Flowscript & Woody

Webapps made easy with Cocoon

Sylvain Wallez
http://apache.org/~sylvain

www.anyware-tech.com

Flowscript intro

Flow control in Cocoon

- Aren't actions enough ?
  - Yes, but they require state management
  - Quickly becomes complex, hard to understand and to maintain

  → Actions are the traditional "MVC" controller
- Flowscript is a controller…
  - Calls business logic and chooses the view
- …but also more
  - Keeps the application state
  - Describes page flow as a sequential program
  - Easily defines complex interactions

Flowscript intro

Flow script example

```
var cart;
var user;

function checkout()
{
  while(user == null) {
    cocoon.sendPageAndWait("login.html");

    user = UserRegistry.getUser(cocoon.request.get("name"));
  }
  cocoon.sendPageAndWait("shippingAddress.html", {who: user});

  var address = cocoon.request.get("address");
  cocoon.sendPageAndWait("creditCard.html");

  var creditCard = cocoon.request.get("creditCard");
  cocoon.sendPageAndWait("confirmOrder.html");

  EnterpriseSystem.placeOrder(user, cart, address, creditCard);
  cocoon.sendPage("orderPlaced.html");
}
```

Flowscript intro

Why JavaScript ?

- Simpler than Java, although powerful
- Integrates well with Java
- Well-known in the web world
- Allows faster roundtrips (save and reload)
- Supports *continuations*

Calling the view

cocoon.sendPage()

- **`cocoon.sendPage`** invokes the output page (view) with two arguments
  - The view URL, relative to current sitemap
  - A context object made available to the view

    → Can be a Java or JavaScript object

```
cocoon.sendPage("checkout.html",
        {user: loggedUser, email: address});
```

- **`cocoon.sendPage("view.html")`** is like redirecting to "cocoon:/view.html"
- Control then comes back to the script

  → Should normally terminate

Calling the view

cocoon.sendPageAndWait()

- Similar to **`cocoon.sendPage`**
  - Invoke the view with a context object
- The script is *suspended* after the view is generated

  → the whole execution stack saved in a *continuation* object
- Flow between pages becomes *sequential code*

  → No more complicated state automata

Continuations

What is it ?

- Contents of a continuation:
  - Stack of function calls
  - Value of local variables

  → Most often a lightweight object

→ Creating a continuation does not halt a thread !!

- A continuation object is associated with a unique identifier available to the view

  → Later used to "resurrect" it

Continuations

Sample flow script revisited

⇨ saved continuations

```
var cart;
var user;

function checkout()
{
  while(user == null) {
    cocoon.sendPageAndWait("login.html");

    user = UserRegistry.getUser(cocoon.request.get("name"));
  }
  cocoon.sendPageAndWait("shippingAddress.html", {who: user});

  var address = cocoon.request.get("address");
  cocoon.sendPageAndWait("creditCard.html");

  var creditCard = cocoon.request.get("creditCard");
  cocoon.sendPageAndWait("confirmOrder.html");

  EnterpriseSystem.placeOrder(user, cart, address, creditCard);
  cocoon.sendPage("orderPlaced.html");
}
```

View layer

How to define the view ?

- It's a regular Cocoon pipeline
  → Preferably in an `internal-only="true"` pipeline
- Two generators providing tight integration
  - JXTemplateGenerator
  - JPath XSP logicsheet
  → Easy access to the context object

View layer

JXTemplateGenerator

- An XML template language inspired by JSTL
- Doesn't allow code, but only access to context variables
  → Simpler than XSP
- Flow values are provided as variables :

```
sendPage("checkout.html",
         {"customer": user, "cart": cart});
```

```
<p>Welcome, #{customer/firstName}</p>
```

- Two expressions languages:
  Jexl with `${…}` and JXPath with `#{…}`

View layer

JXTemplate example

```
Your cart:
<ul>
  <jx:forEach var="item" items="${cart.items}">
    <li>${item.quantity} ${item.name}</li>
  </jx:forEach>
</ul>
<a href="kont/${continuation.id}">Continue</a>
```

```
Your cart:
<ul>
  <li>3 Cocoon T-Shirt</li>
  <li>1 Washing machine</li>
</ul>
<a href="kont/bf6c433aa3148f8ca083f18a83813f81">Continue</a>
```

Will "resurrect" the flow script

Putting it all together

```
<map:pipelines>
  <map:pipeline>

    <map:match pattern="checkout">
      <map:call function="checkout"/>
    </map:match>

    <map:match pattern="kont/*">
      <map:call continuation="{1}"/>
    </map:match>

  </map:pipeline>

  <map:pipeline internal-only="true">

    <map:match pattern="*.html"/>
      <map:generate type="jxt" src="{1}.xml"/>
      <map:transform src="page2html.xsl"/>
      <map:serialize/>
    </map:match>
…/…
```

❶ Call a flow function

❸ Resurrect a continuation

❷ View called by the flow

Putting it all together

FOM: the Flow Object Model

- Provided by the "cocoon" global object
- Access to the environment
  - "`request`", "`response`", "`session`" & "`context`" properties
  - "`parameters`" : sitemap parameters
- Access to the framework
  - Logging, using Avalon components
- Page flow control
  - `cocoon.sendPage()`, `cocoon.sendPageAndWait()`
  - `cocoon.redirectTo()`

Session variables

Global scope = session scope

- Global variables are attached to the session
  - Saved across top-level function invocations
  - Specific to each user

→ Removes most of the needs for session attributes !

Session variables

Example

Shows the login screen only if needed

Won't pass through if not logged in !

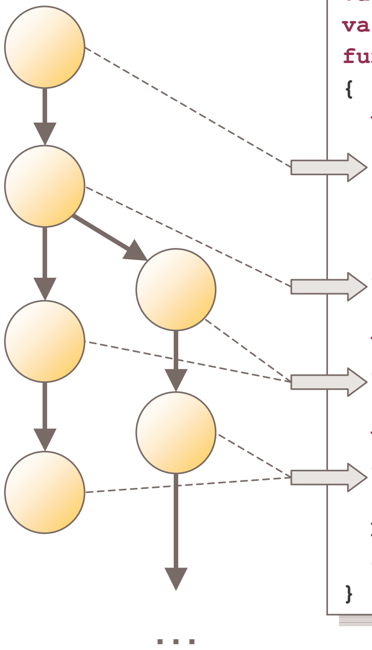
Just clear user info to log out

```
var user = null;

function login() {
  while (user == null) {
    sendPageAndWait("login.html");
    user = UserRegistry.getUser(
      cocoon.request.getParameter("name"),
      cocoon.request.getParameter("password") );
  }
}

function placeOrder() {
  login();
  Accounting.placeOrder(user);
  sendPage("orderPlaced.html");
}

function logout() {
  user = null;
  sendPage("bye.html");
}
```

Managing continuations

Continuation trees

- Browser "back" or "new window"



```
var cart;
var user;
function checkout()
{
  while(user == null) {
    cocoon.sendPageAndWait("login.html");

    user = UserRegistry.getUser(cocoon.request.get("name"));
  }
  cocoon.sendPageAndWait("shippingAddress.html", {who: user});

  var address = cocoon.request.get("address");
  cocoon.sendPageAndWait("creditCard.html");

  var creditCard = cocoon.request.get("creditCard");
  cocoon.sendPageAndWait("confirmOrder.html");

  EnterpriseSystem.placeOrder(user, cart, address, creditCard);
  cocoon.sendPage("orderPlaced.html");
}
```

Managing continuations

Continuation trees

- Browser "back" : the previous path is lost
- No fear : a continuation is lightweight
  - → Reference to the parent continuation
  - → Local variables since the parent continuation
- Browser "new window"
  - Creates a new branch
  - → Allows "what if ?" navigation in the application

Managing continuations

Expiring continuations

- Manual expiration :
  - **`sendPageAndWait()`** returns its continuation
  - **`k.invalidate()`** invalidates the continuation and its subtree :

```
var k = sendPageAndWait("start.html");
...
BusinessService.commit();
// Cannot go back again
k.invalidate();
```

  - → Again, avoids complicated state management
- Automatic expiration
  - An inactive continuation expires after a delay

Conclusion

Flow script

- Gives control back to the server
  - → We always know "where" the browser is
- Allows sophisticated flow screens
  - → No need for state automata
- Increases security and robustness
  - → Forbids direct access to form submission URLs
  - → Handles "back" and "new window"

Questions ?

Answers !

(next: Woody)

Woody intro

The need for form handling

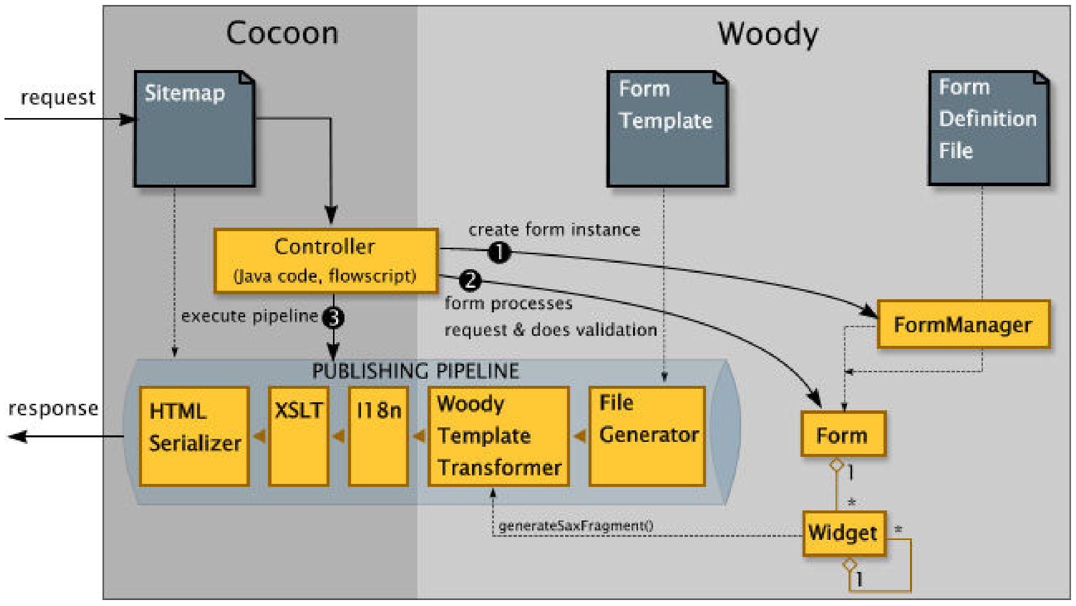
- Cocoon started as a publication framework
  - → Many pages, limited user feedback
  - → Content was mostly written "outside"
- Evolution towards a general-purpose web framework
  - → Published content has to be managed
  - → Used for more and more data-centric applications

→ Need for good form handling features

- Various attempts before Woody: FormValidatorAction, Precept, XMLForm, JXForms

Woody principles

The big picture



Woody principles

The form object model

- Composed of "widgets"
  - Represents "something" that appears in the form
  - Can read, parse and validate itself
  - Can output its XML representation
- Some widgets are non-terminal
  → Support for tables and rows

Woody principles

Form definition overview

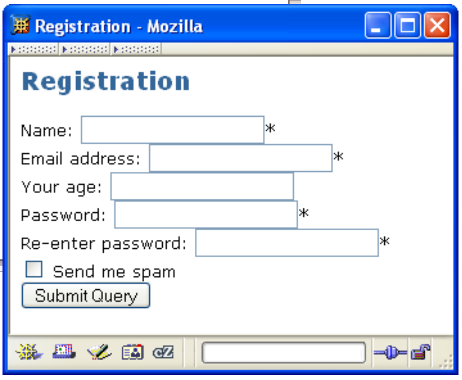
```
<wd:form xmlns:wd="http://apache.org/cocoon/woody/definition/1.0">

  <wd:field id="name" required="true">
    <wd:label>Name:</wd:label>
    <wd:datatype base="string">
      <wd:validation>
        <wd:length min="2"/>
      </wd:validation>
    </wd:datatype>
  </wd:field>

  <wd:field id="email" required="true">
    <wd:label>Email address:</wd:label>
    <wd:datatype base="string">
      <wd:validation>
        <wd:email/>
      </wd:validation>
    </wd:datatype>
  </wd:field>

  …/…
</wd:form>
```

Woody principles

Form template overview

- Embeds widget references in target markup

```
<html xmlns:wt="http://apache.org/cocoon/woody/template/1.0">
  <head>
    <title>Registration</title>
  </head>
  <body>
    <h1>Registration</h1>
    <wt:form-template action="registration" method="POST">
      <wt:widget-label id="name"/>
      <wt:widget id="name"/>
      <br/>
      <wt:widget-label id="email"/>
      <wt:widget id="email"/>
      <br/>
      …/…
      <input type="submit"/>
    </wt:form-template>
  </body>
</html>
```



Registration

Name:	*
Email address:	*
Your age:	
Password:	*
Re-enter password:	*

☐ Send me spam

The form definition file

Widgets

- Available widgets
  - <wd:form> : the main form widget
  - <wd:field> : "atomic" input field
  - <wd:booleanfield> : boolean input
  - <wd:mutivaluefield> : multiple selection in a list
  - <wd:repeater> : collection of widgets
  - <wd:output> : non-modifiable value
  - <wd:action> : action button (intra-form)
  - <wd:submit> : submit button (exits the form)
- They're all defined in cocoon.xconf

  → Add your own if needed

The form definition file

The <wd:field> widget

- Definition overview

```
<wd:field id="..." required="true|false">
  <wd:label>...</wd:label>
  <wd:datatype base="...">
     [...]
  </wd:datatype>
  <wd:selection-list>
     [...]
  </wd:selection-list>
</wd:field>
```

- The label can contain abitrary markup

```
<wd:label>Your <b>name</b></wd:label>
```

  - Including i18n references

```
<wd:label>
  <i18n:text key="name-field-label"/>
</wd:label>
```

The form definition file

Defining the data type of a field

- Mandatory "base" type
  - Defines the Java type
  - "string", "long", "decimal", "date", "boolean"

    → Pluggable components : add your own !
- Optional conversion and validation

Parsing and formatting

Validation

```
<wd:datatype base="...">
  <wd:convertor>
    [...]
  </wd:convertor>
  <wd:validation>
    [...]
  </wd:validation>
</wd:datatype>
```

The form definition file

Data type parsing and formatting

- Each base type has a set of converters
  → Pluggable components : add your own !
- Example : date's "formatting" converter
  - based on `java.text.SimpleDateFormat`
  → locale-dependent patterns

```
<wd:datatype base="date">
  <wd:convertor type="formatting">
    <wd:patterns>
      <wd:pattern>yyyy-MM-dd</wd:pattern>
      <wd:pattern locale="en">MM/dd/yyyy</wd:pattern>
      <wd:pattern locale="fr">dd/MM/yyyy</wd:pattern>
      <wd:pattern locale="nl-BE">dd/MM/yyyy</wd:pattern>
      <wd:pattern locale="de">dd.MM.yyyy</wd:pattern>
    </wd:patterns>
  </wd:convertor>
</wd:datatype>
```

OrixO the xml business alliance

ANYWARE TECHNOLOGIES

The form definition file

Data type validation

- A validation rule checks value validity
  - length, range, regexp, creditcard, assert, email
  → Pluggable components : add your own !
- A datatype can have several validation rules

```
<wd:field id="email">
  <wd:datatype base="string">
    <wd:validation>
      <wd:email/>
      <wd:length max='100'>
        <wd:failmessage>Your address it too long!</wd:failmessage>
      </wd:length>
    </wd:validation>
  </wd:datatype>
</wd:field>
```

OrixO the xml business alliance

ANYWARE TECHNOLOGIES

The form definition file

Selection lists

- Provide enumerations to the user
  - List of items having a value, with optional label

```
<wd:field name="OS">
  <wd:datatype base="string"/>
  <wd:selection-list>
    <wd:item value="Linux"/>
    <wd:item value="Windows"/>
    <wd:item value="Mac OS"/>
    <wd:item value="Solaris"/>
    <wd:item value="other">
      <wd:label><i18n:text key="other"/></wd:label>
    </wd:item>
  </wd:selection-list>
</wd:field>
```

- Selection lists can be external and dynamic

```
<wd:selection-list src="cocoon:/build-list.xml">
```

The form definition file

The <wd:repeater> widget

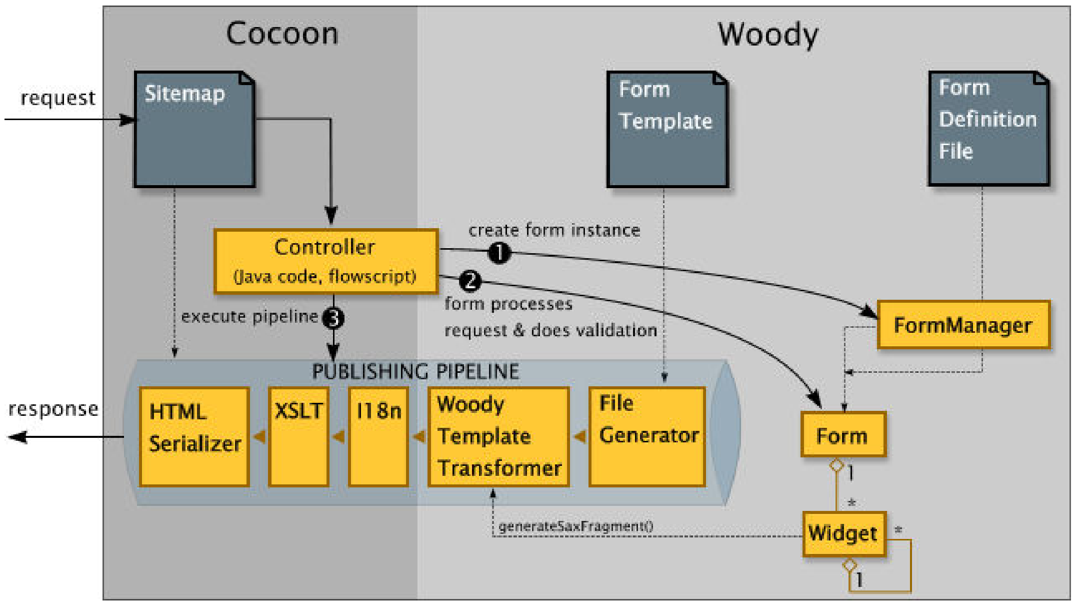
- Repeats a number of child widgets
  → Used to manage collections, tables, etc.

```
<wd:repeater id="contacts">

  <wd:field id="firstname">
    <wd:label>Firstname</wd:label>
    <wd:datatype base="string"/>
  </wd:field>

  <wd:field id="lastname">
    <wd:label>Lastname</wd:label>
    <wd:datatype base="string"/>
  </wd:field>

</wd:repeater>
```

- Specialized <repeater-action> widgets
  → Automatic row addition/deletion

The form template

The big picture (again)



The form template

Role of the WoodyTransformer

```
<html xmlns:wt="http://apache.org/cocoon/woody/template/1.0">
  <head>
    <title>Registration form</title>
  </head>
  <body>
    <h1>Registration</h1>
    <wt:form-template action="registration"
      <wt:widget-label id="name"/>
      <wt:widget id="name"/>
      <br/>
      <wt:widget-label id="email"/>
      <wt:widget id="email"/>
      <br/>
      …/…
      <input type="submit"/>
    </wt:form-template>
  </body>
</html>
```

```
<html xmlns:wt="http://apache.org/cocoon/woody/instance/1.0">
  <head>
    <title>Registration form</title>
  </head>
  <body>
    <h1>Registration</h1>
    <wi:form-template action="registration" method="POST">
     Name:
     <wi:field id="name">
       <wi:label>Name:</wi:label>
       <wi:value>Cocoon</wi:value>
     </wi:field>
     <br/>
     Email address:
     <wi:widget id="email">
       <wi:label>Email address:</wi:label>
       <wi:value>foo</wi:value>
       <wi:validation-message>
         Invalid email address
       </wi:validation-message>
     </wi:widget>
     <br/>
     …/…
     <input type="submit"/>
    </wi:form-template>
  </body>
</html>
```

Expanded widgets

Validation failed

The form template

Role of the WoodyTransformer

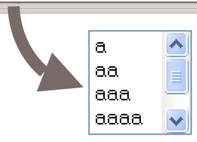
- Expand all "wt" elements in their "wi" counterpart
  - → "wt" = Woody template
  - → "wi" = Woody instance
- Output of the transformer goes to styling
  - Provided : HTML styling
  - Other stylings are possible (e.g. WML)
  - → Woody does not hardcode the presentation !

The form template

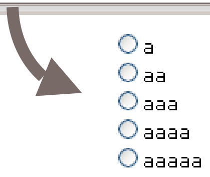
The <wt:widget> element

- Produces the corresponding widget instance
  - Markup depends on the actual widget
  - For fields : <wi:label>, <wi:value>, <wi:selection-list>
- <wt:widget> can contain styling information
  - Drives the styling stylesheet
  - → Contents of <wi:styling> depends on the stylesheet !

```
<wt:widget id="fourchars">
 <wi:styling list-type="listbox"
             listbox-size="4"/>
</wt:widget>
```



```
<wt:widget id="fourchars">
 <wi:styling list-type="radio"/>
</wt:widget>
```



The form template

The <wt:repeater-widget> element

- Iterates on the contents of a <wd:repeater>

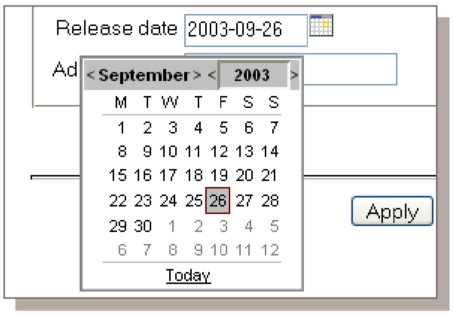
```
<table>
  <tr>
    <th>
      <wt:repeater-widget-label
          id="contacts" widget-id="firstname"/>
    </th>
    <th>
      <wt:repeater-widget-label
          id="contacts" widget-id="email"/>
    </th>
  </tr>
  <wt:repeater-widget id="contacts">
    <tr>
      <td>
        <wt:widget id="firstname"/>
      </td>
      <td>
        <wt:widget id="email"/>
      </td>
    </tr>
  </wt:repeater-widget>
</table>
```

```
<table>
  <tr>
    <th>Name</th>
    <th>Email address</th>
  </tr>
  <tr>
    <td>
      <wi:field id="contacts.0.firstname">
        <wi:label>Name</wi:label>
        <wi:value>Harry</wi:value>
      </wi:field>
    </td>
    <td>
      <wi:field id="contacts.0.email">
        <wi:label>Email address</wi:label>
        <wi:value>harry@potter.com</wi:value>
      </wi:field>
    </td>
  </tr>
  <tr>
    <td>
      <wi:field id="contacts.1.firstname">
        <wi:label>Name</wi:label>
        <wi:value>Anakin</wi:value>
      </wi:field>
    </td>
    <td>
      <wi:field id="contacts.1.email">
        <wi:label>Email address</wi:label>
        <wi:value>anakin@skywalker.com</wi:value>
      </wi:field>
    </td>
  </tr>
</table>
```

Built in HTML styling

Field styling

- Basic styling: html input
- <wi:styling type="...">
  - "password", "hidden", "textarea", "date"
  - "listbox", "radio" for selection-lists

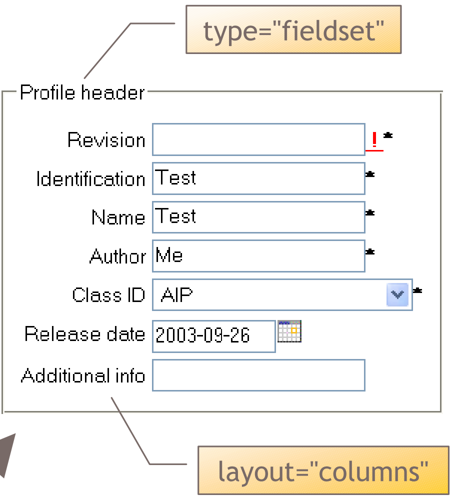

Release date 2003-09-26

Built in HTML styling

<wi:group> styling

- Instance-only widget providing high-level styling
  - → No corresponding <wd:> nor <wt:>

```
<wi:group>
  <wi:label>Profile header</wi:label>
  <wi:styling type="fieldset" layout="columns"/>
  <wi:items>
    <wt:widget id="revision"/>
    <wt:widget id="identification"/>
    <wt:widget id="name"/>
    <wt:widget id="author"/>
    <wt:widget id="classID"/>
    <wt:widget id="releaseDate">
      <wi:styling type="date"/>
    </wt:widget>
  </wi:items>
</wi:group>
```



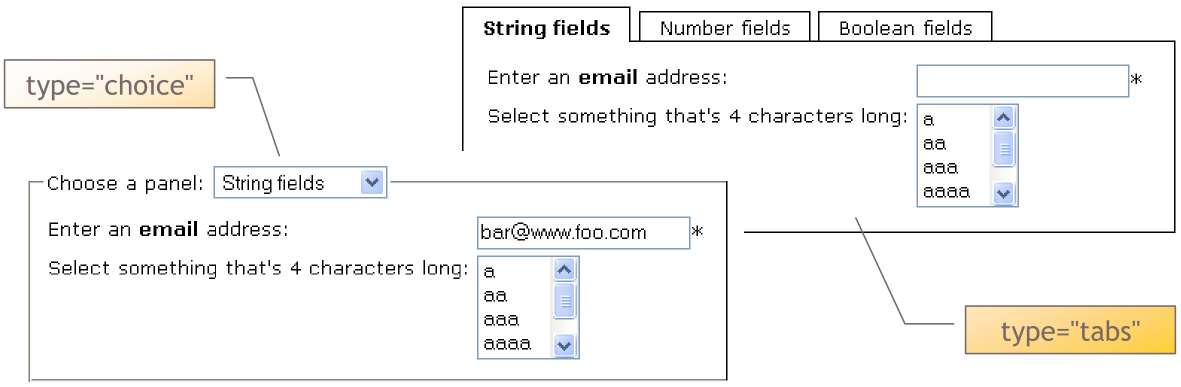
Profile header

Revision	! *
Identification	Test *
Name	Test *
Author	Me *
Class ID	AIP *
Release date	2003-09-26
Additional info	

Built in HTML styling

<wi:group> styling

- Container rendering
  - → "type" attribute : "fieldset", "tabs", "choice"
  - → Tabs defined with CSS



Enter an **email** address: *

Select something that's 4 characters long: a / aa / aaa / aaaa

Choose a panel: String fields

Enter an **email** address: bar@www.foo.com *

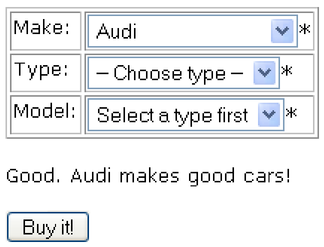
Select something that's 4 characters long: a / aa / aaa / aaaa

Interactive forms

Server-side event handler, client-side trigger

```
<wd:field id="make" required="true">
  <wd:label>Make:</wd:label>
  <wd:datatype base="string"/>
  <wd:selection-list src="cocoon:/cars" dynamic="true"/>
  <wd:on-value-changed>
    <javascript>
      var value = event.newValue;
      var type =
        event.source.parent.getWidget("type");
      if (value == null) {
        type.setSelectionList(new
          EmptySelectionList("Select a maker first"));
      } else {
        type.setSelectionList("cocoon:/cars/"+value);
      }
      typewidget.setValue(null)
    </javascript>
  </wd:on-value-changed>
</wd:field>
```

```
<wt:widget id="make">
  <wi:styling submit-on-change="true"/>
</wt:widget>
```


Make: Audi *
Type: – Choose type – *
Model: Select a type first *

Good. Audi makes good cars!

Change the type selection list

Linking forms to application data

An additional binding definition file

- Associates widget names to XPath expressions on the data model
  - Example : binding to an XML document

```
<wb:context
  xmlns:wb="http://apache.org/cocoon/woody/binding/1.0"
  xmlns:wd="http://apache.org/cocoon/woody/definition/1.0"
  path="user" >

  <wb:value id="email" path="email" readonly="true"/>

  <wb:value id="number" path="number/@value">
    <wd:convertor datatype="long"/>
  </wb:value>

  <wb:value id="choose" path="choose/@value">
    <wd:convertor datatype="boolean"/>
  </wb:value>
</wb:context>
```

Set the context of included paths

Associates a widget to a path

Read-only widget

Binding convertor (XML is text)

Putting it all together

The woody.js library

- Provides a Form class
  - Constructor takes a form definition file
- Method Form.showForm() to display the form
  - Returns when validation ok or non-validating submit
    → Internal loop on sendPageAndWait()

```
function edit_header() {
  var data = Application.getData();
  var form = new Form("forms/profile-header-def.xml");

  form.createBinding("forms/profile-header-binding.xml");
  form.load(data);

  form.showForm("view-profile-header.html", {foo: bar});

  if (form.submitId == "ok") {
    form.save(data);
    sendDialog("Thanks a lot");
  } else {
    sendDialog("Bye bye");
  }
}
```

Load app. data

Show form and wait

Test submit button

Save app. data

Putting it all together

The sitemap

```
<map:match pattern="edit-*.html">
  <map:select type="method">
    <!-- GET : start the flow for this screen -->
    <map:when test="GET">
      <map:call function="editor_{1}"/>
    </map:when>
    <!-- POST (form submission) : continue the flow -->
    <map:when test="POST">
      <map:call continuation="{request-param:continuation-id}"/>
    </map:when>
  </map:select>
</map:match>

<map:match pattern="view-*.html">
  <map:generate type="jxtemplate" src="forms/{1}-tmpl.xml"/>
  <map:transform type="woody"/>
  <map:transform type="i18n"/>
  <map:transform type="xslt" src="resources/editor-styling.xsl"/>
  <map:serialize type="html"/>
</map:match>
```

Selection by http method: form's action is "" (same URL)

"editor_" prefix restricts access to flowscript functions

JXTemplate to use showForm's context data

Conclusion

- A powerful form framework
  - Rich datatypes and validation rules
  - Easy extension to specific needs
  - Event handling for sophisticated interaction
  - Fancy builtin stylesheets
  - Easy to use with flowscript
- A community development
  - Initiated by Outerthought
  - Welcomes additions and contributions

→ Woody will be *the* form framework for Cocoon

See also http://wiki.cocoondev.org/Wiki.jsp?page=Woody

Questions ?

Answers !