Segurify view system specificiation

# Case study

### Purpose

The purpose of the Segurify view system is to create an streamlined, dynamic application user interface rendering engine.

## Existing solutions

A number of solutions are already existing for the purpose, which most are working at the w3 standard. These view system are based on an HTTP request layer and are them all not suitable for modern web application where the interface are regulary update in accordance with the changes in the content. Among such changes include adding items to lists, and the common solution for this is the well known and used Ajax solution.

## Why this solution?

We need an totally new solution for dynamic interface that are able to meet the common expections today of modern web application but at the same time has an limited effort at the computer. Our viewpoint is that the common layers has become to complex for webmasters, coders, authors to manage. To many aspects and layers are involved to create the end user experience of the web. For example, many primitive tasks has to be make possible with dealing with to many technologies together in an complex situations.

Even the most simple apporoach at the web, such managing blog posts and text form submission involves today an decent and complex relations between a wide amount of technology pardigams, which all has slightly different behavior, and make it slightly difficult to support and the variety of code and technology standard among the platforms (HTTP, javascript, HTML/XMKL, system) making primitive problems slightly difficult and time consuming to solve.

# The Segurify technology

## Three level rendering engine

The Segurify rendering engine is divided into three levels.

Active Rendering Engine

Rendering Elements

Layout Markup

Preprocessor

User experience

### Layer 1: Preprocess

The preprocessing layer is compiling the view’s conditions. It are done by an bitwise preprocessing protocol which supports any preprocessing languages implementing an interface class called IScriptEngine.

The preprocessing is done **locally and not on the server.** The reason is to enasure the user has full control over the pre-rendering state. The purpose of preprocessor is to compile a set of layout markup which can make the layout rules.

**Preprocessing is one the client because these reasons:**

* **Client customization.** The preprocessor can customize the behavior in accordance with the client state, access by secured permissive system (measured for each app). This make interaction easier to manage.
* **Separate user interface and semantic information!** We propose web application should do the semantic communication with the client by feeds such Web Services, RSS or XML schemes, and not as with the todays sent of HTML. This will reduce costs for mobile users slightly and make an reduction of amount data traffic usage.

### Layer 2: Layout Elements

The preprocessor is rendering an preliminary set of XML elements called “Layout Elements”. It consists of an concept we’re familiar today, and has some characteristic found in XUL/HTML. The layout elements define the user experience.

### Layer 3: Rendering Elements

The next phase is the Rendering Elements. The Layout Elements are converted into Rendering Elements upon render time. The rendering elements has their distinct role at the end execution and will define how the layout will behave. The rendering elements are also more complex, and the reason to split up the rendering system to an additional abstraction layer is to make consistency of the rendering engine.

For example, rendering elements are specifying explicit how to behave and multiple rendering elements can together be a an result of an single layout element. Developers can also create new sets of rendering elements to custom Layout Elements but also inflate other sets of layout elements into another Layout and make them behave unique by instance parameters (xml attributes).

Another important aspect is that while Layout Elements has an specified set of attributes, rendering elements have an dictionary of arbitrary properties that can apply to all elements.

### Head-content view subset

All views is divided into two parts – header and content. The header view consists of an filed supporting main html content and is powered by Chromium. The content view is however powered by MediaChrome and is used for playlist contents. The division between the views is done by the “Split” tag which also defines the height of the head content and is placed before the beginning of the elements tag.

## Active Rendering Layer

The rendering engine is active in maintain state after loading. The engine will look for updated conditions and update all rendering elements with new properties or add elements previous not there by recurring interval.

The engine uses an synchronized buffer. When the runtime has downloaded and parsed an updated view, it associate it

Spotify Ultra uses a distinct View where all view consists of a mako like template. These template can be used to tweek the view to an particular query and all view is defining an particular namespace.

An preprocessing statement begins with an % throughout line until the line end or between <? ?> blocks. The preprocessing iself is done by the javascript engine jint. All views are an xml file divided into so called “section” statements and inside them resides some special kind of html like content.

# Hello World

<?

// Our classical statement hello world

var output = “Hello World!”;

// Repeat this five time

var count\_repeat = 5;

?>

<view>

<section name=”Overview”>

<h3>Hello World</h3>

<ul>

% for(var i=0; i < count\_repeat; i++){

<li>Item @{i}</li>

% }

</ul>

</section>

</view>

Will generate this output

<view>

<section name=”Overview”>

<h3>Hello World</h3>

<ul>

<li>Item @{i}</li>

<li>Item @{i}</li>

<li>Item @{i}</li>

<li>Item @{i}</li>

<li>Item @{i}</li>

</ul>

</section>

</view>

## Feed syndication

The advantage is to be able to compile JSON statements directly into the asserted view with the function synchronize\_data(str json). We have an feed (feed.json) with following content on our wwwhost root at localhost:

[**http://localhost/feed.json**](http://localhost/feed.json)

{

"items":[

{

"title":"Hello World",

"author":"By Me",

},

],

]

View’s source code

<?

// Our classical statement hello world

var data = synchronize\_data(“http://localhost/feed.json”);

?>

<view>

<section name=”Overview”>

<text left=”20” top=”30”>@{output}</text>

% for(var i=0; i < data.feed.length; i++){ var item = data.feed[i];

<text left=”20” top=”@TOP”> @{item.title} by @{item.author}</text>

% }

</section>

</view>

When ready it looks like

<view>

<section name=”Overview”>

<text left=”20” top=”30”>@{output}</text>

<text left=”20” top=”@TOP”> Hello world by Me</text>

% }

</section>

</view>

## View inflation

This is for the content part of the view, also after the space element.

Views can be used as component through an inflation system. As many items are used repeatedly, if there are some errors it could be exhausting to edit all files. Therefore there is an feature called “View inflation”.

This feature will allow the developer to develop components as own views, and then reuse the sub view. These components can have their own **parameter** which are set by the parent object by their attributes.

### Example

Album view is a common way to view songs. We put the album piece in own view, called album.xml and call this the album component. All xml attributes specified by the parent code is provided as an variable **arg\_{parameter}** for the child code. Notice this preprocessing is happening after the host preprocessing, so code between the host view and the inflated preprocessing **can not be shared.** The design reason of this is because those views should be treated as **components** and thus be independent of what context they’re put on. However @{variables} can define xml attributes for the child view before it are executed.

Album.xml

<?

/\*\*\*

\* Album view component. Not to use separetely

\*\*/

var album\_name = arg\_name;

var track\_count = arg\_track\_count;

?>

<view>

<section name="component">

<label title="@{album\_name}" height="23" top="@top" width="0"/>

<img src="http://farm1.static.flickr.com/2/1703693\_0412c29a4f\_m.jpg" width="128" height="128" top="@TOP" persistent="true"/>

<!-- An example tracklist -->

% for(var i=0; i < 10; i++){

<entry left="150" top="@TOP" height="18" author="Test" title="Item @{i}" collection="Test" album="test"/>

% }

</section>

</view>

This view is then put together in this file, artist view example:

<?

/\*\*

This is the view for the main view of Spotify Ultra

\*/

var current\_user = "drsounds";

var testdata = synchronize\_data("http://krakelin.com/sync.json");

?>

<view>

<section title="Overview">

<h1>Welcome to SpotifyUltra </h1>

<label left="20" top="@TOP" width="100" height="60">Welcome to Spotify Ultra. User is @{current\_user} </label>

<!-- Recent activity -->

<header width="320" height="20" top="@top" >Entries</header>

% for(var i=0; i < 10; i++){

**<inflate src="components/album.xml" track\_count="5" name="Album @{i} t"/>**

% }

</section>

</view>

Note the bold text. It are the inflation block. Pay attention to the concept where the attributes of the element is relative to the arg\_ variables we seen on previous page.