Type and resource identifiers

Joop Ringelberg 25-08-21, 20-07-22 Version: 2

Introduction

Dealing with a Perspectives model and instances involves many things that must be *identified*, for example:

- Models
- Types (contexts, roles)
- Instances (contexts, roles)
- Storage locations (stores)

The Perspectives Distributed Runtime must be able, too, to *find* all physical resources that represent these models, types, instances and stores. In this text I explain how we build identifiers, when identifiers are locations and how we find resources whose identification is not a location.

Furthermore, we have to deal with the fact that things change. Models evolve, meaning that type definitions may change and sometimes even their names might change. The physical location of a store may change, too. This of course is complicated because the Perspectives Universe is full of things that *refer* to each other – by identifier. Change introduces the concept of *versioning*.

Finally, there is the dimension of *context of usage* in which an identifier must, well, identify. For some identifiers we can firmly establish that they will never leave a particular context of use: this holds for the identifier of a type in a model, for example. Others, however, may be used worldwide and should be globally unique.

The text does not begin with a full, final definition of the shape of identifiers but works towards them gradually, to build understanding of the parts and their function.

Some definitions

A **model** is a collection of types.

A **namespace** is an identifier, used to qualify types in a collection. Hence a model identifier is a namespace, but so is a context type identifier, for its roles.

A **model text** is a readable text file that defines a model. The PDR can transform it into a domeinfile.

A **domeinfile** is a resource (a json file) that holds the types in a model in a form that is used without modification by the PDR to reflect on types.

An **instance** of a context or role is represented by a resource (json file).

A **store** holds a collection of resources. Perspectives distinguishes domeinfile stores from instance stores.

A **resource** is a file.

On Uniform Resource Identifiers

We make use of the definitions given in rfc3986, *Uniform Resource Identifier (URI): Generic Syntax*. Briefly: a URI consists of a *scheme name*, separated by a colon from an identifier that is constructed according to the scheme. The http and https schemes are well-known examples. The URIs defined by these schemes may be classified as *locators* and hence can be called a Uniform Resource Locator (URL).

In the text below, we define three custom schemes for use in Perspectives.

Model identification

As stated above, a model is a collection of types. These types are organised hierarchically. Context (types) contain context types and many others, role (types) contain property types, etc. The hierarchy begins with a context type that has exactly the same identifier as the model itself.

Models can be authored by anyone, all over the world. Models can also be *used* by anyone, in any combination. This requires model identifiers to be globally unique. We do not assume there will be a dedicated global register of models, hence an author cannot check whether the model name he intends to use is unique by comparing it to existing models. Instead, we need to provide a method for constructing a globally unique identifier.

Model names need to be communicable, too, for example because authors may want to advertise their models. This precludes Globally Unique Identifiers (GUIDs) as defined in rfc4122, *A Universally Unique IDentifier (UUID) URN Namespace*. These identifiers are anything but human-readable, are very hard to remember and so do not lend them for human communication.

Instead, we construct a custom URI scheme with the name *model*. rfc4122 states: *Many URI schemes include a hierarchical element for a naming authority so that governance of the name space defined by the remainder of the URI is delegated to that authority*. For Perspectives, we will rely on the domain name system (DNS) as the authority that governs name spaces for the internet, to ensure that model names are unique.

What identifiers can we construct under this scheme? Here is an example of a full URI:

```
model://perspectives.domains/System
```

`model` is the scheme name, separated with a colon (as prescribed) from the identifier constructed under the scheme. The authority has to be delimited up front by two forward

slashes and at the end by another forward slash[1]. In the model scheme, the authority itself has no user- or port information. Following the authority is the local name for the model. This name must start with an upper case character and must be unique in the domain identified by the authority to make the URI unique.

Referring back to rfc3986 we define this syntax for the identifiers under the model scheme:

```
Identifier = "//" reg-name "/" segment-nz-nc
```

A reg-name (registered name) intended for lookup in the DNS uses the syntax defined in Section 3.5 of RFC1034 and Section 2.1 of RFC1123. A segment-nz-nc is a non-zero-length segment without any colon ":". We add the constraint that it must start on an upper case character.

Our model scheme has the advantage that it can be easily mapped onto a URL. As a matter of fact, the identifiers constructed under the model scheme are a subset of those that can be constructed under the https scheme (after substituting *model* by *https*). This means we can locate a unique resource by means of a model identifier (more on this below).

Mapping URIs to URLs

We use a simple mapping from URI to URL. The authority (a reg-name) is used twice:

- Once as the authority of an URL in the https scheme;
- Once as part of a single step in the path of that URL. That single step is prefixed with “models_” and all dots are replaced by underscores.

For example:

```
model://perspectives.domains/System
```

maps to:

```
https://perspectives.domains/models_perspectives_domains/System.json
```

Another example:

```
model://social.perspectives.domains/System
```

maps to:

```
https://perspectives.domains/models_social_perspectives_domains/System.json
```

In this way, we can have multiple hierarchical levels in our model namespaces, while always mapping them to a location in a simple domain[2].

[1] In identifiers in the https scheme it may also be delimited by a question mark and various other means. However, not so in the model scheme.
[2] By which we mean: a top level domain, such as “com” and one subdomain (such as “google”).

Model stores

The Perspectives Distributed Runtime uses Pouchdb, that relies on the conventions of Couchdb, to access resources. For our purposes, this means that a webserver must map URLs of the form `https://perspectives.domains/models_social_perspectives_domains/System.json` to a database in a Couchdb installation (for example, a local installation). It is up to the webserver to provide that mapping[3]. However, we suggest a simple scheme that just uses the first step of the path as the database name. In this case, the json resource might be retrieved from Couchdb using this string: `models_social_perspectives_domains/System.json`. That is, we want the resource `System.json` in the database `models_social`[4].

Storage service providers

Suppose Perspect BV io would provide storage services to third parties, how would it handle them? It could offer an author like me to use a subspace of their namespace `perspectives.domains`, e.g. `joopringelberg.perspectives.domains`. I could then create a model with this name: `model://joopringelberg.perspectives.domains/JoopsModel`. This would map to the following url: `https://perspectives.domains/models_joopringelberg_perspectives_domains/JoopsModel.json`. Their server would consequently look for the resource `JoopsModel.json` in the database `models_joopringelberg_perspectives_domains`.

While perfectly usable, I'd have to rename my model if I wanted to move it to a different provider, because I'd have tied my namespace to theirs (it is a subspace of theirs). That would be very impractical.

It so happens that I own the domain name `joopringelberg.nl`. Suppose I created a model named `model://joopringelberg.nl/JoopsModel` (notice the `.nl` part!), the PDR would map it to:

```
https://joopringelberg.nl/models_joopringelberg_nl/JoopsModel.json
```

My server does not host a Couchdb. However, I could redirect[5] that to:

```
https://perspectives.domains/models_joopringelberg_nl/JoopsModel.json
```

The perspectives.domains server would then request the resource `JoopsModel.json` from the database `models_joopringelberg_nl`. All is well!

Can I have subspaces in my namespace? Yes:

---

[3] But there are three restrictions. See the text *Mapping Model Identifiers to Storage Locations*.
[4] According to its documentation, Couchdb allows forward slashes in its database names. In the practice of version 3.1.0 this runs into problems. Hence we replace slashes by underscores.
[5] See the last chapter on Cross Origin Resource Sharing.

```
model://professional.joopringelberg.nl/JoopsModel
```

maps to

```
https://joopringelberg.nl/models_professional_joopringelberg_nl/JoopsModel.json
```

and is forwarded to:

```
https://perspectives.domains/models_professional_joopringelberg_nl/JoopsModel.json
```

and leads the server to request the resource `JoopsModel.json` from the database `models_professional_joopringelberg_nl/joopringelberg/nl`. Again, all is well.

The takeaway is that I could identify my models like this: `model://joopringelberg.nl/JoopsModel`. Identifiers like this would remain valid as long as I own the `joopringelberg.nl` domain, while I could switch storage providers at will.

Model versions

We want to introduce model versioning in Perspectives using semantic versioning. Version numbers defined according to this scheme are: MAJOR.MINOR.PATCH, where each of the three parts are non-negative integers, and MUST NOT contain leading zeroes..

Version numbers will be appended to model identifiers in such a way that

- They are accepted as part of the `segment-nz-nc`;
- They are accepted as part of couchdb document names.

The semantic version number as such (consisting of numbers and ".") can be part of both. The "@" character can be, too, so we extend our definition of the model scheme to the following production:

```
Identifier = "//" reg-name "/" segment-nz-nc "@" version core
version core = numeric_identifier "." numeric_identifier "." <numeric identifier>
```

(see semantic versioning for the production of <numeric identifier>).

The versioned version of our previous example:

```
model://perspectives.domains/System
```

would be, for example:

```
model://perspectives.domains/System@1.0.0
```

and be mapped to the url

```
https://perspectives.domains/models_perspectives_domains/System@1.0.0
```

while the server would map this to the document `System@1.0.0` in the database `models_perspectives_domains`.

Pre-release versions

An author needs to maintain her model and this involves creating versions that are not accessible to the public. Yet, with the mapping from model identifier to storage location, we seem to have precluded this practice.

In order to restore it, we extend the semantic version with *pre-release information*. In short, we require model storage providers to use that information in the mapping of URLs to databases. See the text *Mapping Model Identifiers to Storage Locations* for details.

As a result, the identifier under the model scheme becomes:

```
Identifier = "//" reg-name "/" segment-nz-nc "@" version core [ - <pre-release>]
```

An *optional* pre-release string may be appended to the model name, separated from it by a hyphen.

Model description: a public context

A domeinfile is a resource that holds the machine readable version of a model. However, end users will want to inform themselves about a model before taking it into use. For this we introduce the convention of a *model description*. The description of a model is itself a resource, a context instance to be precise. Its type is defined in `model://perspectives.domains/System`. The instance contains descriptive text, an expanded name, etcetera.

A model description should be accessible to everyone, or at least to everyone subscribing to a model repository (subscription may require a fee). A model description qualifies as a *public context*: it's type defines a Visitor role (see the text *Universal Perspectives*).

Crucially, a public context is the same for everybody[6]: each participant has the same (consulting) perspective. This means that end users can share a single resource representation.

By convention, we will have a model description instance in a location that can be derived from the model URI. We have seen before that

`model://perspectives.domains/System`

maps to:

https://perspectives.domains/models_perspectives_domains/System.json

But we can also map it to:

https://perspectives.domains/cw_perspectives_domains/System.json

and at this location the model description instance is found.

[6] Except for the model author.

Public context stores

A server that manages a `models` database for the domain X should therefore also manage a `cw` database for X, to store public instances in.

For example:

https://perspectives.domains/cw_perspectives_domains/System.json

looks for `System.json` (the `ModelManifest` for `model://perspectives.domains/System`) in the database `cw_perspectives_domains`.

Obviously, like we saw above, it may forward these URLs to another domain, if that is convenient.

DOCUMENT IS ACTUEEL TOT OP DIT PUNT.

Type identifiers

Top level model types, like contexts and roles, have names that are scoped to model namespaces. This means that their name is prefixed with a namespace identifier. For example, the type `PerspectivesSystem` is identified by:

`model://perspectives.domains/System$PerspectivesSystem`

Type versioning

Peers send deltas to a Distributed Runtime, so the installation may update its instances. A delta contains type information. We have to accommodate the situation where a peer might have another version of the model containing the type, than the receiver. Therefore we need to version types, too:

`model://perspectives.domains/System$PerspectivesSystem@1.0.0`

Consider a domeinfile, representing a model at version 1.0.0. Now the author modifies the context `PerspectivesSystem`, but nothing else. This means that just the identifier of `PerspectivesSystem` changes: all other identifiers will retain their previous version.

`model://perspectives.domains/System$PerspectivesSystem@1.1.0`

Obviously, all references to `PerspectivesSystem` in the model will be updated, too (but this does not cause those referring types to have version 1.1.0, too).

This allows for quick checks when a delta comes in to create an instance. All deltas from a peer using model version 1.1.0 will be allowed, only a delta to create an instance of `PerspectivesSystem` will be reason for further analysis[7].

[7] Newer type versions may be downward compatible. For example, a context with an extra role is shape-compatible with instances without that role.

Type renaming

A frequent kind of change is when the author chooses a *new local name* for a type. For example, `PerspectivesSystem` might be renamed to `PSystem`. This is not a structural change and has no consequences whatever, in runtime. Obviously, it does have consequences in model time:

- Existing references to the name in the model text must be updated;
- Existing references to the name *in other model texts* must be updated, too.

However, type renaming does not cause an increase of the semantic version of a model. If there are other reasons to increase the version, renamed types retain their original version.

Nevertheless, instances refer to types by identifier. How can we make that work? How can a type identifier change, while existing instances do not change their reference?

This is because a reference to a type name is not by its visible name (the local name entered by the author, prefixed by namespace), but by a generated local name (prefixed by namespace). The domeinfile contains a table that maps the two to each other.

When a model is first parsed and saved, all local names are replaced by an integer. Integers start with 0 (for the root type, i.e. the namespace itself, the model identifier) and then increasing by 1 for every next type that the parser encounters. For example:

```
model://perspectives.domains/System$PerspectivesSystem@1.0.0
```

is referred to in the domeinfile and in instances with:

```
model://perspectives.domains/System$0@1.0.0
```

If the author modifies PerspectivesSystem to PSystem, he should provide an instruction to the parser:

```
context PSystem [renamed from PerspectivesSystem]
```

After a successful parse and save, he may (but need not) remove the instruction. The parser looks up the old name in its table and replaces it with the new name.

Handling backwards-incompatible changes in instances

Let’s say that an author changes the type of a property from Boolean to Integer. Role instances that have a value for that property are no longer described by the new type. A property change like that needs to be followed by a change of the shape of the value in the instances.

We may construct a scheme of automatic repairs to be carried out on data on the occasion of such model changes. Lacking that, some changes can be carried out automatically to ensure proper functioning, but possibly to the cost of semantics. For example, every type

can be mapped to a String. Booleans may be mapped to Numbers according to some scheme (e.g. 0 for false, 1 for true), etc.

It turns out that very few model changes do actually cause a problem with the shape of the instances (see the text *Model versions and compatibility*).

Imports

A model text imports dependencies in this way:

```
use sys for model://cw.perspectives.domains/System@1.1.0
```

Type names imported from another namespace will be replaced by using the name table of the corresponding domeinfile.

If the author updates the version of an import, the parser MAY compare the name table of the previously used version with the that of the new version, if the author provides an instruction:

```
use sys for model:cw.perspectives.domains/System#1.15.0 [up from 1.11.0]
```

Imported names must be fully qualified (either written in full, or with a prefix). Hence the parser can scan the model text for names that are replaced in the import (using the prefix, if applicable) and replace them automatically in the text.

The next parse is then guaranteed to be able to replace the each imported identifier by its number.

The model text may refer to types that have been dropped in the new version of the import. The parser MAY report these to the author.

Instance identifiers

DOCUMENT IS ACTUEEL VANAF DIT PUNT.

We introduce two more custom schemes for instances of contexts and roles, respectively:

```
context:{GUID}
```

is an URI identifying a context, while

```
role:{GUID}
```

identifies a role.

Stores

With the introduction of public contexts comes the notion of multiple stores for instances. Stores will be managed in a particular installation using a Perspectives model, that

enables the user to associate a symbolic name with a particular storage endpoint. This association is unique to each user (each user can have her own mapping).

Multiple instance stores means we have to decide, for any instance identifier, from which store to fetch it. We make this possible by having instance identifiers contain a reference to their store. To that end, we append the symbolic store name (a simple string) to the URI, separating it from the URI by a character that cannot be part of the GUID (we assume the pipe character here):

```
context:{GUID}|MYOTHERSTORE
```

If no suffix is appended, the default local store is assumed. We call such identifiers Local Resource Identifiers, or LRI.

This means, however, that one user might have another identifier for an instance than another, as it is the end user's prerogative to decide where to store his instances[8]. We cannot, therefore, consider such an identifier to be an URI (it is not universal!).

Luckily this is not a problem, because resources representing an instance are unique for a single user (various installations for the same user must, of course, use the same LRIs!). Resources are never shared as such (except for public resources, see below).

However, peers communicate deltas that refer to instances. From the above we learn that we cannot use the LRI to construct a delta. Instead, we stick to the two schemes introduced above to identify resources in deltas. A peer receiving a delta to, say, create a context, must use type reflection to find out in which store to put it and will extend the received URI with the store's symbolic name.

Public instance identification

Not all instances are private: some are public. We've seen above we identify them with URLs in a database whose name begins with `cw_`. It is, therefore, simple to distinguish private from public instances. They are recognisable by their scheme:

- `context:{GUID}` is a private context instance in the default private store;
- `context:{GUID}|MYOTHERSTORE` is a private context instance in another private store;
- `https://perspectives.domains/cw_perspectives_domains/OurModels.json` is a public context instance that can be found at the given URL.

Determining the location of a public context: the default

The modeller can use the symbolic store names in a model text, to instruct the PDR to create public context identifiers at a specific location (remember that a public context is

[8] Moreover, the same store name might mean a different location for various users. In other words, store names are *indexed*.

identified by an URL). Symbolic store names can be mapped onto concrete locations (i.e. couchdb databases) using screens generated from a yet-to-be-constructed model. For instance:

```
case ModelManifest public NAMESPACESTORE
```

directs the PDR to create an instance of `ModelManifest` in whatever location its user has associated with the symbolic name `NAMESPACESTORE`.

Actually, `NAMESPACESTORE` is a special case. It is mapped to the location of public contexts associated with the namespace of the type. Casu quo: `sys:ModelManifest` expands to `model://perspectives.domains/System$ModelManifest` and that, in turn is mapped onto https://perspectives.domains/cw_perspectives_domains.

Non-default locations of public contexts

While that is perfectly usable for models in the perspectives.domains namespace[9], it does not extend to authors that create models in other namespaces. For example, we expect to be able to fetch the `ModelManifest` of `model://joopringelberg.nl/JoopsModel` with the URL https://joopringelberg.nl/cw_joopringelberg_nl/JoopsModel.json.

We handle this with an action that constructs the `ModelManifest` context and provides a calculated name for it, like so:

```
user User (mandatory)
  ...
  perspective on ModelManifests
    in state ReadyToMake
      action CreateModel
        create_ context ModelManifest (Namespace + "/" + ModelName + ".json") bound to origin
...
context ModelManifests filledBy sys:ModelManifest
  -- e.g. "JoopsModel"
  property ModelName (String)
  -- e.g. "model://joopringelberg.nl"
  property Namespace (String)

    state ReadyToMake = (exists ModelName) and (exists Namespace) and not exists binding
```

The screen that is generated from these model fragments allows the author to enter a namespace - e.g. `model://joopringelberg.nl` - and a model name - e.g. `JoopsModel` -

[9] Generally authored by the Perspectives organization.

and then run an action that combines the two strings into the URL identifying the `ModelManifest` instance.

A similar case exists for instances of `Repository`. A The author must be able to specify an arbitrary URL for a repository that she creates[10].

The default repository

An InPlace installation cannot function without the system model. Moreover, every installation needs access to certain basic models (such as `model://perspectives.domains/BrokerServices`). These models and their manifests are stored in databases on the server https://perspectives.domains. This model database is described by an instance of Repository that is identified by (and located in) `https://perspectives.domains/cw_perspectives_domains/BaseRepository.json`. Fetching the system model is hardwired into the PDR.

Part of the installation routine is to create an instance of `sys:PerspectivesSystem`. This instance is created complete with an instance of the role `sys:PerspectivesSystem$Repositories`, that is filled with this public repository[11].

DOCUMENT IS ACTUEEL TOT OP DIT PUNT.

Cross Origin Resource Sharing

The 'same origin policy' implies that a script is allowed to request resources just from the same domain it itself is served[12]. The PDR is served from https://inplace.works. This would imply that the PDR could only request models (and other resources) from that same domain. It would preclude the repositories at arbitrary locations as described in this document.

However, a server may be configured such that it sends CORS headers. Couchdb supports such configuration. Part of that configuration is to declare a set of *origin domains*. In our case, that would be https://inplace.works.

Every hosting party that supplies a Couchdb server for repositories, should therefore configure CORS in the same way. As a consequence, PDR sources, served from https://inplace.works, are allowed to see resources served from such servers.

[10] There is a subtlety involved here: she needs to store the context that *describes* a repository in some location that she has rights to write to; and she must register *in that description* the location of a database that actually functions as a repository of model files.
[11] This role should be computed by fetching the instances of Repositories from the (local) database; but then this should be an Aspect role that can be reused in model:CouchdbManagement.
[12] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Redirection

In paragraph *Model stores* we suggest that if a domain is hosted by party A, while the repository where models in that domain are stored is hosted by party B on another domain, party A *redirects* requests to party B's domain. However, CORS does not always allow this[13].

This holds especially for so called 'pre-flight requests' (made with the OPTION verb). From MDN:

> *Not all browsers currently support following redirects after a preflighted request. (...) The CORS protocol originally required that behavior but was subsequently changed to no longer require it. However, not all browsers have implemented the change, and thus still exhibit the originally required behavior.*[14]

In contrast, redirect is always allowed on *simple requests*. The PDR requests models in a way that seems to satisfy the criteria for such simple requests, excepting that the content-type header is application/json (which is not allowed). Nevertheless, in Chrome (version 100.0.4896.88) no pre-flight request seems to be done.

The redirecting party should implement CORS for inplace.works, too.

We implement the PDR on the assumption that browsers allow redirection on our CORS requests.

An example redirection directive for Apache, for example to be used in the `perspectives.domains` configuration file:

```
RedirectMatch permanent "^/models(.*)$" https://inplace.works/models$1
```

A similar effect (but without redirect HTTP status code) can be achieved by a reverse proxy:

```
ProxyPassMatch "^/models(.*)$" https://inplace.works/models$1
```

Observations

On the local version of `perspectives.domains`, we observe that

- the redirection fails because of the preflight problem with CORS[15];

[13] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors/CORSExternalRedirectNotAllowed
[14] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#simple_requests
[15] The preflight request cannot be observed in Chrome.

- the reverse proxy works, but only if the database is public (i.e. if no members or admins are defined.

Obviously, the PDR does not send credentials for the reverse proxy (`inplace.works`) with the request for the domain (`perspectives.domains`). So, while the reverse proxy works, no credentials are sent along with it.

While *retrieving* models without credentials might be ok, uploading models certainly needs credentials. This is a problem to be solved.

HTTPS and certificates

All domains should be approached using the https scheme. This holds for domains that redirect, too. So, in our example, the server that redirects from joopringelberg.nl should have a certificate for that domain.