User Guide

Carlos Magno Abreu

Capítulo 1 : Introdução

O processador de Workflows Sagitarii

O Sagitarii é um Sistema de Workflows para Ciência de Dados, desenvolvido pela Escola de Informática e Computação (EIC) do CEFET-RJ.

Principais características

Ambiente WEB	O Sagitarii foi desenvolvido em Java, sendo um servidor web que provê acesso ao usuário através de um navegador de internet.
Fácil de usar	O Sagitarii foi concebido de forma que o usuário não precise conhecer sobre programação ou banco de dados. A criação e execução de um workflow é feita de forma intuitiva e acompanhada por assistentes práticos e fáceis de usar. A tela de visualização de dados permite acompanhar a produção dos dados em tempo real diretamente pelo sistema. A criação de consultas personalizadas oferece ao usuário a possibilidade de executar consultas SQL complexas. Estas consultas são criadas por um administrador e disponibilizadas ao usuário ao alcance de um clique, o que permite que usuários que não conhecem SQL possam usar o sistema com facilidade, se preocupando apenas com a solução de seu problema, dentro da sua área de conhecimento.
Versátil	A criação e execução de um workflow não necessita modificações no Sagitarii, sendo o sistema completamente desacoplado do trabalho do usuário. Isso o torna versátil o suficiente para ser aplicado nas mais diversas soluções. O sistema é altamente configurável, sendo fácil ajustá-lo à diversas condições de execução.
Inteligente	O Sagitarii conta com uma série de algoritmos de análise para avaliar o desempenho das atividades de um workflow, permitindo uma melhor otimização no processo de execução. O sistema também possui uma rede flexível, permitindo que os nós de processamento entrem e saiam da rede sem qualquer configuração adicional ou interrupção do trabalho.
Eficiente	O Sagitarii conta com um sistema de escalonamento que o permite executar vários workflows ao mesmo tempo. Seus nós de processamento podem executar Phytom, R e executar programas em Java para encapsular qualquer outro programa que seja necessário, tornando-o capaz de executar qualquer tipo de trabalho.
Proveniência de dados	No Sagitarii, os dados produzidos pelas atividades são rastreáveis, o que possibilita acompanhar a qualquer momento, toda a sequencia de produção e consumo de uma determinada informação.
Transparente	Toda a execução de um workflow pode ser monitorada. Cada arquivo transferido pelos nós de processamento, o tempo de cada tarefa, o consumo de CPU e memória dos nós de execução podem ser acompanhados por gráficos e barras de progresso. O usuário pode também acompanhar a saída do console de cada tarefa que está sendo executada pelos nós de processamento diretamente na interface do Sagitarii, o que permite uma proximidade e transparência sem precedentes. Ao administrador o sistema oferece um conjunto de logs configuráveis com altíssimo nível de detalhamento.
Livre	O Sagitarii é livre. Seu código pode ser modificado a fim de ser aprimorado pela comunidade ou para atender à necessidades específicas.
Instalação simplificada	Instale o banco de dados, faça o deploy do arquivo no servidor e pronto! O Sagitarii não requer configurações complexas em infindáveis arquivos.

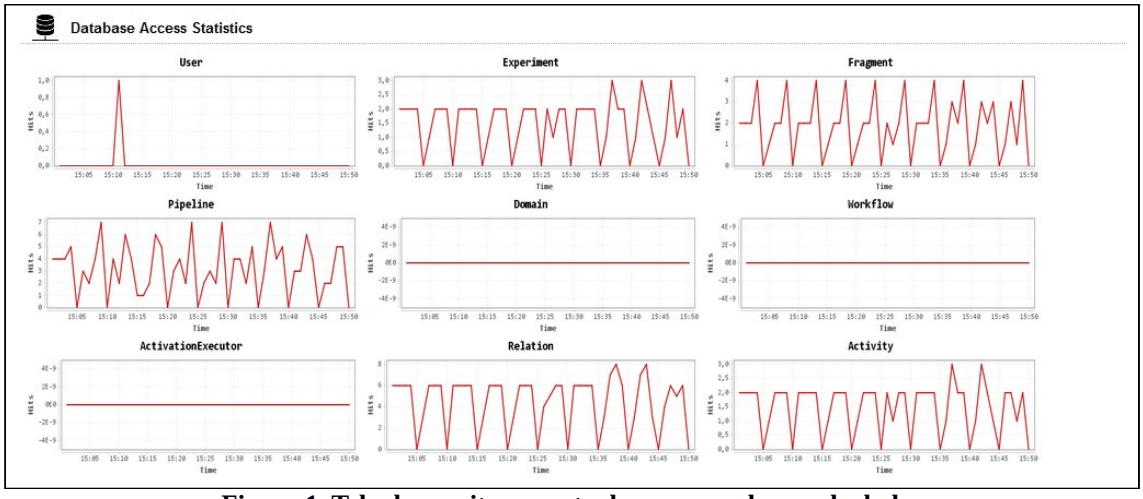


Figura 1. Tela de monitoramento do acesso ao banco de dados

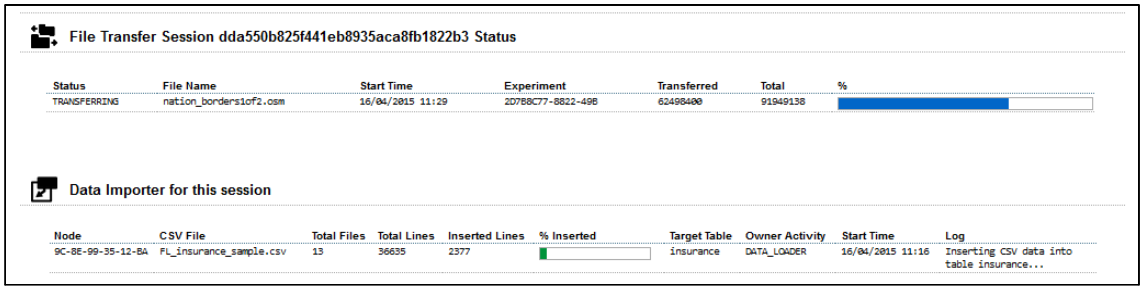


Figura 2. Tela de acompanhamento de transferência de arquivos

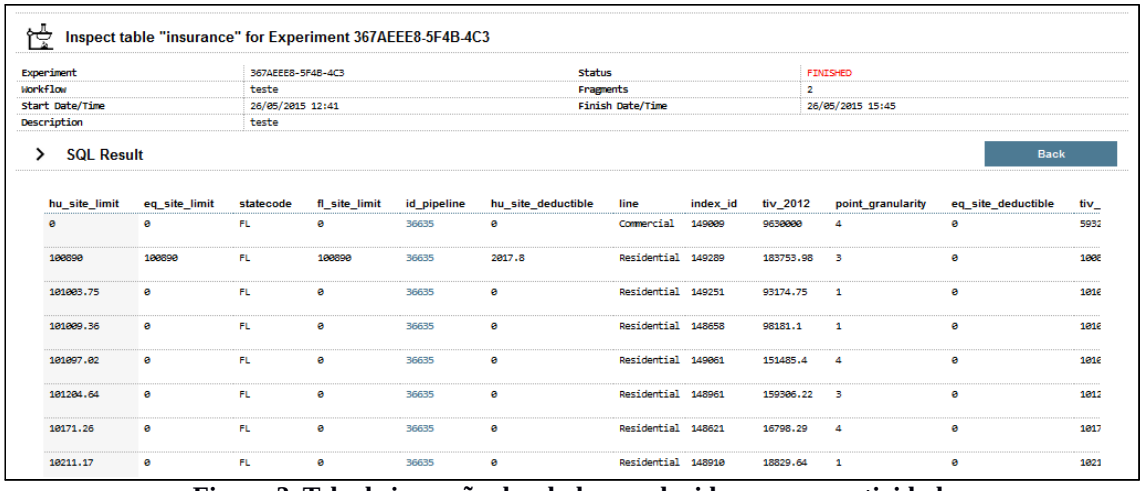


Figura 3. Tela de inspeção dos dados produzidos por uma atividade

O processamento de tarefas em nós de execução

O Sagitarii executa as atividades de workflow de forma distribuída, o que significa que é necessário possuir computadores para executar seus nós de processamento. Em cada nó de processamento será executado o software de execução das atividades, chamado de Teapot Node. Um administrador pode facilmente instalar um novo nó de processamento baixando os arquivos necessários pelo link existente no canto superior direito da interface do Sagitarii. Em teoria, qualquer computador que execute o Teapot e esteja configurado adequadamente pode se tornar um nó de processamento do Sagitarii, o que torna o sistema altamente escalável, tendo em vista que os nós de processamento podem entrar e sair da rede sem comprometer o funcionamento do sistema.

O Teapot funciona em modo multitarefa, o que significa que ele pode executar as atividades em paralelo. A quantidade de atividades que o Teapot pode executar ao mesmo tempo é configurada pelo administrador em arquivo de configuração e, embora o Teapot não possua qualquer limitação para este número, é conveniente respeitar a quantidade de núcleos do processador na máquina que está executando o nó de processamento. No arquivo de configuração existe um parâmetro que força o Teapot a respeitar a quantidade de núcleos, independente do valor atribuído para a quantidade de tarefas em paralelo. Os arquivos de configuração serão cobertos em um capítulo futuro.

O Teapot envia, de tempos em tempos ao Sagitarii, as informações de consumo de CPU e memória, o que pode ser útil para o administrador avaliar a capacidade de processamento de cada nó e configurar estes valores de acordo.

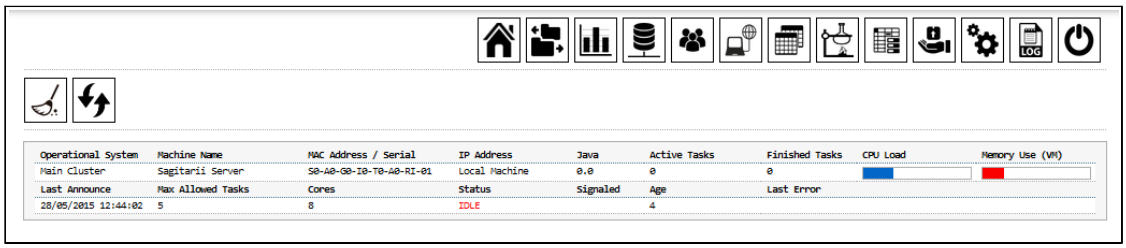


Figura 4. Tela de monitoramento de nós de processamento

Colocar um nó de processamento para funcionar no seu limite durante a execução de um workflow pode causar uma sobrecarga neste nó quando um workflow mais robusto for executado. Cabe ao administrador avaliar cada caso e escolher os valores que achar mais conveniente. A alteração na configuração do Teapot requer que o mesmo seja reiniciado para que tenham efeito.

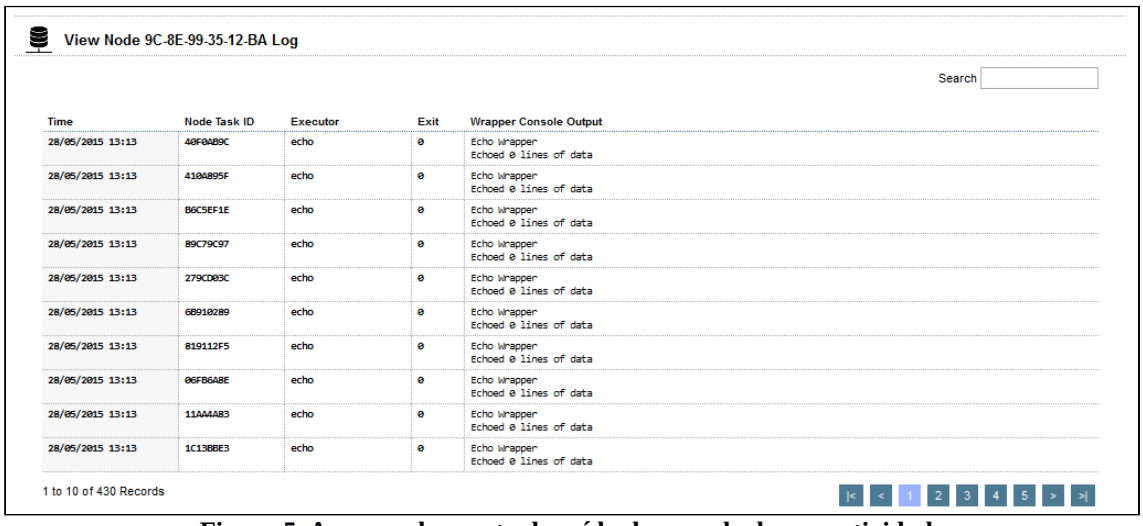


Figura 5. Acompanhamento da saída do console de uma atividade

Adquirindo o Sagitarii

O Sagitarii pode ser baixado diretamente de seu repositório principal no GitHub. Este manual cobre a versão 1.0 do Sagitarii e do Teapot.

```
https://github.com/eic-cefet-rj/sagitarii
```

Caso seja necessário compilar o Sagitarii, é necessário instalar o Maven. Após instalar e configurar o Maven, vá para o diretório raiz do código (mesmo local do arquivo pom.xml) e digite na console de seu sistema operacional:

```
mvn clean package
```

Existe a opção de configurar o Sagitarii antes de instalar e depois de instalar. A opção de configurar antes da instalação requer que o Sagitarii seja compilado novamente. A configuração após a instalação requer que o servidor seja reiniciado. Os únicos arquivos que configuram o Sagitarii são:

Arquivo	Finalidade
config.xml	Configurações do funcionamento do servidor Sagitarii.
log4j2.xml	Configuração do log do servidor Sagitarii. É possível configurar o log por nível (avisos, erro e debug) e por aspecto (somente determinadas classes do sistema serão auditadas). O nível de debug causará uma saída intensa de informações, portanto não é aconselhável permanecer nesta configuração em um ambiente de produção.
hibernate.cfg.xml	Configuração do acesso ao banco de dados.

Estes arquivos estão localizados na pasta “resources”. Uma cobertura completa da configuração do servidor será apresentada no capítulo 3.

É necessário criar o banco de dados no servidor PostgreSQL. A versão atual do Sagitarii utiliza o PostgreSQL versão 9.4.1. Abra o pgAdminIII e crie um banco de dados com o nome “sagitarii”. Caso deseje um banco de dados com outro nome, modifique esta informação no arquivo log4j2.xml.

Após obter o arquivo WAR, faça a instalação no servidor Tomcat. Esta versão do Sagitarii foi testada no Tomcat v7.0.29. Se você optou por configurar o servidor após a instalação, pare a execução do Tomcat, configure o Sagitarii e inicie o Tomcat novamente.

Após sua primeira execução, o Sagitarii irá criar todas as tabelas necessárias e instalar o usuário de adminstração.

Primeiro acesso

Para acessar o Sagitarii, utilize o endereço de seu servidor, seguido do contexto “/sagitarii”. O primeiro acesso é feito com o usuário “admin” usando a senha “admin”.

```
http://sagitarii.server.org/sagitarii
```

Se tudo correu bem, o Sagitarii está pronto para uso.

Figura 6. Tela de login

Requisitos para o Sagitarii:

Requisito	Versão
Java	1.7.0_03
Tomcat	7.0.29
PostgreSQL	9.4.1
Disco (Sagitarii)	80Gb
Disco (PostgreSQL)	O Sagitarii usa o banco de dados PostgreSQL como repositório de arquivos. A capacidade do disco irá limitar o tamanho do repositório e a capacidade do sistema em armazenar os arquivos produzidos pelas atividades.

Instalação do nó de processamento Teapot

Após o Sagitarii ser instalado e estar online, é hora de instalar o primeiro no de processamento. Não é aconselhável instalar um nó de processamento no mesmo computador onde o Sagitarii esteja sendo executado, embora nada impeça esta prática.

Abra um navegador de internet e acesse o Sagitarii. Não é preciso estar logado no servidor. No canto superior direito da tela encontra-se o link para download do Teapot.

Existe a opção de baixar o código fonte do Teapot, para tanto, acesse o repositório no GitHub.

```
https://github.com/eic-cefet-rj/sagitarii-teapot
```

Para compilar o Teapot, use o Maven.

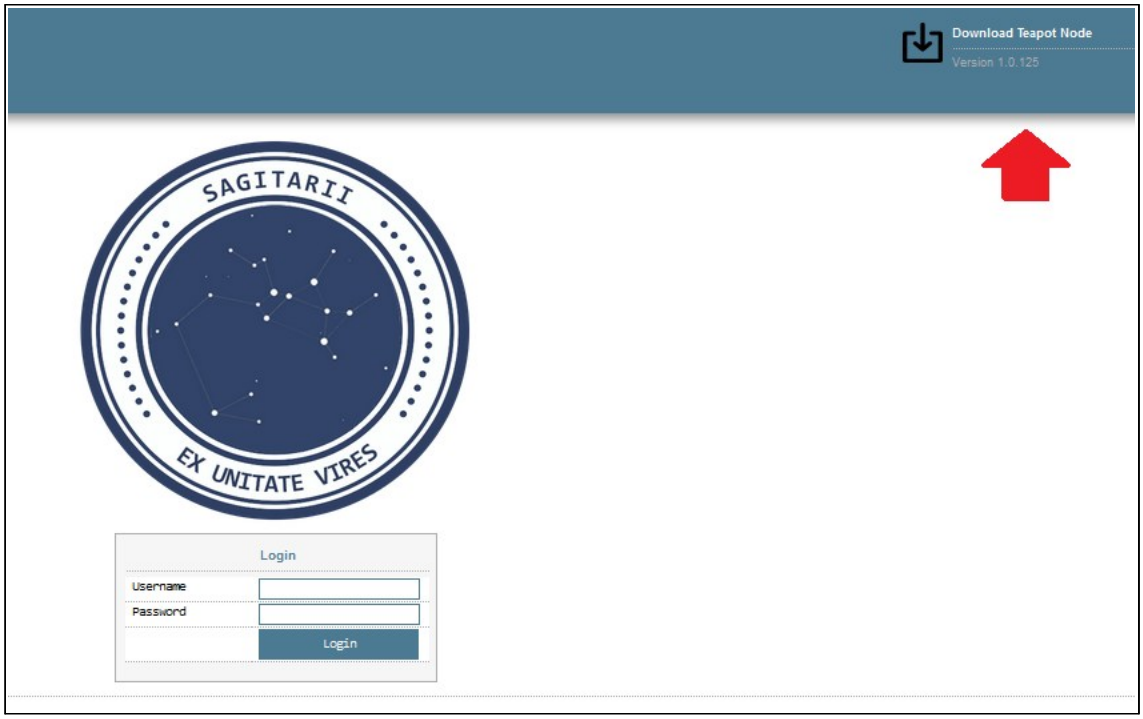


Figura 7. Download do Teapot

A execução do Teapot requer um computador com uma boa configuração, pois o desempenho de cada nó de processamento vai influenciar diretamente no desempenho do sistema. Os requisitos mínimos para executar o Teapot estão na tabela a seguir:

Requisito	Versão
Java	1.7.0_03
Disco	A execução das atividades requer manipulação de arquivos. Dependendo das características do workflow sendo executado, a quantidade de arquivos na pasta de trabalho do Teapot poderá ser excessiva. O Sagitarii possui uma opção para solicitar ao Teapot que limpe o seu diretório de trabalho, mas só poderá ser acionada quando não há workflows em processamento. O Teapot também possui uma opção em sua configuração para limpar a pasta de trabalho de cada atividade ao final de seu processamento. O tamanho do disco influenciará na capacidade do Teapot de processar arquivos de atividades.
Sistema Operacional	Ubuntu Server 14.04.2 LTS
Processador	Intel Core i5 650 (4 núcleos)
RAM	4Gb

O primeiro passo para instalar o Teapot é criar um usuário. Isto é necessário para isolar as operações do Teapot no sistema operacional, pois as atividades irão executar códigos e programas de terceiros, que podem colocar em risco o sistema operacional caso as credenciais do usuário que executa o Teapot permitam.

Execute na console de seu sistema operacional (crie usuário e grupo teapot):

```
adduser teapot
```

Crie uma pasta em /etc/teapot

```
sudo mkdir /etc/teapot
```

Associe o usuário e o grupo à pasta criada:

```
sudo chown teapot:teapot /etc/teapot
```

Dê permissões de escrita, execução e leitura para o usuário e grupo teapot na pasta criada. Os demais usuários poderão somente ler a pasta:

```
sudo chmod 774 /etc/teapot
```

Descompacte os arquivos do Teapot na pasta criada. O Teapot possui os seguintes arquivos de configuração:

Arquivo	Finalidade
config.xml	Configurações do funcionamento do Teapot.
log4j2.xml	Configuração do log do Teapot. É possível configurar o log por nível (avisos, erro e debug) e por aspecto (somente determinadas classes do sistema serão auditadas). O nível de debug causará uma saída intensa de informações, portanto não é aconselhável permanecer nesta configuração em um ambiente de produção. Quando o Teapot é executado em modo interativo, a saída do log para a console é omitida.

Configure o Teapot. Os detalhes dos arquivos de configuração são cobertos no capítulo 3.

Se tudo correu bem, o Teapot pode ser executado. É aconselhável executar o Teapot como um serviço, desta forma ele não ocupará o terminal. Também é possível executá-lo em modo interativo. Neste modo, uma console será apresentada e será possível interagir com o Teapot, consultando detalhes sobre as tarefas sendo executadas, status do nó de processamento e alterando alguns aspectos da configuração.

O Teapot também pode ser utilizado para enviar dados para as tabelas de entrada das atividades de um workflow. Isto é necessário para a execução do mesmo. As diversas formas de executar o Teapot são detalhadas no capítulo 4.

Logue como o usuário “teapot” no computador. Para executar o Teapot digite em seu terminal:

```
java -jar teapot-1.0.125.jar &
```

Este comando irá liberar a console. Caso deseje ocupar a console até o encerramento do programa, remova o “&” no final do comando. Na interface do Sagitarii, na tela de monitoramento dos nós de processamento, existem opções para reiniciar e encerrar o Teapot.

A cada execução o Teapot limpa a pasta de trabalho e o arquivo de log.

Configuração do Sagitarii

O sagitarii possui 3 arquivos que controlam sua configuração:

Arquivo	Finalidade
config.xml	Configurações do funcionamento do servidor Sagitarii.
log4j2.xml	Configuração do log do servidor Sagitarii. É possível configurar o log por nível (avisos, erro e debug) e por aspecto (somente determinadas classes do sistema serão auditadas). O nível de debug causará uma saída intensa de informações, portanto não é aconselhável permanecer nesta configuração em um ambiente de produção.
hibernate.cfg.xml	Configuração do acesso ao banco de dados.

Os arquivos “log4j2.xml” e “hibernate.cfg.xml” são arquivos de configuração padrão para o Apache Log4J e Hibernate, respectivamente. O arquivo config.xml é para a configuração do próprio Sagitarii.

O Arquivo “log4j2.xml” modifica o comportamento do log do sistema. Este arquivo deve ser alterado somente quando for necessário inspecionar alguma possível falha no sistema, pois produz uma grande quantidade de dados. O log pode ser direcionado para a console ou para um arquivo. Caso o log seja direcionado para um arquivo, este pode ser consultado na própria interface do Sagitarii, pois na barra de botões existe a opção de visualização do log.

```
21      <Logger name="cmabreu.sagitarii.core.sockets.FileImporter" level="error" additivity="false">
22          <appender-ref ref="Console"/>
23      </Logger>
24
25      <Logger name="cmabreu.sagitarii.api.ExternalApi" level="error" additivity="false">
26          <appender-ref ref="Console"/>
27      </Logger>
28
29      <Logger name="cmabreu.sagitarii.core.mail.MailService" level="error" additivity="false">
30          <appender-ref ref="Console"/>
31      </Logger>
32
33      <Logger name="cmabreu.sagitarii.core.sockets.FileSaver" level="error" additivity="false">
34          <appender-ref ref="Console"/>
35      </Logger>
```

Figura 8. Trecho do arquivo de log “log4j2.xml”

Através deste arquivo, cada componente interno do sistema pode ser instruído de forma independente a exibir o log de suas operações. Por exemplo: caso seja necessário acompanhar o andamento do núcleo do sistema (entrega e recebimento de instâncias, controle do buffer, execução de fragmentos, etc...) deve-se trocar a opção “level” do componente "cmabreu.sagitarii.core.Sagitarii" de “error” para “debug”.

Os níveis de log podem ser:

- ERROR: Somente haverá saída de log em caso de erro no componente especificado.
- DEBUG: O componente especificado exibirá todos detalhes de sua operação interna.

Para alterar a saída do log entre a console ou o arquivo de saída, modifique a opção “appender-ref” para “Console” ou “File” conforme for o caso.

Em um ambiente de produção, o ideal é manter o log para o arquivo e todos os componentes marcados como “level” = “error”.

O arquivo “hibernate.cfg.xml” controla os aspectos de acesso ao banco de dados e pool de conexões.

```
4  <hibernate-configuration>
5      <session-factory>
6          <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
7          <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
8
9
10         <property name="hibernate.connection.url">jdbc:postgresql://localhost/sagitarii?ApplicationName=Sagitarii</property>
11         <property name="hibernate.connection.username">postgres</property>
12         <property name="hibernate.connection.password">admin</property>
13
14
15         <property name="hibernate.hbm2ddl.auto">update</property>
16
17         <property name="show_sql">false</property>
18         <property name="format_sql">true</property>
19
20         <property name="hibernate.id.new_generator_mappings">true</property>
```

Figura 9. Trecho do arquivo “hibernate.cfg.xml”

As únicas opções que podem ser alteradas neste arquivo estão nas linhas 10, 11 e 12. A linha 10 indica a localização e o nome do banco de dados, bem como um nome para identificar o Sagitarii na engine do PostgreSQL. As linhas 11 e 12 permitem configurar o usuário e a senha usados para acessar o banco de dados em questão.

ATENÇÃO: Não altere as demais informações contidas neste arquivo, a menos que saiba exatamente o que está fazendo. Configurações incorretas neste arquivo podem causar desde perda de dados por reconstrução do banco até queda de desempenho do sistema, bem como corrupção de dados e travamento do servidor PostgreSQL.

```
15 <configuration>
16
17     <orchestrator>
18         <poolIntervalSeconds>1</poolIntervalSeconds>
19         <pseudoClusterIntervalSeconds>5</pseudoClusterIntervalSeconds>
20         <pseudoMaxTasks>5</pseudoMaxTasks>
21         <maxInputBufferCapacity>300</maxInputBufferCapacity>
22         <mainNodesQuant>1</mainNodesQuant>
23         <fileReceiverPort>3333</fileReceiverPort>
24         <fileReceiverChunkBufferSize>100</fileReceiverChunkBufferSize>
25         <CSVDelimiter>,</CSVDelimiter>
26     </orchestrator>
27
28 </configuration>
```

Figura 10. Trecho do arquivo “config.xml” do Sagitarii

O arquivo “config.xml” controla alguns aspectos do funcionamento do próprio Sagitarii. Este arquivo é descrito em detalhes na tabela abaixo:

Item	Finalidade
poolIntervalSeconds	É a “batida de coração” do sistema. Configura o intervalo, em segundos, que o Sagitarii vai executar as operações de verificação e criação de instâncias, término de execução de fragmentos e experimentos, ciclo do escalonamento de experimentos, etc. Um valor muito alto fará com que o sistema trabalhe mais lentamente, enquanto um valor muito baixo poderá causar acessos desnecessários ao banco de dados com frequência. Um valor entre 3 e 5 pode ser ideal na maioria dos casos.
pseudoClusterIntervalSeconds	Configura o intervalo de funcionamento do nó de processamento interno (destinado a processar executores tipo “SELECT”. É nesta frequência que o nó de processamento interno vai consultar o pool de instâncias se existem atividades tipo “SELECT” a serem executadas. Um valor alto demais fará o sistema ficar “preguiçoso” enquanto um valor muito baixo poderá causar acesso desnecessário ao pool de instâncias. 5 segundos é um valor razoável na maioria dos casos.
pseudoMaxTasks	Quantidade de tarefas que o nó de processamento interno poderá processar ao mesmo tempo.
maxInputBufferCapacity	Quantidade de instâncias que o Sagitarii vai armazenar em buffer de memória para ser entregue aos nós de processamento. A cada intervalo de “poolIntervalSeconds” o Sagitarii vai ao banco de dados em busca de instâncias prontas para execução de um determinado experimento. Um valor muito alto irá processar mais instâncias de um experimento e fará menos acessos ao banco de dados, mas poderá comprometer o escalonamento de experimentos, pois um único experimento ficará com uma fatia de processamento muito grande. Um valor baixo aumentará o escalonamento, processando vários experimentos de uma forma mais justa, porém o sistema poderá ficar “preguiçoso”, visto que o buffer irá esvaziar antes de “poolIntervalSeconds” completar um ciclo e ir ao banco para coletar mais instâncias.
mainNodesQuant	Número de nós de processamento internos para tarefas tipo “SELECT”. Somente em casos em que existam workflows com muitas atividades “SELECT” ou se é previsto que atividades “SELECT” demorem muito, este número deverá ser alterado. O padrão é 1.
fileReceiverPort	Porta de acesso para o receptor de arquivos. O Teapot tentará conectar nesta porta para enviar ao Sagitarii os arquivos produzidos pelas atividades que executa. O padrão é 3333. Este valor deverá ser o mesmo que o existente na configuração do Teapot.
fileReceiverChunkBufferSize	Tamanho, em bytes, do bloco de transferência de arquivos entre o Teapot e o Sagitarii. O padrão é 100 bytes. Este valor deverá ser o mesmo que o existente na configuração do Teapot.
CSVDelimiter	Caractere que delimita as colunas de um arquivo CSV. O padrão é a “vírgula”.

Configuração do Teapot

O Teapot possui dois arquivos de configuração, conforme a tabela abaixo:

Arquivo	Finalidade
config.xml	Configurações do funcionamento do Teapot.
log4j2.xml	Configuração do log do Teapot. É possível configurar o log por nível (avisos, erro e debug) e por aspecto (somente determinadas classes do sistema serão auditadas). O nível de debug causará uma saída intensa de informações, portanto não é aconselhável permanecer nesta configuração em um ambiente de produção. Quando o Teapot é executado em modo interativo, a saída do log para a console é omitida.

```
18      <cluster>
19          <hostURL>http://localhost:8580/sagitarii/</hostURL>
20          <poolIntervalMilliSeconds>1000</poolIntervalMilliSeconds>
21          <activationsMaxLimit>50</activationsMaxLimit>
22          <rPath>C:/rJava/jri</rPath>
23          <clearDataAfterFinish>false</clearDataAfterFinish>
24          <CSVDelimiter>,</CSVDelimiter>
25          <storageHost>localhost</storageHost>
26          <storagePort>3333</storagePort>
27          <fileSenderDelay>200</fileSenderDelay>
28          <useSpeedEqualizer>true</useSpeedEqualizer>
29          <enforceTaskLimitToCores>false</enforceTaskLimitToCores>
30      </cluster>
```

Figura 11. Trecho do arquivo "config.xml" do Teapot

O arquivo "log4j2.xml" finciona da mesma forma descrita para a configuração do Sagitarii, já o arquivo "config.xml" configura os aspectos do funcionamento interno do próprio Teapot e é descrito em detalhes conforme a seguir:

Item	Finalidade
hostURL	Endereço (URL) do servidor Sagitarii. O Teapot se comunica com o Sagitarii utilizando requisições GET / POST através desta URL.
poolIntervalMilliSeconds	É a "batida de coração" do Teapot. A cada intervalo de tempo especificado (em milissegundos) o Teapot vai questionar o Sagitarii se existem tarefas a serem executadas. Um valor muito baixo fará com que o Teapot sobrecarregue o Sagitarii (vários nós podem causar flood e/ou negação de serviço no servidor). Um valor muito alto fará o sistema ficar "preguiçoso". Intervalos entre 1000 e 3000 podem ser aceitáveis na maioria dos casos. Um valor acima de 5000 poderá induzir ao Sagitarii a achar que o nó de processamento está "morto", pois ao questionar o Sagitarii, o Teapot envia informações sobre o nó e também um sinal de "keepalive" (KA).

activationsMaxLimit	Quantidade de atividades que o Teapot pode executar em paralelo. Não há um limite para este valor, mas deve-se considerar a capacidade de processamento do computador que está executando o Teapot para não criar “gargalos” na execução do workflow. Mantendo este valor igual a quantidade de núcleos da máquina, fará com que cada atividade seja executada em um núcleo.
rPath	Caminho das bibliotecas do R. O padrão para o Ubuntu é /usr/lib64/R/library/rJava/jri/
clearDataAfterFinish	Indica ao Teapot que ele deve limpar os dados de uma atividade assim que ela terminar. É útil para manter o espaço em disco sob controle e deve ser usado em ambiente de produção.
CSVDelimiter	Caractere que delimita as colunas de um arquivo CSV. O padrão é a “vírgula”.
storageHost	O endereço do receptor de arquivos do Sagitarii. Normalmente é usado o endereço IP do Sagitarii.
storagePort	Porta do receptor de arquivos do Sagitarii. Deve ser o mesmo que o especificado em “fileReceiverPort” na configuração do Sagitarii. O padrão é 3333.
fileSenderDelay	Tempo, em millisegundos, que o uploader deverá aguardar antes de fechar a conexão de envio. Um valor muito próximo de zero poderá causar um fechamento prematuro da conexão enquanto o Sagitarii ainda está gravando o arquivo. Um valor muito alto fará com que uma atividade demore mais para encerrar. O padrão é 200 milissegundos.
useSpeedEqualizer	Instrui ao Teapot que ele deve balancear o intervalo de execução (“poolIntervalMilliSeconds”) de acordo com a quantidade de tarefas sendo executadas. O Teapot vai aumentar o intervalo se estiver próximo da ociosidade, fazendo com que menos requisições por segundo cheguem ao Sagitarii, até o limite de 3.500ms. Vai aumentar este valor também caso esteja trabalhando no limite de execuções. Ele vai reduzir este valor de estiver no meio-termo, aumentando também a velocidade de processamento (se as tarefas estão terminando rápido, então é melhor pedir mais) até o limite de 1000ms.
enforceTaskLimitToCores	Obriga ao Teapot a limitar “activationsMaxLimit” de acordo com a quantidade de núcleos no processador da máquina, independente da configuração. Isso irá desligar o balanceamento (“useSpeedEqualizer”).

Capítulo 4 : Modos de Execução do Teapot

O Teapot poderá ser executado de 3 formas distintas:

- Modo normal: Inicia a operação de processamento de atividades.
- Modo interativo: Inicia a operação de processamento de atividades, desliga a saída de log para a console, caso haja, e disponibiliza uma console de interatividade com o sistema.
- Modo de carga de dados: Faz o upload dos dados (CSV e arquivos) para a tabela de um determinado experimento e encerra a operação.

Para iniciar o Tepot em modo normal, use o seguinte comando:

```
java -jar teapot-1.0.125.jar &
```

Isso fará com que o Teapot conecte-se com o Sagitarii, baixe todos os executores disponíveis no servidor, limpe o diretório de trabalho e inicie o processamento, enviando os dados do computador, consumo de CPU e memória ao Sagitarii a cada intervalo de tempo definido em "`poolIntervalMilliSeconds`". Todas as vezes que o Sagitarii receber estas informações ele vai entregar ao Teapot um pacote com instruções para executar determinadas atividades (instância), caso haja.

O Teapot poderá ser usado para fazer o upload dos dados iniciais para o experimento de um workflow. Para executar esta operação, é necessário que o experimento já exista, bem como a tabela que vai receber os dados. O upload de dados iniciais será coberto em detalhes em capítulo futuro.

O comando para enviar dados usando o Teapot é:

```
java -jar teapot-1.0.125.jar upload csv_file target_table exp_serial work_dir
```

Com o comando acima, o Teapot vai iniciar, baixar os executores, limpar o diretório de trabalho e iniciar o upload do arquivo indicado em "csv_file" para que o mesmo seja inserido na tabela "target_table", no contexto do experimento "exp_serial". É necessário informar a pasta onde está o arquivo CSV ("work_dir"), para que o Teapot verifique a subpasta "outbox" em busca de potenciais arquivos a serem enviados. O processo de carga de tabelas será coberto em detalhes em capítulo futuro.

O último modo de execução do Teapot é o modo interativo. Neste modo de execução o Teapot inicia, baixa os executores, limpa a pasta de trabalho, inicia o processamento de atividades e libera uma console para interatividade.

O comando para iniciar o Teapot em modo interativo é:

```
java -jar teapot-1.0.125.jar interactive
```

Atualmente a interatividade é limitada a apenas alguns comandos básicos, descritos na tabela abaixo:

Comando	Parâmetros	Finalidade
system	pause	Faz com que o Teapot pare de iniciar novas tarefas. O Sagitarii pode interpretar que este nó de execução está “morto”. As tarefas que estiverem em execução continuam até terminar.
	resume	O Teapot retoma o trabalho pausado.
	whoami	Exibe o nome do computador, o endereço MAC e o endereço IP.
	loadlevel	Exibe a quantidade de atividades que estão sendo processadas e a carga de CPU.
show	tasks	Exibe detalhes das atividades que estão sendo executadas (hora de início e tempo gasto).
	total	Exibe o número de atividades processadas desde o início da execução.
	instances	Mostra detalhes de cada instância processada. Cada instância pode conter uma ou mais atividades, sendo assim o comando irá exibir, para cada instância, o status de suas atividades e o tempo gasto de cada uma.
quit		Encerra imediatamente a execução do Teapot. As atividades em execução serão perdidas.
upload	(mesmos do modo de envio de dados)	Executa a mesma operação quando se inicia o Teapot em modo de envio de dados. Os parâmetros são os mesmos.
help		Exibe a lista de comandos disponíveis.
logger	enable	Reabilita a exibição do log, caso esteja habilitado no arquivo de configuração. Tornará a tela da console confusa.
	disabe	Desabilita novamente a exibição do log, caso tenha sido habilitado pelo comando “logger enable”.

A rede Sagitarii

O Sagitarii funciona em conjunto com outros dois componentes em sua rede: o nó de processamento Teapot e o servidor de banco de dados PostgreSQL.

O Teapot não possui acesso direto ao banco de dados. Toda sua comunicação é feita diretamente com o Sagitarii.

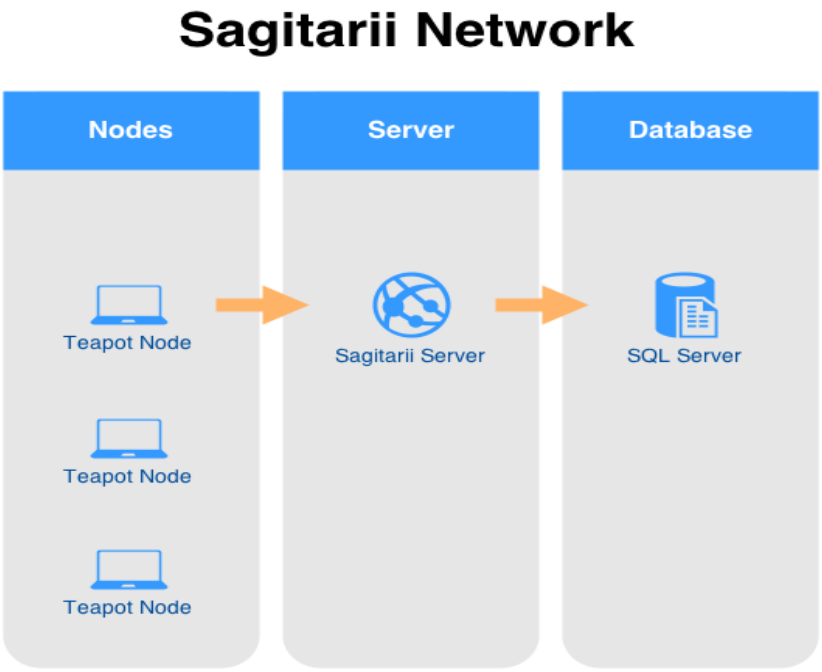


Figura 12. Diagrama da rede do Sagitarii

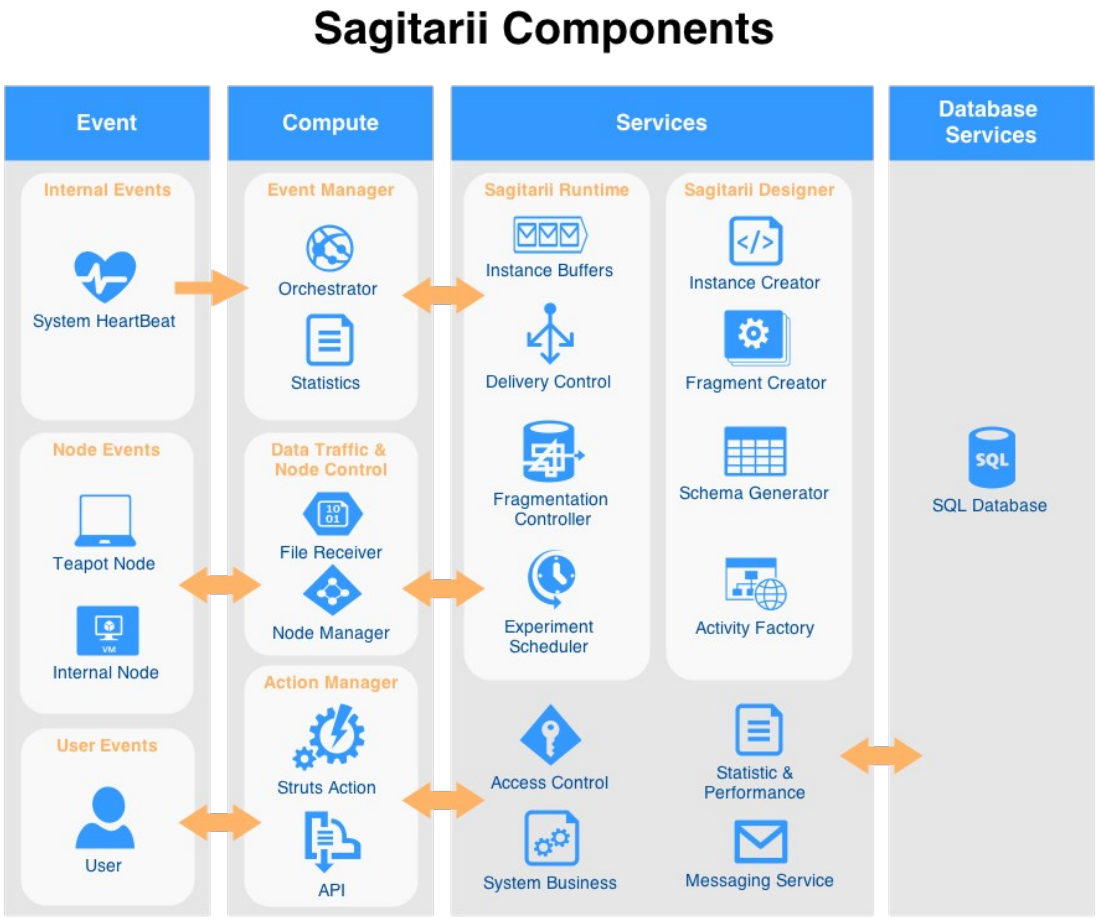


Figura 13. Componentes internos do Sagitarii

Workflow

No Sagitarii, um workflow é a especificação do trabalho a ser feito. Um workflow consiste em um nome, uma descrição e as especificações das atividades (fluxo e tabelas). Não é possível executar diretamente um workflow.

Experimento

Um experimento é a instância executável de um workflow. Um workflow pode ter vários experimentos, sendo possível alterar alguns aspectos na definição das atividades, modificando o fluxo, o tipo de executor ou a fonte de dados das atividades. A ideia é que o usuário possa "experimentar" várias formas de executar o mesmo workflow a fim de determinar um melhor resultado de seu trabalho.

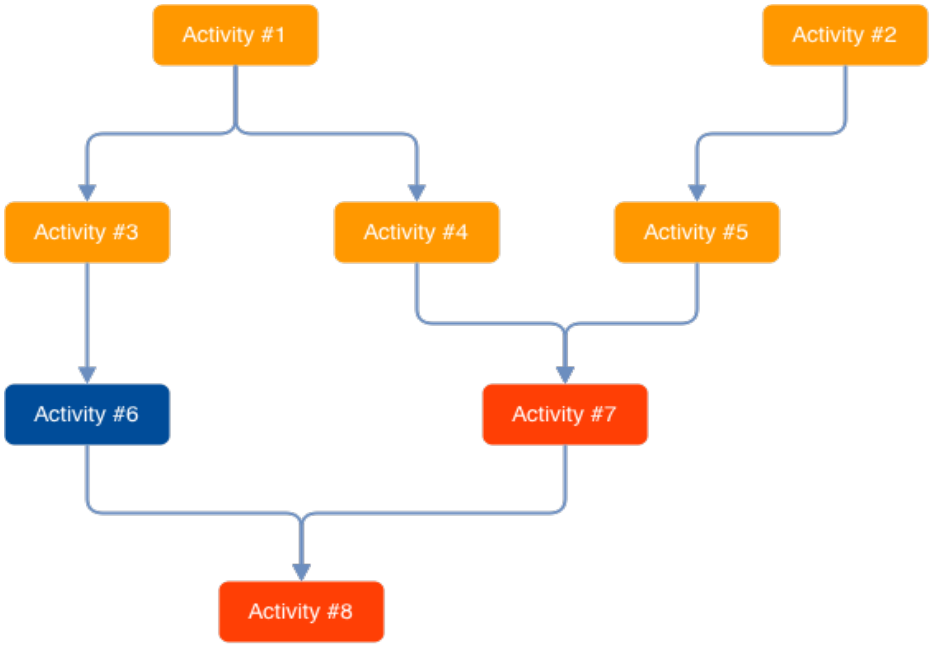


Figura 14. Exemplo de Workflow

O experimento contém, inicialmente, a cópia fiel das definições de um workflow, podendo ser executado imediatamente ou editado. O experimento possui também um número de série que o diferencia dos demais experimentos.

Usuários diferentes podem criar experimentos a partir de um mesmo workflow, mas seus dados e resultados não serão visíveis uns aos outros.

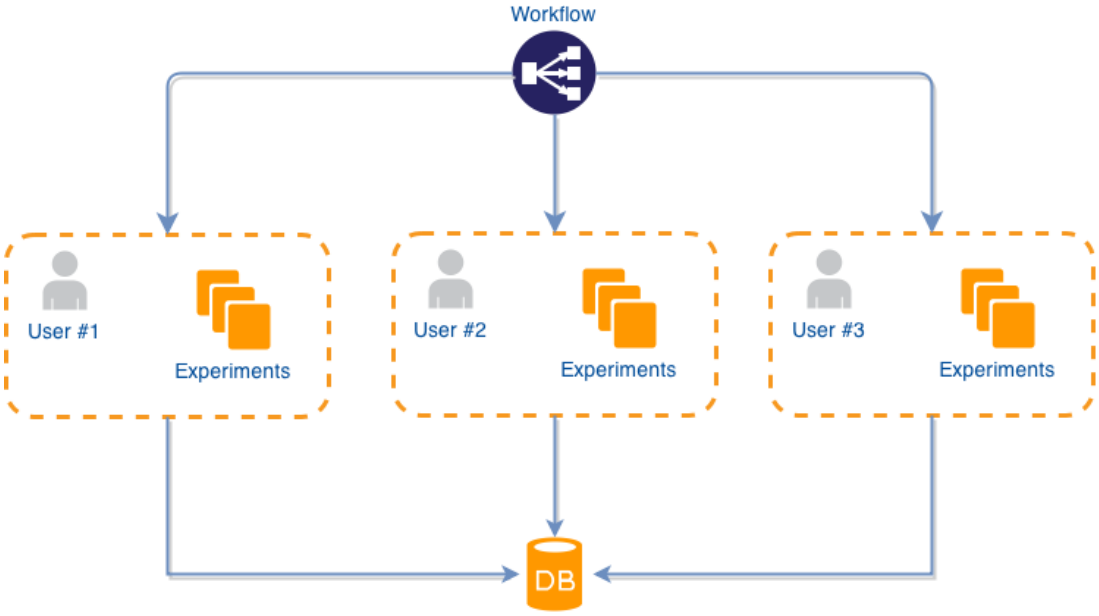


Figura 14. Workflow e Experimentos

Atividades

Uma atividade é uma unidade de execução de um workflow. Cada atividade vai executar uma operação algébrica em uma ou mais tabelas de entrada e seu resultado será armazenado em uma tabela de saída, que poderá ser consumida pela atividade seguinte, e assim sucessivamente até o final da execução.

Fragmento

Um fragmento é um conjunto de atividades que podem ser executadas em série. Dada a natureza das operações algébricas que regem o processamento dos workflows no Sagitarii, algumas atividades podem ser executadas em cadeia por um mesmo nó de processamento, para agilizar o processamento. É o caso das atividades com operador tipo "MAP", representadas na cor laranja na figura 14. Existem outras atividades que devem aguardar a conclusão de todas as atividades anteriores antes de iniciar seu próprio processamento, pois precisam consumir dados de mais de uma tabela de entrada ou usar um critério seletivo em sua tabela de entrada. É o caso das atividades tipo "REDUCE" e "SELECT", representadas respectivamente nas cores azul e vermelha na figura 14.

Os operadores algébricos serão cobertos em detalhes em capítulo futuro.

Wrapper

Um wrapper é um aplicativo que empacota a execução de outro aplicativo. O conceito de wrapper foi criado para flexibilizar a execução de programas complexos nos nós de execução, assim, o Sagitarii não precisa conhecer detalhes sobre cada programa que será utilizado na execução de um workflow, apenas possuir os wrappers corretos para executar cada atividade. O Sagitarii se comunica com todos os wrappers da mesma forma: passando um arquivo CSV com os dados a serem consumidos e eventuais arquivos necessários para execução da atividade.

O wrapper é, normalmente, um pequeno programa feito em Java, que interpreta o arquivo de entrada enviado pelo Sagitarii e passa para a aplicação que foi empacotada de forma que ela compreenda os dados. O benefício é que o Sagitarii não precisa saber como cada aplicação espera os dados.

Figura 15. Diagrama de execução de um wrapper

Instância

Uma instância é uma ordem de execução de uma ou mais atividades, dada pelo Sagitarii à um nó de processamento. Quando o Sagitarii executa um workflow, primeiramente ele avalia as atividades que podem ser executadas (se a atividade é o ponto de entrada de seu fragmento, se as suas tabelas de entrada possuem dados e, conforme o caso, se as atividades anteriores já terminaram sua execução).

Capítulo 7 : Operadores da Álgebra Relacional no Sagitarii

Capítulo 8 : Otimização de Workflows

Capítulo 9 : Wrappers

****** DAQUI PARA BAIXO ESTÁ DESATUALIZADO ******

O que é um wrapper?

O Sagitarii foi concebido para executar praticamente qualquer tipo de tarefa de modo distribuído. Porém, prever quais os tipos de programas seriam executados nos nós de processamento seria uma tarefa inviável, pois comprometeria a versatilidade do Sagitarii.

Para resolver este problema, criou-se o conceito de “wrapper”, que funciona como intermediário entre o Sagitarii e o programa a ser executado nas máquinas que estão processando as atividades (nós de processamento).

Quando existe uma tarefa a ser realizada (compactar arquivos, produzir gráficos, etc) o Sagitarii envia os dados da tarefa para um nó de processamento juntamente com o nome do wrapper que será encarregado de executar esta tarefa. Ao iniciar sua execução, o Teapot baixa do Sagitarii todos os wrappers disponíveis, assim, todas as máquinas que estão servindo ao Sagitarii como nó de processamento possuem todos os wrappers necessários nas mesmas versões.

Ao receber os dados e o nome do wrapper, o Teapot cria uma pasta exclusiva para esta instância de execução do wrapper, salva os dados que recebeu do Sagitarii para esta atividade nesta pasta e executa o wrapper, passando seu caminho completo como parâmetro ao wrapper.

O wrapper quando inicia a execução, deve recuperar o caminho de sua pasta exclusiva de seu parâmetro e carregar os dados que recebeu do Sagitarii, formatar estes dados conforme for conveniente e então iniciar o programa para qual foi criado para empacotar, passando os parâmetros de maneira adequada a este programa.

Caso os dados que recebeu do Sagitarii instrua o wrapper a usar algum arquivo (esta já deverá estar armazenado no Sagitarii), o Teapot o terá baixado do Sagitarii e gravado na subpasta “inbox” da pasta de trabalho da atividade.

Quando o programa que o wrapper empacotou terminar, o wrapper deve receber os dados de saída deste programa e enviar ao Sagitarii, juntamente com eventuais arquivos produzidos, encerrando assim a execução da atividade.

Existem duas formas de enviar dados ao Sagitarii: enviar um arquivo diretamente ao servidor e/ou produzir dados no formato CSV. Quando um programa cria algum arquivo (imagem, arquivos compactados e arquivos binários em geral, mais ligados ao experimento propriamente dito) é necessário enviar este ao servidor. Já os dados de atividade (mais ligados ao fluxo da informação) são normalmente em formato CSV e são os dados de entrada e saída das atividades. Estes são armazenados nas tabelas de produto e consumo destas atividades.

Por exemplo, suponha que seja necessário realizar um certo cálculo sobre determinada informação e o resultado deste cálculo deva ser compactado em um arquivo para ser utilizado na atividade seguinte. O Sagitarii envia os dados (oriundos da tabela de entrada da atividade) ao Teapot. Este executa o wrapper sobre estes dados, que os repassa ao programa especialista. O arquivo resultante (compactado) é enviado diretamente ao Sagitarii pelo Teapot ao final da execução do wrapper, mas o wrapper deve salvar este arquivo em sua pasta de saída (outbox), só assim o Teapot saberá que precisa enviar arquivos para o servidor. O wrapper passa os dados necessários para a tabela de saída de atividade na forma de CSV ao Teapot gravando-os em um arquivo de texto chamado “sagi_output.txt” na raiz da pasta de trabalho da atividade. Quando o wrapper termina, o Teapot lê este arquivo, enviando-o ao Sagitarii, que decompõe este CSV e armazena cada coluna em sua coluna correspondente na tabela de saída da atividade. Cada linha do CSV irá se tornar uma tupla nesta tabela. O Sagitarii possui capacidade para descobrir se uma coluna existe ou não na tabela que receberá o CSV, então, as colunas do CSV que existem na tabela receberão dados e as que não existem serão ignoradas. No caso do exemplo, estes dados poderiam conter o nome do arquivo produzido, para que a próxima atividade saiba como encontrá-lo mais tarde. Quando uma coluna do arquivo CSV “sagi_output.txt” possui o nome de um arquivo e ele existe na pasta “outbox”, o Teapot envia-o ao Sagitarii para que seja armazenado no repositório de arquivos.

Os tipos de wrapper existentes

Embora o conceito de wrapper tenha tornado praticamente infinitas as tarefas a serem executadas pelo Sagitarii, este precisa classificar os wrappers de acordo com as atividades a serem executadas. Quando uma atividade está sendo criada, somente os wrappers correspondentes são exibidos para serem selecionados. Existem seis tipos de wrappers:

- MAP: Os wrappers do tipo MAP executam atividades do tipo MAP. A informação recebida do Sagitarii é um CSV contendo uma linha que representa uma tupla da tabela de entrada da atividade e a informação produzida deverá ser uma linha CSV que será inserida como uma tupla na tabela de saída da atividade. No momento do desenvolvimento de um wrapper do tipo MAP, deve-se esperar e obedecer este formato de dados.

- REDUCE: Os wrappers do tipo REDUCE executam atividades do tipo REDUCE. A informação recebida é composta por uma ou mais linhas CSV contendo uma seleção com distinção em cima de atributos da(s) tabela(s) de entrada da atividade (figura 1). A informação produzida deverá ser composta por um CSV contendo uma ou mais linhas que serão inseridas na tabela de saída da atividade e/ou arquivos de dados.

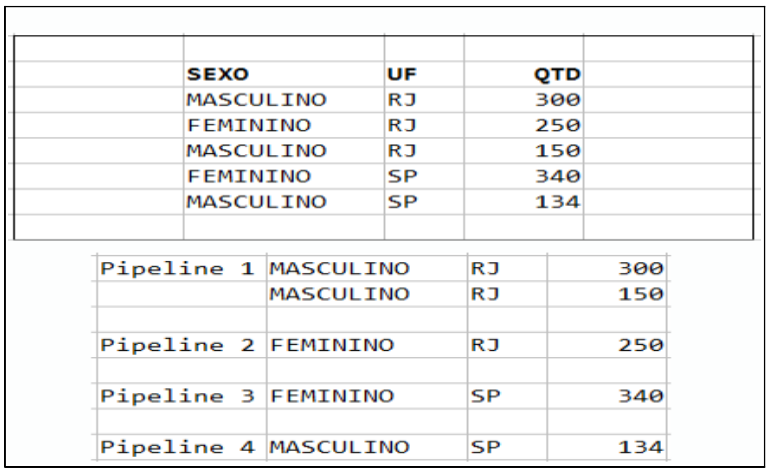
- SPLIT-MAP: Wrappers tipo SPLIT-MAP executam atividades tipo SPLIT-MAP. A informação recebida é composta por uma ou mais linhas CSV. A informação produzida deverá ser composta por um CSV contendo uma ou mais linhas que serão inseridas na tabela de saída da atividade e/ou arquivos de dados.

- R-SCRIPT: Os wrappers tipo R-SCRIPT são arquivos de texto contendo um script na linguagem “R”. Estes arquivos são executados por wrappers tipo R-RUNNER, que executam o processador “R” e passam o script como parâmetro. Atividades tipo MAP e REDUCE podem executar wrappers do tipo R-SCRIPT.

- R-RUNNER: Os wrappers tipo R-RUNNER precisam do processador “R” instalado nos nós de processamento para executar os scripts escritos usando a linguagem “R”. A instalação padrão do Sagitarii oferece um wrapper tipo R-RUNNER, não havendo necessidade de substituição ou criação de um outro wrapper.

É importante informar corretamente o tipo de wrapper ao Sagitarii, para que a formatação dos dados a serem enviados aos nós de processamento seja de acordo com a atividade que está sendo executada. Da mesma forma, é importante formatar os dados de saída de acordo com o tipo de wrapper.

Qualquer arquivo de biblioteca, necessário ao wrapper, que deva ser enviado aos nós de processamento poderá ser cadastrado como um wrapper (futuramente será criado o tipo LIBRARY, que não poderá ser associado a nenhuma atividade). Os nós de processamento receberão todos os wrappers cadastrados assim que iniciarem a execução do Teapot Cluster.



	SEXO	UF	QTD
	MASCULINO	RJ	300
	FEMININO	RJ	250
	MASCULINO	RJ	150
	FEMININO	SP	340
	MASCULINO	SP	134

Pipeline 1	MASCULINO	RJ	300
	MASCULINO	RJ	150
Pipeline 2	FEMININO	RJ	250
Pipeline 3	FEMININO	SP	340
Pipeline 4	MASCULINO	SP	134

Figura 1. Dados enviados para um wrapper tipo “REDUCE” (cada pipeline)

Como o Teapot Cluster executa um wrapper

Ao iniciar a execução, o Teapot Cluster baixa um arquivo de manifesto (em formato XML) do servidor Sagitarii. Este arquivo descreve todos os wrappers cadastrados no Sagitarii (figura 2). Após baixar o arquivo de manifesto, o Teapot inicia o download de todos os wrappers descritos no arquivo (figura 3).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <manifest>
3      <wrapper name="CONSOLIDA_DADOS" type="REDUCE" target="ANY" version="1.0">
4          <activityFile>consolida_dados.jar</activityFile>
5          <reload>true</reload>
6      </wrapper>
7      <wrapper name="BAR_FUNCTION" type="RSCRIPT" target="ANY" version="1.0">
8          <activityFile>bar.r</activityFile>
9          <reload>true</reload>
10     </wrapper>
11     <wrapper name="SEL_001" type="SELECT" target="ANY" version="1.0">
12         <activityFile>null</activityFile>
13         <reload>true</reload>
14     </wrapper>
```

Figura 2. Trecho de um arquivo de manifesto

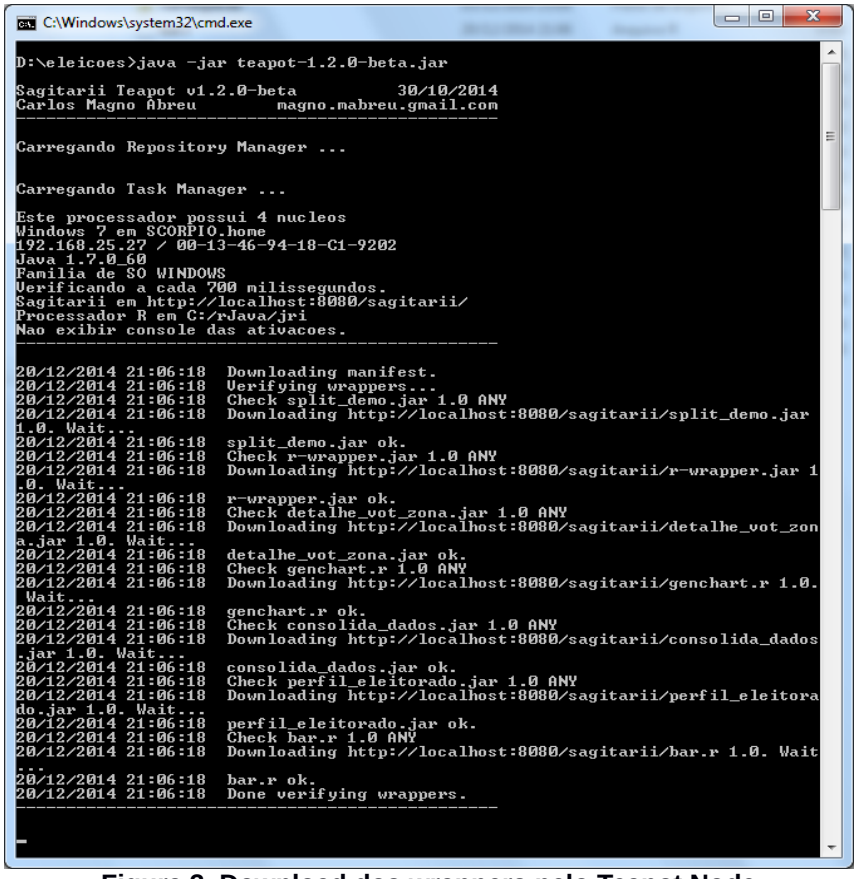


Figura 3. Download dos wrappers pelo Teapot Node

Quando o Sagitarii envia instruções para um nó de processamento executar uma tarefa, estas instruções vêm em formato XML, contendo dados como: nome do workflow, do experimento, da atividade, o nome dos wrappers que deverão ser executados e os dados que devem ser passados a este wrapper (figura 4). Este arquivo é chamado de pipeline.

```xml
<?xml version='1.0' encoding='UTF-8'?>
<pipeline
    workflow='ANALISE_ELEITORAL'
    experiment='C8174597-5480-470'
    serial='0B23F3E2-725B-4'
    fragment='5349E2A4'>
    <activity>
        <order>0</order>
        <serial>94B7EF2B</serial>
        <executorType>MAP</executorType>
        <taskIdChain>
        </taskIdChain>
        <executor>PROJ_PER_ELEIT</executor>
        <type>MAP</type>
        <inputData>
            grau_de_escolaridade;uf;sexo;faixa_etaria;qtd_eleitores_no_perfil
            LÊ E ESCREVE;BA;FEMININO;70 A 79 ANOS;495
        </inputData>
        <sourceId>90233</sourceId>
        <sourceTable>perfil_eleitorado</sourceTable>
        <command>perfil_eleitorado.jar</command>
    </activity>
</pipeline>
```

Figura 4. Um arquivo de Instância

Uma instância pode conter instruções para executar vários wrappers em sequência. Ao receber uma instância, o Teapot identifica o nome do wrapper que deve ser executado (tags “command” e “executor”) e cria uma pasta exclusiva para esta execução, salvando os dados de entrada (tag “inputData”) em um arquivo chamado “sagi_input.txt”. Embaixo desta pasta também são criadas mais duas pastas (inbox e outbox), a pasta “inbox” serve para que o Teapot grave os arquivos necessários à execução da atividade. Já o uso da pasta “outbox” é imperativo quando o desenvolvedor do wrapper necessita enviar os arquivos produzidos pela atividade para o Sagitarii, pois ao encerrar a execução do wrapper, o Teapot avalia a pasta “outbox” e envia os arquivos ali contidos ao servidor, devidamente identificados.

Após este trabalho inicial, o wrapper é executado como um “thread” interno do Teapot. A quantidade de wrappers que são executados ao mesmo tempo é definida pelo arquivo de configuração do Teapot.

Parâmetros passados pelo Teapot Cluster para o wrapper

Ao iniciar a execução do “thread” do wrapper, o Teapot passa algumas informações como parâmetro (tabela 1).

Tabela 1. Parâmetros passados ao wrapper pelo Teapot Node

Argumento 0	Caminho da pasta exclusiva criada pelo Teapot cluster para a execução do wrapper. O arquivo sagi_input.txt estará nesta pasta e as pastas “inbox” e “outbox” estarão abaixo dela. O Teapot também gravará uma cópia do arquivo XML do pipeline nesta pasta.

Como receber e enviar arquivos e dados CSV

Após iniciar a execução, o wrapper deverá carregar os dados que recebeu do Teapot, enviados pelo Sagitarii, abrindo o arquivo indicado pelo primeiro parâmetro. Este arquivo está em formato CSV. A forma como este arquivo será carregado e utilizado dependerá da implementação do desenvolvedor do wrapper, que deverá respeitar o tipo de wrapper que está escrevendo.

Caso o wrapper necessite de algum arquivo (este já deverá estar devidamente armazenado no Sagitarii, seja por carga inicial ou por produto de outra atividade), deverá acessar o endereço da API de download de arquivo do Sagitarii passando o nome do arquivo desejado (figura 5). O quarto parâmetro

contêm o endereço do Sagitarii, bastando então adicionar o nome da API, o nome do arquivo desejado e o identificador do experimento (quinto parâmetro).

Ao encerrar a tarefa, o wrapper deverá enviar os dados produzidos ao Sagitarii. Para padronizar o trâmite de dados, o wrapper nunca envia dados diretamente ao Sagitarii, apenas com a intervenção do Teapot Cluster.

Os dados da atividade que deverão ser gravados nas tabelas de saída deverão ser enviados diretamente para a saída padrão (console, tela), pois o wrapper está sendo executado como um thread do Teapot. O Teapot receberá estes dados e enviará de forma adequada ao servidor. Os arquivos que precisam ser enviados ao servidor deverão ser gravados na pasta "outbox". Ao encerrar o thread do wrapper, o Teapot verificará esta pasta e enviará todos os arquivos ali contidos ao servidor Sagitarii.

```
110    /**
111     * Download files from Sagitarii if you need.
112     */
113    public static void downloadFile(String fileName) throws Exception {
114        // Sagitarii server host address is always args[3]
115        // Remember, the activation working folder is always args[1].
116        // Teapot also gives you an inbox folder at args[1]/inbox so you can put
117        // your files there.
118
119        URL website = new URL( hostUrl + "/getFile?fileName="+ fileName + "&experiment=" + experimentSerial );
120        String destination = workFolder + "/inbox/" + fileName;
121        ReadableByteChannel rbc = Channels.newChannel(website.openStream());
122        FileOutputStream fos = new FileOutputStream( destination );
123        fos.getChannel().transferFrom(rbc, 0, Long.MAX_VALUE);
124        fos.close();
125    }
```

Figura 5. Baixando um arquivo do servidor Sagitarii

Caso seja necessário enviar arquivos ao servidor, normalmente os nomes dos arquivos são enviados como dados CSV, para que sejam encontrados pela próxima tarefa que receber estes dados, mas esta é uma decisão do desenvolvedor do wrapper. O importante é adicionar, como prefixo no nome do arquivo, o identificador único da tarefa (terceiro argumento) para garantir que este será mantido único. Não se deve esquecer que várias instâncias do wrapper executarão o mesmo trabalho e, apesar de poderem gravar arquivos com o mesmo nome (possuem pastas de trabalho diferentes), eles não podem enviar arquivos com mesmo nome ao Sagitarii dentro do mesmo experimento.

1. Criando um wrapper tipo MAP

Os wrappers possuem várias características em comum, então é natural que possuam trechos de código em comum. O código de um wrapper tipo MAP e tudo o que for comum entre os vários tipos de wrappers será mostrado neste capítulo, enquanto os demais se limitarão a mostrar somente o que for específico daquele tipo.

Juntamente com o código fonte do Sagitarii existe uma classe implementando um wrapper de exemplo. Esta classe chama-se "SampleWrapper" e está no pacote "cmabreu.sagitarii.wrappers".

Todo wrapper deve estar preparado para receber os parâmetros que serão passados pelo Teapot, então é conveniente preparar alguns atributos privados para facilitar a sua utilização. A figura 6 ilustra a preparação destes atributos.

```
public class MyMAPWrapper{
    // To hold parameters...
    private static String inputFile;        // args[0]
    private static String workFolder;       // args[1]
    private static String taskId;           // args[2]
    private static String hostUrl;          // args[3]
    private static String experimentSerial; // args[4]
```

Figura 6. Criação dos atributos do wrapper para receber os parâmetros

Na inicialização do wrapper (método *main*), estes atributos receberão os valores dos parâmetros passados pelo Teapot. Recomenda-se manter uma padronização na criação dos wrappers, então, mesmo

que nem todos os parâmetros sejam utilizados, é conveniente receber todos de qualquer forma. A figura 7 mostra como receber os parâmetros do Teapot nos atributos criados.

Se for necessário receber alguma informação que não foi passada por parâmetro, é sempre possível ao programador que está criando o wrapper ler o arquivo XML de pipeline. Este arquivo é gravado pelo Teapot na pasta exclusiva da ativação (parâmetro "workFolder") e pode ser acessado usando o caminho

<workFolder>/sagi_source_data.xml

Neste caso, o programador que desenvolve o wrapper fica encarregado de criar um método de acesso e interpretação do XML de pipeline (parser).

```
public static void main(String[] args) throws Exception {
    inputFile = args[0];        // CSV input data file
    workFolder = args[1];       // Working folder
    taskId = args[2];           // Task serial ID
    hostUrl = args[3];          // Sagitarii host URL
    experimentSerial = args[4]; // Experiment tag ID
```

Figura 7. Recebendo os parâmetros passados pelo Teapot Cluster

O próximo passo é carregar os dados CSV que o Teapot gravou na pasta exclusiva de execução da ativação. Este arquivo (*sagi_input.txt*) contém os dados CSV que o Sagitarii enviou ao Teapot utilizando o arquivo XML de pipeline (figura 4). O Teapot, após processar o pipeline, separa os dados CSV e grava neste arquivo antes de iniciar uma instância do wrapper que irá processá-lo. Lembre-se de que cada instância de execução do wrapper (ativação) possui uma pasta exclusiva para trabalhar com arquivos, que é apagada após a execução do wrapper. O caminho completo e o nome do arquivo é entregue pelo Teapot no parâmetro args[0] (atributo *inputFile*). É conveniente criar um método separado para carregar este arquivo em uma lista de *strings* (figura 8).

```
public static List<String> readFile(String file) throws Exception {
    String line = "";
    ArrayList<String> list = new ArrayList<String>();
    BufferedReader br = new BufferedReader( new FileReader( file ) );
    if (br != null) {
        while ( (line = br.readLine() ) != null ) {
            list.add( line );
        }
        br.close();
    }
    return list;
}
```

Figura 8. Método para ler o arquivo CSV em uma lista de *Strings*

O método recebe um nome de arquivo (com caminho completo) e devolve uma lista de Strings contendo o conteúdo do arquivo (linhas CSV).

Retornando ao método de inicialização do wrapper (main), é hora de carregar o arquivo CSV usando o parâmetro passado pelo Teapot (*inputFile*). A figura 9 ilustra este processo.

```
    // Read the CSV input data file
    List<String> inputData = readFile( inputFile );
    // If we have data...
    if( inputData.size() > 0 ) {
        // we will work only if we have data!
    }
```

Figura 9. Carregando os dados CSV enviados pelo Teapot Cluster

Uma vez que se possui os dados enviados pelo Sagitarii, o seu processamento depende de cada

wrapper e a implementação deste processamento depende de cada programador e da tarefa que foi atribuída ao wrapper. Um wrapper tipo MAP vai receber um CSV contendo o cabeçalho com os nomes das colunas e uma linha com os respectivos valores e deverá devolver um CSV com o mesmo formato (um cabeçalho com os nomes das colunas e uma linha com os respectivos valores).

É imperativo que os nomes das colunas do CSV produzido esteja de acordo com os nomes das colunas da tabela de saída da atividade a que pertence o wrapper, sendo que a ordem das colunas é irrelevante. As colunas que existirem no CSV e não existirem na tabela de saída da atividade serão ignoradas e as colunas que existirem na tabela de saída da atividade e não existirem no CSV receberão valores nulos.

Para facilitar o processamento do arquivo CSV, é recomendado criar dois métodos para processar as colunas (cabeçalho) e a linha de dados (valores). A figura 10 ilustra a chamada ao processamento do cabeçalho do CSV (modificando o código da figura 9). Processar o cabeçalho fará sentido quando for necessário repassar ao CSV de saída (e por consequência a tabela de saída da atividade) os nomes das colunas do CSV de entrada (e por consequência, da tabela de entrada da atividade). Normalmente isso é feito quando é necessário manter a rastreabilidade dos dados (quais os valores de entrada produziram este valor de saída). Para isso, basta replicar as colunas desejadas no CSV de saída, e, é claro, seus respectivos valores.

```
// If we have data...
if( inputData.size() > 0 ) {
    // Get the first line : the columns
    String header = inputData.get(0);
    String[] columns = header.split(";");
    // Do something with the columns
    processHeader( columns );
    ...
}
```

Figura 10. Chamando o método de processamento das colunas CSV

A parte mais relevante é o processamento das linhas de dados. Cabe ao programador decidir como trabalhar estas informações de acordo com a tarefa a ser executada pelo wrapper. No caso de um wrapper tipo MAP, os valores resultantes formarão uma linha no CSV de saída e devem estar de acordo com os tipos de dados esperados na tabela de saída da atividade. A figura 11 mostra mais uma alteração no código das figuras 9 e 10, desta vez chamando um método para processar a linha contando os dados em CSV, recebidos do Teapot. Ainda no método de inicialização do wrapper (*main*). Ainda com foco na reutilização e padronização, é conveniente tratar estes dados como se possuíssem mais de uma linha, ainda que os dados provenientes de atividades tipo MAP possuam apenas uma única linha. Na figura 11, um laço tipo "*for*" processa todas as linhas do arquivo CSV, neste caso, apenas uma única linha será processada. Quando for a hora de criar wrappers de outros tipos, este método de inicialização do wrapper (*main*) estará flexível o suficiente para ter um mínimo de modificações, bastando copiar e colar o código no novo wrapper.

```
// If we have data...
if( inputData.size() > 0 ) {
    // Get the first line : the columns
    String header = inputData.get(0);
    String[] columns = header.split(";");
    // Do something with the columns
    processColumns( columns );
    // If we have lines of data
    if( inputData.size() > 1 ) {
        // With each line of data...
        for( int x=1; x<inputData.size(); x++ ) {
            String[] lineData = inputData.get(x).split(";");
            // ... do something
            processLine( lineData );
        }
    }
}
```

Figura 11. Processando as linhas do arquivo CSV

Um wrapper vai produzir, invariavelmente, um conjunto de dados no formato CSV e/ou arquivos a serem enviados ao Sagitarii. Embora não seja uma regra, os dados no formato CSV normalmente são tratados como dados de fluxo do experimento (são obtidos de tabelas e serão armazenados em tabelas durante o processamento do experimento). Já os arquivos normalmente são tratados como dados de resultado, pertencentes ao experimento, e serão gravados em local apropriado para serem acessados pelo responsável pelo experimento.

Um wrapper entrega os dados CSV ao Teapot (para ser enviado ao Sagitarii) através da saída padrão do sistema operacional (neste caso, a tela). Isto significa que um wrapper deverá literalmente "escrever" na tela o conteúdo CSV que for produzido.

Por conveniência, é recomendado criar um atributo privado e global para o wrapper manter os dados que forem produzidos para a saída. A figura 12 ilustra uma modificação do código da figura 6 (criação de atributos globais para o wrapper) contendo a declaração deste atributo.

```
public class MyMAPWrapper {
    // To hold parameters...
    private static String inputFile;        // args[0]
    private static String workFolder;       // args[1]
    private static String taskId;           // args[2]
    private static String hostUrl;          // args[3]
    private static String experimentSerial; // args[4]
    // The output CSV data holder ( index 0 = header )
    private static List<String> outputData = new ArrayList<String>();
```

Figura 12. Um atributo para manter os dados CSV de saída

A medida que os dados de saída forem produzidos, estes deverão ser armazenados no atributo "*outputData*", que é uma lista de *Strings*, sendo que a primeira linha desta lista deverá ser a lista de colunas do CSV, separadas por ponto e vírgula, e as demais os dados, também separados por ponto e vírgula.

Uma vez produzidos os dados, é hora de enviá-los ao Teapot, para que este os envie ao Sagitarii. Para tanto, é necessário "escrever" o conteúdo do atributo "*outputData*" na tela. A figura 13 mostra a sugestão para um método que faz este trabalho.

```
    /**
     * Print out the outputData content to screen
     * ( send to Teapot by using standard out )
     */
    private static void printOutput() {
        for ( String line : outputData  ) {
            System.out.println( line );
        }
    }
```

Figura 13. Método para enviar os dados CSV de saída ao Teapot

Mais uma vez vamos modificar o código do método de inicialização do wrapper (*main*) para chamar o método de envio dos dados de saída, após processar as colunas e as linhas (no caso do MAP, uma única linha). A figura 14 mostra a alteração no código da figura 11.

```
// If we have data...
if( inputData.size() > 0 ) {
    // Get the first line : the columns
    String header = inputData.get(0);
    String[] columns = header.split(";");
    // Do something with the columns
    processColumns( columns );
    // If we have lines of data
    if( inputData.size() > 1 ) {
        // With each line of data...
        for( int x=1; x<inputData.size(); x++ ) {
            String[] lineData = inputData.get(x).split(";");
            // ... do something
            processLine( lineData );
        }
    }
    // Print out the outputData content to screen ( send to Teapot by using standard out )
    printOutput();

}
```

Figura 14. Método "main" : processando colunas, linhas e enviando a saída

Como o atributo "*outputData*" é global, os métodos de processamento de colunas e linhas podem gravar dados diretamente nele, tendo o cuidado de inserir primeiro as colunas (chamar "processColumns") e depois as linhas (chamar "processLine").

Vale ressaltar que os métodos "processColumns" e o método "processLine" devem ser implementados de acordo com a necessidade e exigência da tarefa a ser executada pelo wrapper, sendo esta a razão por não terem sido exibidos neste guia. O método "processColumns", por exemplo, pode nem ser necessário, já que as colunas a serem produzidas são de prévio conhecimento do programador (são as mesmas da estrutura da tabela de saída da atividade que executou o wrapper). Neste caso, a primeira linha do atributo "*outputData*" (lista de "*Strings*"), que representa as colunas do CSV, pode ser produzida "manualmente" por concatenação, já no início do método "*main*".

É importante observar a ordem dos dados produzidos e suas respectivas colunas no atributo de saída. Quando os dados forem produzidos no método "processLine", estes devem ser concatenados na mesma ordem em que as colunas foram criadas. Por exemplo, supondo que a primeira linha do atributo "*outputData*" contenha as seguintes colunas CSV:

nome;rua;idade

Quando o método "processLine" receber os dados de entrada, processá-los e for concatenar a linha de saída para inserir no atributo "*outputData*", deverá colocar os dados na sequência:

Carlos Abreu;Rua Jardim;42

Respeitando assim a ordem das colunas, o que é importante para permitir que o Sagitarii insira corretamente os dados na tabela de saída da atividade. Note que a tabela de saída, no exemplo, espera dados nos tipos String para o campo "nome", String para o campo "rua" e Integer para o campo "idade". Inverter a ordem destes campos fará com que a inserção dos dados gere um erro de SQL no Sagitarii, o que causará a perda dos dados e por consequência, uma possível interrupção na execução do experimento. Na melhor das hipóteses (dados invertidos com mesma tipagem) o experimento irá fornecer um resultado incorreto.

Por conveniência, como um wrapper do tipo MAP recebe uma linha de dados e produz uma linha de dados, não será considerado neste capítulo o recebimento e envio de arquivos. O recebimento de arquivos será mostrado no capítulo sobre wrappers do tipo SPLIT e o envio de arquivos será coberto no capítulo sobre wrappers do tipo REDUCE.

Para testar o wrapper, crie um arquivo CSV com dados de teste e execute o wrapper passando os parâmetros esperados na ordem exigida (neste caso, forneça o seu arquivo e crie parâmetros falsos para os

demais). O wrapper deverá escrever na tela um CSV no formato correto: primeira linha para as colunas CSV e demais linhas para os dados, separados por ponto e vírgula e sem separação por aspas. Cada nova linha usando o caractere “NEWLINE” (/n). Além disso, é imperativo poder executar o wrapper usando o mesmo formato de chamada que o Teapot usa (sem fornecer o *classpath* ou outros parâmetros que não os já mencionados):

java -jar MyMAPWrapper.jar /home/user/mycsvfile.csv aa bb cc dd

Tendo o wrapper funcionado corretamente, ele já poderá ser cadastrado no Sagitarii para ser associado à uma atividade.

2. Criando um wrapper tipo SPLIT

Um wrapper tipo SPLIT receberá um CSV com uma linha de dados e deverá produzir um CSV com uma ou mais linhas de dados e/ou um ou mais arquivos a serem enviados ao Sagitarii. Caso o arquivo de dados CSV contenha nomes de arquivos a serem processados, estes deverão ser baixados do Sagitarii.

Em primeiro lugar, será coberto o caso mais simples: o recebimento de um CSV contendo uma linha de dados e o envio de um CSV contendo uma ou mais linhas de dados.

Neste caso, a única modificação em relação ao exposto no capítulo anterior é que a lista de Strings contendo os dados de saída (atributo “*outputData*”) conterá mais de uma linha de dados. Já deixamos o método “printOutput”, que envia o conteúdo de “*outputData*” para o Teapot, pronto para enfrentar esta situação (figura 13).

Um outro caso seria a utilização de algum arquivo para processar (descompactar, por exemplo). Neste caso, o nome deste arquivo viria no CSV de entrada e seria necessário baixá-lo do servidor Sagitarii. Este arquivo deverá estar armazenado no Sagitarii seguindo dois caminhos: a) foi enviado pelo responsável pelo experimento, fazendo parte dos dados iniciais (consulte o Guia do Usuário para saber como enviar arquivos ao Sagitarii); b) foi enviado por uma das atividades anteriores. Nesse caso, consulte o capítulo sobre criação de wrappers tipo “REDUCE” para saber como um wrapper envia arquivos para o Sagitarii. O Sagitarii armazena os arquivos no escopo do experimento, o que significa que cada experimento deve possuir arquivos com nomes únicos. Uma maneira de garantir que os arquivos sejam únicos é adicionar o número de série da tarefa ao nome do arquivo. O número de série da tarefa é passado ao wrapper pelo Teapot usando o terceiro parâmetro.

A não existência do arquivo solicitado causará erro na execução da atividade (representada pela execução de todas as suas ativações) e por consequência, do experimento.

Supondo que o método “processLine”, descrito no capítulo anterior, identifique o nome do arquivo necessário a ser utilizado pelo wrapper, o próximo passo é acessar a API de fornecimento de arquivos do Sagitarii. Esta API responde no endereço:

http://<sagitariihost>/getFile?fileName=<file_name>&experiment=<exp_id>

O endereço do servidor Sagitarii já é fornecido pelo Teapot através do quarto parâmetro, que foi armazenado no atributo “hostUrl” e o número de série do experimento foi passado pelo quinto parâmetro e armazenado no atributo “experimentSerial” (figura 7). A figura 15 ilustra um método para realizar o download de um arquivo do Sagitarii, que podemos acrescentar à classe do wrapper que já foi criado no capítulo anterior ou criar um novo wrapper copiando o código do primeiro.

```
/**
 * Download files from Sagitarii if you need.
 */
public static void downloadFile(String fileName) throws Exception {
    // Sagitarii server host address is always args[3]
    // Remember, the activation working folder is always args[1].
    // So, you must use args[1]/<file_name> to save your downloaded file.

    URL website = new URL( hostUrl + "/getFile?fileName="+ fileName + "&experiment=" + experimentSerial );
    String destination = workFolder + "/inbox/" + fileName;
    ReadableByteChannel rbc = Channels.newChannel(website.openStream());
    FileOutputStream fos = new FileOutputStream( destination );
    fos.getChannel().transferFrom(rbc, 0, Long.MAX_VALUE);
    fos.close();
}
```

Figura 15. Método para baixar arquivos do Sagitarii

Observando o método na figura 15 percebe-se a utilização do atributo “workFolder”, que representa o segundo argumento passado ao wrapper pelo Teapot, contendo o caminho completo da pasta exclusiva criada para a execução do wrapper. O Teapot, por conveniência, já fornece a pasta “inbox” abaixo da pasta da ativação (“workFolder”), então ficará mais prático que os arquivos baixados do Sagitarii sejam gravados nesta pasta, pois ao término da execução da instância do wrapper, tudo será apagado.

Uma vez de posse do arquivo, o wrapper poderá processar e enviar o resultado de volta ao Sagitarii, seja por dados CSV (como mostrado no capítulo anterior) ou enviando arquivos (como será mostrado no próximo capítulo). A figura 16 ilustra um exemplo do método “processLine”, que recebe uma linha de dados do arquivo CSV, identifica o nome do arquivo, faz o download chamando o método da figura 15 e chama um suposto método de descompactação “splitFile” (considerando que o arquivo em questão é um arquivo compactado).

```
public static void processLine( String[] lineData ) {
    // get the filename as first element in the list
    String fileName = lineData[0];
    try {
        // download it from Sagitarii
        downloadFile( fileName );
        // uncompress contents in outbox folder
        // (will send contents back to Sagitarii !)
        splitFile( fileName, workFolder + "/outbox");
    } catch ( Exception e ) {
        e.printStackTrace();
        // Exit code will remain in table data
        // so we will know something was wrong
        System.exit(1);
    }
}
```

Figura 16. Exemplo de processamento de uma linha que envolve um download

Na figura 16 pode-se perceber a chamada ao método “splitFile” passando como parâmetro a pasta exclusiva seguida de “outbox”, que é a pasta de saída do wrapper. Qualquer arquivo existente nesta pasta após o encerramento do wrapper será enviado ao Sagitarii dentro do escopo do experimento, então tudo que este exemplo faz é baixar um arquivo compactado, descompactar e enviar seu conteúdo de volta ao Sagitarii.

A figura 17 mostra mais uma alteração no método “main” (exibido pela última vez na figura 14). Desta vez, a inclusão das colunas CSV (linha 0 da lista de saída “outputData”) é feita “manualmente”, informando que a saída será um CSV contendo o nome do arquivo compactado e seu respectivo conteúdo (obviamente o nome do arquivo compactado irá se repetir para todas as linhas, mas isso fará sentido quando todos os dados estiverem juntos e for necessário saber de conde veio cada arquivo). Ainda no método “main” (figura 17), nota-se a chamada ao método “processLine” para cada linha do CSV de entrada (provavelmente apenas uma no tipo SPLIT). Em “processLine” (figura 16) baixamos o arquivo que identificamos como o elemento zero na linha CSV processada e descompactamos seu conteúdo. O apêndice “B” mostra o código do método “splitFile”, onde pode-se notar a inclusão das linhas de dados propriamente ditas na lista de saída “outputData”, respeitando a ordem das colunas, já inseridas na lista (figura 17). Então o método “main” encerra enviando todo o conteúdo de “outputData” para o Teapot (figura 13), que o envia ao Sagitarii.

```
// if we have data...
if( inputData.size() > 0 ) {
    // prepare output CSV header
    // ( must match activity destination table schema in Sagitarii )
    outputData.add("zipfile;contentfile");

    // If we have lines of data
    if( inputData.size() > 1 ) {
        // With each line of data...
        for( int x=1; x<inputData.size(); x++ ) {
            String[] lineData = inputData.get(x).split(";");
            // ... do something
            processLine( lineData );
        }
    }
    // Print out the outputData content to screen
    // ( send to Teapot by using standard out )
    printOutput();
}
```

Figura 17. Criação manual do cabeçalho do CSV de saída

3. Criando um wrapper tipo REDUCE

Em eleboração.

4. Criando um wrapper tipo RRUNNER

Em eleboração.

5. Cadastrando arquivos de suporte (bibliotecas) para os wrappers.

Em eleboração.

6. Cadastrando o wrapper no Sagitarii

Apêndice “A”: Tabela descritiva do arquivo de instância

pipeline	Elemento raiz do documento XML. Possui os atributos “workflow”, “experiment”, “serial” e “fragment”. Cardinalidade: um.
activity	Contém os dados para uma ativação. Elementos filhos: “order, “serial”, “executorType”, “taskIdChain”, “executor”, “type”, “inputData”, “sourceId”, “sourceTable” e “command”. Sem atributos. Filho de “pipeline”. Cardinalidade: um ou muitos.
order	Número inteiro contendo a ordem de execução da atividade. Sem atributos. Filho de “activity”. Cardinalidade: um.
serial	Alfanumérico representando o número de série da atividade. Filho de “activity”. Sem atributos. Cardinalidade: um.
executorType	Tipo do executor que processará os dados. Enumerado. Filho de “activity”. Sem atributos. Cardinalidade: um.
taskIdChain	Alfanumérico que representa o ID da primeira tarefa que foi executada antes desta. O ID da tarefa é repassado entre os pipelines para manter o rastreamento do processamento. Sem atributos. Filho de “activity”. Cardinalidade: um.
executor	Alfanumérico com o nome (alias) do executor como cadastrado no Sagitarii. Sem atributos. Filho de “activity”. Cardinalidade: um.
type	Enumerado. Tipo da atividade associada a este pipeline. Sem atributos. Filho de “activity”. Cardinalidade: um.
inputData	CSV. Dados a serem repassados ao wrapper para processamento. Sem atributos. Filho de “activity”. Cardinalidade: um.
sourceId	Número inteiro (ou vazio quando for REDUCE). ID da tupla (linha) de origem dos dados de entrada contidos na tag “inputData”, na tabela de entrada da atividade. Sem atributos. Filho de “activity”. Cardinalidade: um.
sourceTable	Alfanumérico com o nome da tabela de entrada da atividade de onde os dados contidos na tag “inputData” foram retirados. Para saber a origem destes dados, executar “*select * from <sourceTable> where id_custom = <sourceId>*”. Sem atributos. Filho de “activity”. Cardinalidade: um.
command	Alfanumérico com o nome do arquivo java do wrapper que processará os dados. Sem atributos. Filho de “activity”. Cardinalidade: um.

Apêndice “B”: Método splitFile()

```java
/**
 * Uncompress a ZIP file
 *
 * @param zipFile
 * @param outputFolder
 */
public static void splitFile( String zipFile, String outputFolder ) {
    byte[] buffer = new byte[1024];
    try {
        File folder = new File( outputFolder );
        if( !folder.exists() ){
            folder.mkdirs();
        }
        ZipInputStream zis = new ZipInputStream(new FileInputStream( workFolder + "/inbox/" + zipFile ) );
        ZipEntry ze = zis.getNextEntry();
        while(ze!=null) {
            String fileName = ze.getName();
            outputData.add(zipFile + ";" + fileName);
            File newFile = new File(outputFolder + File.separator + fileName);
            new File(newFile.getParent()).mkdirs();
            FileOutputStream fos = new FileOutputStream(newFile);
            int len;
            while ((len = zis.read(buffer)) > 0) {
                fos.write(buffer, 0, len);
            }
            fos.close();
            ze = zis.getNextEntry();
        }
        zis.closeEntry();
        zis.close();
    } catch ( IOException ex ){
        ex.printStackTrace();
    }
}
```

API de Acesso externo

O Sagitarii oferece uma API para acesso externo de outros programas. A comunicação é feita através de

requisições POST usando os dados formatados em JSON. A resposta do Sagitarii será uma String no formato JSON com os dados requisitados ou com um código de retorno. É necessário requisitar um token de segurança antes de qualquer operação.

URL da API	/externalAPI
Tipo de acesso	POST (formulário HTML)
Nome Atributo / Tipo	externalForm / String
Valor	Estrutura JSON no formato { “chave1” : ”valor1”, “chaven” : ”valorn” }
Retorno	String

```
<html>
	<form action='http://www.sagitarii.server.org/externalAPI'>
	<input type='text' name='externalForm'
value='{"SagitariiApiFunction":"apiGetToken","user":"john","password":"doo"}'>
	</form>
<html>
```

1. **apiGetToken**

   Autentica na API e recebe um token de segurança para ser usado nas demais operações da mesma seção. O token é descartado após 30 minutos de inatividade.

Atributo	Valor
SagitariiApiFunction	apiGetToken
user	Usuário do Sagitarii
password	Senha do usuário

```
{
	"SagitariiApiFunction":"apiGetToken",
	"user":"john",
	"password":"1234"
}
```

Retorno: uma String contendo o token de segurança a ser utilizado nas próximas chamadas.

2. **apiCreateExperiment**

   Cria um novo experimento a partir de um workflow já existente.

Atributo	Valor
SagitariiApiFunction	apiCreateExperiment
workflowTag	Tag do workflow existente

securityToken	Token de segurança

```
{
        "SagitariiApiFunction":"apiCreateExperiment",
        "workflowTag":"SPECTRAL_PORTAL",
        "securityToken":"b4abc9870b2b41bd91e16250cdc76465"
}
```

Retorno: uma String contendo o número de série (identificador) do experimento criado.

3. apiStartExperiment

Executa um experimento já existente.

Atributo	Valor
SagitariiApiFunction	apiStartExperiment
experimentSerial	Número de série do experimento
securityToken	Token de segurança

```
{
        "SagitariiApiFunction":"apiStartExperiment",
        "experimentSerial":"A39DC069-1726-482",
        "securityToken":"b4abc9870b2b41bd91e16250cdc76465"
}
```

Retorno: uma String contendo o número de série (identificador) do experimento.

4. apiReceiveData

Insere dados em uma tabela no contexto de um experimento.

Atributo	Valor
SagitariiApiFunction	apiReceiveData
experimentSerial	Número de série do experimento
securityToken	Token de segurança
tableName	Nome da tabela

data	Array de dados no formato JSON

```
{
	"SagitariiApiFunction":"apiReceiveData",
	"tableName":"spectral_parameters",
	"experimentSerial":"A39DC069-1726-482",
	"securityToken":"b4abc9870b2b41bd91e16250cdc76465",
	"data":[{
		"adjacency":"on",
		"laplacian":"on",
		"slaplacian":"on",
		"optiFunc":"\lambda",
		"caixa1":"min",
		"gorder":"2",
		"minDegree":"4",
		"maxDegree":"7",
		"triangleFree":"on",
		"allowDiscGraphs":"on",
		"biptOnly":"on"
		}, { … }, { … }, { … }]
}
```

Retorno: uma String contendo o número de série da instância de carga gerada e o número de série da atividade de carga gerada, separadas por ponto e vírgula (;). Os valores devem ser enviados entre aspas, mas devem possuir o formato (tipo) esperado pelo atributo de mesmo nome na tabela de destino. No exemplo acima, “minDegree” deverá ser do tipo “Integer” na tabela “spectral_parameters” e “adjacency” deverá ser do tipo “String”.

5. apiGetFilesExperiment

Retorna arquivos produzidos por um experimento. É necessário informar a tag da atividade que produziu os arquivos, bem como o registro inicial e o final (é obrigatório paginar o resultado). O Sagitarii enviará os nomes e índices dos arquivos já gravados pelo experimento mesmo durante a execução do mesmo.

Atributo	Valor
SagitariiApiFunction	apiGetFilesExperiment
experimentSerial	Número de série do experimento
activityTag	Tag da atividade produtora
rangeStart	Paginação: início
rangeEnd	Paginação: fim
securityToken	Token de segurança


```
{
        "SagitariiApiFunction":"apiGetFilesExperiment",
        "experimentSerial":"A39DC069-1726-482",
        "activityTag":"GENFORMULA",
        "rangeStart":"90",
        "rangeEnd":"100",
        "securityToken":"b4abc9870b2b41bd91e16250cdc76465"
}
```

Retorno: uma String no formato JSON contendo a lista de todos os arquivos produzidos até o momento pela atividade e experimento indicados. A paginação é obrigatória.

```
{
        "data":[
           {
              "fileName":"MYPDFWORK.PDF",
              "fileId":"123"
           },
           {
              "fileName":"WORKDATA.DAT",
              "fileId":"1237"
           }, { … }, { … }, { … }
        ]
}
```

O Sagitarii oferece uma URL para download dos arquivos armazenados em seu repositório, bastando fornecer o ID do arquivo.

```
http://sagitarii.server.org/getFile?fileId=1234
```

Os arquivos armazenados no repositório do Sagitarii estão no formato GZIP, mesmo sendo mantida a extensão original, sendo necessário descompactá-los antes se usar.

6. apiCreateTable

Cria uma tabela no Sagitarii.

Atributo	Valor
SagitariiApiFunction	securityToken
tableName	Nome da tabela a ser criada
securityToken	Token de segurança

tableDescription	Descrição da tabela a ser criada
Nome do atributo 1	Tipo do atributo 1
Nome do atributo n	Tipo do atributo n

```
{
        "SagitariiApiFunction":"apiCreateTable",
        "tableName":"my_new_table",
        "securityToken":"b4abc9870b2b41bd91e16250cdc76465",
        "tableDescription":"A new table",
        "my_field_integer":"INTEGER",
        "my_field_string":"STRING",
        "my_field_float":"FLOAT",
        "my_field_date":"DATE",
        "my_field_time":"TIME",
        "my_field_file":"FILE",
        "my_field_stext":"TEXT"
}
```

Retorno: String contendo o código de retorno.

7. apiGetExperiments

Retorna todos os experimentos de um usuário.

Atributo	Valor
SagitariiApiFunction	apiGetExperiments
securityToken	Token de segurança

```
{
        "SagitariiApiFunction":"apiCreateTable",
        "securityToken":"b4abc9870b2b41bd91e16250cdc76465"
}
```

Retorno: String no formato JSON contendo os experimentos do usuário que solicitou o token de segurança, seu status e a data de início da execução.

```
{
        "data":[
           {
```

```
            "tagExec":"A39DC069-1726-482",
            "startDate":"21/10/2014 23:54",
            "status":"RUNNING"
         },
         {
            "tagExec":"C51AF452-4690-354",
            "startDate":"21/10/2014 12:30",
            "status":"FINISHED"
         }, { … }, { … }, { … }
      ]
}
```