

## D7.4 – Report on Battery Interface Ontology Case Study

### VERSION

VERSION	DATE
V1	31.08.2022

### PROJECT INFORMATION

GRANT AGREEMENT NUMBER	957189
PROJECT FULL TITLE	Battery Interface Genome - Materials Acceleration Platform
PROJECT ACRONYM	BIG-MAP
START DATE OF THE PROJECT	1/9-2020
DURATION	3 years
CALL IDENTIFIER	H2020-LC-BAT-2020-3
PROJECT WEBSITE	big-map.eu

### DELIVERABLE INFORMATION

WP NO.	7
WP LEADER	SINTEF
CONTRIBUTING PARTNERS	EPFL, DTU
NATURE	Report
AUTHORS	Simon Clark
CONTRIBUTORS	Casper W Andersen, Jesper Friis, Francesca Lønstad Blekken
CONTRACTUAL DEADLINE	31.08.2022
DELIVERY DATE TO EC	31.08.2022
DISSEMINATION LEVEL (PU/CO)	PU

### ACKNOWLEDGMENT



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957189. The project is part of BATTERY 2030+, the large-scale European research initiative for inventing the sustainable batteries of the future.



---

## ABSTRACT

The objective of this deliverable is to demonstrate a working proof-of-concept for how the Battery Interface Ontology (BattINFO) can be used to create semantically annotated battery data. Achieving this objective means developing not only the ontology itself, but also the associated data models and RDF triple mapping and querying infrastructure needed to perform practical actions on real battery datasets. In this first use-case, we create battery cell metadata for a cell that has been reported in the literature and link it with simulated time-series data obtained from the open-source Battery Modelling Toolbox (BattMo). The resulting metadata are mapped to BattINFO terms using RDF triples and saved in a triplestore. We demonstrate a simple semantic query of the triplestore using SPARQL. The code for generating this use-case – along with future use-cases to be developed in the BIG-MAP project - is publicly available on GitHub: <https://github.com/BIG-MAP/FAIRBatteryData>



---

## TABLE OF CONTENTS

<b>1. INTRODUCTION</b>	<b>4</b>
<b>2. METHODS</b>	<b>4</b>
2.1 BATTERY INTERFACE ONTOLOGY (BATTINFO)	4
2.2 DLITE	5
2.3 DATA SOURCES	5
2.4 USE-CASE REPOSITORY	6
<b>3. FIRST USE-CASE</b>	<b>6</b>
3.1 CREATE DLITE DATA MODELS	7
3.2 ASSIGN CELL DATA AND TIME-SERIES DATA TO DLITE INSTANCES	11
3.3 CREATE A TRIPLESTORE AND ADD MAPPING TRIPLES	17
3.4 QUERY THE TRIPLESTORE USING SPARQL	18
<b>4. SUMMARY</b>	<b>19</b>
<b>REFERENCES</b>	<b>20</b>

## 1. Introduction

The world is generating more and more battery data every day. However, despite this abundance of information, the battery community is using the available data to only a fraction of its full potential. This is due largely to the challenges of working with heterogeneous data that comes from different sources (e.g. machines, models, labs, companies, etc.). In the absence of clear community guidelines, each source reports its data in its own format and may or may not include the appropriate metadata needed to fully understand the context. As a result, the battery community is losing valuable resources in the quest for new energy storage technologies.

The EU project BIG-MAP has developed the Battery Interface Ontology (BattINFO) to help address this challenge [1]. BattINFO is a domain ontology for batteries and electrochemistry under the umbrella of the top-level Elementary Multi-perspective Materials Ontology (EMMO). The purpose of BattINFO is to provide a common community-wide conceptualization of batteries and their data to fully enable data interoperability and automated machine reasoning about battery data. BattINFO is structured as a machine-readable data model that describes batteries and their data as a collection of concepts linked together by relations. Properties of battery data can then be mapped to concepts in the ontology. For example, if you have two battery data sets – each with properties that map to the same term in the ontology – a machine is then able to reason that because those two properties are equivalent to a third concept, they are therefore equivalent to each other. Ontological reasoning can go a step further and navigate the web of relationships connecting concepts to identify dependencies that may not have been obvious otherwise.

The purpose of this deliverable is to create a first use case demonstrating a proof-of-concept for how BattINFO can be used to enhance the battery data space. The selected use case will focus on interpreting and post-processing time-series battery data (e.g. cell cycling data), which is one of the most common types of data generated in the battery field. We will present the ontological concepts necessary to describe both the data itself and the associated metadata for the cell. We use a tool developed within the framework of EU data science research called DLite to create generic data models that include the mappings to the BattINFO ontology. We can then create ontology-informed functions to perform common operations and calculations on the data. As a result, we will demonstrate the easy interoperability between datasets that were originally reported in different formats. Finally, we will discuss next steps for further use case development.

## 2. Methods

In this section, we review the tools and methods used to create the case study.

### 2.1 Battery Interface Ontology (BattINFO)

BattINFO is a free, open-source domain ontology for batteries developed beneath the umbrella of the top-level EMMO. It is available for download on github, using the following link: <https://github.com/BIG-MAP/BattINFO> In this case study, we are working with version 0.3.0. The domain ontology can be obtained by cloning the git repo using the command:

```
git clone https://github.com/BIG-MAP/BattINFO.git
```

There are two tools we recommend to explore and use the ontology. The first is the free tool Protégé, developed by Stanford University. Protégé, provides an easy-to-use interface to explore and edit terms in the ontology. It is available for download using the following link: <https://protege.stanford.edu/products.php#desktop-protege>

The second tool we recommend is a python package for working with EMMO and its associated domain ontologies called EMMOntoPy. It can be installed using the following command:

```
pip install EMMOntoPy
```

## 2.2 DLite

To create mappings between ontological terms and actual data sets, we use a lightweight data-centric framework for semantic interoperability called DLite. DLite is a C implementation of the SINTEF Open Framework and Tools (SOFT), which is a set of concepts and tools for how to efficiently describe and work with scientific data.

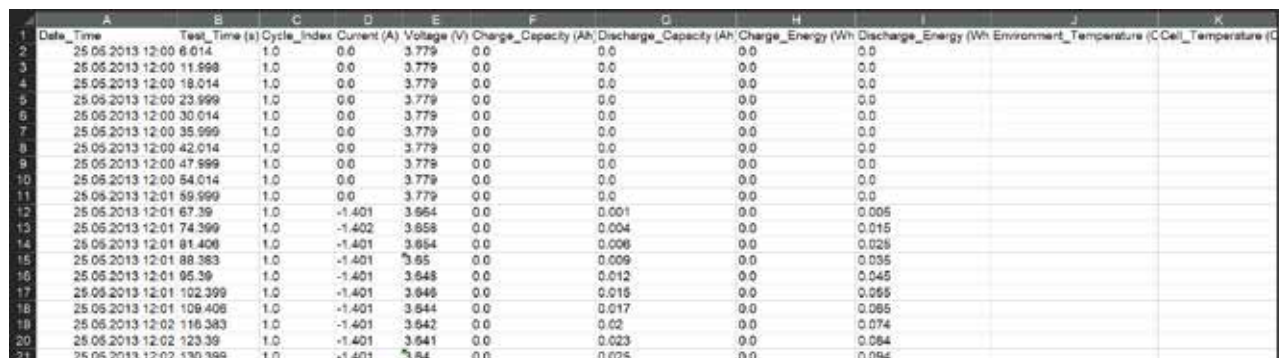
If you are using Python, the easiest way to install DLite is with pip:

```
pip install DLite-Python
```

Note, currently only Linux versions for Python 3.7, 3.8, 3.9 and 3.10 are available. But Windows versions will soon be available.

## 2.3 Data sources

There are a growing number of publicly available databases and simulation tools for battery data. The online database <https://www.batteryarchive.org/>, collects open data sets describing the cycling performance of many different types of Li-ion batteries. An example is shown in Figure 1. They include some basic metadata descriptors for properties like electrode materials and form factor, which are encoded in the entry name.



	A	B	C	D	E	F	G	H	I	J	K
	Date_Time	Test_Time (s)	Cycle_Index	Current (A)	Voltage (V)	Charge_Capacity (Ah)	Discharge_Capacity (Ah)	Charge_Energy (Wh)	Discharge_Energy (Wh)	Environment_Temperature (C)	Cell_Temperature (C)
2	25.05.2013 12:00 6.014	1.0	0.0	3.779	0.0	0.0	0.0	0.0	0.0		
3	25.05.2013 12:00 11.999	1.0	0.0	3.779	0.0	0.0	0.0	0.0	0.0		
4	25.05.2013 12:00 18.014	1.0	0.0	3.779	0.0	0.0	0.0	0.0	0.0		
5	25.05.2013 12:00 23.999	1.0	0.0	3.779	0.0	0.0	0.0	0.0	0.0		
6	25.05.2013 12:00 30.014	1.0	0.0	3.779	0.0	0.0	0.0	0.0	0.0		
7	25.05.2013 12:00 35.999	1.0	0.0	3.779	0.0	0.0	0.0	0.0	0.0		
8	25.05.2013 12:00 42.014	1.0	0.0	3.779	0.0	0.0	0.0	0.0	0.0		
9	25.05.2013 12:00 47.999	1.0	0.0	3.779	0.0	0.0	0.0	0.0	0.0		
10	25.05.2013 12:00 54.014	1.0	0.0	3.779	0.0	0.0	0.0	0.0	0.0		
11	25.05.2013 12:01 59.999	1.0	0.0	3.779	0.0	0.0	0.0	0.0	0.0		
12	25.05.2013 12:01 67.39	1.0	-1.401	3.954	0.0	0.001	0.0	0.005	0.0		
13	25.05.2013 12:01 74.399	1.0	-1.402	3.958	0.0	0.004	0.0	0.015	0.0		
14	25.05.2013 12:01 81.406	1.0	-1.401	3.954	0.0	0.006	0.0	0.026	0.0		
15	25.05.2013 12:01 88.383	1.0	-1.401	3.95	0.0	0.009	0.0	0.035	0.0		
16	25.05.2013 12:01 95.39	1.0	-1.401	3.948	0.0	0.012	0.0	0.045	0.0		
17	25.05.2013 12:01 102.399	1.0	-1.401	3.946	0.0	0.015	0.0	0.055	0.0		
18	25.05.2013 12:01 109.406	1.0	-1.401	3.944	0.0	0.017	0.0	0.065	0.0		
19	25.05.2013 12:02 116.383	1.0	-1.401	3.942	0.0	0.02	0.0	0.074	0.0		
20	25.05.2013 12:02 123.39	1.0	-1.401	3.941	0.0	0.023	0.0	0.084	0.0		
21	25.05.2013 12:02 130.399	1.0	-1.401	3.94	0.0	0.025	0.0	0.094	0.0		

Figure 1. An example of battery time-series data, downloaded from BatteryArchive.org

Other open-source simulation tools like PyBaMM, cideMod, and BattMo are becoming widely used across both industry and research. Over the course of the BIG-MAP project, we intend to offer support for annotating data from these and other public sources.

For this initial demonstrator use-case, we use cell data reported by Chen et al. in their well-documented parameterization of cylindrical 21700 Li-ion battery cells [2]. This is combined with time-series cell data simulated using the battery simulation software BattMo. The intention of this use case is to demonstrate the workflow for annotating and transforming battery data. It is the method – more than the actual data itself – that is the key component of this work.

The data used for the cell description can be found in the publication by Chen et al. [2]. The cell simulation data can be obtained by executing the example runChem2020.m in the BattMo repository: <https://github.com/BattMoTeam/BattMo>

## 2.4 Use-case repository

A public repository has been created on GitHub for the purpose of documenting FAIR battery data use-cases supported by BattINFO. The repository can be accessed using the following link:

<https://github.com/BIG-MAP/FAIRBatteryData>

The code for the first use-case is included in this repository under the location: examples/scripts/demo-1-createAnnotatedBatteryData.py. Further use-cases will be added and documented as the BIG-MAP project progresses.

## 3. First use-case

The selected first use-case is to annotate cell-level and time-series battery data with BattINFO. Annotating data to ontology terms is achieved by defining semantic triples, which take the form: subject, predicate, object. For example, if we wanted to map the cell voltage data from some battery measurement (with a specific UUID) to the BattINFO term for cell voltage, we might define a triple that takes the form:

```
BattINFO:CellVoltage map:mapsTo UUID#cell_voltage
```

In this example, the subject is the CellVoltage term defined in the BattINFO ontology, the predicate is the mapsTo relation defined by the EMMO Mapping ontology, and the object is the cell\_voltage property of the measurement indicated by its specific UUID. While in this case the subject and the predicate can be obtained from the relevant ontologies, we need an adequate data model to serve as a basis for identifying the object of the triple. We achieve this by creating instances in DLite. DLite instances are simple metadata that can be linked to or generated from ontologies. They provide property fields that can be populated with data coming from standard sources such as csv, xlsx, json, or data bases like Postgresql, etc. A conceptual overview of the layers linking raw data to ontology terms is presented in Figure 2.

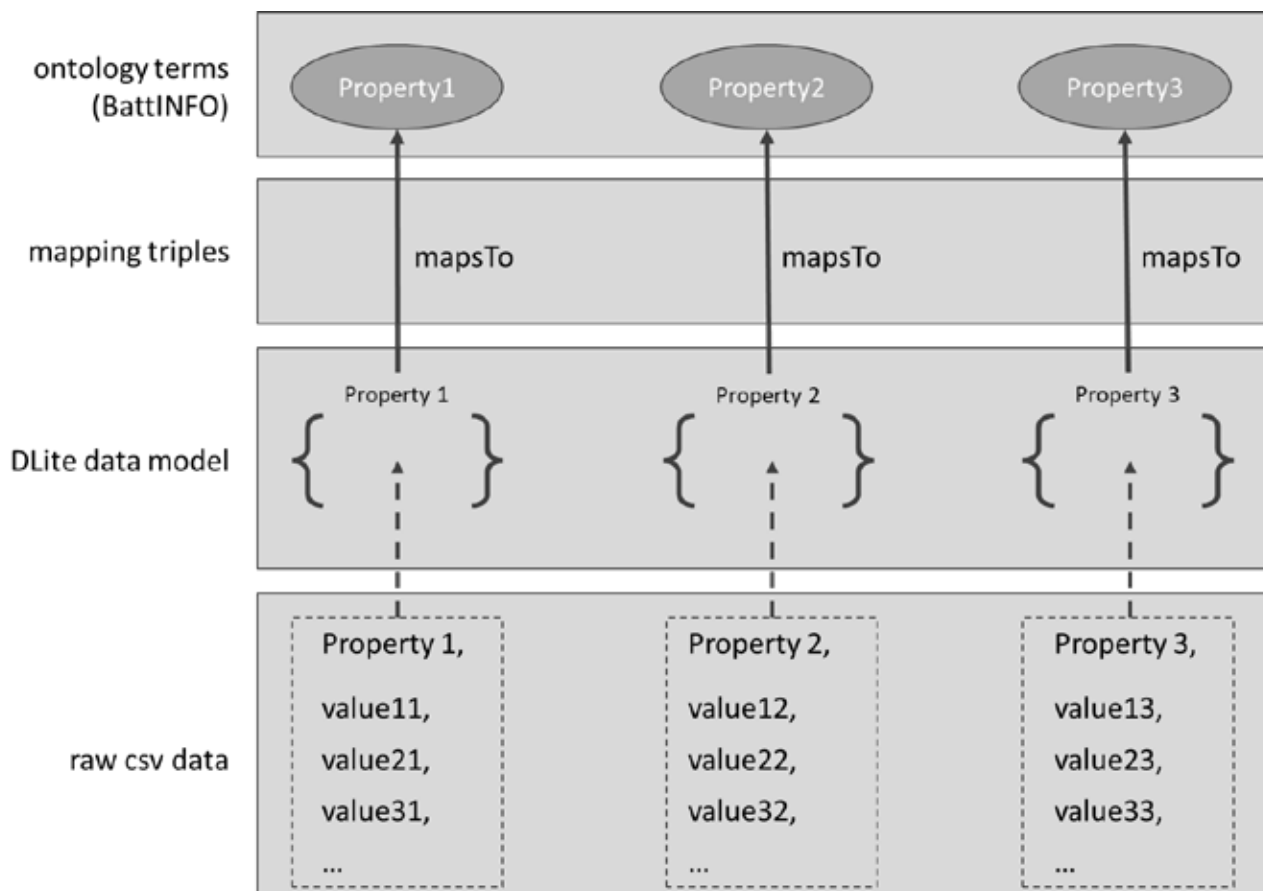


Figure 2. Schematic overview of the layers used to link raw data to ontology terms. The figure shows an example for raw data coming from a csv file, but data from many sources (e.g. JSON, SQL database, excel, etc.) could be considered.

The following steps are undertaken to create the use case:

- i. Create a DLite data models describing the battery and its associated time-series data.
- ii. Assign cell data and read the time-series data from its raw file contents (\*.csv).
- iii. Instantiate a triplestore containing BattINFO triples and add triples mapping the data
- iv. Query the triplestore using SPARQL

The details of each step are described in the following sub-sections.

### 3.1 Create DLite data models

DLite instances as the basis for ontology-based data interoperability. The first step in creating DLite instances and collections is to establish a generic data model for the data in question. In this example, we define two foundational data models: BatteryCellMetadata and BatteryTimeSeriesData.

The first data model describes the metadata for the battery cell itself. This includes references to sub-models for the cell components and materials. The second data model holds the state quantities from the time-series data. We choose to separate the state quantities and derived quantities because the raw data itself is inconsistent. Different raw data files often report different derived

data quantities. Therefore, to make the approach as widely applicable as possible, we only collect the state quantities from the raw data itself. We then define functions to calculate the derived quantities and map them to the corresponding data model.

Figure 3 shows a schematic overview of the structure of a DLite instance. A DLite instance has four parts: a unique universal identifier (UUID), a description of the metadata schema used to generate the instance, dimensions, and properties that contain the actual data. An instance is defined from a metadata schema which itself has three parts: a URI, dimensions and properties. Each property definition must have a name, a description, and a type (e.g. string, float, int, etc.). If it is necessary to constrain the dimensions, this can be set using the dims field. If the property is a physical quantity with units, the type of unit can be defined with the unit field.

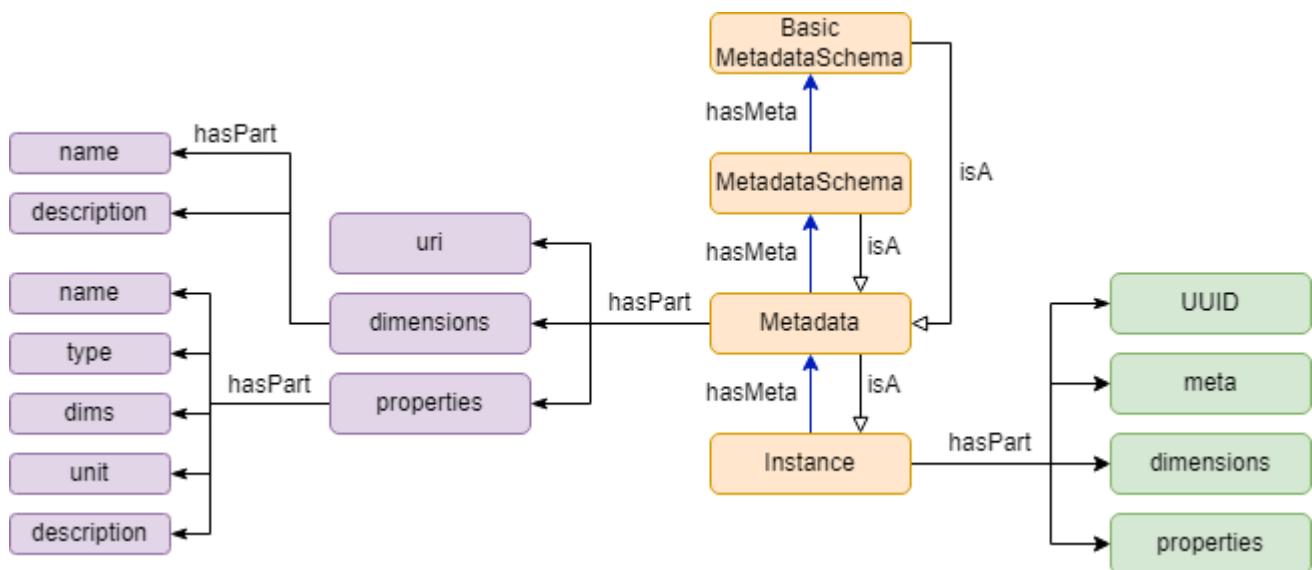


Figure 3. An overview of the DLite approach to creating data models.

Figure 4 shows an example DLite data model definition for BatteryCellMetadata. The URI for the data model is composed from the combination of the name, version, and namespace fields. The description is a human-readable text defining the scope of the data model. We define the dimensions of the data model as  $N$  number of battery cells, followed by a list of properties.

```
{
  "name": "BatteryCellMetadata",
  "version": "0.1",
  "namespace": "http://onto-ns.com/meta",
  "description": "A description of battery cell metadata",
  "dimensions": [
    {
      "name": "N",
      "description": "Number of battery cells"
    }
  ],
  "properties": [
    {
```



```
    "name": "uui d",
    "type": "string",
    "dims": ["N"],
    "description": "UUID of the battery cell"
  },
  {
    "name": "positive_electrode",
    "type": "ref",
    "$ref": "http://onto-ns.com/meta/0.1/BatteryElectrodeMetadata",
    "dims": ["N"],
    "description": "positive electrode metadata"
  },
  {
    "name": "negative_electrode",
    "type": "ref",
    "$ref": "http://onto-ns.com/meta/0.1/BatteryElectrodeMetadata",
    "dims": ["N"],
    "description": "negative electrode metadata"
  },
  {
    "name": "electrolyte",
    "type": "ref",
    "$ref": "http://onto-ns.com/meta/0.1/BatteryElectrolyteMetadata",
    "dims": ["N"],
    "description": "electrolyte metadata"
  },
  {
    "name": "separator",
    "type": "ref",
    "$ref": "http://onto-ns.com/meta/0.1/BatterySeparatorMetadata",
    "dims": ["N"],
    "description": "separator metadata"
  },
  {
    "name": "casing",
    "type": "ref",
    "$ref": "http://onto-ns.com/meta/0.1/BatteryCasingMetadata",
    "dims": ["N"],
    "description": "cell casing metadata"
  }
]
}
```

Figure 4. BatteryCellMetadata DLite data model.

The first property is the battery cell identifier. Each battery cell is assigned a UUID to provide a consistent unique identifier for both tracking of the physical battery cell and for associating the cell with its data. The second property is the positive electrode of the cell. The property type "ref" references another DLite data model for battery electrode metadata given by the uri: <http://onto-ns.com/meta/0.1/BatteryElectrodeMetadata>. Referencing other data models allows designers to make encapsulation decisions for re-useable data models that avoid excess duplication. In the BatteryCellMetadata description, we take the same approach for the negative electrode, electrolyte, separator, and casing. All of the data models referenced in this use-case are included in the FAIRBatteryData repository under: [examples/entities](#).

Figure 5 shows the DLite data model definition for the BatteryTimeSeriesStateData. The first property of the data model is the battery\_id of the battery that was used to generate the data. This references the UUID from the BatteryCellMetadata model. The second property is for the data and timestamp generated by the measurement. The data model then includes five state quantities for describing the time-series battery data: the test\_time, battery\_current, battery\_voltage, battery\_temperature, and environment\_temperature.

```
{
  "name": "BatteryTimeSeriesStateData",
  "version": "0.1",
  "namespace": "http://onto-ns.com/meta",
  "description": "A description of battery time series data",
  "dimensions": [
    {
      "name": "n_measurements",
      "description": "Number of measurements."
    }
  ],
  "properties": [
    {
      "name": "battery_id",
      "type": "string",
      "dims": ["n_measurements"],
      "description": "UUID of the battery that the data describes"
    },
    {
      "name": "date_time_stamp",
      "type": "string",
      "dims": ["n_measurements"],
      "description": "Date and time stamp of the measurement."
    },
    {
      "name": "test_time",
      "type": "float",
      "dims": ["n_measurements"],
      "unit": "s",

```

```

    "description": "Time of the measurement relative to the start of the
experiment. "
  },
  {
    "name": "battery_current",
    "type": "float",
    "dims": ["n_measurements"],
    "unit": "A",
    "description": "Measured instantaneous electric current through the
battery. "
  },
  {
    "name": "battery_voltage",
    "type": "float",
    "dims": ["n_measurements"],
    "unit": "V",
    "description": "Measured instantaneous voltage of the battery. "
  },
  {
    "name": "battery_temperature",
    "type": "float",
    "dims": ["n_measurements"],
    "unit": "degC",
    "description": "Measured instantaneous temperature of the battery. "
  },
  {
    "name": "environment_temperature",
    "type": "float",
    "dims": ["n_measurements"],
    "unit": "degC",
    "description": "Measured instantaneous temperature of the environment. "
  }
]
}

```

Figure 5. BatteryTimeSeriesStateData DLite data model

### 3.2 Assign cell data and time-series data to DLite instances

First, we create a DLite collection of instances that describes the properties of the battery cell. This includes not only an instance for the cell itself, but also all of the necessary components such as the electrodes, electrolyte, active materials, etc. Figure 6 shows the code that we use, which instantiates the cell metadata descriptions in four steps:

1. **Define the metadata models.** This step identifies the relevant DLite data models and prepares them for instantiation.
2. **Create instances.** This step instantiates empty instances of the data models.

3. **Group instances into a DLite collection.** Because the battery cell description requires multiple data models, we group all of the instances into a collection, such that they can be accessed later.
4. **Bind instances together in the cell model.** This step binds the individual instances that we have defined into a single cell model.

```
# 1. Define metadata models
datamodel = "BatteryCellMetadata.json"
datamodel_path = os.path.join(entitydir, datamodel)
BatteryCellMetadata = dlite.Instance.from_url(f'json://{datamodel_path}')

datamodel = "BatteryElectrodeMetadata.json"
datamodel_path = os.path.join(entitydir, datamodel)
BatteryElectrodeMetadata = dlite.Instance.from_url(f'json://{datamodel_path}')

datamodel = "BatteryElectrodeActiveMaterialMetadata.json"
datamodel_path = os.path.join(entitydir, datamodel)
BatteryElectrodeActiveMaterialMetadata =
dlite.Instance.from_url(f'json://{datamodel_path}')

datamodel = "BatteryElectrodeCurrentCollectorMetadata.json"
datamodel_path = os.path.join(entitydir, datamodel)
BatteryElectrodeCurrentCollectorMetadata =
dlite.Instance.from_url(f'json://{datamodel_path}')

datamodel = "BatteryElectrodeAdditiveMetadata.json"
datamodel_path = os.path.join(entitydir, datamodel)
BatteryElectrodeAdditiveMetadata =
dlite.Instance.from_url(f'json://{datamodel_path}')

datamodel = "BatteryElectrodeBinderMetadata.json"
datamodel_path = os.path.join(entitydir, datamodel)
BatteryElectrodeBinderMetadata =
dlite.Instance.from_url(f'json://{datamodel_path}')

datamodel = "BatteryElectrolyteMetadata.json"
datamodel_path = os.path.join(entitydir, datamodel)
BatteryElectrolyteMetadata = dlite.Instance.from_url(f'json://{datamodel_path}')

datamodel = "MolecularEntityMetadata.json"
datamodel_path = os.path.join(entitydir, datamodel)
MolecularEntityMetadata = dlite.Instance.from_url(f'json://{datamodel_path}')

datamodel = "LiquidSolventMetadata.json"
datamodel_path = os.path.join(entitydir, datamodel)
```

```
liquidSolventMetadata = dlite.Instance.from_url(f'json://{datamodel_path}')
datamodel = "BatterySeparatorMetadata.json"
datamodel_path = os.path.join(entitydir, datamodel)
BatterySeparatorMetadata = dlite.Instance.from_url(f'json://{datamodel_path}')

datamodel = "BatteryCasingMetadata.json"
datamodel_path = os.path.join(entitydir, datamodel)
BatteryCasingMetadata = dlite.Instance.from_url(f'json://{datamodel_path}')

collection = dlite.Collection()

# 2. Create instances
cell = BatteryCellMetadata(dims=[1], id='Chen2020Example')

pe      = BatteryElectrodeMetadata(dims=[2])
pe_am   = BatteryElectrodeActiveMaterialMetadata()
pe_cc   = BatteryElectrodeCurrentCollectorMetadata()
pe_bnd  = BatteryElectrodeBinderMetadata()
pe_add  = BatteryElectrodeAdditiveMetadata()

ne      = BatteryElectrodeMetadata(dims=[2])
ne_am   = BatteryElectrodeActiveMaterialMetadata()
ne_cc   = BatteryElectrodeCurrentCollectorMetadata()
ne_bnd  = BatteryElectrodeBinderMetadata()
ne_add  = BatteryElectrodeAdditiveMetadata()

elyte   = BatteryElectrolyteMetadata(dims=[1, 1])
salt    = MolecularEntityMetadata()
solvent = LiquidSolventMetadata(dims=[2])
solvent_component_1 = MolecularEntityMetadata()
solvent_component_2 = MolecularEntityMetadata()

sep     = BatterySeparatorMetadata(dims=[1])
case   = BatteryCasingMetadata()

# 3. Group instances into a DLite collection
collection.add(label='cell', inst=cell)
collection.add(label='positive electrode', inst=pe)
collection.add(label='positive electrode active material', inst=pe_am)
collection.add(label='positive electrode current collector', inst=pe_cc)
collection.add(label='positive electrode binder', inst=pe_bnd)
collection.add(label='positive electrode additive', inst=pe_add)

collection.add(label='negative electrode', inst=ne)
collection.add(label='negative electrode active material', inst=ne_am)
```

```
collection.add(label='negative electrode current collector', inst=ne_cc)
collection.add(label='negative electrode binder', inst=ne_bnd)
collection.add(label='negative electrode additive', inst=ne_add)

collection.add(label = 'electrolyte', inst=elyte)
collection.add(label = 'salt', inst=salt)
collection.add(label = 'solvent', inst=solvent)
collection.add(label = 'solvent component 1', inst=solvent component 1)
collection.add(label = 'solvent component 2', inst=solvent component 2)
collection.add(label = 'separator', inst=sep)
collection.add(label = 'casing', inst=case)

# 4. Bind instances together in the cell model
cell.positive electrode = [pe]
cell.positive electrode[0].active material = pe_am
cell.positive electrode[0].current collector = [pe_cc]
cell.positive electrode[0].binder = pe_bnd
cell.positive electrode[0].additive = pe_add

cell.negative electrode = [ne]
cell.negative electrode[0].active material = ne_am
cell.negative electrode[0].current collector = [ne_cc]
cell.negative electrode[0].binder = ne_bnd
cell.negative electrode[0].additive = ne_add

cell.electrolyte = [elyte]
cell.electrolyte[0].salt = [salt]
cell.electrolyte[0].solvent material = [solvent]
cell.electrolyte[0].solvent material[0].molecular entity[0]=[solvent component 1]
cell.electrolyte[0].solvent material[0].molecular entity[1]=[solvent component 2]

cell.separator = [sep]
cell.casing = [case]from pathlib import Path
import os
import dlite
from oteapi.datacache import DataCache
from oteapi_dlite.strategies.parse_excel import DLiteExcelStrategy

thisdir = Path(__file__).resolve().parent
entitydir = thisdir.parent / 'entities'
xlsxfile = (thisdir / "../data/BatteryTimeSeriesData" /
            "timeseries_full.xlsx")

cache_key = DataCache().add(xlsxfile.read_bytes())

config = {
```

```

"downloadUrl": xlsfile.as_uri(),
"mediaType": "application/vnd.dlite.xlsx",
"configuration": {
  "excel_config": {
    "worksheet": "rawdata",
    "header_row": "1",
    "row_from": "2",
  },
},
}

coll_raw = dlite.Collection()
coll_processed = dlite.Collection()
session = {"collection_id": coll_raw.uuid, "key": cache_key}

parser = DLiteExcelStrategy(config)
session.update(parser.initialize(session))

parser = DLiteExcelStrategy(config)
parser.get(session)
    
```

Figure 6. Example code for instantiating battery cell metadata using DLite.

Next, we assign specific values to the DLite instances. In this use-case, the specific values used are those reported by Chen et al. for the INR21700 M50 cell from LG Chem. The code used to assign the values is shown in Figure 7.

```

# assign properties to the instances. In this demo, we use the properties stated
# by Chen et al in 10.1149/1945-7111/ab9050
cell.name = "INR21700 M50"
cell.manufacturer = "LG Chem"

pe.coating_width = [6.5e-2, 6.5e-2]
pe.coating_length = [79e-2, 79e-2]
pe.coating_thickness = [75.6e-6, 75.6e-6]
pe.coating_porosity = [0.335, 0.335]
pe.coating_tortuosity = [4.8, 4.8]
pe.coating_bruggeman_coefficient = [2.43, 2.43]
pe.loading = 24.69
pe_am.conventional_name = 'NMC811'
pe_am.crystal_density = 4950
pe_am.molecular_weight = 94.87
pe_cc.thickness = 16e-6

ne.coating_width = [6.5e-2, 6.5e-2]
ne.coating_length = [77.5e-2, 83.5e-2]
ne.coating_thickness = [85.2e-6, 85.2e-6]
    
```

```

ne. coating_porosity      = [0.25, 0.25]
ne. coating_tortuosity    = [14.25, 13.93]
ne. coating_bruggeman_coefficient = [2.92, 2.90]
ne. loading               = 14.85
ne_am.conventional_name   = 'Graphite-SiOx'
ne_am.crystal_density     = 2260
ne_cc.thickness           = 12e-6

salt.conventional_name    = 'Lithium Hexafluorophosphate'
salt.linear_formula       = 'LiPF6'
salt.smiles               = '[Li+].[P-](F)(F)(F)(F)F'
salt.molar_mass           = 151.905

solvent_component_1.conventional_name = 'Ethylene Carbonate'
solvent_component_1.linear_formula    = 'C3H4O3'
solvent_component_1.smiles            = 'C1COC(=O)O1'
solvent_component_1.molar_mass        = 88.062
solvent_component_2.conventional_name = "Ethyl Methyl Carbonate"
solvent_component_2.linear_formula    = 'C4H8O3'
solvent_component_2.smiles            = 'CCOC(=O)OC'
solvent_component_2.molar_mass        = 104.10
solvent.molecular_entity_volume_fraction = [0.3, 0.7]

sep.porosity              = 0.47
sep.tortuosity            = 3.27
sep.thickness             = 12e-6
sep.coated                = 'true'
    
```

Figure 7. Example code for assigning values to a battery cell metadata instance.

Now we need to read some time-series data to associate with the cell. One feature of DLite is the possibility to automatically generate a metadata model from a file if none is provided. In this case, we have simulated a 1C discharge of the cell using the battery simulation framework BattMo and saved the results as a csv file.

```

uri = 'http://onto-ns.com/meta/0.1/BatteryTimeSeriesData'
id = 'simulated-discharge-battmo' # Give the dataset a human-readable name
raw_data = dlite.Instance.from_location(
    driver='csv',
    location=datadir / 'BatteryTimeSeriesData/Chen2020_simulated_discharge.csv',
    # Comment out the below line to automatically generate the metadata
    # options=f'infer=false;meta={uri};id={id}',
)
    
```

Figure 8. Example code for reading raw battery time-series data from a csv file and automatically generating a DLite metadata model.



In principle, it is perfectly reasonable to work directly with the metadata model that is generated by DLite automatically. However, if one wishes to conform multiple data sources to a single data model, this can be done, for example, using the code shown in Figure 9.

```
datamodel = "BatteryTimeSeriesData.json"
datamodel_path = os.path.join(entitydir, datamodel)
BatteryTimeSeriesData = dLite.Instance.from_url(f'json://{datamodel_path}')

processed_data = BatteryTimeSeriesData(dims=[raw_data.rows])
processed_data.test_time = raw_data.time
processed_data.battery_voltage = raw_data.voltage
processed_data.battery_current = raw_data.current
processed_data.battery_temperature = raw_data.cell_temperature
processed_data.environment_temperature = raw_data.environment_temperature
```

Figure 9. Example code for reading raw battery time-series data from a csv file and automatically generating a DLite metadata model.

At this point, our DLite instances for both the cell description and the simulated time-series data have been created and filled with data. The next step is to create a triplestore that contains BattINFO terms to perform the mappings.

### 3.3 Create a triplestore and add mapping triples

A triplestore is a way to collect and store a set of RDF triples. This allows us to not only collect triples defined in an ontology like BattINFO, but also to make new triples about how the properties of our data relate to BattINFO terms. Figure 10 shows the code we use to setup the triplestore, `ts`, and load the BattINFO triples. Because BattINFO is a domain ontology of the EMMO, it uses UUID values when creating its object IRIs. This can create problems for humans who want to identify terms based on their `prefLabels` rather than random strings. This is a known issue that will be addressed in future releases. Currently, to deal with that issue we create a dictionary, `d`, mapping object `prefLabels` to the IRI values.

```
ts = Triplestore("rdflib")
ts.parse(f"{ontodir}/battinfo-merged.ttl")

# BattINFO namespace
BATTINFO = ts.bind(
    'battinfo', 'https://big-map.github.io/BattINFO/ontology/BattINFO#')

# Dict mapping prefLabel to IRI
d = {o.value: s for s, o in ts.subject_objects(SKOS.prefLabel)}
```

Figure 10. Code for instantiating a triplestore and populating it with triples from BattINFO.

We now create a set of triples that map aspects of our data to BattINFO terms. We use the `add_mapsTo` method of the triplestore, as shown in Figure 11. In this example, we use our dictionary

of BattINFO prefLabels, d, to find the IRI corresponding to "InstantaneousCurrent" and map it to the battery\_current property of the BatteryData object.

```

# map cell components
ts.add mapsTo(d['Electrode'], cell, 'positive electrode')
ts.add mapsTo(d['Electrode'], cell, 'negative electrode')
ts.add mapsTo(d['Electrolyte'], cell, 'electrolyte')
ts.add mapsTo(d['Separator'], cell, 'separator')
ts.add mapsTo(d['BatteryCellContainer'], cell, 'casing')

# map positive electrode properties
ts.add mapsTo(d['Porosity'], pe, 'porosity')
ts.add mapsTo(d['Tortuosity'], pe, 'tortuosity')
ts.add mapsTo(d['Width'], pe, 'coating width')
ts.add mapsTo(d['Height'], pe, 'coating length')
ts.add mapsTo(d['Thickness'], pe, 'coating thickness')
ts.add mapsTo(d['MolarMass'], pe_am, 'molecular weight')
ts.add mapsTo(d['Thickness'], pe_cc, 'thickness')

# map negative electrode properties
ts.add mapsTo(d['Porosity'], ne, 'porosity')
ts.add mapsTo(d['Tortuosity'], ne, 'tortuosity')
ts.add mapsTo(d['Width'], ne, 'coating width')
ts.add mapsTo(d['Height'], ne, 'coating length')
ts.add mapsTo(d['Thickness'], ne, 'coating thickness')
ts.add mapsTo(d['MolarMass'], ne_am, 'molecular weight')
ts.add mapsTo(d['Thickness'], ne_cc, 'thickness')

# map raw data properties
ts.add mapsTo(d['CellVoltage'], raw_data, 'voltage')
ts.add mapsTo(d['InstantaneousCurrent'], raw_data, 'current')

# map processed data properties
ts.add mapsTo(d['CellVoltage'], processed_data, 'battery voltage')
ts.add mapsTo(d['InstantaneousCurrent'], processed_data,
'battery_current')ts.add mapsTo(d['InstantaneousCurrent'], BatteryData,
'battery_current')

```

Figure 11. Example of data mapping

### 3.4 Query the triplestore using SPARQL

SPARQL is a semantic query language that can retrieve and manipulate data that is stored in our RDF triplestore. Here, we create a simple demonstration that selects all the triples in the triplestore that have subjects which map to the BattINFO term for CellVoltage (electrochemistry:EMMO\_4ebe2ef1\_eea8\_4b10\_822d\_7a68215bd24d). Figure 12 shows the code

used to generate the query and Figure 13 shows the result. In this case, the query returns the properties voltage from our raw\_data instance (e028f01d-3534-4ea1-8b4e-9a565d92b219) and battery\_voltage from our processed\_data instance (8b26fc17-cbf9-4a9f-8207-a4c645535045). This is a simple example for demonstration purposes. SPARQL queries can be powerful tools in exploring datasets, which will be investigated in future use-cases in greater detail.

```
# query the triplestore
query_text = """
PREFIX map: <http://emmo.info/domain-mappings#>
PREFIX emmo: <http://emmo.info/emmo#>
PREFIX electrochemistry: <https://big-
map.github.io/BattINFO/ontology/electrochemistry#>

SELECT *
WHERE {
  ?subject map:mapsTo electrochemistry:EMMO_4ebe2ef1_eea8_4b10_822d_7a68215bd24d
  .
}
"""
query_result = ts.query(query_text)
```

Figure 12. Example SPARQL query of the triplestore, requesting all entities mapped to the BattINFO term CellVoltage.

```
e028f01d-3534-4ea1-8b4e-9a565d92b219#voltage
8b26fc17-cbf9-4a9f-8207-a4c645535045#battery_voltage
```

Figure 13. Result of the SPARQL query

## 4. Summary

The purpose of this deliverable is to demonstrate a first use-case applying terms from the Battery Interface Ontology (BattINFO) to semantically annotate battery data. We select a use-case of battery time-series data, which is among the most abundant type of battery data generated. To create mappings to BattINFO terms, we begin by creating DLite data models for a battery and its time-series data. The DLite data model acts as a template which can be populated with actual data and serve as a basis for creating mapping triple statements. Using data reported in the literature and simulated using the battery modelling toolbox BattMo, we create metadata description of the cell and automatically parse the time-series data from a csv file. We then instantiate a triplestore and fill it with triples from BattINFO. Additional triples are added that create direct mappings between properties of the DLite instance and terms in BattINFO. The triples can then be queried using SPARQL.

Future development will focus on extending the concepts laid out in this deliverable towards other types of battery data. As a next step, we will map the input and output files for battery cell simulations with the goal of establishing easy reproducibility and interoperability of results among different battery simulation software.



---

## References

- [1] S. Clark *et al.*, "Toward a Unified Description of Battery Data," *Adv. Energy Mater.*, vol. 2102702, 2021.
- [2] C.-H. Chen, F. Planella, K. O'Regan, D. Gastol, D. Widanagea, and E. Kendrick, "Development of Experimental Techniques for Parameterization of Multi-scale Lithium-ion Battery Models," *J. Electrochem. Soc.*, vol. 167, p. 080534, 2020.