OGC Testbed-13

Executable Test Suites and Reference Implementations for NSG WMTS 1.0 and WFS 2.0 Profiles with Extension

Table of Contents

Publication Date: 2018-01-08

Approval Date: 2017-12-07

Posted Date: 2017-11-06

Reference number of this document: OGC 17-043

Reference URL for this document: http://www.opengis.net/doc/PER/t13-NG010

Category: Public Engineering Report

Editor: Nuno Oliveira

Title: OGC Testbed-13: Executable Test Suites and Reference Implementations for NSG WMTS 1.0 and WFS 2.0 Profiles with Extension

OGC Engineering Report

indicate compliance with any LICENSOR standards or specifications.

This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

None of the Intellectual Property or underlying information or technology may be downloaded or otherwise exported or reexported in violation of U.S. export laws and regulations. In addition, you are responsible for complying with any local laws in your jurisdiction which may impact your right to import, export or use the Intellectual Property, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

Chapter 1. Summary

This Engineering Report (ER) describes the development of the compliance tests and implementation in GeoServer of the Web Feature Service (WFS) 2.0 and Web Map Tile Service (WMTS) 1.0 National System for Geospatial Intelligence (NSG) profiles. The NSG of the United States (US) National Geospatial Intelligence Agency (NGA) is the combination of technologies, policies, capabilities, doctrine, activities, people, data and communities needed to produce geospatial intelligence (GEOINT) in an integrated, multi-intelligence, multi-domain environment. The work can be grouped into four main topics:

- critical review of the NSG profiles for WFS 2.0 and WMTS 1.0
- implementation of the profiles in GeoServer
- validation of the implementation using OGC Compliance tests and tools
- lessons learn during the implementation of these profiles and their validation

Both NSG profiles are Class 2 profiles. WMTS profiles OGC WMTS 1.0. WFS profiles the DGIWG Profile of OGC WFS 2.0. The first topic provides a review of these profiles along with a description of the main extensions and restrictions introduced by them.

The second topic covers the implementation of the NSG profiles in GeoServer. It describes the software architecture and technical decisions, along with the deployment and configuration of the server.

The third topic covers the validation process of the implementation using OGC validation (sometimes referred to as CITE) tests and tools. It also covers how the tests can be run and how to configure GeoServer for these tests.

The last topic contains an evaluation of the work, reached goals, lessons learned and the best practices that can be applied in future work.

1.1. Requirements

The requirements addressed by this ER are the implementation of the NSG WFS 2.0 profile and NSG WMTS 1.0 profiles in the latest GeoServer version, as of October 2017, and the validation of the implementation with the corresponding CITE tests.

1.2. Key Findings and Prior-After Comparison

Before testbed 13 there were no implementations of NSG WFS 2.0 and NSG WMTS 1.0 profiles. GeoServer compliance status regarding the base standards WFS 2.0 and WMTS 1.0 was unknown. CITE tests for the NSG profiles didn’t exist and the existing tests for the base standards had to be extended.

After Testbed 13 completion the profiles tests were completed and available in the OGC validation web sites and GeoServer was compliant with the NSG profiles mandatory requirements, including the base standards that the profiles were based on.

1.3. What does this ER mean for the Working Group and OGC in General

The implementation of the NSG WFS 2.0 and NSG WMTS 1.0 profiles will help advance the architecture of profiles and architecture of conformance classes of OGC standards. The lessons learned on the development of the NSG profiles will help clarify aspects of new versions of WFS 2.0 and WMTS 1.0.

The NSG WFS profile is built on top of the DGIWG Web Feature Service 2.0 profile which is based on the WFS 2.0 standard. In light of current OGC efforts leading to more modular specifications, the ability to build composite compliance tests is a key capability. The experience with WFS will be valuable information for the Architecture SWG in particular.

GeoServer is an open source project with a diverse and vibrant community that has been involved in this work. The interaction with this community helped the development of this document and should provide a positive outcome for the OGC involved standards.

At last, having CITE tests suites that allow to test the compliance of the involved OGC standards will help other organizations in adopting them.

1.4. Document Contributor Contact Points

All questions regarding this document should be directed to the editor or the contributors:

Table 1. Contacts

Name	Organization
Nuno Oliveira	GeoSolutions
Simone Giannecchini	GeoSolutions
Luis Bermudez	OGC

1.5. Foreword

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

Chapter 2. References

NOTE Only normative standards are referenced here, e.g. OGC, ISO or other SDO standards. All other references are listed in the bibliography.

The following normative documents are referenced in this document.

- OGC 09-025r2, OpenGIS Web Feature Service 2.0 Interface Standard [https://portal.opengeospatial.org/files/09-025r2]
- OGC 07-057r7, OpenGIS Web Map Tile Service Implementation Standard [http://portal.opengeospatial.org/files/?artifact_id=35326]
- DGIWG 122, DGIWG – Web Feature Service 2.0 Profile, (November 16, 2015) [https://portal.dgiwg.org/files/?artifact_id=11487&format=pdf]
- NGA.STND.0062_1.0_WFS, National System for Geospatial-Intelligence (NSG) Web Feature Service 2.0 Implementation Profile (January 11, 2017) [https://nsgreg.nga.mil/NSGDOC/files/doc/Document/NSG_Web_Feature_Service_2.0_Profile%2020170111.pdf]
- NGA.STND.0063_1.0_WMTS, National System for Geospatial-Intelligence (NSG) Web Map Tile Service 1.0.0 Implementation Interoperability Profile (September 15, 2016) [https://nsgreg.nga.mil/NSGDOC/files/doc/Document/NSG_WMTS_1%200_Implementation_Profile_v4_09142016.doc]
- OGC 09-025r2, OpenGIS Web Feature Service 2.0 Interface Standard [https://portal.opengeospatial.org/files/09-025r2]
- OGC 07-036, OpenGIS Geography Markup Language (GML) Encoding Standard [http://portal.opengeospatial.org/files/?artifact_id=20509]
- OGC 09-026r2, OGC Filter Encoding 2.0 Encoding Standard [https://portal.opengeospatial.org/files/09-026r2]
- OGC 06-121r3, OpenGIS Web Service Common Implementation Specification [http://portal.opengeospatial.org/files/?artifact_id=20040]

Chapter 3. Terms and Definitions

For the purposes of this report, the definitions specified in Clause 4 of the OWS Common Implementation Standard OGC 06-121r9 [https://portal.opengeospatial.org/files/?artifact_id=38867&version=2] shall apply.

3.1. Abstract Test Suite (ATS)

```
a set of testable assertions about the functionality of a standard, which an
implementation must support in order to achieve compliance to the standard. ATS are
based on the conformance clauses defined in the standard. (source: OGC 08-134r10)
```

3.2. Conformance

```
a standard's "abstract conformance" to Standards Packages for that standard (see ISO
19105:2000 Geographic information - Conformance and Testing at
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=26010
).
```

3.3. Compliance

```
a state of a specific software product, which implements an OGC Standard and has
passed the Compliance Testing Evaluation. (source: OGC 08-134r10)
```

3.4. interoperability

```
capability to communicate, execute programs, or transfer data among various functional
units in a manner that requires the user to have little or no knowledge of the unique
characteristics of those units (source: ISO 19119)
```

3.5. profile

```
specification or standard consisting of a set of references to one or more base
standards and/or other profiles, and the identification of any chosen conformance test
classes, conforming subsets, options and parameters of those base standards, or
profiles necessary to accomplish a particular function. (adapted from ISO/IEC TR
10000-1)
```

Chapter 4. Abbreviated Terms

- API Application Programming Interface
- CITE Compliance and Interoperability Testing
- NGS National System for Geospatial-Intelligence
- WFS Web Feature Service
- WMTS Web Map Tile Service
- WMS Web Map Service
- WCS Web Coverage Service
- DGIWG Defence Geospatial Information Working Group
- CTL Compliance Test Language
- TEAM Test, Evaluation, and Measurement
- ABAC Attribute Based Access Control
- IC Intelligence Community
- DoD Department of Defense

Chapter 5. Overview

This Engineering Report (ER) describes the development of the compliance tests and implementation in GeoServer of the WFS 2.0 and WMTS 1.0 National System for Geospatial Intelligence (NSG) profiles.

A profile is a specification or standard consisting of a set of references to one or more base standards and/or other profiles, and the identification of any chosen conformance test classes, conforming subsets, options and parameters of those base standards, or profiles necessary to accomplish a particular function.

It can, for example, make mandatory optional capabilities or define extensions where permitted by the base standard. Standards are organized in conformance class, which are essentially containers for a coherent set of requirements (tests). A profile is based on a core conformance class of a standard and possibly other optional conformance class of the standard.

The NSG WFS 2.0 and NSG WMTS 2.0 profiles both extends and restricts the base specifications. Being compliant with these profiles requires being compliant with all of the mandatory requirements and any optional requirements which are restricted or extended by these profiles. Therefore, a compliance test for a profile must first successfully pass the compliance test for the conformance classes being profiled.

GeoServer passed the profiles tests. The process started first with making GeoServer compliant with the base specifications mandatory requirements. Then it was enhanced to pass the tests for each NSG profile.

The CITE tests suites for the NSG WFS 2.0 profile have a fairly good coverage. NSG WFS 2.0 profile depends on DGIWG Web Feature Service 2.0 Profile. The DGIWG profile depends on the WFS 2.0 test.

5.1. State of the Art

GeoServer [1] is a free open source project designed for geospatial data interoperability and is an implementation of a number of open standards such as WFS, Web Map Service (WMS) and Web Coverage Service (WCS).

CITE tests suites are part of the GeoServer development process, relevant CITE tests suites are run automatically each day against the supported series. If a test fails it is considered to be a release blocker. This ensures that compliance with the different specifications is not broken.

This is the first time GeoServer compliance against WFS version 2.0 and WMTS versions 1.0 were tested within OGC programs. In a preliminary analysis several issues were found, mainly related to the WFS 2.0 specification, that needed to be addressed.

CITE tests suites relevant for these profiles are implemented in TestNG and CTL, TestNG is being used for developing new tests suites. CITE tests are run using TEAM Engine Validator. OGC Validation tools are free, open source and available from GitHub.

NSG WFS 2.0 CITE tests depend on WFS 2.0 CITE tests, running NSG WFS 2.0 CITE tests will also run the WFS 2.0 CITE tests. NSG WMTS 1.0 CITE tests will only implement the requirements that come directly from the NSG WMTS profile. WMTS 1.0 CITE tests will need to be run separately.

5.2. Objectives

The main objective of this work was to implement the NSG profiles in GeoServer and the necessary CITE test suites to validate the implementation.

A secondary objective is to make GeoServer compliant with WFS 2.0 and WMTS 1.0 and to introduce the correspondent CITE tests suites as part of the daily checks.

Chapter 6. NSG Profiles Critical Review

When standards reach a certain degree of complexity they tend to start classifying some of their capabilities as mandatory, optional or recommended. This mechanism allows standards to introduce advanced capabilities while at the same time keeping the standards viable to implement and providing a clear definition of said features. New OGC standards organize the capabilities into conformance classes that help implementers understand a set of group requirements that can be implemented or not as a whole.

Several mechanisms exist to allow clients to understand which capabilities of a certain standard are actually implemented by a specific software. A mechanism common to OGC standards is the capabilities documents, which are documents that are generated by implementations that clearly define which capabilities of the implemented standard are supported.

Certain organizations or communities have specific needs and might have to push the boundaries of an existing specification with new capabilities. Instead of creating a fork of the original standards, those organizations or communities can create a profile clearly defining their expectations in terms of existing capabilities and new ones.

Profiles are a very powerful mechanism that gives the necessary flexibility and modularity to extend and/or restrict existing profiles while promoting the reutilization of a common base standard.

NSG WFS 2.0 and NSG WMTS 1.0 profiles are Class 2 profiles, which means that they extend by introducing new capabilities and restrict the base specifications which are WFS 2.0 and WMTS 1.0 respectively. They also depend on other standards and profiles.

Profiles also have an impact on implementations, in particular generic servers such as GeoServer. Generic servers need to develop modular support for the profiles. Usually, the core of the software implements the base standard while the profiles are made available as plugins.

6.1. Developing Profiles on Top of Other Profiles

From the profile implementor point of view, profiles make a good job in making clear which requirements from the sub profiles or standards are restricted or made mandatory. The summary table of the NSG WMTS profile is a significant help during implementation.

Something that could be improved in the profiles is the connection between concepts, i.e. when two concepts are connected or when a concept depends on another one. For example, the *PageResults* operation descriptions would benefit in having an explicit link to the *index* result type description. The same is valid for feature versioning and resource identifiers.

A profile always has one or more dependencies that must be heeded when developing an implementation or a conformance test suite. For example, a test suite for any WMS profile will include tests covering one or more conformance classes in the OGC WMS conformance test suite. The DGIWG profile requires conformance at the "Queryable WMS" level. The associated test suite also includes tests that cover the specific requirements of that profile.

A test suite for a profile generally selects one or more conformance classes (or levels) from the relevant set of base specifications. These tests then implicitly become part of the dependent test suite. The base tests are invoked in the course of running the profile-specific tests.

The source code can be included directly. The way in which the code is included depends on how a test suite was implemented (TestNG or CTL). The type of the base test suite determines the type of the profile test suite. If the base test suite is implemented in TestNG, the profile test suite must also be implemented in TestNG. The same applies to CTL.

A test suite that covers the requirements of an application profile is accessed and executed just like any other test suite. It appears in the listing of available suites, and it can be selected and run in the same manner.

6.2. NSG WFS 2.0 Profile

The main goal of this profile is to promote interoperability within the GEOINT communities (defense agencies, NATO, coalition partners, etc.) This profile is built on top of DGIWG Web Feature Service 2.0 profile and its base specification is WFS 2.0. This profile has also a strong dependency on these standards: GML 3.2.1 [OGC 07-036], Filter 2.0.2 [OGC 09-026r2] and WSC 1.1 [OGC 06-121r3].

As stated on the profile, the extensions and restrictions introduced are designed to achieve the following:

- Maintain compatibility with OGC WFS 2.0 and Web Feature Service 2.0 profile
- Integrate with other Intelligence Community (IC) and Department of Defense (DoD) standards for Discovery and Retrieval
- Support for the IC and DoD Attribute-Based Access Control (ABAC) infrastructure
- Support for time-based versioning of content

The following operations are supported in the profile:

- GetCapabilities
- GetPropertyValue
- GetFeature
- DescribeFeatureType
- ListStoredQueries
- DescribeStoredQueries
- LockFeature
- GetFeatureWithLock
- Transaction
- CreateStoredQuery
- DropStoredQuery
- PageResults

The mandatory service bindings for these operations are HTTP GET / POST, and SOAP is optional.

The only mandatory features encoding is GML 3.2. Service exception reports should use the defined exception codes and correctly identify operations. The profile also introduces a mandatory response timeout functionality, which will be extensively discussed in the implementation chapter.

Capabilities documents should clearly identify in the service identification metadata section which profile is being used, the predefined statement should be used. Specific metadata related with access constraints and service constraints need also to be added.

Feature versioning support is mandatory and must be based on date and time. It must also be compatible with the filter encoding standard, allowing queries to filter the features based on their version.

The new *PageResults* operation introduces a better support for pagination by allowing random access of the paginated results. This operation depends on the new introduced *index* result type.

6.2.1. Random Pagination Access

NSG WFS profile (page 63 section 8.5) introduces a new operation for WFS 2.0.2 named *PageResults*. This operation will allow clients to access paginated results using random positions, instead of following next/previous links as the base WFS 2.0 specification suggests. To support this operation the NSG profile (page 41 section 7.6.4) introduces also a new result type named *index*.

Current Pagination Support

The current WFS 2.0.2 OGC specification 2 defines a basic pagination support (page 29 section 7.7.4.4) that can be used to navigate through features responses results. Pagination is activated when parameters *count* and *startIndex* are used in the query, for example:

```
http://<base_url>?service=WFS&version=2.0.0&request=GetFeature&typeNames
=topp:tasmania_roads&count=5&startIndex=0
```

In this case each page contains five features. The returned feature collection has the next and previous attributes allowing clients to navigate through the results pages, i.e. previous page and next page:

```
<wfs:FeatureCollection>
    previous="http://<base_url>?REQUEST=GetFeature&
    VERSION=2.0.0&
    TYPENAMES=topp:tasmania_roads&
    SERVICE=WFS&
    COUNT=2 &
    STARTINDEX=0"
    next="http://localhost:8080/geoserver/wfs?
    REQUEST=GetFeature&
    VERSION=2.0.0&
    TYPENAMES=topp:tasmania_roads&
    SERVICE=WFS&
    COUNT=2 &
    STARTINDEX=4"
  numberMatched="14"
  numberReturned="2"
```

The client cannot assume anything about the previous and next attributes URLs, each server is free to implement its own pagination URL scheme. This implies a sequential navigation, if the client is showing page two and the user wants to see page five, the client will have to:

1. request page three and use the provided next URL to retrieve page four
2. request page four and use the provided next URL to retrieve page five

This is far from an ideal solution to access random pages, which is quite a common action. *PageResults* operation improves on this by allowing clients to request random pages directly.

Index Result Type

The WFS 2.0 *resultType* parameter (page 21 section 7.6.3.6) can be used with WFS *GetFeature* to control the content returned to the client. The possible values in the WFS 2.0 core specification are:

- results
- hits

The *results* value makes the *GetFeature* operation return the features read from the underlying data source. The *hits* value makes *GetFeature* operation only return the count of the features matched by the query, this is the same behavior of an SQL count.

If the *resultType* parameter is omitted *GetFeature* defaults to *results*. Here is an example of a *GetFeature* request that uses the *resultType* parameter:

```
http://<base_rul>?service=WFS&version=2.0.0&request=GetFeature
&typeNames=topp%3Atasmania_roads&resultType=hits
```

The response of the *GetFeature* request above looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<wfs:FeatureCollection
  numberMatched ="14"
  numberReturned ="0"
  timeStamp="2017-08-02T13:08:04.185Z"
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengis.net/wfs/2.0/>
```

The *index* result type, introduced by the WFS NSG profile, extends the WFS *hits* result type by adding an extra attribute named *resultSetID* to the response. The *resultSetID* attribute can then be used by the *PageResults* operation to perform random access through the results.

A *GetFeature* request using the index result type looks as follows:

```
http://<base_url>?service=WFS&version=2.0.0&request=GetFeature
&typeNames=topp%3Atasmania_roads& resultType=index
```

The response of the *GetFeature* request above is:

```
<?xml version="1.0" encoding="UTF-8"?>
<nsg:FeatureCollection
  numberMatched="14"
  numberReturned="0"
  resultSetID ="ef352924-77a0-11e7-b5a5-be2e44b06b34"
  timeStamp="2017-08-02T13:08:04.185Z"
  xmlns:nsg="http://www.opengis.net/nsg/2.0"/>
```

The *resultSetID* is a unique identifier that identifies the result set produced by the original request. Clients use the *resultSetID* with the *PageResults* operation to reference the original result set. This implies that the server needs to keep track of the original result set in some way. For example, by dumping the full result for later usage, or storing the original request in order to re-run it later.

Note that if pagination is used, the existing behavior is preserved, i.e. the previous and next attributes should appear as needed:

```
<?xml version="1.0" encoding="UTF-8"?>
<nsg:FeatureCollection
  numberMatched="14"
  numberReturned="0"
  resultSetID ="ef352924-77a0-11e7-b5a5-be2e44b06b34"
  next ="http://<base_url>?REQUEST=GetFeature&amp;RESULTTYPE=hits&amp;
  VERSION=2.0.0&amp;TYPENAMES=topp%3Atasmania_roads&amp;SERVICE=WFS&amp;
  COUNT=2&amp;STARTINDEX=4"
  previous ="http://<base_url>?REQUEST=GetFeature&amp;RESULTTYPE=hits&amp;
  VERSION=2.0.0&amp;TYPENAMES=topp%3Atasmania_roads&amp;SERVICE=WFS&amp;
  COUNT=2&amp;STARTINDEX=0"
  xmlns:nsg="http://www.opengis.net/nsg/2.0">
```

PageResults Operation

The *PageResults* operation allows clients to query random positions of an existing result set identified by a *resultSetID* that was previously created using the *index* result type. The available parameters are these ones:

Table 2. PageResults operation parameters

Name	Mandatory	Default Value
service	yes	
version	yes	
request	yes	
resultSetID	yes	
startIndex	no	0
count	no	10
outputFormat	no	application/gml+xml; version=3.2
resultType	no	results
timeout	no	300

A typical *PageResults* request looks as follows:

```
http://<host>/geoserver/ows?service=WFS&version=2.0.2&request=PageResults&
resultSetID=ef352924-77a0-11e7-b5a5-be2e44b06b34&startIndex=5&count=10
&outputFormat=application/gml+xml;%20version=3.2&resultType=results
```

Note that this is a *GetFeature* request where the query expression has been replaced by the *resultSetID* parameter.

6.2.2. Features Versioning

The NSG WFS profile restricts the notion of versioning to a time-based versioning approach. The main use case is based on the interest of clients that are only interested in the changes that happened since their last update.

This versioning mechanism assumes the stored vector data is being updated continuously without any synchronization between the involved systems. Clients can update the data using WFS at any time while others can ask for the data that has been updated since a given time stamp.

Feature versioning is strongly related with resource identifiers. Since the same feature may appear multiple times with different versions, a way to uniquely identify each version of the feature is needed.

The NSG WFS profile defines two types of resource identifiers:

- Instance Resource Identifier
- Entity Resource Identifier

The *Instance Resource Identifier* uniquely identifies each version of a certain feature. When a feature is encoded in GML the instance identifier should be encoded in the *identifier* attribute of the *AbstractGML* element. This is the identifier that should be used by the *GetResourceByID* operation.

The *Entity Resource Identifier* identifies the entity that is represented by the feature. Multiple versions of an entity should have the same entity identifier. When a feature is encoded in GML the entity identifier should be encoded in the *name* attribute of the *AbstractGML* element.

6.3. NSG WMTS 1.0

The base specification of this profile is WMTS 1.0 whose base operations are:

- GetCapabilities
- GetTile
- GetFeatureInfo

This profile requires that a REST interface must be exposed, HTTP GET /POST methods are also mandatory and SOAP support is optional. Any possible order of the variables and values in the URL template is valid but a specific order is recommended.

Coordinate systems CRS:84 WGS84 and EPSG:4326 WGS84 must be supported, as well as projections EPSG:3395, EPSG:5041 and EPSG:5042. Scales for World Mercator EPSG:3395 defined in Annex B of the profile must also be supported.

Regarding responses MIME types, image formats *image/png*, *image/jpeg* and *image/gif* must be supported and GetFeatureInfo request must support *text/xml*, *text/html* and *application/gml+xml; version=3.2* as output formats.

Caching information (expiration data) for the data must be provided using HTTP control headers.

Some specific meta-data and keywords as defined in the profile must also to be provided through the capabilities document.

6.3.1. RESTful API

WMTS RESTful API only supports the HTTP GET method, allowing clients to retrieve the following resources:

- capabilities document
- tile
- feature info

Clients parse the capabilities document to discover how to invoke the RESTful API. The specification doesn’t define any resource path or query parameters, each implementation is free to use the paths and query parameters they want. For tile resources and feature info resources, the paths need to be defined in the capabilities document using a template language with some mandatory terms.

This decoupling between the RESTful API definition and the actual implementation gives more freedom to each WMTS implementation to choose its technology stack. For example, simple use cases can be implemented using a basic HTTP server providing static content, taking full advantage of specific operating system optimizations.

For more complex uses cases, for example supporting a generic number of dimensions for each layer, or working against dynamic data, a dynamic implementation needs to be used.

Chapter 7. Implementing NSG Profiles Using GeoServer

GeoServer [1] is a free open source project designed for geospatial data interoperability and is an OGC compliant implementation of a number of standards such as WFS, WMS and WCS. GeoServer is built on top of the GeoTools [2] library and integrates natively with GeoWebCache [3] providing a flexible and easy to use tile cache mechanism. GeoWebCache implements several standards including the WMTS service.

In GeoServer, modularity and flexibility are first class citizens, with the default package caring for most common needs, and extension supporting a variety of other functionality. This includes the most common data stores such as Shapefile, GeoPackage, PostgreSQL and GeoTIFF as well the most common used OGC services as such as WFS, WMS and WMTS.

GeoServer offers a large amount of extension points allowing anyone to add new functionality in a modular way. Some of those extensions are contributed back to GeoServer becoming official GeoServer plug-ins: currently around 80 plug-ins are available for GeoServer. These plug-ins extend GeoServer in a variety of ways: adding support for new data sources, adding new services or extending existing ones, adding new security methods, and so on.

Plug-ins can be divided in two main categories: community-modules and extensions. Community modules are generally considered experimental in nature and can be undergoing significant development. Once a community module development is stable and has proven to be useful it can be promoted to an extension. To become an extension a community module needs also an official maintainer, pass the code quality requirements and have an official documentation. If an extension is judged to be important enough or is commonly used, it can be promoted to a GeoServer core module.

CITE tests suites are part of the GeoServer development process. Relevant CITE tests suites are run automatically each day against GeoServer supported versions: a CITE test failure is considered a release blocker. This ensures that compliance with the different specifications is not broken.

As discussed in the previous section NSG WFS and NSG WMTS profiles are built on top of other profiles and WFS 2.0 and WMTS 1.0 standards, respectively. WFS is implemented by GeoServer as a core service and WMTS is implemented by GeoWebCache as a core service. As already said, GeoServer integrates natively in a transparent way with GeoWebCache.

All capabilities made mandatory by the NSG profiles that are defined in the base standards were directly implemented in GeoServer and GeoWebCache. New capabilities or restrictions defined by the profile are being implemented in two community modules: *nsg-wfs-profile* and *nsg-wmts-profile*.

7.1. Time Versioning

Versioning of data is a vast subject with a great number of techniques and algorithms. The NSG WFS profile makes it easier by focusing only on time versioning and by providing a definition of a resource identifier compatible with this versioning technique.

For each feature, the system should keep information about when the feature was first created and for each subsequent modification. It should also be possible to identify each feature uniquely and the entity represented by each feature. An entity may be represented by multiple features in different versions.

In practice, this means that the server internally needs to associate each feature with a timestamp and an entity identifier. An important question arises related to how that association should be stored by the server: Should the data schema already support this information as two attributes (e.g. columns in a relational database) or should they be stored separately?

The main drawback of storing that extra data separately is that it makes the system more complex, requiring a join of the different elements at runtime and making the management of the data more difficult. If a relational database is used as storage, the service implementation would need permissions to create a new table or the database administrator would need to create it explicitly.

If the data schema already accounts for these two attributes, then the administrator has to specify which attribute should be used for the timestamp and which one identifies the entity represented by a certain feature. This approach was chosen in the GeoServer implementation, the choice of the attributes is proposed on the administration UI during the layer configuration.

The WFS NSG profile versioning requirement is built on top of a very specific use case: a data set can be continuously updated while clients retrieve the data that was updated since their last check. The OGC WFS 2.0 standard is not explicit about the semantics of the *create*, *update* and *delete* operations. The NSG profile already provides several details which are absent from the OGC WFS 2.0 standard, it may be worth also clarifying the semantics of these operations. Taking into account the context of the requirement, the following semantics were assumed:

- a *create* operation should create a new feature with the associated timestamp
- an *update* operation should create a version of the target entity based on the most recent version, unless no feature can be found for the targeted entity
- a *delete* operation should remove the targeted feature

By requesting the delta between the last check and the current time, the client will be able to understand which features have been updated but not which ones have been deleted.

Detecting which features have been deleted since the last check can be done by performing two requests with consequent time envelopes and then comparing the results to detect the deleted features. This solution is impracticable with big data sets, because in practice the whole data set will need to be requested each time we want to detect deleted features.

A sensible approach to this problem would be the possibility of the client to know which operation (e.g. *create*, *update* or *delete*) led to the status of a particular feature or the ability to filter features by the operation that changed their status (e.g. give me all the features that where created or deleted in the last hour).

7.2. Random Pagination

Random pagination is the ability to perform a request to the server and index the result allowing clients to request parts of that result in any order. A common doubt when implementing pagination is how to deal with concurrent changes, i.e. changes to the data that occur after the initial pagination request.

Consider the following data:

Table 3. Example data

Name	Age
Thoma	16
John	17
James	20
Robert	22
Charles	23
Paul	26
Mark	28

A client makes a pagination request to the server asking for all the persons that are older than 18 years sorted by their age. Then the client starts viewing the results starting in index zero with a pagination of two:

Table 4. First page

Name	Age
James	20
Robert	22

Table 5. Second page

Name	Age
Charles	23
Paul	26

Table 6. Third page

Name	Age
Mark	28

Now consider that the user is currently viewing the first page and that in the meantime *Robert* is removed from the dataset. Then the user wants to see page two, so the client requests that page. Depending on the pagination algorithm the client can see these two different results:

Table 7. Second page, result 1

Name	Age
Charles	23
Paul	26

Table 8. Second page, result 2

Name	Age
Paul	26
Mark	28

The first result corresponds to algorithms that are capable of identifying the dataset elements that remain unchanged since the initial pagination request hit the server. These types of algorithms are quite complex and only possible to implement with the help of the data store and only work for very specific use cases.

Implementing this type of algorithms for a generic dataset would require the capability of storing the result set that corresponds to an initial pagination request and then iterate over that result set. Pagination is only really needed when requests return long results. Storing those request results, even in a compact format, means replicating significant portions of the original dataset in the server. This will allow clients to take down a server with a few requests.

The second result corresponds to algorithms that don't store the initial pagination request result set but instead store the original pagination request. When a client requests a certain page starting at a certain index, the server executes the pagination request using the provided limits, i.e. start index a page size. The issue with this type of algorithms is that like in the example the user may miss some elements, or get the same element twice.

It should also be considered that algorithms of the first type may show invalid results. Let's assume removing *Charles* instead of *Robert*. When the client requests the second page, depending on the pagination algorithm the client receives one of the two different results:

Table 9. Second page, result 3

Name	Age
Charles	23
Paul	26

Table 10. Second page, result 4

Name	Age
Paul	26
Mark	28

Algorithms of the first type will still show *Charles* even if that user was already removed, but algorithms of the second type will correctly not show that person.

Let's say the perfect pagination algorithm is implemented, that would basically require a versioning mechanism keeping track of all the data changes and be able to return the correct portion of the data that corresponds to a certain page.

When missing a result is not that acceptable, there are several interesting approaches mixing algorithms of the second type and user experience patterns.

A common and simple approach is to tell the user that the page he is viewing has been updated and to update that page content, then the user will not miss an existing element or see outdated ones. This approach is common in certain mail clients, for example. The simplest way for a client to detect that the page content show to the user is outdated is to request the same page a check for differences.

Independently of the followed approach, both store some content on the server (either the full result set, or the definition of the original query and its timestamp). The profile doesn't provide or define any mechanism to discover when that content is not needed anymore and can be removed. The implementation should implement a configurable mechanism cleaning any information related with a paginated request that has not been used since a certain amount of time.

7.3. WMTS RESTful API

The WMTS service capabilities documents contains several XML elements allowing web clients to invoke the WMTS RESTful API. Three types of resources can be retrieved using the RESTful API:

- The WMTS capabilities document
- Tiles
- Features information

The end-point that can be used to retrieve the capabilities document is static for each GeoServer instance and looks as follows:

```
http://<base_url>/WMTSCapabilities[.xml]
```

The retrieved capabilities document needs to contain the top-level element (direct child of the root element) `<ServiceMetadataURL>` that tells clients what is the WMTS capabilities document resource URI, it shall look like this:

```
<ServiceMetadataURL xlink:href="<base_url>/WMTSCapabilities.xml"/>
```

Each WMTS published layer needs to have several `<ResourceURL>` elements that define the resources associated to that layer that can be retrieved through the RESTful API. The available resources can be divided into two categories: tiles (GetTile) and feature info (GetFeatureInfo).

The tile resources `<ResourceURL>` elements shall look as follows, note that the `<ResourceURL>` element needs to be repeated for each supported image format of the layer:

```
<ResourceURL format="<imageformat>" resourceType="tile"
  template="<base_url>/<layer>/{style}/{TileMatrixSet}/{TileMatrix}/
  {TileRow}/{TileCol}?format=<imageFormat>&
  <firstDimensionName>={firstDimensionName}&
  <lastDimensionName>={lastDimensionName}">
```

The layer dimensions (time, elevation, etc.) support is a bit tricky, since it is not possible to predict the dimensions a layer will have and which ones the client will like to use. So the resource template should make available all the layers dimensions as query parameters and is up to the client to set the values for the dimensions he wants to use.

Consider a layer named temperature. It has two dimensions: time and elevation. It supports PNG and JPEG image formats. It has several styles and tile matrix sets. The `<Layer>` element corresponding to the temperature layer in the capabilities document shall then contain these two `<ResourceURL>` children:

```
<ResourceURL format="image/png" resourceType="tile"
  template="<base_url>/temperature/{style}/{TileMatrixSet}/{TileMatrix}/
  {TileRow}/{TileCol}?format=image/png&time={time}&elevation={elevation}">
<ResourceURL format="image/jpeg" resourceType="tile"
  template="<base_url>/temperature/{style}/{TileMatrixSet}/{TileMatrix}/
  {TileRow}/{TileCol}?format=image/jpeg&time={time}&elevation={elevation}">
```

The requests sent by clients may look as follows:

```
http://<base_url>/temperature/default/WholeWorld_CRS_84/30m/4/5?format=image/png&
time=2016-02-23T03:00:00.000Z&elevation=500
```

Note, that only the *format* query parameter is mandatory. The client may choose to not use the dimensions query parameters.

The feature info resources `<ResourceURL>` element shall look like this, note that the `<ResourceURL>` element shall be repeated for each supported feature info format of the layer:

```
<ResourceURL format="<featureInfoFormat>" resourceType="FeatureInfo"
  template="<base_url>/<layer>/{style}/{TileMatrixSet}/{TileMatrix}/{TileRow}/

{TileCol}/{J}/{I}?format=<featureInfoFormat>&<firstDimensionName>={firstDimensionName}
  &<lastDimensionName>={lastDimensionName}">
```

Feature info `<ResourceURL>` elements are very similar to the tile resources ones. Layer dimensions are handled in the same way dimensions are handled for tile resources.

Consider a layer named temperature. It has two dimensions: time and elevation. It supports HTML and XML feature info formats. It has several styles and tile matrix sets. The `<Layer>` element corresponding to the temperature layer in the capabilities document shall then contain these two `<ResourceURL>` children:

```
<ResourceURL format="text/html" resourceType="FeatureInfo"
  template="<base_url>/temperature/{style}/{TileMatrixSet}/{TileMatrix}/{TileRow}/
  {TileCol}/{J}/{I}?format=text/html&time={time}&elevation={elevation}">
<ResourceURL format="text/xml" resourceType="FeatureInfo"
template="<base_url>/temperature/{style}/{TileMatrixSet}/{TileMatrix}/{TileRow}/
  {TileCol}/{J}/{I}?format=text/xml&time={time}&elevation={elevation}">
```

The requests sent by clients may look as follows:

```
http://<base_url>/temperature/default/WholeWorld_CRS_84/30m/4/5/23/35?format=text/html
&
time=2016-02-23T03:00:00.000Z&elevation=500
```

Note that, as similar as requesting tiles resources, only the *format* query parameter is mandatory, the client may choose to not use the dimensions query parameters.

The NSG WMTS profile suggests to use the following order for template variables:

> Any possible order of the variables and values in the URL template is valid. Nevertheless, recommend the following order: style, firstDimension, …, lastDimension, TileMatrixSet, TileMatrix, TileRow, TileCol, J and I.

The template variables order used by the implementation is different:

```
.../{style}/{TileMatrixSet}/{TileMatrix}/{TileRow}/{TileCol}/{J}/{I}?format=text/html&
time={time}&elevation={elevation}
```

The implementation provides the format and dimensions as query parameters. GeoServer allows users to associate a layer with a time dimension, elevation dimension or a custom dimension. It is not possible to predict how each value of a dimension will be represented, hence encoding the dimension values using the profile suggested order makes the handling difficult to implement on the server side, and prone to interpretation errors in the URL.

Consider a layer having a time dimension. Instead of querying a specific time, GeoServer supports the common use case of requesting a time range. Also, the data can have features that have an interval of validity instead of a single point in it. An interval is expressed as follows:

```
2002-09/2002-12
```

If using the templates variables suggested order, the following requests would be valid for a layer with an elevation and time dimensions:

```
http://<base_url>/default/2016-02-23T03:00:00.000Z/500/WholeWorld_CRS_84/30m/4/5/23/35
http://<base_url>/default/2002-09/2002-12/500/WholeWorld_CRS_84/30m/4/5/23/35
```

It will be very difficult to implement a URL matching pattern on the server side that would be generic enough to handle all the possible time dimension request combinations. It will not cover all the possible use cases. A human being looking at the URL would also be rather confused about the meaning of the URL.

The WMTS standard is not clear if or when a client can ignore a template variable. The standard only says that the client should substitute any template variable with the correct value. In some situations, an empty value can be considered a correct value. A possible approach is to make template variables that belong to the base path mandatory and query parameters optional.

As already discussed before, the RESTful API template mechanism allows a decoupling between the implementation and the RESTful API definition which gives a welcomed flexibility to implementors. The main drawback of this mechanism is that it adds an extra complexity to clients. Clients need to parse the capabilities documents and build resources URI using the templates.

Another issue is that since the resource format is not a valid template variable, a new resource element needs to be added to the capabilities document for each format supported by a certain resource. For example, GeoServer supports several tile formats so for each layer the following XML elements will be added:

```
<ResourceURL format="image/gif" resourceType="tile" template="
  <base_url>/rest/wmts/rastertestlayer/{style}/{TileMatrixSet}/
  {TileMatrix}/{TileRow}/{TileCol}?format=image/gif"/>
<ResourceURL format="image/jpeg" resourceType="tile" template="
  <base_url>/rest/wmts/rastertestlayer/{style}/{TileMatrixSet}/
  {TileMatrix}/{TileRow}/{TileCol}?format=image/jpeg"/>
<ResourceURL format="image/png" resourceType="tile" template="
  <base_url>/rest/wmts/rastertestlayer/{style}/{TileMatrixSet}/
  {TileMatrix}/{TileRow}/{TileCol}?format=image/png"/>
<ResourceURL format="image/png8" resourceType="tile" template="
  <base_url>/rest/wmts/rastertestlayer/{style}/{TileMatrixSet}/
  {TileMatrix}/{TileRow}/{TileCol}?format=image/png8"/>
<ResourceURL format="text/plain" resourceType="FeatureInfo" template="
  <base_url>/rest/wmts/rastertestlayer/{style}/{TileMatrixSet}/
  {TileMatrix}/{TileRow}/{TileCol}/{J}/{I}?format=text/plain"/>
<ResourceURL format="text/html" resourceType="FeatureInfo" template="
  <base_url>/rest/wmts/rastertestlayer/{style}/{TileMatrixSet}/
  {TileMatrix}/{TileRow}/{TileCol}/{J}/{I}?format=text/html"/>
<ResourceURL format="application/vnd.ogc.gml" resourceType="FeatureInfo" template="
  <base_url>/rest/wmts/rastertestlayer/{style}/{TileMatrixSet}/
  {TileMatrix}/{TileRow}/{TileCol}/{J}/{I}?format=application/vnd.ogc.gml"/>
```

Since the available formats are published in the capabilities document, the format could become a template variable allowing to reduce the impact of the RESTful API in the capabilities document.

7.4. WFS Timeout

The NSG WFS profile introduces the *timeout* parameter that can be used to control the maximum time spent by WFS to process a request. If the timeout period expires, the server should cancel the request and send an exception report to the client with the appropriate code.

This implies that the *timeout* parameter only controls the time the server spends producing the result but not the time the server spends sending the result to the client. Once the server starts sending the results to the client it is not possible to cancel the request and send an exception report, canceling at this stage would give the client an incomplete/corrupt document but no exception code.

GeoServer typically follows a lazy approach when processing data. The data is only retrieved when needed and if possible (when the output format allows it) the result is built and streamed to the client in parallel. This gives a significant performance boost in terms of response time and decreases the server resources usage. It is a common pattern in server implementations.

For example, let's consider a client that executed a WFS *GetFeature* operation requesting as output a GeoPackage file. GeoServer parses the WFS request, creates a SQLite database on the file system, starts reading from the source and in parallel writes the elements in the GeoPackage. Once the GeoPackage file is ready GeoServer sends it to the client. In this case the *timeout* parameter will control a significant portion of the execution because the result is completely created on the server side and then send to the client.

Let's suppose a GML document was requested instead. In this case GeoServer starts reading from the source and in parallel encode the elements in GML and stream them to the client. This means that the execution of the query (in a relational database for example) and the streaming of the result to the client happen in parallel. The consequence of this is that the *timeout* parameter only controls a very small portion of the request execution.

The alternatives allowing cancellation for this last use case are:

- not use the "reading, encoding and streaming in parallel" optimization, adding significant extra load on the server (not to mention potential server file system exhaustion)
- make the exception report optional and allow the server to cancel the streaming to the client

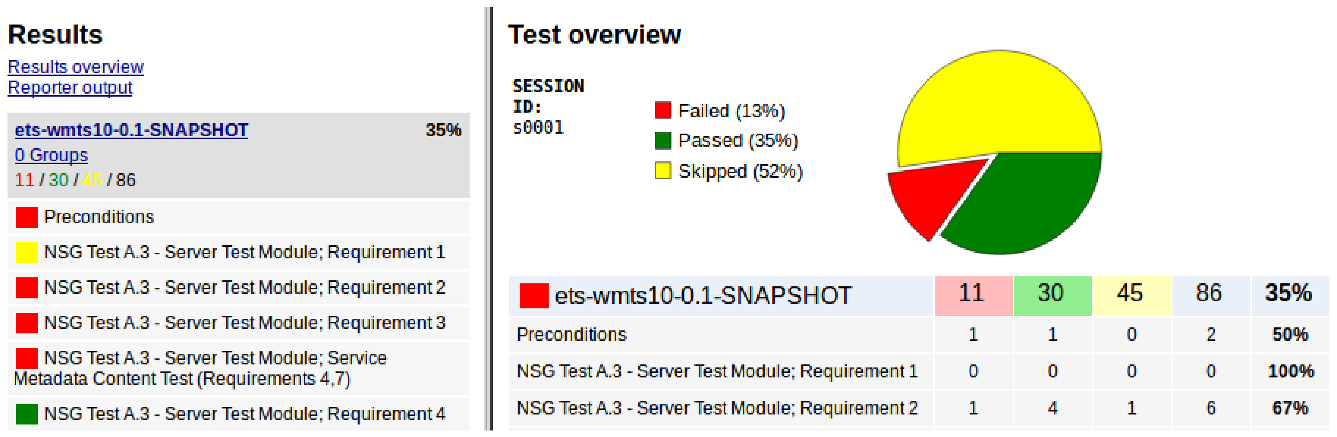
Another issue with the WFS timeout parameter is that there is no reliable way to test this capability, i.e. the tests have no reliable way of forcing an operation to execute for longer than the configured timeout.

Chapter 8. Working With CITE Tests

The CITE tests user guide [1] documents several ways of executing CITE tests. During the implementation of these profiles GeoServer implementers followed two approaches:

- running the tests from the Web interface using a local installation of the TEAM Engine
- running the tests from the integrated development environment (IDE)

The tests were run from the Web interface by installing a local version of the TEAM Engine. Then the CITE tests were installed and configured. The main advantage of this approach is that once the tests have been executed the detailed report can be easily accessed. The following image shows a report that was obtained during the implementation of the NSG WMTS profile. It is very easy to understand and identify which requirements tests were passing and which ones were failing:



Results

Results overview
Reporter output

ets-wmts10-0.1-SNAPSHOT 0 Groups 11 / 30 / 45 / 86	35%
🟥 Preconditions	
🟨 NSG Test A.3 - Server Test Module; Requirement 1	
🟥 NSG Test A.3 - Server Test Module; Requirement 2	
🟥 NSG Test A.3 - Server Test Module; Requirement 3	
🟥 NSG Test A.3 - Server Test Module; Service Metadata Content Test (Requirements 4,7)	
🟩 NSG Test A.3 - Server Test Module; Requirement 4	

Test overview

SESSION ID: s0001

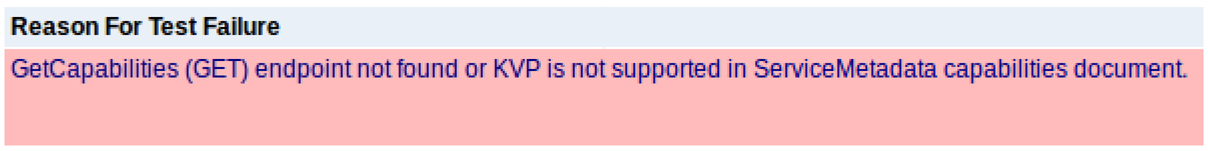
🟥 Failed (13%)
🟩 Passed (35%)
🟨 Skipped (52%)

🟥 ets-wmts10-0.1-SNAPSHOT	11	30	45	86	35%
Preconditions	1	1	0	2	**50%**
NSG Test A.3 - Server Test Module; Requirement 1	0	0	0	0	**100%**
NSG Test A.3 - Server Test Module; Requirement 2	1	4	1	6	**67%**

Figure 1. Partial print screen of a TEAM Engine test session execution report.

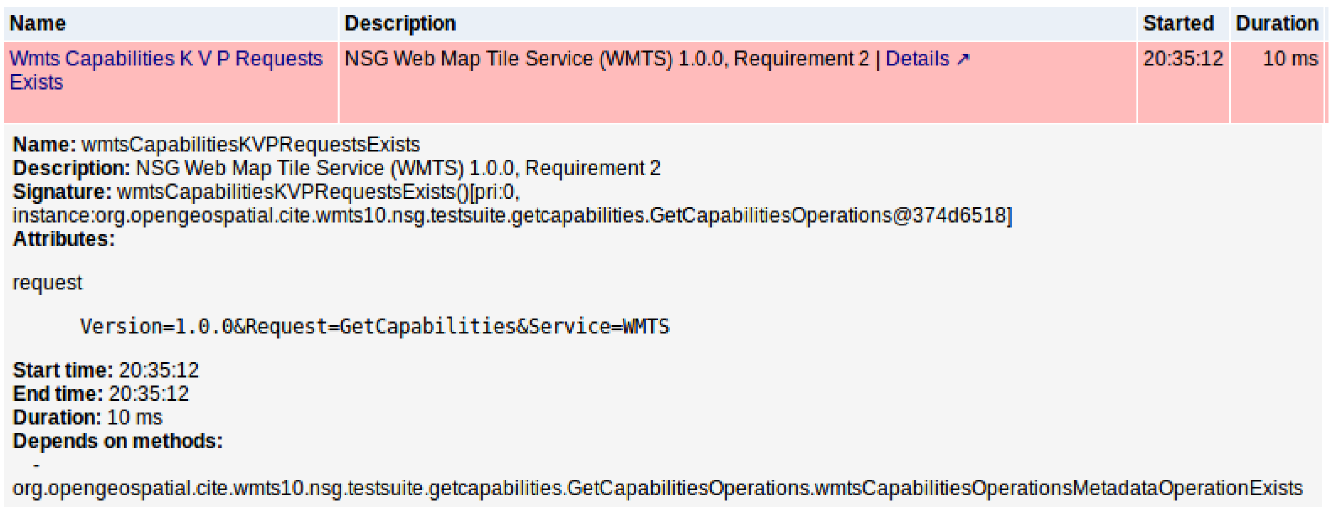
In Figure 1 we can see that several tests were skipped, this happens when the preconditions necessary to run a certain test were not met. There are several types of preconditions, for example, requiring that a certain test is successful or requiring a specific dataset to be configured in the server.

Clicking on a failed test, will show the tests failure information:



Reason For Test Failure
GetCapabilities (GET) endpoint not found or KVP is not supported in ServiceMetadata capabilities document.

Figure 2. Test failure reason as shown by the TEAM Engine test session execution report.



Name	Description	Started	Duration
Wmts Capabilities K V P Requests Exists	NSG Web Map Tile Service (WMTS) 1.0.0, Requirement 2 | Details ↗	20:35:12	10 ms

Name: wmtsCapabilitiesKVPRequestsExists
Description: NSG Web Map Tile Service (WMTS) 1.0.0, Requirement 2
Signature: wmtsCapabilitiesKVPRequestsExists()[pri:0, instance:org.opengeospatial.cite.wmts10.nsg.testsuite.getcapabilities.GetCapabilitiesOperations@374d6518]
Attributes:

request

```
Version=1.0.0&Request=GetCapabilities&Service=WMTS
```

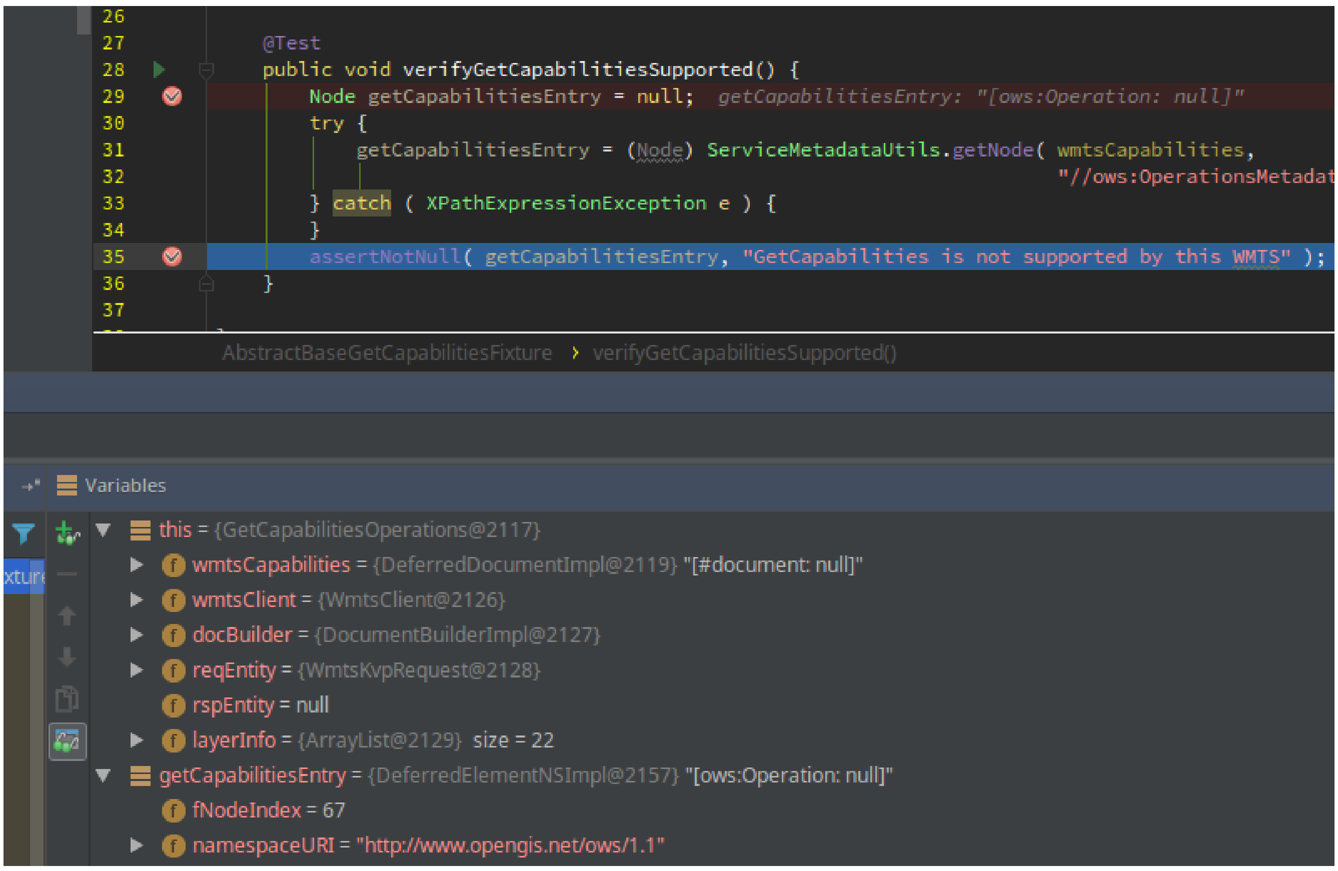
Start time: 20:35:12
End time: 20:35:12
Duration: 10 ms
Depends on methods:

- org.opengeospatial.cite.wmts10.nsg.testsuite.getcapabilities.GetCapabilitiesOperations.wmtsCapabilitiesOperationsMetadataOperationExists

Figure 3. Test details as shown by the TEAM Engine test session execution report.

The second approach used involved running the tests from the IDE, in this experiment IntelliJ IDEA community edition was used. It was useful when a test needed to be debugged or when one specific requirement was being investigated and a quick test was required.

It is faster to run the tests from the IDE than going through the Web interface. The workflow followed to debug a test involved starting the tests from the main tests controller (e.g. *WmtsNSGTestNGController*) and then placing a debug point in the concerned test:



```

    @Test
    public void verifyGetCapabilitiesSupported() {
        Node getCapabilitiesEntry = null;  getCapabilitiesEntry: "[ows:Operation: null]"
        try {
            getCapabilitiesEntry = (Node) ServiceMetadataUtils.getNode( wmtsCapabilities,
                                                                       "//ows:OperationsMetadat
        } catch ( XPathExpressionException e ) {
        }
        assertNotNull( getCapabilitiesEntry, "GetCapabilities is not supported by this WMTS" );
    }

```

AbstractBaseGetCapabilitiesFixture › verifyGetCapabilitiesSupported()

Variables

```
▼ this = {GetCapabilitiesOperations@2117}
   ▶ f wmtsCapabilities = {DeferredDocumentImpl@2119} "[#document: null]"
   ▶ f wmtsClient = {WmtsClient@2126}
   ▶ f docBuilder = {DocumentBuilderImpl@2127}
   ▶ f reqEntity = {WmtsKvpRequest@2128}
     f rspEntity = null
   ▶ f layerInfo = {ArrayList@2129}  size = 22
▼ getCapabilitiesEntry = {DeferredElementNSImpl@2157} "[ows:Operation: null]"
     f fNodeIndex = 67
   ▶ f namespaceURI = "http://www.opengis.net/ows/1.1"
```

Figure 4. Partial print screen of a CITE test debug session in IntelliJ IDEA community edition.

The preferred way to communicate with the CITE tests development team was through GitHub issues, the same applies for suggestions or doubts regarding the TEAM Engine.

Chapter 9. Conclusion

The implementation of NSG WFS and NSG WMTS profiles triggered several technical discussions that have been documented in this ER for the benefit of future implementations and the concerned standards.

Several approaches were proposed for the implementation of time versioning, random pagination and request timeout capabilities. Each one of these approaches brings its own limitations and benefits. These approaches have been discussed with the author of the NSG WFS profile. Lessons learned through the implementation and testing of this profile will be used to provide more complete documentation in the next version of the profile and possibly the related standards.

During the execution of CITE tests using GeoServer, some compliance issues were found. Those issues were fixed and the fixes contributed to GeoServer. Some capabilities made mandatory by the profiles were directly contributed to GeoServer vanilla, for example the WMTS RESTful API.

Advanced or specific NSG WFS and NSG WMTS profiles capabilities are being implemented in two GeoServer plugins allowing anyone interested in these profiles to easily test them. To allow the implementation of those capabilities as extensions several extension points were created in GeoServer, increasing the flexibility and modularity capabilities of GeoServer.

CITE tests were produced for these two profiles allowing future implementations to easily test in a reproducible way their compliance.

9.1. Recommendations

The implementation and testing of the profiles raised the following recommendations. The implementation section provides a more detailed description of the recommendations:

- The NSG WFS profile should clarify the semantics associated with *create*, *update* and *delete* operations in the context of time versioned data. It should also be clarified if a *delete* operation could or not target a certain entity and consequently delete all is associated versions.
- The NSG profile specification of the *PageResults* could be improved with some examples and an explicit link to the *index* output format.
- The NSG WFS profile could introduce a new operation used to release a resultset (created with a *GetFeature* operation *index* output format). Right now there is no reliable way to know when a resultset is not necessary anymore and can be safely removed.
- The NSG profile should clarify how the *timeout* parameter should be handled when the server implements the read, encode and stream in parallel optimization.
- It should be discussed with WMTS related working groups the possibility of making the resource format a template variable for the RESTful API definition. This would help reduce the number of XML elements that need to be added to the capabilities document.
- The WMTS should also clarify when or if it is acceptable for a client to ignore a RESTful resource template variable. A possible approach is to make template variables that belong to the base path mandatory and query parameters optional.

9.2. Best Practices

Understanding standards and translating them to a concrete implementation is a challenging task. Interpretation doubts and ambiguous definitions need to be discussed and analyzed with the involved groups (e.g. editors of the standard, working groups, and test developers) as soon as possible.

The implicit modularity of profiles should be matched by the implementation, allowing general servers to add profile support via pluggable modules.

Appendix A: Revision History

Table 11. Revision History

Date	Release	Editor	Primary clauses modified	Descriptions
June 30, 2017	0.1	Nuno Oliveira	All	Initial ER
September 30, 2017	0.2	Nuno Oliveira	All	Draft ER
October 05, 2017	0.3	Andrea Aime	All	Draft ER
October 30, 2017	0.4	Luis Bermudez	All	Draft ER
November 16, 2017	0.5	Nuno Oliveira	All	Draft ER
November 27, 2017	0.5	Nuno Oliveira	All	Draft ER

Appendix B: Bibliography

[1] GeoServer web site: http://geoserver.org

[2] GeoWebCache web site: http://www.geowebcache.org

[3] GeoTools web site: http://geotools.org

[4] CITE tests user guide: http://opengeospatial.github.io/teamengine/users.html