Design of RESTful API for NFFG verifier

1. Conceptual structure of the resources

1.1 Basic structure

The conceptual structure of the data to be represented in the service has a hierarchical structure that is:

- there is a collection of **NFFGs** (top level resource)
- the collection is a set of **NFFG** (child resource), that contain information about nodes and links
- each **NFFG** has a child resource that represents **policies**
- policies is a collection of **policy** resources
- each policy can have a **result** resource
- each **NFFG** has a child resource for verification of policies that won't be stored

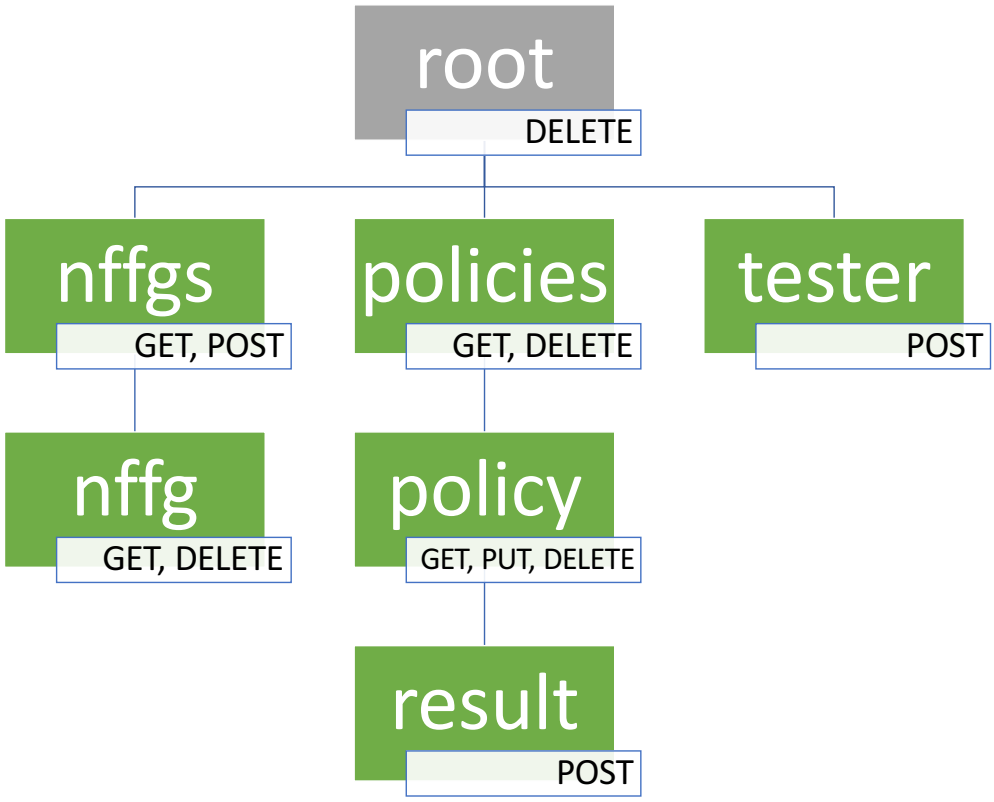
However this structure has some problems with the details of the specifications provided. The first big problem is that the policies need to be accessible by clients without knowing the NFFG they belong to. For this reason the **policies** resource should be a top level one instead of being a child resource of the **NFFG** resource. This is a direct consequence of the fact that the policies are identified by their name in the global scope insted of being restricted to the scope of the single NFFG they belong to.

The verification of policies that won't be stored has been moved as a top level resource in order to have consistency with the responses provided in case of errors: if the verification was a child resource of **NFFG**, the 404 status code should have been used by the service if the policy refers an unknown NFFG because the NFFG would have been in the path. But since the **policies** resource are detached from the **NFFGs** the checks performed by the service when receiving a policy to be stored don't use 404 because the path will not include the information about the related NFFG.

Details about the solution to these problems are provided below.

1.2 Effective structure

This is the effective structure that has been designed.



The resources are:

- **nffgs**: represents the stored collection of nffgs
- **nffg**: represents a single stored nffg belonging to the collection
- **policies**: represents the stored collection of policies
- **policy**: represents a single stored policy belonging to the collection
- **result**: represent the result of a policy
- **tester**: the endpoint for verification of policies that won't be stored

Policies placement

The policies are made available as root elements because the clients may want to get the whole set of policies stored inside the server. Since the name of a policy is unique not only iniside a single NFFG, but has a global scope, it is appropriate that also the child resource policy belongs to this subtree.

Policy creation and modification

Single policies can be created and updated using the same procedure: a PUT request on the policy resource child of policies. Because the requirements specify that if a new policy is submitted with the same name as an already stored one a replacement will occurr, to update or to create a policy the client can use the same PUT request. The HTTP method chosen is PUT, because it is idempotent and the resource path is chosen by the client (see the discussion about IDs below).

2. Mapping of the resources to URLs

The tree structure of the resources previously shown is reflected on the URLs used. Curly braces are used in the following when the path contains an identifier.

URL	resource type	method	usage
/	-	DELETE	delete all the data stored in the service
/nffgs	nffgs	GET	obtain the collection of NFFGs
		POST	store a new NFFG
/nffgs/{nffg_name}	nffg	GET	obtain a single NFFG given its name
		DELETE	delete a single NFFG given its name
/nffgs/{nffg_name}/policies	policy	GET	obtain the collection of policies belonging to a NFFG whose name is given
/policies	policies	GET	obtain the collection of all the policies
		DELETE	delete all the policies
/policies/{policy_name}	policy	GET	obtain a single policy given its name
		PUT	store a policy on this resource (both creation or update)
		DELETE	delete a single policy given its name
/policies/{policy_name}/result	result	POST	recompute the result and obtain the policy with the updated result
/tester	policy	POST	verify the policy provided in the request testing it against an existing NFFG

2.1 IDs

Since both the NFFGs and the policies are identified by their name (as specified in the requirements of assignment 1) and since their pattern is quite strict (only alphabetical characters), the names can be used directy as path elements and therefore the ID of the resources is chosen by the client. In this way the client is able to retrieve the wanted data by simply knowing the name of the nffg / policy. This reflects the methods of the interface NffgVerifier, that allows to read the data in a similar way.

For the NFFGs creation, the POST method is used because the creation is not idempotent as explained in the requirements (creation of an NFFG with the same name as an existing one is forbidden). Instead for the policies, since the creation of a policy with the same name as one already stored in the service is allowed and is used for replacement, the creation of a policy uses the same method as the replacement / update. In this way the client directly submits a PUT request to the path that contains the name of the policy, and since PUT is idempotent the creation follows the same approach as an update.

3. Operations by resource

All the operations are available with Accept and Content-Type with value application/xml. The Accept can also be using application/json since the JAXB framework can handle this too, but for the Content-Type this data representation is not available because of implementation details: the requests are validated against an XML schema, and since the JSON schema is not required the validation of JSON could not be done. Some automatic generators of JSON schema from XML schema exist, but I preferred not to use them.

The types used for data exchange are the ones contained in the XSD: nffg and policy.

All the resources also have in common some HTTP response status codes (that are automatically managed by tomcat):

- 405: the client is using an HTTP method not allowed
- 406: the client is asking for an Accept that is not allowed
- 500: generic internal server error when something unexpected happens in the service

Some custom exceptions have been designed for specific situations:

- 409: when the client tries to violate the contraint about uniqueness based on the name
- 400: used both when validation of request fails against the schema or when the request references some data that is not stored in the service
- 500: when something in the service is not working as expected. This could be a communication error with neo4j or some constraint failures. Before reaching some points that will generate unmanaged runtime exceptions, a specific exception is thrown providing a message to the client.

The following paragraphs show the details about each resource with the allowed methods.

/

The root resource is made available only for cleaning purposes. It cannot be read (for this the clients need to read the nffgs and policies separately) but can be used to clear all the data stored inside the service.

method	request type	explaination	result status	response	meaning
DELETE	-	delete all the data	204 no content	-	all the data has been deleted

/nffgs

The collection of nffgs. Can be used to read the data about nffgs or to create a new nffg. There is no possibility to delete the whole set of NFFGs because, since all the policies stored refer an NFFG, to keep the constrains valid also the whole set of policies should be removed. For this reason the deletion of all the data has been moved to the root resource.

method	request type	explaination	result status	response	meaning
GET	-	get the collection of NFFGs	200 OK	set of nffgs	in the response the set of NFFGs
POST	nffg	create a new NFFG	201 CREATED	created nffg	the NFFG has been created and is provided back to the client
			400 BAD REQUEST	error string	schema validation error
			409 CONFLICT	error string	a NFFG with the same name is already stored

The POST request must contain the field name, that will be the identifier of the created resource if the request succeeds. In case the name is already used by another stored NFFG, the service returns a HTTP 409 error. Instead if the request itself contains an error when doing validation of the data contained, the service returns a HTTP 400 error.

/nffgs/{nffg_name}

A single nffg identified by its name. It can be read or deleted.

method	request type	explaination	result status	response	meaning
GET	-	get the NFFG	200 OK	nffg	in the response the NFFG corresponding to nffg_name
			404 NOT FOUND	error string	no NFFG exists with this name
DELETE	-	delete the NFFG	200 OK	deleted nffg	the NFFG has been deleted and is provided back to the client
			403 FORBIDDEN	error string	impossible to delete the NFFG because some policies refer to it

			404 NOT FOUND	error string	no NFFG exists with this name

The DELETE has an optional queryParam that is required in the case that some policies are attached to this NFFG. The queryParam is force and the behaviour of the service is the following:

DELETE	force=true	force not true or missing
NFFG without policies	success (removed NFFG)	success (removed NFFG)
NFFG with policies	success (removed NFFG and all the policies referring it)	FAIL

/policies

Policies collection. Can be used to read the data about policies (with some filtering using queryParams) and to delete the whole set of policies.

method	request type	explaination	result status	response	meaning
GET	-	get the collection of policies	200 OK	policies	in the response the set of policies
DELETE	-	delete all the policies	204 no content	-	all the policies have been deleted

queryParams to be used in GET to filter the data:

- nffg: only get policies for a specific NFFG (whose name is the value of the parameter)
- from: get policies that have been verified after the specified time and date (not implemented)

/policies/{policy_name}

A single policy identified by its id. Can be used to read the policy, delete it or sending a policy for creation or update purposes.

method	request type	explaination	result status	response	meaning
GET	-	get the policy	200 OK	policy	in the response the policy corresponding to policy_name
			404 NOT FOUND	error string	no policy exists with this name
DELETE	-	delete the policy	200 OK	deleted policy	the policy has been deleted and is provided back to the client
			404 NOT FOUND	error string	no policy exists with this name
PUT	policy	update/create the policy	200 OK	updated policy	the policy has been updated successfully and is provided back to the client
			201 CREATED	created policy	the policy has been created successfully and is provided back to the client
			400 BAD REQUEST	error string	validation error or invalid reference to stored resources

/policies/{policy_name}/result

The corresponding result for this policy. This subresource can only be used to ask to the service an update of the verification result. The verification is done using neo4j service, and the updated policy is returned to the client.

method	request type	explaination	result status	response type	meaning
POST	-	update the policy result	200 OK	policy with updated result	the reachability policy has been tested and updated in the service and the policy is provided back to the client
			404		

			NOT FOUND	error string	no policy exists with this name

/tester

Verification endpoint for client policies, not stored on the service. The reachability policy is tested by using neo4j service.

method	request type	explaination	result status	response type	meaning
POST	policy	verify this policy	200 OK	policy	the reachability policy has been tested and the policy is provided back to the client with the updated result
			400 NOT FOUND	error string	validation error or invalid reference to stored resources

4. Implementation details

4.1 Data storage

The data are stored using two maps: one for NFFGs and one for policies. While the map for policies contains directly the objects belonging to the generated class Policy, the map of NFFGs contain object belonging to the class NffgStorage. This class contains:

- the object belonging to the generated class Nffg (the data that clients can get)
- a map that contains the mapping between the node names and the ids used for the communication with neo4j: this mapping is local to each NFFG because there can be noded with the same name belonging to different NFFGs
- some flags to keep a state of the state of the communication with neo4j: those flags allow better performance in the concurrency management (see below)

4.2 Concurrency management

Analysis of methods

The synchronization without considering the removal of NFFGs is simply obtained by using ConcurrentMap.

- All the getters perform atomic operations on a single map (nffgs or policies). Then some filters can be applied on the set of values, but this operation is performed after the read of the data, so it is not a problem
- storeNffg performs a single atomic operation on the nffgs map: a putIfAbsent that is checking the existence and storing the new nffg in atomic way
- storePolicy is checking the references to nffg and nodes (src and dst) then is putting the new policy into the policies map. But since an NFFG cannot be deleted, after the check the references cannot be invalidated in any ways, so also in this case there is no need of additional synchronization
- deletePolicy is removing the policy from the map of policies in a single atomic operation
- deleteAllPolicies is clearing the policy map in a single atomic operation
- updatePolicyResult is first getting the policy from its name then is verifying its result. Also if this is not a single operation and a deletion can occur in between, this is not a problem because in this case the serialized view of the events would be that the deletion occurred after the update of the result, without side effects because the update of the result does not operate on the map but only on an object that is stored inside it, and can be safely removed from the map preventing other threads to reach this policy that still exist for the thread that is handling the update
- verifyResultOnTheFly is first validating the references contained in the policy (nffg, src, dst) and then verifying the result. Since the referenced nffg cannot be deleted (or updated) the data are still valid also if these operations are not performed atomically

Preponing the put in the NFFGs map with respect to the communication with neo4j

The method storeNffg seems to act in a strange way: first it inserts the NFFG into the map (by checking the constraints) then it handles the storage of nodes inside neo4j service. This has some consequences:

- the communication with neo4j could fail: the creator of the NFFG will receive a 500
- some GET could read the NFFG before its node are stored inside neo4j: this is not a problem because the NFFG data won't change
- some GET could read a NFFG that whose nodes have not been uploaded correctly to neo4j: those clients won't detect any problems until they want to test some policies
- some policies could refer this NFFG, and no errors will be detected

When some policies referring this NFFG will be requested an update, the check on the completeness of the storage inside neo4j is performed (because the getId method of the NffgStorage class is synchronized and checks the flags of the status). The possible situations are:

- the NFFG was successfully uploaded to neo4j: the ids will be retrieved and the verification of reachability will take place
- the NFFG was not successfully uploaded to neo4j: the getId will return null and the verification won't be executed
- the NFFG is still being uploaded to neo4j: the getId will wait for the end of the operation (success or fail):

    - on success it will proceed as above
    - on fail it will stop as above

The advantages of this solution are:

- there is no need of synchronization because the operation done on the map is a single putIfAbsent instead of doing a containsKey then after the neo4j communication doing the put
- avoiding the synchronization, more than a single storeNffg can be executed in parallel without unwanted race conditions. Since the communication with neo4j could take a lot of time (some seconds for a huge NFFG), this would kill the performances.

Considering the deletion of NFFGs (not required but implemented)

Considering also the deletion, the modifications done are the following:

- a RWLock is added for operations that modify the policies in order to make the assumptions above explained to be still valid:
    - the exclusive lock is used by the deleteNffg method. In this way, when this lock is acquired no other threads can operate on the policies (modification)
    - the shared lock is used by methods that modify the policies map
    - the getters need no synchronization if the modification keep the state always valid during the execution of single operations

In details the usage of locks by each method:

- deleteNffg in the critical section protected by exclusive lock is:
    - checking if some policies are linked and in this case can block the execution if the request does not force the removal
    - removing all the policies linked to this nffg from the policies map
    - removing the nffg from the nffgs map
- deleteAll also uses the exclusive lock
- storePolicy uses a shared lock to validate the references and then storing in the policies map the new one. In this way the removal cannot occur between the two operations, and therefore the references are still valid
- deletePolcy uses a shared lock because the iteration that is occurring in the deleteNffg over the collection of policies acts on the valueSet that is not explicitly concurrent-safe also if it coming from a ConcurrentMap. In order to avoid any problems, the deletion of a single policy is done in a protected block
- deleteAllPolicies uses the shared lock for the same reason as above
- updatePolicyResult uses the shared lock because after getting the policy from the name the verification is accessing the related nffg in order to use the ids, that must not be deleted between the two operations
- verifyResultOnTheFly uses the shared lock because after checking the references, the nffg must continue to be stored inside the nffgs map
- storeNffg does not need any locks because acts on completely new data. If the nffg stored with the same name is still being removed but not yet from the map, the serialized view of events will have the store before the deletion, without causing side effects.
- the getters don't require any lock because they read the data from a single map in atomic way. They also don't modify the data so they don't produce side effects.
- getPolicies is the only critical getter, that could give back a partial set of policies because the method removeIf called on the entrySet by the deleteNffg could be iterating and having removed only some policies belonging to the nffg that is being deleted. To avoid this problem, this method also uses the shared lock