Joao Pedro Schmitt

SOLR Custom Data Import Handler (Customizing the JDBC API to support REST services)

schmittjoaopedro . computer science, Data Mining, Java 1 01+00:00 Janeiro 01+00:00 2019 10 Minutes

This project aims to explain how to extend SOLR to create a custom SOLR data import handler. Remember that if your necessities are well supplied by the available data import handlers of SOLR, make their use before to think about creating a custom indexer. A custom data import handler is a good choice when we want to index JSON objects keeping the SOLR project simplicity. In these cases, we remove the necessity of third-party applications to manage custom data-extraction and indexation by means of REST calls to the SOLR API. To create a custom indexer the process is very straightforward due to the fact that the SOLR projects is very extensible. Following will be presented the details about the implementation.

(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer#solr-configuration)SOLR Configuration

Download the latest version (in this example the version used was 7.6.0) from the official SOLR link (http://www.apache.org/dyn/closer.lua/lucene/solr/7.6.0/solr-7.6.0.tgz). After that, extract the binary to some folder from your file system, e.g. *C:\java\solr-7.6.0*. Then start the SOLR using the following command line:

```
cd C:\java\solr-7.6.0\bin
solr start
```

To create a new SOLR core execute the following command:

```
solr create_core -c solr-custom-data-indexer-core
```

We need to create a lib folder. This folder will hold all dependencies necessary to create our custom SOLR indexer.

```
cd server\solr\solr-custom-data-indexer-core
mkdir lib
```

Download and copy the data import handler jar solr-dataimporterhandler-7.6.0 (https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/blob/master/custom-solr-data-indexer/solr-configuration/lib/solr-dataimporthandler-7.6.0.jar) to the lib folder. This jar is available in the *dist*folder of SOLR project. The classes inside this jar are used by our custom indexer to extend the SOLR API.

(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer#configuring-managed-schema)Configuring Managed Schema

As we are simulating here an example scenario, the data is not real (company and employee). The data used here is available on the assets (https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/blob/master/custom-solr-data-indexer/assets)folder of this GIT (https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer) repo. The dataset used here contains a relationship between employees with departments (given the foreign key *companyId* field). To persist this data in our SOLR core, first is necessary to configure the *managed-schema* file, therefore add the following entries at the *managed-schema* file after the id entry.

```
...
<field name="id" type="string" indexed="true" stored="true" required="true" r
<field name="firstName" type="string" indexed="true" stored="true" multiValue
<field name="lastName" type="string" indexed="true" stored="true" multiValued
<field name="age" type="pint" indexed="true" stored="true" multiValued="false
<field name="companyId" type="pint" indexed="true" stored="true" multiValued=
<field name="companyVersion" type="pint" indexed="true" stored="true" multiVa
<field name="companyName" type="string" indexed="true" stored="true" multiVal
...
```

(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer#configuring-solrconfigxml)Configuring solrconfig.xml

In *solrconfig.xml* we need to add a new entry to determine the dataimport handler support. Add the following entry to the file:

```
...
<requestHandler name="/dataimport" class="org.apache.solr.handler.dataimport.
    <lst name="defaults">
        <str name="config">data-config.xml</str>
    </lst>
</requestHandler>
...
```

After that, create the *data-config.xml* file inside the */conf* folder inside our solr core. The *data-config.xml* file holds the configuration to determine how the data import handler will process the data to be indexed. Our file is customized because we are extending the default engine of SOLR to support indexation with REST services. The processor tag is modified to indicate our custom classes that will handle the data import, differently from the SOLR docs, the datasource configuration is not necessary in this case.

```
<dataConfig>
    <!--
    In this case the JDBC configuration isn't necessary
    because we are creating a custom implementation
    -->
    <document>
        <entity
            name="employee"
            pk="id"
            query="true"
            deltaQuery="true"
            deltaImportQuery="true"
            deletedPkQuery="true"
            processor="com.github.schmittjoaopedro.EmployeeProcessor" >
            <entity
                name="company"
                query="true"
                processor="com.github.schmittjoaopedro.CompanyProcessor">
            </entity>
            <field name="id" column="id" />
            <field name="firstName" column="firstName" />
            <field name="lastName" column="lastName" />
            <field name="age" column="age" />
            <field name="departmentId" column="departmentId" />
            <field name="departmentName" column="departmentName" />
        </entity>
    </document>
</dataConfig>
```

The full SOLR core configuration is available in this link (https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer/solr-custom-data-indexer-core).

(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer#java-project)Java project

One of the easiest ways to create a custom data import handler is using a Maven Java Project.

(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer#maven-configuration)Maven configuration

In this case, create a project and add the following dependencies in the *pom.xml*. All provided dependencies are included for development purposes because they are supplied by SOLR when we start the server. The JUnit is used for tests, and the remaining dependencies are artifacts to allow our custom implementation retrieve data via REST API (jetty client) and process the JSON response (Gson).

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-client</artifactId>
        <version>9.4.11.v20180605</version>
    </dependency>
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.8.5</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.24</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>joda-time</groupId>
        <artifactId>joda-time</artifactId>
        <version>2.2</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-lang3</artifactId>
        <version>3.6</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.solr</groupId>
        <artifactId>solr-core</artifactId>
        <version>7.6.0</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.solr</groupId>
        <artifactId>solr-dataimporthandler</artifactId>
        <version>7.6.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

After we built the jar we need to copy this one to the lib folder created in our SOLR core. As we have external dependencies (Gson and Jetty Client) we need to make a jar with the dependencies. Then add the following plugin to the *pom.xml* file.

```
<plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
    </configuration>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer#data-structure)Data structure

The dataset used to simulate ou custom extension is available in the assets (https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer/assets) link. The assets folder contains the *company_v1_full*and *company_v2_full* files, that simulate the database of companies in version one and two. Besides that, in our *assets* folder, the employee is organized in the same way, where *employee_v1_full* and *employee_v2_full* contains the full database in version one and two, while file *employee_v2_modified* contains the created/updated delta between version one and two and the file *employee_v2_removed* contains the removed entries.

(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer#extending-the-solr-api)Extending the SOLR API

As defined in our *data-config.xml* we are creating two custom processors. In this case, is necessary to create two Java classes that will extend the *EntityProcessorBase* class from SOLR API.
The *EntityProcessorBase* is used by SOLR to identify and prepare the documents to be indexed. In

these classes, the *Map<String, Object>* represents a single document that is saved on SOLR.
The *EmployeeProcessor* class presents our custom implementation class, here we define the methods to find the modified and removed documents.

```
package com.github.schmittjoaopedro;

import org.apache.solr.handler.dataimport.Context;
import org.apache.solr.handler.dataimport.EntityProcessorBase;
import org.joda.time.DateTime;
import org.joda.time.LocalDateTime;
import org.joda.time.format.DateTimeFormat;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.lang.invoke.MethodHandles;
import java.util.Date;
import java.util.List;
import java.util.Map;

public class EmployeeProcessor extends EntityProcessorBase {

    private static final Logger LOG = LoggerFactory.getLogger(MethodHandles.l

    private EmployeeService employeeService;

    @Override
    public void init(Context context) {
        super.init(context);
        // In the context object, we receive a lot of metadata to support
        // our custom import handler operation.
        // For example, here we are storing the SOLR core name (in case
        // of multiples cores this information can be useful to reuse
        // source code).
        employeeService = new EmployeeService(context.getSolrCore().getName()
    }

    // SOLR always call this method to retrieve the next row to be processed.
    // In the first call (when a full or delta import is called) the rowItera
    // is null, then at this time, we query the external service to load the
    // to be processed.
    @Override
    public Map<String, Object> nextRow() {
        if (rowIterator == null) {
            initEmployees(); // Load all employees to be indexed in the SOLR
        }
        return getNext(); // Get next record to be indexed
    }

    // This method is called to identify what entries must be created or upda
    // In this case, a smart query can be used to load only the ID informatic
    // because after that for each entry the nextRow() method will be called
    // to retrieve the full entry to be indexed.
    @Override
    public Map<String, Object> nextModifiedRowKey() {
```

```
    if (rowIterator == null) {
        initModifiedEmployees();
    }
    return getNext();
}

// This method is called after the nextModifiedRowKey() have finished def
// the records to be deleted. Then SOLR can make the difference between m
// records and removed ones to save processing time.
@Override
public Map<String, Object> nextDeletedRowKey() {
    if (rowIterator == null) {
        initDeleteEmployees();
    }
    return getNext();
}

private void initEmployees() {
    Date startDate;
    List<Map<String, Object>> employeesList;
    String id;
    String revision;
    if (Context.FULL_DUMP.equals(context.currentProcess())) { // If the f
        // Define a start date to consider the database, this information
        startDate = new DateTime(1990, 01, 01, 0, 0).toDate();
        employeesList = employeeService.getEmployeesAfterDate(startDate);
        LOG.info("Full import of " + employeesList.size() + " employees");
        rowIterator = employeesList.iterator();
    } else if (Context.DELTA_DUMP.equals(context.currentProcess())) { //
        // Get information about what record we want to update, this meth
        id = String.valueOf(context.getVariableResolver().resolve("dih.de
        revision = String.valueOf(context.getVariableResolver().resolve("
        // We expect one record to be updated
        employeesList = employeeService.getEmployee(id, revision);
        if (employeesList.size() == 1) {
            LOG.info("Delta header of grouper " + employeesList.get(0).ge
            rowIterator = employeesList.iterator();
        }
    }
}

private void initModifiedEmployees() {
    // Select next data to update based on last_index_time information ke
    // After the import was finished, the last_index_time is updated with
    String dateString = context.getVariableResolver().resolve("dih.emplo
    LocalDateTime startDate = LocalDateTime.parse(dateString, DateTimeFor
    List<Map<String, Object>> materialList = employeeService.getDeltaEmpl
    if (!materialList.isEmpty()) {
        LOG.info("Found " + materialList.size() + " modified groupers");
        rowIterator = materialList.iterator();
    }
```

```
    }

    private void initDeleteEmployees() {
        // Select next data to delete based on last_index_time information ke
        // After the import was finished, the last_index_time is updated witl
        String dateString = context.getVariableResolver().resolve("dih.employ
        LocalDateTime startDate = LocalDateTime.parse(dateString, DateTimeFor
        List<Map<String, Object>> materialList = employeeService.getDeleteEmp
        if (!materialList.isEmpty()) {
            LOG.info("Found " + materialList.size() + " modified groupers");
            rowIterator = materialList.iterator();
        }
    }
}
```

The company sub-entity will be loaded for each employee during the indexation time. So the *CompanyProcessor* defines for each employee how to load the company information. Following is the class implementation.

```
package com.github.schmittjoaopedro;

import org.apache.solr.handler.dataimport.Context;
import org.apache.solr.handler.dataimport.EntityProcessorBase;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.lang.invoke.MethodHandles;
import java.util.Map;

public class CompanyProcessor extends EntityProcessorBase {

    private static final Logger LOG = LoggerFactory.getLogger(MethodHandles.l

    private CompanyService companyService;

    @Override
    public void init(Context context) {
        super.init(context);
        companyService = new CompanyService(context.getSolrCore().getName());
    }

    // For each employee this method is called as a sub-query to load
    // the company data associated with the employee.
    @Override
    public Map<String, Object> nextRow() {
        if (rowIterator == null) {
            initCompanyValues();
        }
        return getNext();
    }

    private void initCompanyValues() {
        // Based on the employee values load the company information.
        String id = String.valueOf(context.getVariableResolver().resolve("emp
        String version = String.valueOf(context.getVariableResolver().resolve
        if (isValidParam(id) && isValidParam(version)) {
            rowIterator = companyService.getCompanyValues(id, version).iterat
            LOG.info("Loading values of company " + id + "/" + version);
        } else {
            LOG.info("Not found values of company " + id + "/" + version);
        }
    }

    private boolean isValidParam(String value) {
        return value != null && !value.trim().isEmpty();
    }
}
```

More details about this implementation are available in the project on GitHub (https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer).

(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer#executing-custom-jar-on-solr)Executing custom jar on SOLR

To execute our custom data indexer on SOLR, first build the application using the maven command:

mvn clean package

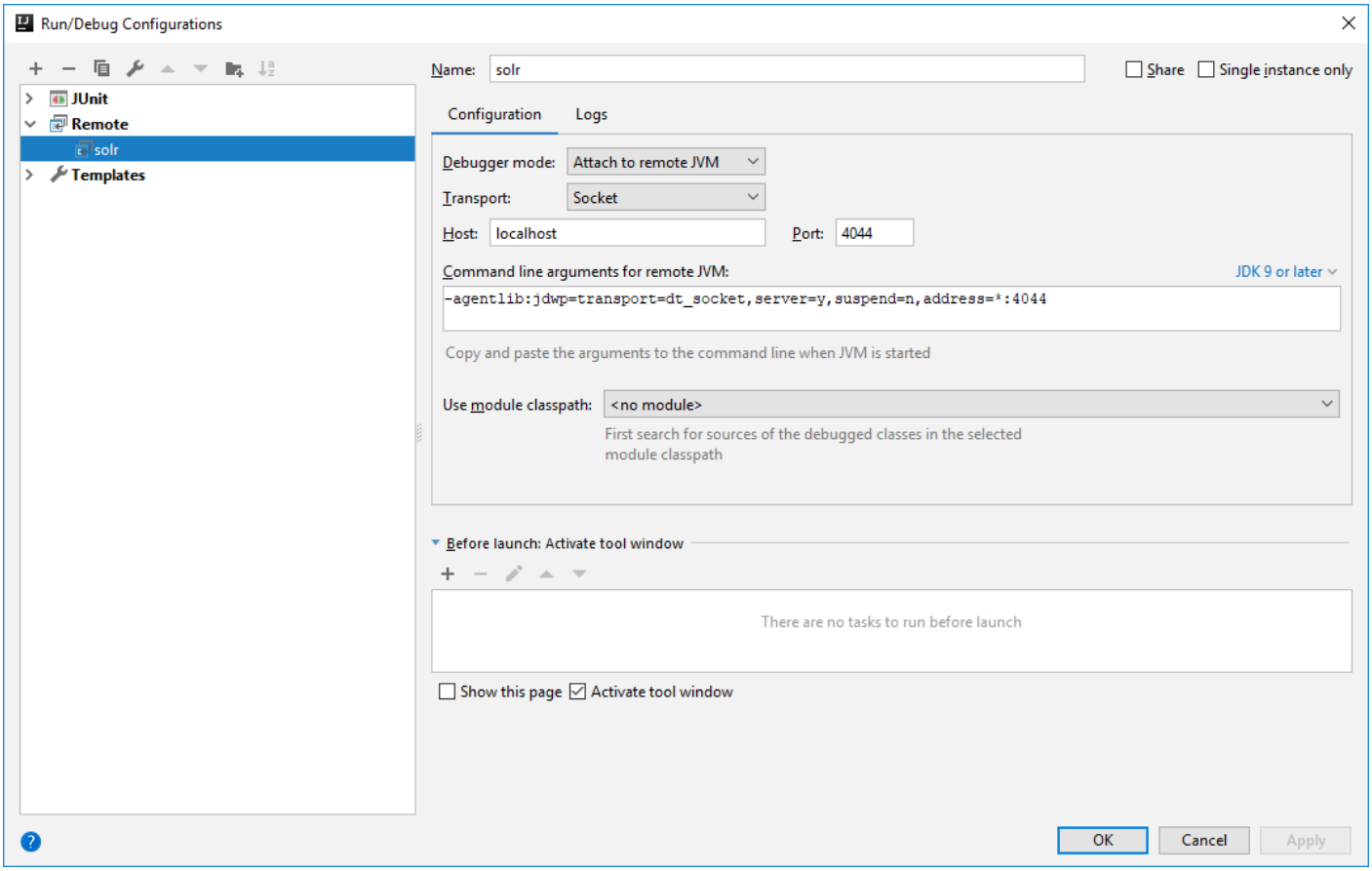
After that, copy the .jar generated under the lib folder into the SOLR core folder (C:\java\solr-7.6.0\server\solr\solr-custom-data-indexer-core\lib). In the case of debugging, starts the SOLR using the following commands. First, stop all SOLR instances:

solr stop -all

Then starts a new instance in debug mode:

solr start -a "-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=4044"

And configure a remote debug in IntelliJ (or eclipse) like the following image:



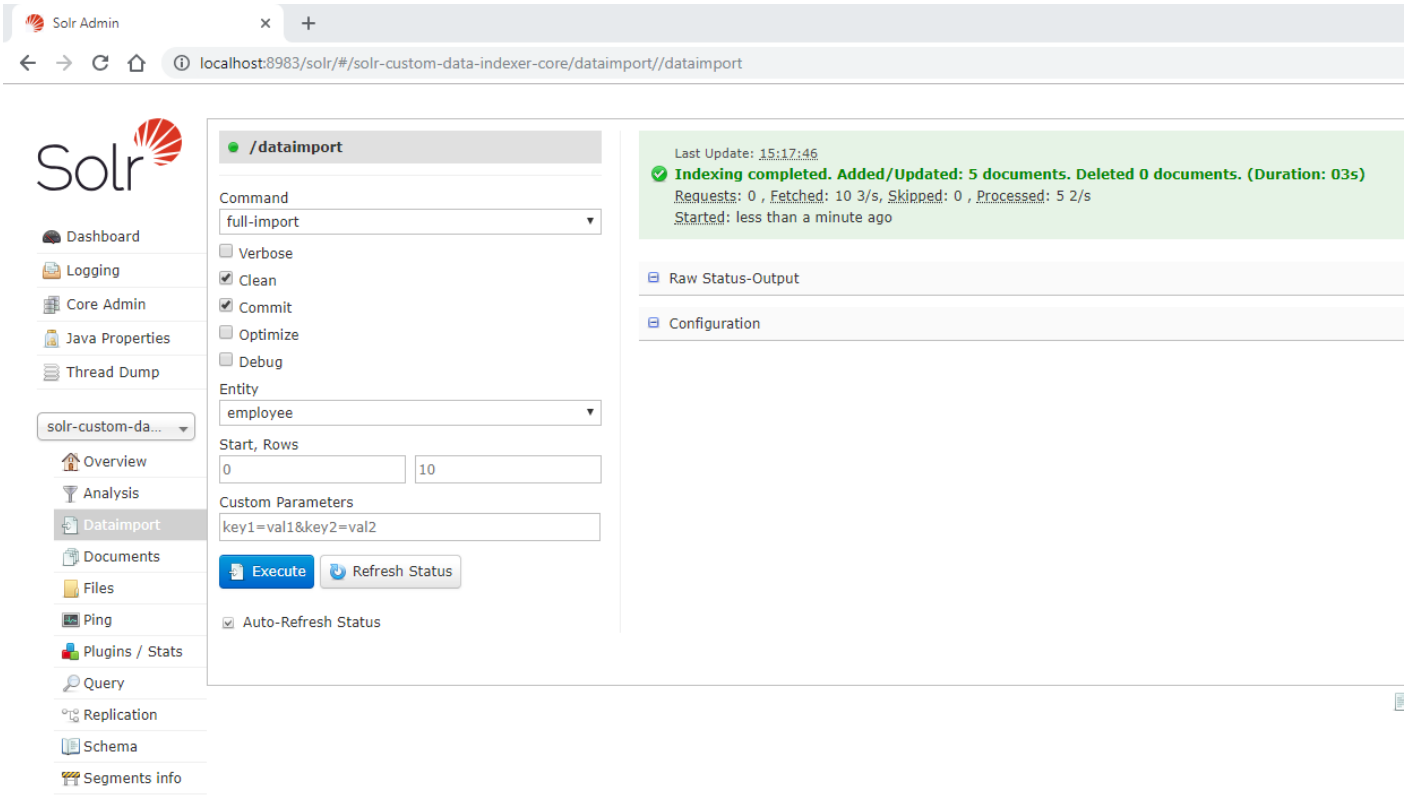
(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/blob/master/custom-solr-data-indexer/imgs/remote_solr_debug.png)

(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer#executing-indexation)Executing Indexation

After the core has been created and the project jar has been copied, then start the SOLR using the command:

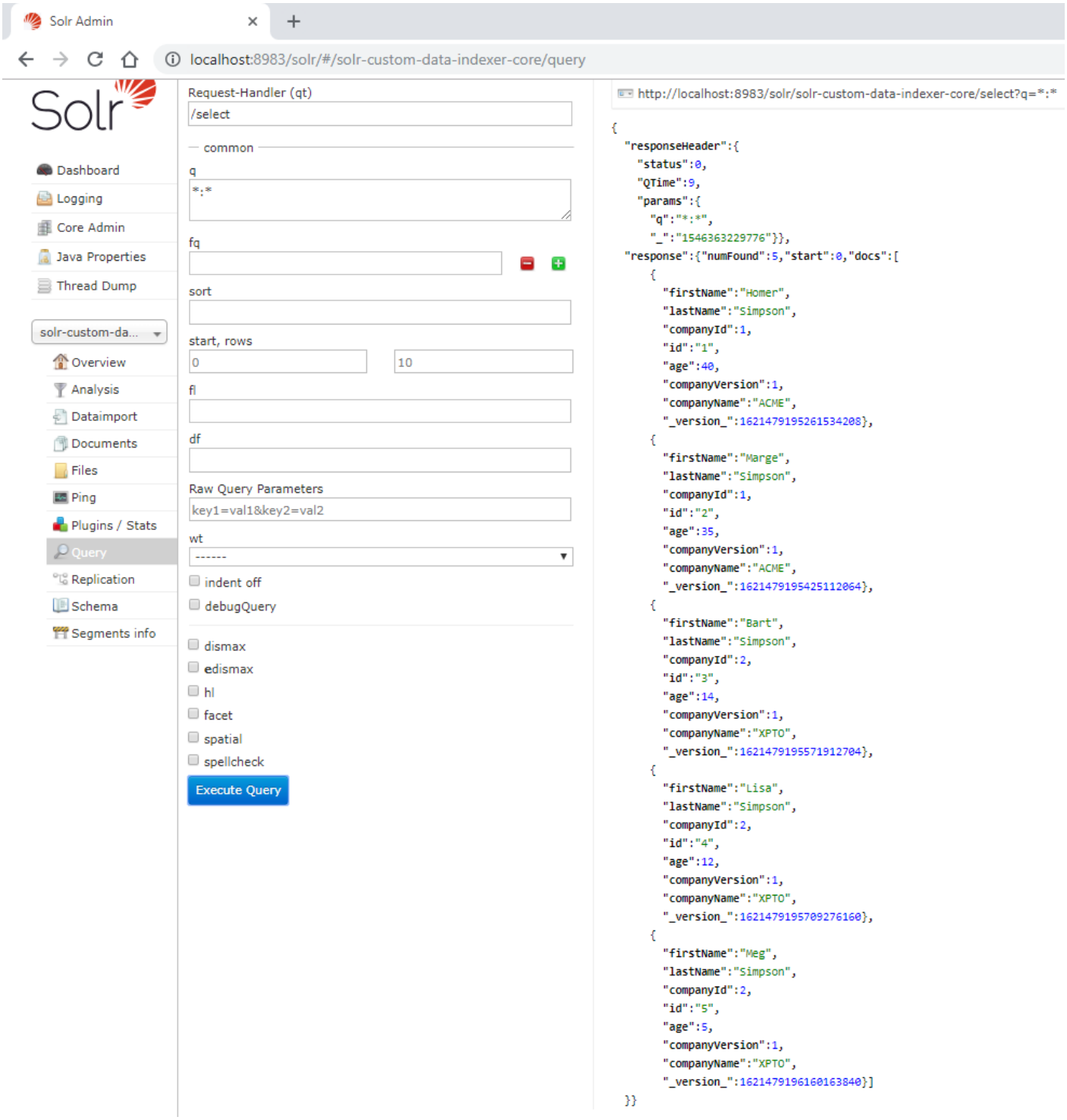
solr start

And open the URL http://localhost:8983 (http://localhost:8983/) and go the indexation page, select the *full-import* option, mark the option *Clean* and *Commit*, select the *employee* Entity, and click on *Execute* button. If everything gonna right, the indexation completed success message must be shown as following:



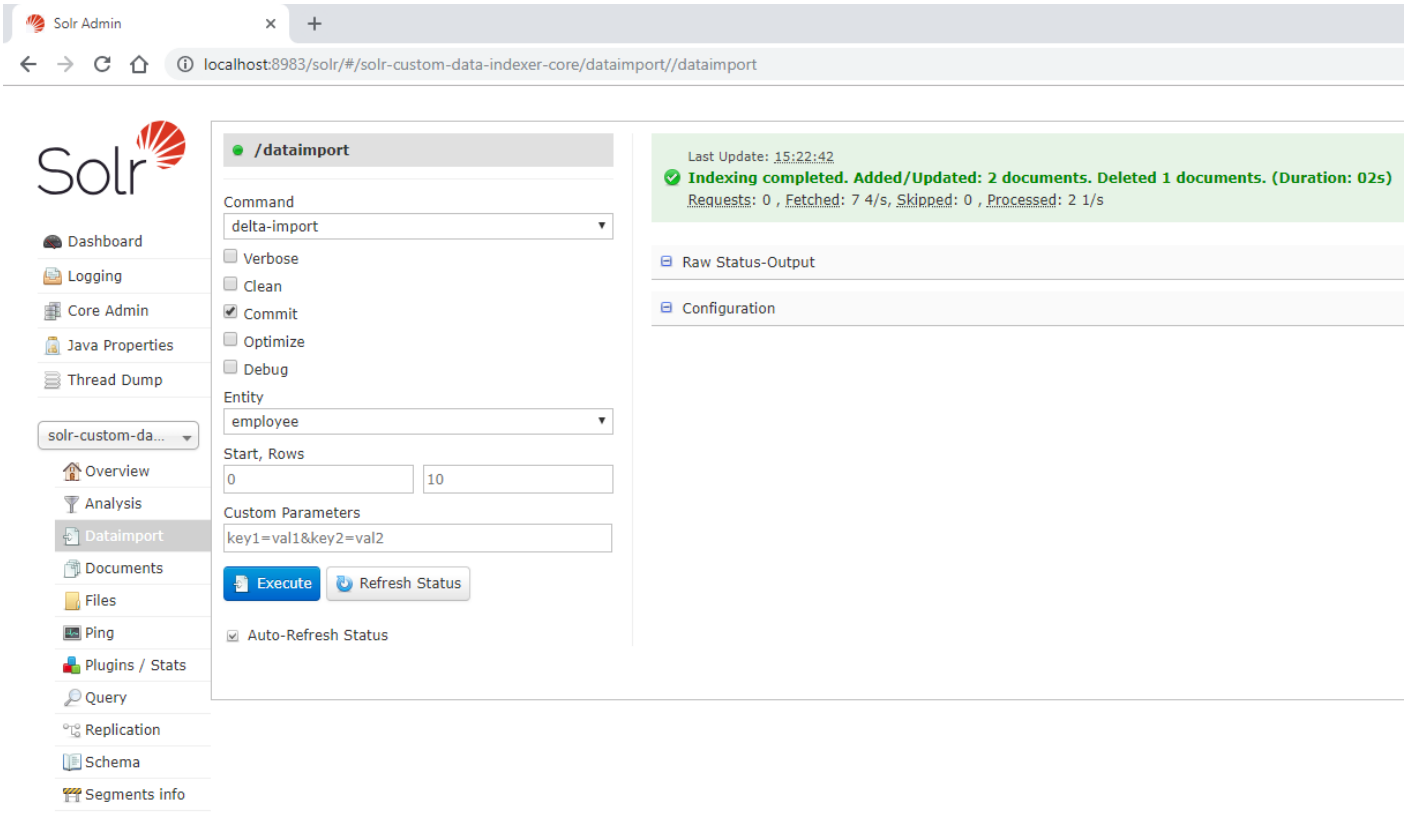
(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/blob/master/custom-solr-data-indexer/imgs/solr-full-index.PNG)

If we query the documents, a result like the following must be presented:



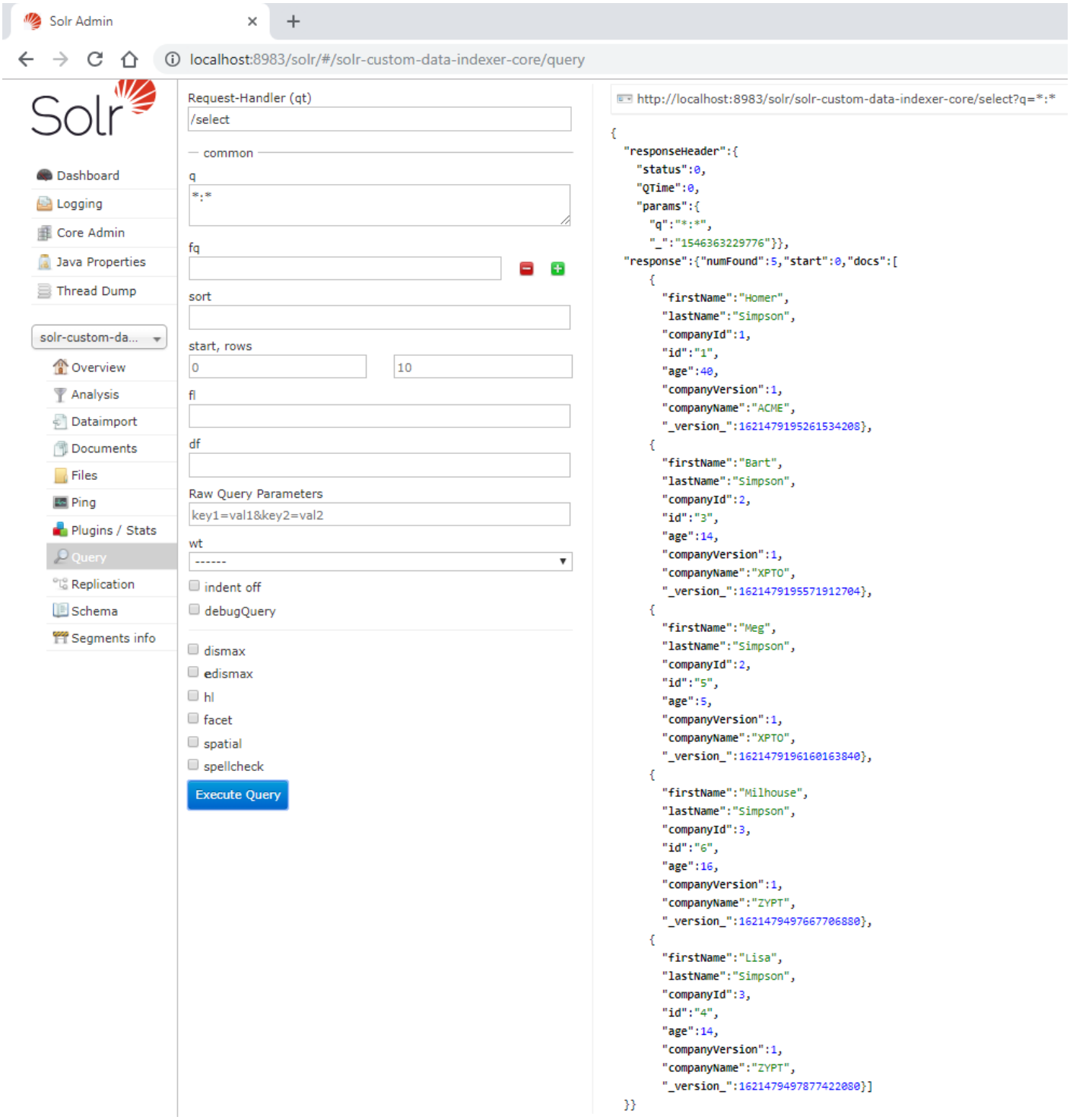
(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/blob/master/custom-solr-data-indexer/imgs/solr-full-query.PNG)

After that, go back to import view and select *delta-import* and un-mark the *Clean* checkbox. Finally, click in the *Execute* button. The following image presents the details, if the indexation went right the success message will indicate that two documents were updated/created and one was removed.



(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/blob/master/custom-solr-data-indexer/imgs/solr-delta-index.PNG)

If we query again, now we note that Lisa has changed the age and companyId, the Marge document was deleted and Milhouse was created, see the image below.



(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/blob/master/custom-solr-data-indexer/imgs/solr-delta-query.PNG)

(https://github.com/schmittjoaopedro/joaoschmitt.wordpress.com/tree/master/custom-solr-data-indexer#conclusions)Conclusions

Though SOLR custom implementations can be dangerous, if we take sufficient care about the methods used, we can consider that, in many cases, custom implementations are simpler and easy to adapt inside the SOLR project. The gains about this methodology are related to simplicity in the architecture removing third-party applications to manage the ETL and the default mechanism of SOLR to care about the nodes synchronization, document preparation, etc. We recommend to use this kind of architecture when the default SOLR mechanism to import documents is not sufficient to attend the technical requirements.

Com as etiquetas :
API,
architecture,
java,
lucene,
maven,
solr

Publicado por schmittjoaopedro

Graduado como bacharel em Sistemas de Informação pelo Centro Universitário Católica de Santa Catarina campus Jaraguá do Sul. Formado no Ensino Médio pelo Senai com Técnico em Redes de Computadores Articulado. Atualmente desenvolvedor JEE/Web em Sistemas de Engenharia na WEG. Pesquisador no período de faculdade em Informática pela Católica de Santa Catarina. Contato 47 - 99615 2305 E-mail: schmittjoaopedro@gmail.com Web page: https://joaoschmitt.wordpress.com/ Linkedin: https://www.linkedin.com/in/joao-pedro-schmitt-60847470/ Curriculum lattes: http://lattes.cnpq.br/9304236291664423 Twitter: @JooPedroSchmitt Ver todos os artigos de schmittjoaopedro

2 thoughts on “SOLR Custom Data Import Handler (Customizing the JDBC API to support REST services)”

Best Insurance in Town | Getting the Best in Life diz:
23 23+00:00 Abril 23+00:00 2019 às 4:28 | Editar
I do not even know how I ended up here, but I thought this post was good.

I do not know who you are but definitely you are going to a famous blogger if you are not already 😉 Cheers!

 Responder
Justin diz:
23 23+00:00 Abril 23+00:00 2019 às 13:38 | Editar
Thanks, it is quite informative

 Responder