Writing OpenHydroQual Plugins

A Developer Reference for Defining Component Types

Arash Massoudieh*

June 3, 2026

1 Introduction

The components you place in the OpenHydroQual GUI — catchments, tanks, pipes, groundwater cells, reactions — are not hard-coded into the program. Each is defined declaratively in a JSON *plugin* file. A plugin file is simply a dictionary of *object type* definitions: it tells OpenHydroQual what types exist, what properties each one exposes, how those properties appear and are validated in the GUI, and — crucially — the mathematical expressions that govern the object's behaviour during a simulation.

This means you can add entirely new physics to OpenHydroQual without touching its source code. To create a new kind of block, link, source, or reaction, you write a JSON definition and load it as a plugin. The built-in components ship as plugin files in the `resources/` sub-directory of the installation (`main_components.json`, `groundwater.json`, and so on); reading them is the fastest way to learn the format, and this reference is organized around real examples drawn from them.

This document is intended for *component developers*. It assumes you are comfortable with the modelling concepts from Tutorial 1 (blocks, links, sources, and the storage mass balance) and with basic JSON syntax.

2 File and Object Structure

A plugin file is a single JSON object whose keys are *object type names* and whose values are the definitions of those types. The type name is the internal identifier (it is what appears in a model's `.ohq` file as `type=...`); the human-readable label shown in the GUI comes from the `description` field.

```
{
  "Groundwater cell": {
    "type": "block",
    "typecategory": "Blocks",
    "description": "Groundwater cell",
    "icon": { "filename": "Groundwater.png" },

    "<property name>": { ... },
    "<property name>": { ... }
  },
```

*EnviroInformatics

```
    "groundwater_link": {
      "type": "link",
      ...
    }
  }
```

Every object definition carries a few top-level fields, followed by one entry per property.

The `solutionorder` directive. A plugin file may also include a top-level `solutionorder` key — a file-level directive rather than an object definition. It names the state variable the solver integrates for, as a list:

```
{
  "solutionorder": [ "Storage" ],
  "Precipitation_w_forcast": { ... },
  ...
}
```

In practice this is almost always `Storage`, the storage state variable of the mass balance. It tells the solver which balance quantity to solve the system for.

Top-level object fields.

- `type` — what kind of object this is. The values in use are `block`, `link`, `source`, `constituent`, `Reaction`, `ReactionParameter`, `parameter`, `objective_function`, and `observation`. The first four (and reactions) are the everyday modelling objects; the last three support calibration and optimization.
- `typecategory` — the branch of the Object Browser and the toolbar group the object appears under (e.g. `Blocks`, `Connectors`, `Sources`, `Constituents`, `Reactions`). Note the GUI label differs from `type`: links use `"typecategory": "Connectors"`.
- `description` — the label shown for the type in the GUI.
- `icon` — an object `{ "filename": "...png" }` naming the toolbar/canvas icon.
- `normalizing_quantity` (reactions only) — the quantity a reaction rate is normalized by, e.g. `area`.

3 Anatomy of a Property

Each property is itself a JSON object. Its `type` determines how it behaves; the remaining fields control its appearance, validation, units, and output. The full field vocabulary observed across the built-in plugins is collected in table 1; the property types in table 2.

A representative value property, from the groundwater cell:

```
"hydraulic_conductivity": {
  "type": "value",
  "description": "Hydraulic Conductivity",
  "ask_user": "true",
  "delegate": "UnitBox",
  "unit": "m/day;ft/day;cm/s;in/s;m/s",
  "estimate": "true",
  "default": "1",
  "criteria": "hydraulic_conductivity>0",
  "warningmessage": "Hydraulic conductivity must be greater than zero",
  "helptext": "Rate at which water moves through the subsurface."
}
```

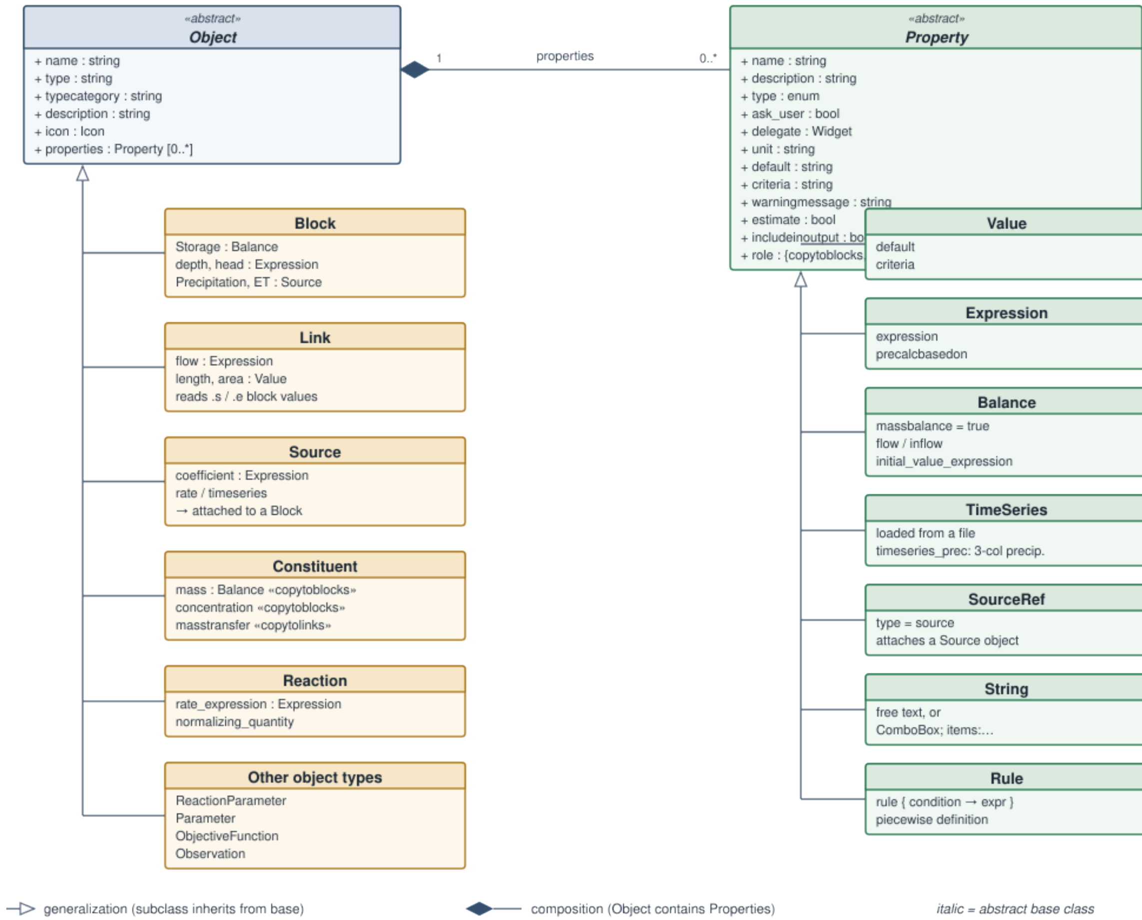

Figure 1: The OpenHydroQual object model. Every component is an *Object*; the concrete types (Block, Link, Source, Constituent, Reaction, and the analysis types) inherit from it. Each Object is composed of *Property* instances, which themselves come in distinct types (Value, Expression, Balance, TimeSeries, and so on).

Table 1: Property-level fields.

Field	Meaning
`type`	The property kind (see table 2). Required.
`description`	Label shown in the property panel. A description containing a semicolon, e.g. `"Initial Storage; Storage"`, gives two labels: the initial-condition label and the running-variable label.
`helptext`	Longer tooltip / help-dialog text.
`ask_user`	`"true"` if the user sets this property; `"false"` for internally computed quantities that should not be editable.
`default`	Default value used when the user does not supply one.
`delegate`	Which GUI editor widget to use (see table 3).
`unit`	Semicolon-separated list of selectable units; the first is the default. The internal unit is the first listed.
`criteria`	A boolean expression that must hold for a valid model (e.g. `"area>0"`).
`warningmessage`	Message shown when `criteria` fails.
`estimate`	`"true"` marks the property as eligible to be a calibration parameter.
`includeinoutput`	`"true"` writes this quantity to the results file and exposes it in the Results menu.
`role`	For shared/templated properties: where the property is propagated — `copytoblocks`, `copytolinks`, `copytoreactions` (see section 7).
`expression`	The formula for an `expression`-type property (see section 5).
`applylimit`	`"true"` lets the solver limit/clip this flux to keep the balance physical (e.g. prevent withdrawing more than is stored).

Table 2: Property `type` values.

`type`	Meaning
`value`	A user-supplied constant (a parameter of the object).
`expression`	A quantity computed from other properties via a formula in `expression`. Recomputed as the simulation proceeds.
`balance`	A state variable integrated through time by the mass balance (see section 4).
`timeseries`	A time series supplied by the user (interpolated instantaneous values).
`timeseries_prec`	A precipitation-style time series. Use this (rather than `timeseries`) when the file is precipitation data in the three-column start-time/end-time/depth format documented in Tutorial 1, as opposed to a two-column instantaneous-value series.
`source`	A slot to which a named *source* object is attached (the `copytoblocks` forcing slots).
`string`	A text value (e.g. `name`) or a selection from a combo box.
`rule`	A piecewise definition: a map of condition → expression (see section 6).

Table 3: `delegate` values (the editor widget).

`delegate`	**Widget**
`ValueBox`	Plain numeric entry.
`UnitBox`	Numeric entry with a unit selector (use whenever `unit` is set).
`String`	Free text entry.
`Browser`	File browser, used to load a time series file.
`Browser;time series`	File browser specialized for time-series input.
`ComboBox;Sources`	Drop-down of available source objects (for `source` slots).
`ComboBox;BlockLinks`	Drop-down of blocks and links in the model (used by objective functions / observations).
`ComboBox;items:a,b,c`	Drop-down with a fixed list of choices, e.g. `ComboBox;items:normal,log-normal`.
`expressionEditor`	A multi-line editor for entering a formula.

4 The Mass Balance: `balance` Properties

A `balance` property is a state variable the solver integrates through time — the storage at the heart of every block. Its defining fields wire it into the mass balance of Tutorial 1:

```
"Storage": {
  "type": "balance",
  "massbalance": "true",
  "flow": "flow",
  "inflow": "inflow",
  "initial_value_expression": "area*depth*moisture_content",
  "includeinoutput": "true",
  "description": "Initial Storage; Storage",
  "ask_user": "false",
  "estimate": "true",
  "unit": "m^3;ft^3"
}
```

- `massbalance`: `"true"` declares this as an integrated state variable.
- `flow`: the name of the property (on connected links) that carries flux *out* through links — the $\sum_l Q_l$ term. Links define a `flow` property; the block's balance references it by this name.
- `inflow`: a comma-separated list of property names that add to the balance (direct inflows and attached sources — the I_i and q^{src} terms). The groundwater cell lists `inflow`; a catchment lists several, e.g. `"Precipitation,Evapotranspiration,loss"`.
- `initial_value_expression`: the initial storage, computed from other properties (here the cell geometry and moisture content). A block may instead let the user enter the initial value directly.
- `rigid` (seen elsewhere): `"true"` marks a quasi-steady balance with no real storage dynamics (used for rigid junctions).

Derived quantities are then computed from the state variable through `expression` properties — for the groundwater cell, `moisture_content = Storage/(depth*area)` and a `head` expression built from it.

5 Expressions and the `.s`/`.e` Convention

An `expression` property holds a formula evaluated from the object's other properties. Within a single object, properties are referenced by name:

```
"head": {
  "type": "expression",
  "expression": "depth+bottom_elevation+(moisture_content-porosity)/specific_storage",
  "includeinoutput": "true",
  "unit": "m;ft"
}
```

Cross-object references on links. A *link* connects a *start* block and an *end* block. Inside a link's expressions, a property of the start block is written with the suffix `.s` and a property of the end block with `.e`. This is how a link reads the state of the blocks it joins. The groundwater link's flow is a clear example:

```
"flow": {
  "type": "expression",
  "expression": "0.5*area*(hydraulic_conductivity.s+hydraulic_conductivity.e)*(head.s-
     head.e)/length",
  "applylimit": "true",
  "unit": "m^3/day;ft^3/day;m^3/s"
}
```

Here `head.s` and `head.e` pull the hydraulic head from the two connected cells, and `length`/`area` are the link's own properties. The computed `flow` is what the connected blocks' balances reference through their `flow` field — positive flow leaves the start block and enters the end block, conserving mass. A link can even take a property entirely from one end, as in `lake2groundwater_link`, whose `area` is `"area.s"` and whose `hydraulic_conductivity` is `"hydraulic_conductivity.e"`.

Built-in functions. Expressions support the usual arithmetic and a library of helper functions, prefixed with an underscore. Those seen in the built-in plugins include `_pos` (positive part, $\max(x, 0)$), `_hsd` (Heaviside step), `_sgn` (sign), `_abs`, `_sqt` (square root), `_min`, `_max`, `_log`, `_exp`, and `_mbs`. These are used to keep expressions physical — for example wrapping a depth in `_pos(...)` so a negative never propagates.

Two further functions smooth a time series, which is useful for forecasting or delaying a forcing signal:

- `_ekr(series; lambda)` — exponential-kernel smoothing of `series` with rate `lambda` (typically 1/mean delay).
- `_gkr(series; delay; spread)` — Gaussian-kernel smoothing of `series` with the given mean delay and spread.

The precipitation-forecast plugin uses these to turn a raw rain series into a smoothed forecast that then drives a flow rule (section 6).

Advective transport: _ups. A function of particular importance for transport components is `_ups(flow; quantity)`, the upstream (upwind) operator. Evaluated on a link, it returns the value of `quantity` taken from whichever connected block is *upstream* according to the sign of `flow` — that is, it selects `quantity.s` or `quantity.e` depending on the flow direction. This is the standard

upwinding scheme for advective transport. In `main_components.json`, a constituent's advective mass transfer is exactly

```
"advective_masstransfer": { "type": "expression",
  "expression": "_ups(flow;concentration)" },
"diffusive_masstransfer": { "type": "expression",
  "expression": "(diffusion_coefficient*area+dispersivity*flow)/length*(concentration.s
      -concentration.e)" },
"masstransfer": { "type": "expression",
  "expression": "advective_masstransfer+diffusive_masstransfer" }
```

so that the link carries the upstream concentration with the flow (advection) plus a Fickian gradient term (diffusion/dispersion), the two summing to the total mass transfer. Any time a quantity must be carried in the direction of flow rather than averaged across a link, `_ups` is the operator to use.

6 Piecewise Behaviour: the `rule` Type

The `rule` type defines a property piecewise, as a map from a condition to the expression that applies when the condition holds. The reservoir-outflow rule in `main_components.json` is the canonical example:

```
"flow": {
  "type": "rule",
  "description": "Flow",
  "rule": {
    "Storage.s<S_min": "Q_min",
    "S_min<Storage.s<S_max": "Q_min+((Storage.s-S_min)/(S_max-S_min))*(Q_max-Q_min)",
    "Storage.s>S_max": "Q_max",
    "unit": "m^3/day;ft^3/s"
  },
  "applylimit": "true",
  "ask_user": "false"
}
```

Each key of the `rule` object is a boolean condition and each value is the expression used when it is satisfied; a `unit` entry sets the units of the result. This lets you express rating curves and operating rules without nesting conditionals into a single expression.

The condition need not be a raw state variable — it can be any computed quantity. In the precipitation-forecast plugin, a flow-control device opens between a minimum and maximum flow according to a *kernel-smoothed precipitation forecast*:

```
"kernel_smoothed_forcast": {
  "type": "expression",
  "expression": "_ekr(timeseries;lambda)"
},
"flow": {
  "type": "rule",
  "rule": {
    "kernel_smoothed_forcast<S_min": "flow_coefficient*Q_min",
    "S_min<kernel_smoothed_forcast<S_max":
      "flow_coefficient*(Q_min+((kernel_smoothed_forcast-S_min)/(S_max-S_min))*(Q_max-
          Q_min))",
    "kernel_smoothed_forcast>S_max": "flow_coefficient*Q_max"
```

```
  },
  "applylimit": "true",
  "unit": "m^3/day;ft^3/day;m^3/s"
}
```

Here the rule is driven by `kernel_smoothed_forcast`, an `expression` that smooths the incoming precipitation series with the exponential kernel `_ekr`. This pattern — compute a derived signal with an expression, then branch on it with a rule — composes the building blocks of sections 5 and 6 into a complete operating policy.

7 Shared Properties and `role`

Some properties are not defined on a block directly but are *propagated* from another object — this is how constituents and reactions inject properties into every block, link, or reaction in the model. The `role` field controls the propagation:

- `copytoblocks` — the property is added to every block.
- `copytolinks` — the property is added to every link.
- `copytoreactions` — the property is added to every reaction.

How a constituent uses the roles. The `Constituent` object in `main_components.json` is the clearest example, because it divides its properties between blocks and links — and that division *is* the discretized transport equation. Its block-bound (`copytoblocks`) properties are the storage side of transport:

- `mass` — a `balance` property, so *every block gains its own integrated constituent mass*, exactly as it has its own water `Storage`;
- `concentration` = `mass/(Storage+0.00001)` — the concentration in each block;
- the loading terms `inflow_loading`, `constant_inflow_concentration`, `external_source`, and the external mass-flow series.

Its link-bound (`copytolinks`) properties are the transport side — the fluxes that move constituent between blocks:

```
"advective_masstransfer": { "role": "copytolinks",
  "expression": "_ups(flow;concentration)" },
"diffusive_masstransfer": { "role": "copytolinks",
  "expression": "(diffusion_coefficient*area+dispersivity*flow)/length*(concentration.s
      -concentration.e)" },
"masstransfer": { "role": "copytolinks",
  "expression": "advective_masstransfer+diffusive_masstransfer" }
```

Finally, `stoichiometric_constant` carries `role: copytoreactions` so each reaction can reference it.

The effect is that defining a *single* constituent endows every block in the network with a constituent mass-balance state variable and every link with the advection-plus-diffusion flux that transports it. This mirrors the water mass balance of Tutorial 1 exactly: just as a block's `Storage` is driven by the `flow` on its links, each block's constituent `mass` is driven by the `masstransfer` on its links — only now the quantity is a chemical species rather than water. You write the physics once, on the constituent, and the roles distribute it across the whole model graph without your editing any individual block or link.

8 Special Object Types for Calibration

Three object types are not physical components but support parameter estimation; they are worth knowing when authoring plugins that should be calibratable.

- `parameter` — a calibration parameter with `low`/`high` bounds, a `value`, and a `prior_distribution` (`normal` or `log-normal`). Any object property marked `"estimate": "true"` can be tied to one.
- `objective_function` — references an `object` (via `ComboBox;BlockLinks`), a `method` (`integrate`, `value`, `maximum`, `variance`, `exceedance`), an `expression`, and a `weight`.
- `observation` — pairs a modeled `expression` on a chosen `object` with an `observed_data` time series, for comparison against measurements.

9 Advanced: Pre-computed Inversions (`precalcbasedon`)

A handful of expression properties carry a `precalcbasedon` field. It is a performance optimization: rather than evaluating the expression from scratch at every solver iteration, the program first builds a piecewise lookup of the expression's value as a function of an independent variable, then reads the pre-computed values during the run. The single built-in example is the sewer channel segment's depth:

```
"depth": {
  "type": "expression",
  "expression": "normalized_depth*diameter",
  "precalcbasedon": "Storage:log:0.000001:1000000",
  "description": "Initial Water Depth; Water Depth",
  "delegate": "UnitBox",
  "unit": "m;ft",
  "criteria": "depth>-0.0000001"
}
```

The field value has the form `variable:scale:min:max`. Here the depth is pre-tabulated as a function of `Storage` on a logarithmic grid spanning 10^{-6} to 10^{6}. This is most useful when the expression encodes an implicit or expensive relationship — such as inverting the depth–storage geometry of a circular channel — that would otherwise be costly to evaluate repeatedly. Use it when an expression is a smooth function of a single state variable and is evaluated very often.

10 Loading a Plugin

A plugin is registered with the model through the template-loading commands that appear at the top of every `.ohq` file:

```
loadtemplate; filename = .../resources/main_components.json
addtemplate; filename = .../resources/groundwater.json
```

`loadtemplate` begins a fresh template set; `addtemplate` merges an additional plugin into it. In the GUI this corresponds to enabling a plugin under **File** → **Preferences** → **Add Plugin**: the objects defined in the file then appear in their `typecategory` groups on the toolbar, ready to place.

11 A Minimal Worked Example

To define a new component, the recommended workflow is: copy the closest existing object from a built-in plugin, rename it, and adjust its properties and expressions. As a minimal illustration, a simple linear-reservoir block — storage that drains at a rate proportional to itself — can be written as:

```
{
  "Linear reservoir": {
    "type": "block",
    "typecategory": "Blocks",
    "description": "Linear reservoir",
    "icon": { "filename": "linear_reservoir.png" },

    "Storage": {
      "type": "balance",
      "massbalance": "true",
      "flow": "flow",
      "inflow": "inflow",
      "initial_value_expression": "S0",
      "includeinoutput": "true",
      "description": "Initial Storage; Storage",
      "unit": "m^3;ft^3"
    },
    "S0": {
      "type": "value", "description": "Initial storage",
      "ask_user": "true", "delegate": "UnitBox", "unit": "m^3;ft^3",
      "default": "0"
    },
    "inflow": {
      "type": "timeseries", "description": "Inflow time series",
      "ask_user": "true", "delegate": "Browser",
      "includeinoutput": "true", "applylimit": "true",
      "unit": "m^3/day;m^3/s"
    },
    "k": {
      "type": "value", "description": "Recession coefficient",
      "ask_user": "true", "delegate": "UnitBox", "unit": "1/day",
      "estimate": "true", "default": "0.1",
      "criteria": "k>0", "warningmessage": "k must be positive"
    },
    "head": {
      "type": "expression", "expression": "Storage*k",
      "description": "Outflow rate", "includeinoutput": "true"
    }
  }
}
```

Saved as a `.json` file and loaded as a plugin, this defines a new block that any compatible link can drain. From here, adding constituents, reactions, or head-driven links follows the same patterns shown above.