How to Make a LightTable

David Siegel

DesignTable@outlook.com

Introduction

Light Table is Chris Granger's open source IDE, written in ClojureScript.

http://www.lighttable.com/

Light Table's source can be found on github.com at:

https://github.com/LightTable/LightTable

Light Table Source

by the Numbers

Directory	File count
lt	2
lt/objs	46
lt/objs/clients	4
lt/objs/editor	2
lt/objs/langs	2
lt/objs/sidebar	4
lt/plugins	4
lt/util	7

First Impressions

The numbers point to a relatively small object-based system, and suggest a relatively flat structure.

These impressions are true, but also misleading.

Second Impressions

The first step in building Light Table from source is downloading a 51 meg tarball. That tarball expands into 155 meg of node-webkit and plugins.

The system is somewhat bigger than our first glance indicated.

To understand it, we'll have to start with node-webkit.

Dance of the Matryoshkas

Node.js

According to its website, nodejs.org, node.js is:

> *a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.*

node.js is known for its single-threaded implementation, and the callback-heavy coding style it uses.

The Light Table source code generally isolates its use of callbacks. For example, the files module exports a synchronous set of file operations.

Node-Webkit

Node-Webkit supports plugins.

Light Table is a Node-Webkit plugin.

Node-Webkit plugins can contain nested plugins

LightTable has plugins.

Clojure support is provided by a Light Table plugin.

LightTable has user plugins.

User plugins can override system plugins.

Light Table plugins can inject code into client processes.

Node-Webkit starts from a manifest file: package.json

```
    "name": "LightTable",
    "main": "core/LightTable.html",
    "js-flags": "--harmony",
    "single-instance": true,
    "webkit": {
        "plugin": true
    },
    "chromium-args" : "--disable-threaded-compositing ...",
    "window": {
        "icon": "core/img/lticon.png",
        "width": 1024,
        "height": 700,
        "min_height": 400,
        "min_width": 400,
        "position": "center",
        "show":false,
        "toolbar": false,
        "frame": true
    }
}
```

The key "main" points to the starting webpage, *core/LightTable.html*.

Startup

LightTable.html loads the libraries:

- node_modules/lighttable/util/keyevents.js
- node_modules/lighttable/util/throttle.js
- node_modules/lighttable/bootstrap.js

before initiating LightTable with:

```
lt.objs.app.init();
```

keyevents.js and throttle.js are small JavaScript libraries for controlling event processing. bootstrap.js is Light Table compiled.

Take Away

Within a deployed LightTable system, LightTable code can be found at:

- LightTable_Home/core/node_modules/lighttable
- LightTable_Home/plugins
- LightTable_USERDIR/plugins

Light Table Architecture

- Object Definition
- Object
- Command
- Behavior
- Tag

Objects, but not all the way down

LightTable's Object Model

An object is a map wrapped in an atom. Objects are mutable, and change in response to events.

Objects are defined by obj-defs (classes, more or less) that contain default values for the object, and an initialization function.

These are generally high-level objects; many are singletons.

From lt/obj/app.cljs:

The object definition of **app**:

```
(object/object* ::app
                :tags #{:app :window}
                :delays 0
                :init (fn [this]
                        (ctx/in! :app this)))
```

Object initialization

The creation of the singleton instance of **app**:

```
(def app (object/create ::app))
```

The object is created from the object-definition named *::app*.

The name of the object-definition becomes the ::type of the object.

An ::id is assigned.

{delays 0} is added to the new object.

The tags #{:app :window} are added to the new object.

Some additional system-managed slots (:args, :behaviors, :listeners, :tags) are allocated.

Object initialization (cont.)

The init function is run; it may modify the new object.

If the init function returns a vector, the crate library converts it into a DOM element node, and it is assigned to the :content slot. ::app does not return a content node.

Animating the Model

Commands

From lt/objs/app.cljs:

```
(cmd/command {:command :window.new
              :desc "Window: Open new window"
              :exec (fn []
                      (let [w (open-window)]))})
```

A command is, basically, a named function. Command functions bound to keys take no arguments. Commands invoked from code (cmd/exec!) can be passed arguments.

Behaviors

From lt/objs/app.cljs:

```
(behavior ::focus-class
          :triggers #{:focus :show}
          :reaction (fn [this]
                      (dom/add-class (dom/$ :body) :active)
                      (dom/remove-class (dom/$ :body) :inactive)))
```

When this behavior is triggered, the html class :active is added to the DOM's body element, and the class :inactive is removed.

Tags

The **Duct Tape** of Light Table?

What's a Tag?

A tag is a ClojureScript keyword.

Every object has 1 or more tags. An object starts with the tags of its object-definition.

Each tag can be bound to many commands and many behaviors.

Tags can be added to objects dynamically. When a tag is added to an object, all the commands and behaviors bound to the tag are now attached to the object.

Similarly, tags can be dynamically removed from an object, resulting in removing the bound commands and behaviors.

Keymaps Bind Commands to Tags

Tags act as "scopes" for keyboard commands.

from default.keymap:

```
{:+ {:app {"pmeta-r" [:refresh-connected-browser]
           "pmeta-shift-n" [:window.new]}}}
```

For objects tagged **:app**, the command :window.new is bound to the key sequence "pmeta-shift-n". Only the singleton app object carries the :app tag; it has global scope.

These commands only apply in the document search input section of the sidebar:

```
{:+ {:sidebar.doc.search.input {"enter" [:docs.search.exec]
                                "esc" [:docs.search.hide]}}}
```

Binding Behaviors to Tags

from default.behaviors:

```
{:+
 {:app [:lt.objs.clients.local/startup-with-local-client
        :lt.objs.app/focus-class]}}
```

Here we bind 2 behaviors to the tag :app.

Raising a Trigger

```
(object/raise obj :trigger arg1 arg2)
```

Here's a simplified view of the event handling mechanic in Light Table:

```
For each of obj's tags,
    for each behavior activated on :trigger
        execute that behavior reaction, passing obj and any args
```

Advanced Features

1. Dynamic Tags

Tags can be added or removed from objects dynamically. When this happens, behaviors and commands bound to those tags are attached or detached from the object.

Here's one way this feature is used:

The tag :files is associated with an object that maintains a registry mapping extensions to tags.

It maps, for example, the extension ".behaviors" to the tag ":editor.behaviors". When an editor is created to edit a file with extension ".behaviors", the editor will be dynamically tagged ":editor.behaviors", and there are some specific behaviors that come along with that tag.

Extending a Registry

Behaviors are used to make the :files registry extensible.

A language plugin can extend this registry by binding a behavior to the :files tag that responds to the trigger :lt.objs.files/file-types.

After plugins are loaded, :lt.objs.files/file-types is raised on the :files object, and all listeners add their extension to tag mappings.

The Clojure plugin, for example, uses this technique to map the ".cljs" extension to the tag :editor.cljs.

From clojure.behaviors:

```
{:+
  {:files [(:lt.objs.files/file-types
            [{:name "Clojure" :exts [:clj] :mime "text/x-clojure"
                 :tags [:editor.clj :editor.clojure]}
             {:name "ClojureScript" :exts [:cljs] :mime "text/x-clojurescrip
                 :tags [:editor.cljs :editor.clojurescript]}])]}}
```

2. Tag Specificity

Tags are more specific if they have more parts. Periods divide tags into parts.

:editor.clj (two parts, 1 period) is more specific than :editor.

Behaviors bound to more specific tags take precedence over behaviors bound to less specific tags.

3. Exclusive Behavior

If a behavior has the property :exclusive set to true, then other behaviors of the same name are ignored.

Constructing an Override

Dynamic tags, registry extension, tag specificity and exclusive behavior can be combined to override behaviors.

As we've seen, the Clojure plugin can arrange for editors of .cljs files to be tagged :editor.cljs

Let's say there's a behavior *b* bound to :editor.

The Clojure plugin can introduce a new behavior *b* bound to the more specific tag :editor.cljs. If the new behavior is marked :exclusive true, then it will block less specific behaviors *b*, like the one bound to :editor.

4. instant Behavior

If a behavior is given the trigger :object.instant, then it is executed immediately after being loaded, rather than waiting for the object to raise the trigger.

In effect, an :object.instant behavior acts as a directive.

Here's an extract from my user.behaviors file:

```
{:+ {:editor [(:lt.objs.style/font-settings "Anonymous Pro" 14 1)]
}}
```

The font is set immediately after user.behaviors are loaded, font-settings is not actually bound to :editor.

Summary

An object-definition sets the starting values for its instances.

Its name becomes their :type, and the return value of its init function becomes their :content.

An object is a map wrapped in an atom, stored in a registry.

Objects are not garbage collected, and must be explicitly destroyed.

WeakMap support is available for node-webkit (https://github.com/rogerwang/node-webkit/issues/298), and Light Table might be able to use it. It would require revalidating the object model.

Reserved Object keys

Set by Light Table:

::id
::type
:args
:content
:behaviors #{} :tags #{} :triggers [] :listeners {} :children {}

Can be set by user:

:name "myName"

Reserved Behavior Keys

:exclusive true
:throttle msec
:debounce msec

Important Functions

The following namespace definition is conventional in the LightTable code base:

```
(ns myPlugin
  (:require [lt.object :as object]
            [lt.objs.command :as cmd])
  (:require-macros [lt.macros :refer [behavior defui]]))
```

Object Creation

object/object* create an object definition
object/create
behavior
defui
cmd/command

Other Functions

Lookup

object/instances-by-type

Message Send

object/raise

Object Modification

object/refresh!

object/destroy!

object/merge!, object/update!, object/assoc-in!

--these should be followed by refresh!

Building Plugins

LightTable is Optimized for Writing LightTable

An expert is a person who has found out by his own painful experience all the mistakes that one can make in a very narrow field. - Niels Bohr

Plugin0

Plugin0 is intended to be the simplest demonstration plugin.

It displays some fixed content in a tab.

To make it work, though, we have to implement all the basic plugin mechanics.

Location

You can find the location of Light Table's user plugins directory by evaluating:

```
(lt.objs.files/lt-user-dir "plugins")
```

On Linux: ~/.config/LightTable/plugins/
It's different on Mac and Windows.

project.clj

```
(defproject plugin0 "0.1.0-SNAPSHOT"
  :description "plugin0: simplest LightTable plugin"
  :dependencies [[org.clojure/clojure "1.5.1"]])
```

No cljsbuild directives!

We will build the plugin from Light Table, which will just compile and combine the plugin ClojureScript code into "plugin0_complled.js".

No libraries will be added to the compiled JavaScript. The compiled JavaScript will be loaded into LightTable, and have access to Light Table's copy of the support libraries (e.g., the ClojureScript runtime).

Manifest

plugin.edn or plugin.json

This name is fixed.

```
{:name "plugin0"
 :version "0.0.1"
 :author "DesignTable"
 :source "https://github.com/DesignTable/plugin0"
 :desc "Simplest plugin for Light Table"
 :dependencies []
 :behaviors "plugin0.behaviors"}
```

Note the key "behaviors".

LightTable will load the referenced behavior file when it loads the plugin.

Behaviors

plugin0.behaviors -- referenced in the manifest

```
{:+ {:app [(:lt.objs.plugins/load-js ["plugin0_compiled.js"])
           (:lt.objs.plugins/load-keymap "plugin0.keymap")]
     :plugin0.panel [:dt.plugins.plugin0/destroy-on-close]}
}
```

Every plugin should start by loading its compiled javascript and its keymap.

The last line scopes the *destroy on close* behavior to the "class" *plugin0.panel*

Keymap

plugin0.keymap -- referenced in plugin0.behaviors

```
{:+ {:app {"ctrl-c 0" [:dt.plugins.plugin0/start-plugin0]}
     }
}
```

Anywhere in the app the key sequence *ctrl-c 0* will run the command *start-plugin0*

Plugin Source

from src/lt/plugins:
namespace declaration:

```
(ns dt.plugins.plugin0
  (:require [lt.object :as object]
            [lt.objs.command :as cmd]
            [lt.objs.tabs :as tabs])
  (:require-macros [lt.macros :refer [behavior defui]]))
```

Plugin Source (cont.)

```
(defui plugin0-panel [this]
  [:h1 "Plugin0: Hello!"])

; from browser
(behavior ::destroy-on-close
                  :triggers #{:close}
                  :reaction (fn [this]
                              ;(println "p0 destroy")
                              (object/destroy! this)))

(object/object* ::plugin0.panel
                :tags [:plugin0.panel]
                :name "plugin0"
                :init (fn [this]
                        (plugin0-panel this)))
```

Plugin Source (cont.)

```
(cmd/command {:command ::start-plugin0
              :desc "plugin0: Raise window"
              :exec (fn []
                      (println "start p0")
                      (tabs/add-or-focus! (object/create ::plugin0.panel)))})
```

Critique

Pro

Works
Small (the Light Table part!)
Coherent
Excellent for Interactive Web Development

Con

Lack of Documentation (esp. object-definition and object triggers)
Fail Fast vs. Robust

Thank you!