TE REST API – Development Guide

This guide is an attempt to help new developers of the TE REST API project create the most typical endpoints. These being, simple CRUD endpoints. This guide assumes that the reader has access to the existing code

Table of Contents

[Basic architecture](#_gjdgxs)com/tripwire/terestapi/SpecificationWithMockedPML.groovy

[Components Diagram](#_30j0zll)

[Endpoints architecture](#_1fob9te)

[PML Search Design](#_3znysh7)

[TE Rest API utility classes for DB queries](#_2et92p0)

[Using PML to fetch something from the DB](#_3fwokq0)

[Using the perform strategy](#_4d34og8)

[Using the apply strategy](#_17dp8vu)

[Creating a new endpoint](#_lnxbz9)

[Basic Endpoint](#_35nkun2)

[ObjectValidator](#_1ksv4uv)

[ObjectConverter](#_44sinio)

[ObjectBusiness](#_2jxsxqh)

[ObjectRestService](#_z337ya)

[Utility Classes](#_3j2qqm3)

[DateUtils](#_1y810tw)

[PMLEntityFactory](#_4i7ojhp)

[PMLSearchFactory](#_2xcytpi)

[PMLQueryHelper](#_1ci93xb)

[SearchConditionsTrait](#_3whwml4)

[NodeDataTrait & others](#_2bn6wsx)

[Unit Tests](#_qsh70q)

[Tools of the trade](#_3as4poj)

[Spock](#_1pxezwc)

[JMockit](#_49x2ik5)

[Jacoco](#_2p2csry)

[Tests definition](#_147n2zr)

[Given](#_3o7alnk)

[When](#_23ckvvd)

[Then](#_ihv636)

[Where and Unroll](#_32hioqz)

[Expect](#_1hmsyys)

[Testing with PML objects](#_41mghml)

[Examples](#_vx1227)

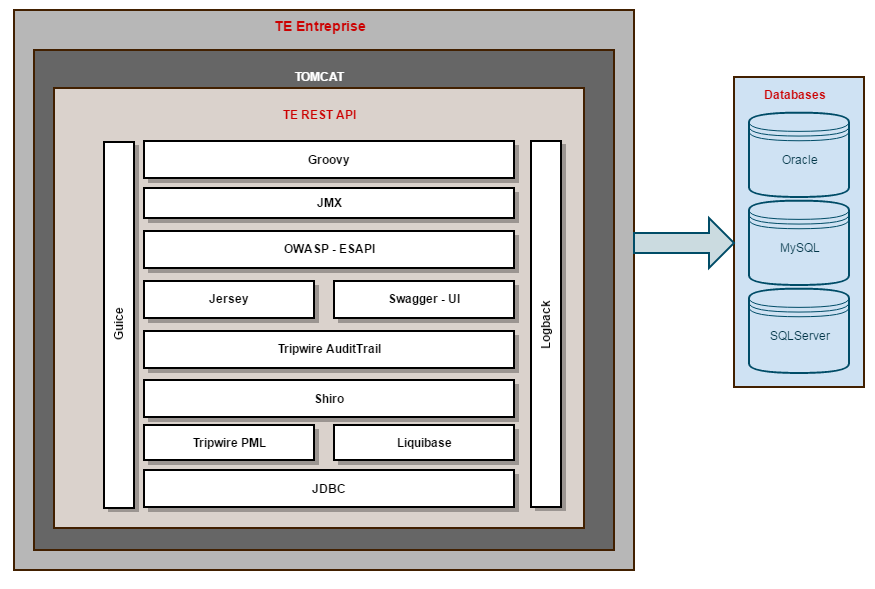
|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Contributor** |
| 1.0 | 2016-11-02 | Javier Deferrari |

# Basic architecture

The REST API is a simple java war running on a Tomcat server inside TE. The webapps folder, containing the different wars that TE uses can be found on TE\_HOME/Server/webapps.

Running in the TE environment, the REST API has access to TE’s internal API, which allows us to use TE’s data layer (aka PML) and processes.

## Components Diagram



**Groovy** – The API is written in groovy

**JMX** – Used to retrieve performance and usage metrics

**OWASP – ESAPI** – Used to avoid known hacking strategies (eg. XSS)

**Jersey** – Used to define the REST endpoints

**Swagger UI** – Web client generated by swagger to easily use the API

**Tripwire AuditTrail** – Tripwire library used to log the use of the API into TE SAL logs

**Shiro** – Used for user permission handling

**Tripwire PML** – Tripwire libraries for handling of TE objects. Including access to the database and other processes

**Liquibase** – Used to create new tables and views necessary for the API

**JDBC** – Connectivity to the databases

**Logback** – Logging framework

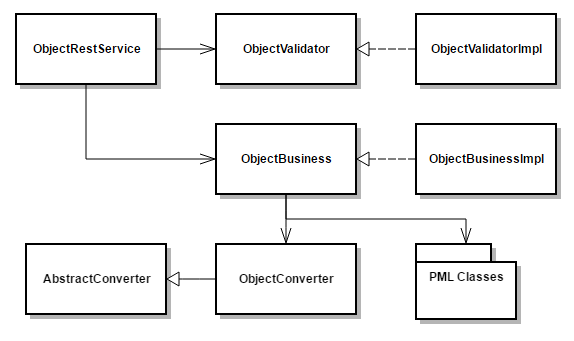
**Guice** – Dependency Injection Framework used to glue the different parts of the API

## Endpoints architecture

Our endpoints all follow the same layer structure. A Rest layer, Business layer and Data layer. The data layer is typically PML, which is used by the Business layer.

Each set of endpoints for one same resource (eg. all Node endpoints) share several classes on each layer (eg. NodeRestServices and NodeBusiness).

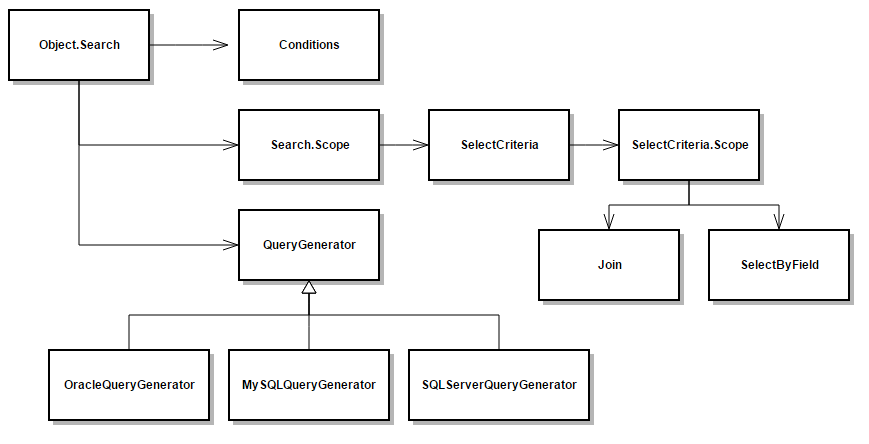
Here is a simplified diagram of how the most basic design of the endpoint looks.



The ObjectRestService works as the REST interface for a specific Object (eg. Node). ObjectRestService will ask the ObjectValidator to validate the input (eg. request body) and reply with an error if needed. Given the validation passes, ObjectRestService will ask ObjectBusiness to do the hard work and will reply with the necessary HTML code according to the ObjectBusiness result

ObjectBusiness uses PML for all DB working and TE processing. From this it will get the PML object (eg. Node) and will create the needed POJO using an ObjectConverter. This POJOs will be what the Business will return to the REST layer.

# PML Search Design

Database searches are usually done through PML. Each Object has a Search class which controls queries to the DB for that given object.

In order to do a DB query a Search has to be instantiated. Conditions will be added (WHERE of the SQL) to filter the wanted results. Joins (Inner or Outer) can be added if needed. The search object has a number of utility methods to add common conditions (eg, *forIdentity(long)* will add a Condition that checks for the object’s id)

After having a ready Search *perform(Transaction)* can be invoked. This will execute the query and create a SearchResults object with the results of the query.

*getObjects(Object.class)* or *getOnlyObject(Object.class)* can be invoked on SearchResults to get the results as a PML Object (or list).

Another option is to use the apply strategy. This means invoking *apply(Transaction)* instead of perform. Apply will execute the query and call a reader with the results of said query. The reader can get the data from those results and process it as it wants. This option is normally more performant but has a higher complexity.

# TE Rest API utility classes for DB queries

In order to simplify development and enable unit testing to a business level we created several utility classes.

* PMLSearchFactory
  + Creates the Search objects
  + Was created to simplify mocking the Search objects on unit tests
* PMLEntityFactory
  + Creates PML entities that access the db when instantiating
  + Was created to simplify mocking the Search objects on unit tests
* PMLQueryHelper
  + Contains multiple helper methods to manipulate the search
  + We should use this class instead of the traits whenever possible
* SearchConditionsTrait
  + A groovy trait that includes several methods to add conditions to a search
  + A class that needs to use it must implement it
    - Groovy will add all the methods to this class
* ObjectTrait (eg. NodeTrait)
  + Contains multiple helper methods to simplify searching the specific object
  + A class that needs to use it must implement it
    - Groovy will add all the methods to this class

Groovy traits are supposed to be a way of doing composition of functionality by small pieces. The current use does not follow that pattern and has no benefits over using simple helper classes. On the other hand, traits will add code to the java class generated by groovy, adding complexity to it and stopping the use of code coverage tools among others.

Because of this we want to slowly migrate traits as SearchConditionsTrait to helper classes as PMLQueryHelper.

# PML Basics

PML as tripwire’s ORM has some basic concepts that are used for every data object. This includes how search works with the DB.

## PML Concepts

**Object ID**: An object id is an object’s unique numeric identifier. This id is always stored on the oid column of the table

**Mapping ID**: A mapping id is an object type’s unique numeric identifier. Each type of object (basically each class) has an unique id which identifies them. For groupable objects this is stored on the Grpbl table on the ref\_pmid column, along with other data.

**Readable Key**: All PML objects have a readable key which works as a complex id. RKs are formed by a mapping id and an object id encoded with a radix of 36 and separated by a colon. Eg. *-21asdpo1238u9:-1ad4a6sd184* is a valid Readable Key with -21asdpo1238u9 as its mapping id (encoded) and -1ad4a6sd184 as its object id (encoded)

**Mapping**: Every object type has a Mapping. The object Mapping is an inner class of the object’s class which contains the object’s db metadata (eg, the columns). Some of this information is also available on the object itself (eg, the table name) but if it’s not there it can usually be found on its Mapping. The mapping id is found on the object’s mapping

**Groupable**: Most PML objects extend from the groupable class. This class represents any object that can be found on a folder or under other objects in PML (eg. Nodes, Policies). The Groupable’s data is found on the Grpbl table, including name, id, mapping id and description.

## Strategies to fetch PML data from DB

Some of the work is the same no matter we use perform or apply.

1. Get a transaction
   1. @Transactional will create a new transaction if necessary
   2. Transaction.active() gets the active transaction
2. Create the Search object
   1. Use PMLSearchFactory for it
3. Set ordering if necessary
4. Add paging limits
   1. SearchConditionsTrait has a method for it
5. Add filtering conditions
   1. The Search object has some useful methods for it
   2. SearchConditionsTrait and PMLQueryHelper have methods for it
   3. The column information is on the object’s Mapping
6. Execute the query and fetch objects
   1. Perform or apply strategy

### Using the perform strategy

Perform is used to fetch PML objects. These have a performance issue since several queries may be necessary to get all the needed data. Eg. After executing the base SQL query to fetch nodes an additional query is needed to fetch Tags, Licenses and other information (per Node).

If performance is not an issue, the use is much simpler than apply.

The following example is taken from NodeGroupBusinessImpl (and simplified)



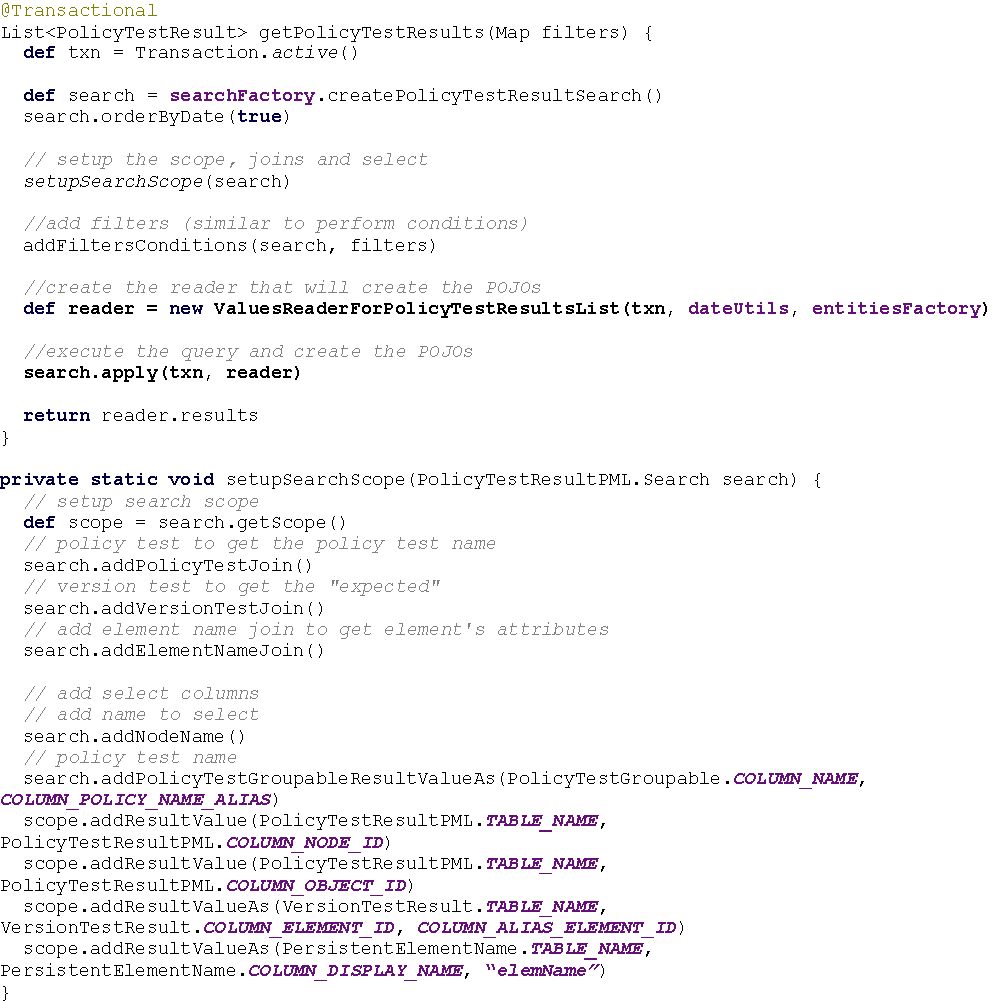
### Using the apply strategy

Apply allows to avoid the use some of the PML classes, especially the queried object itself. Following on the Node example, the Search can be setup to include all the information from the database that we need. A reader will then receive the result from the DB and create the POJO on one query.

A Search object will create the Select part of the query automatically if you don’t add any field to it. But once you add one field you need to add all of them. When we explicitly set the Select fields only those that were added will be used by the query.

So, if you have a Node.Search object and add a field for the node’s tagsets that search will only return the tagsets.

The following example is taken from PolicyTestResultBusinessImpl (and simplified)



# Creating PML mappings and search

PML Mappings are created on a complex hierarchical structure, on which each subsequent mapping inherits the table and columns of it’s father. This way, every groupable object’s mapping (eg. Nodes) include in their metadata the Groupable data. Which includes, the Grpbl table, mapping id column, name column, description column and other.

Mapping objects have a *registerTables* method which set the database metadata on the mapping. This method receives a Table to which database information (eg. columns metadata) will be added. Normally *registerTables* will first call it’s super, this will create a chain of configurations that will end with the current’s mapping setup (current table and column) and start with the first class of the hierarchy. This means that all groupable objects will have Grpbl as the main table. Meaning that SQL queries for the object will always be from Grpbl (*SELECT FROM Grpbl*) joining with the specific object table.

Following the Node’s example:

*Node* extends from *NodeGroupable* which extends from *PersistentGroupable*.

The *Node* information is on table ***Node***

The *NodeGroupable* information is on table ***NODEG***

The *PersistentGroupableInformation* is on table ***Grpbl***

An SQL query performed with a Node.Search object will be of the form: **SELECT FROM Grpbl JOIN NODEG ON NODEG.oid=Grpbl.oid JOIN Node ON Node.oid = Grpbl.oid**

This is useful if the intent is to recover the full node information. But when we only want to know the information that is stored on the *Node* table, the extra joins are unnecessary and add to the time needed to execute the query.

It’s not possible to modify an existing Mapping or Search object to take out the unwanted joins, but we can create a new Mapping and Search object with only the needed tables and columns.

In order to do this we need to create a new Mapping class which overrides the *registerTables* method and calls *realize* on its constructor. *realize* will call *registerTables* and the new Mapping will be setup accordingly.

An example of this is *PolicyBasicMapping*, which is used to retrieve the information from the *Policy* table and includes

**class** PolicyBasicMapping **extends** PersistenceMapping {

PolicyBasicMapping() {

**super**(Policy.***TABLE\_NAME***)

realize()

}

@Override

**protected void** registerTables(ClassTableMap classTableMap) {

**if** (!classTableMap.isTableMapped(Policy.***TABLE\_NAME***)) {

**def** table = **new** Table(Policy.***TABLE\_NAME***);

table.setAppendable(**true**);

table.addColumn((**new** IdentityColumn(Policy.***COLUMN\_OBJECT\_ID***)).setPrimary());

table.addColumn(**new** BooleanColumn(Policy.***COLUMN\_PURGE\_OLD\_DATA***)).setNotNullable();

table.addColumn(**new** IntegerColumn(Policy.***COLUMN\_PURGE\_OLD\_DATA\_DAYS***)).setNotNullable();

table.addColumn(**new** BinaryDataColumn(**'nodeValue'**));

table.addColumn(**new** BinaryDataColumn(**'nodeName'**));

table.addColumn(**new** StringColumn(Policy.***COLUMN\_SCAP\_PROFILE\_ID***, 1023));

classTableMap.addDescendantTable(table);

}

}

}

*registerTables* first create a table object with the column metadata for *Policy.TABLE\_NAME* (which is ‘Policy’). The *primary* column (Policy.COLUMN\_OBJECT\_ID, oid) will be used for subsequent joins to this table.

*classTableMap* is a *ClassTableMap* object present on every Mapping object which has the database metadata. The first table added through the method *addDescendentTable* will be the root table of SQL queries generated by *Search* objects that use this mapping.

This *Mapping* can the be used by a *Search* to create the query. For this to happen, a *ConcreteSearch* is created using this *Mapping*. This is done on the *createGenericSearch* method of *PMLSearchFactory*

It is helpful to note that any Search object to which no join or column values was added will use the information on *ClassTableMap* to create the query, additional joins can be added, but those included by tables on *ClassTableMap* can’t be removed. On the other hand if a specific column value is added (*addResultValue* method on *Search.scope*), the columns present on *ClassTableMap* won’t be used (as explained on Using the [apply strategy](#_17dp8vu))

# Creating a new endpoint

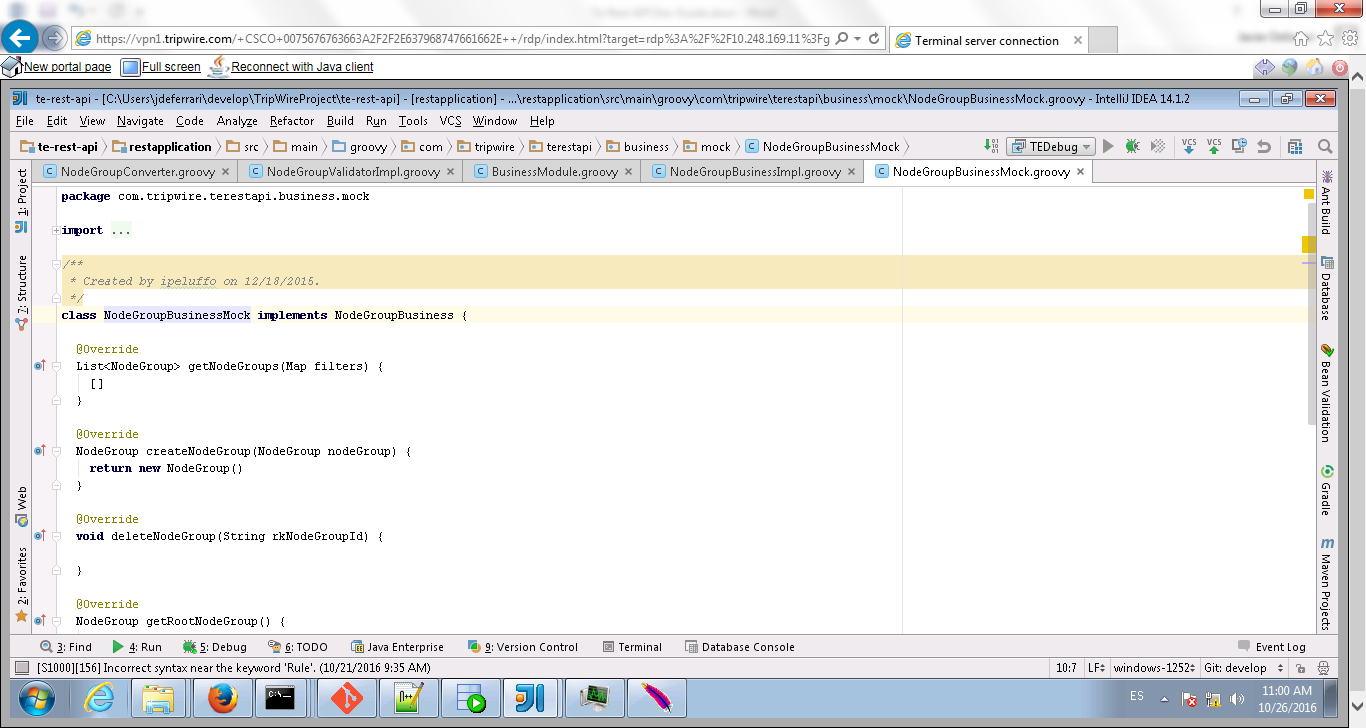
There are 4 basic classes we need to write for a typical endpoint. ObjectRestService, ObjectBusiness, ObjectValidator and ObjectConverter (replace Object by the actual object name eg, NodeValidator, TaskRestService, PolicyTestRestService)

* ObjectRestService
  + This is the Rest layer of the endpoint, the endpoint definition itself.
  + This will have one public method per endpoint related to the Object
* ObjectValidator
  + Class in charge of doing simple validations of the endpoint’s input. Eg. type checks, value ranges
  + ObjectValidator is an interface implemented by ObjectValidatorImpl
  + Will be used from the ObjectRestService
* ObjectBusiness
  + Class which does the heavy work.
  + ObjectBusiness is an interface implemented by ObjectBusinessImpl
  + Once the input was validated, ObjectRestService will call this class to execute the actual process (eg. fetch the requested data of a GET endpoint)
* ObjectConverter
  + Does the conversion from a PML class to a POJO
  + ObjectRestService should only know about POJOs and not PML classes. ObjectBusiness will use the converter to transform the PML objects to POJOs before giving them back to the Rest layer
  + Only needed when using the perform strategy

## Basic Endpoint

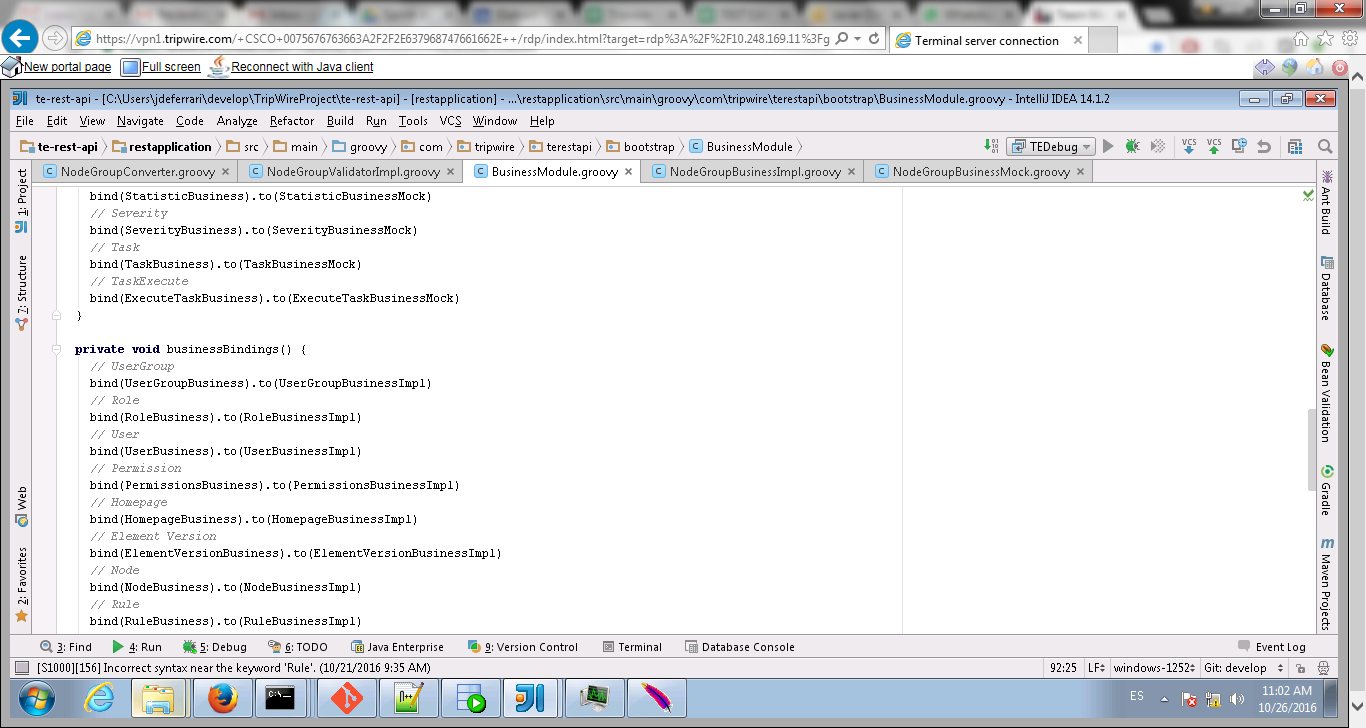
To create a new endpoint follow this steps

1. Create the ObjectConverter (as a Singleton) if there wasn’t one already created
2. Create the ObjectValidator and its implementation (as a singleton) if there wasn’t one already created and create a validation method for the endpoint
3. Create the ObjectBusiness and its implementation (as a singleton) if it wasn’t already created.
4. Create the ObjectBusinessMock if it wasn’t already created
   1. Swagger will use this one when creating the documentation



1. If new Business objects were created, add them to the BusinessModule guice module
   1. ObjectBusinessImpl in the businessBindings method
   2. ObjectBusinessMock in the businessMockBindings

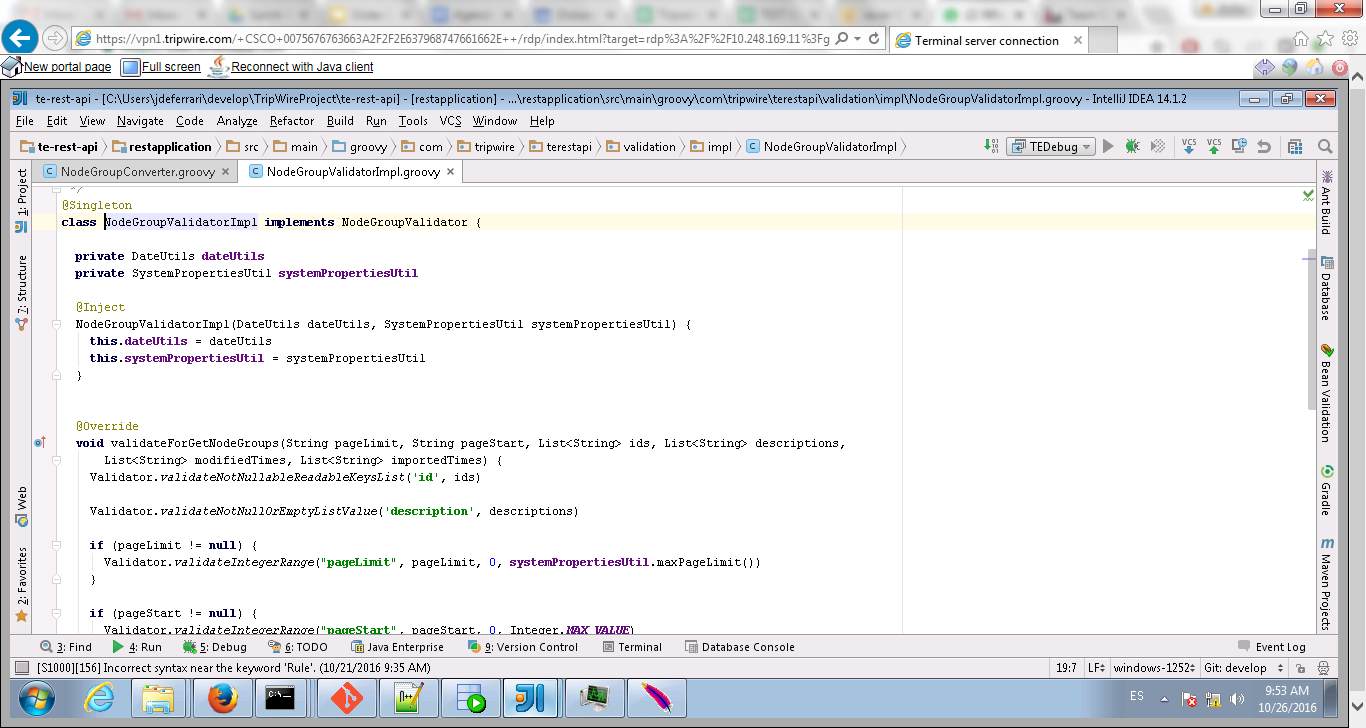
This is necessary to bind the class to Guice and be able to use IoC



1. Create the ObjectRestService if it wasn’t already created
   1. Annotations are used for both the endpoint definition (jersey) and the swagger documentation

### ObjectValidator

The ObjectValidator does simple validations on the input of an endpoint, checking that required values were sent, making sure specific text formats are being used, checking for dates, etc… Anything that can be validated by only looking at the input should be validated here



A helper class with some validations already exists (Validator).

The validator should be created as a Guice singleton (@Singleton) and have an interface and implementation. The constructor must be annotated with @Inject to give control to Guice for IoC

If new Validator objects were created, add them to the ValidatorModule guice module ObjectValidatorImpl in the ValidatorBindings method.

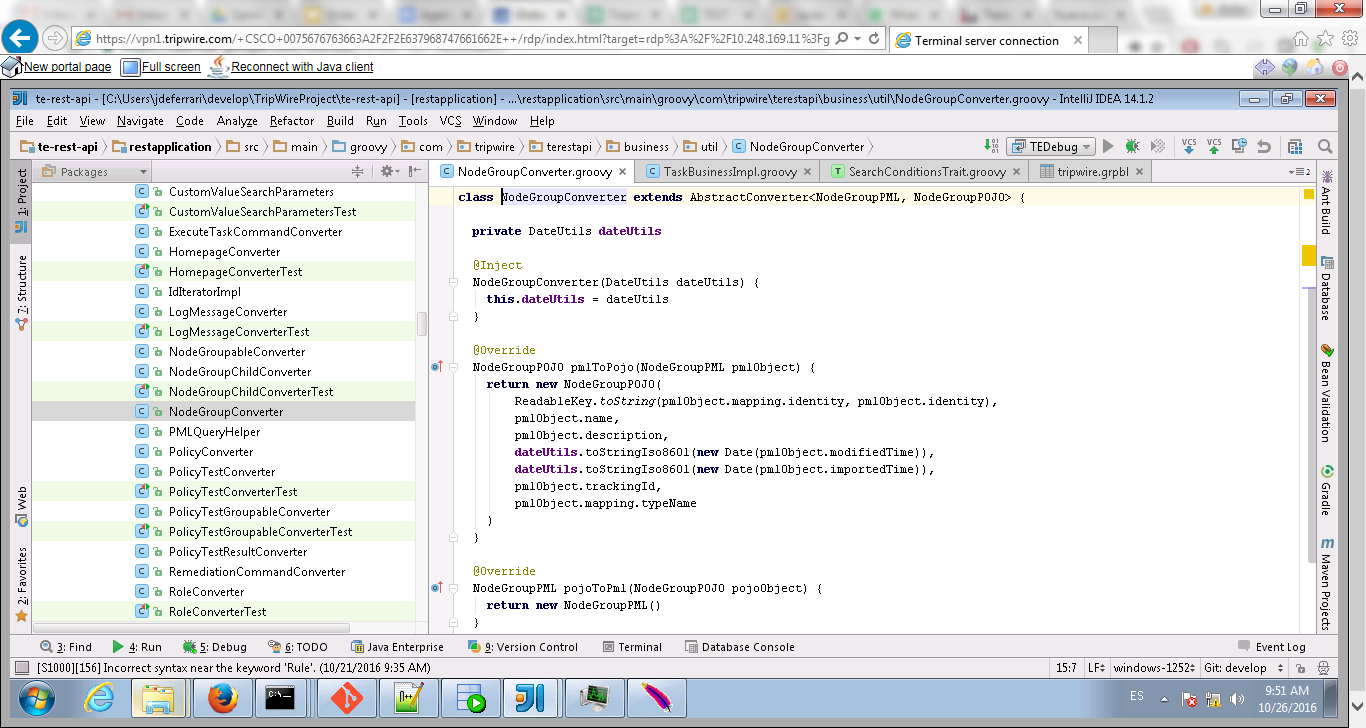
This is necessary to bind the class to Guice and be able to use IoC

### ObjectConverter

The ObjectConverter has 2 methods, one to receive a PML object and create a POJO and one to do the opposite. ObjectConverters extend AbstractConverter and must be created as Guice singleton (@Singleton). The constructor must be annotated with @Inject to give control to Guice for IoC

Sometimes it may be necessary to do database queries on a converter (PML may need to fetch data from the db), in that case Transaction.active() can be used to acquire the currently active transaction, without modifying the method’s signature.

Sometimes both methods are not necessary, in that case make sure the one that is not used throws a NonImplementedException



### ObjectBusiness

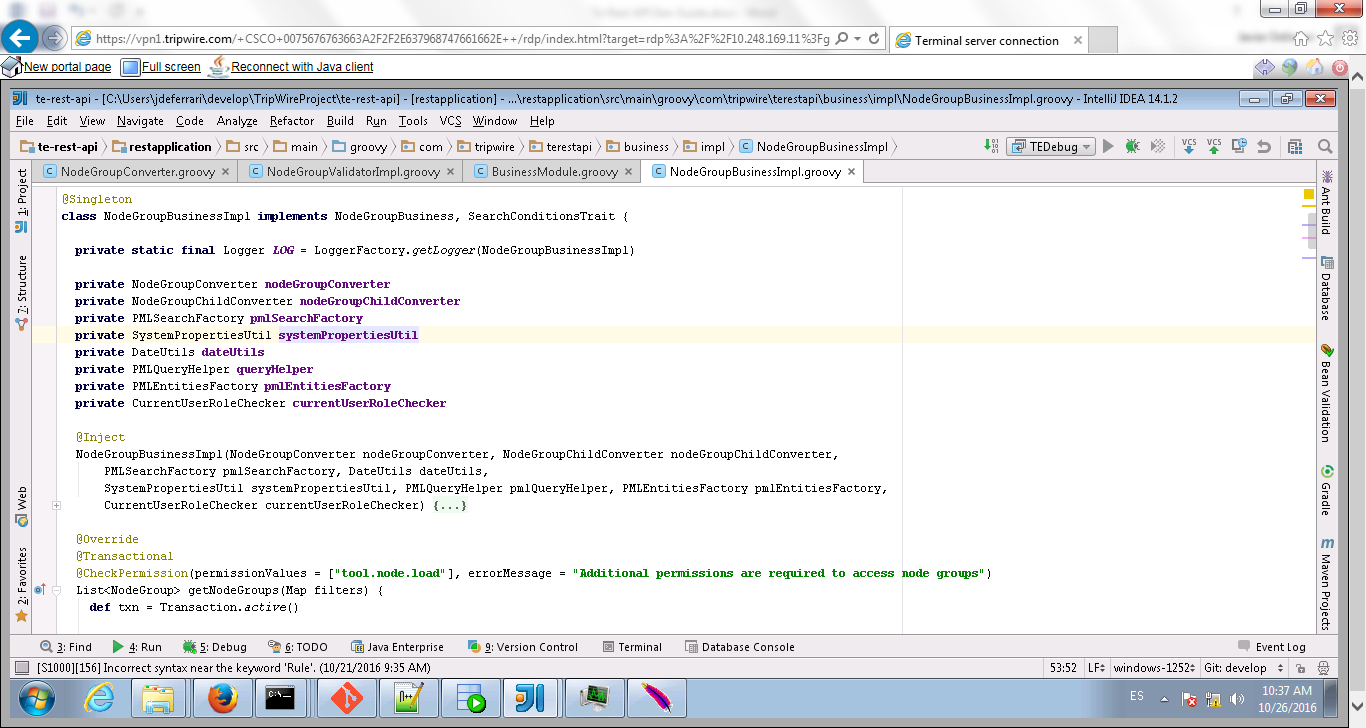
All complex logic is done on the business classes. Each should have an interface and its implementation and be created as a Guice singleton (@Singleton).

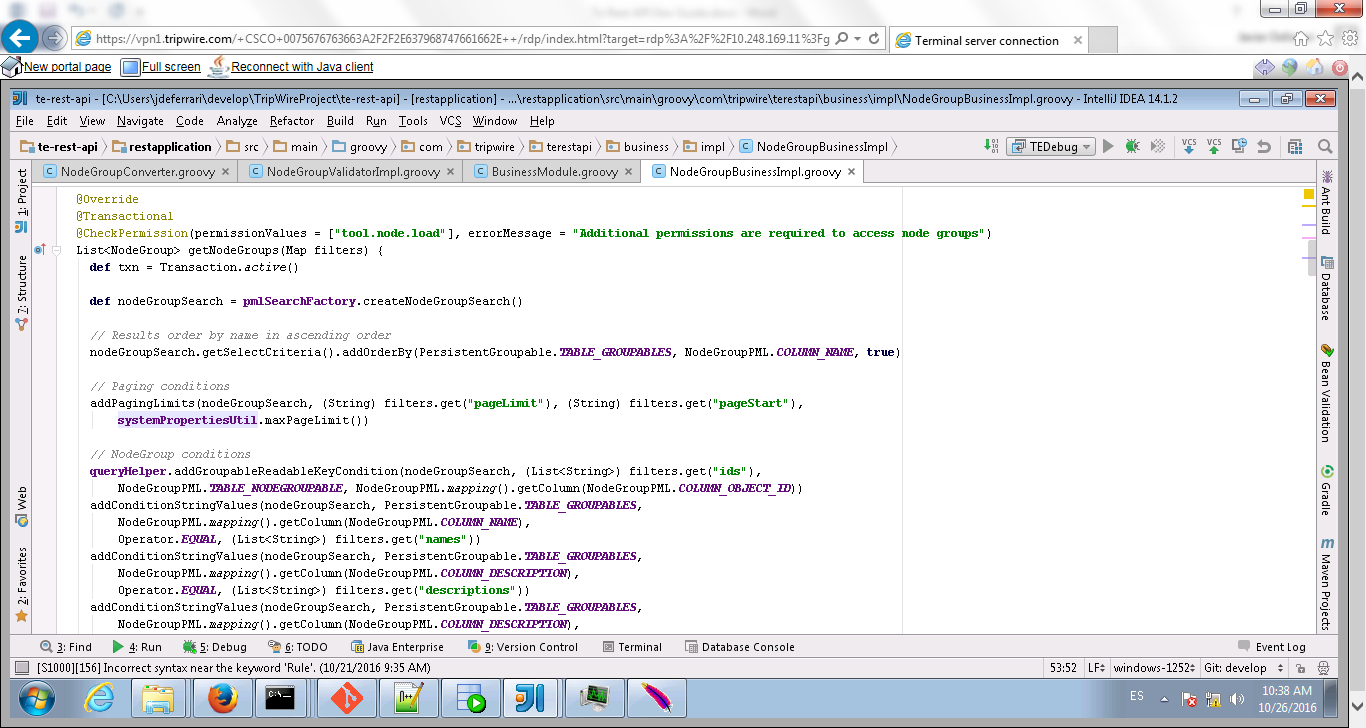
In order to acquire a new transaction, the @Transactional annotation is used. This will create a new transaction when the method is invoked and Transaction.active() can be used to retrieve this transaction

To make sure only users with the needed permissions use the endpoint, the @CheckPermission annotation needs to be added. Users failing to have the permissions will receive a 403.

To help with db queries, SearchConditionsTrait and PMLQueryHelper can be used. To use SearchConditionsTrait or any other Groovy trait, we need to add it as an interface of the class

The constructor method has to be annotated with @Inject, so Guice can inject all private fields to the object. IoC is made through the constructor.





When working with PML, **PMLSearchFactory** has to be used to create new Search objects. This Search will then be setup as needed to make the necessary query to the database

PMLEntityFactory must be used to create any PML objects that need to be mocked on unit tests. This are normally those objects that have interactions with the database.

DateUtils includes all date helper methods.

### ObjectRestService

The endpoint definition is done on the ObjectRestService using Jersey annotations.

The constructor must be annotated with @Inject to give control to Guice for IoC

@Path on the class will set the base path of the endpoint, which will be completed by other @Path annotations on different methods (if any).

Each method should have the @Produces annotation unless 204 is the succesful response and a @Consumes if there is a body on the request (as on POSTs)

@QueryParam can be used on a method attribute to mark it as a query parameter (one that is present on the URL after the interrogation mark ?, eg. /api/v1/nodes?name=…)

@PathParam can be used for path parameters (one that is on the resource description, eg /api/nodes/123 where 123 is the id of the node). For these to work, the @Path annotation must include the parameter between brackets. Eg. @Path("{rkNodeGroupId}/links")

The method must return a Response object, which will have the http code and any response body and headers. Successful requests will be handled by this Response, but failing requests (eg. 400 responses) will instead throw a WebApplicationException which will include the Response that Jersey will use later to send to the user. Exceptions for each type of error are included on the *exceptions.web* package of the project, being BadRequestException and NotFoundException the most common. 401 and 403 errors are, to an extent, already being handled and won’t normally need any coding.

401 will be thrown by TE itself if the user tries to use the API without being on a session or without sending the credentials with Basic Auth.

403 will be thrown by the API if the permissions check when using the @CheckPermission annotation failed for the user.

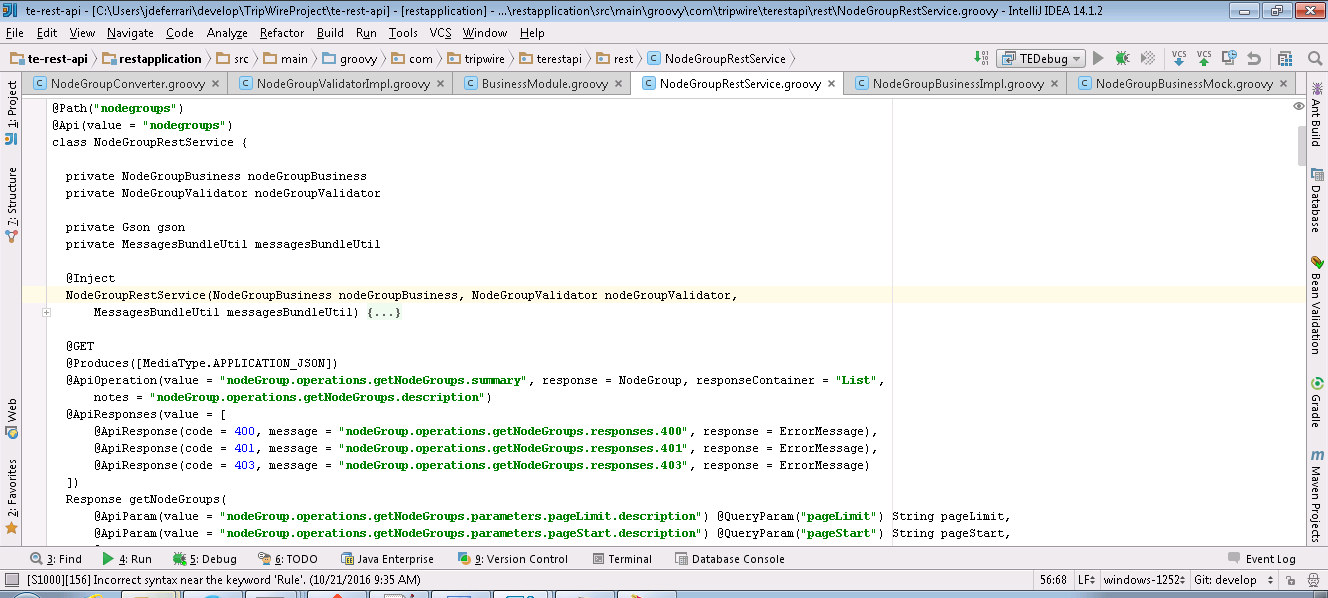
Normally the flow of a method in ObjectRestService is:

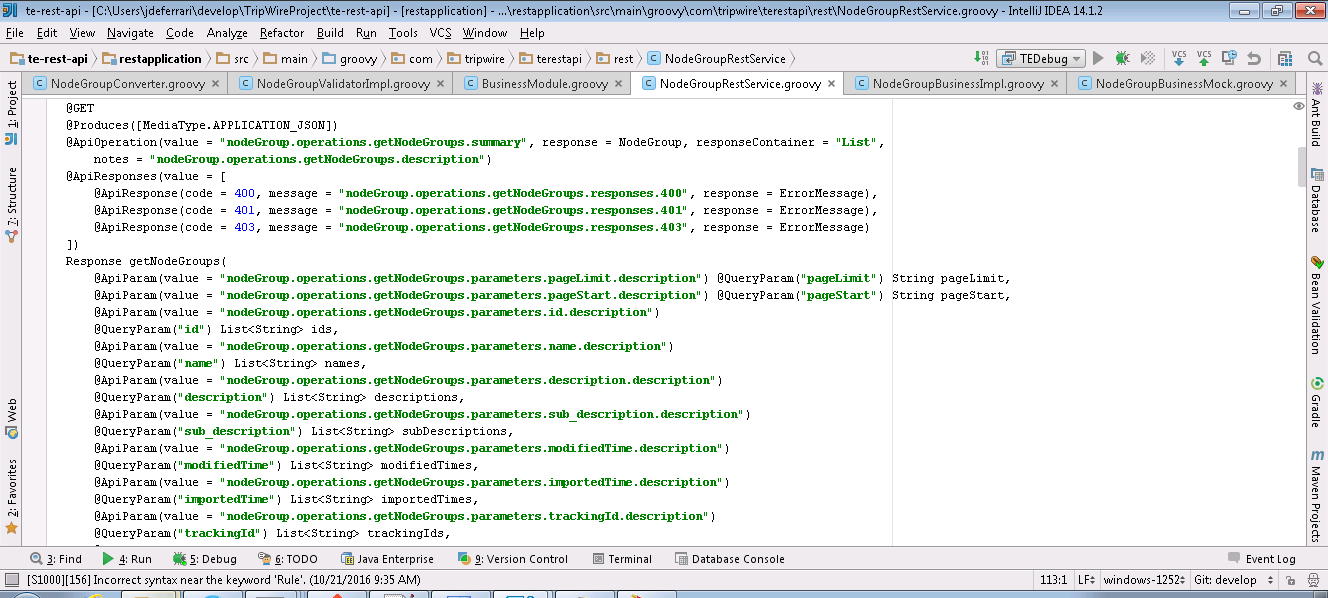
1. Validate input using the ObjectValidator
2. Invoke the BusinessObject
3. If BusinessObject threw an exception, catch it and throw the right exception
   1. The error message is taken using MessageBundleUtil, the message itself should be on the annotations.properties file
4. Return a Response with the successful status (200 normally) and add the BusinessObject response as an entity to it after marshalling to a json with Gson
   1. Return Response.ok().entity(gson.toJson(res)).build()

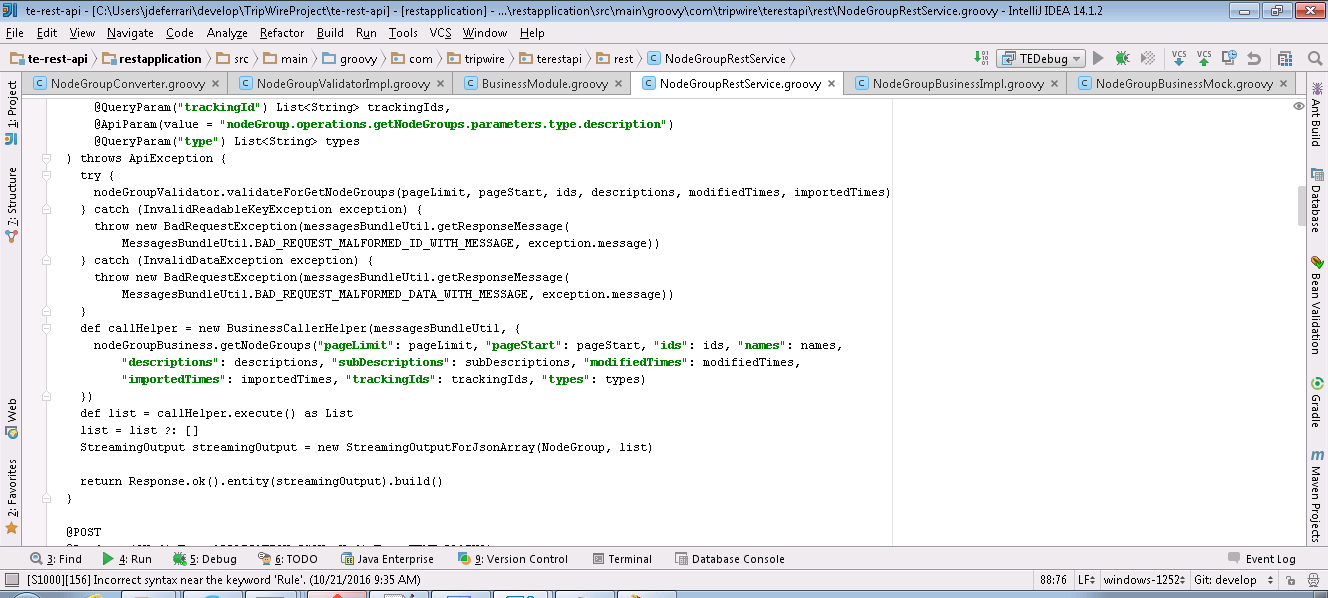
If the response is large enough (as on a GET all endpoint), then an OutputStream should be used to return the data.

Eg.

*def streamingOutput = new StreamingOutputForJsonArray(NodeGroup, res)  
return Response.ok().entity(streamingOutput).build()*







Swagger annotations are used to create the endpoint’s documentation

## Utility Classes

All utility classes are treated as singletons injected by Guice. We don’t use static public methods since these may be more difficult to mock during unit tests

### DateUtils

DateUtils has a bunch of utility methods for dates. Any date manipulation method that can be considered generic must go to this class

### PMLEntityFactory

Some PML objects try to connect to the database during the construction, PMLEntityFactory will create those objects instead of directly calling the constructor on your method. Mocking the construction of a method is tricky and not always possible with spock, this will allow us to mock the PML objects during unit tests by using PMLEntityFactoryMock (will return a mock instead of a real object)

### PMLSearchFactory

Also created to mock PML during unit tests. Any Search object must be created using this factory to allow easy mocking during testing.

### PMLQueryHelper

PMLQueryHelper has utility methods to simplify the manipulation of Search objects. Eg. adding a condition to filter by a readable key, adding a condition to filter by the type of the object.

We want to slowly move out of the current traits (eg. SearchConditionTrait) into singleton helpers like this one.

### SearchConditionsTrait

Groovy will copy all the methods of a trait into the class that implements them. This is really useful but the use we do of it is not the best. SearchConditionsTraits has a lot of utility methods to manipulate Search objects. Eg. Adding a string condition to a search. These would be better fitted on a helper class like PMLQueryHelper.

### NodeDataTrait & others

These traits have utility methods for specific types of object. NodeDataTrait has methods for Nodes while WaiverDataTrait has methods for waivers. This was done so if any business object needed to find an object it could use that trait.

# Unit Tests

We are working in improving our code coverage percentage of unit tests. At beginnings of 2016 unit tests were almost nonexistent, since then we have worked towards reaching a percentage as high as possible.

## Tools of the trade

For unit tests we use Spock, JMockit and Jacoco.

### Spock

The most used unit test framework for groovy. It allows for easy mocking and test definition. Mock of java classes is possible only if they are used by groovy classes. Mocks of java static methods is supposed to work but we found that that’s not always true.

Tests done with Spock need to extend the Specification class. This makes all public methods test but setup and setupSpec. Setup is run once before every test while setupSpec is only run once before the entire test suite.

### JMockit

Used to mock static methods of PML classes. This is necessary to make sure those mocks will work.

### Jacoco

Jacoco will measure the code coverage of unit tests. Is important to note that coverage is measured based on the java classes generated by groovy and not on the groovy code written by the developers. This means that reaching 100% coverage may be impossible or not worth the effort.

Example of this is exception throwing and comparisons with ==

The jacoco configuration can be found on jacoco-coverage.gradle

Results of Jacoco code coverage can be found on *${project-path}\restapplication\build\reports\jacoco\test\jacocoTestReport.xml*

test\jacocoTestReport.xml

## Tests definition

Most Spock unit tests are composed by a Given, When, and Then sections. Some can instead use Given, When, Expect and Where. These are tests which will be executed multiple times according to what was defined on the where section.

### Given

Given deals with the definition of the data needed by the tests.

### When

When is the tested piece, here we execute whatever we want to test

### Then

Then includes all assertions. Any condition that needs to be checked after the execution.

Then Conditions to check include exception throwing and

### Where and Unroll

Spock tests can be annotated with @Unroll, this means that the test will be run multiple times using different configurations that are defined on the Where section. Each line on the Where section is a configuration for Spock to run.

### Expect

Basically the same as Then for tests with Unroll

## Testing with PML objects

We don’t do unit testing for the data layer since putting up the environment for it (creating a mock db, etc…) is not worth the effort at this point. Because of this PML object need to be mocked, which not always is an easy task because of how PML is designed. Constructors often try to connect to the db as well as other methods which you wouldn’t expect (eg. a simple setter). To cope with this difficulty we created 2 base classes which mock most of PML

SpecificationWithPMLUtils and SpecificationWithMockedPML. The former includes some methods to mock generic objects in PML (Eg. mappings) while the last does the actual setup of mocks.

Any test which needs PML mocking can extend one of those, depending on the level of mocking needed. Normally we simply extend SpecificationWithMockedPML which makes the tests a bit slower but simplifies development. SpecificationWithMockedPML will also inject all the classes to the testing environment by using the TestModules

## Examples

**def "Test toDateIso8601 - success"**() {  
 **given**:  
 String date = ***DATE\_UTC\_0***;  
  
 **when**: **"toDateIso8601() is executed"  
 def** value = **dateUtils**.toDateIso8601(date)  
  
 **then**: **"No exceptions"** value.compareTo(**new** Date(0)) == 0  
}

**def "Test toSimpleString - success"**() {  
 **given**:  
 Date date = **new** Date();  
  
 **when**: **"toSimpleString() is executed"** String value = **dateUtils**.toSimpleString(date)  
  
 **then**: **"No exceptions"** value ==~ **"(\\d){4}\\-(\\d){2}\\-(\\d){2} (\\d){2}:(\\d){2}:(\\d){2}"**}

This last example shows the use of regex comparison in groovy

@Unroll(**"Get millisecond range with value #value"**)  
**def "Test getMillisecondRange - success"**() {  
 **expect**: **'returned date ranges are of 1 millisecond'** rangeValues.collect(**dateUtils**.&toDateIso8601) == **dateUtils**.getMillisecondRange(**dateUtils**.toDateIso8601(value))  
  
 **where**:  
 value | rangeValues  
 **null** | []  
 **"2016-01-14T07:00:00.009Z"** | [**"2016-01-14T07:00:00.008Z"**, **"2016-01-14T07:00:00.010Z"**]  
 **"2016-03-24T00:00:00.000Z"** | [**"2016-03-23T23:59:59.999Z"**, **"2016-03-24T00:00:00.001Z"**]  
 **"2015-12-31T23:59:59.999Z"** | [**"2015-12-31T23:59:59.998Z"**, **"2016-01-01T00:00:00.000Z"**]  
}

Here the test will be run once per Where row, replacing *value* and *rangeValues* with the specified value.