SmartMet Server

GRID SUPPORT

Finnish Meteorological Institute

Table of Contents

1 GRID support

1.1 Introduction

The purpose of this document is to describe the general functionality of the SmartMet grid support. The current project has been quite long, because there have been hundreds of details and a lot of bigger and smaller problems that have to been solved.

In spite of that the implementation of this project differs a lot from the existing SmartMet component implementations, it is still quite compatible with the existing SmartMet architecture.

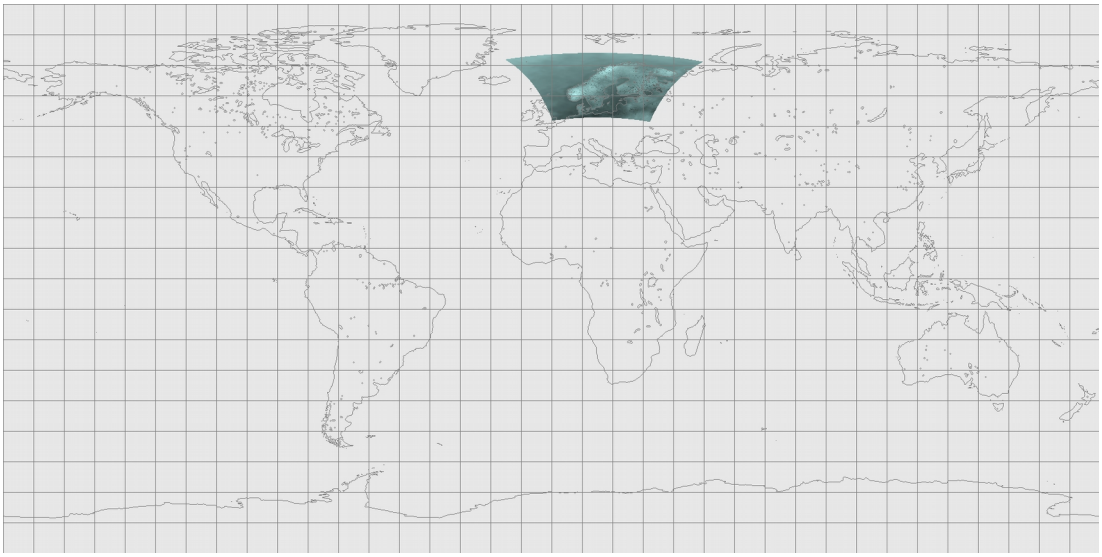
The most challenging task has been to keep all things configurable. In other words, we have not hard-coded things, which means that we can change the system behavior just by changing its configuration files. A drawback of this approach is that now we have a lot of different configuration files. This document tries to explain the content of these configuration files and also tell why they are needed.

2 Grid files

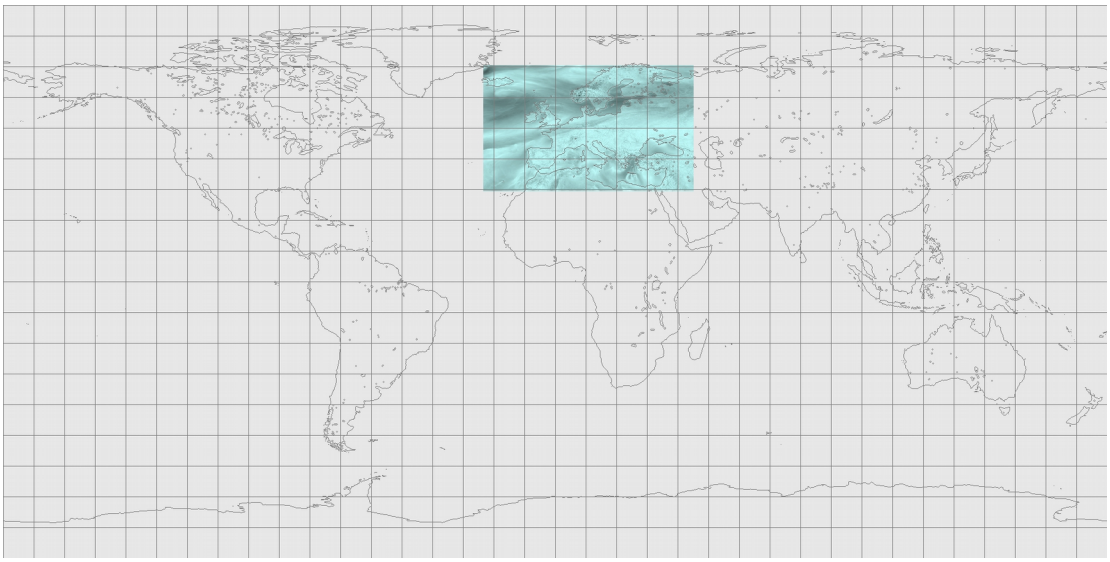
2.1 Introduction

Probably the easies way to understand a grid is to think it as a two dimensional matrix of points. Each point has a 1) geographical location and 2) a value. These values can represent for example weather forecast values (temperature, pressure, etc.) in the current geographical locations. A grid is not necessary always a rectangle, but at the moment we support only rectangular grids.

Geographical distances between the points might be equal when distances are expressed in meters or kilometers (for example 15km). In this case the grid does not look rectangular in the world map (where longitudes are X-coordinates and latitudes are Y-coordinates), because the world is a sphere. This is shown in the figure below.



However, when distances are expressed for example in latitude/longitude degrees, geographical distances of the grid point varies. If grid point distances are expressed in latitude/longitude degrees then the grid looks rectangular in the world map where longitudes are X-coordinates and latitudes are Y-coordinates. This is shown in the figure below.



2.2 Information

In order to store grid information, we need to store at least the following things:

1. Grid dimensions

If the grid is rectangular then we need to know how many rows and columns it has.

2. Geographical locations of grid points (=> coordinates)

Grid location information is usually stored by using projections / geometry definitions. For example, we can define the location of one grid corner by using latitudes and longitudes. After that we can define the location of the other grid points by using x- and y-distance definitions (for example, x-distance is 15km and y-distance is 12.5 km, or x-distance is 0.20 degrees and y-distance is 0.18 degrees).

3. Grid point values

These values might represent for example temperature values in the current geographical area. We should store a value for each grid point. If the grid has N rows and M columns then we should store N*M values.

In addition, we might want to store the following things as well:

4. Grid data identification

If we have multiple grids containing different kind of information (temperatures, pressures, wind speed, etc), we probably want to store some identification information that helps us to recognize the content of these grids.

5. Level information

In some cases we need to define also the vertical dimension of the geographical location. In this case we usually define the type of the level (ground level, sea level, pressure level, etc.) where the actual level value is measured, and the value of the actual level (for example, 2m, 10m, 1km, 100000Pa, 50000Pa, etc.).

6. Time information

Usually grid data has direct relationship to time. For example, if a grid is used for weather forecast, it must contain a time-stamp that indicates when the current forecast is valid. It does not make sense to generate values for a weather forecast if we do not know when the forecast is valid.

2.3 Grib files

Grids can be stored by using many kinds of file format. Currently our system supports GRIB-1 and GRIB-2 file formats. Unfortunately these formats are quite popular, but from a programmer's point of view they are just terrible.

There are several reasons why these file formats are really bad:

1. Information extraction

It's not possible to extract information from GRIB files without processing the actual information content. For example, if the field X has value 1 then the structure of the rest of the file is totally different compared to case where the field X has value 2, which is totally different compared to case where the field X has value 3. Unfortunately GRIB files contain several similar fields with tens of different value options, which all effect how the information can be extracted from these files.

That's why extracting information from GRIB files requires several thousands of code lines. With a better designed format we should be able to extract all necessary sections and information attributes with few hundred code lines. The actual processing of this information can require a lot of code, but the information extraction should be as simple as possible.

2. Information addition

When new projections or products are added into GRIB file definitions, this means usually that also the software code that is using GRIB files must be updated. That's because the information cannot be extracted from GRIB files without actual processing of the information of the current file.

3. Information definition

GRIB files contain a lot of fields which values are defined in external tables. These tables define meaning for values used in certain fields. For example, table 4.5 defines values for grid level types. Unfortunately, each GRIB file defines which version of these tables it is using. At the moment, there are at least 19 versions of these tables.

We assume in our implementation that already defined values do not change in different versions (i.e new definitions can be added but existing definitions are not overridden). For example, in the table 4.5 there is definition that value 101 means "Mean Sea Level". We assume that this value does not change event that table versions are changing. If this is not the case then this could be the most brainless file format ever.

4. Information identification

Identification of the actual grid data is just a nightmare. GRIB-1 and GRIB-2 use totally different fields and values to identify the grid data. This means in practice that we cannot say that the both files contains the same data without using external mapping files. So, we need mappings that say for example that temperature values are identified with number A, B and C in GRIB-1 files and with number D, E and F in GRIB-2 files.

On the other hand, different producers might use different values in order to identify same parameters. For example, the producer A might use value 123 to identify temperature data meanwhile the producer B identifies the same temperature data with value 56.

I addition, the identification requires usually 2 - 8 different fields, which causes "overlapping" identifications. For example, "temperature" is identified in GRIB-2 by using fields "discipline", "parameterCategory" and "parameterNumber", meanwhile "temperature-2m" uses the same fields plus expects that "levelType" field is 103 and the "level" field is 2. What if we try to identify a grid which level type is "undefined", but the level value is 2. Should it be identified to be "temperature-2m" or "temperature"? This is just a total chaos.

In spite of that the level type and the level value are important information, data identifiers should never depend on this kind of information. The point is that we should be able to said that the current data is "temperature", "pressure", "wind speed", etc. data, and the level type and the level value just define its location from vertical point of view (just like coordinates defines its geographical location).

Unfortunately GRIB files are so widely used that it is difficult to get rid of them very soon.

2.4 Usage

From a programmer's point of view, the biggest challenge with GRIB files is to extract information from these files. However, from a user's point of view, the biggest challenge is to map this information in some usable form. For example, how GRIB-1 and GRIB-2 temperature identifiers can be mapped so that from a user's point of view there is just one temperature identifier that the user can use for searching temperature values from different levels. The point is that each level should not have its own temperature identifier.

The Grid-Files Library was developed during the grid support project. It is used for opening and reading grid files. In practice, all component of the grid support implementation use this library in order to process grid files. When a grid file is opened with this library, it tries automatically identify content of the current file. This means in practice that it tries to map information used in the current file format into information used in the SmartMet server environment.

For example, when the library opens a GRIB file, it tries to find values for grid data identifiers (fmiParameterId, fmiParameterName, newbaseParameterId, newbaseParameterId, gribParameterId) used in the SmartMet environment. This requires a lot of different mappings. Unfortunately there are no clear "one-to-one" mappings available between different file formats and these identifiers. That's why we have to define these mappings by ourselves.

The point of these mappings is to define direct mappings between different identifiers without any conversion. This means that the parameter types should match as well as the parameter units. In other words, we should not try to map Kelvin temperature to Celsius temperatures in this library. This operation can be done in other components. For example, the Query Engine (that will be described later) is able to do this kind of mappings, because it has a capability to convert mapped data. At the moment we are describing a low level functionality that does not contain this kind of conversion.

In practice, we have some mappings already available (for example, GRIB1 <=> gribParameterId, GRIB2 <=> gribParameterId, gribParameterId <=> fmiParameterId, etc), but these mappings are not complete. On the other hand, these mappings might contain some incorrect or unwanted mappings. For example, some times Kelvin temperatures might be directly mapped to Celsius temperatures. This is possible, because there are thousands of grid data identifiers which might have a little bit different meanings.

So, these mapping files need to be updated if there are missing or incorrect mappings. Luckily these files can be updated even if the system is running. Updated mappings are automatically loaded into the system in few seconds after the mappings have been saved.

The main configuration file of the Grid-Files Library is usually named as "**grid-files.conf**". In practice, it names the files that contain actual parameter definitions and mappings. So, if you want to modify a parameter mapping, you should first check the main configuration file and find out the name of the correct mapping file. After that you can edit the current mapping file and define new mappings.

Notice that the most of the configuration parameters can have a list of values (in spite of that the most of them have just a single value). The main reason for this is that, we might want to generate some mappings automatically (for example from a database). On the other hand, we might want to add some mappings manually. In this case we can use two different mapping files at the same time. In this way we can ensure that automatically generated mappings do not override our manually added mappings, because they are in different files.

The main configuration file of the Grid-Files Library looks like this:

```
# This is the main configuration file of the grid-files library. This is a static configuration file and it is read only once during the library initialization phase. All
# the other files referenced from this file are dynamic configuration files, which means that they are automatically loaded into the system when they are updated.
#
# Parameter values can contain environment variables, which are automatically replaced with their values when configuration file is read. For example,
# value "$(HOME)" is replaced by the value of the HOME environment variable.
#
# It is also possible to define own variables that can be replaced. For example:
#
# MY_DIR = "/smartmet/mydir"
# MY_FILE = "$(MY_DIR)/myfile.txt"
#
# There is a special syntax for the absolute path of the current configuration file. %(DIR) is replaced by this path. The point is  that when a configuration file
# is included by another configuration file, it is just merged into it. In this case all relative paths in the current configuration file are not valid anymore. This problem
# is solved by %(DIR), which is replaced before the merge operation with the absolute path of the current configuration file.

smartmet : { library : { grid-files :
{
        grib :   # GRIB definitions that are common for all GRIB versions.
        {
                # List of files where common GRIB parameters are defined (i.e. common for GRIB-1 and GRIB-2).
                parameterDef = ["%(DIR)/grib_parameters.csv"];

                # List of files where GRIB table definitions can be found (these files are automatically generated).
                tableDef = ["%(DIR)/grib_tables.csv"];

                # List of files that contain GRIB unit definitions.
                unitDef = ["%(DIR)/grib_units.csv"];
        }

        grib1 :    # GRIB 1 definitions.
        {
                # List of files where GRIB 1 parameters are defined.
                        parameterDef = ["%(DIR)/grib1_parameters.csv"];

                # List of files where GRIB 1 levels are defined.
                levelDef = ["%(DIR)/grib1_levels.csv"];

                # List of files where GRIB 1 time ranges are defined.
                timeRangeDef = ["%(DIR)/grib1_timeRanges_csv"];
        }

        grib2 :   # GRIB 2 definitions
        {
                # List of files where GRIB 2 parameters are defined.
                parameterDef = ["%(DIR)/grib2_parameters.csv"];

                # List of files where GRIB 2 levels are defined.
                levelDef = ["%(DIR)/grib2_levels.csv"];

                # List of files where GRIB 2 time ranges are defined.
                timeRangeDef = ["%(DIR)/grib2_timeRanges.csv"];
        }
```

```
        fmi :   # FMI parameter definitions
        {
                # List of files where FMI parameters are defined.
                parameterDef = ["%(DIR)/fmi_parameters.csv",  "%(DIR)/fmi_parameters_ext.csv"];

                # List of files where FMI parameter levels are defined .
                levelDef = ["%(DIR)/fmi_levels.csv" ];

                # List of files where FMI Radon geometries are defined. These are needed in order to get "geometry identifiers" for different grids.
                geometryDef = ["%(DIR)/fmi_geometries.csv", "%(DIR)/fmi_geometries_test.csv", "%(DIR)/fmi_geometries_ext.csv"];

                # List of files that contain GRIB parameter mappings to FMI parameters.
                parametersFromGrib = ["%(DIR)/fmi_parameterId_grib.csv" ];

                # List of files that contain GRIB 1 parameter mappings to FMI parameters.
                parametersFromGrib1 = ["%(DIR)/fmi_parameterId_grib1.csv"];

                # List of files that contain GRIB 2 parameter mappings to FMI parameters.
                parametersFromGrib2 = ["%(DIR)/fmi_parameterId_grib2.csv"];

                # List of files that contain Newbase parameter mappings to FMI parameters.
                parametersFromNewbase = ["%(DIR)/fmi_parameterId_newbase.csv"];

                # List of files that contain GRIB 1 level mappings to FMI parameter levels.
                levelsFromGrib1 = ["%(DIR)/fmi_levelId_grib1.csv" ];

                # List of files that contain GRIB 2 level mappings to FMI parameter levels.
                levelsFromGrib2 = ["%(DIR)/fmi_levelId_grib2.csv" ];

                # List of files that contain GRIB producer mappings to Radon producers.
                producersFromGrib = ["%(DIR)/fmi_producerId_grib.csv"];
        }

        newbase :  # Newbase definitions
        {
                # List of files where newbase parameters are defined
                parameterDef = ["%(DIR)/newbase_parameters.csv"];
        }
}
}
}
```

Notice that there are several SmartMet components (like the Grid Engine and the Grid-GUI Plugin) that are using the Grid-Files Library, which means that they need to know the location of this main configuration file in order to initialize the current library. That's why this location is usually defined also in their own configuration files.

It is also possible that other components create their own configuration files with different mappings and initialize the Grid-Files Library with their own setting. The point is that there are no limitations or standards how to map GRIB data identifier to another identifiers, or even what kind of parameters we have in the first place. So, everybody can define their own identifiers and create mappings for these identifiers if they want.

The current implementation uses two organization specific identifiers for grid parameters. We call these identifiers as "FMI identifiers" and "Newbase identifiers". They are both used by Finnish Meteorological Institute (FMI). Other organizations do not need to use these FMI specific identifiers if they do not want to. They can replace them with their own identifiers or use just the GRIB identifiers (which are not so easy to use, because they are numbers).

In spite of that this all is possible, it requires a lot of configuration, because all organizations seem to have their own way to define things. So, there are no standard way to do this.

2.5 Getting started

The easiest way to test the Grid-Files Library configuration and mappings is to use "**grid_dump**" program, which prints the content of the current GRIB file. For example:

```
grid_dump /gribs/PRECFORM2-N_height_0_rll_331_289_0_006_3_41.grib2
```

The result looks something like this:

```
-----------------------------------------------------------------------------------------------
FILE : /gribs/PRECFORM2-N_height_0_rll_331_289_0_006_3_41.grib2
-----------------------------------------------------------------------------------------------
PhysicalGridFile
 GribFile (version 2)
 - fileName         = /gribs/PRECFORM2-N_height_0_rll_331_289_0_006_3_41.grib2
 - fileId           = 0
 - groupFlags       = 0
 - producerId       = 0
 - generationId     = 0
 - numberOfMessages = 1

   ########## MESSAGE [0] ##########

   - filePosition             = 0 (0x0)
   - gribParameterId          = 260015
   - gribParameterName        = ptype
   - gribParameterDescription = Precipitation type
   - gribParameterUnits       = code table (4.201)
   - parameterLevel           = 0
   - parameterLevelId         = 103
   - parameterLevelIdString   = 103
   - fmiProducerName          =
   - fmiParameterId           = 11
   - fmiParameterLevelId      = 6
   - fmiParameterName         = PRECTYPE-N
   - fmiParameterDescription  = Precipitation type, large scale or convective
   - fmiParameterUnits        = JustAnumber
   - newbaseParameterId       = 56
   - newbaseParameterName     = PrecipitationType
   - referenceTime            = 20180423T000000
   - forecastTime             = 20180423T060000
   - gridGeometryId           = 1068
   - gridHash                 = 2945055280374297391
   - gridProjection           = RotatedLatLon
   - gridLayout               = Regular
   - gridOriginalRowCount     = 289
   - gridOriginalColumnCount  = 331
    SECTION [0] Indicator
     - filePosition     = 0 (0x0)
     - reserved         =
     - discipline       = 0
     - disciplineString = 0
     - editionNumber    = 2
     - totalLength      = 28963
     SECTION [1] Identification
     - filePosition                    = 16 (0x10)
     - sectionLength                   = 21
     - numberOfSection                 = 1
     - centre                          = 86 (86)
     - subCentre                       = 0
     - tablesVersion                   = 4
     - localTablesVersion              = 0
     - significanceOfReferenceTime     = 1 (1)
     - year                            = 2018
     - month                           = 4
     - day                             = 23
```

As can be seen, the Grid-Files Library has manage to identify the current GRIB data and also find correct FMI and Newbase parameter names. It has also manage to found a correct geometry identifier for the grid.

3 Content Information

3.1 Introduction

When we are talking about the content information, we mean information that describes what kind of data is available in different grid files. In addition, this information tells us who is the producer of the current grid data. It should also tell if the grid data belongs to a certain forecast run/set (=> generation). Without this information it would be almost impossible to do any reasonable queries to the grid files.

Content information is stored into **the Content Information Storage** as records. Currently the Content Information Storage contains the following information:

1. Producer information
2. Generation information
3. Grid file information
4. Grid content information

3.1.1 Producer information

The purpose of the producer information is to define the "creator" for the grid content information. This means in practice that each grid file should have a link to its producer. Producer information is stored as "ProducerInfo" records, which contains the following fields:

RECORD ProducerInfo	
Name	**Description**
producerId	Unique producer identifier. This is used as a technical key in other records (like GenerationInfo, FileInfo and ContentInfo).
name	Unique producer name, which is usually a short name of the actual producer (for example FMI).
title	Official name of the producer (for example, "Finnish Meteorological Institute").
description	More information about the current producer.
flags	Additional information flags (32-bits) related to the current producer. These flags can be used for example for categorizing different producers.
sourceId	This field identifies the source of this record. In practice, each "feeding program" should have its own source identifier, which helps them to recognize records they have added. That is important, because they are not allowed to delete or modify information that was generated by another program.

3.1.2 Generation information

A producer can generate forecasts many times per day, which means that there can be several forecast sets available at the same time. We call these forecasts sets as generations. The purpose of the generation information is to separate different forecast sets from each other. This means in practice, that each grid file should have a link to the forecast set it belongs to. Generation information is stored as "GenerationInfo" records, which contains the following fields:

RECORD GenerationInfo	
Name	**Description**
generationId	Unique generation identifier. This field is automatically generated when a new GenerationInfo record is added. The field is used as a technical key in other records (like FileInfo and ContentInfo).
generationType	This field can be used for defining a type for the generation. There are no predefined values for this field, which means that all users can define their own values.
producerId	The producer identifier, which links the current generation to a certain producer.
name	Generation name. The name should be unique and easy to remember. This way it is easier to access

	content information from different generations. The recommendation is that the name is in the format "producerName:originTime" (for example "HIRLAM:20170317T000000")
description	This field can contain more detailed description of the current current generation.
analysisTime	This field usually indicates the time when the generation's forecast data was created. This field is the closest match for the "originTime" parameter.
status	The status of the current generation (0 = disabled, 1 = ready, 2 = incomplete).
flags	Additional information flags (32-bits) related to the current generation. These flags can be used for example for categorizing different generations.
sourceId	This field identifies the source of this record (=> feeding program identifier).

3.1.3 Grid file information

Grid file information is needed in order to locate grid files and identify them. Each grid file must have a link to its producer and to the generation it belongs to. File information is stored as "FileInfo" records, which contains the following fields:

RECORD FileInfo	
Name	**Description**
fileId	Unique file identifier. This field is automatically generated when a new FileInfo record is added. The field is used as a technical key in other records (like ContentInfo).
producerId	The producer identifier, which links the current file to a certain producer.
generationId	The generation identifier, which links the current file to a certain generation.
fileType	The type of the forecast file (0 = unknown, 1 = GRIB1, 2 = GRIB2).
name	The filename with the directory path. This field must be unique in content information database.
groupFlags	A file can belong to 32 different groups, which are indicated by 32-bit group flags. The point is that these groups gives us an opportunity to divide files into different groups according to some criteria.
flags	Additional information flags (32-bits) related to the current file. These flags can be used for example to indicate that the content of the current file is predefined, which means that the Data Servers should use these predefinitions and not to try to find out the content by themselves. At the moment following bits are in use: Bit 0 : The content of file is predefined (=> data server do not need to update it) Bit 1 : The grid file is a virtual file (=> data server has created it virtually from existing grid files) Bit 2 : The file is marked to be deleted, but its information is not yet removed from the content storage.
sourceId	This field identifies the source of this record (=> feeding program identifier).
modificationTime	Time when the file was last modified. This field might help "feeding programs" to notice if a file has changed in the file system. In this case the file information should be updated into the Content Information Storage.

3.1.4 Grid Content information

The actual grid data is stored into grid files. Each grid file can contain one or several grids. That's why each grid in the file needs to be registered. In practice, there must be one content information record for each grid in the file. Grid information is stored as "ContentInfo" records, which contains the following fields:

RECORD FileInfo	
Name	**Description**
fileId	The file identifier, which links the current content to a certain file.
fileType	The type of the forecast file (0 = unknown, 1 = GRIB1, 2 = GRIB2).
messageIndex	The index of the message, which links the current content to a certain message in the current file. Notice that the first message in a file has index 0.
producerId	The producer identifier, which links the current content to a certain producer.

generationId	The generation identifier, which links the current content to a certain generation.
groupFlags	A content file can belong to 32 different groups, which are indicated by 32-bit group flags. So, these groupFlags should be the same as the groupFlags in the related FileInfo record.
forecastTime	The time of the (forecast) data. The time format is "YYYYMMDDTHHMMSS" (for example, "20170317T123000").
fmiParameterId	The content parameter identifier used in FMI's Radon database.
fmiParameterName	The content parameter name used in FMI's Radon database.
gribParameterId	The content parameter identifier used by GRIB.
cdmParameterId	The content parameter identifier used by CDM.
cdmParameterName	The content parameter name used by CDM.
newbaseParameterId	The content parameter identifier used in FMI's Newbase definitions.
newbaseParameterName	The content parameter name used by FMI's Newbase definitions.
fmiParameterLevelId	The content parameter level identifier used in FMI's Radon database.
grib1ParameterLevelId	The content parameter level identifier used by GRIB 1.
grib2ParameterLevelId	The content parameter level identifier used by GRIB 2.
gribParameterLevel	The content parameter level.
fmiParameterUnits	The content parameter units used in FMI's Radon database.
gribParameterUnits	The content parameter units used by GRIB.
serverFlags	These flags indicates which Data Servers have registered the current content. In other words, the content is available in these Data Servers. The field has 64 bits and each bit indicates whether a Data Server has registered the current content. For example, if a Data Server (serverId = 1) has registered the content then the first bit of the serverFlags field should be "1".
flags	Additional information flags (32-bits) related to the current content.
sourceId	This field identifies the source of this record (=> feeding program identifier).
geometryId	This field identifiers the geometry used in grid files. This information is usually needed when the content information is queried. In practice, it identifies the grid's geographical region and this way we can check if the requested location is in the current region. On the other hand, producers might have several grids (with a different geometry) that contain same information (temperature, pressure. etc.). When querying this information, we need to know which geometry to use.
modificationTime	Modification time of the grid. This is usually the same as the modification time of the grid file.

As can be seen, the ContentInfo record contains several FMI-specific fields. You do not need to use them or you can replace their values with your own values. For example, Kelvin temperatures in FMI's Radon database are using the identifier "T-K". You could replace this identifier by your own identifier (for example, "TemperatureK").

3.2 Content information storages

As was mentioned earlier, content information is stored into **the Content Information Storage**. Usually there is on one master Content Information Storage in the system. However, there might be several caching content storages which contains the same content information as the master storage. The main difference is that a caching content storage fetches information updates from the master content storage automatically, meanwhile the master content storage information is updated by external programs (=> feeding programs).

Currently there are two Content Information Storage implementations that can be used as the master content storage:

1. Redis implementation
2. Memory-based implementation

These content storages are part of the Grid-Content Library.

3.2.1 Redis implementation

Redis is free, simple and quite fast database that can be accessed over TCP connection, which means that it can be installed in a remote location. In spite of that Redis is a "memory oriented" database, it also stores information into the file system. This means that the database information can be quickly restored when the system is restarted.

Redis is not an optional query database. It is fast when information can be fetched by a key, but quite slow when information needs to be searched.

Information is stored into Redis database usually by using "key - value" -pairs and possible record index keys. Redis does not actually have any good methods for handling record structures. That's why we are using Redis in a little bit strange way. We store information records into Redis as CSV (Comma Separated Value) strings. This means that we have to read and write information as CSV strings, which means that we have to split these strings in each read operation before we can access any fields inside these records.

From the search point of view this is not a good solution. That's why we use Redis only as a centralized information storage, but the actual searching takes place in the Content Server Cache implementations.

We can use the same Redis database with different SmartMet Server installations. For example, we might have different development, testing and production installations which all use the same Redis database. In order to separate different installation from each other, we use an "artificial namespace", because Redis do not contain real namespaces.

We can create an "artificial namespace" by using different "table prefix" definitions in the Content Server configuration files. In practice, the "table prefix" value is just added into the front of used table names.

For example, without any prefix the Redis database contains tables like "producers", "generations", "files" and "content". If we define that the table prefix is "test." then the Redis database are using tables like "test.producers", "test.generations", "test.files" and "test.content".

3.2.2 Memory-based implementation

The Content Information Storage is the master information source only for the other SmartMet Server components. This means in practice, that there must be an external "feeding program" that keeps the Content Information Storage up to date. If this "feeding program" is fast enough or if there are only few thousands grid files, then there is no need to store grid file registrations permanently. That's because because a fast "feeding program" can restore this information into the Content Information Storage in few seconds.

In this case, we can use a pure memory-based implementation of the Content Information Storage. This implementation uses its internal memory structures in order to store content information. It does not store this information permanently into the disk, so the information will be lost if the program is terminated.

On the other hand, content information is stored in such memory structures that searching is much faster from this implementation than the Redis implementation. In smaller installations there is no need to use the Caching Content Server in the Grid Engine, because this memory-based implementation is fast enough.

3.3 Accessing techniques

Content information in the Content Information Storages should never be directly accessed with the database's own clients. For example, in spite of that we use Redis database, we should not access it by its own client programs, because that might compromise the integrity of the current database (if we are modifying something).

Content information should be accessed and managed only through **the Content Server APIs**. The service methods of these APIs can be called by the following ways:

- By using command-line client programs
- By using the HTTP interface from a HTTP client (WWW-browser, wget, etc.)
- By using direct API calls from a C++ software components
- By using the CORBA interface from a CORBA client stub (=> generated from the IDL)

All these techniques use exactly similar looking Content Server APIs. Only the implementation part of these APIs are different. Technically speaking, they are all inherited from the same parent class.

3.3.1 Source Code

The Content Server API and all its implementations is defined in **the Grid-Content Library**. However, the current library does not contain any executable programs. The point is that the current library only contains building blocks that can be used in hundreds of different programs. All programs related to the grid support functionality (client programs, server programs, feeding programs, etc.) can be found from **the Grid Tools module**.

For example, the code below shows how easily we can implement a program that fetches a FileInfo record from the Content Information Storage by using Redis, CORBA or HTTP client implementation of the Content Server API.

```
#include "grid-content/contentServer/corba/client/ClientImplementation.h"
#include "grid-content/contentServer/http/client/ClientImplementation.h"
#include "grid-content/contentServer/redis/RedisImplementation.h"
#include "grid-files/common/Exception.h"
#include "grid-files/common/GeneralFunctions.h"


using namespace SmartMet;


int main(int argc, char *argv[])
{
  try
  {
    if (argc < 3)
    {
      fprintf(stdout,"USAGE: cs_getFileInfoById <sessionId> <fileId>\n");
      fprintf(stdout,"       [[-http <url>] | \n");
      fprintf(stdout,"        [-redis <address> <port> <tablePrefix>] | \n");
      fprintf(stdout,"        [-ior <IOR>]] \n");
      return -1;
    }

    T::SessionId sessionId = toInt64(argv[1]);
    T::FileInfo info;
    uint fileId = toInt32(argv[2]);
    int result = 0;

    if (strcmp(argv[argc-2],"-http") == 0)
    {
      ContentServer::HTTP::ClientImplementation service;
      service.init(argv[argc-1]);
      result = service.getFileInfoById(sessionId,fileId,info);
    }
    else
    if (argc > 4  &&  strcmp(argv[argc-4],"-redis") == 0)
    {
      ContentServer::RedisImplementation service;
```

```
      service.init(argv[argc-3],toInt64(argv[argc-2]),argv[argc-1]);
      result = service.getFileInfoById(sessionId,fileId,info);
    }
    else
    {
      char *serviceIor = getenv("SMARTMET_CS_IOR");
      if (strcmp(argv[argc-2],"-ior") == 0)
        serviceIor = argv[argc-1];


      if (serviceIor == nullptr)
      {
        fprintf(stdout,"Service IOR not defined!\n");
        return -2;
      }

      ContentServer::Corba::ClientImplementation service;
      service.init(serviceIor);
      result = service.getFileInfoById(sessionId,fileId,info);
    }

    if (result != 0)
    {
      fprintf(stdout,"ERROR (%d) : %s\n",result,ContentServer::getResultString(result).c_str());
      return -3;
    }

    // ### Result:
    info.print(std::cout,0,0);

    return 0;
  }
  catch (SmartMet::Spine::Exception& e)
  {
    SmartMet::Spine::Exception exception(BCP,"Service call failed!",nullptr);
    exception.printError();
    return -4;
  }
}
```

As can be seen, the usage of the Content Server API's client implementation is extremely simple. Just initialize the service client object and start to call services:

```
   ContentServer::HTTP::ClientImplementation service;
   service.init(url);
   T::FileInfo info;
   int result = service.getFileInfoById(sessionId,fileId,info);
```

3.3.2 Command-line client programs

The easiest way to access content information manually is to use command-line client programs. There is a command-line client program for each service method defined in the Content Server API. For example, we can fetch a list of content information records from the Redis database by the following command:

```
cs_getContentList 0 250000 0 1 -redis 127.0.0.1 6379 "a."
```

And the response would be something like this:

```
ContentInfoList
        ContentInfo
        - mFileId                = 250000
        - mFileType              =
        - mMessageIndex          = 0
        - mProducerId            = 35
        - mGenerationId          = 3
        - mGroupFlags            = 0
        - mForecastTime          = 20180903T200000
        - mFmiParameterId        = 523
        - mFmiParameterName      = VV-MS
        - mGribParameterId       =
        - mCdmParameterId        =
```

```
- mCdmParameterName      =
- mNewbaseParameterId    = 43
- mNewbaseParameterName  = VerticalVelocityMMS
- mFmiParameterLevelId   =
- mGrib1ParameterLevelId =
- mGrib2ParameterLevelId =
- mParameterLevel        = 2
- mFmiParameterUnits     = m s-1
- mGribParameterUnits    =
- mForecastType          = 3
- mForecastNumber        = 7
- mServerFlags           = 0
- mFlags                 = 1
- mSourceId              = 100
- mGeometryId            = 1078
- mModificationTime      = 20180903T082308
```

Each command should be able to write its usage information when the command is given without any parameters. For example:

```
cs_getContentList
```

Prints to the display:

```
cs_getContentList <sessionId> <startFileId> <startMessageIndex> <maxRecors> [[-http <url>] | [-redis <address> <port>]]
```

3.3.3 HTTP interface

We can access content information via the HTTP interface by using a normal WWW-browser or programs like "wget" or "curl" . For example, we can fetch a list of content information records by writing the following URL into the WWW-browser.

```
http://smartmet.fmi.fi/grid-admin?method=getContentList&sessionId=0&startFileId=250000&startMessageIndex=0&maxRecords=1
```

And the response looks something like this:

```
result=0

contentInfoHeader=fileId;messageIndex;fileType;producerId;generationId;groupFlags;startTime;fmiParameterId;fmiParameterName;gribParameterId;cdmParameterId;cdmParameterName;newbaseParameterId;newbaseParameterName;fmiParameterLevelId;grib1ParameterLevelId;grib2ParameterLevelId;parameterLevel;fmiParameterUnits;gribParameterUnits;mForecastType;mForecastNumber;serverFlags;flags;sourceId;geometryId;modificationTime

contentInfo=250000;0;0;35;3;0;20180903T200000;523;VV-MS;;;;43;VerticalVelocityMMS;3;0;0;2;m s-1;;3;7;0;1;100;1078;20180903T082308;
```

As was mentioned earlier, usually we do not need to manually manage content information. The HTTP interface was implemented because HTTP protocol is quite simple to use from different kinds of scripts. The idea is that different organisations can use this interface and their own scripts in order to manage content information in the Content Information Storage (=> feeding programs).

Notice that also the command-line client programs can support the HTTP protocol directly. So, we can fetch the same information (in a different format) with the following command:

```
cs_getContentList 0 250000 0 1 -http "http://smartmet.fmi.fi/grid-admin"
```

3.3.4 CORBA interface

The CORBA interface was developed for the same reason as the HTTP interface i.e. there was a need to manage and use content information remotely from other software components.

CORBA is a standard communication technique that makes possible that the same services can be accessed from different environments and different programming languages. For example, Java JDK contains an IDL-compiler (idlj) that enables CORBA usage also in Java.

The usage of CORBA requires usually quite advanced programming skills. On the other hand, CORBA communication is usually much faster than HTTP based communication techniques (like SOAP, etc). The speed of 3000 - 10 000 requests/sec is quite normal in CORBA based services when the service implementation or the network is not a bottleneck.

The basic idea of the CORBA services is that the service interface is described with a standard Interface Description Language (IDL). After that each environment can use IDL-compilers for generating client code for the current environment. They can use this client code for accessing the CORBA server interface, which skeleton was generated from the same IDL description.

In order to access a CORBA server interface, the client needs the IOR (International Object Reference) of the current service. This is a long string that the CORBA server usually prints into a file or into a display when the server is started.

```
IOR:010000003600000049444c3a536d6172744d65742f436f6e74656e745365727665722f436f7262612f5365727669636549 6e746572666163653a312e30000000010
000000000000006800000001010200 0a0000003132372e302e302e3100082016000000ff6d7920706f6100436f6e74656e7453657276696365000020000000000000000
8000000010000000054544101000000 1c00000001000000010001000100000001000105090101000100000009010100
```

The client needs to store this string so that it can use it when initializing the communication. The easiest way to do this is to use an environment variable or a configuration file. Notice that also the command-line client programs can support the CORBA directly.

If the IOR is stored into the SMARTMET_CS_IOR environment variable, then we can use command-line client programs like this:

```
cs_getContentList 0 250000 0 10 -ior $SMARTMET_CS_IOR
cs_getContentList 0 250000 0 10
```

The second command works also because the CORBA is the default protocol and the SMARTMET_CS_IOR is the default environment variable used by the command-line clients.

3.4 Feeding systems

It is important to understand that in most cases the content information will be automatically generated by "feeding programs" and stored into the Content Information Storage. That's why we do not need to store content information manually.

We can use several feeding programs at the same time. In this case each feeding program has own unique identifier (= sourceId), which it uses in order to mark records that it adds to the Content Information Storage. The basic idea is that each feeding program is allowed to remove only such information that it has added earlier.

At the moment there are two feeding programs available:

1. filesys2smartmet
2. radon2smarmet

3.4.1 filesys2smartmet

The "**filesys2smartmet**" program continuously polls the file system and searches grid files. When it finds a grid file, it reads it and registers its content into the Content Information Storage. It can also remove this information from the Content Information Storage when files are removed from the file system.

The program assumes that all grid files have the name with the following format:

PROD_YYYYMMDDTHHMMSS_XXXXX.grib

In practice, the name is divided into three parts with '_' character.

The first part is a producer identifier. This identifier should be found from the producer definition file that contain all necessary information needed in order to register the current producer into the Content Information Storage.

The second part is a generation (forecast set) identifier that defines the generation in which the current file belongs to. The recommend format is a generation time-stamp (for example, "20180912T090000"). It identifies the generation, but at the same time it is used as the originTime / analysisTime in the GenerationInfo record.

The last part can contain almost anything. However the file name should have a clear ending (like ".grib") so that it can be found with simple search patterns.

If a grid file name does not follow this naming structure and if it is not possible to change its name, then the "**filesys2smartmet**" program calls a LUA function that should generate the required name from the original file name. The point is that the grid file can be registered with its original name, but we need a structural name in order to find out the necessary producer and generation information. That's because we cannot relay that this information could be found from the actual grid file in such format that we could use it.

The name of the current LUA file and the name of the used LUA function is defined in the program's configuration file.

The "**filesys2smartmet**" program can be started with the following command:

```
filesys2smartmet <configurationfile> <loopWaitTimeInSeconds>
```

The first parameters is the name of the program's configuration file (with full path). The second parameter is time that the program waits before starting the next polling round.

The configuration file looks like this:

```
# This information is needed for initializing the grid-library.
 smartmet.library.grid-files.configFile = "/smartmet/config/library/grid-files/grid-files.conf"

 # SmartMet server contains a lot of different configuration files, with same configuration attribute names. This is not the case with the grid support.
 # Configuration attributes of each component is defined in its own unique namespace.

 smartmet : {  tools : { grid : { filesys2smartmet :
{
  content-source :
  {
        # There might be different feeding programs that update information into the content storage. That's why they need to use different source identifiers.
        source-id = 200

        # Each grid file name should contain a short producer identifier. The following file should contain the actual producer definitions that can be searched
        #  with the current producer identifier.

        producerDefFile = "/smartmet/config/grid-tools/producerDef.csv"

        # List of  directories that are searched (also subdirectories are searched)
        directories =  [ "/media/grid-files/harmonie", "/media/grid-files/test" ]

        # List of file patterns that need to match before the file is accepted.
        patterns =  [ "*.grib", "*.grib1", "*.grib2"]

        # Grid filenames should be in a fixed format (PROD_YYYYMMDDTHHMMSS_XXXXX.grib) . If this is not the case then we should call
        # the LUA function, which returns the fixed filename that  contains the required information. The filename itself is not changed.

        filenameFixer :
        {
                luaFilename = "/smartmet/config/grid-tools/filenameFixer.lua"
                luaFunction = "fixFilename"
        }

        # We might want that some of the grid files will be automatically preloaded into the memory when they are added into the Content Information Storage.
        # This means  in practice that when new content entries are added into the Content Information Storage they can be marked with the "preload requested"
        #  flag. This is how the data servers know which content should be preloaded. The following file should contain information about the content that should
        # be preloaded when the content information is added into the Content Information Storage.

        preloadFile = "/smartmet/config/grid-tools/preload.csv"

  }  // content-source

  # The "filesys2smartmet" program uses the "addFileInfoListWithContent()" service method in order to add file and content records into the content
  # information storage. The current method can add several file and content records with a single request. Usually we should add as many records as
  # possible in the same request, because it is more efficient than adding them one by one. However, some communication protocols and servers do not work
  # so well if the message size is too big. That's why we can limit the size of the request by using smaller number of the records

  maxMessageSize = 5000;  # records

  content-storage  :
  {
        # Selecting the communication type (redis/corba/http) used by the Content Information Storage
        type = "redis"

        # Defining communication details when using "redis".
        redis :
        {
                address                 = "127.0.0.1"
                port            = 6379
                tablePrefix     = "a."
        }
```

```
        # Defining communication details when using "corba".
        corba :
        {
                ior = "$(SMARTMET_CS_IOR)"
        }

        # Defining communication details when using "http".
        http :
        {
                url  = "http://smartmet.fmi.fi/grid-admin"
        }

 } // content-storage

 # Normally the program does not print any information during the search. However, when we are changing some settings, we probably want to see
 # that the program works correctly. For that purpose, the program can be configured to write a log. When the log filename is "/dev/stdout" the program
 # prints this log to into the console.

 debug-log :
 {
        enabled      = true                  // Is the debug logging enabled
        file         = "/dev/stdout"         // Prints information into the display
        maxSize      = 100000000             // When this log size is reached the the log is truncated
        truncateSize = 20000000              // This is the log size after the truncate operation
 }

} // filesys2smartmet
} // grid
} // tools
} // smartmet
```

3.4.2 radon2smartmet

Finnish Meteorological Institute is using a database called "Radon", which contains exact information about all available grid files used by FMI. This means in practice that we know the exact location and content of these files and we can handle this content by using our own identifiers. In this case there is no need to actually open the grid files and try to map their identification parameters to our identifiers.

The "**radon2smartmet**" program is a FMI-specific program that continuously polls the Radon database and synchronizes content in the Content Information Storage according to the information stored in the Radon database. SmartMet does not directly use the Radon database, because it is a private database (i.e. other SmartMet users do not have it). That's why we just copy this information into to the Content Information Storage, which the SmartMet can access.

The configuration file of the "**radon2smartmet**" program looks almost the same as the configuration file of the "**filesys2smartmet**" program. The only difference is that the "**radon2smartmet**" program needs the "**connection-string**" parameter in order to connect to the Radon database.

3.4.3 Other feeding systems

All organizations that are using SmartMet could implement their own feeding systems in order to keep the Content Information Storage up to date. A feeding system can be a simple script or a program that uses one of the Content Server API implementations in order to add, delete and update content information in the Content Information Storage.

3.5 Content Server APIs

3.5.1 Introduction

As was mentioned earlier, content information in the Content Information Storage should be accessed only via the Content Server API. In spite of that we have several techniques to access this API, they all need to use same parameters when they are calling service methods of the current API. So, the command-line clients, the HTTP interface and the CORBA interface are all using about the same method names and the parameter names. That's why we are documenting these only once.

3.5.2 Return values

When a service method is called it should return an integer value, which is the result of the current service request. If the request was successful then the method returns 0 (zero). Otherwise it returns a negative error code. The Content Server API support the following result codes.

```
OK                                      =  0
NOT_IMPLEMENTED                         = -1
DATA_NOT_FOUND                          = -2
UNEXPECTED_EXCEPTION                    = -3
UPDATE_IN_PROGRESS                      = -4
NO_PERMANENT_STORAGE_DEFINED            = -5
MISSING_PARAMETER                       = -6
PERMANENT_STORAGE_ERROR                 = -7
NO_CONNECTION_TO_PERMANENT_STORAGE      =  -8
INVALID_SESSION                         = -10
UNKNOWN_METHOD                          = -11
SERVER_ID_ALREADY_REGISTERED            = -1000
SERVER_NAME_ALREADY_REGISTERED          = -1001
SERVER_IOR_ALREADY_REGISTERED           = -1002
UNKNOWN_SERVER_ID                       = -1003
INVALID_SERVER_ID                       = -1004
PRODUCER_ID_ALREADY_REGISTERED          = -1020
PRODUCER_NAME_ALREADY_REGISTERED        = -1021
UNKNOWN_PRODUCER_ID                     = -1022
UNKNOWN_PRODUCER_NAME                   = -1023
PRODUCER_AND_GENERATION_DO_NOT_MATCH    = -1024
PRODUCER_AND_FILE_DO_NOT_MATCH          = -1025
FILE_NAME_ALREADY_REGISTERED            = -1040
UNKNOWN_FILE_ID                         = -1041
UNKNOWN_FILE_NAME                       = -1042
UNKNOWN_CONTENT                         = -1050
GENERATION_NAME_ALREADY_REGISTERED      = -1060
UNKNOWN_GENERATION_ID                   = -1061
UNKNOWN_GENERATION_NAME                 = -1062
GENERATION_AND_FILE_DO_NOT_MATCH        = -1063
UNKNOWN_PARAMETER_NAME                  = -1080
UNKNOWN_PARAMETER_KEY_TYPE              = -1081
CONTENT_ALREADY_EXISTS                  = -1100
```

3.6 Service methods

3.6.1 Producer information

Producer information is accessed through the Content Server API by using the following methods and parameters.

METHOD addProducerInfo			
Description	Adding a new ProducerInfo record into the Content Information Storage. The method creates also an unique producer identifier (= producerId) for the new producer.		
Parameters	in	sessionId	Session identifier
	inout	producerInfo	ProducerInfo record.

METHOD deleteProducerInfoById			
Description	Deleting producer information from the Content Information Storage according to the producer identifier. This method deletes the ProducerInfo record and all GenerationInfo, FileInfo and ContentInfo records related to the current producer.		
Parameters	in	sessionId	Session identifier
	in	producerId	Producer identifier.

METHOD deleteProducerInfoByName			
Description	Deleting producer information from the Content Information Storage according to the producer name. This method deletes the ProducerInfo record and all GenerationInfo, FileInfo and ContentInfo records related to the current producer.		
Parameters	in	sessionId	Session identifier
	in	producerName	Producer name.

METHOD deleteProducerInfoListBySourceId			
Description	Deleting producer records from the Content Information Storage according to the source identifier. This method deletes the ProducerInfo record and all GenerationInfo, FileInfo and ContentInfo records related to the current source. This method is usually used by a feeding program that wants to delete all information that it has added earlier.		
Parameters	in	sessionId	Session identifier
	in	sourceId	Source identifier (=> feeding program identifier)

METHOD getProducerInfoById			
Description	Fetching a ProducerInfo record according to the producer identifier.		
Parameters	in	sessionId	Session identifier
	in	producerId	Producer identifier.
	out	producerInfo	ProducerInfo record.

METHOD getProducerInfoByName			
Description	Fetching a ProducerInfo record according to the producer name.		
Parameters	in	sessionId	Session identifier
	in	producerId	Producer name.
	out	producerInfo	ProducerInfo record.

METHOD getProducerInfoList			
Description	Fetching all ProducerInfo records from the Content Information Storage.		
Parameters	in	sessionId	Session identifier
	out	producerInfo[]	ProducerInfo records.

METHOD getProducerInfoListByParameter			
Description	Fetching ProducerInfo records accoding to a parameter key. The method searches ContentInfo records and returns all producers that have registered content with the given parameter key.		
Parameters	in	sessionId	Session identifier
	in	parameterKeyType	Parameter key type
	in	parameterKey	Parameter key
	out	producerInfo[]	ProducerInfo records.

METHOD getProducerInfoListBySourceId			
Description	Fetching ProducerInfo records accoding to the source identifier.		
Parameters	in	sessionId	Session identifier
	in	sourceId	Source identifier (=> feeding program identifier)
	out	producerInfo[]	ProducerInfo records.

METHOD getProducerInfoCount			
Description	Counting the number of the ProducerInfo records in the Content Information Storage.		
Parameters	in	sessionId	Session identifier
	out	count	Number of ProducerInfo records in the Content Information Storage

METHOD getProducerNameAndGeometryList			
Description	Fetching a list of producer names and geometries. This list should contain all producers and geometries available in the Content Information Storage. This list is usually used as a base for the Query Engine's "produces.csv" configuration file.		
Parameters	in	sessionId	Session identifier
	out	string[]	List of strings. Each string contain a producer name and a geometry identifier (separated by the ';' character)

3.6.2 Generation information

Generation information is accessed through the Content Server API by using the following methods and parameters.

METHOD addGenerationInfo			
Description	Adding a new GenerationInfo record into the Content Information Storage. The method returns a new GenerationInfo record that contains the "generationId" field, which should be used as a technical key of the current record.		
Parameters	in	sessionId	Session identifier
	inout	generationInfo	GenerationInfo record.

METHOD deleteGenerationInfoById			
Description	Deleting the generation information from the content information database. This method deletes the GenerationInfo record and FileInfo and ContentInfo records related to the current generation.		
Parameters	in	sessionId	Session identifier
	in	generationId	Generation identifier.

METHOD deleteGenerationInfoByName			
Description	Deleting the generation information from the content information database. This method deletes the GenerationInfo record and FileInfo and ContentInfo records related to the current generation.		
Parameters	in	sessionId	Session identifier
	in	generationName	Generation name.

METHOD deleteGenerationInfoListByIdList			
Description	Deleting generation information according to generation identifiers. This method is usually called by a feeding program that want's to remove several old generations at the same time. This is usually much faster than removing generations one by one.		
Parameters	in	sessionId	Session identifier
	in	generationId[]	List of generation identifiers.

METHOD deleteGenerationInfoListByProducerId			
Description	Deleting all generation information related to the current producer. This method deletes all GenerationInfo, FileInfo and ContentInfo records related to the current producer. However, it does not delete the producer information.		
Parameters	in	sessionId	Session identifier
	in	producerId	Producer identifier.

METHOD deleteGenerationInfoListByProducerName			
Description	Deleting all generation information related to the current producer. This method deletes all GenerationInfo, FileInfo and ContentInfo records related to the current producer. However, it does not delete the producer information.		
Parameters	in	sessionId	Session identifier
	in	producerName	Producer name.

METHOD deleteGenerationInfoListBySourceId			
Description	Deleting all generation information related to the source identifier. This method is usually called by a feeding program that want to delete all generation infrormation that it has added earlier.		
Parameters	in	sessionId	Session identifier
	in	sourceId	Source identifier (=> feeding program identifier)

METHOD deleteGenerationInfoListByProducerName			
Description	Deleting all generation information related to the current producer. This method deletes all GenerationInfo, FileInfo and ContentInfo records related to the current producer. However, it does not delete the producer information.		
Parameters	in	sessionId	Session identifier
	in	producerName	Producer name.

METHOD getGenerationInfoById			
Description	Fetching a GenerationInfo record according to the generation identifier.		
Parameters	in	sessionId	Session identifier
	in	generationId	Generation identifier.
	out	generationInfo	GenerationInfo record.

METHOD getGenerationInfoByName			
Description	Fetching a GenerationInfo record according to the generation name.		
Parameters	in	sessionId	Session identifier
	in	generationId	Generation name.
	out	generationInfo	GenerationInfo record.

METHOD getGenerationInfoList			
Description	Fetching all GenerationInfo records from the Content Information Storage.		
Parameters	in	sessionId	Session identifier
	out	generationInfo[]	GenerationInfo records.

METHOD getGenerationInfoListByGeometryId			
Description	Fetching GenerationInfo records according to the geometry identifier. A GenerationInfo record is returned if the current generation has content defined for the current geometry.		
Parameters	in	sessionId	Session identifier
	in	geometryId	Geometry identifier.
	out	generationInfo[]	GenerationInfo records.

METHOD getGenerationInfoListByProducerId			
Description	Fetching GenerationInfo records according to the producer identifier.		
Parameters	in	sessionId	Session identifier
	in	producerId	Producer identifier.
	out	generationInfo[]	GenerationInfo records.

METHOD getGenerationInfoListByProducerName			
Description	Fetching GenerationInfo records according to the producer name.		
Parameters	in	sessionId	Session identifier
	in	producerName	Producer name.
	out	generationInfo[]	GenerationInfo records.

METHOD getGenerationInfoListBySourceId			
Description	Fetching GenerationInfo records according to the source identifier.		
Parameters	in	sessionId	Session identifier
	in	sourceId	Source identifier (=> feeding program identifier)
	out	generationInfo[]	GenerationInfo records.

METHOD getLastGenerationInfoByProducerIdAndStatus			
Description	Fetching the latest GenerationInfo record of the given producer, which has the given status.		
Parameters	in	sessionId	Session identifier
	in	producerId	Producer identifier.
	in	status	Expected generation status.
	out	generationInfo	GenerationInfo record.

METHOD getLastGenerationInfoByProducerINameAndStatus			
Description	Fetching the latest GenerationInfo record of the given producer, which has the given status.		
Parameters	in	sessionId	Session identifier
	in	producerName	Producer name.
	in	status	Expected generation status.
	out	generationInfo	GenerationInfo record.

METHOD getGenerationInfoCount			
Description	Counting the number of the GenerationInfo records in the Content Information Storage.		
Parameters	in	sessionId	Session identifier.
	out	count	Number of GenerationInfo records in the Content Information Storage.

METHOD setGenerationInfoStatusById			
Description	Setting the status field of the GenerationInfo record.		
Parameters	in	sessionId	Session identifier
	in	generationId	Generation identifier.
	in	status	Status.

METHOD setGenerationInfoStatusByName			
Description	Setting the status field of the GenerationInfo record.		
Parameters	in	sessionId	Session identifier
	in	generationName	Generation name.
	in	status	Status.

3.6.3 Grid file information

Grid file information is accessed through the Content Server API by using the following methods and parameters.

METHOD addFileInfo			
Description	Adding a new FileInfo record into the Content Information Storage. The method returns a new FileInfo record that contains the "fileId" field, which should be used as a technical key of the current record.		
Parameters	in	sessionId	Session identifier
	inout	fileInfo	FileInfo record.

METHOD addFileInfoWithContentList			
Description	Adding a new FileInfo record and its predefined content information into the Content Information Storage. The method returns a new FileInfo record that contains the "fileId" field, which should be used as a technical key of the current record. The "flags" field is used to indicate that the content of the current file is predefined.		
Parameters	in	sessionId	Session identifier
	inout	fileInfo	FileInfo record.
	in	contentInfo[]	ContentInfo records.

METHOD addFileInfoListWithContent			
Description	Adding several FileInfo records and ContentInfo records into the Content Information Storage with the same request. This is the fastest way to add content information. The method uses a specific record structure "FileAndContent" that contains a FileInfo record and a list of ContentInfo records.		
Parameters	in	sessionId	Session identifier
	in	fileAndContent[]	List of FileAndContent records.

METHOD deleteFileInfoById			
Description	Deleting the file information from the Content Information Storage. This method deletes the FileInfo record and ContentInfo records related to the current file.		
Parameters	in	sessionId	Session identifier
	in	fileId	File identifier.

METHOD deleteFileInfoByName			
Description	Deleting the file information from the Content Information Storage. This method deletes the FileInfo record and ContentInfo records related to the current file.		
Parameters	in	sessionId	Session identifier
	in	fileName	File name.

METHOD deleteFileInfoListByProducerId			
Description	Deleting all file information according to the producer identifier. This method deletes the all FileInfo and ContentInfo records related to the current producer.		
Parameters	in	sessionId	Session identifier
	in	producerId	Producer identifier.

METHOD deleteFileInfoListByProducerName			
Description	Deleting all file information according to the producer name. This method deletes the all FileInfo and ContentInfo records related to the current producer.		
Parameters	in	sessionId	Session identifier
	in	producerName	Producer name.

METHOD deleteFileInfoListByGenerationId			
Description	Deleting all file information according to the generation identifier. This method deletes all FileInfo and ContentInfo records related to the current generation.		
Parameters	in	sessionId	Session identifier
	in	generationId	Generation identifier.

METHOD deleteFileInfoListByGenerationIdAndForecastTime			
Description	Deleting all file information according to the generation identifier and the forecast time. This method deletes all FileInfo and ContentInfo records related to the current generation and the forecast time.		
Parameters	in	sessionId	Session identifier
	in	generationId	Generation identifier.
	in	forecastTime	Forecast time.

METHOD deleteFileInfoListByForecastTimeList			
Description	Deleting all file information according to the forecast time list. The current list contains "ForecastTime" records, which contain gnerationId, geometryId, forecastType, forcastNumber and forecastTime fields. These fields are used in order to define records that needs to deleted.		
Parameters	in	sessionId	Session identifier
	in	forecastTime[]	List of ForecastTime records.

METHOD deleteFileInfoListByGenerationName			
Description	Deleting all file information according to the generation name. This method deletes all FileInfo and ContentInfo records related to the current generation.		
Parameters	in	sessionId	Session identifier
	in	generationName	Generation name.

METHOD deleteFileInfoListByGroupFlags			
Description	Deleting all file information according to the group flags. This method deletes all FileInfo and ContentInfo records that belong to at least one of the groups defined by the groupFlags parameter. Also content records related to the current files are deleted.		
Parameters	in	sessionId	Session identifier
	in	groupFlags	Group flags.

METHOD deleteFileInfoListBySourceId			
Description	Deleting all file information according to the source identifier.		
Parameters	in	sessionId	Session identifier
	in	sourceId	Source identifier (=> feeding program identifier).

METHOD deleteFileInfoListByFileIdList			
Description	Deleting FileInfo records and related ContentInfo records according to a list of file identifiers. Usually it is faster to delete several records at the same time than to delete them one by one.		
Parameters	in	sessionId	Session identifier
	in	fileId[]	List of file identifiers.

METHOD getFileInfoById			
Description	Fetching a FileInfo record according to the file identifier.		
Parameters	in	sessionId	Session identifier
	in	fileId	File identifier.
	out	fileInfo	FileInfo record.

METHOD getFileInfoByName			
Description	Fetching a FileInfo record according to the file name.		
Parameters	in	sessionId	Session identifier
	in	fileId	File name.
	out	fileInfo	FileInfo record.

METHOD getFileInfoList			
Description	Fetching a list of FileInfo records from the Content Information Storage.		
Parameters	in	sessionId	Session identifier
	in	startFileId	The records search starts from this file identifier.
	in	maxRecords	The maximum number of the returned records .
	out	fileInfo[]	FileInfo records.

METHOD getFileInfoListByProducerId			
Description	Fetching a list of FileInfo records according to the producer identifier.		
Parameters	in	sessionId	Session identifier
	in	producerId	Producer identifier.
	in	startFileId	The records search starts from this file identifier.
	in	maxRecords	The maximum number of the returned records .
	out	fileInfo[]	FileInfo records.

METHOD getFileInfoListByProducerName			
Description	Fetching a list of FileInfo records according to the producer name.		
Parameters	in	sessionId	Session identifier
	in	producerName	Producer name.
	in	startFileId	The records search starts from this file identifier.
	in	maxRecords	The maximum number of the returned records .
	out	fileInfo[]	FileInfo records.

METHOD getFileInfoListByGenerationId			
Description	Fetching a list of FileInfo records according to the generation identifier.		
Parameters	in	sessionId	Session identifier
	in	generationId	Generation identifier.
	in	startFileId	The records search starts from this file identifier.
	in	maxRecords	The maximum number of the returned records .
	out	fileInfo[]	FileInfo records.

METHOD getFileInfoListByGenerationName			
Description	Fetching a list of FileInfo records according to the generation name.		
Parameters	in	sessionId	Session identifier
	in	generationName	Generation name.
	in	startFileId	The records search starts from this file identifier.
	in	maxRecords	The maximum number of the returned records.
	out	fileInfo[]	FileInfo records.

METHOD getFileInfoListByGroupFlags			
Description	Fetching a list of FileInfo records according to the group flags.		
Parameters	in	sessionId	Session identifier
	in	groupFlags	Group flags.
	in	startFileId	The records search starts from this file identifier.
	in	maxRecords	The maximum number of the returned records .
	out	fileInfo[]	FileInfo records.

METHOD getFileInfoListBySourceId			
Description	Fetching a list of FileInfo records according to the source identifier.		
Parameters	in	sessionId	Session identifier
	in	sourceId	Source identifier (=> feeding program identifier).
	in	startFileId	The records search starts from this file identifier.
	in	maxRecords	The maximum number of the returned records .
	out	fileInfo[]	FileInfo records.

METHOD getFileInfoCount			
Description	Counting the number of the FileInfo records in the Content Information Storage.		
Parameters	in	sessionId	Session identifier.
	out	count	Number of FileInfo records in the Content Information Storage.

METHOD getFileInfoCountByProducerId			
Description	Counting the number of the FileInfo records according to the producer identifier.		
Parameters	in	sessionId	Session identifier.
	in	producerId	Producer identifier.
	out	count	Number of FileInfo records.

METHOD getFileInfoCountByGenerationId			
Description	Counting the number of the FileInfo records according to the generation identifier.		
Parameters	in	sessionId	Session identifier.
	in	generationId	Generation identifier.
	out	count	Number of FileInfo records.

METHOD getFileInfoCountBySourceId			
Description	Counting the number of the FileInfo records according to the source identifier.		
Parameters	in	sessionId	Session identifier.
	in	sourceId	Source identifier (=> feeding program identifier).
	out	count	Number of FileInfo records.

3.6.4 Grid content information

Grid content information is accessed through the Content Server API by using the following methods and parameters.

METHOD addContent			
Description	Adding a new ContentInfo record into the Content Information Storage. The current record can be accessed by using its "fileId" and "messageIndex" field i.e. this combination should be always unique.		
Parameters	in	sessionId	Session identifier
	inout	contentInfo	ContentInfo record.

METHOD addContentList			
Description	Adding a list of new ContentInfo records into the Content Information Storage.		
Parameters	in	sessionId	Session identifier
	inout	contentInfo[]	ContentInfo records.

METHOD deleteContentInfo			
Description	Deleting a ContentInfo record according to its "fileId" and "messageIndex" fields.		
Parameters	in	sessionId	Session identifier
	in	fileId	File identifier.
	in	messageIndex	The message index.

METHOD deleteContentListByFileId			
Description	Deleting ContentInfo records according to the file identifier.		
Parameters	in	sessionId	Session identifier
	in	fileId	File identifier.

METHOD deleteContentListByFileName			
Description	Deleting ContentInfo records according to the file name.		
Parameters	in	sessionId	Session identifier
	in	fileName	Filename.

METHOD deleteContentListByGroupFlags			
Description	Deleting ContentInfo records according to the group flags.		
Parameters	in	sessionId	Session identifier
	in	groupFlags	Group flags.

METHOD deleteContentListByProducerId			
Description	Deleting ContentInfo records according to the producer identifier.		
Parameters	in	sessionId	Session identifier
	in	producerId	Producer identifier.

METHOD deleteContentListByProducerName			
Description	Deleting ContentInfo records according to the producer name.		
Parameters	in	sessionId	Session identifier
	in	producerName	Producer name.

METHOD deleteContentListByGenerationId			
Description	Deleting ContentInfo records according to the generation identifier.		
Parameters	in	sessionId	Session identifier
	in	generationId	Generation identifier.

METHOD deleteContentListByGenerationName			
Description	Deleting ContentInfo records according to the generation name.		
Parameters	in	sessionId	Session identifier
	in	generationName	Generation name.

METHOD deleteContentListBySourceId			
Description	Deleting ContentInfo records according to the source identifier.		
Parameters	in	sessionId	Session identifier
	in	sourceId	Source identifier (=> feeding program identifier).

METHOD registerContentList			
Description	A Data Server can use this method for registering content information that is availabe in the current server.		
Parameters	in	sessionId	Session identifier
	in	serverId	Data Server identifier, which is registering the content.
	inout	contentInfo[]	ContentInfo records.

METHOD registerContentListByFileId			
Description	A Data Server can use this method for informing that the file is available also in the current server. The point is that another Data Server already has registered the content of this file so the current file does not need to re-register it.		
Parameters	in	sessionId	Session identifier
	in	serverId	Data Server identifier, which is registering the content.
	in	fileId	File identifier.

METHOD unregisterContentList			
Description	A Data Server can use this method for removing its content registrations. This is usually done when a Data Server is reseting itself.		
Parameters	in	sessionId	Session identifier
	in	serverId	Data Server identifier, which is unregistering the content.

METHOD unregisterContentListByFileId			
Description	A Data Server can use this method for unregistering the content information of a file. In this way it indicates that the current content is not available in the current Data Server anymore.		
Parameters	in	sessionId	Session identifier
	in	serverId	Data Server identifier, which is registering the content.
	in	fileId	File identifier.

METHOD getContentInfo			
Description	Fetching a ContentInfo record according to the file identifier and the message index.		
Parameters	in	sessionId	Session identifier
	in	fileId	File identifier.
	in	messageIndex	Message index.
	out	contentInfo	ContentInfo record.

METHOD getContentList			
Description	Fetching a list of ContentInfo records.		
Parameters	in	sessionId	Session identifier
	in	startFileId	The records search starts from this file identifier.
	in	startMessageIndex	The records search starts from this message index.
	in	maxRecords	The maximum number of the returned records.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByFileId			
Description	Fetching a list of ContentInfo records according to the file identifier.		
Parameters	in	sessionId	Session identifier
	in	fileId	File identifier.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByFileName			
Description	Fetching a list of ContentInfo records according to the file name.		
Parameters	in	sessionId	Session identifier
	in	filename	File name.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByGroupFlags			
Description	Fetching a list of ContentInfo records according to the group flags.		
Parameters	in	sessionId	Session identifier
	in	groupFlags	Group flags. Each bit presents a different group.
	in	startFileId	The records search starts from this file identifier.
	in	startMessageIndex	The records search starts from this message index.
	in	maxRecords	The maximum number of the returned records.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByProducerId			
Description	Fetching a list of ContentInfo records according to the producer identifier.		
Parameters	in	sessionId	Session identifier
	in	producerId	Producer identifier.
	in	startFileId	The records search starts from this file identifier.
	in	startMessageIndex	The records search starts from this message index.
	in	maxRecords	The maximum number of the returned records.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByProducerName			
Description	Fetching a list of ContentInfo records according to the producer name.		
Parameters	in	sessionId	Session identifier
	in	producerName	Producer name.
	in	startFileId	The records search starts from this file identifier.
	in	startMessageIndex	The records search starts from this message index.
	in	maxRecords	The maximum number of the returned records.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByServerId			
Description	Fetching a list of ContentInfo records according to the Data Server identifier.		
Parameters	in	sessionId	Session identifier
	in	serverId	Data Server identifier.
	in	startFileId	The records search starts from this file identifier.
	in	startMessageIndex	The records search starts from this message index.
	in	maxRecords	The maximum number of the returned records.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByGenerationId			
Description	Fetching a list of ContentInfo records according to the generation identifier.		
Parameters	in	sessionId	Session identifier
	in	generationId	Generation identifier.
	in	startFileId	The records search starts from this file identifier.
	in	startMessageIndex	The records search starts from this message index.
	in	maxRecords	The maximum number of the returned records.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByGenerationName			
Description	Fetching a list of ContentInfo records according to the generation name.		
Parameters	in	sessionId	Session identifier
	in	generationName	Generation name.
	in	startFileId	The records search starts from this file identifier.
	in	startMessageIndex	The records search starts from this message index.
	in	maxRecords	The maximum number of the returned records.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListBySourceId			
Description	Fetching a list of ContentInfo records according to the source identifier.		
Parameters	in	sessionId	Session identifier
	in	sourceId	Source identifier.
	in	startFileId	The records search starts from this file identifier.
	in	startMessageIndex	The records search starts from this message index.
	in	maxRecords	The maximum number of the returned records.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByGenerationIdAndTimeRange			
Description	Fetching a list of ContentInfo records according to the generation identifier and the time range.		
Parameters	in	sessionId	Session identifier
	in	generationId	Generation identifier.
	in	startTime	Start time of the time range (YYYYMMDDTHHMMSS).
	in	endTime	End time of the time range (YYYYMMDDTHHMMSS).
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByGenerationNameAndTimeRange			
Description	Fetching a list of ContentInfo records according to the generation name and the time range.		
Parameters	in	sessionId	Session identifier
	in	generationName	Generation name.
	in	startTime	Start time of the time range (YYYYMMDDTHHMMSS).
	in	endTime	End time of the time range (YYYYMMDDTHHMMSS).
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByParameter			
Description	Fetching a list of ContentInfo records according to the parameter key, level and time range.		
Parameters	in	sessionId	Session identifier
	in	parameterKeyType	Parameter key type.
	in	parameterKey	Parameter key.
	in	parameterLevelIdType	Parameter level identifier type.
	in	parameterLevelId	Parameter level identifier.
	in	minLevel	Minimum parameter level.
	in	maxLevel	Maxumum parameter level.
	in	startTime	Start time of the time range (YYYYMMDDTHHMMSS).
	in	endTime	End time of the time range (YYYYMMDDTHHMMSS).
	in	requestFlags	Request flags.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByParameterAndGenerationId			
Description	Fetching a list of ContentInfo records according to the generation identifier, the parameter key, level and time range.		
Parameters	in	sessionId	Session identifier
	in	generationId	Generation identifier.
	in	parameterKeyType	Parameter key type.
	in	parameterKey	Parameter key.
	in	parameterLevelIdType	Parameter level identifier type.
	in	parameterLevelId	Parameter level identifier.
	in	minLevel	Minimum parameter level.
	in	maxLevel	Maxumum parameter level.
	in	startTime	Start time of the time range (YYYYMMDDTHHMMSS).
	in	endTime	End time of the time range (YYYYMMDDTHHMMSS).
	in	requestFlags	Request flags.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByParameterAndGenerationIdAndForecastTime			
Description	Fetching ContentInfo records according to the generation identifier, the parameter key, level and forecast time. If the forecast time does not match then the method returns ContentInfo that are closest (before and after) to the requested time.		
Parameters	in	sessionId	Session identifier
	in	generationId	Generation identifier.
	in	parameterKeyType	Parameter key type.
	in	parameterKey	Parameter key.
	in	parameterLevelIdType	Parameter level identifier type.
	in	parameterLevelId	Parameter level identifier.
	in	level	Parameter level.
	in	forecastType	Forecast type.
	in	forecastNumber	Forcast number
	int	geometryId	Geometry identifier.
	in	forecastTime	Forcast time (YYYYMMDDTHHMMSS).
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByParameterAndGenerationName			
Description	Fetching a list of ContentInfo records according to the generation name, the parameter key, level and time range.		
Parameters	in	sessionId	Session identifier
	in	generationName	Generation name.
	in	parameterKeyType	Parameter key type.
	in	parameterKey	Parameter key.
	in	parameterLevelIdType	Parameter level identifier type.
	in	parameterLevelId	Parameter level identifier.
	in	minLevel	Minimum parameter level.
	in	maxLevel	Maxumum parameter level.
	in	startTime	Start time of the time range (YYYYMMDDTHHMMSS).
	in	endTime	End time of the time range (YYYYMMDDTHHMMSS).
	in	requestFlags	Request flags.
	out	contentInfo[]	ContentInfo records.

METHOD getContentListByParameterAndProducerId			
Description	Fetching a list of ContentInfo records according to the producer identifier, the parameter key, level and time range.		
Parameters	in	sessionId	Session identifier
	in	producerId	Producer identifier.
	in	parameterKeyType	Parameter key type.
	in	parameterKey	Parameter key.
	in	parameterLevelIdType	Parameter level identifier type.
	in	parameterLevelId	Parameter level identifier.
	in	minLevel	Minimum parameter level.
	in	maxLevel	Maxumum parameter level.
	in	startTime	Start time of the time range (YYYYMMDDTHHMMSS).
	in	endTime	End time of the time range (YYYYMMDDTHHMMSS).
	in	requestFlags	Request flags.
	out	contentInfo[]	ContentInfo records.

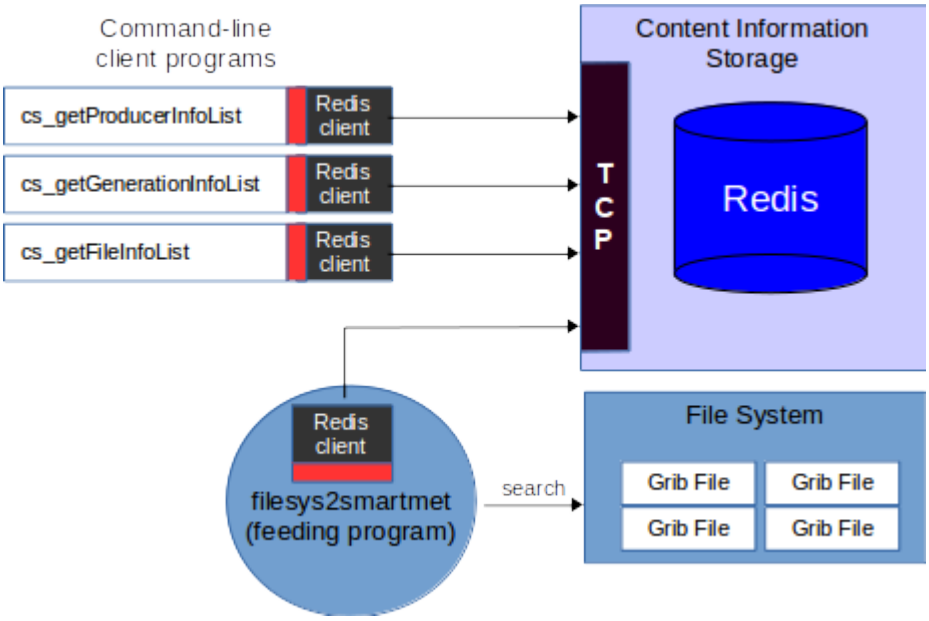
METHOD getContentListByParameterAndProducerIName			
Description	Fetching a list of ContentInfo records according to the producer name, the parameter key, level and time range.		
Parameters	in	sessionId	Session identifier
	in	producerName	Producer name.
	in	parameterKeyType	Parameter key type.
	in	parameterKey	Parameter key.
	in	parameterLevelIdType	Parameter level identifier type.
	in	parameterLevelId	Parameter level identifier.
	in	minLevel	Minimum parameter level.
	in	maxLevel	Maxumum parameter level.
	in	startTime	Start time of the time range (YYYYMMDDTHHMMSS).
	in	endTime	End time of the time range (YYYYMMDDTHHMMSS).
	in	requestFlags	Request flags.
	out	contentInfo[]	ContentInfo records.

METHOD getContentCount			
Description	Counting the number of the ContentInfo records in the Content Information Storage.		
Parameters	in	sessionId	Session identifier.
	out	count	Number of ContentInfo records in the Content Information Storage.

3.7 Getting started

The Content Information Storage is the key component of the system. When you get it up and running, then the rest is easy. If you are using a Redis database as the Content Information Storage then you can start to fill it with content information even that the SmartMet server is not running.

In order to do that, you just need the Redis database and few grib files that are named like the "**filesys2smartmet**" program expects (this was described earlier). The figure below show the basic arrangement of this demo.



Let's assume that you have the following three grid files stored into the file system and each file contains one grid:

```
/media/grid-files/test/FMI_20180912T000000_T-K_1200.grib
/media/grid-files/test/FMI_20180912T000000_T-K_1300.grib
/media/grid-files/test/FMI_20180912T000000_T-K_1400.grib
```

These files are named so that we can easily recognize the producer ("FMI") and the generation ("20180912T000000") of these files.

For registering these files into the Content Information Storage, you should execute the following steps.

STEP 1:
Start the Redis database so that it uses its default port (6379). You can do this by giving the following command :

```
redis-server
```

If you want to start the Redis server in a different port, then you should read the Redis manual in order to find out how this can be done.

STEP 2:
Set the following values into the the "**filesys-to-smartmet.cfg**" configuration file:

```
content-source.producerDefFile                = "%(DIR)/myproducers.csv"
content-source.directories                    = ["/media/grid-files/test"]
content-source.patterns                       = ["*.grib"]
content-source.filenameFixer.luaFilename      = ""
content-storage.type                          = "redis"
content-storage.redis.address                 = "127.0.0.1"
content-storage.redis.port                    = 6379
content-storage.redis.tablePrefix             = "a."
debug-log.enabled                             = true
debug-log.file                                = "/dev/stdout"
```

STEP 3:
Create the "**myproducers.csv**" file and add the following line into it:

```
FMI;FMI;Finnish Meteorological Institute;Finnish Meteorological Institute;
```

This information is used when a new ProducerInfo record is added iinto the Content Information Storage.

STEP 4:
Run the "**filesys2smartmet**" program once with the following command:

```
filesys2smartmet filesys-to-smartmet.cfg 0
```

If everything went well, the Content Information Storage should now contain one producer record, and generation record, three file records and three content records.

We can check this by using the following commands:

```
cs_getProducerInfoList 0 -redis "127.0.0.1" 6379 "a."
cs_getGenerationInfoList 0 -redis "127.0.0.1" 6379 "a."
cs_getFileInfoList 0 1 3 -redis "127.0.0.1" 6379 "a."
cs_getContentList 0 1 0 3 -redis "127.0.0.1" 6379 "a."
```

If you want test the same with other grid files or configuration settings, you might want to clear the content of the Redis database first with the following command:

```
cs_clear 0 -redis "127.0.0.1" 6379 "a."
```

If you want to continue this demo, do not clear the database yet.

Notice that at this stage we can access the Content Information Storage only by using the Redis implementation of the Content Server API. That's because the SmartMet Server is not running and there are no other communication interfaces available.

The next thing that we could try is to access content information via the CORBA interface. This requires that we start the "**corbaContentServer**" program. We can configure this program so that it only converts CORBA service requests to Redis service requests.

On the other hand, we can configure the program so that it it working as a Caching Content Server. In this case it caches all information stored into the Redis database into its own internal memory structures and all information are fetched from these structures.

In order to use "corbaContentServer" we need to execute the following steps:

STEP 1:
Set the following values into the the "**corba-content-server.cfg**" configuration file:

```
content-server.address = "127.0.0.1"
content-server.port = 8200
content-server.cache.enabled = true
content-server.iorFile = "/dev/stdout"
content-server.content-source.type = "redis"
content-server.content-source.redis.address = "127.0.0.1"
content-server.content-source.redis.port = "6379"
content-server.content-source.redis.tablePrefix = "a."
```

STEP 2:
Start the "**corbaContentServer**" program with the following command:

```
corbaContentServer corba-content-server.cfg
```

If everything went well, the program printed its International Object Reference (IOR) into the display. Set this value to the **SMARTMET_CS_IOR** environment variable like this:

```
export SMARTMET_CS_IOR="IOR:010000003600000049444c3a536d6172744d65742f436f6e74656e745365727665722f436f7262612f53657276696365496
e746572666163653a312e3000000001000000000000006800000001010200 0a0000003132372e302e302e3100082016000000ff6d7920706f6100436f6e74656e74
53657276696365000002000000000000000800000001000000005454410100000001c000000010000000100100010000000100010509010100010000000 9010100"
```

After that you should be able to access the Caching Content Server by using its CORBA interface with the command-line clients:

```
cs_getProducerInfoList 0
cs_getGenerationInfoList 0
cs_getFileInfoList 0 1 3
cs_getContentList 0 1 0 3
```

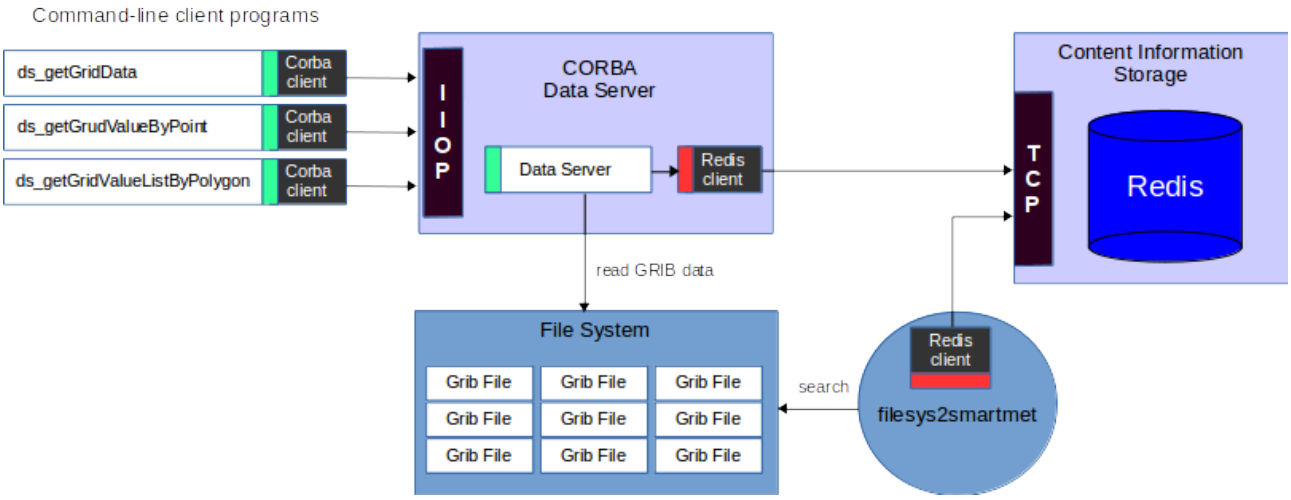
4 Data Server

4.1 Introduction

In the previous chapter we explained how content information of thousands of grib files can be stored into the Content Information Storage. This is necessary from a search point of view, which requires that we know which grids are available and where the actual grid files can be found.

This is not enough. We need also to be able to reach those files and access the actual grid data inside them. Technically we could access them if they are stored into a file system that we can reach. In this case we probably would open those files by using some grid libraries (like the SmartMet Grid-Files Library). This is one way to do it, but probably not the smartest.

A smarter way to access grid data in grid files is to access it via the Data Server.



The Data Server controls access to the actual grid files. In practice, the Data Server monitors the Content Information Storage and this way it knows which grid files should be available. On the other hand, if a grid file is registered into the Content Information Storage without any actual content information, then the Data Server reads the current grid file and fulfills the content registration into the Content Information Storage.

Notice that if there are hundreds of thousands grid files or if the sizes of grid files are huge, then it takes a lot of time to read those files and register their content information into the Content Information Storage. This means in practice, that in bigger real-time systems, the feeding systems should do the content registration at the same time when they add the grid file information into the Content Information Storage.

The current Data Server implementation maps grid files into the computer's virtual memory so that they can be accessed fast when they are once read into the memory. It is good to realize that when there are tens of thousands memory mapped grid files in the system, operating system limitations might cause some problems. In this case there might be a need to tune some of the operating system parameters, like the maximum number of open files or the maximum number of memory mapped files.

The original idea with the content information registration was that we could have several different Data Servers and each Data Server might offer access to different grid files. That's why there is still "serverFlags" field in the FileInfo and ContentInfo records stored in the Content Information Storage.

The point was that each bit of the current field represents one Data Server and this way we know which Data Servers have the requested grid files and which do not.

This functionality is not currently in use, which means that all Data Servers should see the same grid files. So, at the moment Data Server clients do not need to select which Data Server to use, because all Data Servers can access the same grid files.

4.2 Usage

The basic idea of the Data Server usage is that when we are looking for some information (for example Temperature forecast values for a specific location in a certain time), we first need to first locate the current information by using information stored into the Content Information Storage. As the result of this search, we should find 1) **the file identifier** (that identifies the grid file) and 2) **the message index** (that identifies the location inside the grid file). After that we can use the current file identifier and the message index in order to fetch actual grid data from the Data Server.

The Data Server has also the similar **Data Server API** (marked with green color in the previous figure) as the Content Server has - only the service methods are different. In practice, we can use the Data Server API in order to fetch grid data from the grid files in the ways:

1. Fetching all values in the grid.

2. Fetching values of given grid points.

3. Fetching values according to geographical coordinates. It is good to notice that the geographical coordinates do not necessary match to the actual grid points. In this case the returned values can be interpolated from the surrounding grid points.

4. Fetching values of closest grid points according to geographical coordinates. In this case the caller gets four values per a geographical coordinate, and the relative location of the current geographical coordinate inside the rectangle.

5. Fetching grid point values inside areas (circle, rectangle, polygon) defined by grid points.

6. Fetching grid point values inside areas (circle, rectangle, polygon) defined by geographical coordinates.

The basic assumption is that the current Data ServerAPI will be expanded in future quite much. For example, if we want to find out minimum, maximum or mean values of a grid, it does not make sense to fetch the whole grid data from the Data Server. Instead, we can implement service methods that returns these values.

The point is that the grid data can be handled in many ways, but the size of this data is usually so big, that it is quite heavy operation to move it around. That's why we should fetch only data that we need.

Currently there are only three implementations of the Data Server API:

1. The actual Data Server implementation that can access grid files in the file system. It is important to understand that this implementation needs to access to the actual file system. For example, if this implementation is used in the SmartMet Grid Engine, then grid files must be found from the file system, which is visible for the current SmartMet Server.

2. The CORBA client implementation allows the remote usage of the actual Data Server. For example, the previous figure shows how command-line client programs are able to access the Data Server that is located into another computer.

3. The Caching Data Server implementation can be used for caching information that was recently fetched from the Data Server. For example, if the actual Data Server is a remote server then the client side might contain the Caching Data Server implementation. This implementation is not fully complete yet. In other words, it caches only some parts of the information meanwhile quite many of the service requests just go throuh it.

Remote usage of the Data Server makes sense when there are several systems that want to access the same grid files.

As was mentioned earlier, the Data Server maps grid files into the virtual memory. It each system uses the Data Server locally then the Data Server uses their virtual memory. However, if the current systems do not have much physical memory, then the Data Server cannot access grid files very fast, because older information is continuously swapped out of the physical memory.

On the other hand, if there is one Data Server running in a computer that have a lot of physical memory, then more grid files can be in the physical memory at the same time. That's why it is much faster for other systems to fetch grid information from the Data Server that has a lot of physical memory and which has already memory mapped most of the grid files than to use own memory mappings with limited amount of physical memory.

4.3 Virtual Grid Files

The current implementation of the Data Server supports creation of virtual grid files. In practice, virtual grid files are grid files that can be mathematically generated from physical grid files.

For example, we might have a grid file that contains temperature values expressed in Kelvins, but we need the same temperature values expressed in Celsius. In this case the Data Server can create a virtual grid file and register it into the Content Information Storage. When we request data from the current virtual from the Data Server, it reads the requested temperature information from the physical grid file (where temperature value are expressed in Kelvins) and converts this information into Celsius. values before temperature values are returned.

The biggest benefit of this arrangement is that virtual filed do not require any space, because they use data from other grid files. The current information supports virtual file creation from one or two source files. For example, if we have wind speed expressed in U- and V- vector values (i.e. we have two grids), we can create a virtual grid file that uses both of these files for calculating the total wind speed.

Unfortunately this is a little bit complex operation from the Data Server point of view, which means that it should not necessary be used in very busy environments (i.e. when there are tens of thousands continuously changing grid files). The difficulty is that if we are using two different grid files for generation a virtual file, we cannot just pick any two grid files. In other words, they both need to have the same producer, they need to belong to the same generation, their forecast time, geometry, level type, level, etc. must match each other.

4.4 Getting started

Let's assume that you managed to get the Redis content server and also the CORBA Content Server up and running as was described in the previous chapter. The next step is to to start the CORBA Data Server, so that we can remotely access its services.

Notice that this is something that we do not necessary need to do when the Data Server is used as a part of the Grid Engine. However, the purpose of this demo is to show how easy it is to use a remote Data Server.

So, the start situation of this demo is that:

1. We have Redis database running.

2. We have registered three grid files into the Content Information Storage (i.e. to Redis database).

3. We have corbaContentServer running and its service IOR is saved to the SMARTMET_CS_IOR environment variable.

In order to start the remote Data Server, we should execute the following steps:

STEP 1:
Set the following values into the the "**corba-data-server.cfg**" configuration file:

```
data-server.address = "127.0.0.1"
data-server.port = 8201
data-server.iorFile = "/dev/stdout"
data-server.content-source.type = "$(SMARTMET_CS_IOR)"
data-server.grid-storage.directory = ""
```

STEP 2: Start the "**corbaDataServer**" program with the following command:

```
corbaDataServer corba-data-server.cfg
```

If everything went well, the program printed its International Object Reference (IOR) into the display. Set this value to the **SMARTMET_DS_IOR** environment variable like this:

```
export SMARTMET_DS_IOR="IOR:010000003600000049444c3a53432172744d65742f43665432788654243af43465722f436f7262612f53657276696365496e746572666163653a312e3000000001000000000000006800000001010200 0a0000003132372e302e302e3100082016000000ff6d7920706f6100436f6e74656e7453657276696365500000200000000000000800000001000000005454410100000001c000000010000000100010001000000010001050901010001000032189932 14"
```

After that you should be able to access the Data Server by using its CORBA interface with the command-line client programs. However, you should probably first ask from the Content Information Storage what kind of grid information is available. You can do this for example by the following command.

```
cs_getContentList 0 1 0 10
```

Now you can request information from the actual grid files. In practice you identify the grid file by using the grid file identifier that you fetched from the Content Information Storage. A grid file can contain several grids. That's why you need to identify the grid inside the current file by using the message index that you received from the Content Information Storage. In addition, you have to define the location (= location in the grid or geographical location) of the grid point which value you want to ask.

For example, you can ask the value of a specific grid point by using the following command-line client program:

```
ds_getGridValueByPoint <sessionId> <fileId> <messageIndex> <flags> <coordinateType> <x> <y> <interpolationMethod>
```

The "flags" parameter is reserved for future use. The "coordinateType" parameter can have one of the following values:

1	The point location (x,y) is expressed in longitudes (=x) and latitudes (=y).
2	The point location (x,y) is expressed in grid xy-coordinates.
3	The point location (x,y) is expressed by using the original projection coordinates (gaussian,lambert, etc)

The "interpolationMethod" parameter can have one of the following values:

1	Linear interpolation
2	Nearest value (= the value of the closest grid point)
3	Minimum value (= the smallest value of surrounding four greed points)
4	Maximum value (= the biggest value of surrounding four greed points)
500	Values of the closest four points and the relative position of the requested point.
501	Values and angles of the closest four points and the relative position of the requested point.

For example, the following command fetches the value of the grid point (25,43) from the grid "1" located in the grid file "123". The returned value is not interpolated.

```
ds_getGridValueByPoint 0 123 1 0 2 25 43 2
```

On the other hand, we could fetch values from the same grid by using latlon coordinates and use linear interpolation:

```
ds_getGridValueByPoint 0 123 1 0 1 23.80 65.20 1
```

5 Query Server

5.1 Introduction

The Query Server offers a simple way to query information from grib files. In practice, it uses the Content Server and the Data Server in order to fetch the requested data.

Technically it is very easy to fetch data even without the Query Server. However, this is the case only when we know exactly what we are looking for. In order to create an exact query, we need to define the following query parameters:

1. Producer (who is the producer of the requested data)
2. Generation (which generation we should use)
3. Parameter (which forecast parameter we are looking for)
4. Level type (the level type used to indicate parameter levels)
5. Level (the level value in the given level type)
6. Forecast type and number
7. Geometry (which grid geometry we should use if there are several geometries available)
8. Forecast time (time or time range of the requested data)
9. Location (the coordinates of the requested data)

Unfortunately, most of the queries contain only part of this information. That's why the most important functionality of the Query Server is to fullfill incomplete queries before fetching any data. This fulfilling is based on several configuration files.

On the other hand it is good to realize that the Query Server is used for fetching information from geographical points or areas rather than plain grid points. This means in practice that when searching grids the Query Server needs to check that the grid covers the requested geographical points or areas. In order to do that each grid must have a geometry identifier that can be used for fetching the actual geographical information about the area that the current grid covers.

The Query Server is used from the timeseries plugin and the WFS plugin. This means in practice, that their queries are fulfilled and executed in the same way. So, the query logic should not be a part of the plugin itself.

5.2 Query parameters

The Query Server support query parameters that can contain more detailed information about the requested parameters than the query parameters used earlier with the Timeseries Plugin. Instead of using the same global attributes (like producer, level, origintime, etc.) for each query parameter, we can define these attributes separately for each query parameters. In this way we can for example query temperature values from the different producers at the same time. When using the Timeseris plugin we can fetches temperature value from three different producers like this:

http://smartmet.fmi.fi/timeseries?place=helsinki&param=TIME,T-K:SMARTMET,T-K:HL2,T-K:ECG

And the result would be something like this:

TIME	T-K:SMARTMET	T-K:HL2	T-K:ECG
20180917T110000	286.9	287.4	285.8
20180917T120000	287.3	287.7	286.6

We can also query parameters from different levels:

http://smartmet.fmi.fi/timeseries?place=helsinki&param=TIME,T-K:HL2:6:0,T-K:HL2:6:2

The result shows temperatures from heights 0m and 2m:

TIME	T-K:HL2:6:0	T-K:HL2:6:2
201809181400	288.3	288.6
201809181500	288.1	288.6

A query parameter can contain the following fields (separated by ':' character):

1. Parameter name
2. Producer name
3. Level identifiers
4. Level
5. Forecast type
6. Forecast number
7. Generation flags
8. Area interpolation method
9. Time interpolation method
10. Level interpolation method

If these fields are not defined then the Query Server tries to find the closest matching parameter from the mappings files, which are described a little bit later.

Query parameters can also contain functions that can for example calculate different values according to their parameters. For example, we can use the "SUM" function for converting Kelvin temperature values into Celsius. values:

http://smartmet.fmi.fi/timeseries?place=helsinki&param=TIME,T-K,SUM{T-K;-273.16}

The request returns as two temperature values. The first value is the original Kelvin value, the second value is converted from the Kelvin by using the "SUM" function. This function just add all its parameters together. So, when we add "-273.16" to a Kelvin value, we get a Celsius value.

TIME	T-K	SUM{T-K;-273.16}
20180917T140000	287.8	14.7
20180917T150000	288.0	14.9

The current function is defined in a configuration file by using LUA programming language. The function takes a list of float numbers as an input, counts the sum of these numbers, and returns the result.

```
function SUM(numOfParams,params)

    local result = {};
    if (numOfParams > 0) then
        local sum = 0;
        for index, value in pairs(params) do
            if (value ~= ParamValueMissing) then
                sum = sum + value;
            else
                result.message = 'OK';
                result.value = ParamValueMissing;
                return result.value,result.message;
            end
```

```
        end

        result.message = 'OK';
        result.value = sum;
    else
        result.message = 'No parameters given!';
        result.value = 0;
    end
    return result.value,result.message;

end
```

LUA functions can also return for example text strings according to the value of the requested parameter. For example, we can express wind directions in degrees or in text:

http://smartmet.fmi.fi/timeseries?place=helsinki&param=TIME,WindDirection,NB_WindCompass16{WindDirection}

TIME	WindDirection	NB_WindCompass16{WindDirection}
201809181600	240.2	WSW
201809181700	230.2	SW

This means that we can easily express all possible parameter values in text if we want to do so. This gives us a lot of new opportunities to textually describe different weather parameters by using multiple languages. The point is that these definitions are not hard-coded as they were in older versions. We can use new parameters and created new expressions just by editing LUA files.

We can use the same LUA functions also when requesting data over geographical areas. When we are requesting values from an area, we usually get several values per time-step.

http://smartmet.fmi.fi/timeseries?area=turku&param=TIME,T-K

TIME	T-K
201809181500	[288.4 288.5 288.6 288.6]
201809181600	[288.8 288.9 288.8 288.7]

If we want to make calculations over these values, we need to use '@' character in the front of the LUA function name. In this way the Query Server understands that the current value list should be delivered to the requested function as a list of float values.

For example, the following request contains three of this kind of function.

http://smartmet.fmi.fi/timeseries?area=turku&param=TIME,T-K,@AVG{T-K},@MIN{T-K},@MAX{T-K}

And the result would look like this:

TIME	T-K	@AVG{T-K}	@MIN{T-K}	@MAX{T-K}
201809181500	[288.4 288.5 288.6 288.6]	288.5	288.4	288.6
201809181600	[288.8 288.9 288.8 288.7]	288.8	288.7	288.9

In practice, we can define as many LUA functions as we want and use them in our queries. These functions are defined in LUA files, which are automatically loaded in use when they are changed. No restart is needed. The locations of LUA files are defined in the main configuration file (of the Grid Engine or the Query Server). If LUA files are edited, they will be automatically loaded without any restart of the system. Notice that we can use the same

5.3 Configuration

The Query Server can be used as an independent server, but usually it is used as an embedded part of the SmartMet Grid Engine. That's why the configuration file of the SmartMet Grid Engine usually contains a lot of Query Server related configuration parameters.

As was mentioned earlier, fetching grid information would be easy if we could define exactly what information to fetch. Unfortunately this is usually not the case. Because of this, we have to use several configuration files in order to define default behavior for the system when a query does not contain enough information.

For example, if a query does not define the producer of the requested information, then the Query Server just needs to select on. If a query does not define any level type or level value, it just needs to select one. The main purpose of the Query Server configuration files is to help it with these selections.

The names of the configuration files that the Query Server uses are defined in the main configuration file (of the Grid Engine or the Query Server). The point is that the main configuration usually defines connections, components and dynamic configuration files used by the system. This main configuration file is usually read only once, meanwhile dynamic configuration files are automatically read when they are changed.

The Query Server uses these dynamic configuration files for the following purposes:

1. Producer and geometry selection (=> Producer file)
2. Producer alias name definitions (=> Producer alias file)
3. Parameter selection (=> Mapping files)
4. Parameter mappings (=> Mapping files)
5. Parameter alias definitions (=> Alias files)
6. Parameter mapping functions (=> LUA files)
7. Query functions (=> LUA files)

5.3.1 Producer and geometry selection

Usually a query should contain a producer's name or its alias name. The producers' alias names can be defined in the producer alias file (which name is defined by the "**query-server.producerAliasFile**" configuration parameter). The producer alias file contains the producers official name (used in the Content Information Storage) and the alias name. Notice that there can be several alias names for the same producer.

```
SMARTMET:SMART
SMARTMET:SMT_FINLAND
HL2:HIRLAM_EUROPE
HL2:HIRLAM
```

If a query does not contain a valid producer name then we just need test different producers in the order that they are defined in the producers' configuration file.

It is good to realized that the same producer might have several grids with different geometries that contain the same information. For example, a producer might have a grid that contains an overall temperature forecast over the whole planet, but have also a grid that contains more detailed forecast over Europe or just over Finland. That's why we have to define a search order, not only for the producers, but also for their geometries.

This is done in the producers' configuration file (which name is defined by the "**query-server.producerFile**" configuration parameter). The current file is a CSV file that looks like this.

```
SMARTMET;1073;7.6km x 7.5km;
HL2;173;7.9km x 7.2km;
HL2;174;8.0km x 7.1km;
HL2;175;7.9km x 7.1km;
ECG;1007;11.1km x 11.1km;
```

The first field is the name of the producer. This name should be exactly the same that is used in the Content Information Storage. The second field is the grid geometry identifier and the last field is a free description of the current producer-geometry combination. In this example, it just describes an estimated size of the grid cell and this way helps us to pick the most accurate geometries first.

It is important to notice that each grid much have a geometry identifier that defines the grid's geographical area. Otherwise it would not be possible to request information from geographical areas, because we do not know which geographical area is covered by the current grid. These geometry identifiers are defined in the Grid-Files Library's configuration files. When a grid file is registered into the Content Information Storage also the geometry identifier must be in place. That's why adding a new geometry into the Grid-Files Library does not necessary have an immediately effect, if the same information is not found from the Content Information Storage.

If a query contains a producer's name then the Query Server goes through only geometries belonging to the current producer. Otherwise it starts from the beginning of the file and continues until if finds a producer and a geometry that has the requested data. A geometry is selected only if the all requested geographical points or the border points of the requested area are inside the current geometry.

Notice that this file is usually the main reason why a query fails. For example, if we are getting grid files from new producers or grid files that are using new geometries, we must add new definitions into this configuration file. The Query Server cannot search information from such producers or such geometries that are not defined in this file.

5.3.2 Generation selection

It is quite common that a producer is generating several "forecast sets" per day. Usually we want to use the latest of these sets, but occasionally we might want to fetch information also from older sets. At the moment the Query Server fetches data from the latest generation, but fulfills this data with the data from older generations if the latest set does not contain the full time range that was requested. This is the default behavior of the Query Server when a query does not contain any additional information of the generation selection.

This all means that if we want to get data from different generations, we need to define this in queries. At the moment there are two approaches for this.

1. We can use an analysis time / an origin time information in a query and this way the Query Server can pick the generation which analysis time is closest to the requested time. All requested grid parameters will be fetched from the current generation.

2. We can define a generation / generations separately for each query parameter. In this case we do not define exact generations by analysis / origin times. Instead, we use bit-masks to indicate which generations to use. In this case, if the first bit (bit 0) is '1' then we use the latest generation. If the second bit (bit 1) is '1' then we use the second latest generation, and so on. If the first and the second bit are both '1' then we can search parameter from the latest and the second latest

generations. By this arrangement, we can easily compare different forecast sets. For example, when using the Timeseries plugin we can fetch temperature values from the latest and the second latest generation (from the producer SMARMET) by the following request:

http://smartmet.fmi.fi/timeseries?place=helsinki&param=TIME,T-K:SMARTMET:::::1,T-K:SMARTMET:::::2

And the result would be something like this:

TIME	T-K:SMARTMET:::::1	T-K:SMARTMET:::::2
20180917T070000	285.2	285.4
20180917T080000	285.3	285.9

5.3.3 Parameter selection

When we have a lot of grid files from several producers, one of the most challenging task is to define how grids should be selected if queries do not contain all necessary details. For example, it is quite common that forecast information contains temperature forecasts for multiple levels and heights. If a query does not define any level or height then we need to select it automatically by using different configuration files.

On the other hand, it is quite common that some producers might use different parameter identifiers about the same data as other producers. For example, a producer A might identify the temperature parameter by using identifier "123" meanwhile producer B uses identifier "792" about the same data. If we want to make general temperature queries over different grid files we must be able to map these identifiers so that we know that they both mean the same thing.

In addition, we might have a need to use our own parameter names in queries instead of some strange numeric identifiers or letter combinations. For example, it is much easier for a user to remember the parameter name "Temperature" than its numeric identifier (for example, "792").

For all of these reasons, the Query Server uses so called parameter mapping files. There can be several different parameter mapping files for different purposes. The names of these mapping files are defined in the main configuration file (of the Grid Engine or the Query Server). The order of the mapping files names is also the search order of the mapping files.

Mapping files are CSV files which lines contain the following fields:

1. Producer name

2. Query parameter name

3. Type of the Content Parameter Identifier (to which the query parameter will be mapped)

1 = FMI_ID (Parameter number used by FMI Radon Database)
2 = FMI_NAME (Parameter name used by FMI Radon Database)
3 = GRIB_ID (Parameter number used by GRIB)
4 = NEWBASE_ID (Parameter number used by FMI Newbase)
5 = NEWBASE_NAME (Parameter name used by FMI Newbase)
6 = CDM_ID
7 = CDM_NAME

4. Content Parameter identifier / name

5. Parameter level identifier type

1 = FMI (Level identifier type used by FMI Radon Database)
2 = GRIB1 (Level identifier type used in GRIB-1 files)
3 = GRIB2 (Level identifier type used in GRIB-2 files)

6. Level identifier

FMI level identifiers:

1 Gound or water surface
2 Pressure level
3 Hybrid level
4 Altitude
5 Top of atmosphere
6 Height above ground in meters
7 Mean sea level
8 Entire atmosphere
9 Depth below land surface
10 Depth below some surface
11 Level at specified pressure difference from ground to level
12 Max equivalent potential temperature level
13 Layer between two metric heights above ground
14 Layer between two depths below land surface
15 Isothermal level, temperature in 1/100 K

GRIB-1 level identifiers

See GRIB-1 specification

GRIB-2 level identifiers

See GRIB-2 specification

7. Area interpolation method

0 = None
1 = Linear
2 = Nearest
3 = Min
4 = Max
500..999 = List
1000..65535 = External (interpolated by an external function)

8. Time interpolation method

0 = None
1 = Linear
2 = Nearest
3 = Min
4 = Max
1000..65535 = External (interpolated by an external function)

9. Level interpolation method

0 = None
1 = Linear
2 = Nearest
3 = Min
4 = Max
5 = Logarithmic
1000..65535 = External (interpolated by an external function)

10. Group flags

bit 0 = Climatological parameter (=> ignore year when searching)

11. Search match (Can this mapping used when searching mappings for incomplete parameters)

E = Enabled
D = Disabled

12. Mapping function (enables data conversions during the mapping)

A typical mapping file might contain lines like this:

```
GEFS;Temperature;2;T-K;1;2;100000;1;1;;0;D;SUM{$,-273.15};
GEFS;Temperature;2;T-K;1;2;85000;1;1;;0;D;SUM{$,-273.15};
GEFS;Temperature;2;T-K;1;6;00002;1;1;;0;E;SUM{$,-273.15};
GEFS;TotalCloudCover;2;N-PRCNT;1;8;00000;1;1;;0;E;MUL{$,100.000000};
GEFS;WindGust;2;FFG-MS;1;1;00000;1;1;;0;E;;
```

For example, the first line defines that when we are querying "Temperature" parameter from "GEFS" producer, we are actually searching parameter which FMI_NAME identifier has value "T-K". A mapping line can also contain preferred interpolation methods for area, time and level interpolation.

The order of the mapping lines in the mapping file is also the search order of the mappings in the current file. If we are searching the "Temperature" parameter without any level definitions, then the Query Server picks the third line, because its 11th field has value "E" (= enabled). In this case the current mapping line can be selected because all the other fields are matching, except the fields that are missing. If the 11th field is "D" then all fields must match.

Notice that in this example we are mapping the query parameter "Temperature" into a parameter, which temperature values were expressed in Kelvins, but we actually want to get them in Celsius. That's why the last field in the line contains the name of the conversion function (SUM) that the Query Server executes before it returns the query results. This is exactly the same LUA function that was used earlier with the query parameters.

5.3.4 Automated parameter mapping

When new producers or parameters are added into the Content Information Storage, also their mapping information should be added into the mapping files. Otherwise the Query Engine cannot find the new parameters.

In spite of that new mappings are automatically loaded without restart, adding new parameters all the time could be frustrating. That's why we have automated some part of it. At the moment, the Grid Engine / Query Server adds automatically missing parameter mappings into specific mapping files. This guarantees that also new parameters can be queried immediately.

The names of these automatically generated mapping files are defined in the main configuration file (of the Grid Engine or the Query Server). The file names usually have "_auto.csv" endings. At the moment, mappings are automatically generated for FMI Radon and Newbase parameters.

These automatically generated parameters should be moved into more permanent mapping files as soon as possible, because automatically generated mappings are continuously overridden. In other words, you should do not manually change them, because these changes will be overridden.

5.3.5 Alias files

Alias files are used for defining alias names for query parameters. After that these aliases can be uses as normal query parameters. The alias name can be just a short name for a parameter or even more complex query parameter combination. Sometimes alias definitions contain the '$' character in front of the parameter name. In this case the '$' character indicates that the parameter name refers to the real parameter - not to the alias that have the same name as the current parameter. In this way we try to avoid infinite recursion in our alias definitions.

An alias file might look like this:

```
# Temperature (height 2m)
t2m:$Temperature::6:2

# Picking the highest temperature value from three producers
tHigh:MAX{$T-K:HL2;$T-K:SMARTMET;$T-K:EC}
```

6 SmartMet Server

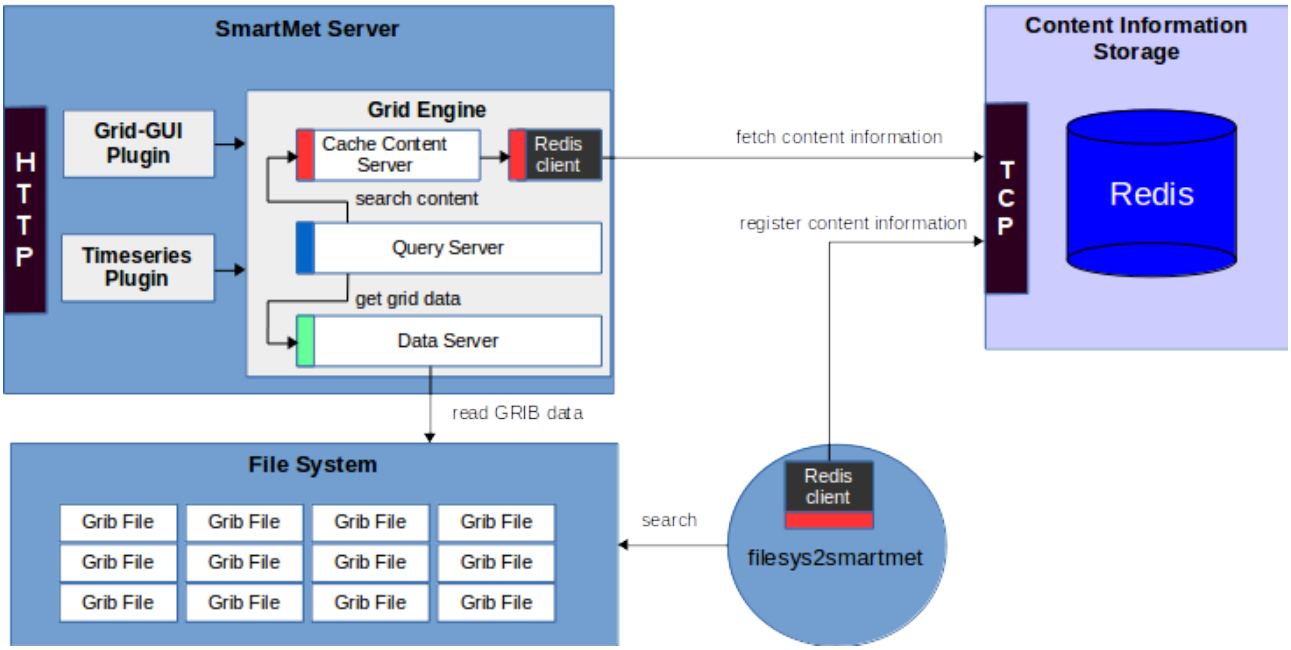
6.1 Introduction

So far we have introduced four main components used by the grid support. These component are:

1. Grid-Files Library
2. Content Server
3. Data Server
4. Query Server

In spite of that these components are developed in the SmartMet project, they do not necessary need SmartMet Server. However, if someone wants to use these components, he/she most likely want to use them via SmartMet Server's user friendly interfaces (Timeseries, Grid-GUI, WFS, etc.) rather than via programming interfaces that these components are offering.

The figure below shows the basic configuration of the SmartMet Server using the grid support.



In this configuration there is an external Content Information Storage (= Redis database) that contains the grid content information, meanwhile the actual grid information management is concentrated into the Grid Engine. In other words, the (Caching) Content Server, the Data Server and the Query Server are embedded into the Grid Engine.

This is the main reason why most of the configuration parameters we described in previous chapters are actually found from the Grid Engines configuration file. When using this kind of configuration, also the grid files must be visible to the Grid Engine (i.e. the file system must be mounted so that the Grid Engine can access it.

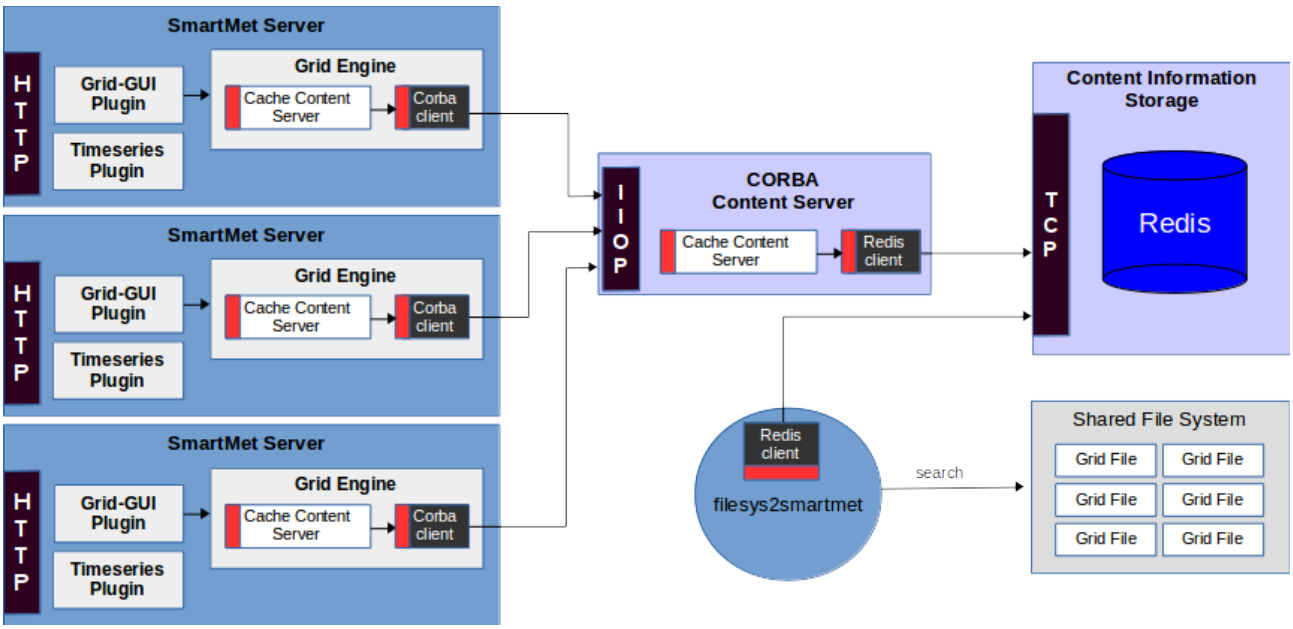
6.2 Architecture

Technically the grid support consists of a collection of flexible components that can be embedded into existing systems or be used as separate components. These components can be easily distributed and replicated, which makes the system real scalable. This does not required any programming. It requires only some tuning of configuration files.

For example, usually there is just one Content Information Storage in the system (and maybe some kind of backup). If there are a lot of continuous information updates going on (= millions of updates per day) and several SmartMet Server instances are polling these updates directly from the Redis Content Server, the load of this component can increase into quite high level.

In this case, the feeding systems (filesys2smartmet, radon2smartmet, etc.) are competing the usage of the same resources, which might slow down them all. In the worst case scenario, the feeding systems do not get enough time in order to make required content information updates into the Content Information Storage. Because of this, the SmartMet Server components are trying to access grid files that were already removed from the system.

That's not a real problem, becaus the grid support is designed to be extreme scalable. The figure below shows how a Caching Content Server can be installed in the from of the Content Information Storage. In this way we can easily reduce the load of Content Information Storage and make sure that the feeding systems get enough time in order to make required content information updates into the Content Information Storage.

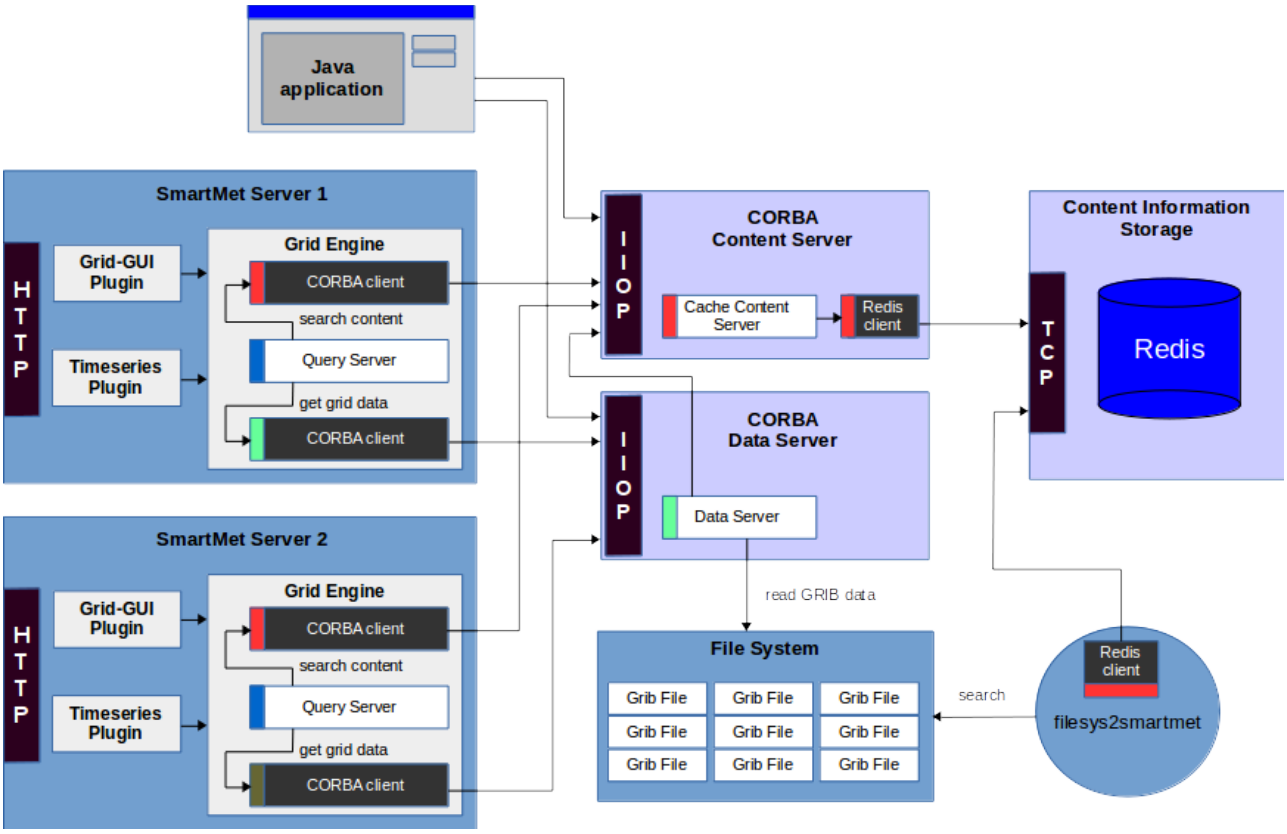


In the figure above, also the Grid Engines have Caching Content Server, which allows them to make fast queries to the content information.

The Caching Content Server is implemented so that all modification requests go through it. For example, if we send a request for adding a new FileInfo record to the Caching Content Server, it automatically forwards this request to the Master Content Server (=> Redis implementation), which adds the new record into the Redis database. In this case the Caching Content Server does not necessary immediately see the addition, because it takes few secods before it gets this information from the Master Content Server.

As was mentioned earlier, all grid support component are distributable. On the other hand, they are also accessible by other components and systems.

For example, Java applications could directly use the grid support services by calling the CORBA/IIOP service interfaces. This is shown in the figure below.



In this configuration, two SmartMet Servers are just similar users from grid services point of view than the Java Application.

This approach is typical for Service Oriented Architectures (SOA). This means in practice that we have a collection of services that everybody can use. Ability to use these kinds of services should not depend on the programming language (C, C++, Java, Python, etc.) or the usage environment (Linux, JRE, Windows, etc.).

6.3 Grid Engine

6.3.1 Introduction

The Grid Engine allows SmartMet Plugins to access services related to the grid support. This mean in practice that they can access information stored into the Content Information Storage by using the Content Server API, fetch grid data from the grid files by using the Data Server API or make queries by using the Query Server API.

All these APIs can be found from the Grid Engine. At the moment, the Grid-GUI Plugin and the Timeseries Plugin are using these services. In future, also the WFS Plugin and the Download Plugin should be able to use these services.

6.3.2 Configuration

The main configuration file of the Grid Engine is read only once when the server is started. The main configuration file of the SmartMet server should point to this file.

When the actual implementations of the grid services are embedded into the Grid Engine (i.e. not used remotely), all these services must be configured when the Grid Engine is configured. In this case the main configuration file contains a lot of references to other configuration files (mapping files, LUA files, alias files, etc.). These configuration files can be changed during the runtime and changes will be automatically loaded without any restart.

The Grid Engine can be configured also in a such way that all grid services are used remotely. In this case the Content Server API, the Data Server API and the Query Server API are client implementations that communicate with the remote services.

The main configuration file of the Grid Engine looks like this:

```
smartmet : { library : { grid-files :
{
        # We are using the Grid-Files Library. In order to use it, we need to initialize it first. That's why need to know the location of its main configuration file.
        configFile = "/smartmet/config/library/grid-files/grid-files.conf"

        # If the Data Server is local then the grid file cache can be used to improve performance. This cache is used caching uncompressed grid data when
        # the original data compression is slow. The grid data is first cached as uncompressed data. If the cached data is not accessed for awhile it will be
        # compressed (with a fast compression) and still kept in the cache. When the cache is full then the oldest data is automatically removed.

        # This caching functionality is a part of the Grid-Files Library. That's why it is configured here.
        cache :
        {
                numOfGrids                              = 50000
                maxUncompressedSizeInMegaBytes          = 10000
                maxCompressedSizeInMegaBytes            = 10000
        }

}}  // grid-files, library

engine : { grid :
{
        # The content server defines the source of the content information. Usually the master content source is the Redis database. However,it can be also
        # a memory-based content server or a caching content server, which are accessed by HTTP or CORBA. The Redis database is not usually fast enough
        # for our searching purposes. That's why  its information is sally cached locally into the memory. In this case the content  information can be sorted so
        # that it can be fetched very fast. In spite of that we can sort the current content information in many ways, we should not sort it all possible ways because
        # this increases the memory consumption and it also makes content updates slower. In practice, we should select the main content identifiers and sort
        # the content information according to them. All the other information should be mapped to those identifiers when querying content information.

        # If the content source itself is a caching content server then there is usually no need to locally cache the same information. Caching content information
        # makes content searching very fast but it might required a lot of memory. That's why it usually makes sense to use a remote caching server if that is
        # possible. On the other hand this means that the grid engine can start much faster because it does not need to cache content information first.
```

```
content-server :
{
    content-source :
    {
        # Content source type (redis / corba / http).
        type = "redis"

        # These parameters needs to be defined when the content source type is "redis".
        redis :
        {
            address     = "127.0.0.1"
            port        = 6379
            tablePrefix = "a."
        }

        # These parameters needs to be defined when the content source type is "corba".
        corba :
        {
            ior         = "$(SMARTMET_CS_IOR)"
        }

        # These parameters needs to be defined when the content source type is "http".
        http :
        {
            url         = "$(HTTP_CONTENT_SERVER_URL)"
        }
    }

    # If the content source  is a Redis database then you probably want to use locally the Caching Content Server. That's makes queries fast. In this
    # case you should define how the cache information should be sorted.

    cache :
    {
        enabled = true

        # Content sorting flags:
        # ----------------------
        #  bit 0 (1)  : reserved
        #  bit 1 (2)  : Sort by fmi-id (radonParameterId)
        #  bit 2 (4)  : Sort by fmi-name (radonParameterName)
        #  bit 3 (8)  : Sort by grib-id
        #  bit 4 (16) : Sort by newbase-id
        #  bit 5 (32) : Sort by newbase-name
        #  bit 6 (64) : Sort by cdm-id
        #  bit 7 (128): Sort by cdm-name

        # (1 + 4)
        contentSortingFlags = 5
    }

    # The processing log is logging service requests and times used for these request. The current log truncates itself automatically when it reaches
    # the maximum size of the current log file. The "truncateSize" is the size of the log file after the truncate operation.

    processing-log :
    {
        enabled      = false
        file         = "/smartmet/logs/contentServer_processing.log"
        maxSize      = 100000000
        truncateSize = 20000000
    }

    # The debug log is used in order to find out what the content server is doing. The current log truncates itself automatically when it reaches
    # the maximum size of the current log file. The "truncateSize" is the size of the log file after the truncate operation.

    debug-log :
    {
        enabled      = false
        file         = "/smartmet/logs/contentServer_debug.log"
        maxSize      = 100000000
        truncateSize = 20000000
    }
}
```

```
# The Data Server is responsible for fetching actual data from the grid files. It is possible to use a local or a remote Data Server. The Data Server uses also
# a lot of memory and maps grid files into virtual memory. That's why it would be smarter to use shared Data Servers when possible. On the other hand,
# the remote Data Server is usually always up and running which means that it can be used immediately. If the Data Server is local then it takes some time
# to start the system and make sure that all grid files are available. If the remote Data Server is disabled then the local Data Server is used in the engine.

data-server :
{
    ######  The remote data server. Notice that the remote data server has its own configuration file.

    remote    = true
    caching   = false
    ior       = "$(SMARTMET_DS_IOR)"

    ######  The local data server. These settings are valid when the "remote" attribute is "false".

    # Location of grid files. If the "preloadEnabled" is true then the Data Server memory maps grid files that contain grids, which are marked with
    # the preload flag in the Content Information Storage. Notice that the memory mapping is done on the file level. So, the whole file is memory mapped
    # even if if contains only one important grid. In the worst case, a grid file might contain hundreds of grids that are not so important.

    grid-storage :
    {
        directory       = ""
        preloadEnabled  = false
    }

    # The data server can generate "virtual grid files" that are based on existing grid files. The virtual file definition file is used in order to define
    # requirements (= required parameters) and rules (= LUA function) for new virtual files.

    virtualFiles :
    {
        enabled         = false
        definitionFile  = "%(DIR)/vff_convert.csv"
    }

    # LUA files are usually needed for generating content for the virtual files. In practice, each virtual file definition contains the name  of the LUA function
    # that needs to be called when the data of  the current virtual file is requested.

    luaFiles = ["%(DIR)/vff_convert.lua"];

    # The processing log is logging service requests and times used for these request. The current log truncates itself automatically when it reaches
    # the maximum size of the current log file. The "truncateSize" is the size of the log file after the truncate operation.

    processing-log :
    {
        enabled         = false
        file            = "/smartmet/logs/dataServer_processing.log"
        maxSize         = 100000000
        truncateSize    = 20000000
    }

    # The debug log is used in order to find out what the data server is doing. The current log truncates itself automatically when it reaches
    # the maximum size of the current log file. The "truncateSize" is the size of the log file after the truncate operation.

    debug-log :
    {
        enabled         = false
        file            = "/smartmet/logs/dataServer_debug.log"
        maxSize         = 100000000
        truncateSize    = 20000000
    }
}

# The Query Server is responsible for making data queries to the Data Server according to the content information that it gets from the Content Server.
# Also the Query Server can be local or remote. Sometimes it is smarter to locate the Query Server closer to the Content Server and the Data Server,
# because there might be a lot traffic between them. If the remote Query Server is disabled then the local Query Server is used in the engine.

query-server :
{
    remote              = false
    ior                 = "$(SMARTMET_QS_IOR)"

    # This file defines the search order of the producers and their geometries when querying data.
    producerFile        = "%(DIR)/producers.csv"


    # This file maps producer alias names to their "official names" used in the Content Information Storage.
```

```
        producerAliasFile      = "%(DIR)/producerAlias.cfg"

        # If the grid Engine notices that there are not valid mappings available for all parameters found from the Content Server, then it automatically adds
        # these unmapped parameters into the following files. The Query Server needs these mappings in order to find the current parameters. You can
        # move these automatically generated mappings into a more permanent mapping file (mappings_fmi_auto.csv => mappings_fmi.csv). On the other
        # hand, when the permanent mapping files are empty then you can get your basic mapping settings from these automatically generated files (i.e. when
        # you are first time configuring your system).

        # Mappings should be generated against "the main key type" used by the Content Server. If the content information is locally cached the current
        # cache should be sorted by this key type (see the "contentSortingFlags" parameter defined earlier in this file). Available options are:  1 = FMI_ID,
        # 2 = FMI_NAME, 3 = GRIB_ID, 4 = NEWBASE_ID, 5 = NEWBASE_NAME, 6 = CDM_ID, 7 = CDM_NAME
        mappingTargetKeyType = 2

        # These files are generated automatically:
        mappingUpdateFile :
        {
            fmi                 = "%(DIR)/mapping_fmi_auto.csv"
            newbase             = "%(DIR)/mapping_newbase_auto.csv"
        }

        # Notice that it is also possible to use some "external" programs or scripts in order to generate own automated mapping files. The point is that
        # the current automated mappings uses parameters that Finnish Meteorological Institute is using. Other organization might want to develop their
        # own automated mapping files. This is possible, because changed mapping files are automatically taken in use.

        # The following files contains all the mappings that are in use. Mapping files are processed in the same order as they are listed here. Notice that also
        # the automatically generated mapping files need to be listed here if we want to use them.
        mappingFiles =
        [
            "%(DIR)/mapping_fmi.csv",
            "%(DIR)/mapping_fmi_auto.csv",
            "%(DIR)/mapping_newbase.csv",
            "%(DIR)/mapping_newbase_auto.csv",
        ];

        # These files contains alias definitions for query parameters.
        aliasFiles =
        [
            "%(DIR)/alias_newbase_extension.cfg"
        ];

        # These files contains LUA functions that can be used in parameters mappings and also in query functions.
        luaFiles =
        [
            "%(DIR)/function_basic.lua",
            "%(DIR)/function_interpolation.lua",
            "%(DIR)/function_conversion.lua",
        ];

        # The processing log is logging service requests and times used for these request. The current log truncates itself automatically when it reaches
        # the maximum size of the current log file. The "truncateSize" is the size of the log file after the truncate operation.
        processing-log :
        {
            enabled             = false
            file                = "/smartmet/logs/queryServer_processing.log"
            maxSize             = 100000000
            truncateSize        = 20000000
        }

        # The debug log is used in order to find out what the query server is doing. This log is useful when we try to find out why our query does not
        # return the expected result. The current log shows how queries are processed.However, it produces so much information that it should
        # be used only for debugging. The current log truncates itself automatically when it reaches  the maximum size of the current log file.
        # The "truncateSize" is the size of the log file after the truncate operation.
        debug-log :
        {
            enabled             = false
            file                = "/smartmet/logs/queryServer_debug.log"
            maxSize             = 100000000
            truncateSize        = 20000000
        }
    }
}}}
```

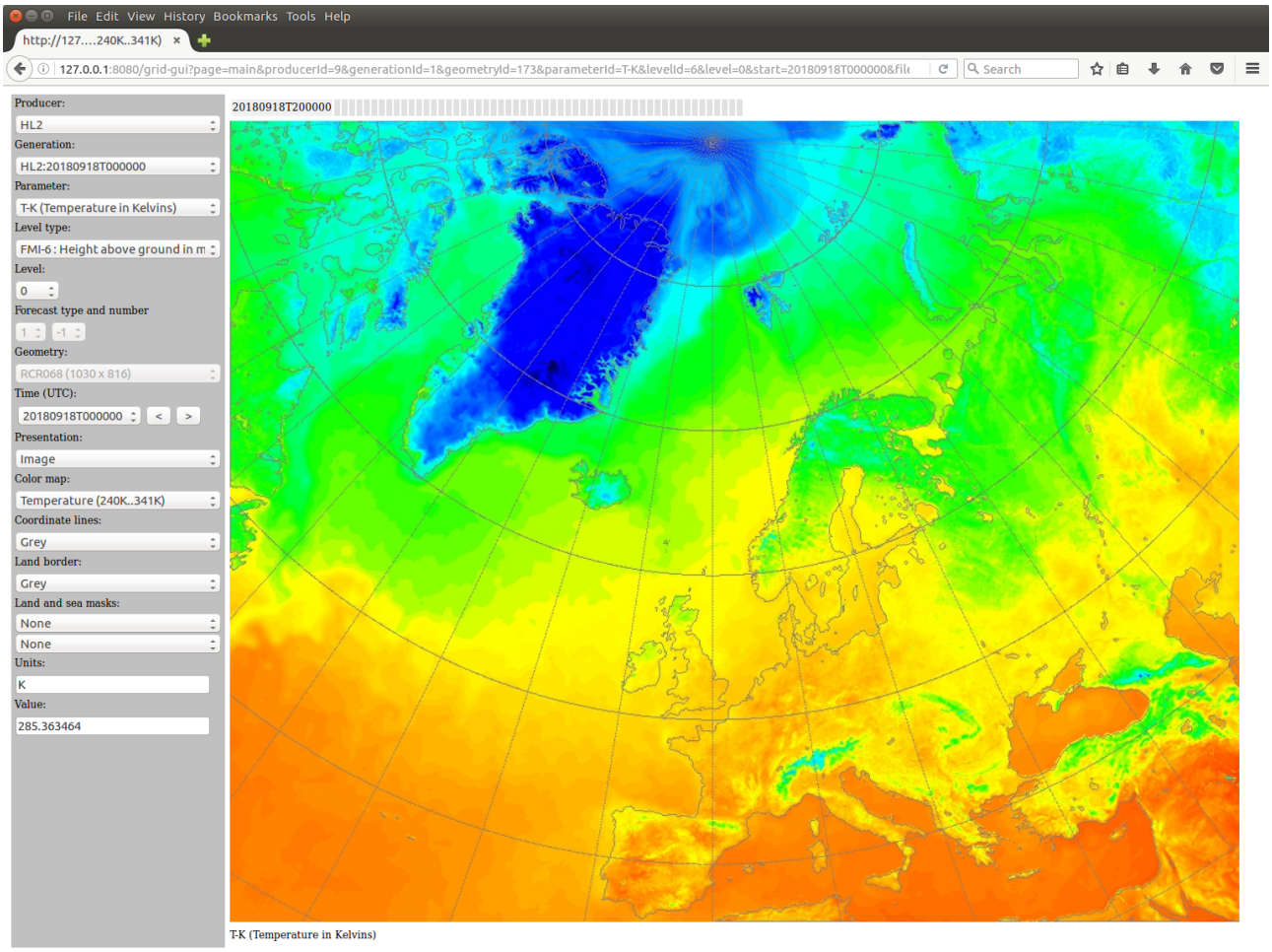
6.4 Grid-GUI Plugin

6.4.1 Introduction

The Grid-GUI Plugin can be used in order to visualize grid files, which information is registered into the Content Information Storage. Originally this plugin was created in order to help the development work in the grid support project. In other words, we just wanted to see that our key components were working correctly and they can open different kinds of grid files and support multiple projections. In addition, it offered us a nice way to see almost all information that was stored into the Content Information Storage.

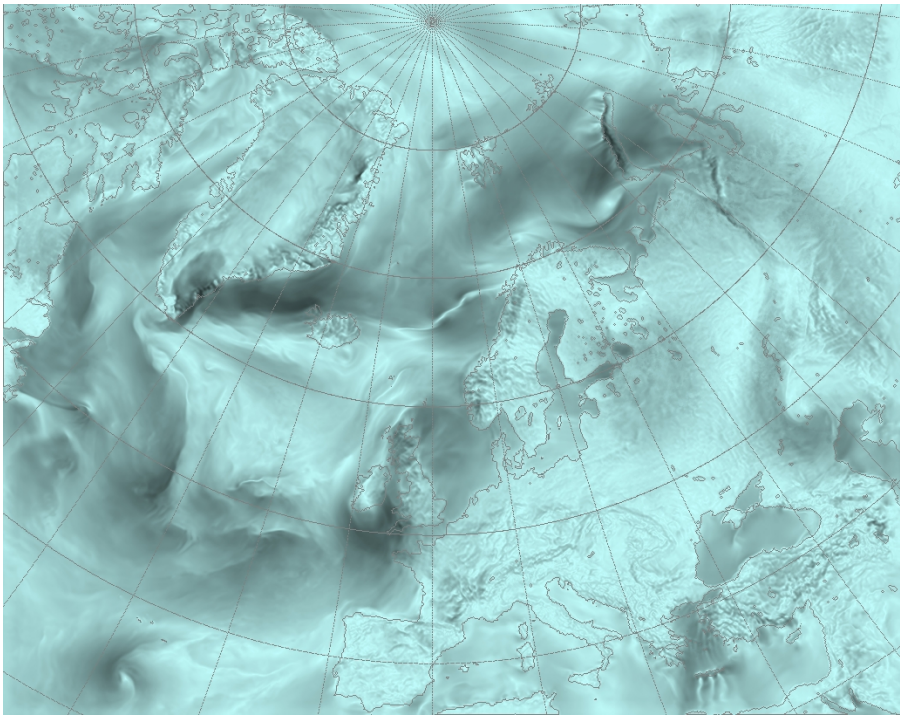
We noticed that the Grid-GUI was very useful and that's why we decided to publish it. However, it is good to understand that this is a very special tool and it was not originally designed for very wide use. In other words, in spite of that this GUI is quite functional, we could develop even better GUIs if we had more time for this. In practice, this GUI demonstrates some of the capabilities that the grid support brings to developers.

The figure below shows the basic outlook of the Grid-GUI.

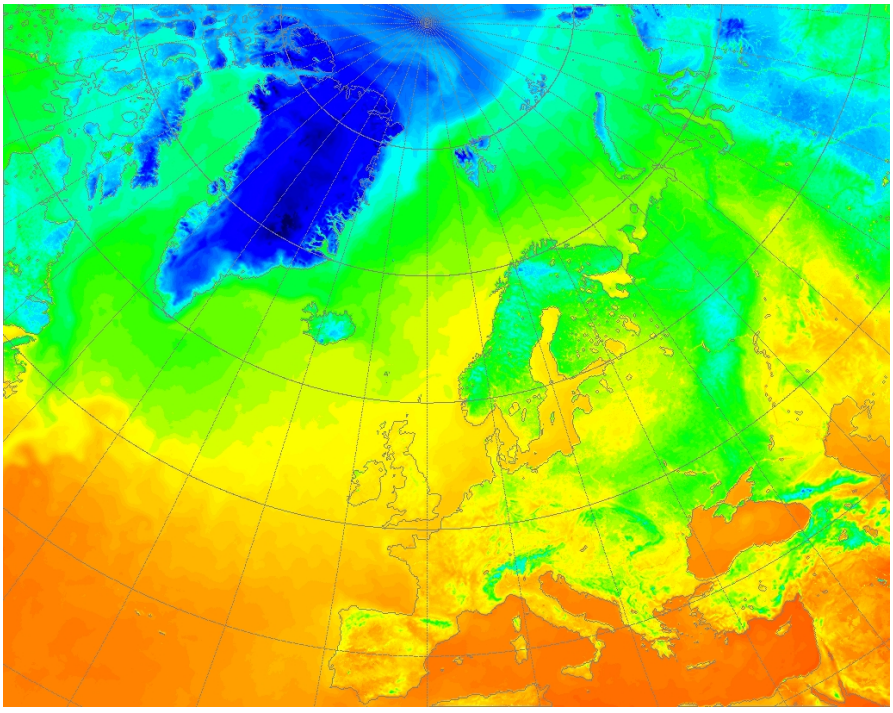


6.4.2 Visualization

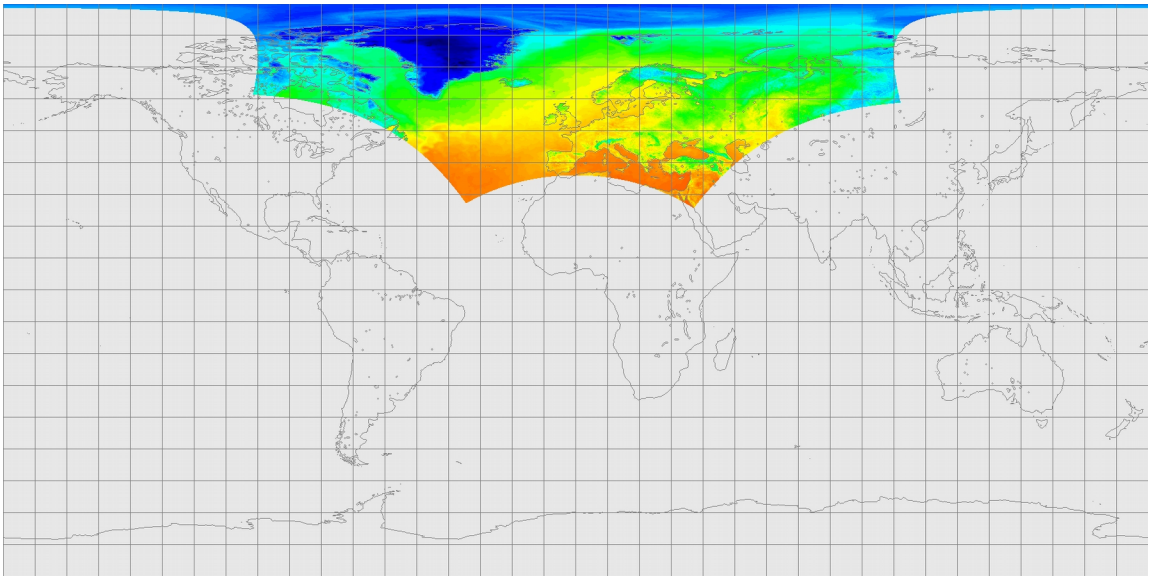
The basic idea of the visualization is that we imagine grids to be two dimensional images that contain different colors. I.e. grid values can be changed to colors. For example, if we have a grid that contains wind speed values we can convert these values to colors so that bigger values (= strong wind) are converted to darker colors and smaller values (= weak wind) are converted to lighter colors.



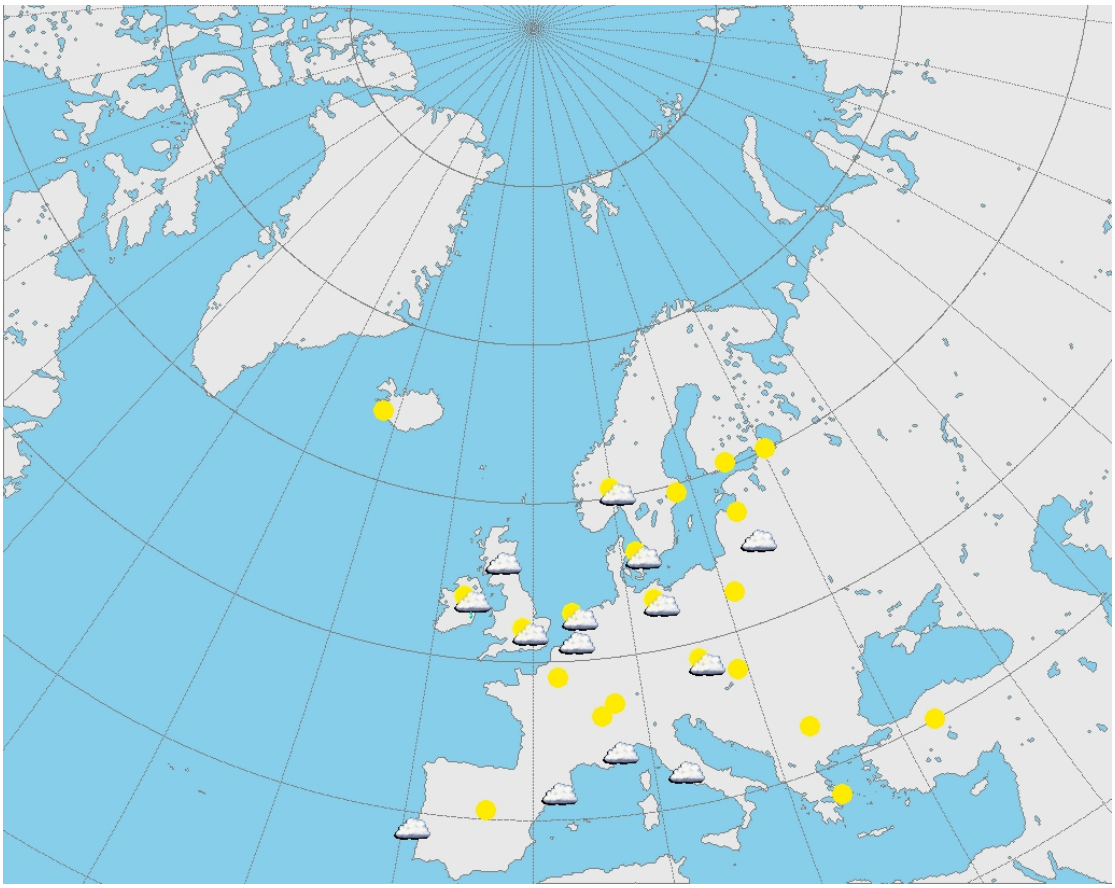
On the other hand, we can define actual color mappings for some grid values. In this case we need to use configuration files where different value ranges are mapped into different colors. The figure below shows, how different temperature values are mapped into different colors.



We have added continental borders and coordinate lines in the previous images. These help us to geographically locate the current grid information. In some cases this is not enough. For example, a grid can be from an area where no recognizable borders are available. In this case we can draw the grid into the world map and this way see its geographical location.



If a grid contains some kind of state of symbol values, we can map these values into actual symbols. We can also define locations for these symbols. For example, we can locate symbols so that they cover all the main cities. In future, we are probably doing the same thing with numbers (for example, showing temperature values in selected locations).



6.4.3 Configuration

The main configuration file of the Grid-GUI Plugin is read once when the server is started. The main configuration file of the SmartMet Server should point to this file.

The configuration file contains a lot of references to other files (color maps, symbol maps, locations, etc.) these files are dynamical configuration files, which are loaded in use automatically if they change. Unfortunately, we can only update these files without restart, but if we want to add new color maps, symbols maps or locations files, we have to restart the system. This is something that we should change in future.

A typical Grid-GUI Plugin configuration file looks like this:

```
smartmet :
{
plugin :
{
grid-gui :
{
    # We are using the Grid-Files Library. In order to use it, we need to initialize it first. That's why need to know the location of its main configuration file.
    grid-files :
    {
        configFile = "/smartmet/config/library/grid-files/grid-files.conf"
    }

    # The land-sea-mask JPG file. This file is used in order to find out if a grid point is over the land or over the sea. This map covers the whole globe, but
    # it is not very accurate. For example, if the grid covers only a small geographical area then the land border would be quite rough.

    land-sea-mask-file = "%(DIR)/land-sea-mask.jpeg"

  # File that contain color name and value definitions, which are shown in the color value lists (for land border, land/sea mask, etc.)

  colorFile = "%(DIR)/colors2.csv"

  # Color mapping files. These files are used for mapping grid values to actual colors. Notice that the same mappings files can be used for multiple
  # purposes. That's why, we should be able to reference these values with multiple names. These names are defined in the beginning of these files and
  # they are visible in the Color Map selection list.

  colorMapFiles :
  [
      "%(DIR)/colormaps/values_-33_to_68.csv",
      "%(DIR)/colormaps/values_240_to_341.csv",
      "%(DIR)/colormaps/values_0_to_1.csv",
      "%(DIR)/colormaps/values_0_to_10.csv",
      "%(DIR)/colormaps/values_0_to_100.csv",
      "%(DIR)/colormaps/values_-42_to_42.csv"
  ]

  # Symbol mapping files. These files are used for mapping grid values to symbols. It is quite common that there are multiple files for same symbols. In
  # this case only symbol sizes are different.

  symbolMapFiles :
  [
      "%(DIR)/symbolmaps/weather-hessaa-30.csv",
      "%(DIR)/symbolmaps/weather-hessaa-40.csv",
      "%(DIR)/symbolmaps/weather-hessaa-50.csv",
      "%(DIR)/symbolmaps/weather-hessaa-60.csv",
  ]

  # When using symbols or showing single values we need to define locations which to use. For example, we might want to show symbols so that they
  # cover the main cities in the area. That's why we can define location lists, which contain location names and coordinates. These lists are defined
  # in separate location files.

  locationFiles :
  [
      "%(DIR)/locations/europe-main-cities.csv",
      "%(DIR)/locations/finland-main-cities.csv"
  ]
```

```
    # On the top of the Grid-GUI there is a list of gray block that can be used for changing images fast. This is quite nice feature, but from the server's point
    # of view this kind of animation is quite heavy operation. That's why it not necessary allowed when there are a lot of users. This  animation feature can
    # be enabled or disable by this parameter:

    animationEnabled = true

    # The Grid-GUI caches some of the images that it generates. This is especially important when the animation feature is used. The following parameters
    # define the cache location and its size limits.

    imageCache :
    {
        # Image storage directory
        directory = "/tmp/"

        # Delete old images when this limit is reached
        maxImages = 1000

        # Number of images after the delete operation
        minImages = 500
    }
}
}
}
```

6.5 Grid-Admin Plugin

6.5.1 Introduction

The Grid-Admin Plugin is a simple plugin that offers an HTTP interface to the Content Information Storage. So, technically other system (for example, feeding systems) can use this interface for updating and fetching content information from the Content Information Stroage

The Grid-Admin Plugin does not use the Grid Engine. Instead, it connects directly to the Content Server by using its own configuration information.

6.5.2 Configuration

The main configuration file of the Grid-Admin Plugin is read once when the server is started. The main configuration file of the SmartMet Server should point to this file.

The configuration file look like this:

```
smartmet :
{
plugin :
{
grid-admin :
{
        content-server  :
        {
                # Content server type (redis / corba / http).
                type = "redis"

                # These parameters must be updated when the content server type is "redis".
                redis :
                {
                        address         = "127.0.0.1"
                        port            = 6379
                        tablePrefix     = "a."
                }
```

```
                # These parameters must be updated when the content server type is "corba".
                corba :
                {
                        ior             = "$(SMARTMET_CS_IOR)"
                }

                # These parameters must be updated when the content server type is "http".
                http :
                {
                        url             = "http://smartmet.fmi.fi/grid-admin"
                }
        }
}
}
}
```