Lively Code Database

Seminar Web-based Development Environments

Tilman Giese and Marko Röder

Hasso-Plattner-Institut, Potsdam
{tilman.giese,marko.roeder}@student.hpi.uni-potsdam.de

Abstract. The Lively Kernel is a web-based development environment that is increasingly gaining popularity. Its server-side persistency is currently based on Subversion. As a file-based revision control system, Subversion does not allow for a more fine-granular revision control for JavaScript modules, classes, or methods. This paper presents an alternative persistency layer based on CouchDB that was integrated into the Lively Kernel to allow developers to store JavaScript code objects in a database.

1 Motivation and Goals

With the increasing capabilities and performance of today's web browsers web-based development environments have become popular. The approach to develop applications within the browser without any additional tool is very intriguing. The Lively Kernel is such a development environment that encourages developers to explore this new hands-on way of creating web applications.

The Lively Kernel is currently based on Subversion [1] as its persistency layer. The Subversion repository is directly accessed by the browser to retrieve all necessary files, in particular the XHTML and JavaScript files that contain the actual code to be executed. As Subversion is a file-oriented versioning control system, the versioning granularity in the Lively Kernel is also a file. However, using files as the smallest entities to contain JavaScript code entails several shortcomings. It also has a serious impact on how JavaScript code can be maintained by means of the built-in Lively Kernel source code browsers.

The first and foremost shortcoming of this approach is that JavaScript source code has to be parsed all the time in order to provide a fine-granular view on classes within a module and methods within a class. Syntax errors in the JavaScript file might render the entire file unparseable and thus no classes or methods might be visible in a code browser. By just relying on the means of JavaScript source code, it is also quite difficult to introduce metadata on classes or methods (like documentation or method categories). The current approach requires the developer to follow certain conventions, e.g. by declaring a class property with a particular name or by adding a special comment on top of a method definition.

And further issues arise from files being the entity of versioning control. Every small change to a method will always result in the entire file being saved and

assigned a new Subversion revision number. This can eventually lead to a very huge database. But more importantly, the connection between changes that logically belong together is lost as each change will create a new Subversion revision. Without specific knowledge of how changes were done it is thus impossible to revert back to a consistent state.

The goal of this project was to provide a more fine-granular revision control for JavaScript code objects. A code object is a source code artifact within the Lively Kernel environment that has a semantical notion of its own. A single code object can be composed of other smaller code objects. Examples of such code objects are methods, classes, and modules. The code objects should then be stored in a separate database rather than the Subversion repository. The existing Lively Kernel source code browsers should be extended to read and write code objects without the developer noticing the change in persistency. An interface should be provided to easily access code objects in the database. The JavaScript source code should still be accessible as a file and modules should be loadable from the database in the same way they were previously loaded. Figure 1 summarizes the change in persistency.

Fig. 1: Persistency Change

2 Implementation

The following sections present our solution in more detail and explain how it integrates into the Lively Kernel. The overall architecture is briefly described followed by an overview of the three major parts of the solution: the Code Database API to access the database, the Code Database Browser as the tool to develop applications and the Lively Kernel core extension that allows developers to load code from the database.

2.1 High-Level Architecture

As a preset requirement, CouchDB [2] had been chosen very early as the database technology to store code objects. CouchDB and Subversion have a different notion of revision, therefore the term needs clarification. A code object revision is a particular persistent representation of a code object at a specific point in time. Each code object revision corresponds to a particular CouchDB document. A code

object revision is identified by a code object revision number that is sequentially incremented for each new revision starting with 1 for the first revision. The revisions are collected in a code object revision history. Code object revisions can be flagged as drafts meaning they are not visible to developers unless specifically requested. Listing 1.1 shows the CouchDB document for a class revision. The revision properties are self-explanatory. CouchDB documents are JavaScript objects in JSON form.

Listing 1.1: CouchDB Document: Class Revision

```
{
   "_id": "Revision::25::TestModule::TestClass",
   "_rev": "25-2a4bb26e8b88a5447215e71e942f6870",
   "type": "class",
   "name": "TestClass",
   "documentation": "This is a class for testing purposes",
   "superclass": "Object",
   "methods": [
      "method1", "method2", "method3"
   ],
   "module": "TestModule"
}
```

In order to group several code objects together, there is also the concept of a change set. A set of changes to a set of code objects is called a change set thereby change set implicitly defines a logical unit of work. Persisting a change set in the database will create a change set revision. Similarly to a code object revision history there is also a change set revision history that keeps track of all the change sets. Listing 1.2 shows an extract of the CouchDB document that represents the change set revision history. It stores who committed the change set and which code objects were involved respectively what actions were performed on these code objects.

Listing 1.2: CouchDB Document: Change Set Revision History

```
{
   "_id": "ChangeSetHistory",
   "_rev": "200-c4a81c3b50ac4849e5595554a796e4fb",
   "currentRevision": 58,
   "revisionHistory": [
       ...
       {
           "revision": 36,
           "author": "m.roeder",
           "date": "2010-07-20T14:07:55Z",
           "message": "my commit message",
           "objects": [
               {
                   "name": "TestModule2",
                   "revision": 1,
                   "action": "add"
```

```
                },
                {
                    "name": "TestModule2::TestClass",
                    "revision": 1,
                    "action": "add"
                },
                {
                    "name": "TestModule1",
                    "revision": 2,
                    "action": "update"
                }
            ]
        },
        ...
    ]
}
```

In order to avoid name clashes when storing documents in CouchDB, there are the following conventions for document identifiers:

Listing 1.3: CouchDB Document Identifiers

```
- Revision::{RevNumber}::{ModuleName}[::{ClassName}[::{MethodName}]]
- RevisionHistory::{ModuleName}[::{ClassName}[::{MethodName}]]
- ChangeSetHistory
```

2.2 Code Database API

The Code Database API is the interface to all code objects stored in the database. It reflects the concepts described in the previous section. Figure 2 shows the classes that are involved.

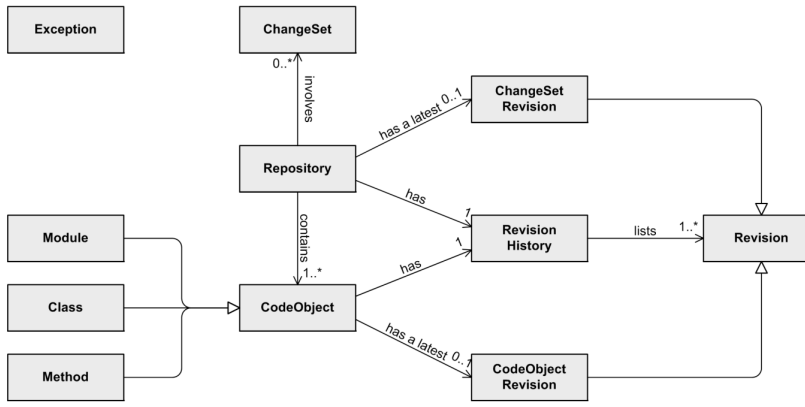

Fig. 2: Code Database API: Class Diagram

The main entry point is the class `Repository`. It provides methods to retrieve code objects and to create a new change set:

Listing 1.4: `Repository` Interface

```
# to retrieve a single code object
getCodeObject(type?, name+, includeDrafts?)
  e.g. getCodeObject(CDB.Module, 'TestModule')
  e.g. getCodeObject('TestModule', 'TestClass', true)

# to list code objects
listCodeObjects(type, name+, includeDrafts?)
  e.g. listCodeObject(CDB.Method, 'TestModule::TestClass')

# to create a new change set
createChangeSet()
```

A typical program flow is depicted in figure 3. First, a reference to the code database repository is created along with a new change set. Afterwards, the database can be queried for code objects using either the `getCodeObject` or `listCodeObjects` method. Before a code object can be modified, it has to be added to the change set. Saving the code object will persist all its properties in the database using a draft revision. Code objects can, of course, be saved several times until the change set is finally committed.

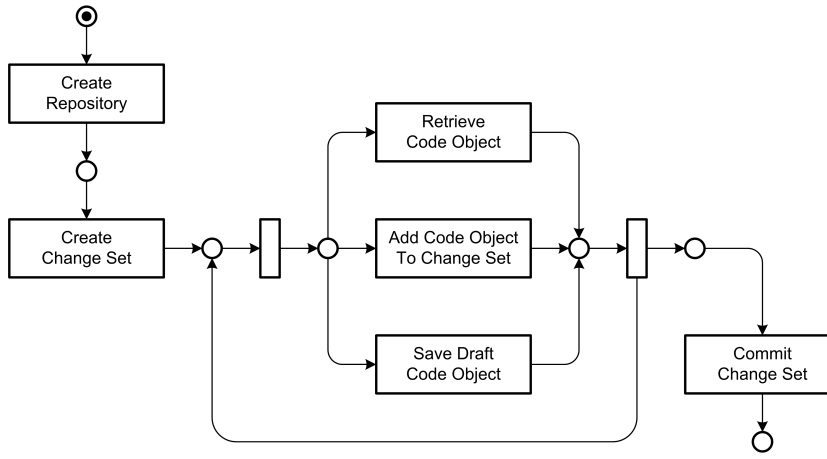

Fig. 3: Typical Program Flow

Draft revisions of code objects are snapshots that do not necessarily have to be in an executable state or consistent with other code objects. Saving a draft revision will create a new CouchDB document for the revision and modify (or create if the object does not exist yet) the CouchDB document for the code object revision history. Editing conflicts with other developers are detected upon

changing the revision history[1]. When the change set is committed, consistency is checked more properly, for example a class also has to be part of the change set if a method was added to it. After all consistency checks have successfully passed, the draft flag is removed from the latest revisions of all code objects and a new change set revision is created. Listing 1.5 shows a small example that adds a new method to an existing class, saves and commits the changes.

Listing 1.5: Example Program

```
var rep = new CDB.Repository();
var cs = rep.createChangeSet();

var klass = rep.getCodeObject(CDB.Klass, 'TestModule', 'TestClass');
var method = new CDB.Method('myNewMethod');
klass.addMethod(method);

method.documentation = 'Put here what the method does';
method.source = 'function() { ... }';

cs.add(klass);
cs.add(method);

klass.save();  // saves the changes as draft
method.save();

cs.commit();   // commits the changes
```

Throughout the Code Database API error conditions are signaled using exceptions. For example, `getCodeObject` will throw an `ObjectNotFoundException` if the specified code object is not in the database. commit can throw a `Consistency Exception` if any of the consistency constraints are not met or, like all database operations, a `DatabaseException` if there is technical problem with the database.

2.3 Code Database Browser

One implementation that uses the Code Database API is the Code Database Browser. It is the tool that lets a developer browse and edit the Lively Kernel source code stored inside a CouchDB database. By giving the user this kind of tool, there is no need to directly interact with the database. Everything that needs to be done – like adding or removing a code object (e.g. a method) and setting its attributes or contents – can be done with the browser. Figure 4 shows the Code Database Browser and names its parts.

To make use of the look and feel of the Lively Kernel browser-style, the Code Database Browser was built on top of already existing classes that are used to show the JavaScript files from the Subversion repository. Using this basic browser

[1] Whenever a CouchDB document is updated the document revision hash (which is a CouchDB identifier) of the previous revision has to be provided. If another revision was added in the meantime, CouchDB can thus detect an update conflict.

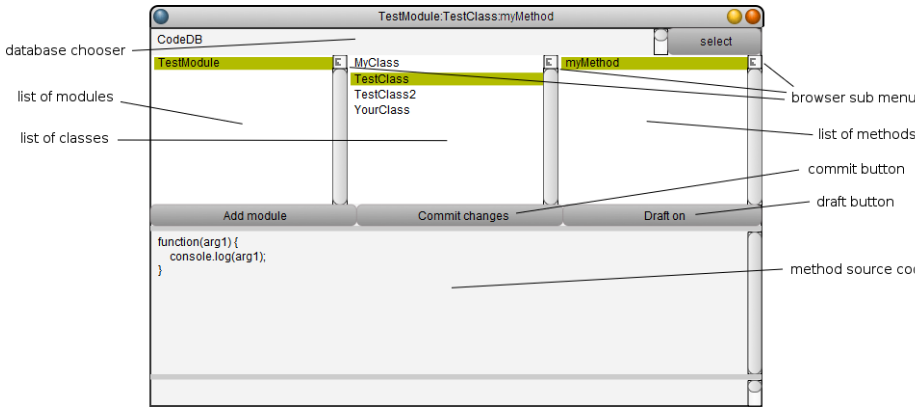

Fig. 4: The Code Database Browser

there is not only a common look and feel but also the same understanding of how to represent and work with each code object. Thus it will be easy to integrate with the default code browser or even replace it in one of the next steps.

A specialty of the Code Database Browser is its differentiation between saving and committing changes. Derived from the two ways code objects can be stored – either as draft or as a commit when semantically grouped – there are also two ways to persist changes. The first one is the default action which is carried out e.g. after changing a method and pressing the keyboard shortcut to save (depending on the operation system that could be CMD + S). This invokes the `save` method on the affected code objects and stores them as drafts inside the database. Having the `draft` status the changes have only been safely persisted but they do not affect the currently loaded and executed code. If the developer wants to activate the changes instead, then the commit button has to be pressed and all the changes done inside the current change set will be committed and activated. Since the developer might not always want to see the changes made as drafts, the draft button switches the browser mode between only displaying activated code and also displaying drafts.

Adding and removing a module, class or method which can be done by the "Add module" button and the browser sub menus automatically creates a draft for the corresponding code object.

2.4 Kernel Extension

In the last two sections we focused on how to manage code objects with the Code Database API and the Code Database Browser. In this section we will explain how the code that is stored inside CouchDB can be loaded into and executed inside the Lively Kernel.

Listing 1.6 shows the structure of a module that can be loaded from the Subversion repository into the Lively Kernel system. The module parameter creates a new module and namespace called `lively.Example` that has dependencies to

`lively.Tools` and `lively.Helper`. Inside that namespace, a class definition creates the new class `SubClass` which is derived from `SuperClass`. The class `SubClass` can have methods like `initialize` and `aMethod` and attributes like `documentation`.

Listing 1.6: Example for the module structure

```
module('lively.Example').requires('lively.Tools', 'lively.Helper').
    toRun(function(example, tools, help) {
  SuperClass.subclass('lively.Example.SubClass', {
    documentation: 'This is a subclass of SuperClass.',

    initialize: function($super) {
      ...
    },

    aMethod: function(arg1, arg2) {
      ...
    },
    ...
  });
  ...
});
```

Our goal was to extend the current core system to load source code from the code database in the same manner. So we introduced a new prefix for all the source code that is loaded from a database. This prefix starts with a $-sign which is followed by the name of the database and a dot. So the prefixed module name for the module of listing 1.6 would look like `$code_db.lively.Example` when loaded from the code database. In this case `code_db` is the database name of a CouchDB database containing Lively Kernel source code.

Furthermore the same style of references can be made inside requirements and when subclassing. This allows a developer of the Lively Kernel to adapt to the new persistency layer more easily.

What is done when the Loader of the Lively Kernel comes across one of the new prefixed module references is that the request of loading the JavaScript file from a URL is modified to not use the current Subversion repository but the CouchDB instance that has been configured. On that CouchDB the selected database is queried for a list (one of CouchDB's querying techniques) that builds a JavaScript file with the same kind of module structure used by the Lively Kernel until today.

3 Discussion

The previous sections introduced the changes we made to the Lively Kernel to make it use a CouchDB database as a (second) persistency layer. Now there shall be a discussion on the decision to use CouchDB as a revision control system (RCS) and the advantages and disadvantage we did discover working on it.

CouchDB itself is one of the new database technologies that arose within the NoSQL movement. Its document-oriented style and the use of JavaScript to define queries (based on the map/reduce algorithm) makes it far more appropriate to store the source code of the Lively Kernel inside it than a relational database. However, its simple key-document storage has also some disadvantages when being used as RCS for object-oriented source code because nowadays every object-oriented programming language has some kind of namespace concept. So the Lively Kernel built upon JavaScript has that too: methods and attributes belong to a class, classes belong to a module and modules have a path-like namespace structure. To support that we had to come up with a mapping of this whole structure to a single key (see section 2.1 for that).

Another important point which was already mentioned is how to manage the revisions. At first we completely wanted to rely on what CouchDB calls a document revision for our code objects. Doing this and storing a code object (e.g. a method) inside only one CouchDB document should lead to code object revisions stored as document revisions. What we did ignore following this approach was that CouchDB revisions are not revisions as in RCSs. Old revisions could for example be removed when compacting a database or omitted making a backup copy to another CouchDB instance. This surely is unwanted when storing source code where one day you might go back in time and restore an old version or use multiple versions of the same file/library in different parts of the system in parallel. So to get fully persistent revisions we ended up with a revisioning system that stores a document for each revision of a code object which was one of two possible ideas [3].

At last we needed a simple and efficient way to reconstruct JavaScript code from the code objects stored inside the CouchDB. This is a point where CouchDB again can show its advantages of document storage. Using map/reduce inside a mixture of views and lists (two of the querying techniques) we were able to construct JavaScript files for modules that look exactly the same as modules that are stored as files inside the Subversion repository. As an alternative to using map/reduce code to construct the JavaScript files, a template-based approach that is also supported by CouchDB could have been taken. But as long as the resulting structure of the JavaScript files is that simple creating a template would just be more overhead. However, in both cases the resulting CouchDB lists can easily be accessed by a URL which by now is the access paradigm for Subversion files too. Therefore no deep changes inside the Lively Kernel had to be done to execute source code that comes out of the CouchDB.

4 Related Work

Like other revision control systems such as ENVY/Developer [4], the Lively Code Database provides the developer with a toolset that is implemented on the core system to help with configuration management and version control. With a similar type of structuring code objects – modules, class and methods – and a browser to develop and maintain these objects the Code Database enables

changes on the method level. Unlike ENVY/Developer our approach does not have component ownership but an author for each revsion.

In contrast to RCS like Subversion [1], GIT [5] or Mercurial [6], our Code Database on top of a CouchDB database does not use files as the finest granularity for source code but instead it breaks it down into modules, classes and methods. All these parts are separatly versioned and kept together by an encapsulating change set. So there are less conflicts if more than one developer is working on the same part of the system.

We are working with revisions similarly to Perforce [7] in terms of letting the server have a database with meta information on the versioned source code (e.g. revision numbers and relations between different revisions) and storing the source code as separate documents. Additionally we have change sets (that are called change lists in Perforce) that group multiple changes on code objects and name the action that is carried out (like `added`, `deleted`, `updated`). However, we have a much finer grained look on code objects as we do not use files to store the source code.

5 Summary and Outlook

With the work done so far, there are three libraries to enable CouchDB as one of the persistency layers that the Lively Kernel can rely on. These three libraries are: the Code Database API which is based on the simple JavaScript API to interact with a CouchDB instance, the Code Database Browser which was created on top of the Code Database API and the small library of core enhancements of the Lively Kernel to make CouchDB databases a valid source to load and execute source code from.

On the CouchDB side there is only one design document that contains all the map/reduce functions to let the core extension and the Code Database API query the database. And this document can easily be installed on a new CouchDB database by simply pointing the Code Database API or the Code Database Browser to the database URL and instructing it to "livelyfy" that database.

To conclude by now there is a transparent replacement of the current persistency from the perspective of the source code and revision management.

Nevertheless there are still some points missing that need further work. One of it is that the entire Lively Kernel source code has to be imported into one database. Doing this there also has to be done some clean up and extension work since the current JavaScript code used inside the Lively Kernel does not only consist of modules, classes and methods/attributes but of some plain JavaScript code to create the base of the system (like the namespaces etc.). Furthermore there are not only JavaScript files that make the Lively Kernel but also XHTML files containing all the elements inside a "world". Either there has to be a way to convert that XHTML files to CouchDB documents too or these files have to stay inside a parallel Subversion repository.

As final ideas of what might be coming next, there still is some work to do on transactions and a better conflict resolution when saving source code to the code database. Also the previously discussed option to use different revisions of the same module or class and the tagging of a revision as version might add a great flexibility to the Lively Kernel.

References

1. Pilato, M.: Version Control With Subversion. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2004)
2. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide Time to Relax. O'Reilly Media, Inc. (2010)
3. Anderson, J.C.: Simple document versioning with couchdb. http://blog.couch.io/post/632718824/simple-document-versioning-with-couchdb (2010)
4. Pelrine, J., Knight, A., Cho, A.: Mastering ENVY/Developer. Cambridge University Press, Camebridge, United Kingdom (2001)
5. Loeliger, J.: Version Control with Git. O'Reilly Media, Inc., Sebastopol, CA, USA (2009)
6. O'Sullivan, B.: Mercurial: The Definitive Guide. O'Reilly Media, Inc., Sebastopol, CA, USA (2009)
7. Wingerd, L.: Practical Perforce. O'Reilly Media, Inc., Sebastopol, CA, USA (2006)