Kynetx Report

The Kynetx Rule Language

The First Internet Application Platform

Phillip J. Windley, Ph.D.
Chief Technology Officer
Kynetx
August, 2010

kynetx
this changes everything™

Reactive Systems

Imagine walking into Borders and having your smartphone alert you to the fact that the book you put on your Amazon wish list this morning is available right now and on sale. As another example, think about an application that gathers relevant articles from your RSS and Twitter feeds based on searches you've performed or that are related to an email you received from a friend today.

You can view the Internet is as a big reactive system. When you browse, tweet, or email, the Internet reacts to what you're doing.

These examples show the power that can be achieved when applications can work across multiple domains and multiple protocols at the same time. We think of this as "programming the Internet" and the results are much more impressive than those achieved by building a mere Web site. There's no reason that clients in different domains, like your smartphone and Web browser, shouldn't be cooperating under your guidance to help you get things done. But to make that happen, we need new architectures and programming paradigms.

One way of viewing the Internet is as a big reactive system. When you browse, tweet, email, and so on the Internet reacts to what you're doing, or so it seems. Thus, programming the Internet requires reacting to user activities. Existing Web programs do this in a fairly *ad hoc* manner because most Web frameworks provide little support for managing program data and control flow across individual user interactions.

This document describes a new programming language, the Kynetx Rule Language or KRL, and the system that runs it, the Kynetx Network Service or KNS. KRL is designed for programming the Internet and makes it easy for developers to create applications, or apps, that behave like the scenarios imagined above. KRL is a programming language for building reactive systems that respond to complex scenarios across multiple Internet protocols, domains, clients, and devices.

Linguistic expression and abstraction give programmers the tools to do amazing things without making heroic efforts.

When we invented KRL our goal was to build notational support for the hard things that Web programmers face everyday—especially on the client-side. Our mantra is "let the machine take care of the details." Linguistic expression and abstraction give programmers the tools to do amazing things without making heroic efforts.

Benefits of Using KRL

KRL is purpose-built for programming on the Internet. KNS provides a cloud-based platform for executing KRL programs.

The following lists specific benefits provided by KRL and KNS:

- **User-centric and user-controlled**—Apps built using KRL are intrinsically tied to users. The architecture of KNS is such that rulesets are always evaluated on behalf of an entity (user). Users control endpoints and thus the apps that they run and the events they raise.

- **Event controlled**—events provide a powerful, unifying abstraction for building reactive systems. Developers can easily write applications that respond to complex event scenarios.
- **Cross domain**—Kynetx apps can work across domains so that user purpose can be advanced regardless of online location. KRL is designed to cross the silos that have sprung up, as stand-alone Web applications, so developers can create applications that mash-up data from all across the Internet regardless of location or protocol.
- **Cross protocol**—Kynetx apps easily work across Internet protocols such as the Web, email, and so on. KNS is easily extensible by developers to support any protocol.
- **Data and context driven**—KRL and KNS are designed to easily and naturally work with the burgeoning array of data and APIs available online. Correlated data provides context about users. Using KRL and KNS, developers can create applications that respond to user context for a more compelling experience.
- **Cloud based**—because Kynetx apps are cloud based, they work consistently and ubiquitously. They can be accessed from multiple platforms while providing the same context, identity, and experience. Cloud-based programming means that programs always work because they are updated without user interaction in response to changing conditions.
- **Browser independent**—Kynetx apps work in all the major browsers without modification. The browser has become a sort of universal application platform, but browser differences make programming on them difficult. KRL provides a unifying framework for easily working with all of the popular browsers.
- **Internet app centric language and design**—KRL provides a powerful notation for creating apps that run across the Internet. KNS provides the platform that makes that possible.
- **Security and privacy are built-in**—the architecture of KNS is designed to limit nefarious activity structurally. In addition, operating in the cloud makes it easy to turn off apps that are misbehaving. User control provides the means to create privacy respecting apps.
- **Late binding—**Kynetx apps run at the exact moment that the user needs them. They bind to data and functionality that is appropriate for the user's current context. In contrast, conventional Web applications exist at a single location and operate without the benefit of user context.
- **Multi endpoint**—KNS provides application program endpoints that work with Web browsers, email servers, and other Internet systems. Kynetx plans to provide endpoints for popular and important Internet protocols and applications as part of its ongoing development roadmap. Developers can easily extend KNS to include endpoints for any Internet protocol.
- **Developer-friendly**—KRL is designed to provide developers with a powerful and easy to use abstraction layer for apps. KRL provides a notation that lets programmers easily complete Internet programming tasks that previously took many lines of code. Event

expressions, datasets, and data sources are just a few examples. Because Kynetx apps are hosted, developers are spared operational and maintenance headaches that come with servers.

A New Programming Model for the Internet

The following pages contained a brief description of the primary components of what entails a new programming model for applications that work across the Internet in behalf of the user—as opposed to the typical Web program that works on a single site on behalf of the site owner. The model and services we describe form a platform for creating Internet applications.

In the discussion that follows we explore in some detail the primary concepts in this new model: events and rules. We'll also briefly describe the architecture of the system of services that support this model. The document ends with three examples showing applications built to take advantage of this model.

Events and Rules

We call the activities like viewing a Web page, sending an email, or arriving at a new location an *event*. Events are one of the key concepts in KRL.

KRL programs react to events.

Complicated event scenarios are not uncommon in Web applications. KRL makes responding to them easy.

An event is a notification of an activity that happened at a particular time with specific attributes. We call a meaningful, related group of events an *event scenario*.

In the bookstore scenario I describe above, the events are

1. User updates Amazon wishlist
2. User arrives at bookstore

To notify the user that their book is available nearby, an application would have to take note of these events and, when they have been satisfied in the given order, take an action that sends the notification.

Scenarios like this are not unusual in Web applications. Surprisingly the most common way to deal with them programmatically is to build *ad hoc* logic that recognizes the event scenario and dispatches procedures to handle each scenario..

We'll discuss what kind of events KRL recognizes and what operators are available for describing complicated event scenarios in a later section.

KRL programs are made up of rules.

KRL programs, or rulesets, comprise a number of rules that respond to events. The basic pattern for a rule in KRL is

when	*an event occurs*
if	*some condition is true*
then	*take some action*

This pattern is not unique to KRL, many rule languages are event-condition-action, or ECA, rule languages. The ECA pattern is typical of most business rule systems and rule languages based on that pattern are a widely accepted way to build reactive systems.

Rules have three main parts: the event selector, the condition, and the action. KRL programs are collections of rules.

Rule languages have been heavily used in planning and reasoning systems, including expert systems and natural language processing. KRL is not a planning or reasoning system. Rather, KRL is a programming language for creating complete Internet applications.

The following shows a simple KRL rule.

```
rule morning is active {
  select when pageview “/archives/”
  if morning() then
    notify(“Welcome!”, “Good morning!”)
}
```

This rule would send a “good morning” notification to visitors of any page in the archives of a Web site (as denoted by the URL path) if its morning where the user is.

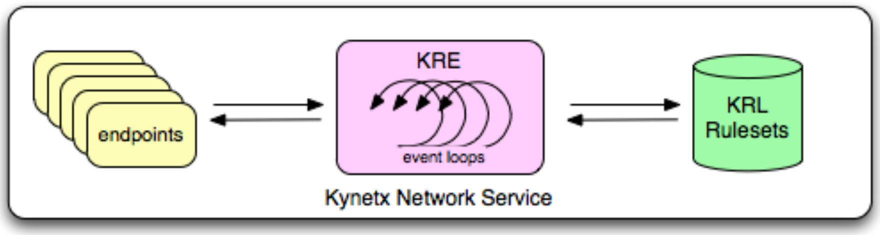
In the rule shown, the event expression is denoted by the keyword `select`, the condition by the expression following the keyword `if`, and the action by the notify action following the keyword `then`. The rule has a name: `morning`.

The rule `morning` is selected when the scenario given in the event expression is true (i.e. when there is a Web pageview event with a URL that matches the path expression given, “`/archives/`”). Once a rule is selected, the condition is tested. If the condition is true, then the action is taken and we say that the rule *fired*. Note that for a rule to fire, it has to be selected and the condition has to be true.

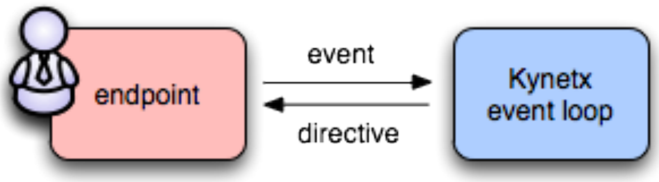
In addition to the basic rule syntax shown above, rules can have preludes where variables are declared and postludes where rule effects are computed. These, and other KRL features are described in a later section.

Event Loops and Endpoints

The Kynetx Network Service (KNS) is made up of endpoints and the Kynetx Rules Engine, or KRE. KRE uses rulesets to define the interactions it has with endpoints. The following diagram illustrates that relationship:



From the standpoint of an endpoint or group of endpoints operating on behalf of a user, the diagram can be abstracted to be simpler. The endpoint raises events with an event loop running in the cloud and receives directives in reply. The particular event loop that the endpoint interacts with is defined by the ruleset executing on behalf of the user.



KNS is built so that developers can define their own endpoints and the event domains they manage, but most developers will use pre-built endpoints that Kynetx supplies. At present Kynetx offers Web endpoints in the form of browser extensions for popular browsers and an IMAP endpoint[1].

Endpoints

Endpoints raise events and respond to action requests on behalf of a user or, more generally, an entity. Raising the correct event requires that the endpoint be aware of the entity it represents, what rulesets, or apps, the entity deems important, and which events those rulesets want to see (called event *salience*). Endpoints generally operate for one event domain.

Endpoints raise events on behalf of a user and respond to directives from rulesets when they're evaluated.

Endpoints raise events using the Kynetx event API[2]. The event API is implemented as an HTTP service. Using the API, an endpoint can construct a service call that contains relevant information about the event and the rulesets that should be run against that event.

Endpoints process directives. Often these directives are Javascript programs, but may be JSON data structures or other information appropriate to the endpoint. The endpoint is responsible for interpreting any directive it receives.

For example, a browser extension serving as a Kynetx endpoint would raise the appropriate events for the `web` domain including `pageview`, `submit`, and so on. The browser extension also responds to directives from the Kynetx event loop—the most important being to execute the returned Javascript in the context of the page.

As another example, an email endpoint would raise events for the `email` domain, including `received`, `sent`, etc. and respond to the directives that might include actions like `delete`, `forward`, and so on.

Event Loops

The Kynetx Rule Engine, or KRE, is the primary service provider in KNS. KRE interacts with endpoints and executes KRL programs. KRE can be envisioned as a system of event loops that run in the cloud. An event loop is a message dispatcher. The loop runs, waiting for events, and responding to them. KRE is extremely flexible, allowing developers to invent their own event types and then write KRL programs that respond to those events with free form directives.

KNS is a collection of programmable event loops that run in the cloud on behalf of a user.

As we will discuss in detail in the next section, event expressions in KRL can be designed to respond to complex event scenarios. KRL includes a set of event operators for combining event primitives into event expressions.

KRE responds to event API calls by executing the appropriate rulesets against the event that has been raised. Each rule in the ruleset is examined to determine whether or not it should be selected for the event and, if selected, executed. The result of a ruleset execution is usually a set of directives that are returned to the endpoint[3].

Event Expressions

Event expressions play such a critical role in understanding the operation of KRL. There are two types of event expressions:

- **primitive events** –comprise the specific event types in a given event domain.
- **compound events** –combine primitive events using event constructors such as `then` or `between`.

Primitive Events

As we've mentioned, a rule is selected when the rule's event expression is satisfied. For primitive events, that means that the event is raised and the conditions around that event have been met. The syntax for a select statement with a single primitive event[4] looks like this:

```
select when <event_domain>? <event_type>
  {<param_name> <regexp>}*
  [setting (<var>*)]
```

The event domain is a namespace within which events are raised. Endpoints are responsible for setting the event domain when they raise the event. Kynetx provides endpoints for the Web and email that raise events in the event domains `web` and `email` respectively. If the event domain is missing, it defaults to `web`.

The event type gives the specific event name for the event within a specific event domain. For the event domain `web` the following event types are currently defined:

- `pageview` –user viewed a Web page
- `submit` –user submitted a form
- `click` –user clicked on a page element
- `change` –user changed a form element

Primitive events are the basic building block of KRL event expressions. Primitive events are specific to an endpoint type.

The event types and what they mean are defined by the endpoint[5]. Other event domains have their own relevant event types[6]. For example, the Kynetx email endpoint defines the following event types:

- `received`—email received
- `sent`—email sent

Event parameters can be sent when an event is raised. A primitive event expression need not test all (or, in fact, any) of the event parameters that are submitted.

The setting clause allows variables to be set from captured portions of the regular expressions[7] in the event parameter tests.

A primitive event expression is only satisfied when the event domain and event type match exactly and any named event parameter matches the associated regular expression.

Event Expression Examples

As an example consider the following select statement from a KRL rule:

```
select when mail received
```

In this event expression, `mail` is the event domain and `received` is the event type. Any rule containing this statement will be selected when events matching the event domain and type are raised. Of course event expressions can be more complicated:

```
select when mail received from "(.*)@windley.com"
  setting(user_id)
```

This statement adds a parameter to test (`from`) and regular expression (`"(.*)@windley.com"`) along with a setting clause to name the captured variable. You can have as many parameter checks as needed. The select statement shown above will only be satisfied when the event domain and type both match *and* there is a parameter called `from` that has a value that matches the given regular expression. The endpoint is responsible for setting event parameters when it raises the event.

Compound Events

A good event language adds considerable power to rules. Compound event expressions allow us to combine primitive events[8] to form event scenarios[9]. Event expressions provide a robust notation for developers to express the situations in which their rules should be selected. By combining primitive events into event scenarios, developers can create sophisticated applications without requiring that they manage the state machine necessary to recognize those scenarios.

The following event operators are available:

`A before B` - event A occurred before event B:

```
      select when pageview "bar.html"
             before pageview "/archives/(\d+)/x.html"
          setting (year)
```

`A then B` - event A occurred then event B occurred with no intervening salient events

```
      select when pageview "bar.html"
             then pageview "/archives/(\d+)/foo.html"
         setting (year)
```

`A and B` - event A occurred and event B occurred in any order.

```
      select when pageview "bar.html"
              and pageview "/archives/\d+/foo.html"
```

Compound event expressions allow developers to succinctly specify complicated scenarios in their rulesets.

`A or B` - event A occurred or event B occurred.

```
      select when pageview "bar.html"
               or pageview "/archives/(\d+)/foo.html"
```

`A between(B, C)` - event A occurred between event B and event C

```
      select when pageview "mid.html"
      between(pageview "firs(.).html" setting(b),
              pageview "las(.).html" setting(c))
```

`A not between(B, C)` - event A did not occur between event B and event C

```
      select when pageview "mid.html"
          not between(pageview "firs(.).html" setting(b),
                      pageview "las(.).html" setting(c))
```

Of course, these can be nested as well. Parentheses specify execution order where precedent is not apparent

```
      select
        when pageview "mid.html"
          between(pageview "firs(.).html" setting(b),
                  pageview "las(.).html" setting(c))
        before pageview "/archives/(\d+)/foo.html"
            setting (year)
```

Other event expressions operators will be added as needed[10]. There's no restriction that requires all of the primitive events in a particular scenario coming from a single event domain. In fact, the most interesting event scenarios will employ multiple event domains.

Conditions and Actions

As we've seen, rules have more to them than just events—they need to conditionally respond to those events.

Conditions

Rules don't always fire when they are selected. Often, we want other factors to influence the outcome of a ruleset execution. We refer to the information

Conditions allow rules to take user context into account and give KRL the power to respond to individual user's circumstances.

surrounding a ruleset's execution as *context*[11]. Much of that context is specific to a particular user. Context includes event parameters, persistent data stored by KNS about the entity or application, and data from other network APIs and services such as Twitter, Facebook, Google GData, Microsoft ODATA, Amazon, and so on.

Responding to context in ruleset execution requires testing that context. KRL includes a full-featured expression language that can be used to gather and manipulate data, compute new values, and create predicates for use in conditional rules. The KRL expression language includes primitive literals, compound data structures, standard arithmetic and string operators, conditional expressions, and first-class functions (including closures).

Actions

The heart of any rule is the action that it takes. As we've mentioned actions are endpoint specific. A given rule can take multiple, or compound, actions. Compound actions can execute all of their actions, or just one at random (to enable A/B testing using KRL rules).

Actions are the heart of any rule. Rules can take multiple actions.

Because of its roots, KRL has a large number of built-in actions for the web domain[12]. Common actions in the web domain include:

- **notify**—place a notification box on the page with a message of the user's choice. Many rulesets use this for giving the user simple messages.
- **annotate_search_results**—annotate a list on a Web page (it doesn't have to actually be a search result, although that's the most common use case). Any item in the list that meets a developer specified criterion is annotated—usually with a picture. AAA of Washington used this to show AAA users where they get discounts in search results.
- **sidetab**—place a slideout tray on the side of the page. When the user selects the sidetab, the tray slides out to reveal the information the developer wishes to show. 7Bound used this to show conference goers an agenda and information about speakers.

Actions that are built-into KRL are designed to work across multiple Web sites and browsers. So, for example, the AAA search annotation mentioned above works in all the popular browsers and on all popular search engines.

KRL Structure and Features

A KRL ruleset is the primary unit of evaluation. A ruleset is a sequence of rules. These rules are, by default, executed in order when the ruleset is executed, although there are ways for developers to influence execution order. In addition to a sequence of rules, rulesets also contain other information necessary for the execution of the ruleset including meta data and global declaration sections (described below).

In addition to the basic attributes of rules discussed in the previous section, KRL has a number of important features that add to its power.

Preludes

Rules have preludes (denoted by the `pre` keyword) where values are computed before the condition is tested. The prelude contains a list of variable declarations that are executed in order and affect the rule environment (i.e. those declarations are valid within the scope of the current rule). The right hand side of the declaration can be any valid KRL expression.

Postludes

Rules can also have postludes (denoted by the keywords `fired`, `notfired`, or `always`) that are used to initiate effects that will last after the rule has finished executing. The following types of effects are possible:

- **explicit events**—raise an explicit event to potentially fire additional rules in the current or another ruleset.
- **ruleset control flow**—stop ruleset execution after this rule
- **persistent variables**—store values that can be used to alter the behavior of future rule executions.
- **explicit logging**—store data in the logs for analysis.

Foreach

Nested `foreach` statements are allowed immediately after the select statement of any rule. For each iteration of the loop, the entire rule body is executed once[13].

Thus, any given rule can be thought of as a FLWOR[14] (foreach, let, where, order by, result) statement. The rule prelude functions in the capacity of a "let," the rule premise (condition on the action) functions in the capacity of a "where," and the rule action itself is the "result." Sorting the array that the `foreach` iterates over does ordering.

Data

Data is critical to making interesting rulesets. KRL is designed to be a programming language for the Internet and thus has built-in features for linking to data all over the Web:

- **datasets**—datasets from any source can be referenced. The data in the data set is automatically cached and made available to the endpoint. This is particularly important for browser extensions where cross-site scripting concerns limit the ability of Javascript to gather data from domains other than the one where the script originated.
- **datasources**—datasources are KRL's general integration to APIs. Any URL can be used in a datasource. The rule can query the datasource using any context data available at the time of execution. KNS automatically caches data under programmer control.
- **intrinsic data**— KRL has over a dozen libraries[15] that intrinsically integrate APIs and information from around the Internet as well as general facilities for making API calls over HTTP. For APIs that require it, such as the Twitter API, KRL integrates OAuth so that

developers don't have to manage the OAuth protocol interaction by themselves.

Little Languages

KRL makes use of a number of embedded little languages to add power to the overall language and ease the programmer's job:

- **regular expressions**—regular expressions are used in KRL for checking event parameters, matching, replacing, and testing variables.
- **JSONPath**—JSONPath is the JSON analog of XPath for XML. JSONPath provides a convenient language for traversing and manipulating JSON, one of the most popular data formats on the Internet and the default data format in KRL.
- **jQuery selectors**—selectors allow portions of an HTML document to be selected and manipulated. They are used widely in KRL for working with HTML pages and data.

Meta Data

Rulesets can have meta data such as a name, description, and author declared in their meta data section (denoted by the `meta` keyword). In addition, the meta data section of the ruleset contains keys needed for API access (OAuth consumer keys, AWS developer keys, and so on). The meta data section is where outside resources (such as CSS and Javascript libraries) are declared.

Global Declarations

Every ruleset can have a global section (denoted by the `global` keyword) for declarations that are common to all of the rules in the ruleset. For example, data sources might be declared for query in later rules or functions might be declared for use in abstracting common functionality for several rules.

Example: Using Twitter Data

KRL has built-in primitives for using Twitter. Because an app that uses Twitter will use OAuth to obtain delegated authority to use the user's Twitter feed, a user will see significantly different behavior from such an app than their friend might; an app that uses the Twitter library will use my Twitter data when I run it and your Twitter data when you run it. Using OAuth and other authorization technologies, Kynetx apps can be personalized.

Using OAuth and other authorization protocols, Kynetx apps can be personalized in a permissioned way.

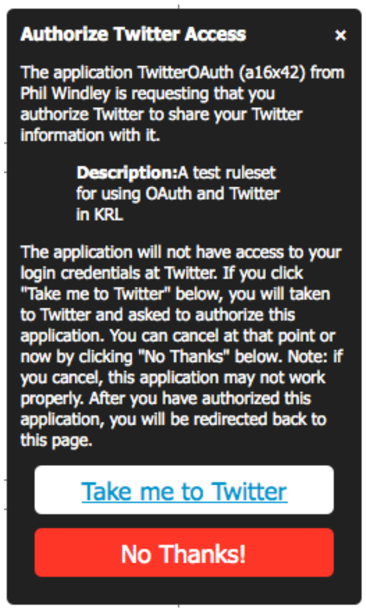
This example demonstrates the use of Twitter data inside a Kynetx app by KRL. Using Twitter data inside a KRL app generally involves two KRL patterns: **authorize then use** and **initialize then populate.**

In the **authorize then use pattern**, a rule is put in place to check if the app is authorized to take a certain action and, if not, do what is necessary to initiate the authorization ceremony. What makes this work is using the rule postlude to ensure that the rest of the rules (which presumably rely on the authorization) don't run. Here's an example:

```
rule auth is active {
  select using ".*" setting ()
  if(not twitter:authorized()) then
      twitter:authorize()
        with opacity=1.0 and
             sticky = true

  fired {
     last
  }
}
```

Notice that this rule only fires if the predicate `twitter:authorized()` is false. The action, `twitter:authorize()`, is what initiates the OAuth ceremony. The action will pop up a notification in the user's browser that looks like this:



Authorize Twitter Access ×

The application TwitterOAuth (a16x42) from Phil Windley is requesting that you authorize Twitter to share your Twitter information with it.

Description:A test ruleset for using OAuth and Twitter in KRL

The application will not have access to your login credentials at Twitter. If you click "Take me to Twitter" below, you will taken to Twitter and asked to authorize this application. You can cancel at that point or now by clicking "No Thanks" below. Note: if you cancel, this application may not work properly. After you have authorized this application, you will be redirected back to this page.

Note: for all the rules in the examples in this paper that are selected by web pageviews, I've made the URL pattern as general as possible (.*). In a real ruleset, these would likely be much more restrictive.

The postlude of the rule (inside `fired {...}`) runs the `last` statement if the rule fires to ensure that nothing else happens. Of course, if the app is authorized, the rule doesn't fire, the OAuth ceremony is not initiated, the `last` statement is never executed, and the remaining rules in the ruleset are evaluated.

The **initialize then populate** pattern is important any time you're working with complex data. With complex data, you will frequently need to do something for each component of an array. That's what the `foreach` statement does as part of the rule selector: executes a rule once for each member of an array.

The problem is that if we use a `foreach` to loop over the tweets in the array and use `notify` to place them on the page, we'll end up with one notification box for each tweet…not very pretty.

A better solution is to use a rule to place the notification box (the initializer) and another rule to loop over the tweets and place them in the notification box (the populater).

Here's the initialization rule:

```
rule init_tweetdom is active {
  select using ".*" setting ()
  pre {
    init_div = <<
<div id="tweet_list">
</div>
>>
  }
  notify("Friends Tweets", init_div)
    with sticky=true and
         opacity = 1.0
}
```

Because KRL is a rule language, it has patterns and idioms that differ from languages like Java or C. Learning new patterns helps programmers get up to speed quickly.

This simple rule places an empty notification box on the page.

The real work is done by the populating rule:

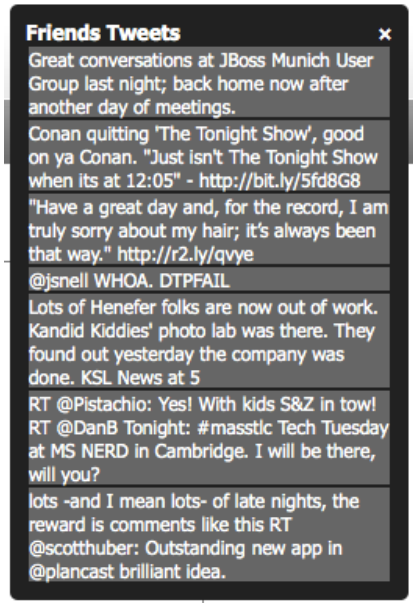
```
rule populate_tweetdom is active {
  select using ".*" setting ()
    foreach tweets setting (tweet)
      pre {
        text = tweet.pick("$..text");
        div = "<div>#{text}</div>";
      }
      append("#tweet_list", div)
}
```

This rule loops over the tweets using `foreach`, grabs data out of them using the `pick` operator and a JSONPath expression, and appends the result to the div called `#tweet_list` in the notification box.

The tweets variable was set in the global block:

```
tweets =
  twitter:authorized() =>
         twitter:friends_timeline({"count": 7})
       | [];
```

After you've gone through the OAuth ceremony at Twitter, wherever you run this app, you will see a box that contains the last seven tweets from your friends timeline on Twitter. Here is an example:



Great conversations at JBoss Munich User Group last night; back home now after another day of meetings.

Conan quitting 'The Tonight Show', good on ya Conan. "Just isn't The Tonight Show when its at 12:05" - http://bit.ly/5fd8G8

"Have a great day and, for the record, I am truly sorry about my hair; it's always been that way." http://r2.ly/qvye

@jsnell WHOA. DTPFAIL

Lots of Henefer folks are now out of work. Kandid Kiddies' photo lab was there. They found out yesterday the company was done. KSL News at 5

RT @Pistachio: Yes! With kids S&Z in tow! RT @DanB Tonight: #masstlc Tech Tuesday at MS NERD in Cambridge. I will be there, will you?

lots -and I mean lots- of late nights, the reward is comments like this RT @scotthuber: Outstanding new app in @plancast brilliant idea.

The ability to personalize apps by appealing to personal data elsewhere on the Web is an important feature in KRL.

Example: Remembering User Input and Explicit Events

This example shows how to gather, remember, and use user-supplied data. The pattern also uses explicit events to fire additional rules in response to user actions.

Entity variables allow KRL programs to use data across ruleset invocations. Explicit events allow rule chaining inside and between rulesets.

The basic idea is to store the data in an entity variable. Trails (a type of persistent variable) are the most appropriate type of entity variable to use. The ruleset pattern has four rules: initialize, send the form, process the form, use the data. The actual ruleset has five because I added one to delete the user data since it makes testing much more convenient. I'll go over each rule in order.

Forget: The first rule clears the entity variable `ent:name` when you visit a particular web page. You could, of course, do this under user control with a form submission or something, but this method is simple and suits our purpose.

```
rule clear_name is active {
   select when web pageview "www.foobar.com"
   noop();
   always {
      clear ent:name;
      last
   }
 }
```

Note that if the rule is selected (the page URL matches) then the entity variable is cleared and this is the last rule executed in this ruleset.

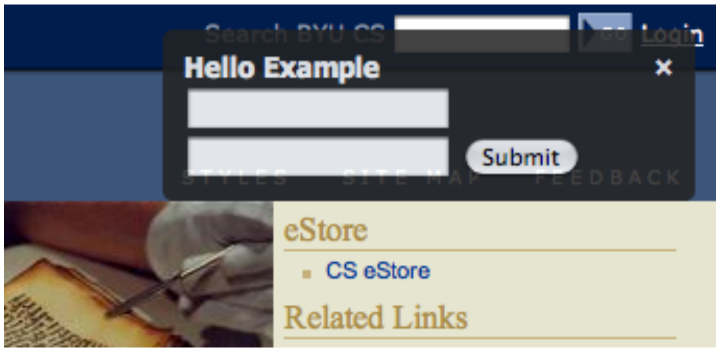
Initialize: I prefer to initialize the area of the page I'm going to write or modify and then do the modification in later rules since I can have different rules do different things to the area as needed. Consequently, the initialization rule just puts up an empty notification box with a div named `#my_div` that we'll use in later rules.

```
rule initialize is active {
  select when pageview ".*"
   pre {
    blank_div = <<
<div id="my_div">
</div>
    >>;
  }
  notify("Hello Example", blank_div)
      with sticky=true;
}
```

Send the Form: This rule puts the form into the div we initialized in the last rule if the entity variable `ent:name` is not empty. The rule also sets a watcher on that form so that an event is raised when the user submits it. If this rule fires (i.e. the rule is selected and the condition is true) then this will be the last rule executed in this ruleset since we just want to send the form.

```
rule set_form is active {
  select when pageview ".*"
  pre {
    a_form = <<
<form id="my_form" onsubmit="return false">
<input type="text" name="first"/>
<input type="text" name="last"/>
<input type="submit" value="Submit" />
</form>
    >>;
}
  if(not seen ".*" in ent:name)  then {
    append("#my_div", a_form);
    watch("#my_form", "submit");
  }
  fired {
     last;
  }
}
```

Whenever this rule fires, you get a notification box that looks like this:



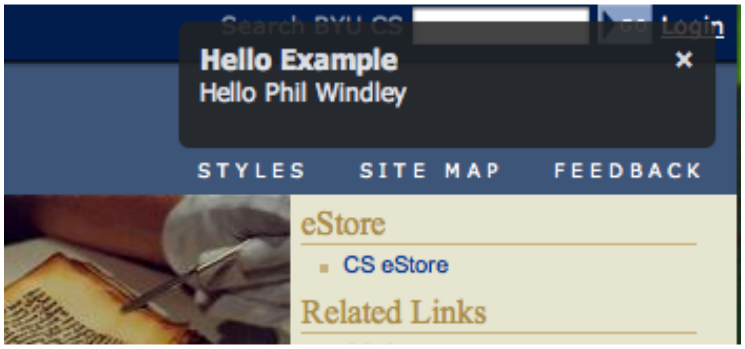
Process the form: We need a rule to process the form. This rule is selected when the form is submitted. The rule doesn't have an action (i.e. the action is noop). All the real work is done in the postlude where we store the name in the entity variable `ent:name` and then `raise` an explicit event that will cause another rule to be selected.

```
rule respond_submit is active {
  select when web submit "#my_form"
  pre {
     first = page:param("first");
     last = page:param("last");
  }
  noop();
  fired {
     mark ent:name with first + " " + last;
     raise explicit event got_name
  }
}
```

Use the data: The final rule uses the data in the entity variable to put the user's name in the div we placed in the initialization rule. This rule has two selection conditions. It can be selected on a pageview like the other rules or when an explicit event named `got_name` is raised. Remember that the previous rule raises that event in the postlude.

```
rule replace_with_name is active {
   select when explicit got_name
            or web pageview ".*"
   pre {
     name = current ent:name;
   }
   replace_inner("#my_div", "Hello #{name}");
}
```

The action replaces the contents of the div having the ID `#my_div` with a hello message that includes the name. When this rule fires, the notification box looks like this:



Whenever this ruleset is evaluated in the future the user will not see the form but simply see this box because the system remembers the name in the entity variable and has no need to ask for it again. If the data gets cleared, then the user is prompted to submit the form again.

This ruleset demonstrates the use of explicit events. If we don't raise the explicit event `got_name` in the rule `respond_submit`, nothing will replace the contents of the notification box to show the user that the form submission was successful. We could have done it in that rule, but then we'd have two rules replacing the contents with a message and if the message changed we'd have to make sure we changed it in two places. This technique allows us to have one rule responsible for putting the hello message in the div. We just select it under two different circumstances.

Example: An Echo Endpoint

As we have mentioned, KNS allows developers to create their own endpoints and define event domains, event types, and directives for that endpoint. While most developers use the pre-built endpoints and pre-defined domains (e.g. `web` and `email`), seeing how endpoints and KRE interact is instructive. This example shows the creation of a simple endpoint and a ruleset to interact with it.

KNS is extensible so the developers can create their own endpoints and events to go with them. This flexibility makes KNS the first Internet application platform.

In this example, the developer's job is to

1. invent the events that the endpoint will raise
2. design the directives that the endpoint will respond to
3. create an endpoint that does these things
4. write rulesets that the endpoint uses.

Defining events that an endpoint can raise and directives that it can consume is similar to creating a protocol. The quintessential introductory example for a protocol is an echo server. For this example, we will define a simple event domain, `echo`, and two event types, `hello` and `message`.

Of course, defining the events is only half of the game. We need to respond to them. The `send_directive` action provides a general way for developers to send structured information (JSON) to an endpoint.

For example the following action will send a directive named `say` with a parameter named `something` that has the value "`Hello World`".

```
send_directive("say") with
```

```
          something = "Hello World";
```

A ruleset may send zero or more directives to the endpoint as the result of a single event being raised. The endpoint can interpret them any way it wants.

In our ruleset, we'll define two rules: one for each event type given above. We could, or course, have more if we want to respond to different parameters.

```
rule hello_world is active {
  select when echo hello
  send_directive("say") with
    something = "Hello World";
}

rule echo is active {
  select when echo message input "(.*)" setting(msg)
  send_directive("say") with
    something = msg;
}
```

The rule `hello_world` responds to the `hello` event by sending the directive named `say` with the parameter `something` set to "`Hello World`".

The rule `echo` responds to an `echo` event with a parameter called `input`. That entire value of the input is captured and bound to the variable `msg`. The `echo` rule sends a directive named `say` with the parameter `something` set to the value of `msg`.

It's critical to note that the underlying Kynetx rules engine doesn't know anything about the event domain `echo` or the event types `hello` and `message`. We could define these to be anything we wanted and the example would work the same.

This ruleset, with its understanding of the `echo` events and directives, is useless without a corresponding endpoint to raise these events and consume the directives. Appendix A contains a Perl program that functions as a simple endpoint for this event domain.

The program has the possibility of taking the event type from the command line with the -e switch. If none is given, the event type defaults to `hello`. Consequently running this program with no arguments results in the `hello_world` rule firing and the directive `say` "`Hello World`" being sent to the program that prints the message.

Running the program with the -e switch like so:

```
./echo.pl -e message -m 'KRL programs the Internet!'
```

results in the string "`KRL programs the Internet!`" being echoed to the terminal.

Try Out KRL

You can try out KRL for free by creating an account at Kynetx[16]. Free Kynetx accounts can develop multiple applications and run them for non-commercial use. Inquires about partnerships and VAR sales should be directed to the email in the contact information at the end of this paper.

Summary

The Kynetx Rule Language provides a powerful notation for programming the Internet. The Kynetx Network Service provides a platform that brings that notation to life and makes it actionable. Together they give developers the means to create applications that span multiple protocols, domains, and systems and create entirely new kinds of applications. These apps take into account user context and other data available on the Internet.

KRL and KNS have events built-in. They are flexible enough to respond to any Internet protocol or domain. KNS is the first Internet application platform—a system for programming the Internet.

Lexicon

action - the effects that a rule has on the calling page are called actions. A rule may have more than one action. Depending on the structure of the compound action, all of the actions or one chosen at random may be taken.

condition– the predicate phrase that is evaluated to determine whether the rule fires. When a rule fires, the consequent, containing actions, is evaluated. The condition may be empty. In that case the rule *always* fires when it is selected.

consequent - the statement of the effect a rule will take.
A consequent is made up of one or more actions. When the consequent is evaluated, directives are sent to the endpoint.

context –the information surrounding and influencing ruleset execution. Much context is user specific.

endpoint - the software or device that is responsible for raising salient events to KNS and mediating the interaction with the client including responding to KNS directives to the client

event — notification of an activity that took place at a particular time with specific attributes. Usually endpoints raise events, although some rules may raise explicit events in order to elicit further action from KNS.

event scenario — a meaningful group of events.

fire - a selected rule is evaluated to determine if the predicate in its condition is true. If it is, the rule is said to have "fired." When a rule fires, its actions are evaluated.

rule - a unit of computation in KRL. A rule says what actions should be taken when the rule is selected and its condition is true. Each rule has a rule name that should be unique within the ruleset.

ruleset—a collection of rules in KRL. Rulesets also have meta data and global declarations. Rulesets are uniquely identified by ruleset ID, or RID.

ruleset evaluation – (also RSE). KNS evaluates rulesets in response to events raised by the endpoint. When an event is raised, the event expression in the select statement for each rule in the ruleset is evaluated. When the event expression is met, the rule is scheduled for evaluation. For each active rule selected by a satisfied event expression, the rule condition is evaluated and if true, the rule is fired. For billing purposes, a ruleset is only considered to have been evaluated when one or more rules in the ruleset fire (called a BRSE).

ruleset id—also *RID*, the ruleset ID is the unique identifier for a ruleset.

salience –events are only raised by and endpoint if they are meaningful to the rulesets that the endpoint is tracking.

selected - Each rule contains a selector statement that contains an event expression. When the event expression is met or satisfied, the rule we say that the rule has been "selected." The selected rules are typically a small subset of the total number of rules in the ruleset. Selected rules are scheduled for further evaluation. See *fire* and *ruleset evaluation*.

Appendix A: A Perl Endpoint

This Perl program functions as a simple endpoint for this event domain shown in the echo example:

```
#!/usr/bin/perl -w
use strict;

use Getopt::Std;
use LWP::Simple;
use JSON::XS;

use Kynetx::Raise;

# global options
use vars qw/ %opt /;
my $opt_string = 'h?e:m:';
getopts( "$opt_string", \%opt ); # or &usage();

my $event_type = $opt{'e'} || 'hello';
my $message = $opt{'m'} || '';

my $event = Kynetx::Raise->new('echo',
                               $event_type,
                               'a16x66',
                               {'host' => '127.0.0.1'}
                               );

my $response = $event->raise({'input' => $message});

foreach my $d (@{$response->{'directives'}}) {
  if ($d->{'name'} eq 'say') {
    print $d->{'options'}->{'something'}, "\n";
  }
}
```

This simple script uses a module called `Kynetx::Raise` that takes the relevant information about the event, create the right URL for the Kynetx event API, raises the event by calling the URL and processes the response.

Contact Information

Kynetx, Inc.
World Headquarters
3098 Executive Parkway
Suite 325
Lehi, UT 84043
(801) 649-4601
www.kynetx.com

Endnotes

1 Kynetx anticipates building endpoints for many standard Internet systems and protocols. Our product roadmap for the next 6 months includes mobile endpoints.

2 See the Event API documentation: (http://docs.kynetx.com/kns/kynetx-network-services-kns/#Event)

3 At present, directives can only be sent to the endpoint that raised the event. Future versions of KNS will allow directives to be sent to all of the endpoints cooperating on behalf of an entity.

4 For clarity, I have ignored compound event expressions in this syntax.

5 In the case of the Web, the endpoint is a combination of the browser extension, the browser, and a Javascript runtime library supplied by Kynetx. Browser extensions and proxies initialize the endpoint by loading that runtime library. This shows the power of endpoints that understand Javascript: by updating the runtime library we update the functionality of every endpoint that uses it.

6 For historical reasons, event expressions for the web domain have a slightly different syntax than what is shown above. See the KRL documentation for more details (http://docs.kynetx.com/krl/report-on-krl/rules/#Rule_Selection)

7 For more information on regular expressions, see…

8 Much research on event expressions was done in the area of active databases in the 90's. The ideas developed for specifying and implementing complex event expressions in databases serve nicely for the kinds of things we do in KRL. A bibliography of this research is available upon request.

9 You might be wondering how we can efficiently compute compound event scenarios. The key is that event expressions are actually no more powerful than regular expressions over the alphabet of primitive events. We compile each event expression into a corresponding state machine as part of the ruleset optimization process. Then we simply keep a state marker for each rule in each ruleset that each user has installed. The storage requirements aren't that great and reacting to an event only requires looking at the user's current state for a rule and calculating the next state. If the state machine is in a final state the rule is selected and the state machine reset. The exact details of how event expressions are compiled to state machines are left as an exercise for the reader.

10 In particular, we anticipate the need for compound event operators that enable temporal comparisons such as "event A occurred within 5 days of event B happening" or "event A happened on Tuesday before 5pm" and so on.

11 See the Kynetx Report "Context Automation" for a more detailed discussion of context.

12 A full list of the actions available for the Web are given in the KRL documentation (http://docs.kynetx.com/krl/kynetx-rule-language-documentation/actions/)

13 KRE optimizes rules so that only the parts of the body that depend on the value set by the loop are repeated. Portions of the rule body that are not dependent on that value are automatically moved outside the loop for efficiency.

14 More information on FLWOR (pronounced "flower") statements can be found on Wikipedia (http://en.wikipedia.org/wiki/FLWOR)

15 For more information about KRL libraries see the documentation (http://docs.kynetx.com/krl/report-on-krl/libraries/)

16 See http://www.kynetx.com/signup