79                 Map<String, NetworkSocket> bloodStreamActiveNetworkSockets = BloodStream.getInstance().getBloodStreamActiveNetworkSockets();
80                 if (bloodStreamActiveNetworkSockets.isEmpty()) {
81                     return null;
82                 }
83                 String selectedBloodstream = bloodStreamActiveNetworkSockets.keySet().iterator().next();
84                 MessageFactory.getInstance().sendNonTransactional(arg0, "PRINT", requesterFootprint, selectedBloodstream, seed, packageMessage);
85                 return null;
86             }
87
88             @Override
89             protected void lockTimeOutException(LockTimeOutException ltoe) {
90                 LOGGER.error("UPS", ltoe);
91             }
92         };
93         send.run();
94     }
95 }
```

Célula 2.1.2

- Para esta célula construiremos un artefactos:
 - El servicio "SUBS" que recibirá como entrada dos números, los restará y enviará el resultado al servicio "PRINT".

```

1 v /**
2  * Subtract two numbers
3 *
4 * @author Gabriel Escobedo Ramirez
5 * @since January 2020
6 */
7 @ServiceInformation(serviceNumber = 123, protocol = Protocol.ALL, name = "SUBS")
8 v public class Subtraction extends Service {
9
10 v     /**
11      * Bitacora de La clase
12      */
13     private static final Logger LOGGER = Logger.getLogger(Subtraction.class);
14 v     /**
15      * First number
16      */
17     @InputAttribute(key = "num1")
18     Double num1;
19 v     /**
20      * Second number
21      */
22     @InputAttribute(key = "num2")
23     Double num2;
24 v     /**
25      * Seed
26      */
27     @InputAttribute(key = "seed")
28     String seed;
29
30     @Override
31     public void run() throws Exception {
32         /**
33          * Get the result
34          */
35         Double result = num1 - num2;
36         LOGGER.info("Subtraction> " + num1 + " - " + num2 + " = " + result);
37         /**
38          * Send
39          */
40         PackageMessage pm = new PackageMessage();
41         pm.putData("result", result);
42         pm.putData("operation", (short) 123);
43         sendMessage(pm, Utilities.getMD5(seed));
44     }
45
46     @Override
47     public void exception(Throwable throwable) throws Exception {
48         LOGGER.error("In the subtraction", throwable);
49     }
50
51     private void sendMessage(PackageMessage packageMessage, String seed) throws Exception {
52         SecureSenderTransaction<Void> send = new SecureSenderTransaction<Void>() {
53             @Override
54             protected Void businessSender(Transaction arg0) throws Exception {
55                 Map<String, NetworkSocket> bloodStreamActiveNetworkSockets = BloodStream.getInstance().getBloodStreamActiveNetworkSockets();
56                 if (bloodStreamActiveNetworkSockets.isEmpty()) {
57                     return null;
58                 }
59                 String selectedBloodstream = bloodStreamActiveNetworkSockets.keySet().iterator().next();
60                 MessageFactory.getInstance().sendNonTransactional(arg0, "PRINT", requesterFootprint, selectedBloodstream, seed, packageMessage);
61                 return null;
62             }
63
64             @Override
65             protected void lockTimeOutException(LockTimeOutException ltoe) {
66                 LOGGER.error("UPS", ltoe);
67             }
68         };
69         send.run();
70     }
71 }
```

Célula 2.1.3

- Para esta célula construiremos un artefactos:
 - El servicio "MULT" que recibirá como entrada dos números, los multiplicará y enviará el resultado al servicio "PRINT".

```

1 v /**
2   * Multiply two numbers
3   *
4   * @author Gabriel Escobedo Ramirez
5   * @since January 2020
6   */
7   @ServiceInformation(serviceNumber = 124, protocol = Protocol.ALL, name = "MULT")
8 v public class Multiplication extends Service {
9
10 v    /**
11   * Bitacora de La clase
12   */
13   private static final Logger LOGGER = Logger.getLogger(Multiplication.class);
14 v   /**
15   * First number
16   */
17   @InputAttribute(key = "num1")
18   Double num1;
19 v   /**
20   * Second number
21   */
22   @InputAttribute(key = "num2")
23   Double num2;
24 v   /**
25   * Seed
26   */
27   @InputAttribute(key = "seed")
28   String seed;
29
30   @Override
31   public void run() throws Exception {
32     /**
33      * Get the result
34      */
35     Double result = num1 * num2;
36     LOGGER.info("Multiplication> " + num1 + " * " + num2 + " = " + result);
37     /**
38      * Send
39      */
40     PackageMessage pm = new PackageMessage();
41     pm.putData("result", result);
42     pm.putData("operation", (short) 124);
43     sendMessage(pm, Utilities.getMD5(seed));
44   }
45
46   @Override
47   public void exception(Throwable throwable) throws Exception {
48     LOGGER.error("In the multiplication", throwable);
49   }
50
51   private void sendMessage(PackageMessage packageMessage, String seed) throws Exception {
52     SecureSenderTransaction<Void> send = new SecureSenderTransaction<Void>() {
53       @Override
54       protected Void businessSender(Transaction arg0) throws Exception {
55         Map<String, NetworkSocket> bloodStreamActiveNetworkSockets = BloodStream.getInstance().getBloodStreamActiveNetworkSockets();
56         if (bloodStreamActiveNetworkSockets.isEmpty()) {
57           return null;
58         }
59         String selectedBloodstream = bloodStreamActiveNetworkSockets.keySet().iterator().next();
60         MessageFactory.getInstance().sendNonTransactional(arg0, "PRINT", requesterFootprint, selectedBloodstream, seed, packageMessage);
61         return null;
62       }
63
64       @Override
65       protected void lockTimeOutException(LockTimeOutException ltoe) {
66         LOGGER.error("UPS", ltoe);
67       }
68     };
69     send.run();
70   }
71 }
```

Célula 2.1.4

- Para esta célula construiremos un artefactos:
 - El servicio "DIV" que recibirá como entrada dos números, los multiplicará y enviará el resultado al servicio "PRINT".

```

1 v /**
2   * Divide two numbers
3   *
4   * @author Gabriel Escobedo Ramirez
5   * @since January 2020
6   */
7   @ServiceInformation(serviceNumber = 125, protocol = Protocol.ALL, name = "DIV")
8   public class Division extends Service {
9
10  /**
11   * Bitacora de La clase
12   */
13  private static final Logger LOGGER = Logger.getLogger(Division.class);
14  /**
15   * First number
16   */
17  @InputAttribute(key = "num1")
18  Double num1;
19  /**
20   * Second number
21   */
22  @InputAttribute(key = "num2")
23  Double num2;
24  /**
25   * Seed
26   */
27  @InputAttribute(key = "seed")
28  String seed;
29
30  @Override
31  public void run() throws Exception {
32  /**
33   * Get the result
34   */
35  Double result = num1 / num2;
36  LOGGER.info("Division>> " + num1 + " / " + num2 + " = " + result);
37  /**
38   * Send
39   */
40  PackageMessage pm = new PackageMessage();
41  pm.putData("result", result);
42  pm.putData("operation", (short) 125);
43  sendMessage(pm, Utilities.getMD5(seed));
44  }
45
46  @Override
47  public void exception(Throwable throwable) throws Exception {
48    LOGGER.error("In the division", throwable);
49  }
50
51  private void sendMessage(PackageMessage packageMessage, String seed) throws Exception {
52    SecureSenderTransaction<Void> send = new SecureSenderTransaction<Void>() {
53      @Override
54      protected Void businessSender(Transaction arg0) throws Exception {
55        Map<String, NetworkSocket> bloodStreamActiveNetworkSockets = BloodStream.getInstance().getBloodStreamActiveNetworkSockets();
56        if (bloodStreamActiveNetworkSockets.isEmpty()) {
57          return null;
58        }
59        String selectedBloodstream = bloodStreamActiveNetworkSockets.keySet().iterator().next();
60        MessageFactory.getInstance().sendNonTransactional(arg0, "PRINT", requesterFootprint, selectedBloodstream, seed, packageMessage);
61        return null;
62      }
63
64      @Override
65      protected void lockTimeOutException(LockTimeOutException ltoe) {
66        LOGGER.error("UPS", ltoe);
67      }
68    };
69    send.run();
70  }
71 }
```

Célula 2.2.1

- Para esta célula construiremos un artefactos:
 - El servicio "COUNT" que recibirá como entrada un número, lo aumentará en 1 y enviará el resultado al servicio "PRINT".

```

1 v /**
2  * Counter
3  *
4  * @author Gabriel Escobedo Ramírez
5  * @since January 2020
6  */
7 @ServiceInformation(serviceNumber = 121, protocol = Protocol.ALL, name = "COUNT")
8 public class Counter extends Service {
9
10 v /**
11  * Bitacora de la clase
12  */
13 private static final Logger LOGGER = Logger.getLogger(Counter.class);
14 v /**
15  * First number
16  */
17 @InputAttribute(key = "num1")
18 Double num1;
19 v /**
20  * Seed
21  */
22 @InputAttribute(key = "seed")
23 String seed;
24
25 @Override
26 v public void run() throws Exception {
27  /**
28   * Get the result
29   */
30  Double result = num1 + 1D;
31  LOGGER.info("Counter>> " + num1 + " ++ " + " = " + result);
32 v /**
33   * Send
34   */
35  PackageMessage pm = new PackageMessage();
36  pm.putData("result", result);
37  pm.putData("operation", (short) 121);
38  sendMessage(pm, Utilities.getMD5(seed));
39 }
40
41 @Override
42 v public void exception(Throwable throwable) throws Exception {
43  LOGGER.error("In counter", throwable);
44 }
45
46 v private void sendMessage(PackageMessage packageMessage, String seed) throws Exception {
47  SecureSenderTransaction<Void> send = new SecureSenderTransaction<Void>() {
48    /**
49     protected Void businessSender(Transaction arg0) throws Exception {
50       Map<String, NetworkSocket> bloodStreamActiveNetworkSockets = BloodStream.getInstance().getBloodStreamActiveNetworkSockets();
51 v        if (bloodStreamActiveNetworkSockets.isEmpty()) {
52          return null;
53        }
54        String selectedBloodstream = bloodStreamActiveNetworkSockets.keySet().iterator().next();
55        MessageFactory.getInstance().sendNonTransactional(arg0, "PRINT", requesterFootprint, selectedBloodstream, seed, packageMessage);
56        return null;
57      }
58
59      /**
60       protected void lockTimeOutException(LockTimeOutException ltoe) {
61         LOGGER.error("UPS", ltoe);
62       }
63     };
64     send.run();
65   }
66 }
```

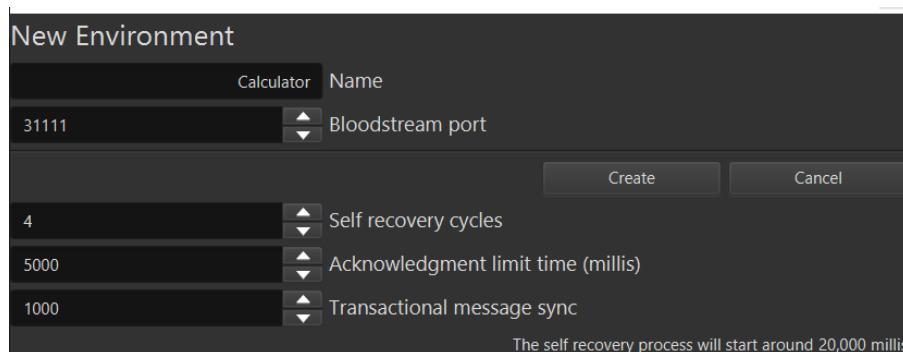
Fase 4: Implantación

- En esta fase llevaremos el modelo lógico a producción. Usando el Aditi Manager, desplegaremos las células diseñadas y sus respectivos artefactos.
- Desde la ventana de *Artefactos* añadimos los archivos jar generados en la fase 3 al Servidor de Aditi Manager.

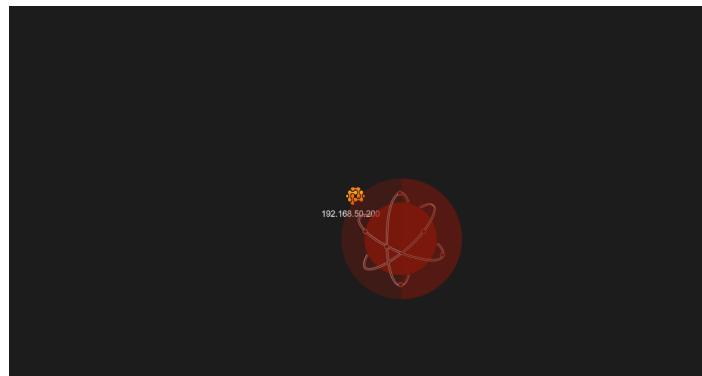
File Name	Group	Name	Version	Since	Packaging	Size (bytes)
Addition-1.0.0.jar	org.mithikel.aditi.demo	Addition	1.0.0	Fri Jul 24 15:54:06 CDT 2020	jar	5,1
CalculatorScreen-1.0.0.jar	org.mithikel.aditi.demo	CalculatorScreen	1.0.0	Fri Jul 24 15:54:19 CDT 2020	jar	24,2
Counter-1.0.0.jar	org.mithikel.aditi.demo	Counter	1.0.0	Fri Jul 24 15:54:34 CDT 2020	jar	5,1
Division-1.0.0.jar	org.mithikel.aditi.demo	Division	1.0.0	Fri Jul 24 15:54:41 CDT 2020	jar	5,1
Multiplication-1.0.0.jar	org.mithikel.aditi.demo	Multiplication	1.0.0	Fri Jul 24 15:54:47 CDT 2020	jar	5,2
Subtraction-1.0.0.jar	org.mithikel.aditi.demo	Subtraction	1.0.0	Fri Jul 24 15:54:58 CDT 2020	jar	5,2

Dependencies Files + - Download

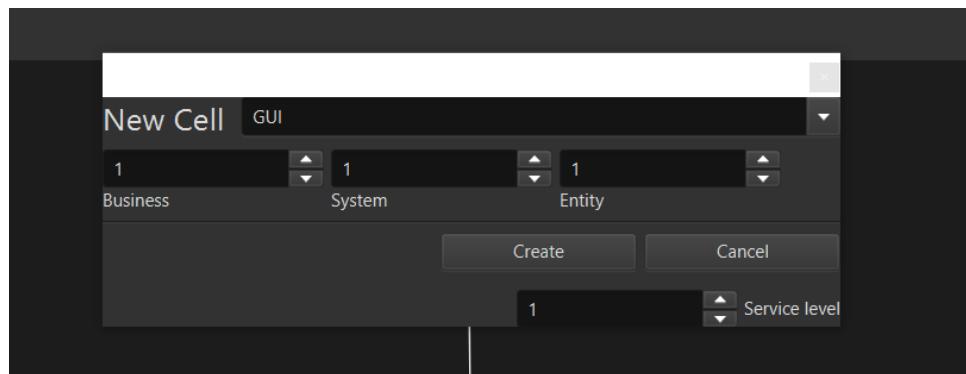
- En la ventana de *Environments* creamos el ambiente *Calculator*.



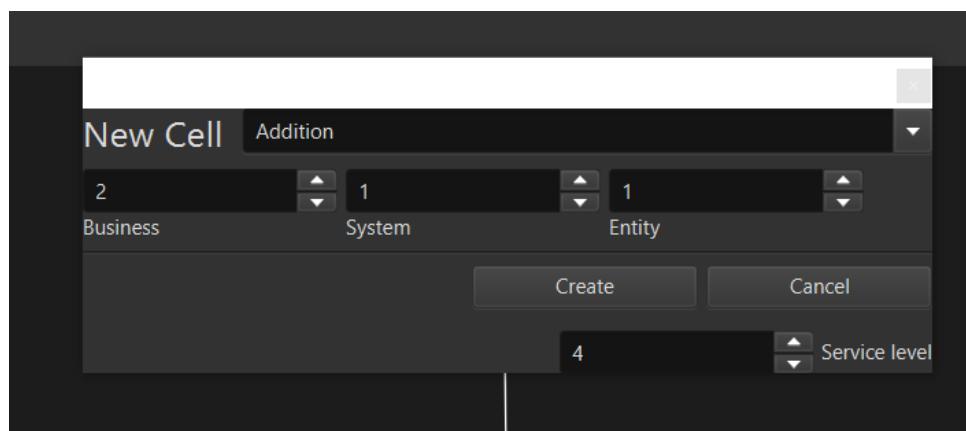
- Instalamos un nodo de Aditi Net en alguna de las computadoras disponibles. En este ejemplo únicamente se cuenta con una computadora; en caso de tener más equipos al alcance, es recomendable instalar un nodo en cada uno.



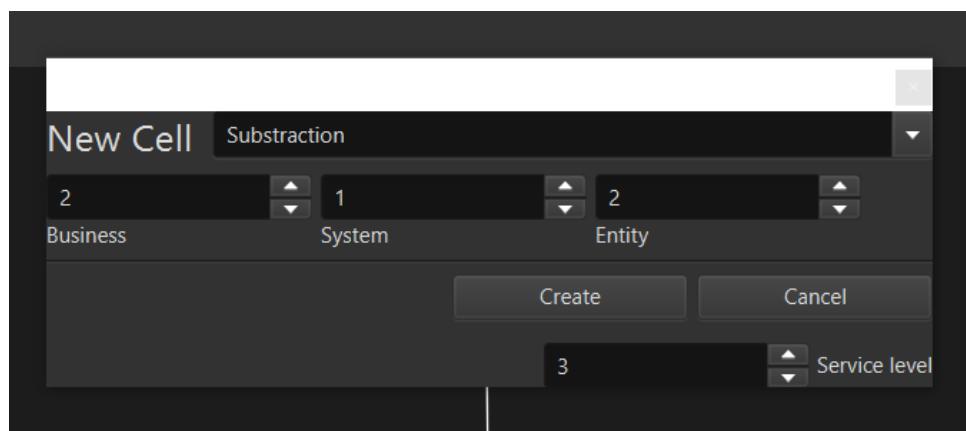
- Una vez que se muestra el nodo instalado en el grafo, instalamos la célula 1.1.1 (GUI) con nivel de servicio 1.



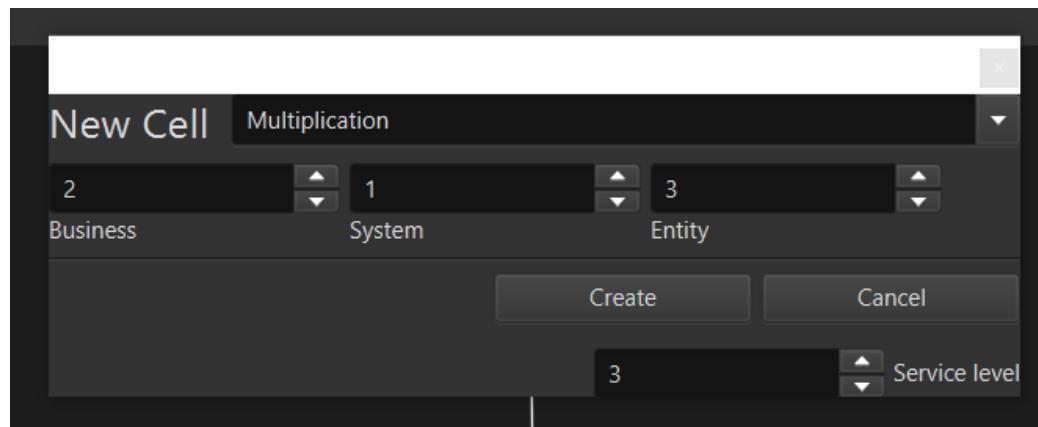
- Instalamos la célula 2.1.1 (Suma) con nivel de servicio 4.



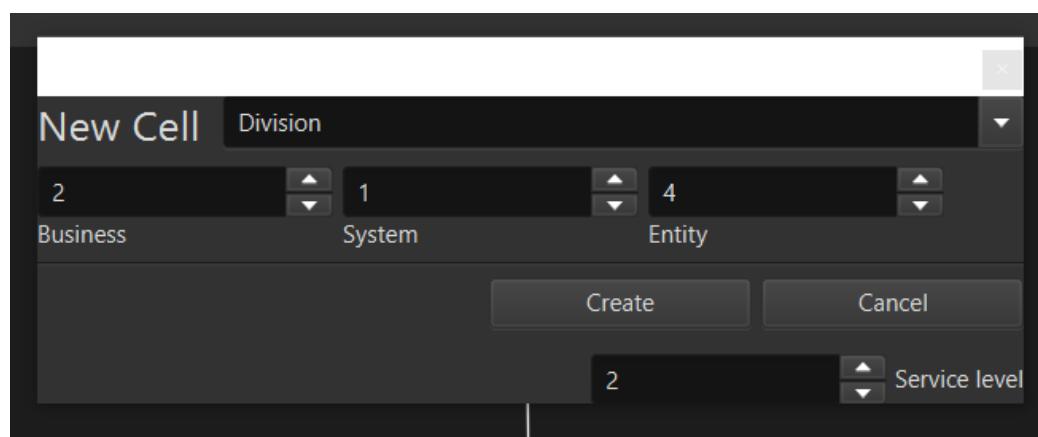
- Instalamos la célula 2.1.2 (Resta) con nivel de servicio 3.



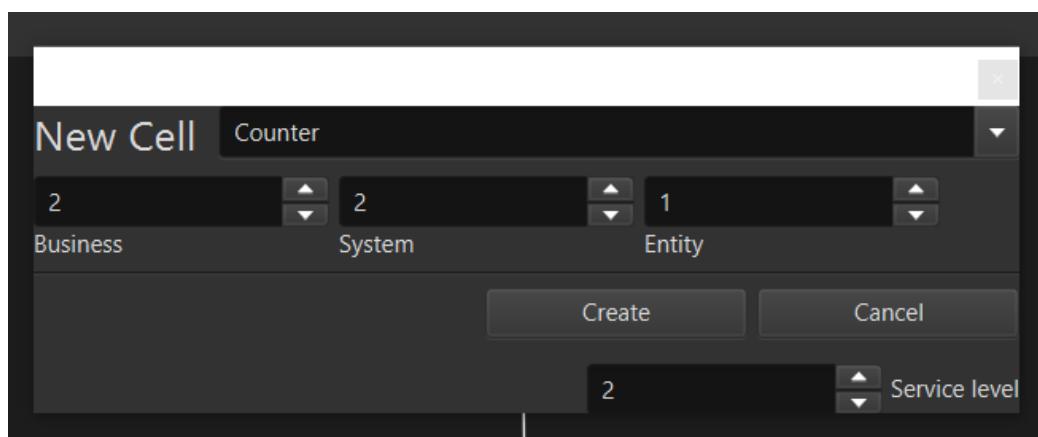
- Instalamos la célula 2.1.3 (Multiplicación) con nivel de servicio 3.



- Instalamos la célula 2.1.4 (División) con nivel de servicio 3.



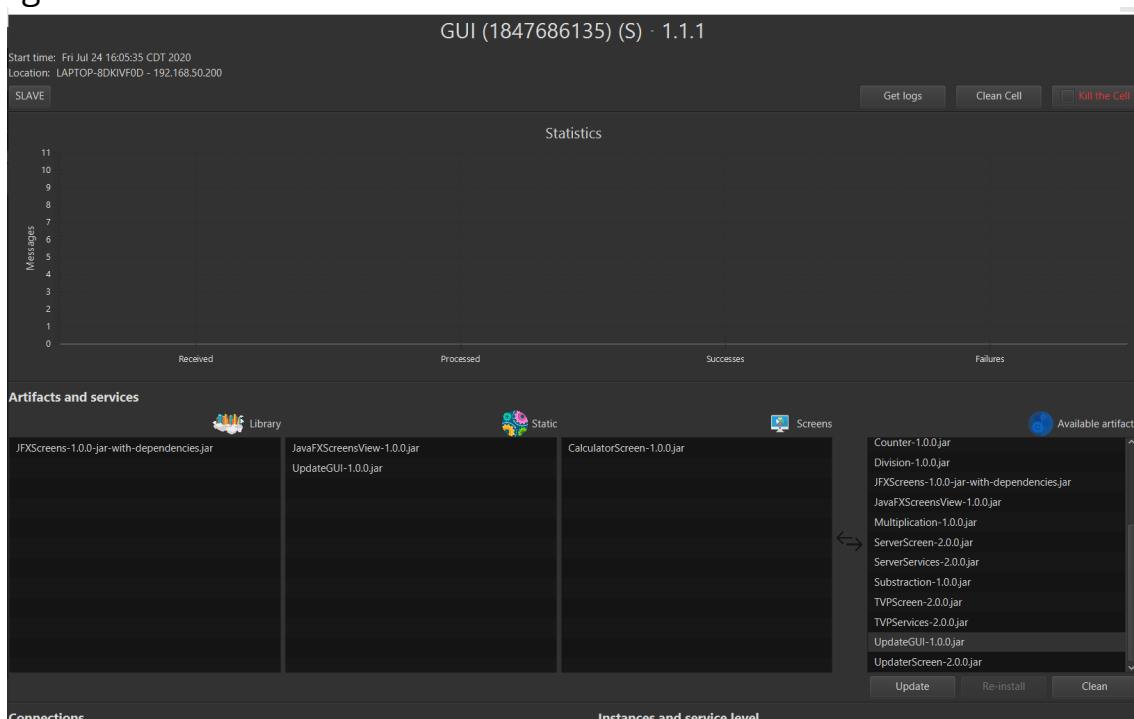
- Instalamos la célula 2.2.1 (Contador) con nivel de servicio 2.



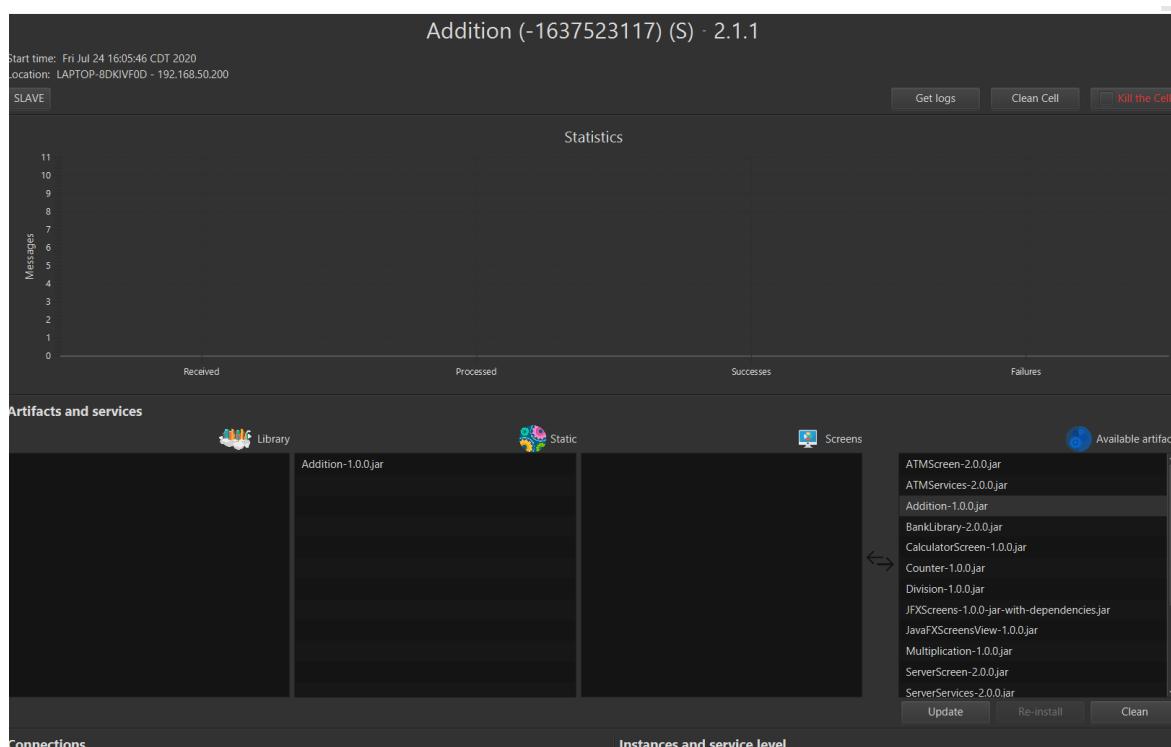
- Para este punto, el grafo de la ventana *Environments* luce así:



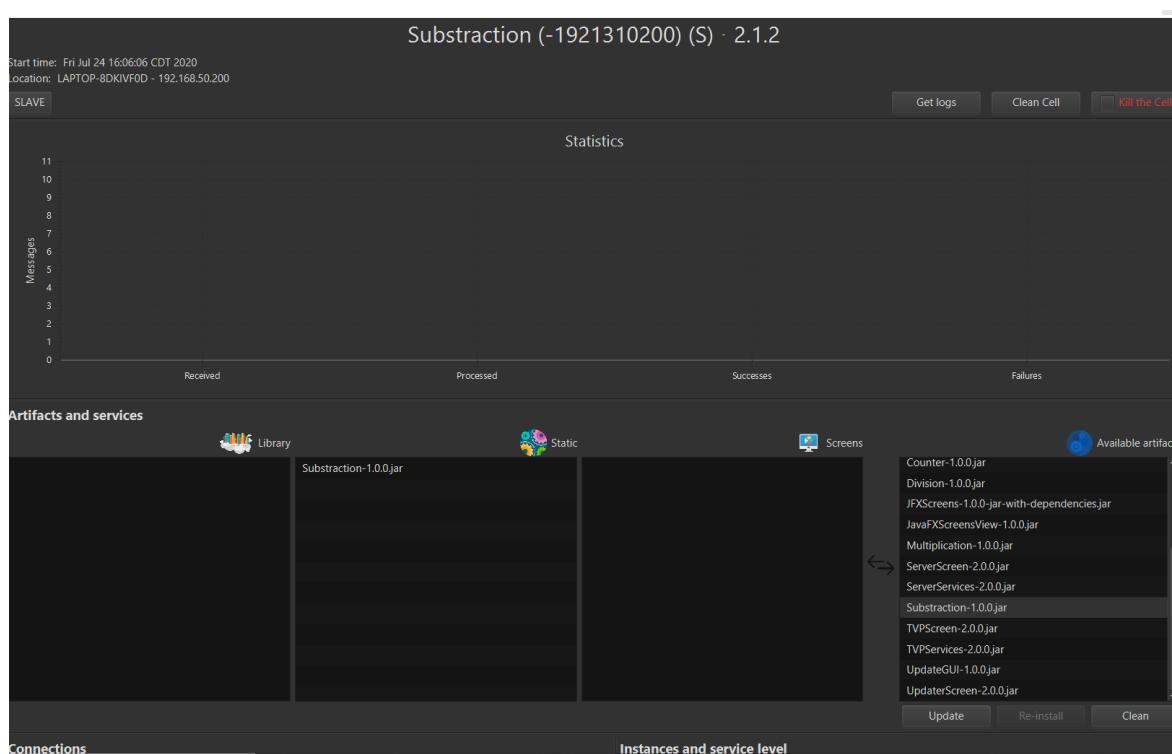
- Con doble click sobre la célula GUI abrimos la ventana de configuración y agregamos los artefactos.



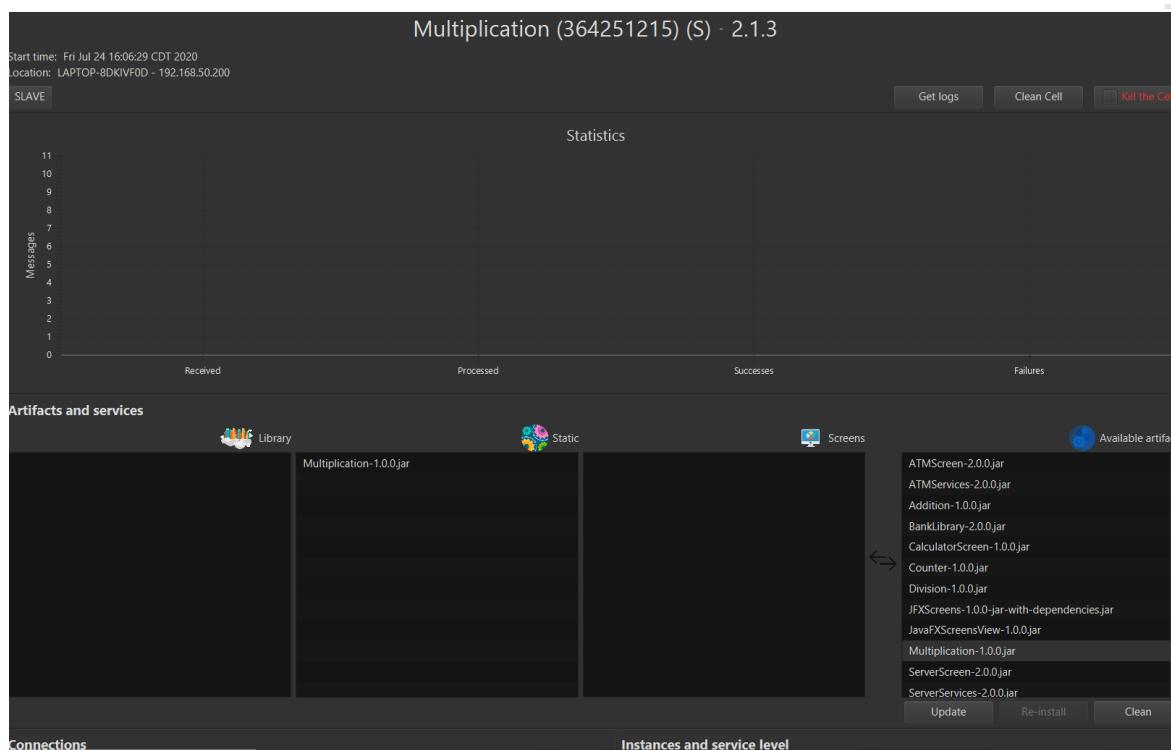
- Configuramos la célula Addition con el artefacto de su servicio.



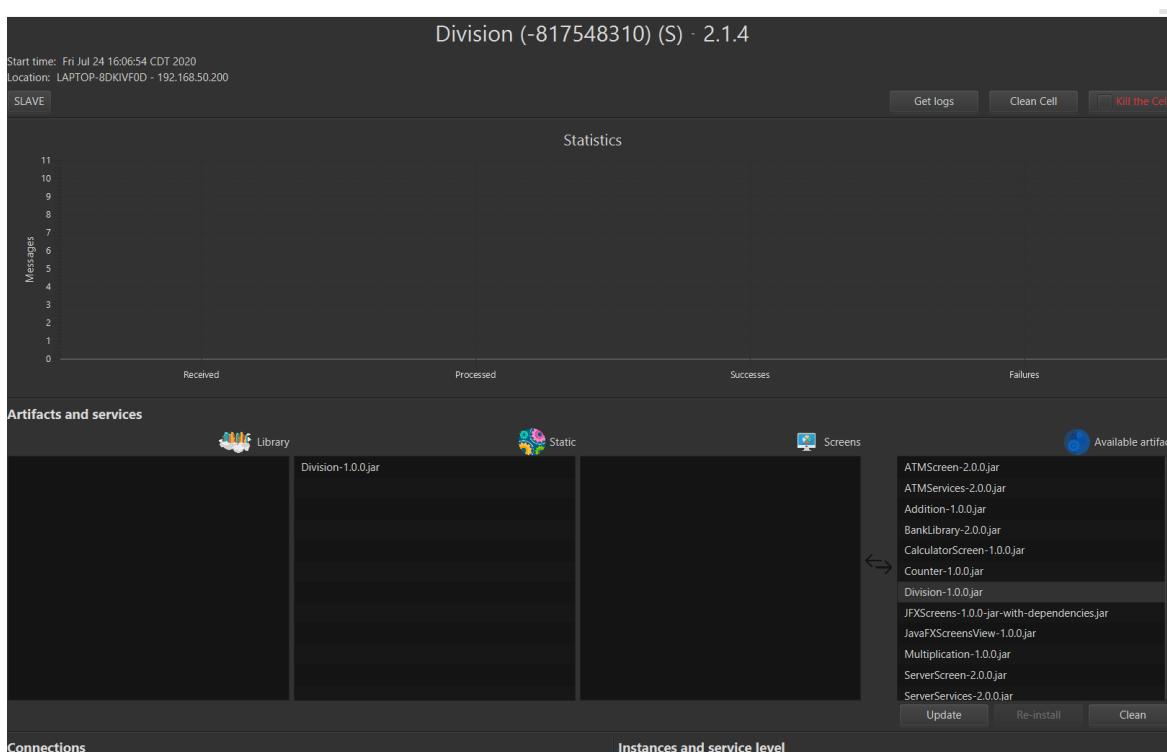
- Configuramos la célula Subtraction con el artefacto de su servicio.



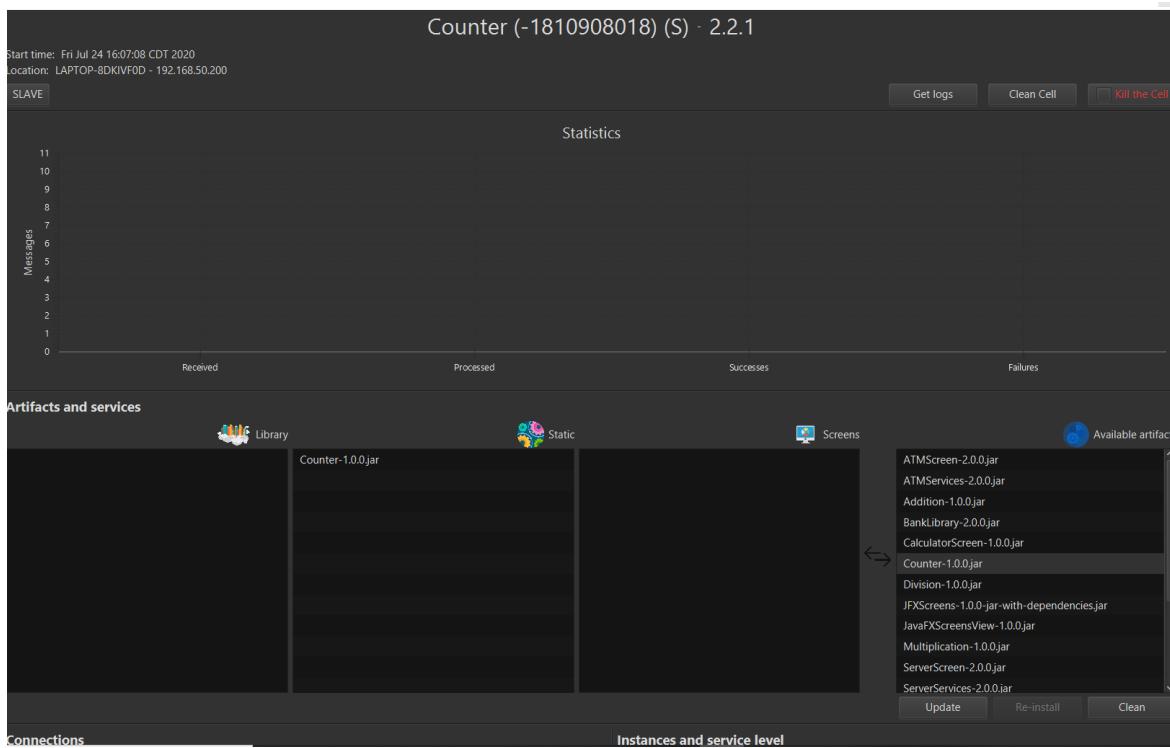
- Configuramos la célula Multiplication con el artefacto de su servicio.



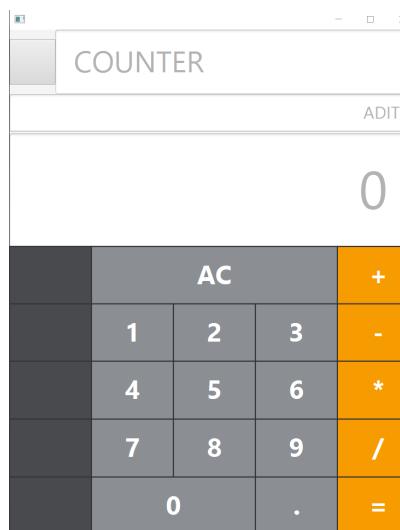
- Configuramos la célula Division con el artefacto de su servicio.



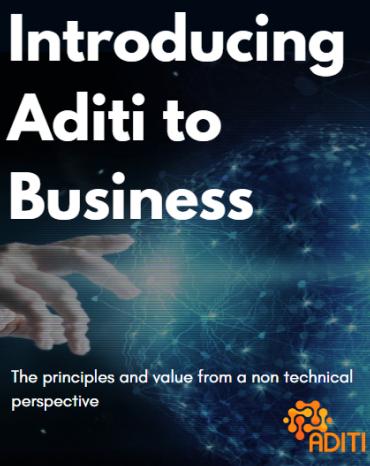
- Configuramos la célula Counter con el artefacto de su servicio.



- Una vez terminada la configuración de todas las Aditi Cell, podemos utilizar la GUI para solicitar las operaciones. Nótese que solo instalamos una sola célula de cada tipo, aunque todas las que ofrecen el servicio de operaciones tienen un nivel de servicio igual o mayor a 2. En ese estado, cuando sea solicitada determinada operación, la célula se clonará sucesivamente hasta lograr el nivel de servicio.



More resources to understand the value of Aditi



available at
mithikel.com