State change: cache, save, sync

Joop Ringelberg 28-11-19 Version: 1

Introduction

A computation in Perspectives proceeds in part by changing state (the other part is: consulting state). State changes in two ways:

- by creating Context and Role instances (always attaching the latter to the former)
- and by changing them.

Changes are limited to:

- adding role instances to contexts, or removing them
- binding or unbinding roles;
- changing property values.

State is saved in Couchdb. State is also partly saved in memory: this we call the *cache*.

Because multiple users have access to the same contexts, roles and property values (though two users *never* share exactly the same contexts and roles), changes must be *synchronised*.

In this text we document the relation between changing state and caching, saving and syncing that state change.

Model files

A model consists of the description of many types. Types are part of the state of a computation. Models are saved as model files. They are cached as well. In fact, there is no real technical difference between a model and a context or role with respect to caching and saving. We do not synchronise model files, however. Models are published on a server. We have a mechanism of *subscription* that allows a PDR to stay up to date w.r.t. models. As publishing is a conscious act by a modeller, model updates are not spread in real time as the modeller changes his model.

Cache, database and synchronisation: mechanisms

We briefly characterise these mechanisms.

Cache

State is cached in `MonadPerspectives`. This is a Monad based on ReaderT and Aff. Through ReaderT we have access to an AVar. This AVar holds a record with fields. We modify the contents of the AVar in order to change state. This arrangement allows us to modify state asynchronously.

Database

Couchdb stores JSON documents. Each Context or Role instance is serialised as a JSON document. Instances are stored in a database in Couchdb whose name is constructed from the unique string that identifies a user, thus allowing a single Couchdb instance to host data for many users. An important facet of Couchdb is its synchronisation mechanism. This in turn depends on versioning. In order to be able to change a document in Couchdb, one must send the current *revision string* along to the http request. To speed up the process, we keep the revision string in cache for documents we've retrieved from Couchdb before.

Synchronisation

Synchronisation is a two-step process:

1. the PDR of a user who changed state compiles a *Transaction* of such changes and sends them to relevant other users (in fact, each user receives a Transaction tailored to his perspectives);
2. the PDR of these other users processes the Transaction. This involves checking whether the sender was authorised to perform the changes. If so, it loads the changed entities in cache (if they weren't there before) and applies the changes. Finally it saves them to its own Couchdb instance.

The dynamics of state change

Contexts and roles that are created, are immediately cached. However, we do not always immediately save them to Couchdb. This is because sometimes, after initial creation, resources have to be changed. Think of a context instance: we prefer to wait until all its roles have been created, before we save them. This is because we may yet encounter an error while working out the roles. Instead of retracting prematurely saved roles, we wait with saving them until the entire process has been finished without errors.

For similar reasons we do not immediately dispatch a Transaction as soon as a change has been cached or saved.

Creating and modifying resources

There are no more than two modules where instances are created:

- Perspectives.BasicConstructors, with the functions
  - `constructContext` and
  - `constructAnotherRol`
- Perspectives.LoadCRL

There is a single module through whose functions all changes to resources run:

- Perspectives.Assignment.Update

In turn, all these functions are used in two distinct places:

- Perspectives.Api
- Perspectives.Actions

The first module channels resource creation and modification that are the initiative of the end user. The second module realises the automatic execution of such actions by bots.

The consequence is that all caching, saving and synchronising starts in the above three modules.

Saving and syncing resources

The module Perspectives.SaveUserData contains five functions to save cached Context- and Role instances and to synchronise them:

- `saveContextInstance` (to used in tandem with `constructContext`)
- `saveRoleInstance` (idem, `constructAnotherRol`)
- `removeContextInstance`
- `removeRoleInstance` (to be used in tandem with `removeRol`)
- `removeAllRoleInstances` (to be used in tandem with `deleteRol` and `setRol`)