Creating a Wiki with UCW

Contents

1 Dependencies **1**

2 Introduction **1**
2.1 Getting Started 1

3 The wiki backend **1**

4 Regular Style **2**
4.1 view.ucw 2
4.1.1 view.ucw - Introducing defentry-point 3
4.1.2 view-wiki-page - Introducing components 3
4.2 edit.ucw 4
4.2.1 The entry-point 4
4.2.2 The component 4
4.3 The thankyou page 4

5 UCW Style **5**
5.1 wiki-manipulator 5
5.2 wiki-viewer 5
5.2.1 The view-page action 6
5.2.2 edit-page - Introducing defaction 6
5.3 wiki-editor 6
5.4 wiki-editor-thankyou 7
5.5 wiki-window 7

6 Adding a Login Page **8**
6.1 Regular Style 8
6.1.1 Redefining edit.ucw 8
6.1.2 New page wiki-login.ucw 8
6.2 UCW Style 9
6.2.1 Redefining the edit-page action 9
6.2.2 The wiki-login component and actions 9

1 Dependencies

```
(asdf:oos 'asdf:load-op :cl-ppcre)
```

```
(asdf:oos 'asdf:load-op :ucw.examples)
```

```
(in-package :it.bese.ucw-user)
```

2 Introduction

This tutorial will show you how to create web applications with UnCommon Web (UCW). During this tutorial we will create two distinct, but functionaly similar, wikis. The first, "regular style," will be developed using the same mind set and organization as you would use with any other web framework (though it will use as much of UCW as possible). The second wiki, "UCW style," will be an idiomatic UCW application.

We've named the urls and classes in such a fashion that the two interfaces can coexist within the same image. You can try jumping back and forth between the two styles and compare how they act.

2.1 Getting Started

At some point we may add minimal installation instructions here, until then we'll just assume that you've already downloaded, built and setup ucw.

To load the wiki just load this file after you've started UCW. This wiki depends on cl-ppcre and the ucw example application.

3 The wiki backend

Both of our wikis will have very similar functionality: we will allow users to view and edit pages; when viewing a page words written in StudlyCaps will be converted into hyperlinks to like named wiki pages; pages will be editable by anyone at any time; every change will record the name of the author and a brief summary of the changes.

We won't worry about "RecentChanges" or search functionality (though this may be added later).

```
(defvar *wiki* (make-hash-table :test 'equal))

(defun find-wiki-page (page-name)
  (gethash page-name *wiki*))

(defun (setf find-wiki-page) (page page-name)
  (setf (gethash page-name *wiki*) page))

(defclass wiki-page ()
  ((page-name :accessor page-name :initarg :page-name)
   (contents  :accessor contents  :initarg :contents :initform "")))

(defmacro defwiki-page (name body)
  (rebinding (name)
    `(setf (find-wiki-page ,name)
           (make-instance 'wiki-page
                          :page-name ,name
                          :contents ,body))))
```

Both of the wikis use "WelcomePage" as the default page name, by creating the page here we simplify some of the control flow.

```
(defwiki-page "WelcomePage"
  "Welcome to the wiki.")

(defclass wiki-edit ()
  ((author :accessor author :initarg :author :initform nil)
   (summary :accessor summary :initarg :summary :initform nil)
   (contents :accessor contents :initarg :contents))
  (:documentation "An object representing a single change to a
  single wiki page."))

(defun update-wiki-page (page-name change)
  (symbol-macrolet ((page (find-wiki-page page-name)))
    (unless page
      (setf page (make-instance 'wiki-page :page-name page-name)))
    (setf (contents page) (contents change))
    page))
```

4 Regular Style

4.1 view.ucw

The view.ucw page is used for viewing the contents of a wiki page. The user specifies which page they're insterested in using the "page-name" HTTP request parameter. If no page-name is specified we default to "WelcomePage," if the specified page-name does not exist we assume the user wants to edit (in this case create) the page.

4.1.1 view.ucw - Introducing defentry-point

```
(defentry-point "view.ucw" (:application *example-application*)
    ((page-name "WelcomePage"))
  (if (find-wiki-page page-name)
      (call 'view-wiki-page :page-name page-name)
      (call 'redirect-component
            :target (strcat "edit.ucw?page-name=" page-name))))
```

DEFENTRY-POINT is UCW's way of associating a piece of code to a particular url, it's analogous to handlers or servlet definitions in other frameworks.

In this case we are creating an entry-point named "view.ucw" and tying it to the example-app application. The entry-point takes one parameter: page-name. If the page-name parameter is not passed in the request we will use "WelcomePage."

This entry-point is very simple, if the requested wiki-page exists then we should just view it using the page (aka component) view-wiki-page, if the requested page doesn't exist then we'll just pretend that the user actually wants to edit the page.

What is the "effective" url for this entry point? Since example-app application's url-prefix is "/ucw/examples/" we simply concatentate the entry-point's name and the url-prefix to get "/ucw/examples/view.ucw". The extension needn't be ".ucw" it can be anything you want or nothing at all. When using the araneida or aserve backends the entry-point's url is automatically registered with the server, with the mod_lisp backend you must manually configure apache to send requests for this entry point to ucw.

4.1.2 view-wiki-page - Introducing components

How do we tell UCW to show the user some html? While we could just litter our entry-point with (progn (write-line "¡html¿") ...) we're not going to, it's bad style and UCW is makes that more difficult than it could be.

What we're supposed to do is hand off our request to a component and let it deal with the nitty gritty html stuff. Here's the form which defines the view-wiki-page component:

```
(defcomponent view-wiki-page (window-component
                              template-component)
  ((page-name :accessor page-name :initarg :page-name))
  (:default-initargs :template-name "ucw/examples/view.tal"))

(defmethod template-component-environment nconc
    ((page view-wiki-page))
  (tal-env 'contents
           (cl-ppcre:regex-replace-all
            "([A-Z][a-z]+){2,}"
            (contents (find-wiki-page (page-name page)))
            "<a href=\"view.ucw?page-name=\\&\">\\&</a>")))
```

Notice how much the defcomponent macro looks like defclass, that's not accidental. view-wiki-page now names a class of components. The (call 'view-wiki-page ...) form in our view.ucw entry-point is little more than a call to make-instance.

view-wiki-page, since it's a window-component, is designed to occupy the entire browser window, it has to worry about emiting ¡html¿ and ¡head¿ tags and setting up javascript includes and style sheet links. Since view-wiki-page is also a template component it depends an a TAL file ("ucw/examples/view.tal") to specify what html to output.

4.2 edit.ucw

4.2.1 The entry-point

Editing pages is only slightly more complicated than viewing:

```
(defentry-point "edit.ucw" (:application *example-application*)
  ((page-name "WelcomePage") name summary contents)
  (if contents
      (progn
        (update-wiki-page page-name (make-instance 'wiki-edit
                                                    :author name
                                                    :summary summary
                                                    :contents contents))
        (call 'thankyou :page-name page-name))
      (call 'edit-wiki-page :page-name page-name)))
```

We assume that if the request contains the contents parameter then we're submitting an edit, otherwise we're asking for the edit page form. We've already seen DEFENTRY-POINT, entry-point lambda lists and call, so we can jump directly to the edit-wiki-page component:

4.2.2 The component

```
(defcomponent edit-wiki-page (window-component template-component)
  ((page-name :accessor page-name :initarg :page-name))
  (:default-initargs :template-name "ucw/examples/edit.tal"))
```

Like view-wiki-page this is also a window component based on a TAL template.

4.3 The thankyou page

Just for fun we're going to use YACLML as opposed to TAL for the thankyou component:

```
(defcomponent thankyou (window-component)
  ((page-name :accessor page-name :initarg :page-name)))

(defmethod render-on ((res response) (page thankyou))
  (symbol-macrolet ((page-name (<:as-html (page-name page))))
    (<:html
      (<:head
        (<:title "Thank you for editing " page-name))
      (<:body
        (<:p "Thank you for editing " page-name)
        (<:a :href (strcat "view.ucw?page-name=" (page-name page))
          "View " page-name)))))
```

As you can see by the strcat UCW isn't well adapted to munging strings into urls. [the situation is slightly better in tal pages with tal expression language].

5 UCW Style

UCW styled apps, unlike regular style apps, start with components and actions. Only later do we tie components and actions to urls and pages.

5.1 wiki-manipulator

All of our wiki components are going to subclass wiki-manipulator. This class provides the page-name slot and the method on update-url.

```
(defcomponent wiki-manipulator ()
  ((page-name :accessor page-name
              :initarg :page-name
              :backtrack t)))
```

We define a method on update-url so that bookmarking this page, or requesting it any time after the session has expired, will view the current page (even if the user was editing it when he created the bookmark). Note that there is nothing automatic about this update-url method. It only works because we know that wiki.ucw?page-name=Foo will show the page named Foo. Embedding the wiki in another application would probably make this assumption untrue and require a different update-url method.

```
(defmethod update-url ((component wiki-manipulator) url)
  (setf (ucw::uri.path url) "wiki.ucw")
  (push (cons "page-name" (page-name component)) (ucw::uri.query url))
  url)
```

5.2 wiki-viewer

The wiki-viewer component shows a page of the wiki. In particular the component show the html version of the page named by the value of the viewer's page-name slot. Unlike the wiki-editor component a user will use the same wiki-viewer component during the entire course of their browsing.

```
(defcomponent wiki-viewer (wiki-manipulator)
  ())
```

We split the text of the page into StudlyWords and non StudlyWords. StudlyWords are wrapped in links to view-page actions, everything else is sent as is to the client.

```
(defmethod render-on ((res response) (page wiki-viewer))
  (let ((scanner (cl-ppcre:create-scanner "((?:[A-Z][a-z]+){2,})")))
    (dolist (part (cl-ppcre:split scanner
                                  (contents (find-wiki-page (page-name page)))
                                  :with-registers-p t))
      (if (cl-ppcre:scan scanner part)
          (let ((part part))
            (<ucw:a :action (view-page page part)
              (<:as-html part)))
          (<:as-is part))))
  (<:p (<ucw:a :action (edit-page page (page-name page)) "Edit")))
```

5.2.1 The view-page action

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (arnesi:assert-cc 'edit-page))

(defaction view-page ((w wiki-viewer) page-name)
  (unless (find-wiki-page page-name)
    (edit-page w page-name))
  (setf (page-name w) page-name))
```

view-page simply changes the name of the current page in the wiki-viewer (which is sufficent given how we've defined view-page's render-on method). If the page doesn't already exist we call the edit-page action.

5.2.2 edit-page - Introducing defaction

```
(defaction edit-page ((w wiki-viewer) page-name)
  (call 'wiki-editor :page-name page-name)
  (call 'wiki-editor-thankyou :page-name page-name))
```

This first calls a wiki-editor component, when that returns (by calling answer) the action continues and we call the wiki-editor-thankyou component.

Notice that while this action CALLs multiple components we do not ANSWER. The view-page component basically sits in an infinite loop, it shows various wiki pages and it calls out to various other components, but the wiki-viewer itself will never answer.

If we were to embed a wiki-viewer in other application we would probably subclass wiki-viewer and change its render-on method to produce a link which causes the component to answer.

5.3 wiki-editor

```
(defcomponent wiki-editor (wiki-manipulator)
  ())
```

This action, which is called from the edit form, updates the wiki and then returns control to the calling component.

```
(defaction save-changes ((w wiki-editor) name summary contents)
  (update-wiki-page (page-name w)
                    (make-instance 'wiki-edit
                                   :author name
                                   :summary summary
                                   :contents contents))
  (answer (page-name w)))

(defmethod render-on ((res response) (w wiki-editor))
  (let ((name "")
        (summary "")
        (contents (if (find-wiki-page (page-name w))
                      (contents (find-wiki-page (page-name w)))
                      "")))
    (<ucw:form :action (save-changes w name summary contents)
               :method "POST"
      (<ucw:textarea :rows 15 :cols 80 :accessor contents)
      (<:p
        (<:label :for "author" "Name:")
        (<ucw:text :accessor name :id "author"))
      (<:p
        (<:label :for "summary" "Summary:")
        (<ucw:text :accessor summary :id "summary"))
      (<:p (<:input :type "submit" :value "Save Changes")))))
```

5.4 wiki-editor-thankyou

```
(defcomponent wiki-editor-thankyou (wiki-manipulator)
  ())

(defmethod render-on ((res response) (w wiki-editor-thankyou))
  (<:p "Thank you for editing " (<:as-html (page-name w)))
  (<ucw:a :action (ok w) "Ok."))
```

wiki-editor-thankyou uses the generic OK action. This action, defined in UCW on all standard-components, simply calls answer.

5.5 wiki-window

The wiki components have been designed so that they can be embeded in any other component. What this means is that our trivial application needs some window component we can embed our wiki components in:

```
(defcomponent wiki-app (window-component)
  ((body :accessor body :component (wiki-viewer :page-name "WelcomePage"))
   (title :accessor title)))

(defmethod initialize-instance :after ((app wiki-app)
                                       &key (page-name "WelcomePage")
                                       &allow-other-keys)
  (setf (page-name (body app)) page-name))

(defmethod render-on ((res response) (w wiki-app))
  (when (subtypep (class-of (body w)) (find-class 'wiki-manipulator))
    (setf (title w) (page-name (body w))))
  (<:html
    (<:head
      (<:title (<:as-html (title w))))
    (<:body
      (<:h1 (<:as-html (title w)))
      (render-on res (body w)))))

(defentry-point "wiki.ucw" (:application *example-application*)
    ((page-name "WelcomePage"))
  (call 'wiki-app :page-name page-name))
```

6 Adding a Login Page

Let's pretend that, in order to avoid spurious spamming, we want to make users login before editing pages.

NB: If you are comparing the two wiki implementations you need to remeber that both interfaces use the same session object. Logging in through the "regular style" wiki will couse you to be logged into the "UCW style" wiki and vice versa.

6.1 Regular Style

6.1.1 Redefining edit.ucw

The only way to edit a page is to go through the edit.ucw, so we'll put the login logic there. Every time a user attempts to edit a page we look in the current session for something (anything other than NIL will do) under the key USER. If we find something then continue as before, if we don't we redirect to the wiki-login.ucw page.

```
(defentry-point "edit.ucw" (:application *example-application*)
  ((page-name "WelcomePage") name summary contents)
  (cond
    ((not (get-session-value 'user))
     (call 'redirect-component :target "wiki-login.ucw"))
    (contents
     (update-wiki-page page-name
                       (make-instance 'wiki-edit
                                      :author name
                                      :summary summary
                                      :contents contents))
     (call 'thankyou :page-name page-name))
    (t
     (call 'edit-wiki-page :page-name page-name))))
```

6.1.2 New page wiki-login.ucw

wiki-login.ucw looks for two request parameters, username and password. If they're found, and if they're both equal to "wiki" then we put the username in the session table under the key USER (so that the edit.ucw entry point will find it) and we redirect to view.ucw. If the username and password parameters aren't passed, or they're aren't equal to "wiki" then we show a wiki-login-for-edit component.

```
(defentry-point "wiki-login.ucw" (:application *example-application*)
    ((username "") (password ""))
  (if (and (string= "wiki" username) (string= "wiki" password))
      (progn
        (setf (get-session-value 'user) username)
        (call 'redirect-component :target "view.ucw"))
      (call 'wiki-login-for-edit)))
```

This wiki-login-for-edit component simply creates a very minimal page with a form for the username and password values.

```
(defcomponent wiki-login-for-edit (window-component)
  ())

(defmethod render-on ((res response) (login wiki-login-for-edit))
  (<:html
   (<:head (<:title "Login"))
   (<:body
    (<:form :action "wiki-login.ucw"
      (<:p "Login with wiki/wiki")
      (<:p
        (<:label :for "username" "Username: ")
        (<:input :type "text" :name "username" :id "username"))
      (<:p
        (<:label :for "password" "Password: ")
        (<:input :type "text" :name "password" :id "password"))
      (<:p (<:submit :value "Login"))))))
```

6.2 UCW Style

6.2.1 Redefining the edit-page action

As with the regular style wiki we'll put the login checking just before editing a page. If the the value of (get-session-value 'user) is null than we call wiki-login component and store whatever it answers in the session under the key user. When the wiki-login component has finished (or when we're already logged in) we simply continue as before by calling wiki-editor and wiki-editor-thankyou.

```
(defaction edit-page ((w wiki-viewer) page-name)
  (unless (get-session-value 'user)
    (setf (get-session-value 'user) (call 'wiki-login)))
  (call 'wiki-editor          :page-name page-name)
  (call 'wiki-editor-thankyou :page-name page-name))
```

6.2.2 The wiki-login component and actions

(while we could use ucw's login component we're going to try to keep this example as self contained as possible)

```
(defcomponent wiki-login ()
  ())

(defmethod render-on ((res response) (login wiki-login))
  (let ((username "")
        (password ""))
    (<ucw:form :action (wiki-login login username password)
      (<:p
        "Login with wiki/wiki.")
      (<:p
        (<:label :for "username" "Username: ")
        (<ucw:text :accessor username :id "username"))
      (<:p
        (<:label :for "password" "Password: ")
        (<ucw:password :accessor password :id "password"))
      (<:p
        (<ucw:submit :action (wiki-login login username password))))))

(defaction wiki-login ((login wiki-login) username password)
  (when (and (string= "wiki" username) (string= "wiki" password))
    (answer username)))
```