Seed Version 15 Documentation

Table Of Content

Seed Version 15 Introduction

The Seed Platform Version 15 (Seed V15) is a communication platform meant to capture all necessary building data for environmental simulations. The Seed is usually controlled by a client-side tool such as StepWin and Properate. Being JSON-based means that Seed can easily be translated to and from an object in common programming languages, making it platform-agnostic.

Seed can be used to store user inputs or be sent to the server for performing computation. The result of this computation are "Saplings" which add some results to the Seed. The result fields can be shown to the users or can simply be stripped away to return to a valid Seed. Another acceptable server request is export of a Seed to a format such as H2K.

Design Philosophy

The Seed Platform is designed with the following principles:

- **Complete:** The Seed is the unified method of communication with Lambda Science. The Seed specification must contain all input information necessary to communicate with Lambda servers.
- **Concise:** Each piece of information must be expressed only in one part of the Seed. Duplication of data must be avoided.
- **Sessional:** Seed is stored without a state; however, usage of the Seed may happen in a "session", during which the Seed has a state. Sessions start through a server request for a session token.
- **Compressible:** Multiple similar seeds should be compressible into one.
- **Reconstructible:** For efficiency in communication and storage, the Seed's non-essential information should be erasable. As long as the essential parameters are available, the server should be able to reconstruct the Seed from known data or by looking up standard dictionaries.

Main Sections

The general structure of the seed is as follows:

```
{
    "meta": {}, // MANDATORY: metadata about the project
    "house": { // MANDATORY: components of the house.
        "walls": [],
        "cathedrals": [],
        "attics": [],
        "decks": [],
        "exposedFloors": [],
        "basements": [],
        "crawlSpaces": [],
        "SOGs": [], //Slab On Grade
        "space": {}, //Information about the archetype, loads, number of units,
and air-tightness
        "equipment": {
            "ventilation": {}, //Fans and ventilation rate
            "mech": {}, //Main mechanical system
            "aux": {},
            "hrv": {}, //Heat Recovery Device or the main vent
            "dhw": {},
            "suppl": {}, //Supplementary heating systems such as fireplaces
            "PV": {}, //Solar panel info
        }
    },
    "helpers": {}, // **Sapling Only.** InsightEngine adds frequently used numbers
here to help the takeoff and costing calculations.
    "output": {} // **Sapling Only.** Results of the server simulation.
}
```

So a seed needs at least a `meta` property and a `house` property.

The `meta` property includes the following at minimum:

```
{
    "meta": {
        "project": {}, // MANDATORY: Address, lat-lng of the project. Used to find
climate info.
        "ERS": true, // Making the file to have ERS values
        "unit": "metric", // "metric" (lengths in meters) and "imperial" (length
in feet) are accepted. Metric is strongly recommended.

        "compressed": [], // Compress Batching Data, see batching (optional)
        "swaps": {}, // Swap Batching data, see batching (optional)

        "Objective": "MultiSeed", // Used for running MURB projects all at once,
see MURB Analysis (optional)

        "template": "VanHouse", // The basis for building seed (optional)
        "run": "Van1", // Giving a name to a run (optional)
```

```
        "version": "15.4.1", // Generator's Seed Version
    }
}
```

Items in `house` must be filled based on home's configuration. The items such as `walls` and `decks` that may have multiple instances in the home always contain an array which is then filled by objects for every instance. For example, in the `walls` may have:

```
{
    // ...
    "house": {
        "walls": [
            {
                "area": 15,
                "perimeter": 6,
                "adjacent": "unheated" //a garage-facing wall
                //...
            },
            {
                "area": 100,
                "perimeter": 50,
                //...
            }
        ]
        //...
    }
}
```

Entities

Most objects in `house` have a property called `entity`. Each `entity` represents an assembly or a mechanical system. Separating those specs from areas and other envelope info makes it easier to change them. For example, in `walls`:

```
{
    // ...
    "house": {
        "walls": [
            {
                "area": 100,
                "perimeter": 50,
                "entity": {
                    "r": 3.3, //RSI value of the assembly.
                    "components": [
                        //the layers of the wall are shown here.
                    ],
                    //...
                }
```

```
                //...
            }
        ]
        //...
    }
}
```

Here the user is exploring a wall with the RSI value of 3.3. If they decide to look at another assembly, the `entity`'s value will get replaced with the properties of the other wall, while `area` and `perimeter` won't change. The `entity` is separating wall's dimensions from its construction.

InsightEngine has a material database with information about tens of thousands of building assemblies. Any of those can be fetched and put in a compatible `entity` for simulation. If the InsightEngine has the `entity`, the `entity`'s property can be filled with just the name to conserve storage and bandwidth:

```
{
    // ...
    "house": {
        "walls": [
            {
                "area": 100,
                "perimeter": 50,
                "entity": "wall_12312",
                //...
            }
        ]
        //...
    }
}
```

Each `entity` has a `substance` that tells where it can be used, and an `id` to differentiate it with other entities.

Channels

Once a Seed is about to be used, the client-side opens a communication *channel* with the server. The best time to open the channel is when a Seed is ready to be used. Opening a channel is simple and there is no need to close it. InsightEngine automatically cleans up expired channels. A channel is used as follows:

1. The client sends the prepared seed to the InsightEngine to get a cryptographic token. Once a token is received for a Seed, it cannot change. A new token must be requested after any modifications.
2. The client puts the token in the Seed under `meta.token` and sends this tokenized Seed to the desired destination (simulation, export, etc.).
3. If the process is correct, InsightEngine responds with a `success` status, and responds. Otherwise the server responds with an `error` status.
4. For processes that risk facing a timeout, the InsightEngine responds with that available data but it specifies `done: false`, indicating that more data will be coming.
5. If `done: false` is present, the client must wait some seconds and then send the **exact same** Seed it send the server one more time to request an update until it gets `done: true`.

So channels employ a *long polling* behavior for when the process takes time, such as during a large simulation. The cryptographic token makes the usage more secure and ensures the immutability of the requested Seed. One advantage of this is showing any potential client-side software errors that may arise from unwanted Seed modifications.

Sapling

As shown in the examples above, when asked for a simulation, the server responds with a "Sapling". The structure is the same as the Seed with `helpers`, `output` and some additions. In detail InsightEngine will:

1. Check for errors in the Seed.
2. Convert every unit to metric. So **all Sapling responses are metric**.
3. Unravel the batch if the Seed has batching (see batching).
4. Recreate the `entities` that are in the shorthand format by looking up internal dictionaries.
5. Write in the climate information.
6. Create the Helpers.
7. Calculate item-specific properties such as adjusting SHGC for each window size, and derive mechanical efficiency from AFUE.
8. Run the simulations through R-BEST.
9. "Cherrypick" the best performing simulations
10. Asynchronously respond to client request with the ran simulations.

See the Sapling example for more details.

The server response contains the following:

```
{
    "done": true, //See Channels for more info. Indicates if the client should
expect adjustments to the sent results.
    "cherryPicked": [], //Main payload. The cherrypicked Saplings are here.
    "overview": [], //Gives a overview metrics for all simulations processed. Good
for visualization.
    "timestamp": 1623710141301, //Unix time the simulation started.
    "processed": 18987, //Number of simulations performed.
    "ranges": {} //Gives the cost and energy efficiency ranges of all simulations
processed. Good for understanding where the cherrypicked sit.
}
```

Helpers

Responses from the server contain two types of helpers:

- Building Helpers: can be found under Seed's `helpers`. They contain information about the whole building such as the total area of walls.
- Assembly Helpers: is distributed inside different parts of `house`, giving info about the assembly that contains it.

Refer to the Sapling examples for seeing Building Helpers.

Assembly Helpers are much smaller:

```
{
    //...
    "helpers": {
        "envelope_area": 140.4, //Area of the wall surface (m2)
        "enclosing_area": 42.001094615853816, //Floor area that the wall encloses
(m2)
        "roof_surface_area": 0, //NA
        "wall_area": 140.4, //Wall Surface Area with Windows (m2)
        "roof_projected_area": 0, //NA
        "wall_soilFacing_area": 0, //NA
        "fenestration_area": 6.2554713600000005, //Window/Doors in the wall (m2)
        "roof_opaque_surface_area": 0, //NA
        "wall_opaque_area": 134.14452864, //Wall surface area without
windows/doors (m2)
        "slab_area": 0, //NA
        "slab_perimeter": 0, //NA
        "envelope_opaque_area": 134.14452864 //Wall surface area without
windows/doors (m2)
    }
}
```

The above Assembly Helper is for a wall. Some irrelevant items (e.g. roof_surface_area) are set to 0. The helpers makes it easy to quickly find important building data without the need to re-calculate dimensions.

Energy Simulation Outputs

The `output` contains a number of fields relating to the energy performance, such as `metrics`, `design`, `detailed`, `costs`, `rebates`, and region-specific outputs.

```
{
    "output": {
        //...
        "bc_metrics": {},
        "bc_step": {},
        "detailed": {},
        "design": {},
        "rebates": {}
    }
}
```

The `metrics` in BC are related to the BC Step Code, but generally the EnerGuide Rating System score is the most important property of it.

```
{
    "output": {
        //...
```

```
        "bc_metrics": {
            "TEDI": 120.29574224401827, //Thermal Energy Demand Intensity
kWh/sqm.yr
            "eTEDI": 117.24846484602044, //TEDI for the Reference House TEDI
analysis.
            "MEUI": 27.545994241923566, //Mechanical Energy Usage Intensity
kWh/sqm.yr
            "ERS": 18.42481723, //EnerGuide Rating System Score, GJ
            "ERS_ref": 64.51590838599999, //All items with "Ref" are Reference
House Metrics
            "TEDI_ref": 67.20932256443056,
            "eTEDI_ref": 105.69743547069733,
            "auxiliary_heating_required_kWh": 22350.948908938597,
            "ERS_kWh": 5118.045730149399, //EnerGuide Rating System Score, kWh
            "total_heated_area": 185.8,
            "baseload": 25.623000000, //Load from equipment, based on ERS Standard
Operating Conditions GJ
            "AEC": 44.047817230, //Annual Energy Consumption GJ
        }
    }
}
```

Another region-specific output in BC is how the design performs based on the BC Energy Step Code.

```
{
    "output": {
        //...
        "bc_step": {
            "pre": 1, //Step Code Level without air-tightness adjustment
            "post": 1, //Step Code Level with air-tightness adjustment
            "thermal": 1, //Step of the thermal performance
            "mechanical": 5, //Step of the mechanical performance
            "ach": 1, //Step of the air-tightness
            "targets": { //Threshold of each metric for reaching a Step Code
level.
                //Used for verification and reference.
                "TEDI": {
                    "2": 42,
                    "3": 37,
                    "4": 27,
                    "5": 19
                },
                "MEUI": {
                    "2": 73,
                    "3": 60,
                    "4": 48,
                    "5": 33
                },
                "TEDI_percent": {
                    "2": 5,
                    "3": 10,
                    "4": 20,
```

```
                    "5": 50
                },
                "ERS_percent": {
                    "1": 0,
                    "2": 10,
                    "3": 20,
                    "4": 40
                }
            }
        }
    }
}
```

The `detailed` metrics go further into the where and at what times the consumption is being generated.

```
{
    "output": {
        //...
        "detailed": {
            //... Monthly performance of each metric.
            "totals": {
                "electric_yearly_GJ": 44.04778234,
                "gas_yearly_GJ": 0,
                "equivalent_leakage_area": 0.260382007,
                "ac_yearly_GJ": 0.571060547,
                "heat_loss_year_MJ": 105931.46696,
                "utilized_internal_year_MJ": 23304.385010000005,
                "utilized_solar_year_MJ": 2164.30839
            }
        }
    }
}
```

`design` outputs are for mechanical system sizing.

```
{
    "output": {
        //...
        "design": {
            "gross_space_heating_load_GJ_yr": 105.931,
            "usable_internal_gains_GJ_yr": 23.304,
            "usable_solar_gains_GJ_yr": 2.164,
            "aux_energy_required_GJ_yr": 80.463,
            "space_heating_consumption_kWh_yr": 3840.2,
            "DHW_consumption_kWh_yr": 977,
            "GHG_t_yr": 10.743,
            "design_heat_loss_t": -6.3,
            "design_heat_loss_w": 11533,
            "design_cooling_load_t": 29.5,
```

```
            "design_cooling_load_w": 1981,
            "design_cooling_capacity_w": 2091,
            "energy_consumption_monthly_MJ": {
                //... Monthly metrics are here
            }
        }
    }
}
```

The costing metrics are discussed in the next section.

Costing: Material, Labour, Carbon

In each Sapling, InsightEngine will include `costs` in the `output`. The costs will be all 0 in a pre-upgrade evaluation, if the client hasn't made a mistake, but they will still be reported. InsightEngine calculates the cost of 3 things: `materials`, `labour`, and `carbon`.

`carbon` refers to the "tonnes/year" of Embodied Carbon of the actions taken in the building. Including it in costing makes calculations easier to follow and avoids duplications. The `materials` and `labour` are in Canadian Dollars and `total` is the sum of the two.

The details of where the costs come from can be found in `itemized`. The `substance` of each item under `itemized` clarifies the location for the item and the `name` describes it. The `dimension` and `unit` define the quantity. For convenience, items of the same substance are added together under `categories`.

```
{
    "output": {
        //...
        "costs": {
            "total": 29306.072717504845,
            "materials": 20320.185836476725,
            "labour": 8985.886881028118,
            "carbon": 27718.98282711403,
            "itemized": [
                {
                    "substance": "slab",
                    "unit": "sqm applied",
                    "name": "Foundation cast-in-place concrete slab",
                    "dimension": 171.80520538414618,
                    "materials": 2921.88459937037,
                    "labour": 1442.4493591828407,
                    "carbon": 22678.287110707297,
                    "total": 4364.33395855321
                },
                //...
            ],
            "categories": {
                "slab": {
                    "materials": 3040.803807147814,
                    "labour": 1442.4493591828407,
                    "total": 4483.2531663306545,
```

```
                    "carbon": 22678.287110707297
                },
                //...
            }
        }
    }
}
```

`rebates` is a separate property under `output`. The `total` of it can be deducted from the `total` of cost to tell user the true cost of project.

```
{
    "output": {
        //...
        "rebates": {
            "total": 3200,
            "items": [
                {
                    "total": 1000,
                    "notes": ["For upgrading ACH from 11 to 7", "Greener Homes
Program"]
                },
                //...
            ]
        }
    }
}
```

Batch Analysis

The Batch analysis takes in a number of alternatives (instead of one) for one or more parts of the Seed and finds the most cost-effective option amongst them. For example, if the user is unsure about using a Wood-Framed window or a PVC-Framed alternative, the user will give both to InsightEngine. InsightEngine will "cherrypick" the best result.

Multiple options can be batched together. For example, if the user is also wondering about R40 vs R60 attic, they can add those to their batch analysis too. Currently up to 50,000 batches can be simulated in one run. This limit is made because of the exponential nature of mixing multiple batches, multiplying the number of runs with each added option.

InsightEngine supports two batching processes:

- Compress Batching
- Swap Batching

It is easy to populate the Seeds by using Compress Batching. Turn one or more primitive nested properties inside `house` to an array and mention the key in `meta.compressed`. The result is multiple Seeds.

```
{
    "meta": {
        //..
        "compressed": ["entity", "ach"]
    },
    "house": {
        "space": {
            //...
            "ach": [1, 2, 3.5, 5]
        },
        "equipment": {
            //..
            "mech": {
                //...
                "entity": ["mech_comboHE","mech_baseboard"]
            }
        }
    }
}
```

The above makes 16 Seeds. The first seed will have `ach` of `1` and `entity` of `mech_comboHE` for mechanical system and so on.

The strings passed to `meta.compressed` do not need to be *exact*. For example, `slab_entity` is also captured when `entity` is specified in `compressed`.

Compress Batching is simple to use but has a major weakness: it can't change multiple items at once. For example, if a user wants to test two windows in their home:

```
{
    "meta": {
        //..
        "compressed": ["entity"]
    },
    "house": {
        "walls": [
            {
                //...
                "windows": {
                    //...
                    "entity": ["window_wood", "window_PVC"]
                }
            },
            {
                //...
                "windows": {
                    //...
                    "entity": ["window_wood", "window_PVC"]
                }
            }
```

```
        ],
    }
}
```

This Compression will cause 4 Seeds, 2 of which are unintended because one window is `window_wood` and the other window is `window_PVC`. To change both windows at the same time, the Swap Batching can be used:

```
{
    "meta": {
        //..
        "swaps": {
            "window_wood": ["window_PVC"]
        }
    },
    "house": {
        "walls": [
            {
                //...
                "windows": {
                    //...
                    "entity": "window_wood"
                }
            },
            {
                //...
                "windows": {
                    //...
                    "entity": "window_wood"
                }
            }
        ],
    }
}
```

This will give 2 Seeds. InsightEngine will look for places in `house` where the keyword `window_wood` is used. It will then run another simulation by swapping the `entity` with `window_PVC`. For more windows, simply pass more names to the corresponding swap's array.

MURB Analysis

A Seed by default represents a Single-Family Building, but the platform is perfectly capable of handling Multi-Unit Residential Buildings (MURBs) as well. A MURB analysis always starts with a collection of Seeds (more than 1). The Seeds can be used in two ways:

1. **Assembled:** When each Seed represents a part of a MURB like an apartment. For example, one Seed might specify the common spaces, a second one could be for commercial units, and a third Seed might contain the residential areas. These Seeds are first sent to the server to be *assembled* and then are run as one building with MURB loads.

2. **Entangled:** When the MURBs are ground-oriented (houses in a row with shared walls), they must be assessed individually. Since they are connected, the suggestions across all of them must be the same. For example, if a duplex is modeled as two separate units, the batch analysis may cherrypick an exterior-insulated wall for one, and an interior-insulated wall for the other. This is possible because even if the units perfectly mirror each other, the difference in window orientations changes the loads, and thus the suggestions. Instead, InsightEngine must be instructed to run the simulations in sync and give a single suggestion that works well for all units.

Both system require sending an array of Seeds. For Assembled, the array should go to the assembly end-point and then be used in the usual fashion. For Entangled, the Seeds must all be sent to the main end-point at the same time without assembly.