User and System Identifier

Joop Ringelberg 14-04-20 Version: 1

Introduction

`sys:MySystem` and `sys:Me` are replaced by unique 'private' names, like all other indexed names, but we handle them a little different. This is because we have to generate a unique name for a particular PDR installation *before* the functions are executed that replace all other indexed names. This text explains why and how. It also explains the relation between these two, the user database names and some facilities for testing.

The origin of the system identifier

When InPlace is first fired up on a computer, Couchdb is supposed to be in PartyMode. The user, having no account, enters a username and a password of her choice. The PDR then calls the function `setupCouchdbForFirstUser`. This function will, eventually, generate a guid that will be the base of the system identifier. However, for now, during development, for easy testing, we just use the user identifier that the user just typed in. In this text we call it the `systemIdentifier`.

We also have a function `setupCouchdbForAnotherUser`. Like for the first user, it will in the end generate a guid but now just uses the user name provided to one of its parameters.

Persistence of system identifier, user name and password

We put the user name and password and the system identifier into a data structure `CouchdbUser`:

```
newtype CouchdbUser = CouchdbUser UserInfo

type UserInfo =
  { userName :: UserName
  , couchdbPassword :: String
  , couchdbHost :: String
  , couchdbPort :: Int
  , systemIdentifier :: String
  , _rev :: Maybe String
  }
```

This structure is serialised and stored as a file in Couchdb in the database `localusers`. On logging in, the PDR fetches the document with the name entered by the user[1] and checks the password. If all works out, the PDR starts up with the above data structure as part of `PerspectivesState`.

System identifier as base name

From the systemIdentifier we construct replacements for both `sys:MySystem` and `sys:Me`:

Purescript function	Indexed name	Private name
getMySystem	sys:MySystem	model:User$<systemIdentifier>
getUserIdentifier	sys:Me	model:User$<systemIdentifier>$User

These values are constructed by the two functions given in the first column. They take the value of `systemIdentifier` out of `PerspectivesState`.

User database names are derived from `systemIdentifier`, too:

- `<systemIdentifier>_instances`
- `<systemIdentifier>_models`;
- `<systemIdentifier>_post`.

These databases are constructed by the functions `setupCouchdbForFirstUser` and `setupCouchdbForAnotherUser`.

Putting the systemIdentifier in PerspectivesState

When the PDR fires up, it constructs a `PerspectivesState` that holds, among others, the `systemIdentifier` (as we've shown in *Persistence of system identifier, user name and password*). It then calls `runPerspectivesWithState` on that state and some value in MonadPerspectives to compute.

However, there is another function, `runPerspectives`, that takes, among others an argument bound to its parameter `systemId`, that constructs an instance of PerspectivesState on the fly and then computes a value in MonadPerspectives. So while computing that value, for `systemIdentifier` we have that argument. This means that during that computation

- Models and instances and transactions are taken from and written into a specific set of user databases, based on that argument value;
- `sys:Me` and `sys:MySystem` are replaced by values based on that argument value.

[1] It uses the special account `authenticator` to do so. The password for this account is kept in the source code. This is not particularly safe, but remember the database resides on the user's own machine.

This is very useful for testing. We can run one computation for one user, then another for another user, all in the same test code!

Testing

Accounts

For testing purposes, we have three local user accounts whose names are respectively “test”, “cor” and “joop”. Their system identifier equals their user names. We have for each three user databases and we do not remove them between tests. However, tests may/must clear them out.

Running a test for an account

We can run a test for a specific user by applying the functions `runP` (for “test”), `runPCor` (for “cor”) and `runPJoop` (for “joop”).

```
test "something" (runP $do
     something)
```

Providing a required model and instances

A test that needs instances of `model:PerspectivesSystem` and its `User` role, should apply the function `withModel`. For example:

```
test "something" (runP $ withModel (DomeinFileId "model:System") do
     something)
```

will compute `something` for user “test”, with instances taken from `test_models`, in the presence of the types of `model:System`. `withModel` clears both the instances database and removes the model (but, alas, not its dependencies!).

Loading a CRL file

Use the function `loadAndCacheCrlFile` to create instances from file and put them in cache (but not in Couchdb). This function replaces all (expanded) indexed names in the source code prior to parsing it.

To store the instances in Couchdb, use `loadAndSaveCrlFile`. However, note that this function also updates queries and triggers bot rules.