Mozilla Addon Builder
Definition of the Package Building System

Piotr Zalewa

build — May 3, 2010

1   Syntax

1.1   Objects

`x`, `y`, `z` — *represents* `[a..z]`
`m`, `n` — *represents* `[0..9]+`

`Ux` is the specific User (identified by *User:name*)

`Px` is the specific Package (identified by *Package:name*)
It should always be used within its `type` context as `Lx` — Library or `Ax` — Addon
Every Package has an associated PackageRevision[1] (identified by a triplet `Ux:Py.n`
User/Package/PackageRevision:revisionNumber)

`Mx` is the Module (identified by `Ux:Py.n:Mz` *PackageRevision/Module:name*[2])

1.2   Object identification — revision numbers and HEAD

`Ux:Py.n` defines revision of the Package.
`Ua:La.1` — First revision of Library `La` saved by `Ua`.

`Ux:Py.n:Mz` defines the precise Module revision — a Module inside the PackageRevision.
`Ua:La.1:Ma` — Module `Ma` inside the first revision of Library `La` saved by `Ua`.

`Px` $\implies$ `Uy:Px.n` is the HEAD revision of the Package
`La` $\implies$ `Ua:La.1` — `La`'s HEAD points to the first revision of Library `La` saved by `Ua`.

`Ux:Py.n` $\supset$ `{Ux:Py.m:Mz, ...}` Modules inside the Package revision.
`Ua:La.2` $\supset$ `{Ua:La.1:Ma, Ub:La.2:Mb}` — Second revision of Library `La` saved by `Ua` contains `Ma`
saved by `Ua` in his `La`'s first revision and `Mb` saved by `Ub` in his second `La`'s revision.

---

[1]PackageRevision is not the same as Package version. The latter is just meta-data, a text field of PackageRevision object used only in exported `XPI`. It will no longer be used for data identification.

[2]Every data object is identified by a PackageRevision. The concept is similar to *git*'s commits. In essence, for every saved Module change, a new PackageRevision is created.

2 Relations between database objects

Graph of a sample database stage for the `La` $\implies$ `Ua:La.1` $\supset$ `{Ua:La.1:Ma, Ub:La.1:Mb}`. Every object relates to the appropriate User.
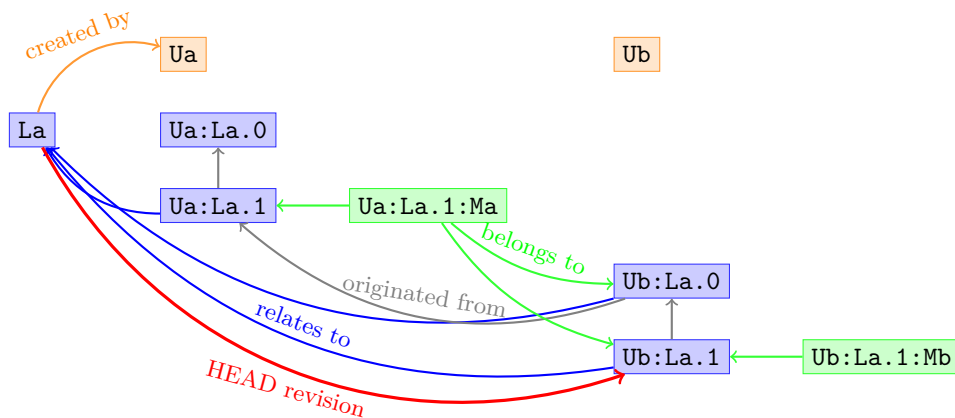
Real world example will be more complicated. In essence a PackageRevision might (and most of the time will) be originated from more than one PackageRevisions. There is also no mention of Library dependencies.

3 Exporting `XPI`

Be aware that it is possible and common to export `XPI`[3] from partially unsaved data. This happens when User will use the "Try in browser" functionality. In this case `XPI` may not be send to `AMO`[4].

3.1 Creating directory structure

Directory structure should be as close as standard Jetpack SDK as possible.

Create temporary directory and copy Jetpack SDK Packages

- `/tmp/packages_{hash}`[5]`/`

  - `development-mode/`
  - `jetpack-core/`
  - `nsjetpack/`
  - `test-harness/`

3.2 Exporting Packages with Modules

1. Create Package and its Modules directories
   `/tmp/packages_{hash}/{Package:name}/`
   `/tmp/packages_{hash}/{Package:name}/lib/`

2. Use collected data to create the Manifest.
   `/tmp/packages_{hash}/{Package:name}/package.json`

---

[3]An XPI (pronounced "zippy" and derived from XPInstall) installer module is a ZIP file that contains an install script or a manifest at the root of the file, and a number of data files.

[4]`http://addons.mozilla.org/`

[5]hash is a random string, different for every exported `XPI`

3. Create Module files
   Iterate over the assigned Modules and create a ".js" file with its content inside Package's `lib/` directory.

4. Export dependencies
   Iterate over Libraries on which a Package depends and repeat this section (*Export the Package with Modules*) for every Library.

3.3 Building `XPI`

System is already in a virtual environment knowing about Jetpack SDK. It is enough to change directory to `/tmp/packages_{hash}/{Package:name}/` and call `cfx xpi`. The `{Package:name}.xpi` file will be created in current directory. Its location is then send to the front-end to be used in further actions, usually calling the *FlightDeck Addon*[6] to download and install the `XPI`.

3.4 Uploading to AMO

Create `XPI` from the database object. Use `mechanize` lib to login to `AMO`and upload the file faking it was done directly from the browser.

4 Editing Package and its Modules

How database evolves by changing the Packages and Modules. This description will be used later to design structure and functionalities of the system.

4.1 Starting point

All next scenarios start from the `Ua:La.1` defined as below.

`La ⟹ Ua:La.1 ⊃ {Ua:La.1:Ma}`

Package `La` is created by User `Ua`.
`La`'s HEAD is PackageRevision identified as `Ua:La.1`
It contains only one module - `Ma`
Following steps had to happen to achieve above status:

	Ua	
Ua creates empty Library La	●	Ua:La.0 ⊃ {}
System sets La's HEAD	●	La ⟹ Ua:La.0
Ua adds new Milestone Ma to La	○	Ua:La.1 ⊃ {Ua:La.1:Ma}
Ua sets the HEAD	●	La ⟹ Ua.La.1
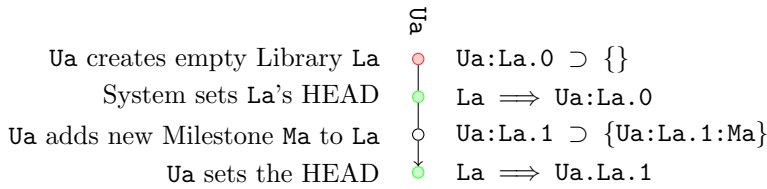
4.2 Scenario (1 Module, 2 Users, no dependencies)

`Ua` and `Ub` are working on `La`
`Ub` modified one module

---

[6]FlightDeck Addon is a Jetpack extension allowing to temporary installation of the `XPI`. It needs to be called with an URL of the `XPI`.

Ua Ub

Ub creates new branch   Ub:La.0 ⊃ {Ua:La.1:Ma}
Ub saved changes to Ma   Ub:La.1 ⊃ {Ub:La.1:Ma}
Ub sends *request* to La's creator to update La's HEAD
Ua accepts the request by setting the HEAD to Ub's version   La ⟹ Ub:La.1

Result: La ⟹ Ub:La.1 ⊃ {Ub:La.1:Ma}

4.3 Scenario (2 Modules, 2 Users, no dependencies)
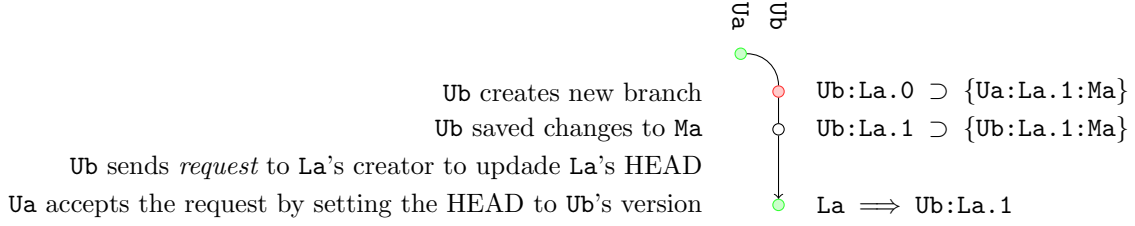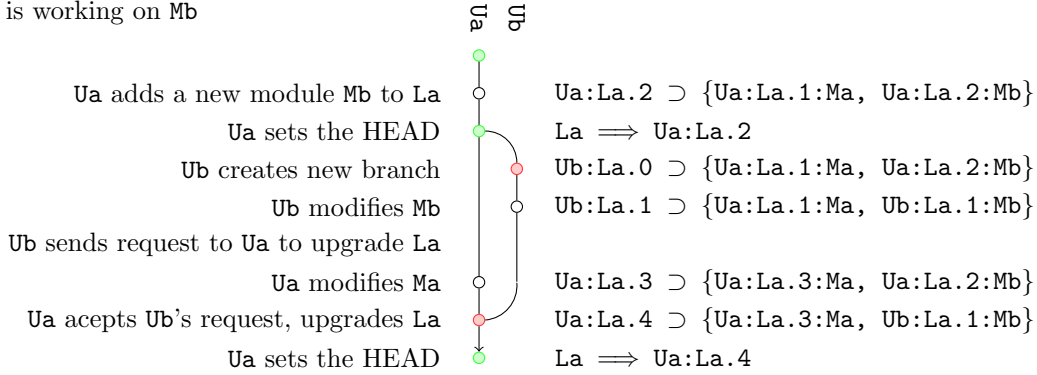
Ua and Ub are working on La
Ua created module Mb
Ub is working on Mb

Ua Ub

Ua adds a new module Mb to La   Ua:La.2 ⊃ {Ua:La.1:Ma, Ua:La.2:Mb}
Ua sets the HEAD   La ⟹ Ua:La.2
Ub creates new branch   Ub:La.0 ⊃ {Ua:La.1:Ma, Ua:La.2:Mb}
Ub modifies Mb   Ub:La.1 ⊃ {Ua:La.1:Ma, Ub:La.1:Mb}
Ub sends request to Ua to upgrade La
Ua modifies Ma   Ua:La.3 ⊃ {Ua:La.3:Ma, Ua:La.2:Mb}
Ua accepts Ub's request, upgrades La Ua:La.4 ⊃ {Ua:La.3:Ma, Ub:La.1:Mb}
Ua sets the HEAD   La ⟹ Ua:La.4

Result: La ⟹ Ua:La.4 ⊃ {Ua:La.3:Ma, Ub:La.1:Mb}

4.4 Scenario (2 Modules, 2 Users, no dependencies)

Ua and Ub are working on La
Ub created module Mb

Ua Ub

Ub creates new branch   Ub:La.0 ⊃ {Ua:La.1:Ma}
Ua modifies Ma   Ua:La.2 ⊃ {Ua:La.2:Ma}
Ub adds a new module Mb to La   Ub:La.1 ⊃ {Ua:La.1:Ma, Ua:La.1:Mb}
Ub modifies Mb   Ub:La.2 ⊃ {Ua:La.1:Ma, Ub:La.2:Mb}
Ub sends request to Ua to upgrade La
Ua accepts Ub's request   Ua:La.3 ⊃ {Ua:La.2:Ma, Ub:La.2:Mb}
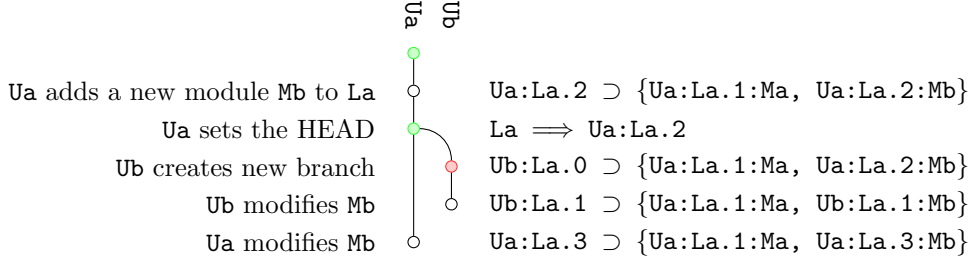Ua sets the HEAD   La ⟹ Ua:La.3
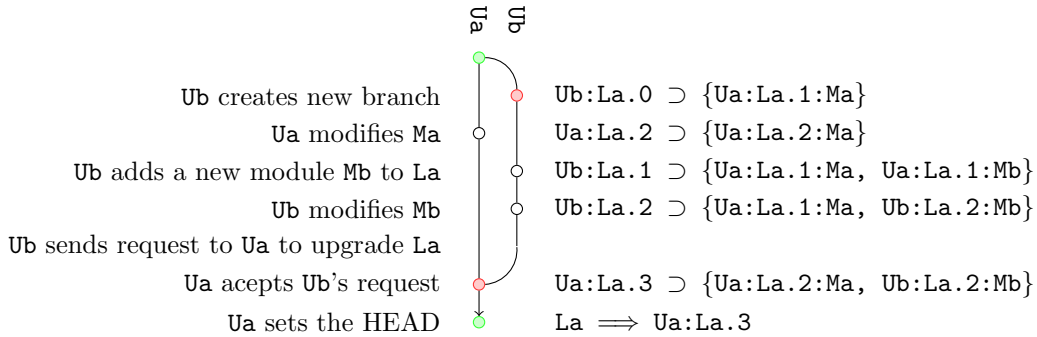
Result: La ⟹ Ua:La.3 ⊃ {Ua:La.2:Ma, Ub:La.2:Mb}

4.5 Scenario with conflict (2 Modules, 2 Users, no dependencies)

Ua and Ub are working on La
Ua created module Mb
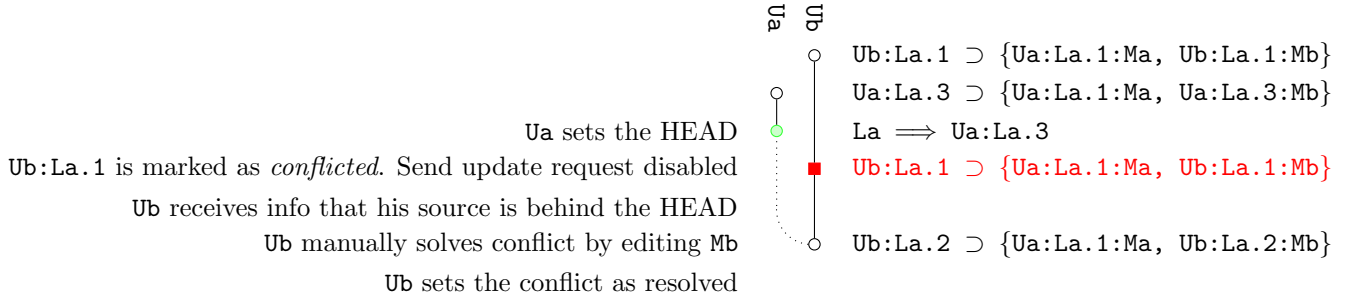
4

`Ua` and `Ub` are working on `Mb`
Conflict arises...

Steps leading to the conflict:

<div style="text-align:center">

Ua Ub

</div>

`Ua` adds a new module `Mb` to `La`	`Ua:La.2` $\supset$ {`Ua:La.1:Ma, Ua:La.2:Mb`}
`Ua` sets the HEAD	`La` $\implies$ `Ua:La.2`
`Ub` creates new branch	`Ub:La.0` $\supset$ {`Ua:La.1:Ma, Ua:La.2:Mb`}
`Ub` modifies `Mb`	`Ub:La.1` $\supset$ {`Ua:La.1:Ma, Ub:La.1:Mb`}
`Ua` modifies `Mb`	`Ua:La.3` $\supset$ {`Ua:La.1:Ma, Ua:La.3:Mb`}

Libraries `Ub:La.1` and `Ua:La.3` are **conflicted** because `Ub:La.1:Mb` and `Ua:La.3:Mb` are both an evolution of the `Ua:La.2:Mb`. From that moment many scenarios may happen. Just a few of them will follow.

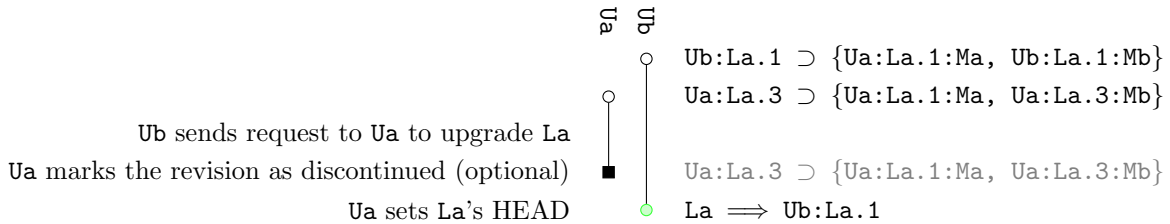### 4.5.1   `Ua` sets HEAD and `Ub`'s revision is outdated

`La`'s manager — `Ua` has chosen the HEAD. At that moment he doesn't know about `Ub`'s changes to `Mb`.

<div style="text-align:center">

Ua Ub

</div>

	`Ub:La.1` $\supset$ {`Ua:La.1:Ma, Ub:La.1:Mb`}
	`Ua:La.3` $\supset$ {`Ua:La.1:Ma, Ua:La.3:Mb`}
`Ua` sets the HEAD	`La` $\implies$ `Ua:La.3`
`Ub:La.1` is marked as *conflicted*. Send update request disabled	`Ub:La.1` $\supset$ {`Ua:La.1:Ma, Ub:La.1:Mb`}
`Ub` receives info that his source is behind the HEAD	
`Ub` manually solves conflict by editing `Mb`	`Ub:La.2` $\supset$ {`Ua:La.1:Ma, Ub:La.2:Mb`}
`Ub` sets the conflict as resolved	

From that moment `Ub:La.2` becomes a normal (not conflicted) PackageRevision. `Ub` may send Package manager an upgrade request which could end by switching `La`'s HEAD to `Ub:La.2`. It is important to note, that the `Ub:La.2` is not an evolution of `Ua:La.3`, it will not be originated from it.[7]

4.5.2   `Ub` sends update request, `Ua` decides to drop his changes

`Ub` thinks his change to `Mb` is finished and requests update of the Library from its manager — `Ua`. He accepts the request and marks his version of this module as discontinued. This mark prevents from the automatic set to conflicted revision.

<div style="text-align:center">

Ua Ub

</div>

	`Ub:La.1` $\supset$ {`Ua:La.1:Ma, Ub:La.1:Mb`}
	`Ua:La.3` $\supset$ {`Ua:La.1:Ma, Ua:La.3:Mb`}
`Ub` sends request to `Ua` to upgrade `La`	
`Ua` marks the revision as discontinued (optional)	`Ua:La.3` $\supset$ {`Ua:La.1:Ma, Ua:La.3:Mb`}
`Ua` sets `La`'s HEAD	`La` $\implies$ `Ub:La.1`

---

[7] Decide if this is the right thing to do.

Draft/Ideas

update Library if Library HEAD has been changed something should tell the User that an update is possible. It should then (on request) change the versions of all Modules which are not in conflict with updating Library. In essence, if

`Ua:La.1` ⊃ {`Ua:La.1:Ma, Ub:La.2:Mb`} is a Library to be updated and

`La` ⟹ `Uc:La.3` ⊃ {`Ub:La.1:Ma, Uc:La.3:Mb, Uc:La.1:Mc`} is current HEAD, then

`Ub:La.2:Mb` should be updated to `Uc:La.3:Mb` and `Uc:La.1:Mc` should be added.

User should receive a notification that `Ua:La.1:Ma` is not in sync with HEAD.

To be continued. . .