hmd-tmpl-modeler

Release 1.0

HMD Labs

Apr 12, 2024

CONTENTS

NeuronSphere Modeler is a polyglot data model collaboration platform.

You may take a portion of a *standard* or *extended* model and export it through templates to produce code, configuration, documentation, and more.

This repository contains the templates used in the NeuronSphere Modeler, as well as tooling and convenience functions to allow creating and extending new template sets.

CHAPTER

ONE

BASIC USE

NeuronSphere Modeler uses the freely-available code-generating tool *hmd mickey*. Mickey is a python-based cli that uses the templating library jinja2.

To render these templates with your data models on your computer:

1. Install *mickey*

```
projects %> pip install hmd-cli-mickey
```

2. Clone the template repository

```
projects %> git clone https://github.com/NeuronSphere/hmd-tmpl-modeler.git
```

3. Add your models - while *hmd mickey* can consume an array of data formats containing essentially any data model(s), the templates in this repository are all designed to consume NeuronSphere Entity Models and their extensions.

   Use one of these options:

   *. **Use the NeuronSphere Modeler to create and collaborate on** your data model and export it. Within the exported zip file there is a 'src' directory containing a 'schemas' folder. Place that schemas folder within the 'src' folder of this (hmd-tmpl-modeler) repository.

   . ***Manually create the appropriate *.hms* files and place them into 'src/schemas/<namespace>' in** this (hmd-tmpl-modeler) repository. This is possible, but annoying to do manually.

   . *Use a tool or script to extract *.hms* model files from relational or other data sources.

4. Run mickey for all templates

```
projects %> cd hmd-tmpl-modeler
hmd-tmpl-modeler %> hmd mickey build
```

1.1 Repository Layout

- /docs - docs and drawings as code, rendered with *hmd-cli-bartleby*
- /src - souce code and models
- /test - RobotFramework test suites. Will be extended to include unified test drivers across the primary template outputs.
- /meta-data - NeuronSphere manifest.json, VERSION file, and other NeuronSphere tooling & project metadata.

This repository is a *NeuronSphere Compliant Repository*

CHAPTER

TWO

NEW TEMPLATES

Adding new templates is a simple process:

1. Add new director(ies) and template(s)
    - Create a new directory in {project_root}/src/mickey/templates to contain your new templates. Simple name, no spaces or special characters.
    - Use *mickey magic folders* such as {{ mickey.src_folder }} as a folder name and it's content will be placed in {project_root}/src. You may also use: {{ mickey.docs_folder }}, {{ data.namespace }}, and {{data.name}} (based on template cardinality).
2. Update manifest configuration
    - Add a new entry in the file {project_root}/meta-data/manifest.json under the *generate* key. This is the configuration that maps a logical name in the mickey config section of the manifest to a collection of models, templates, and hooks.

      This example adds a config named *pydantic*

```
"pydantic": {
    "contexts": ["ref:entity_data"],
    "templates": ["hmd-tmpl-modeler/pydantic"],
    "hooks": ["./src/mickey/hooks/add_jinja_filters.py"]
}
```

      Note: The *contexts* key above determines the *template set cardinality*. This determines if the templates are rendered once per *Entity* in the data model, or rendered once, with all *Entities'* rendering in a single output.

      Looking at the examples should clarify this subject.

3. Execute and test until satisfied with output
    - Executing *hmd mickey build* will render all configurations in the manifest, but during highly iterative development or for special uses this will wastes time rendering configurations that are not needed.

      Instead, execute *hmd mickey build pydantic* to render just the new example added above.

2.1 Template Pack Design Guidelines

1. Name the folder containing your templates *well*. What's in a good name? Short, no special characters, meaningful to what the template pack produces.
2. Decide were the outputs will *go* in your project's source tree and as a part of builds. Depending on your technology and code-generation strategies, it may be easy to place all generated code into a /gen directory at the project root, or in a src/gen directory.
3. Your templates should almost always output something into /docs via the {{mickey.docs_folder}}. Generating documentation that matches the code, configuration, and data models is a useful and powerful side-effect of model driven development.
4. Please mind the .gitignore

CHAPTER

THREE

SCM & CI

When using code generators in the software development process one should consider source control and *Continuous Integration* (CI).

Many software development tools are some form of code-generator, at least in an abstract sense. We take *source code* in one language, supply it to a *compiler*, and it produces some output, often either more source code or a *binary* of some kind (which is actually just more *code*, though usually for the machine and not human consumption).

Source Control Management (SCM) tools (e.g.: *git* [hub/lab/etc]) are designed to store multiple versions of our code over time, like an extra-fancy shared file system.

Generally, we don't store the results of code compilation in SCM, rather storing those results in some kind of artifact distribution system e.g.: docker's image repositories, maven repositories, Pypi for python, etc.

With code-generation this presents a challenge as the outputs may not always be in a package based on a technology with a distribution ecosystem. There are a few ways to handle this:

1. Model & Build -> Discard Models, hand-modify output -> Check in output : This is *Full Self-Flagelation**
2. Model & Build -> Check in Models, check in output(unchanged) : Let's call this *Manual Automation*
3. Model & Build -> Check in Models, *CI* pipeline renders output -> package/tag/compile/publish results : *Model Driven Development*

In a near future release, Modeler will support direct template rendering into git branches. If you have strong opinions about how that should work, please let us know.

CHAPTER

FOUR

PERSPECTIVES AND EXTENSIONS - INFINITE MODELING

One way to discect software is into 2 primary parts:

1. Defining data models and structures
2. Defining actions, behaviors, and logic that interacts with the defined models

This useful lens can be applied across a vast number of technologies - examples include:

. *Object Oriented Programming specifically grounds itself in modeling *state* and *behavior* . *SQL subdivides into Data *Definition* Language (DDL) and Data *Manipulation* Language (DML) *. Libraries such as python's Pydantic, or the Lombok library in java serve as advanced ways to

represent object models with validation and standardized behaviors

*. **RCP definition specs such as OpenAPI or Protobuf allow definitions of rich object models** and language agnostic function definitions that use these models

4.1 Model Impedence Mismatches

The base modeling construct in NeuronSphere Modeler is a labeled property graph with an intentionally simplified type & relationship system.

Entities, specifically *Nouns* and *Relationships* form a basic underlying structure that can be used to implement a vast number of more specific data modeling and governance capabilities.

Across technologies, there is over and over a repeated demand to use metadata to generate data representations, but there are 2 primary challenges.

1. Type system mismatches happen between any 2 tech stacks or systems
2. Relationship representations vary broadly, but an attribute-carrying relationship in a property graph allows maximum flexibility across manifestations

Note: Why not an existing schema such as json? Ironically, schema is too powerful, too expressive. Use of a schema 'anchors' so strongly that entire ecosystems will be *off-limits* due to complexity/mismatch.

Put another way, detailed object schema systems are challenged as a lowest-common-denominator as input to an *easily* extensible data modeling & Model Driven Engineering framework.

4.2 Perspectives

Perspectives are the named visual display variations used in Modeler to adapt the user experience to a more specific data modeling paradigm.

Different roles in an organization will use different perspectives to update the relevant parts of a model based on the project and technology chosen.

Examples include:

- There is a "Base SQL" perspective that allows adding metadata about tables and SQL datatypes to entities and their attributes.
- A programmer might want to generate python dataclasses or pydantic models using the python-dataclass and python-pydandic perspectives respectively.
- A data quality perspective allows logical constraint capture, while perspectives allow seeing the implementation in multiple manifestations. We'd like to see that an attribute will materialize as "Not Null" in a database, but will have a similar constraint declaration in programatic data models and APIs based on the same conceptual model.
- We might add a perspective for a particular company or project to allow collaboration on additional data governance metadata that can be output into custom documentation or downstream systems.

In a *SQL Centric* modeling paradigm, we pass from *Conceptual Modeling* into *Logical Modeling* and then *Physical Modeling*. Perspectives on a base conceptual model allows this progression, but also allows similar progressions into other modeling paradigms.

What does this look like? We can have a discussion about a few new entities and attribute changes in a data model, and then percolate those changes through multiple technology stakeholders for feedback, and instantiate any collection of custom data models into git for further analysis.

4.2.1 Entity Graph Overall Display

We *switch to a perspective* in the main graph UI, and it impacts the display of all Nouns and Relationships.

4.2.2 Noun Display

Currently, the behavior of the UI is such that switching perspectives updates the displayed information in the headers, and styles, of all *Nouns* in the display.

The visual format of the attribute list is also dependent on the perspective in effect.

4.2.3 Relationship Display

The visual representation of relationships is a key differentiator between perspectives.

Showing a relationship as a visual rectangle between the Nouns it joins versus visualizing the relationship as a line and using hover/click interactions to access the relationship editor.

> **Warning:** In the core entity model, we are allowed to model attributes on relationships and create a proper logical model. When visualizing this model directly in the tabular perspective, there may be "mismatches" in the cardinalities vs properties vs mapping to physical table that may not be visually obvious at this time.
>
> The simple example is mapping a relationship with several attributes onto a 1-n physical relationship. Placement of the relationship attributes is more naturally in a 1-n/m-1 bridging table.

4.2.4 Attribute Editor

When editing an entity, (noun or relationship), there are several places where the applicaiton of a perspective changes the UI.

A perspective might add *extension* data for all entities, but more often will be applied to nouns or all relationships in a perspective. Examples include:

1. The “table_name” extension available as a property to all entities when using any of the SQL-based perspectives. This is an entity-level extension.
2. A “rel_type” extension available as a property to relationships in a SQL-based perspective.

More common and powerful are *attribute extensions* that are available when a perspective is active in the attribute editor. This presents as new columns to the right of the existing columns in the attribute editor.

Consider the basic attributes:

Name	Desc	Type
fname	False	String
salary	False	Float
height	True	Int
faves	True	Collection

If we activate the python-dataclass perspective when editing the attribute list, we get the option to edit the metadata for each attribute that is specific to the perspective.

Name	Desc	Type	DataclassType
fname	False	String	string
salary	False	Float	float
height	True	Int	integer
faves	True	Collection	list

If we activate the SQL2023 perspective when editing the attribute list, we see several other more advanced options:

Name	Desc	Type	Sql2023.Type	Sql.Nullable
fname	False	String	varchar({p0})	false
salary	False	Float	float({p0},{p1})	true
height	True	Int	int	false
faves	True	Collection	array	true

Note: Much of the purpose of this ability is to allow collecting the broadly polyglot metadata and making it available in the templates associated with the perspective.

4.3 Extensions (Perspective Data Storage)

Perspectives define how a specific modeling construct visually appear, and control what additional metadata may be stored at various levels of the model.

Extensions describe where additional metadata will be stored in the json representation of an object model.

4.4 Import vs Inheritance

We hate to repeat ourselves, and programmers even have a nifty little acronym - DRY (Don't Repeat Yourself). In data modeling, it is the repeating ourselves in impossible-to-track ways that are the roots of many problems.

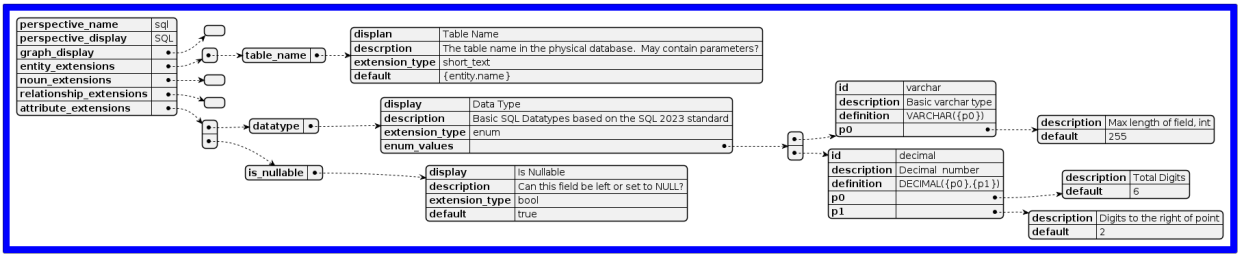
- **Import** is to copy an existing Noun, Relationship, or colleciton thereof into another model. This is a *copy* operation, and the new Entities can be freely edited, snapshotted, and modified. A reference to the original entity is tracked, and a *reconcile with ancestor* function allows tracking the delta between versions of entities across models, assuming access. This is repeating yourself, but is often a reasonable approach for rapid development or high domain-cardinality data models. Changes to original models are not propagated/notified.
- **Inherit** is to create an Entity based on an existing Entity, with the ability to add new attributes and perspectives, but not change existing attributes or extensions. Changes to source models can be automatically applied. Multi-inheritence is allowed/encouranged as a pattern.
- **Attribute References** allow the re-use of individual governed attributes from existing models in a new model.

A programmer might want to start with an existing entity, add several new attributes, and then generate python dataclasses or pydantic models using the python-dataclass and python-pydandic perspectives respectively.

4.5 Defining Perspectives as Metadata

Shows explicit structures for perspective definitions, specifically how the various types/enums are defined.

Examples of the basic SQL perspective:




```
{
    "perspective_name" : "sql",
    "perspective_display" : "SQL",
    "graph_display" : {},
    "entity_extensions" : [
        {
            "table_name":{
```

(continues on next page)

(continued from previous page)

```
                "displan": "Table Name",
                "descrption": "The table name in the physical database.  May contain␣
↪parameters?",
                "extension_type": "short_text",
                "default" : "{entity.name}"
            }
        }
    ],
    "noun_extensions" : {},
    "relationship_extensions": {},
    "attribute_extensions" : [
        {
            "datatype" : {
                "display": "Data Type",
                "description" : "Basic SQL Datatypes based on the SQL 2023 standard",
                "extension_type" : "enum",
                "enum_values" : [
                    {
                        "id":"varchar",
                        "description" : "Basic varchar type",
                        "definition" : "VARCHAR({p0})",
                        "p0" : {
                            "description" : "Max length of field, int",
                            "default" : 255
                        }
                    },
                    {
                        "id":"decimal",
                        "description" : "Decimal  number",
                        "definition"  : "DECIMAL({p0},{p1})",
                        "p0" : {
                            "description" : "Total Digits",
                            "default": 6
                        },
                        "p1" : {
                            "description" : "Digits to the right of point",
                            "default": 2
                        }
                    }
                ]
            }
        },
        {
            "is_nullable" : {
                "display": "Is Nullable",
                "description": "Can this field be left or set to NULL?",
                "extension_type": "bool",
                "default": "true"
            }
        }
    ]
}
```