Nexus API

The Nexus

The Nexus is the web interface to Infusion, which allows arbitrary remote peers to be integrated with an Infusion component tree. This system is in the early stages of being implemented; many of the base libraries required for it are nearly complete. "A Nexus" is an instance of Infusion that contains the Nexus components. The deployment footprint of a Nexus is small: one may exist within the context of a single page in a web browser - as well as scaling up to local and distributed deployments.

Currently, the Nexus API closely mirrors Infusion's public API. The Infusion API takes the form of JavaScript function calls made within a JavaScript VM (e. g. in a browser or a node.js process), and they address the component tree through path strings. The Nexus API takes the form of JSON payloads sent over plain HTTP and WebSockets, and they address the component tree through URLs specifying the host machine and path string. The two APIs are isomorphic and every message that can be sent in one form has a direct equivalent in the other.

Ongoing work on the Nexus API is discussed at Nexus design revisions.

The Nexus uses standard protocols originating in web technologies. The protocols named here are not exclusive, and it would be perfectly possible to construct bindings to the Nexus over other protocols if they were found sufficiently widespread and usefully adopted - such as MQTT, CoAP, etc. The intent of the Nexus and its protocols is to usefully constrain semantics in a way that will promote genuine interoperability - that is, to promote the chance that messages may be successfully understood and acted upon, with minimal fresh programming-language-level development ("code") and architectural overhead - rather than that they may merely be "received" and "decoded", which is the province of the underlying protocols themselves.

The Nexus forms what has been described in the literature as an integration domain (Kell, 2009) - an alternative viewpoint might cast it as a system for composition of synthesized web services (Fan, 2008) (SWS, SWM, SST, etc.). In this case, we make no immediate plans for allowing users to discover the space of states and transitions of component services and hence construct synthesized services by composing the respective machines (although we plan to act as a platform on which such highly complex inference could proceed in the future). Instead, we plan to in the first instance encourage all participants to follow an integral model whereby all interesting states of their components are represented by distinct, fully realised bodies of publically addressable state - and whereby component composition simply consists of the composition of these bodies of state, and component interfacing simply consists of transduction of corresponding values of this state, which are brought into equivalence by means of (primarily) symmetric lenses (Pierce, 2011).

Note that as a result of this much more simplified correspondence - that is, correspondence between the "values of the state" itself, what we refer to under the name Model Transformation (Model Relay) is a very much more simple thing than often goes by that name in the wider literature - which refers to transformations between the structures of "state machines" rather than simply the "state". However, wherever we succeed in representing the correspondences between state as proxies for the states of any respective machines, naturally we will succeed in putting any supervening machines into correspondence as well, without necessarily becoming aware of the fact.

The Nexus API

Read Defaults

Description
Read the specification (as JSON) of a grade with a particular name - a namespaced "type name" in a global namespace of such names

Nexus API
Endpoint: /defaults/<grade name>
Protocol/Method: HTTP GET

The body of the HTTP response to this method consists of the JSON-encoded form of the component defaults, with MIME type application/json

Equivalent Infusion API
Function: fluid.defaults(<grade name>)
Arguments: grade name: String - the name of the component grade whose defaults are to be read
Return: defaults: Object - the defaults of the required component, as a JavaScript (JSON-equivalent) Object

Write Defaults

Description
Write the specification (as JSON) of a grade with a particular name - a namespaced "type name" in a global namespace of such names

Nexus API
Endpoint: /defaults/<grade name>
Protocol/Method: HTTP PUT

The supplied body of this method consists of the JSON-encoded form of the component defaults, with MIME type application/json

Equivalent Infusion API

Function: fluid.defaults(<grade name>, <defaults>)

Arguments:
grade name: String - the name of the component grade whose defaults are to be written
defaults: Object - the defaults of the required component, as a JavaScript (JSON-equivalent) Object

Construct

Description

Construct a component instance at a particular path in the component tree. The minimum information required is i) the path at which it is to be constructed (the parent component of this path must exist already) and ii) the type name for the component to be constructed. The call may also supply iii) additional options to be merged into the definition of this instance - which may designate additional grade names and/or subcomponent definitions.

Nexus API

Endpoint: /components/<path>

Protocol/Method: HTTP PUT

The supplied body of this method consists of the JSON-encoded form of the component options, with MIME type application/json

Equivalent Infusion API

Function: fluid.construct(<path>, <options>)

Arguments:
path: String/Array of String - the path at which the component is to be constructed - specified either as a period-separated String of segments, or an Array of these segments
options: Object - the options of the required component instance, as a JavaScript (JSON-equivalent) Object. At a minimum, this Object must have the field type set, holding the name of the component grade to be instantiated. Other fields may also be populated, in the same pattern as those sent to defaults (see below on component definition format)

Destroy

Description

Destroys a component at a particular path in the tree. If the component does not exist, this action is a no-op.

Nexus API

Endpoint: /components/<path>

Protocol/Method: HTTP DELETE

Equivalent Infusion API

Function: fluid.destroy(<path>)

Arguments:
path: String/Array of String - the path at which the component is to be destroyed - specified either as a period-separated String of segments, or an Array of these segments

Bind Model

Description

Subscribes to notifications to changes in the model state of a component with grade fluid.modelComponent somewhere in the component tree. This interface slightly violates the intent of mirroring Infusion's declarative API one-to-one. In the context of a non-networked Infusion deployment, model bindings are established by defining relay rules as part of a component's defaults or options. However, declarative relay rules only function when both ends of the relay are Infusion components. By contrast, the Nexus API has to support model binding to arbitrary clients, e.g. sensors with JavaScript APIs. Therefore, while an Infusion model relay rule and a Nexus model binding are conceptually the same kind of thing, the latter is not implemented in terms of the former. Instead, it is implemented with the same low-level JavaScript functions.

Nexus API
Endpoint: ws://host/bindModel/<component path>/<model path>

Protocol/Method: WebSockets connection

This sets up a persistent WebSockets linkage which may be used for both reading and writing to the model.

Outgoing messages: The outgoing connection from the Nexus will periodically send messages over the WebSockets bus, each of which holds a JSON-encoded representation of the model state at that point in time. An initial such message will be sent on connection, holding the state of the model at the particular component's model path (that is, the model at path model-path held by the component at component-path in the component tree). Thereafter, one further such message will be sent whenever there is a change in that model state.

Incoming messages: The client may send messages incoming towards the Nexus, each of which consists of a JSON-encoded ChangeRequest object. A ChangeRequest object consists of the following fields:

path: String - The path into the model where the change is to occur.
value: Any JSON type - the updated value to be stored at the supplied path (ignored for requests of type delete)
type: String - either the values add, merge or delete representing the type of change required

These are just the same fields as are supplied in the local API applier.change.

ChangeRequests will be fulfilled relative to the particular path at which the connection is bound - that is, a path of "" within the ChangeRequest corresponds to the model path supplied when the binding was initiated. JSON-formatted packets will continue to flow in both directions of the WebSockets connection as long as it is established. The client of the Nexus signals that they wish to cease observing the target model by closing the connection. If the client finds that the connection has been forcibly closed for any reason, it is invited to continue trying to re-establish it - on re-connection, it will receive a snapshot of the current model state as usual which it may use to resynchronise its own representation.

Equivalent Infusion API

Function: fluid.componentForPath(<path>)

Arguments: path: String/Array of String - the path at which the component is to be read - specified either as a period-separated String of segments, or an Array of these segments

Return: Component - the component at the particular path. The member model of this component may be inspected at any time. The component, which will be designated component may then be supplied the following requests:

Component Function: <component>.applier.modelChanged.addListener(<pathSpec>, <listener>, <namespace>)

Arguments: pathSpec: String - the relative path within this component for which notifications are required.
listener: Function - a function accepting the arguments (value, oldValue, pathSegs) where value represents the freshly updated value of the model held at the path pathSpec and oldValue represents the value it held before this change was triggered.
namespace: String - an optional string representing a namespace for the supplied listener. If such a namespace is supplied, this listener will displace any other with the same namespace from the listener list.

Component Function: <component>.applier.modelChanged.removeListener(<listenerSpec>)

listenerSpec: String or Function - either the identical function handle previously supplied to addModelListener or a value which had been given to namespace in such a call.

Component Function: component.applier.change(<path>, <value>, <type>)

Arguments: path: String or Array of String - the path within the component's model at which the change is to be triggered
value: Any JSON type (Object/Array/String/Number/Boolean) - the updated value to be stored at the supplied path (ignored for requests of type delete)
type: String - either the values add, merge or delete representing the type of change required

This API is identical with the one documented at ChangeApplier API - programmatic style in the Fluid Infusion documentation.

Format of component options

The APIs for Read Defaults, Write Defaults and Construct all either accept or return one value which is formatted according to the expectations of a Grade structure. This contains a subset of those options expected by Fluid Infusion components - some background reading can be consulted at Component Grades in the Infusion documentation, but for the purposes of the Nexus API the supported options and structure are a very small subset of these.

- type: String - Holds the principal grade of this component. This value must always be set. The framework grade fluid.modelComponent may be used as a base grade if no customisation is required.
- parentGrades: String/Array of String - Holds an additional grades for these components. These may be thought of as a Mixin with respect to the principal type of the component. The grade definitions for these grades will be fetched and merged into the document representing the final instance, from right to left - the contents of leftmost grades will take priority over ones to the right.
model - Any JSON type (Array/Object/String/Number/Boolean) An initial value to be held by this component's model on construction.

modelRelay - modelRelay records a set of model relay specifications setting up a permanent relationship between this component's model and any others of interest in the tree. These take a standard form with permitted fields source, target and singleTransform. See Model Relay in the Infusion documentation for an explanation of the format and semantics.

components - hash of String to component options Each grade may designate any number of subcomponents attached to this component, which are provided with component options in the same format as the top-level record. In this way, grades may encode "linked patches" of the component tree which will be instantiated in conjunction.

Note that the model and modelRelay area of the component options will in general make use of IoC References of the form `{context}.path` in order to reference components and model areas in other parts of the component tree. More information about the syntax and semantics of these references can be found at Context in the Infusion documentation.

Example Interaction

Here is a simple interaction between two participants - one of whom, participant A advertises a piece of UI exhibiting a toggleable state of some kind - for example, a hideable panel or a checkbox - and the other, participant B advertises a source of timed but otherwise undifferentiated events - for example a simple puffer switch or jelly bean switch. Either participant B or a third participant can provide the relay specification to "transduce" the stream of switch events onto the toggleable target. Two things are required - the specification of the "addresses" of the source and target, and the exact nature of the transduction rule.

Here is participant A's definition and registration:

```javascript
// Set up the definition of the toggleable type
fluid.defaults("participant-a.toggleable", {
  parentGrades: "fluid.modelComponent",
  model: {
    toggle: false
  }
});

// Construct a particular instance of the toggleable in the Nexus, to peer with an individual UI control
var toggle1 = fluid.construct("participant-a.application.toggle1", { // assume that participant-a.application component has been constructed already
  type: "participant-a.toggleable",
});

toggle1.addModelListener("toggle", function () {
  // bind to actual UI state here
});
</pre>

Let assume that participant B wants to both advertise their switch as well as perform the binding:

```javascript
// Define the "type" for our switch here
fluid.defaults("participant-b.switch", {
  parentGrades: "fluid.modelComponent",
  model: {
    count: 0; // a piece of "integral state" representing how many times the switch has been operated
  }
});

var switchPeer = fluid.construct("participant-b.interface.switch1", {
  type: "participant-b.switch",
  modelRelay: {
    source: "{switch1}.model.count",
    target: "[/ participant-a.toggleable&toggle1].model.toggle" // selector uniquely identifying participant A's toggle1 switch in the tree
    singleTransform: {
      type: "fluid.transforms.countToToggle",
      forward: "liveOnly", // only bind outgoing changes once the link is established
      backward: "initOnly", // only bind incoming changes during startup - the switch's state is not generally writeable,
      // but the relay needs to acquire phase on startup
    }
   
// Then, listening to the switch's actual state, we periodically issue changes in this style:
switchHardware.addListener(function () {
  switchPeer.applier.change("count", switchPeer.model.count + 1);
});
```
We've used the "Local API" binding style in both cases for clarity, but bear in mind that either or both the switch and the toggle could be hosted in different processes on different machines, and use the HTTP/WebSockets "Remote API" style for both type definitions and binding to the Nexus, which might be within either of those processes or a 3rd process.

References


W. Fan, F. Geerts, W. Gelade, F. Neven, and A. Poggi. Complexity and composition of synthesized Web services. In PODS, 2008 [PDF]

Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric Lenses. In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas, January 2011 [ PDF]