Integrated Reservoir Model

Introduction

The Integrated Reservoir Model is a General Adapter module written in Java, developed by Deltares. The model class is part of the Delft-FEWS code base.

The Reservoir model is developed and designed as a relatively straightforward reservoir routing model, that simulates flow through a reservoir with a level-release table defined. Specifically, the model is able to precisely replicate the uncontrolled outlet reservoir behaviour of the legacy Deltares RTC-Tools 1 codebase, which is no longer developed and supported. Because the adapter model is based on Java, it can run on Windows/Linux systems.

The model is introduced in the 2021.01 BoM Delft-FEWS version. The model adapter will be part of the Delft-FEWS code base of Delft-FEWS versions 2023.01 and onwards with in-memory options, to be set in the General Adapter configuration.

Directory structure

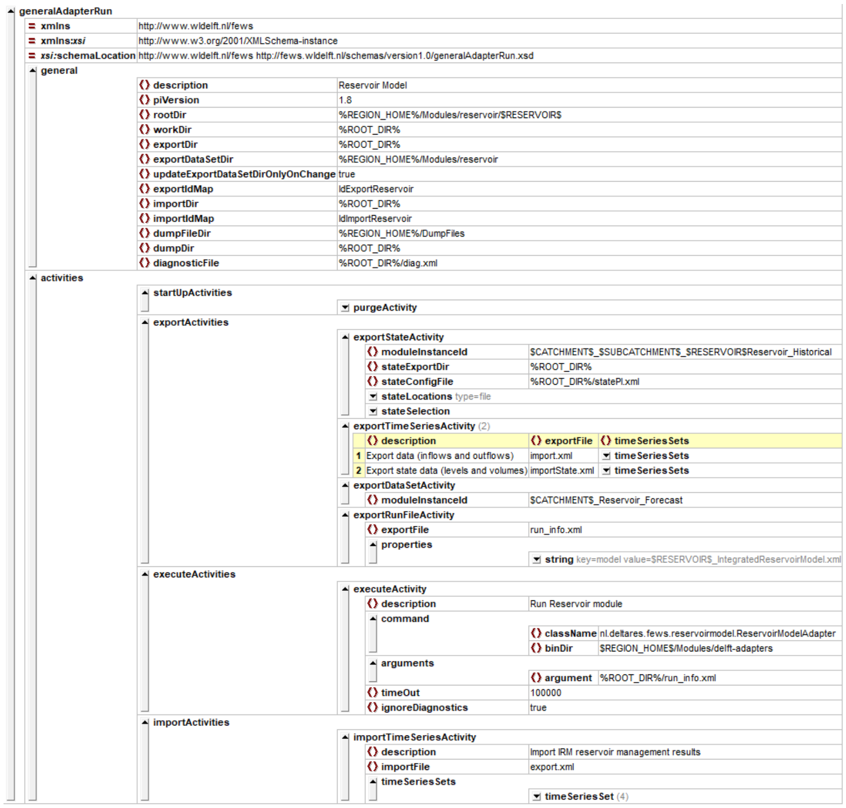
The data directories and configuration files that are required for operating the FEWS Reservoir Model Adapter are shown below.

```
FEWS_SA
+---Config
|   +---ColdStateFiles
|   |   |       namoi_keepit_KeepReservoir_Historical_IRM Default.zip........coldState file
|   +---IdMapFiles
|   |       IdExportIRMReservoir.xml
|   |       IdImportIRMReservoir.xml........................................custom mappings for the IRM
variables and locations
|   |
|   +---ModuleConfigFiles
|   |   |   Reservoir_1h_Forecast_IRM.xml...................................main configuration file of the
adapter
|   |
|   +---ModuleDataSetFiles
|   |   |   Reservoir_Exe.zip...............................................zipped IRM bin files, transported
to Modules\reservoir directory
|   |   |
|   |   |   namoi_IRM_Reservoir_Forecast.zip................................zipped IRM model files for a
specific reservoir, transported to Modules\Reservoir directory
|
+---Modules
|   +---delft-adapters......................................................directory which contains all IRM
adapter java files
|   |   |
|   +---reservoir
|   |   +---Keep_IRM
|   |   |       diag.xml....................................................output FEWS-PI diagnostics file,
imported by Delft-FEWS
|   |   |       export.xml..................................................output FEWS-PI time series files,
imported by Delft-FEWS
|   |   |       exportState.xml.............................................FEWS-PI state output time series
file, imported by Delft-FEWS
|   |   |       import.xml..................................................input FEWS-PI time series files,
exported by Delft-FEWS
|   |   |       importState.xml.............................................FEWS-PI state input time series
file, exported by Delft-FEWS
|   |   |       Keep_IntegratedReservoirModel.xml...........................IRM model file
|   |   |       run_info.xml................................................a file generated by FEWS
containing paths, run options
|   |   |       statePI.xml.................................................PI State file (definition)
|   |   |
```

General Adapter configuration

The General Adapter defines forms the interface between the Delft-FEWS system and the Reservoir model.

The data is provided in a standardized XML interface format, the FEWS Published Interface. For more details about general structure of the General Adapter please check 05 General Adapter Module.



general

Configuring a pi-version 1.8 is required for the diagnostics of the model. The model will write diagnostics to the filename that is configured in the General Adapter (the model reads it from the *outputDiagnosticFile* field in the *run_info* file). The logging will be to the level that is configured in Delft-FEWS (typically *debug/info/warn/error*).

idMapping

The location/parameters used in Delft-FEWS can be transformed to model variableId locations/parameters by ID-mapping. The configuration files for ID-mapping can be of a general form, as long as the reservoir model have been set up with identical variables for the inputs/outputs. The model will look for the required variables (as configured in the IRM model file) in the *parameter* field of the PI timeseries.

- Example idExport file
- Example idImport file

IdMapping is dependent on how the variables have been defined in the model. The Reservoir Model code will try to parse the configured model variables (like IIn, QOut, etc) from the *parameterId* of the PI timeseries, the locationId is not used. This means that the parameterId's of all the input timeseries need to be unique (and identical to the model variables). When writing the output timeseries, the locationId used in the import PI xml files will be used as the locationId in the output PI xml files.

exportStateActivity

The Reservoir model can work with a stateConfigFile, exported from Delft-FEWS. This file should follow the conventions and list the read/write locations. When defined, the model will write an output state timeseries file for the complete run period, for all model export variables.

exportTimeSeriesActivity

The reservoir model requires at a minimum the following timeseries:

- level or storage state value (at model start time)
- inflow timeseries (complete run period)
- release timeseries (optional)

exportRunFileActivity

A run_info file is required input for the Reservoir model, so an exportRunFileActivity needs to be configured in the General Adapter. The Reservoir Model expects a *model* property in the run_info file, that specifies the name of the actual Reservoir model to be run.

```
        <exportFile>run_info.xml</exportFile>
          <properties>
              <string key="model" value="$RESERVOIR$_IntegratedReservoirModel.xml"/>
          </properties>
      </exportRunFileActivity>
```

A typical run_info.xml file will contain the following information:

Run	
xmlns:*xsi*	http://www.w3.org/2001/XMLSchema-instance
xmlns	http://www.wldelft.nl/fews/PI
xsi:schemaLocation	http://www.wldelft.nl/fews/PI http://fews.wldelft.nl/schemas/version1.0/pi-schemas/pi_run.xsd
version	1.8
logLevel	debug
timeZone	0.0
startDateTime date=2021-11-22 time=03:00:00	
endDateTime date=2021-12-07 time=03:00:00	
time0 date=2021-11-23 time=03:00:00	
lastObservationDateTime date=2021-12-07 time=03:00:00	
workDir	D:\FEWS_Systems\FEWS_HyFS\HyFS_SA_svn\Modules\reservoir\Keep
inputStateDescriptionFile	D:\FEWS_Systems\FEWS_HyFS\HyFS_SA_svn\Modules\reservoir\Keep\statePI.xml
inputTimeSeriesFile (2)	Text
1	D:\FEWS_Systems\FEWS_HyFS\HyFS_SA_svn\Modules\reservoir\Keep\import.xml
2	D:\FEWS_Systems\FEWS_HyFS\HyFS_SA_svn\Modules\reservoir\Keep\importState.xml
outputDiagnosticFile	D:\FEWS_Systems\FEWS_HyFS\HyFS_SA_svn\Modules\reservoir\Keep\diag.xml
outputTimeSeriesFile	D:\FEWS_Systems\FEWS_HyFS\HyFS_SA_svn\Modules\reservoir\Keep\export.xml
properties	string key=model value=Keep_IntegratedReservoirModel.xml

executeActivity

The executeActivity runs the model. The model runs of the Delft-FEWS JRE, so the ReservoirModelAdapter binaries can be specified within the *binDir* element. The Reservoir Model class itself is called *nl.deltares.fews.reservoirmodel.ReservoirModelAdapter.* It is required to provide the path of the run_info file as an argument to the model.

```
            <executeActivity>
                <description>Run Reservoir module</description>
                <command>
                    <className>nl.deltares.fews.reservoirmodel.ReservoirModelAdapter</className>
                    <binDir>$REGION_HOME$/Modules/delft-adapters/fews-reservoirmodel-adapter-bin</binDir>
                </command>
                <arguments>
                    <argument>%ROOT_DIR%/run_info.xml</argument>
                </arguments>
                <timeOut>100000</timeOut>
                <ignoreDiagnostics>true</ignoreDiagnostics>
            </executeActivity>
```

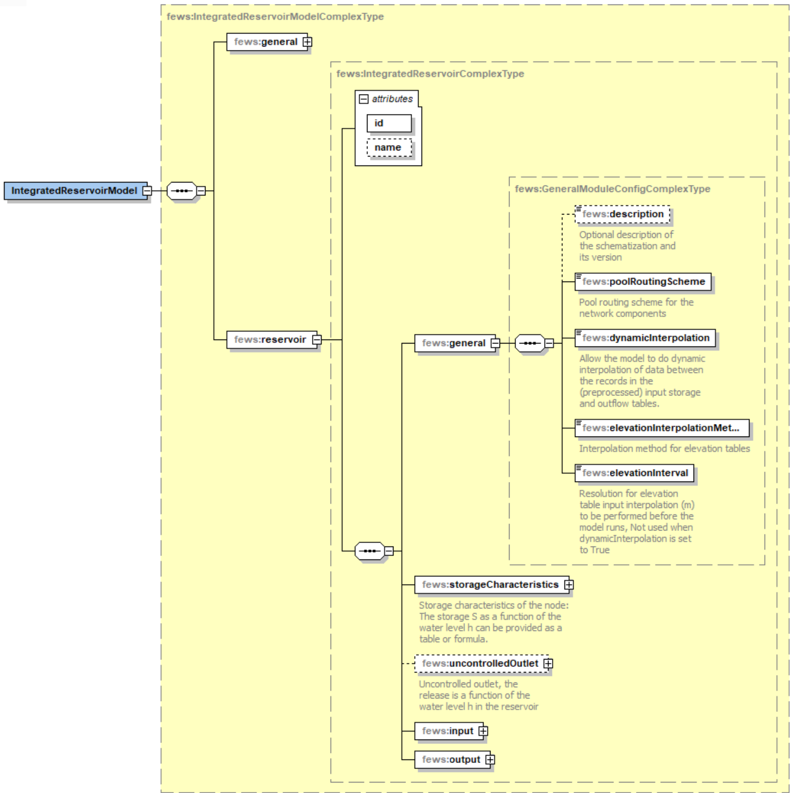
importActivity

The model results (typically consisting of level, storage, inflow and release timeseries) can be imported using the importActivities. The importFile name configured will be written to the run_info file and consequently be created by the Reservoir model. Note that specific idImport configuration is required.

Model

The model definition for the reservoir can be configured in a model file that follows the Integrated Reservoir Model schema. The model options are described below.

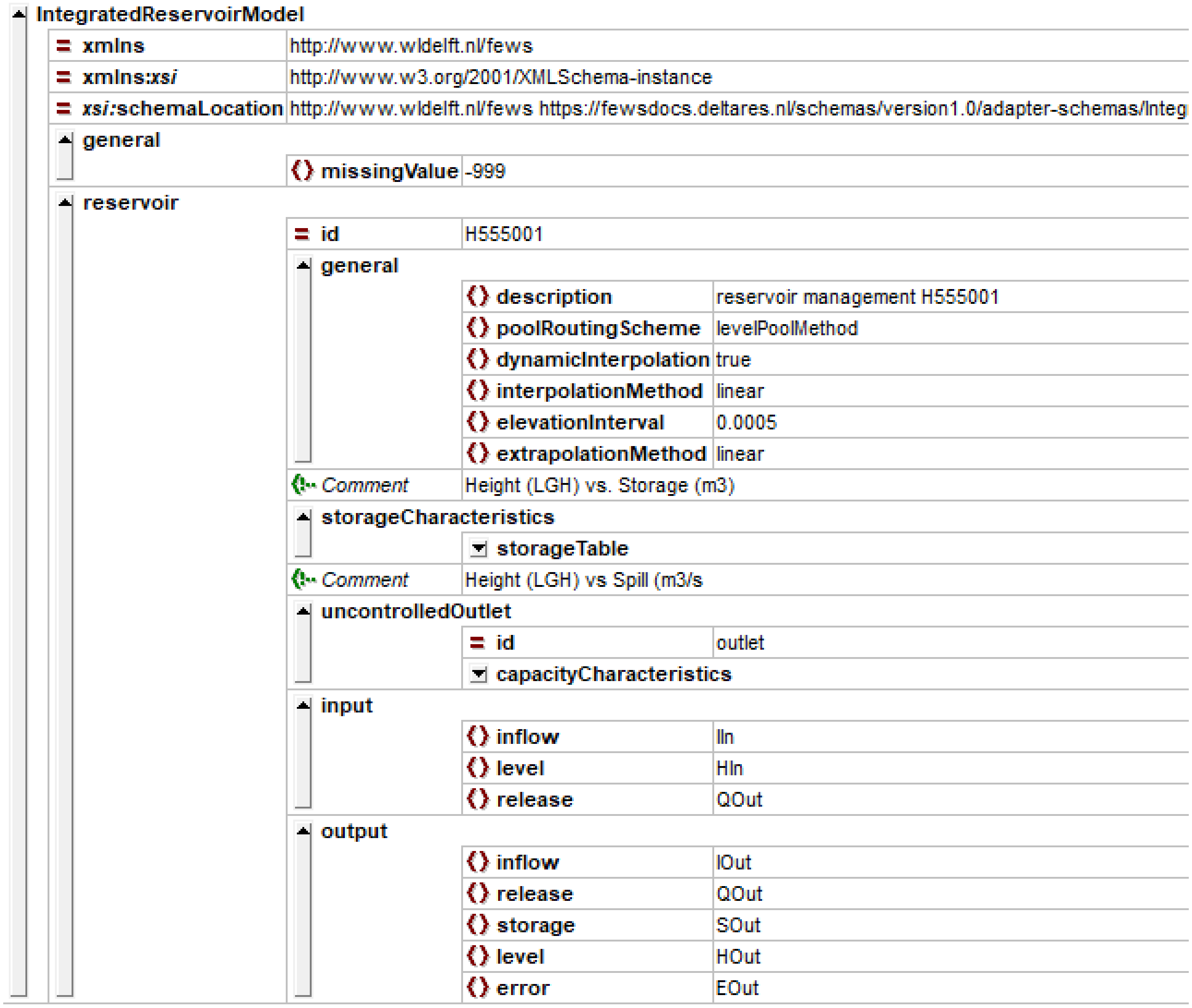
Schema



For reference, an example IntegratedReservoirModel file is attached.

```
<IntegratedReservoirModel xmlns="http://www.wldelft.nl/fews" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://www.wldelft.nl/fews https://fewsdocs.deltares.nl/schemas/version1.0
/adapter-schemas/IntegratedReservoirModel.xsd">
        <general>
                <missingValue>-999</missingValue>
        </general>
        <reservoir id="H555001">
                <general>
                        <description>reservoir management H555001</description>
                        <poolRoutingScheme>levelPoolMethod</poolRoutingScheme>
                        <dynamicInterpolation>true</dynamicInterpolation>
                        <interpolationMethod>linear</interpolationMethod>
                        <elevationInterval>0.0005</elevationInterval>
                        <extrapolationMethod>linear</extrapolationMethod>
                </general>
        <!--Height (LGH) vs. Storage (m3)-->
        <storageCharacteristics>
            <storageTable>
                <elevationStorageRecord elevation="292.9" storage="0"/>
                <elevationStorageRecord elevation="293.0" storage="500"/>
                ...
                <elevationStorageRecord elevation="335.4" storage="720992000"/>
                <elevationStorageRecord elevation="335.6" storage="732795000"/>
            </storageTable>
        </storageCharacteristics>
        <!--Height (LGH) vs Spill (m3/s-->
        <uncontrolledOutlet id="outlet">
            <capacityCharacteristics>
                <outletTable>
                    <elevationOutletRecord elevation="292.9" outlet="0"/>
                    <elevationOutletRecord elevation="293.0" outlet="0"/>
                                        ...
                    <elevationOutletRecord elevation="335.4" outlet="9768"/>
                    <elevationOutletRecord elevation="335.6" outlet="10278"/>
                                </outletTable>
                        </capacityCharacteristics>
                </uncontrolledOutlet>
                <input>
                        <inflow>IIn</inflow>
                        <level>HIn</level>
                        <release>QOut</release>
                </input>
                <output>
                        <inflow>IOut</inflow>
                        <release>QOut</release>
                        <storage>SOut</storage>
                        <level>HOut</level>
                        <error>EOut</error>
                </output>
        </reservoir>
</IntegratedReservoirModel>
```

In XML Grid View this looks the following



general

In the general section of the reservoir model, a *missingValue* element needs to be configured. It is important to match the missingValue as defined in the Delft-FEWS General Adapter configuration for the model run.

reservoir

The reservoir element contains the following sections:

- general
- uncontrolledOutlet
- input
- output

general

The general section of the reservoir element contains the follwing fields:

- poolRoutingScheme
- dynamicInterpolation
- interpolationMethod
- elevationInterval
- extrapolationMethod (from 2023.02)
- storageCharacteristics

poolRoutingScheme

for the *poolRoutingScheme* element, one can choose the following options:

- levelPoolMethod
- backwardsEulerMethod

levelPoolMethod

The Level Pool method is a well known method for reservoir routing. Level Pool routing is a procedure for calculating the outflow hydrograph form a reservoir with a horizontal water surface, given its inflow hydrograph and storage-release characteristics. The level pool routing method is sometimes also referred to as Storage routing, the Storage-Indication method, or the Modified Puls method. For this reservoir model, the method described in Applied Hydrology from V.T.Chow (1988) is used.

The reservoir routing procedure in the Level Pool method is as follows:

We define the value **G** is a function of Storage and Outflow, defined as *G[S] = 2*S/t + O*

where:

- *S* represents the reservoir storage
- *O* represents the reservoir release
- *t* represents the time step

For each row in the the *uncontrolledOutlet capacityCharacteristics outletTable*, we can now precompute a *G[S]* value. This allows the model to look up the release *O* for a given G.

Dt = 3600 sec		G(S) function
Release (O)	**Storage (S)**	**2S/dt + O**
(m3/s)	(m3)	(m3/s)
0	0	0.00000
0	500	0.27778
0	10000	5.55556
0	20000	11.11111
0	30000	16.66667
0	40000	22.22222
0	50000	27.77778
0	59000	32.77778
0	69000	38.33333
0	83000	46.11111
0	104000	57.77778
0	128000	71.11111
0	153000	85.00000

The level pool method makes use of the following relations, that follow from the water balance equations (details in the handbook):

K[t] *= G[t-1] - 2*O[t-1]*

G[t]*= (I[t-1] + I[t])+K[t]*

For the computation, we loop over all time intervals from t=0 to $t=t_{end}$.

1. If t=0
   a. Use the state values as provided in the input files. If either a level, or a storage are provided, look up the equivalent value.
   b. In case both level and storage are provided, use the lookup value to determine any inconsistencies. If found, the level is used as the basis and the storage at t=0 is recalculated
   c. *no computation takes place at t=0*
2. If t=1 then
   a. ***K[1]*** *= 2*S[0]/t - O[0]* (inital storage and release values are known)
   b. **G[1]** is computed with ***G[1]****= (I[0] + I[1])+****K[1]***
   c. Compute outlet *O[1]* by linearly interpolating the precomputed table using *O(S)* and *G(S)*.
   d. In case an outlet *O_input[1]* timeseries is provided as part of the inputs, set *O[1] = O_input[1]*
   e. Compute storage *S[1] = S[0] + t* (I[1] - O[1])*
3. If t>1 then
   a. ***K[t]*** *=* ***G[t-1]*** *- 2*O[t-1]*
   b. **G[t]** is computed with ***G[t]****= (I[t-1] + I[t])+****K[t]***
   c. Compute outlet *O[t]* by linearly interpolating the precomputed table using *O(S)* and *G(S)*.
   d. In case an outlet *O_input[t]* timeseries is provided as part of the inputs, set *O[t] = O_input[t]*
   e. Compute storage *S[t] = S[t-1] + t* (I[t] - O[t])*

A model that is configured to use the level-pool method will write the values for **G** and **K** for each timestep in the the diagnostics when run in debug mode.

backwardEulerMethod

The Backward Euler reservoir routing scheme is an implicit scheme that uses the backward difference approximation for the derivative. The equation for the backward Euler reservoir routing scheme can be written as follows:

S[t+1] = S[t] + t (I[t+1] - O[t+1,S[t]])*

where:

- *S[t+1]* represents the reservoir storage at the next time step (n+1)
- *S[t]* represents the reservoir storage at the current time step (n).
- *t* represents the time step.
- *I[t+1]* is the inflow into the reservoir at the next time step (n+1).
- *O[t+1, S[t]]* is the reservoir release at the next time step (n+1), based on the storage-release relation using *S[t]* for the lookup input.

dynamicInterpolation

When the *dynamicInterpolation* element is set to *true*, the level/storage and level/outlet tables are inteprolated dynamically (every timestep), to the precise value, using the *elevationInterpolationMethod*. In this case, the *elevationInterval* element is ignored.

When the *dynamicInterpolation* element is set to *false*, the level/storage and level/outlet tables are precalculated (only once) using the *elevationInterpolationMethod,* to the specified *elevationInterval.*

interpolationMethod

Linear interpolation is the only available interpolation method

elevationInterval

The *elevationInterval* is the elevation resolution at which the configured level/storage and level/outlet tables need to be recalculated to achieve a higher granularity. Note that the level output at each timestep is processed to that specific elevationInterval. When the model looks up a value from the table, the largest precalculated table elements smaller than the lookup value will be used (i.e. the model always rounds down). The consequence is that reservoir inflows/releases at a timestep that result in *level/storage changes smaller than the interval/resolution* will **not** be taken into account. The model does not perform any shadow accounting to keep track of these volumes. This means that the model will generally underestimate the flow when *dynamicInterpolation* element is set to *false,* and water balance will not be closed for that run type. For larger reservoirs (more volume per unit water disk) the elevationInterval needs to be set to higher resolutions to account for this.

extrapolationMethod

The available extrapolation options are: notAllowed. linear, maxMin

storageCharacteristics

The storageCharacteristic storageTable contains a storage-level lookup table that is *strictly increasing.* Note that this table should have the identical storage inputs as the *capacityCharacteristics outletTable* from the *uncontrolledOutlet*.

uncontrolledOutlet

The *uncontrolledOutlet* element contains *capacityCharacteristics outletTable* which relates storage-release. Note that this table should have the identical storage inputs as the *storageCharacteristic storageTable*.

input / output variables and files

In the *input* section, the model input variables will be configured.

The *<input><inflow>* element is required

The *<input><level>* element is optional, and can be set to the timeseries variable that can overwrite (take precedence) over the level as a result from the release table computation. For a given timestep, if the level input timeseries (e.g. *HIn*) contains a value, this level is applied. The model will determine the resulting *release* from closing the waterbalance (thus not using the release lookup value)

The *<input><release>* element is optional, and can similarly be set to the timeseries variable that can overwrite (take precedence) over the lookup value for the outlet. This means that for a given timestep, if the outlet input timeseries (e.g. *QIn*) contains a value, this release is applied. This input option was added in 2023.02

When both a level and a release input value are available for a timestep, the model will use those values and write them to the output. A waterbalance error term will be calculated and saved for that timestep as well.

The model will look for the required variables in the *parameter* field of the PI timeseries (see idMapping). The Reservoir Model code will try to parse the configured model variables (like *IIn, QOut*, etc) from the *parameterId* of the PI timeseries, the locationId is not used. In the *output* section, the model output variables will be configured. When writing the output timeseries, the locationId used in the import PI xml files will be used as the locationId in the output PI xml files. The output model variableId's will be used as the parameter in the timeseries.

The naming convention of the input and output timeseries filenames are free, the model will determine which files to read for the input based on the *inputTimeSeriesFile* filed in the run_info file. The following two input timeseries files are suggested:

- importState.xml
    - level values, or (HIn)
    - storage values (SIn)

- import.xml
    - inflow (IIn)
    - outlet (optional) (QIn)

The following output timeseries file is suggested:

- export.xml
    - inflow (IOut)
    - release (QOut)
    - storage (SOut)
    - level (HOut)
    - error (EOut)

Note that for the suggested variableId's in the provided example, the postfix **In** and **Out** are used to denote if the series are ***In**puts* for, or ***Out**puts* from the model.

State values

The *startDateTime* and *endDateTime* in the run_info file are used by the model to determine the start (*startDateTime*) and end (*endDateTime*) of the model run. The model will pick the starting (state) value for level/storage (level has precedent in case of an inconsistency), inflow, outlet from the inputTimeSeriesFiles at the specific datetime. It will use the output variables to look for the state timeseries. The model will first check for a state level value. If a starting level value is missing in the input, the model will use the starting storage value. If both starting level/storage cannot be determined from the input files, the model will use the first level element as defined in the storage table as the starting level. A Warning message will be generated to notify the operator of this situation.

Inflow and outlet values at the starting time are not required at the first timestep and these values will not be used for storage calculations (no calculation at the first timestep). When these values are not provided as inputs, a value of 0 is assumed (and written to the output).

When no inflows into the reservoir are defined at all, the model will not calculate, but also it will not error out. It will produce and export.xml with missings (except for the initial values). When some intermediate inflow values are missing, the model will stop calculating at the point. It will not error out (and not throw a warning), it will produces an export.xml with calculated values up until the point an inflow value was missing and it will have missings for all outputs from that moment in time.

In case state functionality is configured in the *run_info* file (inputStateDescriptionFile is defined), the model will also write (all) the outputs to the write location as defined in the stateLocation file (e.g. *exportState.xml*)

stateLoc		
	type	file
	readLocation	D:\FEWS_Systems\FEWS_HyFS\HyFS_SA_svn\Modules\reservoir\Keep_IRM\importState.xml
	writeLocation	D:\FEWS_Systems\FEWS_HyFS\HyFS_SA_svn\Modules\reservoir\Keep_IRM\exportState.xml

Reservoir Model specifics

- The Integrated Reservoir Model can be considered *"timestep ending"*, similar the the timestep definition of Delft-FEWS.
- No calculations/processing is performed at the first timestep (t=0, *startDateTime* as set in the run_info.xml file)

Ensembles

The Reservoir Model can be run in parallel from Delft-FEWS. The *runInLoop* element of the workflow should be set to *false.* The *general* section of the General Adapter configuration should contain the *%TEMP_DIR%* property as the model *rootDir.* And lastly, to enable the parallel running of ensemble members the *runInLoopParallelProcessorCount* entry must be set in the global properties file. Here you either specify the number of cores to use or specify 100 to use all available cores.

The workflow definition for a parallel model run:

```
<activity>
    <runIndependent>true</runIndependent>
    <workflowId>Reservoir_Forecast</workflowId>
    <ensemble>
        <ensembleId>ENSEMBLE</ensembleId>
        <runInLoop>false</runInLoop>
    </ensemble>
</activity>
```

The *general* section of the General Adapter configuration:

```
<general>
  <rootDir>%TEMP_DIR%</rootDir>
  <workDir>%ROOT_DIR%/work</workDir>
  ...
</general>
```

Entry in the global properties:

Config Example	`# to use 4 cores/cpu's:` `runInLoopParallelProcessorCount=4`

See the following page for more details.

In-Memory execution

From Delft-FEWS version 2023.01 onwards, it will be porssible to run the Reservoir Model adapter "in-memory" from Delft-FEWS, using the *inMemoryFileTransfer* element of the *general* section set to *True*. In that case, all exported and imported files are transferred in memory between Delft-FEWS and the executed Reservoir Model.