Cognition

Scott Southworth and Landon Barnickle

Cognition

Scott Southworth and Landon Barnickle

Contents

Design Goals

In general, frameworks get a pretty bad rap. They are bemoaned from one side for enormous bloat; they are denigrated for a lack of features from the other. Developers working within them quickly find limitations and constraints imposed from above in order to satisfy a framework's strange sense of rightness or purity, edginess or Victorian convention.

What begins as an application's foundation can, in the end, become its cage. More time is spent 'working around the framework' than is gained through its presence. Absolutely no one wants to be in this situation.

Every application of sufficient complexity and scope has a framework. The framework is just 'the how' and 'the why' and 'the glue' holding an application together. Sometimes, it makes sense to build exactly what you need; the framework and the application become nearly indistinguishable beasts. Many frameworks, like Python's Django, began as custom applications – and their most appropriate use-cases often reflect their origins.

Off-the-shelf frameworks only work when the design goals of the framework are in solid alignment with the actual needs of an application. Even the best written, most professional and elegant of frameworks will be detrimental to a project if each seeks a separate purpose.

The *Cognition* framework has a very specific set of design goals and interests. First and foremost, it is tailored for the creation of large, sprawling single-page web applications, i.e. the kind of thing you're likely to find on a corporate intranet.

Cognition is a completely client-side framework; it has no server-side rendering options. It enforces a strict reliance on a service-oriented architecture.

It assumes that programmers are fallible and messy and makes an honest attempt to clean things up. If one group wants to use Kendo components and another wants to visualize things in D3, it gives them the leeway to use their preferred tools. It caters to large teams – as they can encapsulate functionality completely within their modules – and limit the access of modules they load. And it handles routing and deep-linking more or less automatically.

But *Cognition* does not handle SEO. Search engines are nearly blind to *Cognition* applications. Public, consumer-facing websites that desire the intrusion of web crawlers and ninja bots are not well served by this framework. Page rank will not be theirs.

Cognition also requires modern ES5-capable browsers. No attempt has been made to sustain legacy systems. This simplifies the code-base and opens up a bevy of technical options and browser features. But it does completely preclude support for older browser environments.

Development Philosophy

composition gang of four

simple

embrace fallibility

frameworks should do their best to stay out of your way, providing minimal limits on creativity. they should handle the things that are onerous, that are easy to screw up, the things that can have hidden consequences.

resource contention – not ioc, command clear military hierarchy, ambiguous leadership kind of bites file management house cleaning and the state of my listeners race conditions error handling encapsulation and coupling – pipes and valves

rogue pragmatism dependency declaration

Catbus

Data Store (Cat?) and Message Bus (Bus!) in Javascript

Catbus API Documentation

Bus Methods

Name	Parameters	Description	Returns
at, location	name: string, (optional) tag: string	Creates or retrieves a Location with the given name stored on the bus. A tag can be given to the Location; this will travel with any messages generated at the Location. The tag is the same as the name by default.	Location
dropHost	name: string	Drops (i.e. destroys) any Sensors or Locations with the host name provided.	boolean (host existed?)
flush	none	Triggers the processing of all pending messages on the bus. This is called automatically	self

Location Methods

Name	Parameters	Description	Returns
on, topic	(optional) topic: string	Creates a Sensor watching this Location for messages associated with the given topic. The default topic is ‘update’.	Sensor
peek	(optional) topic: string	Returns the metadata associated with the last message written to the Location for the given topic (default: ‘update’).	message metadata
read	(optional) topic: string	Returns the last message written to the Location for the given topic (default: ‘update’).	msg: *

Name	Parameters	Description	Returns
write	msg: *, (optional) topic: string, (optional) tag: string	Writes a message to the Location, triggering any Sensors currently watching it under a matching topic (default: ‘update’). The default tag is the one associated with the Location.	self
refresh	(optional) topic: string, (optional) tag: string	Rewrites the current data at the Location, triggering any interested Sensors (note: the change flag would completely suppress this).	self
toggle	(optional) topic: string, (optional) tag: string	Writes the (!current data) at the Location, triggering any interested Sensors.	self
tag	none	Gets the tag of the Location.	tag: string
name	none	Gets the name of the Location.	name: string

Sensor Attributes

Name	Parameter	Description	Setter Default	Sensor Default
at, location	location: string or Location	Assigns a new Location to the Sensor (thus no longer watching a prior Location).	current Location	original Location
on, topic	topic: string	Assigns a new topic to the Sensor (thus no longer following a prior topic).	‘update’	‘update’
name	name: string	Assigns a name to the Sensor.	null	null
run	callback: function	Sets a callback to be invoked by the Sensor when triggered. Can run in specified context attribute.	null	null
filter	handler: function	Sets a function to silently filter messages in the Sensor so they do not trigger or accumulate. The handler will receive (msg, topic, tag) and should return true to continue processing the message. Can run in specified context attribute.	null	null

Name	Parameter	Description	Setter Default	Sensor Default
transform	handler: function	Sets a function to transform messages (if not filtered). The handler will receive (msg, topic, tag) and should return a new modified message. Can run in specified context attribute.	null	null
pipe	location: string or Location	Sets a target Location to which the Sensor writes when triggered.	null	null
change	flag: boolean	Prevents a Sensor from triggering unless an incoming value differs from the last value received.	true	false
batch	flag: boolean	Causes a Sensor to accumulate messages as specified by the Sensor's keep attribute – until flushed (via nextTick(), requestAnimation-Frame() or by manually invoking bus.flush()).	true	false
group	flag: boolean	Causes a Sensor to accumulate messages in a hash by tag as specified by the Sensor's keep attribute – until flushed (via nextTick(), requestAnimation-Frame() or by manually invoking bus.flush()). Often used in tandem with batch and/or retain.	true	false
keep	string: 'last', 'first' or 'all'	Causes a Sensor to keep certain messages (only the first, the last or all of them). If the group flag is set, the keep rules will be applied to messages by tag (not by the full set).	'last'	'last'

Name	Parameter	Description	Setter Default	Sensor Default
defer	flag: boolean	Delays triggering the Sensor until messages without the defer flag have been processed.	true	false
retain	flag: boolean	Retains messages even after a flush in order to accumulate a fuller list (batch) or hash (group)	true	false
host	name: string	Assigns a new host name to the Sensor. When a host is dropped through bus.dropHost(name), all Sensors and Locations assigned to the host are dropped and/or destroyed.	null	null
need	tag(s): string or [strings]	Prevents a Sensor from triggering until it has received messages for all specified tags. Generally used with batch, group and/or retain flags.	null	null
active	flag: boolean	Enables or disables the Sensor ability to trigger.	true	true
max	count: integer	Limits the number of times a Sensor can trigger. When the max is reached, the Sensor will automatically drop(). A value of -1 has no trigger limit.	-1	-1
as	context: object	Sets the ‘this’ context that the filter, run and transform functions will use if needed.	self	self

Sensor Methods

Name	Parameters	Description	Returns
once	none	Sets the max triggers attribute to 1.	self
wake	none	Sets the active attribute to true.	self
sleep	none	Sets the active attribute to false.	self
peek	none	Returns the packet containing the Sensor's last incoming msg and metadata (not filtered or transformed).	msg metadata
read	msg: *	Returns the Sensor's last incoming msg (not filtered or transformed).	self
tell	msg: *, topic: string, tag: string	Writes a message to the Sensor. This should generally only be called by Location objects – but is exposed for hacking or debugging.	self
drop	none	Drops the Sensor's subscription to a Location, effectively destroying it.	self
attr	name: string	Gets the value of the given attribute.	attribute value: *
attr	name: string, value: *	Sets the value of the given attribute.	self
attr	{name: value, … }	Sets multiple attribute values on the Sensor.	self

The Framework

Typed Attributes

Some tag attributes support a type prefix to allow for more complexity (or even clarity). Typed attribute values are preceded by a type name and a space. The *data* type, in particular, will also create *sensors* to dynamically bind the value.

Auto

The *auto* type stores most values as strings but will convert 'true', 'false' and 'null' to their non-string counterparts; this is the default type for the value attribute of *data* tags.

```
<data name="isBigScreen" value="true" />
<data name="isSmallScreen" value="false" />
<data name="selectedTV" value="null" />
```

Note: to store these values explicitly as strings, use 'string true', 'string false' or 'string null'.

Boolean

The *boolean* type results in a stored boolean value of true or false. It expects strings of 'true' or 'false' as inputs.

```
<data name="isBigScreen" value="bool true" />
<data name="isSmallScreen" value="bool false" />
```

Note: This is generally not needed – see the *auto* type.

Config

A *config* attribute is resolved upon instantiation and does not respond to subsequent changes; it's a once and done setup variable.

```
<cog find="someDiv" url="config currentPage" />
```

In this example, the 'url' would be set to whatever string is contained within the *config* option named currentPage.

Data

Data types provide the name of a *data* location. The attribute is bound to the *data's* value via *sensors* that are automatically configured in the background.

For instance, a *cog* can load various URLs dynamically by using a *url* attribute typed as *data*.

```
<cog find="holder" url="data currentPage" />
```

In this case, the DOM element with an *id* of *holder* would always contain a *cog* whose URL was obtained from a *data* tag named currentPage. If the value of currentPage changed, a new *cog* would be loaded, replacing the prior one, using the string value of currentPage as its URL.

Index

The *index* type signifies that an array index will be used and requires no further naming specifics. It is currently only used as the default setting for the *key* attribute in *chain* tags.

Number

The *number* type attempts to cast and store a value as a Javascript Number.

```
<data name="retryCount" value="number 3" />
<data name="roughlyE" value="number 2.718" />
```

Most attributes would store the number as a 'string' without this specification.

Prop

The *prop* type reads a property from the current *cog's* script declaration. This is done only once at instantiation and does not respond to later updates.

```
<data name="appSettings" value="prop defaultSettings" />
```

Data tags can use the *prop* type to initialize complex values such as objects or arrays.

Run

The *run* type executes a function declared in the current *cog's* script declaration. This is done only once at instantiation and does not respond to later updates.

```
<cog find="holder" url="run getCatPage" />
```

This *cog* would resolve its 'url' based on the result of calling the function 'getCatPage' – which should be a method on the current *cog* that returns a string.

```
<data name="lastUser" value="run getLastUser" />
```

Likewise, *data* tags can run a function to set their initial value.

String

The *string* type simply specifies that the following value will be treated as a raw string. For *cogs*, the default type for the 'url' attribute is 'string'.

```
<cog find="div_one" url="cats_of_the_world.html" />
<cog find="div_two" url="string cats_of_the_world.html" />
```

These example statements are thus equivalent. They would both load 'cats_of_the_world.html' with the full path being relative to the location of the *cog* file containing this code.

For *data* locations, the default type is auto – meaning that 'true', 'false' and 'null' would be converted to non-strings.

```
<data name="trulyTrue" value="true" /> <!-- true -->
<data name="stringTrue" value="string true" />  <!-- 'true' -->
<data name="trulyNull" value="null" />  <!-- null -->
<data name="stringNull" value="string null" /> <!-- 'null' -->
<data name="stringString" value="string string" /> <!-- 'string' -->
<data name="justCat" value="string cat" /> <!-- 'cat' -->
<data name="aStringCat" value="string string cat" /> <!-- 'string cat' -->
```

Cognition Tag Reference

Alias

The *alias* tag provides a way to create references to specific URLs. Each of these can reference a parent *alias* via their path attribute in order to construct cascading paths, similar to a poor man's C macro. Although '../' is supported as a convenience, this tag is designed to discourage its usage.

Other cognition tags with url or path attributes can use alias names as values to handle file path resolution.

Using *alias* tags, one can rearrange the directory structure of an existing application without modifying the underlying code.

For clarity (and as a nod to constants and macros), the name attributes of *alias* tags are always fully capitalized with underscores between words by convention.

Alias Tag Examples

```
<alias name="CONTENT_ROOT" url="/public/app/html"  />
<alias name="PAGES"  url="pages" path="CONTENT_ROOT"  />
<alias name="CHART_PAGE" url="charts.html" path="PAGES" />
<alias name="CREDITS_PAGE" url="credits.html" path="PAGES" />
<alias name="BACK_x3" url="../../../"  />
```

Alias Tag Attributes

Name	Description	Required?	Types
name	The actual string used to reference the alias.	Yes	none
url	A relative or absolute path to an actual file or directory.	Yes	none
path	A directory used as a prefix for the *url* attribute. It can be an actual path or the name of another *alias* defined previously. They can be chained together.	No	none

Chain

To manage the life-cycles of an entire array of *cogs*, one can use the *chain* tag. It is inspired heavily by D3.js's system of mapping DOM nodes to discrete data elements with enter, update and exit events.

With *chains*, a series of cogs based on one url are mapped to the keys (or indices) of a source array of data. Data is injected into each via the *chain's* item attribute.

Other options can help to control how these *cogs* are layered and applied to the DOM.

Chain Tag Examples

```
<chain find="body" source="records" item="rec" url="row.html" prop="true" depth="true" />
<chain find="holder" url="COMBO_BOX" source="prop filters" item="config filter" />
<chain find="list" url="list.html" path="LAYOUTS" source="cats" item="item" key="cat_id" />
<chain find="content" source="people" item="prop dude" url="guy.html" build="sort" />
<chain find="blogroll" source="run getBloggers" url="blogger.html" order="true" />
```

Chain Tag Attributes

Name	Description	Required?	Types
find	The DOM element id used as placeholder by the *chain*.	Yes	none
url	A relative or absolute path to a file containing a *cog* definition with at least a display declaration. Used by every *cog* in the chain.	Yes	none
path	A directory used as a prefix for the *url* attribute. It can be an actual path or the name of another *alias* defined previously.	No	none
source	The data source used to construct the chain; it should resolve to an array. By default, it senses changes to a named *data* tag.	Yes	*data*, prop, run, config
item	The data item created within each cog created from the source array. By default, it builds a *data* tag named 'item' that is wired back to the source data.	No	*data* 'item', prop, run, config
key	A way to identify data elements that persist across source changes. An item's index in the source array is used by default. An object property name or a method to compute a key can be used as key values.	No	*index*, string, prop, run, config

Name	Description	Required?	Types
build	By default, build is set to 'append', and will append new elements into the *chain's* container. Use 'scratch' to destroy and rebuild with each change to the data source. Or use 'sort' to rearrange nodes to match their order in the source array.	No	none
order	If set to 'true', the order flag will apply the CSS order style on each *cog* in the *chain* using the source array's index values.	No	none
depth	If set to 'true', the depth flag will apply the CSS z-index style on each *cog* in the *chain* using the source array's index values.	No	none

Cog

The *cog* tag embeds a visible *cog* component within the HTML display of the current *cog*, loading (lazy-style) into (or about) the DOM container referenced by its *id* attribute.

Cog Tag Examples

```
<cog find="header" url="HEADER"  />
<cog find="panel" url="panel.html" path="core/components"  />
<cog find="alert" url="../../indicators/alert.html" />
<cog find="page" url="data currentPage" path="PAGES"  />
<cog find="status" source="run getStatus" item="config status" url="status.html" />
<cog find="footer" and="prepend" url="FOOTER"  />
```

Cog Tag Attributes

Name	Description	Required?	Types
find	The DOM element id used as placeholder by the *cog*.	Yes	none
url	A relative or absolute path to a file containing a *cog* definition with at least a display declaration.	Yes	*string*, data
path	A directory used as a prefix for the *url* attribute. It can be an actual path or the name of another *alias* defined previously.	No	none
source	The name of the data source used to construct the cog, generally an options object.	No	*prop*, data, run, config

Name	Description	Required?	Types
item	The name of the storage item created within the loaded cog to hold its configuration; it defaults to a *config* attribute named 'item'.	No	*config*, data, prop, run
and	The DOM method used to place the *cog* relative to the placeholder identified by the *find* attribute. Defaulting to 'append', other options include 'prepend', 'before', 'after', and 'replace'.	No	none

Config

The *config* tag is not yet implemented.

Data

The *data* tag creates a named location owned and governed by the *cog* in which it is declared. Its name is required to be unique only within this cog. The containing cog and its descendants can read, write and subscribe via *sensors* to the data contained within it. Parent cogs have no direct access to the *data* contained within their children by design.

Data Tag Examples

```
<data name="currentPage" />
<data name="furColor" value="red" prop="true" />
<data name="eyeColor" value="run determineEyeColor"  />
<data name="numArms" value="prop armCount" />
<data name="numLegs" value="number 4" />
<data name="dynamicConfig" value="config configFromFile" />
<data name="isHungry" value="true" inherit="true" />
<data name="theWordTrue" value="string true" />
```

Data Tag Attributes

Name	Description	Required?	Types
name	A unique name within the containing *cog* by which this *data* location may be referenced.	Yes	none
value	The initial value to be stored within this *data* location. By default, no value is assigned and it would read as undefined. Unless specified as a 'string' type, primitive values in the tag are automatically converted to true, false and null.	No	*auto*, string, data, prop, run, config

Name	Description	Required?	Types
inherit	Set the inherit flag to true to override the initial set value with the value of a *data* location of the same name if there is a match higher up the *cog* hierarchy.	No	none
prop	Set the prop flag to true to expose the *data* location object as a property in the *cog's* script declaration.	No	none

Feed

The *feed* tag creates an instance of a communication service – referencing a *service* tag defined above it in the *cog* hierarchy. It is stored in a hash of named feeds owned and governed by the *cog* in which it is declared. Its name is required to be unique only within this cog. It also generates a *data* location (of the same name by default) that holds the last response received by the *feed*.

Feed Tag Examples

```
<feed service="friendsOfCat" />
<feed service="currentWeather" name="weatherFeed" to="weatherData" prop="true" />
<feed service="friendsOfCat" request="true" />
```

Feed Tag Attributes

Name	Description	Required?	Types
service	The name of a *service* defined above this *feed* in the *cog* hierarchy.	Yes	none
name	A unique name within the containing *cog* by which this *feed* instance may be referenced.	No	none
to	The name of the *data* tag generated to store the last *feed* response; it defaults to the name of the *service*.	No	none
request	Set the request flag to true to initiate a service request upon instantiation	No	none
prop	Set the prop flag to true to expose the *feed* instance as a property in the *cog's* script declaration.	No	none

Hoist

The *hoist* tag loads a hidden *cog* above the *cog* in which it is declared. Hoisted *cogs* can act as libraries or state machines and do not use the *display* declaration. They can currently be stacked –

but not nested (yet).

Hoist Tag Examples

```
<hoist url="menuStateMachine.html" />
<hoist url="supplyChainFeed.html" path="APP_FEEDS" />
```

Hoist Tag Attributes

Name	Description	Required?	Types
url	A relative or absolute path to a file containing a *cog* definition that implements *script* and/or *blueprint* declarations. Any *display* declarations would be ignored.	Yes	*string*, data
path	A directory used as a prefix for the *url* attribute. It can be an actual path or the name of another *alias* defined previously.	No	none

Method

Motor

The *motor* tag is not yet implemented.

The *motor* tag provides a way to drive writes and function calls at various times and intervals.

Preload

The *preload* tag requires that a file be loaded before the *cog* containing its declaration is initialized. Preloaded files are not executed and are not placed in the *cog* hierarchy.

Preload Tag Examples

```
<preload url="whiskers.js" />
<preload url="princess.png" />
<preload url="returnOfTheCats.html" path="CAT_FEEDS" />
```

Preload Tag Attributes

Name	Description	Required?	Types
url	A relative or absolute path to a file containing a *cog* definition that implements *script* and/or *blueprint* declarations. Any *display* declarations would be ignored.	Yes	*string*, data
path	A directory used as a prefix for the *url* attribute. It can be an actual path or the name of another *alias* defined previously.	No	none

Prop

Require

The *require* tag requires that a Javascript file be loaded and executed before the *cog* containing its declaration is initialized. Use this tag to declare library dependencies for portions of an application.

Require Tag Examples

```
<require url="paws.js" />
<require url="kittenPower.js" path="JS_LIB" />
```

Require Tag Attributes

Name	Description	Required?	Types
url	A relative or absolute path to a file containing a Javascript library.	Yes	*string*, data
path	A directory used as a prefix for the *url* attribute. It can be an actual path or the name of another *alias* defined previously.	No	none

Sensor

Service

Valve

The *valve* tag is not yet implemented.

The *valve* tag lets one sandbox portions of the *cog* hierarchy. It provides a communications 'white-list' such that descendant *cogs* cannot affect things such as *feeds* or *data* locations that exist above the *valve*.