

Note: The topic of this post, creating a blog, is dependent on features in the KRE-version of the Pico Engine, not the new Pico Engine. Nevertheless, the ideas in this chapter about event programming and even hierarchies (really, graphs) are useful.

Chapter 12. Advanced KRL Programming

So far we've seen rulesets that work in a variety of domains and in a variety of circumstances, but all of the examples have been small, involving just a single ruleset of a few rules. The event interactions have been simple. In this chapter we'll tackle applications that require multiple rulesets and more complicated event hierarchies.

Knowing how to build applications in KRL using multiple rulesets opens up the possibilities of what you can build on the Live Web. Along the way we'll come to understand event hierarchies and normalization as well as some programming principles that support loosely coupled applications where functionality can be added without significant changes to the underlying parts.

In this chapter we'll build a blogging tool using KRL. Building a blog with KRL is easier than you'd think and departs in several significant ways from the examples we've explored previously in this book.

A Blog in KRL

The blog that we'll create will use four rulesets:

1. `KBlog Configuration`—a module that is used to configure the application and hold common definitions.
2. `KBlog`—the main ruleset that controls the HTML assets and presentation. This ruleset functions like the presentation layer of a traditional Web application, responding to user input and displaying results.
3. `KData`—the ruleset that manages the blog data. This ruleset manages the data assets and provides access rules. At first we'll use application variables to store the blog data as a simple way of getting started. In a second instance we'll use online storage to manage the data.
4. `KBlog Posting`—we separate posting from the other parts of the application so that only people who had access to this ruleset

can post. This is not a good solution to controlling access to blog posting, but it is sufficient for demonstration purposes.

The following sections will explore each of these rulesets in turn. Later in this chapter, we'll add a fifth to show how loose coupling supports application extension.

Configuration

KBlog Configuration provides several common definitions for variables, functions, and actions that are used in the other rulesets. KBlog Configuration is built as a module since other rulesets will access the definitions it contains.

The variables declared in the module include `blogtitle` that holds the string that will be used as the blog's title. The various rulesets use `blogtitle` to construct page titles.

The right hand column of each blog page contains some text describing the blog. KBlog Configuration provides a variable called `about_text` to store this text as well even though it's only used in one place. Having it in the configuration module allows the users to go to just one ruleset to change the various, fixed textual elements of the blog.

KBlog Configuration also defines two actions, `paint_container` and `place_button`. We will describe their function later. The `provides` pragma in the meta section of the ruleset makes each of these available to the other rulesets in the KBlog application.

```
provides blogtitle, about_text, paint_container, place_button
```

The other rulesets in the application use the configuration module with the `use module` pragma in their meta section:

```
use module a16x93 alias config
```

Building the Blog

The primary job of the KBlog ruleset is presenting the blog. KBlog does this by preparing the container for the blog articles and then filling it with the articles.

The blog is built in a style that's called single page interface or SPI¹. In SPI-style Web applications, the basic framework of the application, the HTML, CSS, and JavaScript libraries are loaded in an initial call and then partial changes to the page are made incrementally via JavaScript AJAX calls.

Sites like Google, Twitter, Gawker, and others made this style of Web site construction popular. The chief advantage is a faster user experience as a consequence of reduced bandwidth from not requiring that things like the HTML and CSS be downloaded each time. KRL is a natural way to build sites that use this style of construction.

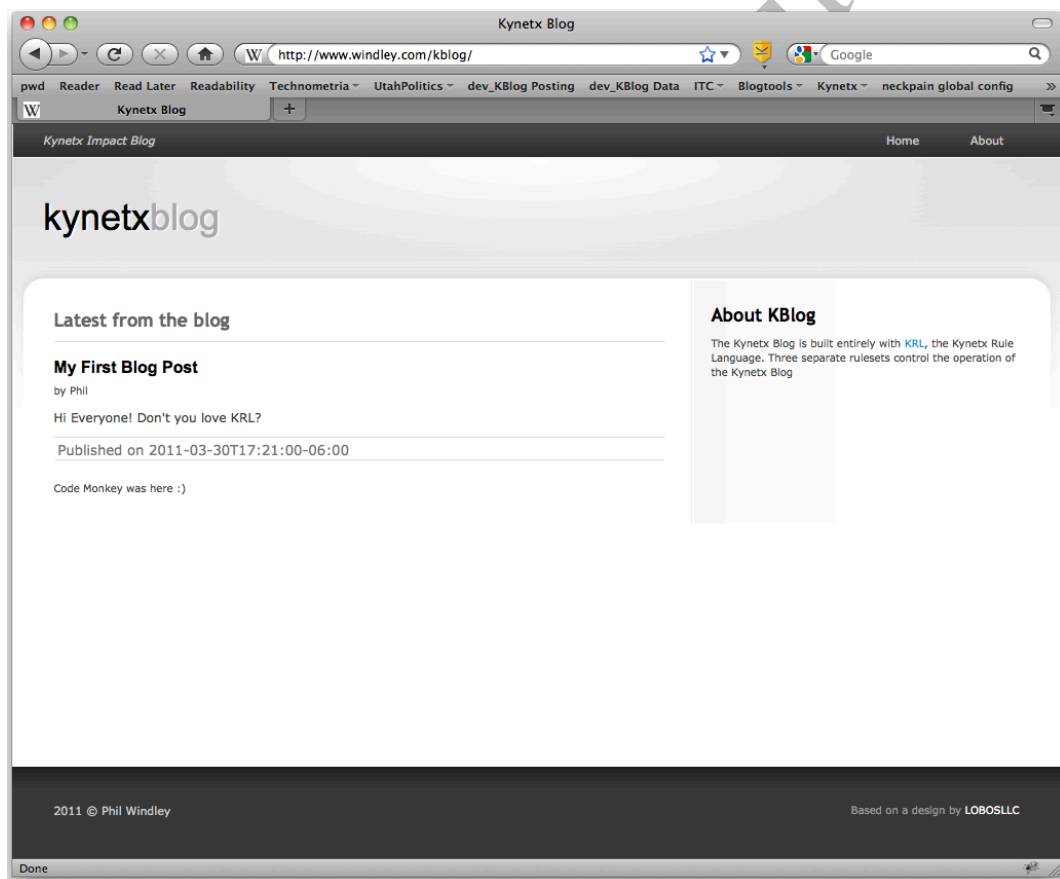


Figure 1. A blog written in KRL

¹ See http://en.wikipedia.org/wiki/Single-page_application for a complete description of the principles behind single page interface Web applications.

The DOM

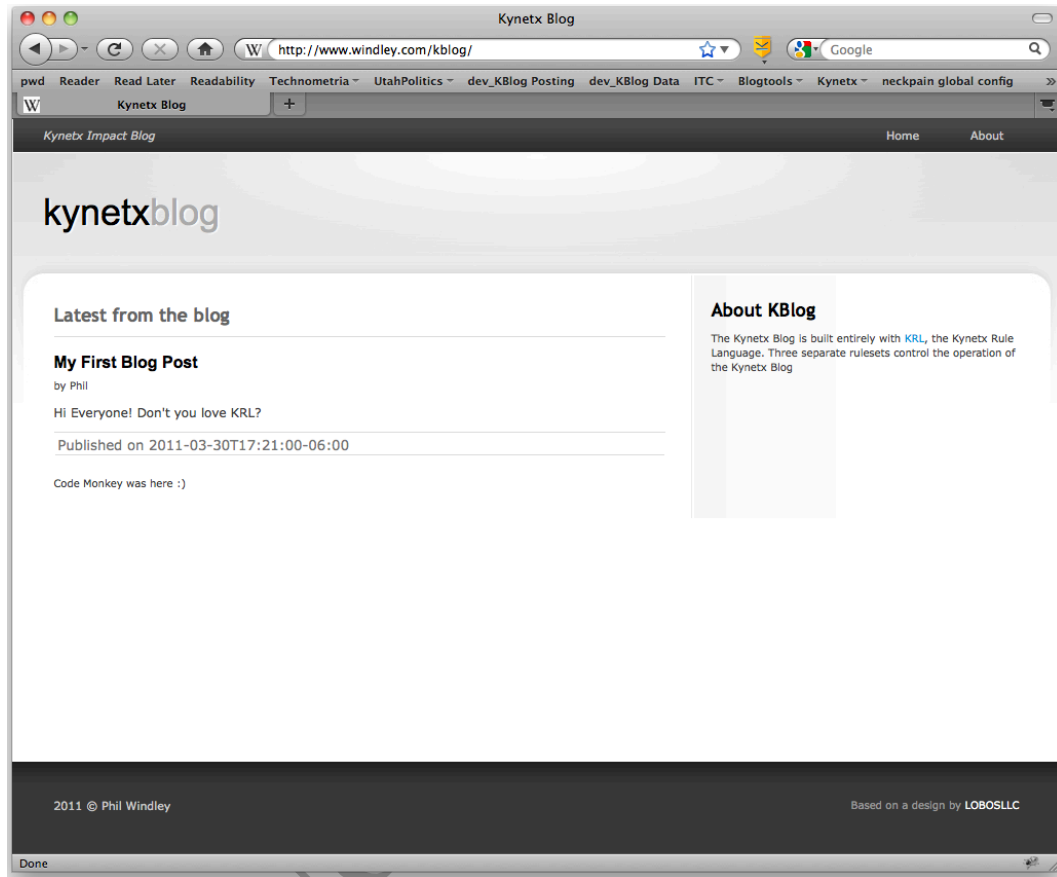


Figure 1 shows the basic layout of the site. A navigation bar at the top contains links that can be used to navigate between different parts of the site: the home page and contact page. The DOM for the navigation bar is initially empty:

```
<nav id="sitenav">
  <ul id="navlist">
  </ul>
</nav>
```

We'll use rules to place buttons in the navigation bar as appropriate.

The primary section of the blog—the left container—is where page content is written. Initially it is empty and has the following structure:

```
<div id="leftcontainer"></div>
```

The information section of the blog—the about section on the right—is also initially empty:

```
<div id="sidebarwarp">
  <h2>About KBlog</h2>
  <p id="about"></p>
</div>
```

Unlike other KRL applications we’ve built in previous chapters, the blog application isn’t triggered by an event from the endpoint. Instead we’ll plant tags in the DOM that trigger the event and cause the blog application to execute. That way, anyone—not just people with an endpoint installed—can see the blog. Here are the JavaScript `<script/>` tags that we place in the page:

```
<script type="text/javascript">
  KOBJ_config= {"rids":["a16x88"],
               "a16x88:kinetic_app_version":"dev",};
</script>
<script src="http://init.kobj.net/js/shared/kobj-static.js" />
```

The first JavaScript tag sets of the configuration variable to identify the ruleset IDs (RIDs) for the KBlog ruleset (a16x88 in this case). The configuration also sets the `kinetic_app_version` configuration variable to “dev” indicating that the development (most recent) version and not the production (last deployed) version of the ruleset should be executed. This ensures that we don’t have to continually deploy the ruleset as we are testing. Later, when the application is finished, we’d either change the value of this variable to “prod” or simply delete it since “prod” is the default. Note that the variable is namespaced using the RID.

The second `<script/>` tag loads the Web endpoint. The endpoint will read the configuration we set up and initiate a `pageview` event for this page. Normally this is all done by a browser extension, but it works just as well to put the `<script/>` tags in place directly for an SPI Web application.

Even though we’re placing the tags in the HTML of the page directly, that doesn’t change the method we’ll use to construct KRL rulesets. The structure and functionality of the application follow from the desired behavior, not the method that is used to raise events.

The Rules

The rules in the KBlog ruleset handle the content of the page. As noted earlier, the overall structure of the page and the CSS stylesheet are loaded when the user goes to the URL for the blog. The `<script/>` tags that we discussed in the previous section initiate a pageview event.

The `init_html` rule is selected when the pageview event is raised. This rule is responsible for painting the “about” text on the right side and setting up the navigation buttons. `init_html` is the only rule in the entire application that is selected on a pageview event. The selector is very general because it should fire whenever the blog is viewed.

```
rule init_html {
  select when pageview ".*" setting ()
  {
    replace_html("#about", config:about_text);
    config:place_button("Home");
    config:place_button("Contact");
  }
  always {
    raise explicit event blog_ready
  }
}
```

`init_html` places the buttons using an action, `place_button()`, from the configuration module. `Place_button()` takes a name as it's only parameter, creates the correct HTML for the button using the name, places it in the navigation bar, and attaches a watcher to the button so that any clicks will raise events:

```
place_button = defaction(button_name) {
  id = "siteNav" + button_name;
  label = button_name;
  button = <<
<li><a href="javascript:void(0);" id="#{id}">#{label}</a></li>
  >>;
  {
    prepend("#navlist", button);
    watch("#" + id, "click");
  }
}
```

`Init_html` raises the explicit event `blog_ready` to indicate that the blog is ready to be populated with the page content. There are two pages in this example: the home page showing the blog articles and the contact page that has contact information.

`Show_home` is selected when one of two events occurs: someone clicks the home link on the navigation bar or there is an explicit `blog_ready` event (note this makes the home page the default). The `show_home` rule sets up the container to receive blog posts and sets the title:

```
rule show_home {
  select when web click "#siteNavHome"
    or explicit blog_ready

  pre {
    container = <<
    <h2 class="mainheading">Latest from the blog</h2>
    <div id="blogarticles">Code Monkey was here :)</div>
    >>;
    title = config:blogtitle;
  }
  config:paint_container(title, container);
  always {
    raise explicit event container_ready;
    raise explicit event need_blog_data for a16x89
  }
}
```

The `show_home` rule makes use of a user defined action, `paint_container()` from KBlog Configuration:

```
paint_container = defaction(title, container) {
  {replace_inner("title", title);
  replace_inner("#leftcontainer", container);
  }
}
```

`Paint_container` is also used by the `show_contact` rule that puts up the contact page when someone clicks on that link. A user-defined action ensures that we do this the same way both times.

`Show_home` raises two explicit events, one that indicates that the container is ready and another that says that data is needed. We'll discuss

the rules that respond to the latter event in the next section, but ultimately, the result of that process is that the explicit `blog_data_ready` event is raised.

Putting the actual blog articles in the container is the job of a rule named `show_articles`. This rule fires when the `container_ready` and `blog_data_ready` events are raised. Both events must occur before we place the articles on the blog, but the order is unimportant.

The rule loops over each member of the hash representing the blog data, formats the correct HTML, and inserts it into the page:

```
rule show_articles {
  select when explicit container_ready
    and explicit blog_data_ready
    foreach event:param("blogdata") setting (postKey, postHash)
    pre {
      postArticle = <<
<article class="post">
  <header>
    <h3>#{ postHash.pick("$.title") }</h3>
    <span class="author">by #{ postHash.pick("$.author") }</span>
  </header>
  <p>#{ postHash.pick("$.body") }</p>
  <footer>
    <p class="postinfo">
      Published on <time>#{ postHash.pick("$.time") }</time></p>
    </footer>
  </article>
  >>;
    }
    prepend("div#blogarticles", postArticle);
}
```

By initializing the blog, placing the page container, and then filling it in with separate rules, we build the blog in pieces. Navigation actions simply update the portion of the structure that is changing, leaving the rest unchanged. In high volume Web sites, this can amount to a considerable savings in bandwidth and increase the user's perception of application responsiveness.

Handling Data

One of my goals in creating the demonstration blog was to separate the handling of data from the rules that performed presentation. Certainly that's good design, even if all the rules are in the same ruleset. The event hierarchy is more interesting and we can better explore loose coupling if we put the data handling rules in a separate ruleset.

For this example, we'll make use of KRL's built-in persistent storage to keep the blog post data. In a production blog, of course, we'd use a database, not persistent variables. Using persistents as a substitute for a database, can speed prototyping, but they are not designed for the kind of heavy-duty data operations that databases are.

But given that we're using persistent variables, application variables are the right construct for our goals because every visitor will see the same set of blog posts. If we used entity variables, every visitor to the blog would see their own posts and no others.

The structure of the application variable will be a map of maps with each entry in the map representing a blog post. Each blog post has an author, title, body, and timestamp. We define a function that formats the map representing a single blog post like so:

```
mk_article = function (author, title, body) {
  postTime = time:now({"tz":"America/Denver"});
  { postTime : {"author" : author,
               "title"  : title,
               "body"   : body,
               "time"   : postTime
             }}
}
```

The function calculates the timestamp and uses it as a key for the map as well as placing it in the blog entry map.

The only place where we can mutate a persistent variable is in the rule `postlude`, so if we are going to use persistents to store blog article data, we have to use a rule to process the data. `Add_article` watches for articles, formats the data with the `mk_article` function, and adds it to the app variable `BlogArticles`:

```
rule add_article {
  select when explicit new_article_available
  pre {
```

```

    post = event:param("post");
    postHash = mk_article(post.pick("$.postauthor"),
                          post.pick("$.posttitle"),
                          post.pick("$.postbody"));
    articles = app:BlogArticles || {};
  }
  always {
    set app:BlogArticles articles.put(postHash);
    raise explicit event new_article_added for a16x88;
  }
}

```

Note that this rule takes no action. The benefit is all in the effects. Whenever a new article is available, `add_article` will use the post data from the event parameter to format a new entry and put it in the application variable `BlogArticles`. The disjunction operator in the declaration of `articles` ensures that `articles` will be defined as an empty map if `BlogArticles` is undefined. Otherwise, the `put` operator won't function correctly.

When `add_article` fires, it raises an explicit event to indicate a new article has been added. The `KBlog` ruleset contains an intermediary rule, `show_new_article` that transforms this event into a `blog_ready` event. As we saw above, the `show_home` rule is listening for the `blog_ready` event and this causes the screen to be repainted.

```

rule show_new_article {
  select when explicit new_article_added
  noop();
  always {
    raise explicit event blog_ready
  }
}

```

We saw in the preceding discussion that the `show_home` rule raises an explicit event `need_blog_data`. This is about as close to a request as we get in this example. A request would direct a specific function to return the data. The event merely says that data is needed. The `show_home` rule doesn't know who will respond.

The `retrieve_data` rule selects on the explicit event `need_blog_data` and raises the explicit event `blog_data_ready`, attaching the blog data from the application variable as an event attribute:

```
rule retrieve_data {
  select when explicit need_blog_data
  noop();
  always {
    raise explicit event blog_data_ready for a16x88
    with blogdata = app:BlogArticles || []
  }
}
```

Posting

All that's left to complete the application is adding the ability to post. As explained earlier, this functionality is in a separate ruleset and layered on top of the basic functionality of the blog. Most users will never need the functionality and won't see it. Only posters need access the rules that enable posting.

The first order of business is to add a navigation button to the bar at the top of the page exposing the functionality. The `place_button` rule fires on a `pageview` event, puts the button in place, and makes it active by attaching a watcher to it using the `place_button()` action we defined earlier:

```
rule place_button {
  select when pageview "kblog"
  config:place_button("Post");
}
```

Note that, as before, we've aliased the configuration module as `config`.

When someone clicks on the Post button, the Web endpoint will raise a click event and the `place_form` rule will fire. `Place_form` creates a form, places it on the page by replacing the left container, and attaches a watcher to the Submit button (some of the HTML in the form has been removed for brevity):

```
rule place_form {
  select when web click "#siteNavPost"
  pre {
```

```

    form = <<
<h2 class="mainheading">Post</h2>
<article class="post">
  <form method="post" class="form" id="blogform">
    <p class="textfield">
      <label for="postauthor"><small>Name</small></label>
      <input name="postauthor" tabindex="1" type="text"></p>
      ...
    <p><input name="submit" type="image" src="submit.png"></p>
    <div class="clear"></div>
  </form>
</article>
    >>;
    title = config:blogtitle + "- Post";
  }
  {
    config:paint_container(title, form);
    watch("#blogform", "submit");
  }
}

```

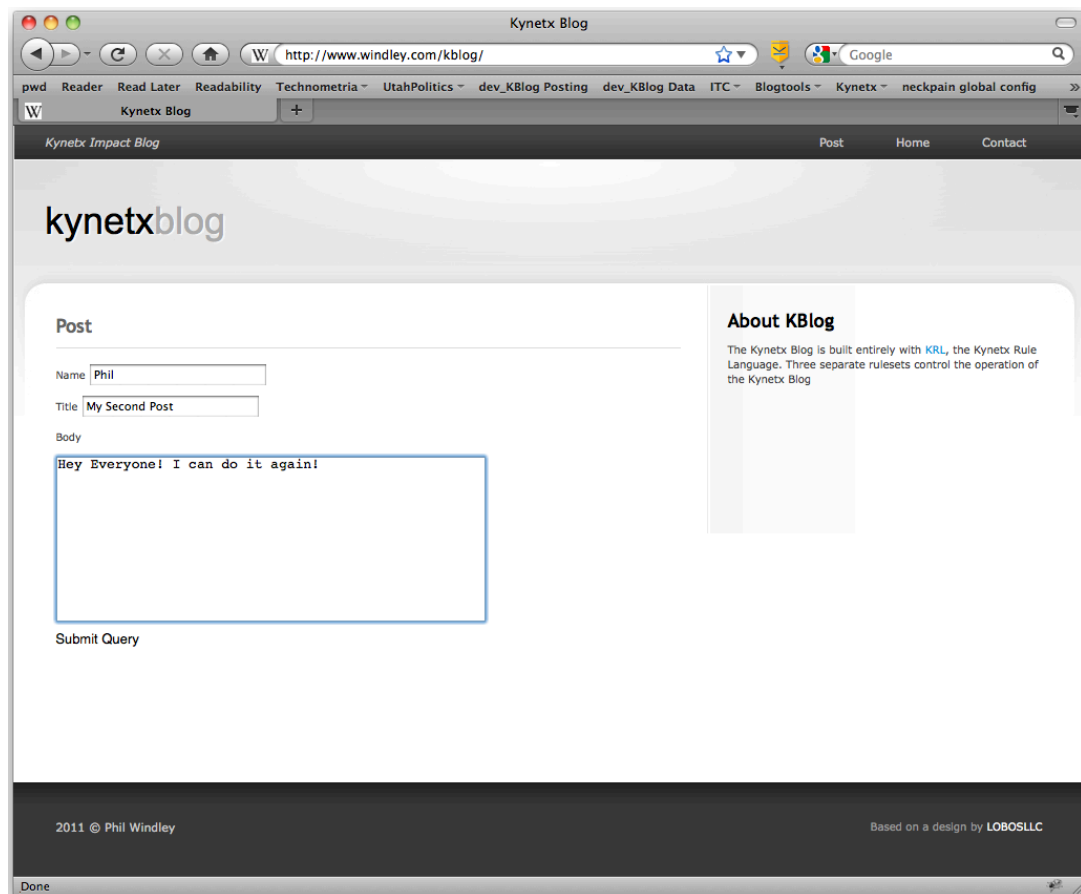


Figure 2 shows the result of this rule firing: the form is painted on the page so the user can fill it out.

When the user clicks on the submit link on the form in

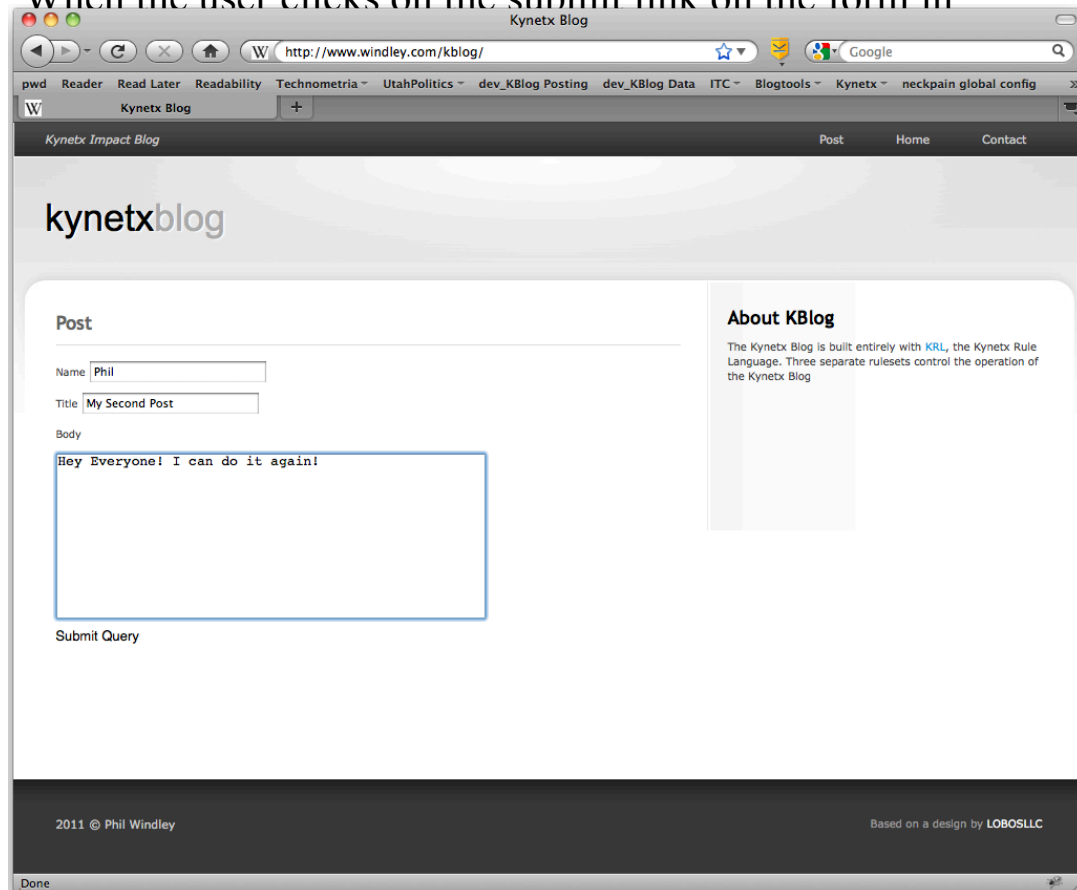


Figure 2, the Web endpoint raises a submit event. The `handle_submit` rule handles that event:

```
rule handle_submit {
  select when submit "#blogform"
  always {
    raise explicit event new_article_available
      for ["a16x89", "a16x91"]
      with post = event:attrs();
  }
}
```

`Handle_submit` raises the `new_article_available` event with the data from the form. As we've seen, the `new_article_event` is handled by `add_article` from the `KBlog Data` ruleset.

With the addition of the KBlog Post ruleset, we now have a complete, albeit simple blog application. The blog owner can post articles and control the content on other pages. Visitors to the blog see the articles that the owner has posted.

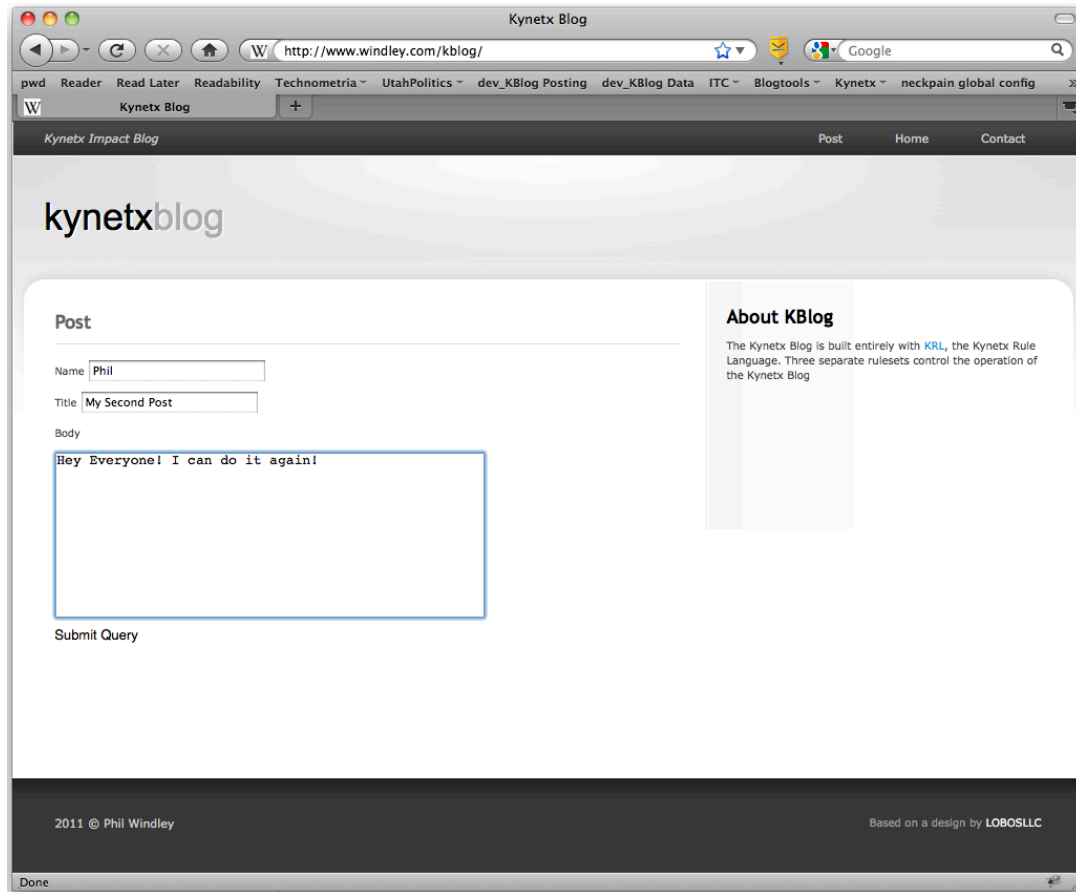


Figure 2. The blog posting form is painted on the blog in place of the blog articles

Event Hierarchies

Building an application with three rulesets is quite a bit more complicated than other examples we've seen. Keeping track of everything that's happening in your head can be difficult—especially since the programming model is likely unfamiliar. One tool we can use to help with the design of the application is called an event hierarchy. An event

hierarchy traces the events through the rules to see the causal relationships.

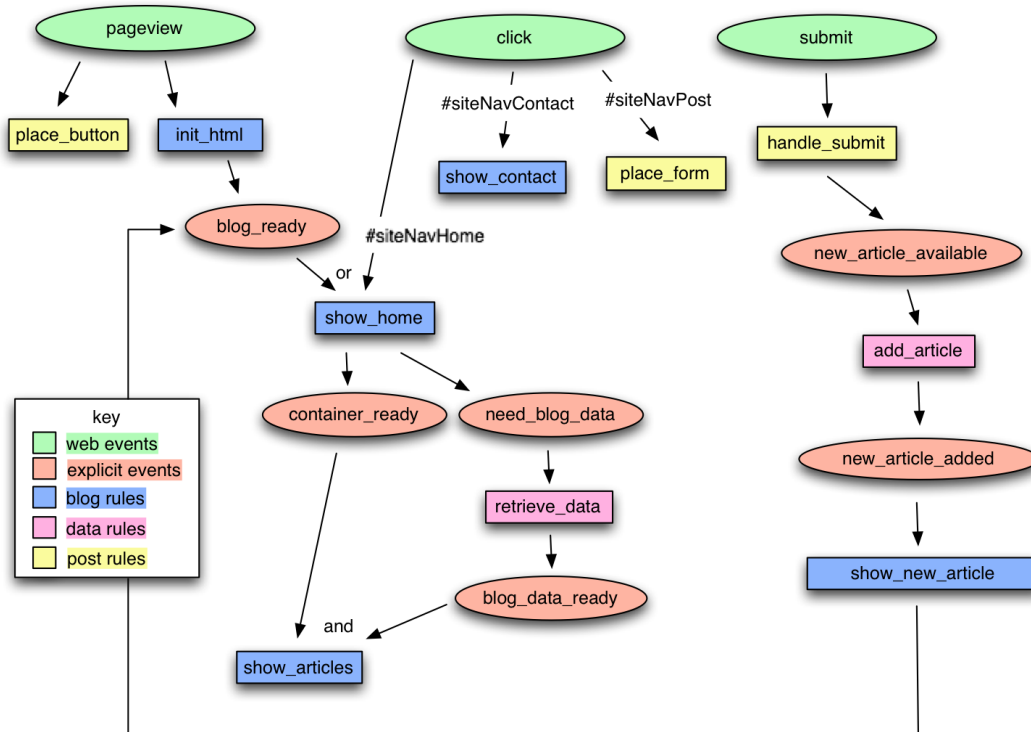


Figure 3 shows the event hierarchy for the KBlog application. In the graph, rectangular boxes are rules. They are named as verbs. The ovals represent events. They are nouns.

Getting the right names and the meanings for events is important. This is called “event normalization.” Thinking of events as nouns and rules as verbs is a useful way to keep your design straight. If you find yourself naming events with verbs, you’re probably not really creating an event-driven application. Rather you’re using events to make a request.

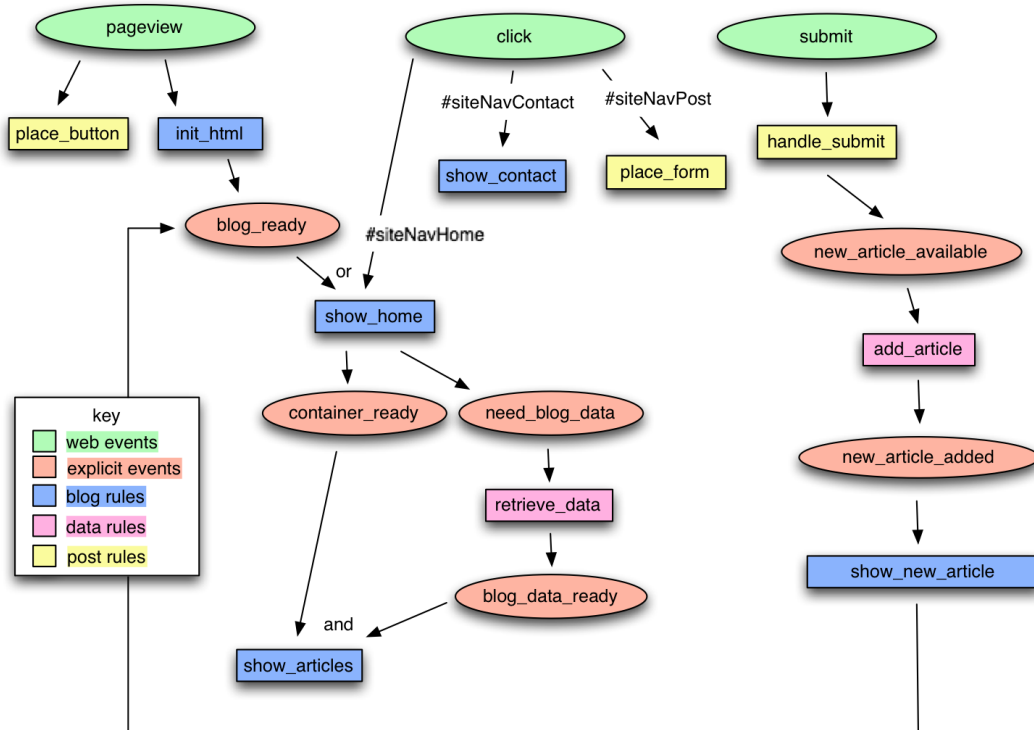


Figure 3. The event hierarchy for the KBlog application

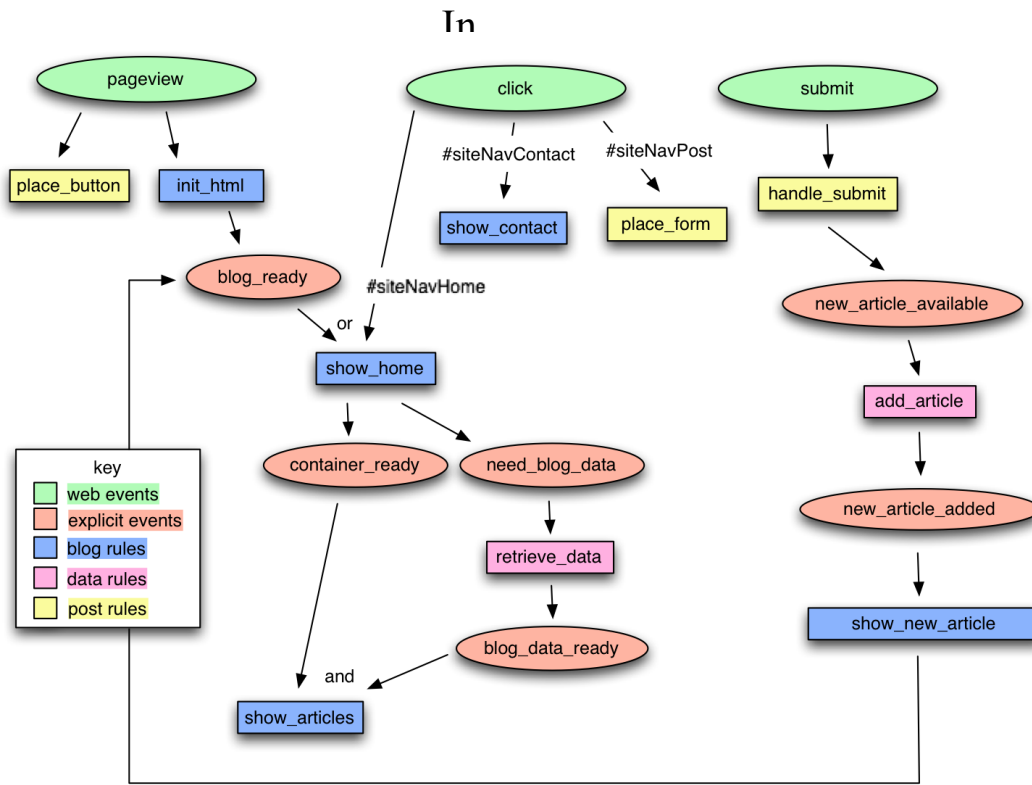


Figure 3, the Web events are the entry point for any path through the graph. There are three Web events: pageview, click, and submit. The click events are differentiated by the name of the element that was clicked on.

Once the event causal chain is initiated from a Web event, the rest of the event hierarchy consists of explicit events raised in the rules as indicated in the graph. Rules that don't produce events are terminals in the graph.

Understanding which rules were listening for which events and how they interact is the key design space. The event hierarchy plays an important role in understanding these interactions. Once that is done, writing the rules is relatively simple.

Looking at the longest chain, from the Web submit event, we see that submitting a blog post results in seven events being raised and six rules firing. This might seem inefficient at first, but we could say the same thing about control flow in a traditional programming model that uses multiple objects or functions.

You might be tempted to think of the event hierarchy as a kind of state diagram, but that's not quite right—the transitions don't happen because of an input. The events themselves are the input. Events are most useful

when they are moving and so there's no notion of “stopping” at some point in the diagram. Once you enter at the top, you flow through to one of the terminal rules.

Design Considerations

One of the objectives of the KBlog application was to create a design where new functionality can be layered on to the base. The KBlog Posting ruleset shows how this works. People who are posting never use that ruleset. Rules in KBlog Posting affect the rest of the application through the use of events.

We'll discover later in this chapter that we haven't gone far enough in raising explicit events. Good rule design might dictate that every rule raise an event when it's done—no terminal rules. Remember, events that don't have a listener are simply ignored. For example, the `show_articles` rule is terminal in the event hierarchy in

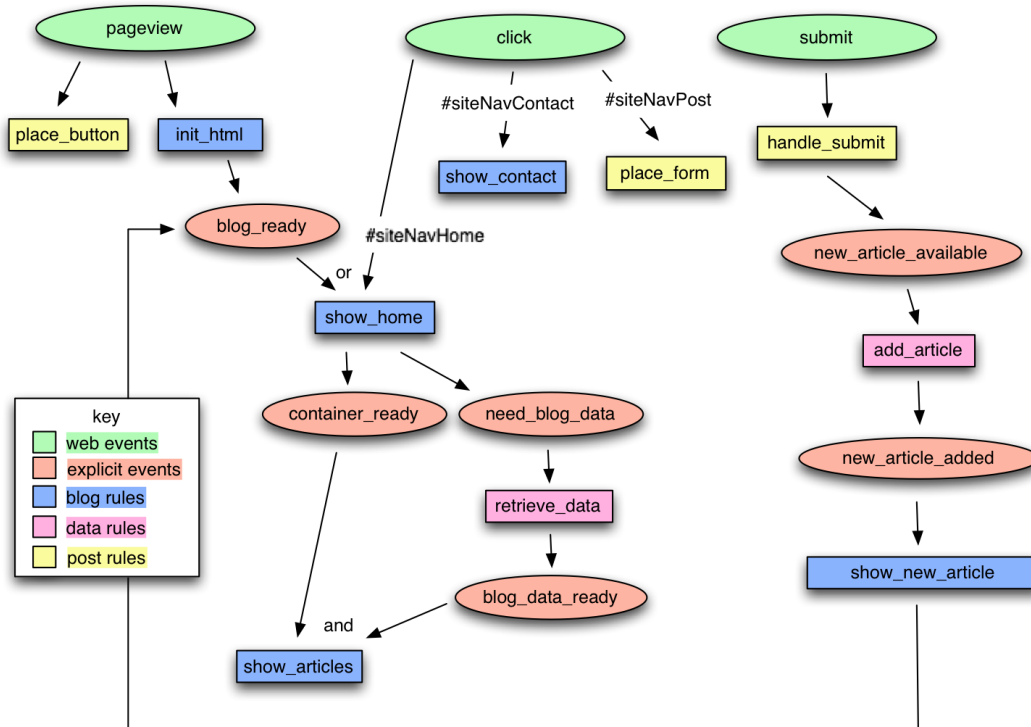


Figure 3. At that point, the blog articles are showing, but there is no event to notify other interested parties of that change in status of the blog.

Nothing in the rulesets we've built would use such an event, but what if someone wants to layer functionality onto this application later in a loosely coupled way? They might want to know when the articles are ready so they can take the next step—whatever that is. Any given rule, terminal now, may not always be so as new functionality is envisioned.

One objection to the design might be that the controller logic that is not clearly delineated in some kind of controller module. For this application the controller is in the various `select` statements. Using more intermediary rules in a controller ruleset could bring the controller logic for the application together, but finding and understanding the `select` statements is easy.

Making the Back Button Work

One issue with the current implementation of the `KBLog` application is that the back and forward buttons in the browser don't work. That's a big problem because many people use them to navigate.

Browser back and forward buttons operate by moving the user around in the browser's history. The browser doesn't know there's a new page and put it on the history unless the URL changes. SPI Web applications get around this by writing URL fragments (the stuff after the `#` symbol) to the URL of the page as the application state changes. Adding a fragment doesn't cause the browser to reload the page because fragments are designed for interpage navigation.

Not only does the application have to rewrite the URL when the application state changes, but it also has to change the application state when the fragment changes. Otherwise the back button will change the URL, but not the page. We'll handle both of these problems in turn.

Change the URL Fragment

There's currently no action in KRL to rewrite the URL. Fortunately KRL rules can emit JavaScript as an action, giving them the ability to flexibly adapt to situations that were not anticipated in the language.

The rules that rewrite the container (`show_home` and `show_contact`) cause a change to the application state and should therefore rewrite the

URL. By deliberately not including `place_form`, the rule that displays form, we keep it out of the history, avoiding the problem of back buttons and form posting.

We'll add a user-defined action to the `KBlog Configuration` module to emit JavaScript that modifies the fragment²:

```
update_frag = defaction(name) {  
  emit <|  
    self.document.location.hash='!#{name}';  
  |>;  
};
```

Remember that we also need to update the `provides` pragma or this definition won't be visible outside the module.

Next we modify the appropriate rules to use the new action. Here's the `show_contact` rule that controls the Contact page:

```
rule show_contact {  
  select when web click "#siteNavContact"  
    or web hash_change newhash "/contact$"  
  pre {  
    contact_html = <<  
      <h2 class="mainheading">Contact</h2>  
      <article class="post">  
        <p>Contact information here</p>  
      </article>  
    >>;  
    title = config:blogtitle + " - Contact";  
  }  
  {  
    config:paint_container(title, contact_html);  
    config:update_frag("/contact");  
  }  
}
```

² The JavaScript `location.hash` variable has an inconsistent interface. When you set it, you don't include the `#`. But when you read it, the string you get back has the `#` prepended.

This ensures that whenever the Contact page is displayed, the URL will have `#!/contact` appended:

`http://www.windley.com/kblog/#!/contact`

We make similar changes to the Home page so that it has `#!/` appended to the URL whenever we visit that page³.

Updating the Page When the Fragment Changes

Now that we've modified each of the rules that control a page, they will all be identified with the right fragment as you navigate from page to page using the links in the navigation bar at the top of the blog. But that's just half the problem. The back and forward buttons are still broken. If you use them the URL will change, but nothing changes the content as the URL changes. As we said before, the browser does not alert the server when the fragment changes.

To remedy this problem, we'll use a jQuery plugin called `hashchange`. Since the KRL Web endpoint has jQuery built in, using jQuery plugins is easy.

The first step is to put the code for the plugin on our server and change the jQuery variable in the final line so that we pass in `$KOBJ` instead:

```
...  
// was --> })(jQuery);  
})($KOBJ);
```

The plugin must call the KRL jQuery library, named `$KOBJ`, because the Web endpoint uses jQuery in extreme compatibility mode to ensure that it doesn't interfere with Web pages that have already loaded jQuery.

³ Technically, I shouldn't be putting the `!` in the fragment since that indicates to search engines that the page is available at the non-fragmented URL for search engine crawlers. While we won't enable this functionality in this tutorial, it's a good idea to make SPI pages crawlable. More information, including techniques for building this functionality can be found here: <http://code.google.com/web/ajaxcrawling/docs/getting-started.html>

The second step is to load the plugin in our ruleset. KRL provides a facility for loading external JavaScript resources as a pragma in the meta section of the ruleset:

```
use javascript resource "http://.../jquery.hashchange.js"
```

This loads the library in the browser once (and only once) when the ruleset is executed.

Third, we need to deploy the `hashchange` watcher so that monitors the fragment and raises an event to KRE when it changes. The `hashchange` watcher is designed so that it runs a function.

You might be tempted to put the call to `hashchange` in the global block. As programmers we're used to "global" blocks being evaluated once. And that's true for KRL as well, but in KRL it's once per event. Note that any given interaction with the blog causes multiple events. Putting `hashchange` in the global block would result in the watcher being added to the page every time someone clicked a button, submitted a form, or viewed the page. Because watcher is not idempotent, the effect would compound—not what we want.

What we should do instead is place the fragment watcher once when we initialize the blog. The `init_html` rule is responsible for initialization and is only run once per blog interaction—on the initial pageview. In fact, recall that `init_html` uses `place_button()` to put the buttons in the navigation bar and place watchers on them to monitor when they're clicked. Here's `init_html` modified to emit the correct JavaScript to watch for fragment changes:

```
rule init_html {
  select when pageview ".*" setting ()
  {
    replace_inner("#about", blogconfig:about_text);
    blogconfig:place_button("Home");
    blogconfig:place_button("Contact");
    emit <|
      self.document.location.hash='!/' ;
      $KOBJ(window).hashchange(function() {
        if(KOBJ.a16x88.previous == undefined ||
          KOBJ.a16x88.previous != self.document.location.hash) {
          var app = KOBJ.get_application("a16x88");
          app.raise_event("hash_change",
```

```

        {"newhash": self.document.location.hash});
        KOBJ.a16x88.previous = self.document.location.hash;});
    |>;
}
always {
    raise explicit event blog_ready
}
}

```

The only thing new is the `emit` action. The JavaScript in that action sets to fragment to the default page fragment and attaches a `hashchange` JavaScript watcher to the window. The function that it calls uses the Web endpoint runtime to get the app object associated with the current ruleset and raise an event called `hash_change` to the KRL engine with the new fragment as a parameter. The whole thing is wrapped in an `if` statement to ensure it only runs once per fragment change.

Note that we are making use of the `KOBJ` object that is defined in the Web runtime to store values. We namespace our definitions with the ruleset ID to avoid variable name collisions with other rulesets the user may running.

The built-in `raise_event()` method allows us to pass event attributes to KRE. We'll make use of them in the rules that respond to the `hash_change` event.

The last modification is to modify the rules that present the pages, `show_home` and `show_contact`, to respond to the `hash_change` event. There are two changes we need to make:

- We add another clause to the select statement that looks for a `hash_change` event that has a parameter named `newhash` with the value matched by the appropriate regular expression to ensure we're looking at the right page (`/contact` for the contact page and `/` for the home page).
- We set the `previous` variable in the browser to ensure that this event only happens once. This is most conveniently done in the `update_frag()` action.

Here's the select statement from the `show_contact` rule showing how it has been changed:

```

select when web click "#siteNavContact"
        or web hash_change newhash "/contact$"

```


The result of the rule is unchanged.

Here is the new definition for `update_frag()` showing the changes to the JavaScript that is emitted:

```
update_frag = defaction(name) {  
  emit <|  
    KOBJ.a16x88.previous = '#!#{name}';  
    self.document.location.hash='#!#{name}';  
  |>;  
};
```

Changes to the Event Hierarchy

Making the changes outlined above has created a new event, `hash_change`. We've also modified rules to watch for that new event. This modifies the event hierarchy as shown in Figure 4.

The event hierarchy diagram is only slightly more complicated:

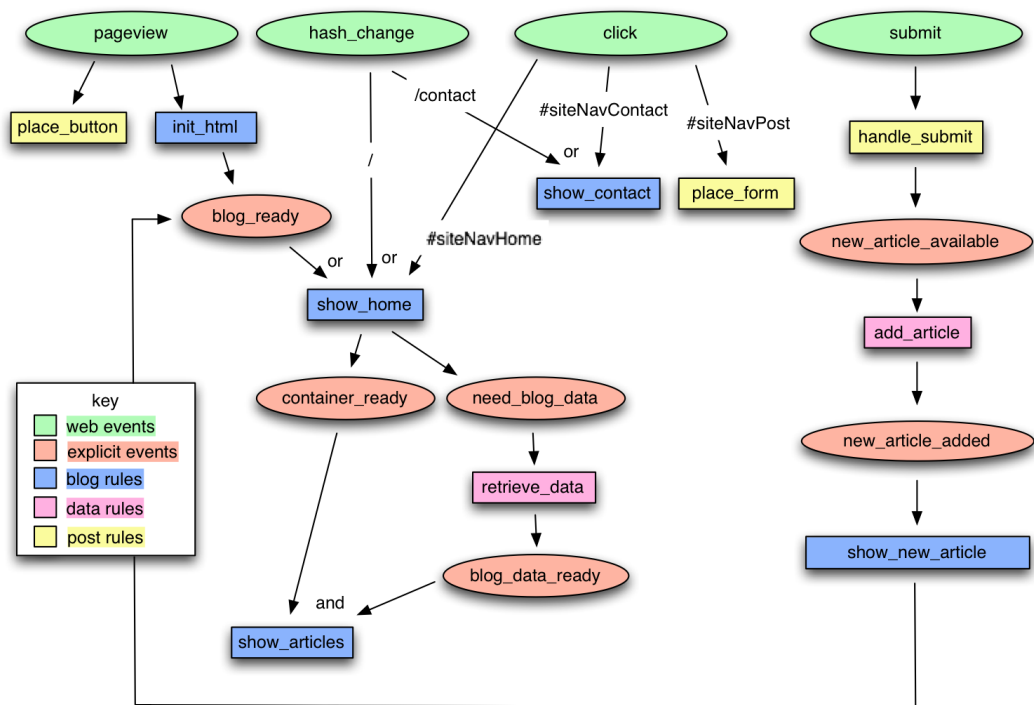


Figure 4. Event hierarchy for `KBlog` application after fixing the back button.

Figure 4 shows the new event at the top and its interaction with the `show_home` and `show_contact` rules. These rules are selected for specific values of the event attributes.

With these changes navigating the blog causes the URL to update and, conversely changing the URL changes the blog. Now the back and forward buttons function correctly.

An Experiment in Loose Coupling

We've created an application that uses multiple rulesets and has a more complicated event hierarchy than the KRL rulesets we've seen in previous chapters. We've used JavaScript to make the back button work. We even added a new event type (`web:hash_change`) using JavaScript.

KBlog makes an interesting laboratory for exploring how event-driven applications can be extended in a loosely coupled way. We've argued in previous chapters that because event-driven applications don't rely on request-response interactions, they can be more loosely coupled. Functionality can be layered onto event-driven applications in ways that's difficult to imagine in more tightly coupled architectural styles.

Unlike a typical program—where control flow is made explicit through the use of procedure calls or a Web program where control flow occurs through the request-response interactions—the event consumers themselves specify control flow in event-driven applications.

Traditionally programming languages have used the concept of 'hooks' to make control flow more flexible. Hooks are not events, but rather are places in the control flow where other programs can insert themselves into the control flow. One key difference is that events provide a level of indirection so that what runs can be dynamically determined.

Tweeting a Post

In this section we'll expand the blog application by layering on functionality that gives the author of a post the option of automatically tweeting the blog post title. The goal is to use what's already in the application to add the functionality with minimal changes to the existing application.

If you recall, the rule that listens for the event raised when a form is submitted, `handle_submit`, raises the event

`new_article_available`. We can post the title of the blog article to Twitter by listening for that event and doing what's necessary to format and send the information to Twitter:

```
rule send_tweet {
  select when explicit new_article_available
  pre {
    post = event:attr("post");
    tweet = <<
New blog post from #{ post.pick("$.postauthor")}:
#{ post.pick("$.posttitle")} http://www.windley.com/kblog
>>;
  }
  if (twitter:authorized()) then {
    twitter:update(tweet);
  }
}
```

`Send_tweet` listens for the `new_article_available` event and uses the built-in KRL Twitter module to check the keys (stored in the key pragma of the meta section of the ruleset) and update the associated Twitter stream. Adding the additional functionality was easy because the event was already being raised and the Twitter module does the heavy lifting.

Making Tweeting Optional

We can take some satisfaction in our ability to easily add new functionality to the original application. The next step—making tweeting the post optional—however, reveals some mistakes in our previous design that get in the way of modifying it by simply adding new rules without modifying the original rules.

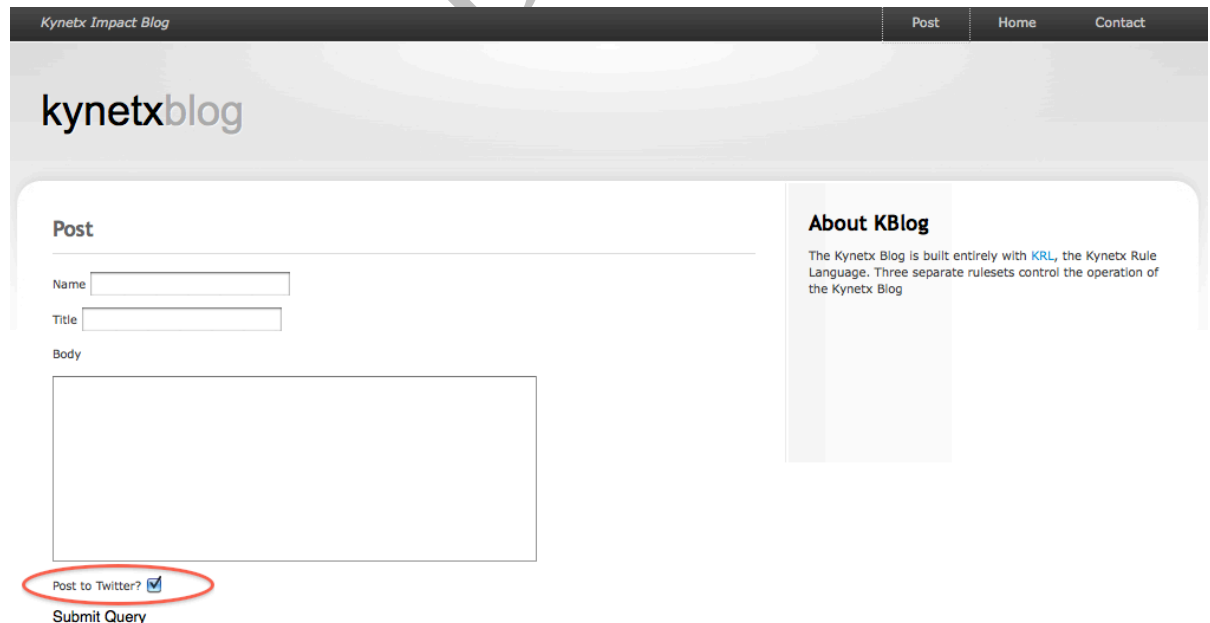
To give the author the option of tweeting or not, we want to add a checkbox to the blog post form. Adding the checkbox is relatively easy if an event reveals when the form is available. In the current design, the `place_form` rule is terminal in the event hierarchy, meaning that it raised no events.

Without an event, there's nothing for the rule that adds the checkbox to watch. We'll modify the `place_form` rule to raise the explicit event

post_form_ready. Now the place_checkbox rule can listen for that event and add a checkbox to the form:

```
rule place_checkbox {
  select when explicit post_form_ready
  pre {
    form_id = event:attr("form_id");
    checkbox_html = <<
    <p class="checkbox">
      <label for="posttitle">Post to Twitter?</label>
      <input id="tweet" type="checkbox" checked="checked"></p>
    >>;
  }
  after(".text-area", checkbox_html);
}
```

This rule adds a checkbox to the form without modifying the original ruleset beyond the fix to raise the event. Other changes to the form for other features could also be added. The form now becomes a canvas that rules can paint as they modify the functionality of the underlying application. Figure 5 shows the change for the form when the rule fires.



The screenshot shows the 'Kynetx Impact Blog' interface. At the top, there is a navigation bar with links for 'Post', 'Home', and 'Contact'. Below the navigation bar is the 'kynetxblog' logo. The main content area is divided into two columns. The left column is titled 'Post' and contains a form with fields for 'Name', 'Title', and 'Body'. Below the 'Body' field is a checkbox labeled 'Post to Twitter?' with a blue checkmark icon, which is circled in red. Below the checkbox is a 'Submit Query' button. The right column is titled 'About KBlog' and contains text stating: 'The Kynetx Blog is built entirely with KRL, the Kynetx Rule Language. Three separate rulesets control the operation of the Kynetx Blog'.

Figure 5. Form with Tweet checkbox added before the submit button

Now that there's a checkbox on the form, the `handle_submit` rule will include it in the event attributes for the `new_article_available` event because of how they are passed:

```
raise explicit event new_article_available
  for ["a16x89", "a16x91"]
    with post = event:attrs();
```

The function `event:attrs()` passes all attributes along even as they change based on other rules. However, when I first wrote `handle_submit`, the `raise` statement looked like this:

```
raise explicit event new_article_available for a16x89 with
  postauthor = event:attr("postauthor") and
  posttitle = event:attr("posttitle") and
  postbody = event:attr("postbody");
```

The design of this explicit event worked fine for the original blog functionality, but obviously, it won't pass new attributes, like our tweet checkbox value. Loose coupling demands that we make accommodation for the uses we don't envision. This sounds hard, but there are rules of thumb that help. We'll explore some of them in the next section.

The `send_tweet` rule can be modified to use the tweet checkbox value by adding an additional boolean test to the rule condition as follows:

```
if (post.pick("$.tweet", true).length() &&
    twitter:authorized()) then {...
```

The optional second parameter to `pick` causes it to always return an array if true. The `pick` operator will return individual values otherwise. The checkbox input in HTML has a value if it's checked, but is just not sent otherwise. By forcing the result to be an array, we can use `length` to determine if it's empty or not. A length of zero means it's missing and we don't want to update Twitter in that case.

Lessons Learned

The new ruleset consists of two rules: `place_checkbox` for putting the checkbox in the form and `send_tweet` for composing and sending the

tweet if the checkbox is checked. The new rules overlay the existing application to add the functionality without interrupting the existing features. If they're not present then the functionality isn't either. The application keeps on working just as it did before. Adding the functionality requires no configuration of the original application or wiring the new functionality in place. Everything fits nicely together and comes apart just as easily because of events.

But as we saw, there were design decisions we made in the original application that made it more tightly coupled than we'd like. We ran into two problems:

1. Terminal rules
2. Over-specified attributes

Let's explore the rules that arise from solving these problems in turn.

Avoid Terminal Rules

Terminal rules reduce opportunities for loose coupling. Because the `place_form` rule was terminal—it didn't raise any events to indicate what it had done—there was no way for another ruleset to extend its functionality.

The lesson here is that terminal rules should be avoided if your goal is to create extensible, loosely coupled applications that don't require code modifications. That said, as long as you have access to the code, adding explicit events to rules when you need them isn't difficult and is unlikely to break anything, so it's a low-cost, low-risk code modification.

Generalize Attribute Passing

Specificity in event attributes leads to tight coupling. When events have attributes, we should send them on as a map. Picking out individual attributes and sending them on by name means that only those attributes will ever be available to other rules.

That doesn't mean you can't filter attribute maps to remove data you don't want available downstream. But the result ought to be everything but the data you filter, not just the data you choose to pass on. Pass attributes as a structure rather than by name for support the greatest flexibility.

Supporting Extensibility

The problems we highlighted in this section are easy to find and a simple test application showed where they existed. Writing applications as collections of rulesets in such a way that they support extending functionality through accretion in a loosely coupled manner is practical, but it does require some planning and design effort.

Changing the Event Hierarchy Again

The modification to the event hierarchy is fairly modest since only two new rules were added. This reflects the loosely coupled nature of the application. Figure 6 shows the changes.

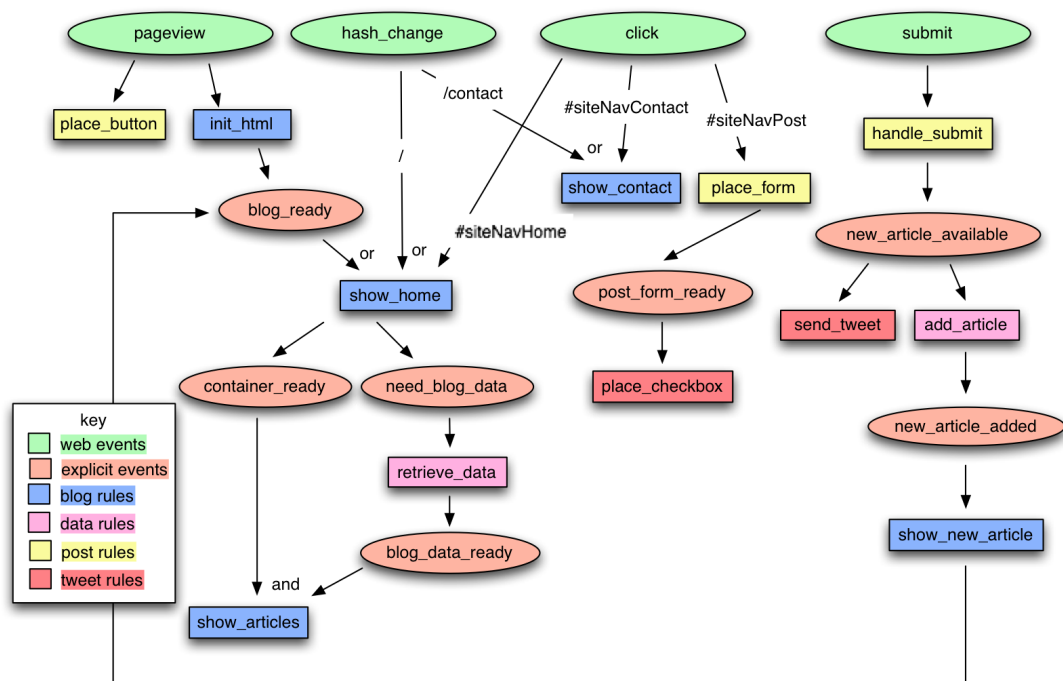


Figure 6. Event hierarchy for KBlog with the ability to tweet blog posts.

There is a new ruleset for the tweet rules. The `place_form` rule now raises the `post_form_ready` event for which the `place_checkbox` rule is listening. The `send_tweet` rule listens for the `new_article_available` event. As they should, the new rules are

accretive to the overall event hierarchy, not requiring changes to it, but merely adding to it.

Building Loosely Coupled Applications

This chapter has focused on an application built from multiple rulesets. The example, a blog application, isn't the first kind of application that one thinks about when building event-driven systems, but we've seen that events are actually a reasonable way to build a blog.

As we mentioned, there are several improvements we could make to the application if we wanted to use it in a production system, including using a database instead of application variables and adding some kind of authentication method to the posting subsystem. These changes would also allow for the application to be multi-tenanted—allowing more than one user to install and use it to manage their own blog.

The ideas of event normalization and even hierarchies provide tools for understanding large event systems. Combined with the rules we discovered for making event systems more loosely coupled, we're equipped to build and understand large event-driven applications.