Iliad, a lightweight but powerful web framework

http://iliad.bioskop.fr
Université de Montpellier II, FRANCE

Nicolas Petton petton.nicolas@gmail.com
Sébastien Audier sebastien.audier@gmail.com

1. Introduction

Iliad is a new web framework for GNU Smalltalk released under the MIT license. We're working on this framework since a few months now. For our personal needs, we wanted to have the following features in the framework:

- *standalone stateful widgets*
- *REST-like applications*
- *simple API*
- *easy to setup and deploy (no complicated configuration step)*

In order to avoid to reinvent the wheel, we started it by reusing pieces of code from other libraries. In particular, we adapted the dispatch pattern from HttpView2, the composite element hierarachy for building HTML from Aida/Web, and the stateful Widgets from Seaside, however without using continuations. A bit of glue code was needed to make this work together, but we quickly ended up with something that actually worked.

The logical next step was to add a javascript layer to fully ajaxify those stateful widgets. Amazed by Weblocks "mark dirty" mechanism, we reused this idea to make it work with Iliad. I must say that the result was nice, and made Iliad applications really smooth and fast. A cool thing is that Iliad degrades nicely to full requests if javascript is not enabled, so the behaviour of Iliad remains the same.

2. Short tutorial

Iliad comes with several examples: the seaside-like counter, a simple blog using Magritte, and a todo list application.

2.1 Elements and Widgets

To use Iliad, you need to understand two important parts of the framework: Widgets and Elements.

Widgets are high level stateful graphical objects, while Elements are composite low level stateless objects for building HTML. Widgets use elements in their #contents method to build themselves.

People who are familiar with Aida/Web will immediately understand how elements work. Each XHTML tag has a corresponding element class which knows how to print itself as HTML with the #printHtmlOn: method.

```
e div
    class: 'example';
    h1: 'Hello world!'

<div class="example">
    <h1>Hello world!</h1>
</div>
```

The #contents method of widgets used to build html returns a block closure which takes an element as parameter:

```
Iliad.Widget subclass: MyWidget [

    contents [
        ^[:e |
            e div class: 'example';
h1: 'hello world!']
    ]
]
```

2.2 Applications

Iliad applications are special widgets, used as entry points of web applications.
Unlike other widgets, they know how to dispatch a request to the corresponding view

method.

```
Application subclass: MyApplication [

    MyApplication class >> path [
        ^'my_application'
    ]

    index [
        <category: 'views'>

        ^[:e |
            e h1: 'Hello world!']
    ]
]
```

The #path class method is important, it tells Iliad what is the base path of the application.

View methods in applications are pretty much like the #contents method of widgets. By default, they must be in the 'views' method protocol, else they won't be allowed to be used as view methods.

The #index method is the default view method, so this view can be reached at: http://localhost:xxxx/myApplication/index or http://localhost:xxxx/myApplication

2.3 The counter widget example

Let's take a look at the Counter widget class.

```
Iliad.Widget subclass: Counter [
    | count |

    initialize [

        super initialize.
        count := 0
    ]

    contents [
        <category: 'building'>
        ^[:e |
            e h2: count printString.
            e anchor
                action: [self
increase];
                text: '++'.
            e space.
            e anchor
                action: [self
decrease];
                text: '--'.]
    ]

    decrease [
        <category: 'actions'>
        count := count - 1.
        self markDirty
    ]

    increase [
        <category: 'actions'>
        count := count + 1.
        self markDirty
    ]
]
```

A counter widget has a count instance variable, initialized to 0. Its #contents method builds a header displaying the current count value, and two anchors, one to increase the count, and one to decrease it.

There is something new in this #contents method: the actions associated to the anchors. Actions are block closure that will be evaluated when the user clicks on the associated link or button. Here we use them to modify the count value with #increase and #decrease methods.

Also note the #markDirty call in #increase and #decrease. These method is really important when updating the state of a widget. It tells Iliad that the widget's state has changed, so it will be rebuilt.

2.4 The counter application

The counter widget is, like all Iliad widgets, a standalone graphical object. Let's create a simple application to see it in action!

```
Iliad.Application subclass:
CounterApplication [
    | counter |

    CounterApplication class >> path
['counter']

    counter [^counter ifNil:
[counter := Counter new]]

    index [
        <category: 'views'>
        ^[:e |
            e build: self counter]
    ]
```

]

As you may have noticed, the counter widget is stored into an instance variable, so a new one won't be created on each request. This is important, because our counter has state, and it must be maintained between requests.

Also, we don't call the #contents method of the widget directly. it should never be called from the outside.

That's it, you can see your counter at: http://localhost:xxxx/counter

NOTE1: When trying the counter example, you will notice that the widget is updated with AJAX requests. Iliad has a nice javascript layer which does that for us :). If javascript is disabled, it will degrades to normal requests, but the behaviour will remain the same.

NOTE2: The counter example code is available in the More/Examples/ directory. CounterApplication is a bit more complete, and shows also a multi-counter example.

3. Links

- Iliad's website: http://iliad.bioskop.fr
- SVN: http://bioskop.fr/svn/gst/iliad
- Related posts: http://smalltalk.gnu.org/tags/iliad