# RESO Standard Event Model

## Goal

The goal of this document is to propose creating a common model for RESO events.

## Background and Motivation

There are many different event-based models in use in the RESO ecosystem currently, with additional ones being proposed as time goes on.

Having many different ways of doing the same thing, each with their own terminology and methodologies, causes duplication of work and reduces extensibility when new event-based models are added to the domain. There is a software development principle called DRY, or Don't Repeat Yourself, which applies here. Beyond increasing complexity and time to market, using different underlying models also causes decoherence in the event model, which is discussed in a later section.

To address this, the recommendation is that a separation of concerns be made between the *underlying event data structure* and *data and metadata that accompany a given event.*

## Precedent

Using a common data structure for events is not a novel concept. There are existing specifications which share this goal as well. Perhaps it even makes sense to consider using existing work for this purpose rather than creating our own standards.

Even if we decide to proceed with our own model, the fact that the CloudEvents project exists and is supported by several large cloud-computing platforms speaks to the fact that having a common event model is not only feasible, but a reasonable design choice when dealing with many different kinds of events.

If we opt to create our own solution, maybe there are principles from prior work that can be reused, as it's nice to avoid reinventing the wheel.

# Event-Driven Systems

## What Are Events?

An *Event* is the fact that something occurred in a given system.

Events are often accompanied by a change in state to one or more entities in that system, which may be processed by subscribers of those events.

Some attributes are common to all events, such as the time at which an event occurred or its source, while others are specific to a given class or type of event.

Event-driven systems use these facts to take *actions* based on the particular type of event that occurred.

## Event-Driven Systems

Event-driven systems function by processing events and their related data, called *streams*, as they occur rather than in batches of static data obtained from a database or API.

Stream processing is not only good for things like data replication, as it can be used to rehydrate or replay a database at any given point in time from historical logs, but has many advantages where analytics are concerned since *stream processors* can be used to interpret event data changes during ingestion that are often lost or unavailable during batch processing.

Another use for streaming data is in anomaly detection, as state changes may be audited on a granular level between any two points in time and alerts triggered on unexpected patterns as they are encountered based on historical trends.

## Events and Stream Coherence

The lack of a common event model in the RESO ecosystem means that different code has to be written to handle many different kinds of events in different ways, which makes it difficult to construct a common narrative from the event stream for a given entity.

At present, Consumers must fetch data from several disparate sources and try and stitch them together on their own rather than the narrative being told in a simple manner by the party responsible for those data: the Producer.

Improving coherence where events are concerned not only leads to increased productivity and software reuse, but allows for greater insight to be gained by event-based systems.

## Life Cycle of RESO Domain Objects

The RESO domain model consists of objects such as Properties (Listings), Members, Offices, Media, and other Data Dictionary resources. Each of these entities have their own event life cycles which have significance in relation to the type of object being acted upon.

For example, Listings go through a Transaction Management process before they're marketed on the MLS, at which point they generate InternetTracking events that signal interest (or lack thereof), until they go off market in some way, sold or otherwise, at which point there is more TM work to be done. During most of this process, HistoryTransactional events occur on the Listing as well, such as price changes, which potentially change the number of InternetTracking events that occur.

In general, the following information is usually captured when an event occurs:

- Who - who or what caused the event.
- What - the item the event applies to.
- Where - where or in which system the event occurred.
- Why - why the event occurred, or event type.
- When - the timestamp of the event.
- How - this would be the resulting state change and any data that accompanied the event.

This is a bit overly simplistic in some ways, but it paints a basic picture that's easy to understand.

Sample Lifecycle

| Who | What | Where | Why | When | How |
|---|---|---|---|---|---|
| Appraiser | Property | Appraiser System | TM: Appraised | 2020-08-01T21:37:58+00:00 | Data: Link to Report |
| TM Coordinator | Property | TM System | TM: Listing Agreement Signed | 2020-08-02T11:37:58+00:00 | Data: Link to Contracts and Documents |
| Member | Property | MLS | HT: Property Listed | 2020-08-03T01:37:58+00:00 | Data: Listing Record |
| Member | Property | MLS | HT: Price Changed | 2020-08-10T11:37:58+00:00 | Data: Price Change Info |
| Consumer | Property | Portal Vendor | IT: Listing Viewed | 2020-08-10T01:37:58+00:00 | Data: Tracking Info (number of views) |
| Consumer | Property | Portal Vendor | IT: Listing Viewed | 2020-08-11T13:37:58+00:00 | Data: Tracking info (number of views) |
| Member | Property | Showing System | TM: Property Shown | 2020-08-11T11:37:58+00:00 | Data: Showing Info |
| Member | Property | Lockbox System | TM: Lockbox Opened | 2020-08-11T14:37:58+00:00 | Data: Lockbox Info |
| TM Coordinator | Property | TM System | TM: Contract Signed | 2020-08-11T22:37:58+00:00 | Data: Link to Contracts and Documents |
| Member | Property | MLS | HT: Sold | 2020-08-12T23:37:58+00:00 | Data: Final Listing Data |

Consider a property that's been listed at too high of a price: the contract is amended and the price is lowered, which hopefully leads to an increase in Internet Tracking views, which leads to an increase in the number of showings, and then hopefully a sale.

It's much easier to construct a narrative of the life cycle of the listing if all of these events are in a single stream rather than split across many different resources that may be difficult or impossible to relate currently. Having event data in a common format facilitates this analysis, and homogeneous data are easier to ingest into machine learning platforms for further analysis as well.

Other entities such as Members and Offices have their own life cycle events, such as continuing education or changes in membership. Having a single narrative in those cases is also helpful,

since the events that relate to Members and Offices don't really exist on their own but rather accompany the entity they're associated with.

# Equivalence of Current RESO Event Models

To demonstrate equivalence of current RESO models, we show that the various models can be represented by a single event model.

Since DL's event structure is more generic than HistoryTransactional and InternetTracking, it's the easiest place to start.

1) Show we can model InternetTracking events using the EventModel.

Let's say that a tracking event with the following data occurred:

**GET /InternetTracking('12345')?$expand=EventSourceSystem**

```
{
  EventKey: "12345",
  ObjectID: "42",
  ObjectIdType: "ListingKey",
  EventSourceSystemId: "100",
  EventSourceSystem: {
    OrganizationalUniqueId: "100",
    OrganizationType: "MLS",
    ...
  },
  ActorType: "Client",
  EventType: "Favorited",
  EventTimestamp: "2020-07-31T20:40:11+00:00",
  EventOriginatingSystemID: "200"
}
```

This is equivalent to the following EventModel record (with some minor enum additions):

**GET /EventModel('12345')**

```
{
  TransactionId: "12345",
  EventSubject: "42",
  SubjectType: "Property",
  SystemType: "MLS",
  Entity: "Client",
  Event: "Favorited",
```

```
  State: "Recorded",
  Recorder: "100",
  Timestamp: "2020-07-31T20:40:11+00:00",
  Application: "200"
}
```

2) Show we can model HistoryTransactional events using the EventModel.

Let's say a history event with the following data occurred:

**GET /HistoryTransactional('111')?$expand=OriginatingSystem,ChangedByMember**

```
{
  HistoryTransactionalKey: "111",
  ResourceRecordKey: "10",
  SubjectType: "Property",
  OriginatingSystemID: "100",
  OriginatingSystem: {
    OrganizationalUniqueId: "100",
    OrganizationType: "MLS",
    ...,
    OrganizationMlsVendorOuid: "200"
  },
  ChangedByMemberKey: "100",
  ChangedByMember: {
    MemberKey: "100",
    MemberType: "Assistant",
  },
  ChangeType: "Status Change",
  ModificationTimestamp: "2020-07-31T20:40:11+00:00"
}
```

Equivalent Event Model record (with some enums added):

**GET /EventModel('111')**

```
{
  TransactionId: "111",
  EventSubject: "10",
  SubjectType: "Property",
  System: "MLS",
  Entity: "Assistant",
  Event: "Status Change",
  State: "Recorded",
  Recorder: "100",
  Timestamp: "2020-07-31T20:40:11+00:00",
```

```
  Application: "200"
}
```

You may have noticed something missing in the previous example…

What about HistoryTransactional change data? Each HistoryTransactional Event usually has a set of fields and values that have changed. Before we show how change sets are represented, it's important to note that there is actually a design issue with HistoryTransactional at the moment.

Generally, if an Assistant changed 10 fields in a given Listing and saved the work, that change set would be wrapped in a single transaction, with its own TransactionID that would be associated with all fields that had changed during the process. However, that's not the case currently. Each change gets its own row and unique key, but the underlying TransactionID for the overall change set doesn't currently exist. This causes potential issues with transactional consistency, as the information about the original transaction is lost. We can address this in a straightforward manner.

The EventModel offers the "ContextSet," which is a key/value pair of data that can be associated with a single event. In the case of HistoryTransactional, it's appropriate that this would be the change data for the given record.

In the previous example, the status changed for the given Listing record. One would also assume the ContractStatusChangeDate of the record was updated as well.

Using the ContextSet, change data may represented as follows (again with some enum additions):

```
GET /EventModel('111')?$expand=ContextSet
```

```
{
  TransactionId: "111",
  EventSubject: "10",
  SubjectType: "Property",
  System: "MLS",
  Entity: "Assistant",
  Event: "Status Change",
  State: "Recorded",
  Recorder: "100",
  Timestamp: "2020-07-31T20:40:11+00:00",
  Application: "200",
  ContextSet: [
    {Key: "StandardStatus", Value: "Sold"},
    {Key: "ContractStatusChangeDate", Value: "2020-07-31"}
```

```
  ]
}
```

Similar to cases (1) and (2) above, we could also show the HistoryTransactional and InternetTracking representation of these events. However it's implied by the fact we were able to convert both of these models to the EventModel proposed by DL with no loss of information.

# Transitioning to a Common Event Model

Before recommendations can be made regarding transitioning to a common event model, we need to decide whether we're going to take this step.

To restate the goal, we want to consider adopting a common underlying event data structure, while using enumerations and additional event data to differentiate between different classes and types of events. It would seem feasibility has been demonstrated at this point.

There's also a question of whether we want to use the CloudEvents model rather than coming up with one of our own.

An example of the previous events in CloudEvents format would be as follows:

```
{
    "id" : "111",
    "specversion" : "1.5.0",
    "type" : "org.reso.events.Property.StatusChange",
    "source" : "https://api.reso.org/Property/10",
    "time" : "2020-07-31T20:40:11+00:00",
    "eventsubject" : "10",
    "subjecttype" : "Property",
    "system": "MLS",
    "entity": "Assistant",
    "event": "Status Change",
    "state": "Recorded",
    "datacontenttype" : "application/json",
    "data" : {
        "StandardStatus" : "Sold",
        "ContractStatusChangeDate" : "2020-07-31"
    }
}
```

Note how closely this resembles what we arrived at in our previous payload:

**GET /EventModel('111')?$expand=ContextSet**

```
{
  TransactionId: "111",
  EventSubject: "10",
  SubjectType: "Property",
  System: "MLS",
  Entity: "Assistant",
  Event: "Status Change",
  State: "Recorded",
  Recorder: "100",
  Timestamp: "2020-07-31T20:40:11+00:00",
  Application: "200",
  ContextSet: [
    {Key: "StandardStatus", Value: "Sold"},
    {Key: "ContractStatusChangeDate", Value: "2020-07-31"}
  ]
}
```

Even if we don't use the CloudEvents specification right away, it might make sense to look into it as a candidate for an Endorsement at some point. We would essentially get interoperability with many of the leading cloud providers as well as tools that could be used out of the box to Produce and Consume events by doing so.