Cineca UI Prototype - Documentation

Introduction

The documentation structure reflects the guidelines for the prototype challenge. For any point we discuss the functional and technological approaches to highlight benefits and future improvements, moreover we suggest possible extensions

1. User Interface Layout: The prototype UI should also fulfill these general layout requirements :

a. The prototype should have a header and a footer.

Functional approach.

Reference:

Sitemesh is an engine to decorate contents of the page http://wiki.sitemesh.org/wiki/display/sitemesh3/SiteMesh+3+Overview

Londinium template: a collection of css and js (we buyed the license for the product that can be donated to DuraSpace or buyed directly - it is very cheap - and allow unlimited use in enduser free product) for the layout of the site.

http://themeforest.net/item/londinium-responsive-bootstrap-3-admin-template/6978619

Using sitemesh a developer can concentrate on functionality (eg submission functionality) without worrying about the external layout because it will be added afterwords by sitemesh. The external layout is called “decorator” and you can define multiple decorators based on the url.

Technological approach

Reference:

http://wiki.sitemesh.org/wiki/display/sitemesh3/Getting+Started+with+SiteMesh+3

http://demo.interface.club/?theme=londinium (layout section)

.navbar class is the header and contains:

- site logo (.navbar-header - .navbar-brand)
- contextual buttons (eg new submission) - shown with “menu” project - identifier“/top.menu”
- login button
- for user logged on - the buttons for switching context (authorization)
- for user logged on - a menu containing particolar link userfull for the user. - shown with “menu” project - identifier “/my.menu”

.page-container is the center of the page and contains:

- sidemenu (on the side of the screen) for navigation - shown with “menu” project - identifier“/module.menu”
- list of breadcrumbs and the “language selector” (.breadcrumb-line)
- the content of the page (functionality decorated by sitemesh)
- footer (.footer)

b. The prototype should have something to help an end user keep track of their location in the system (e.g., a breadcrumb trail).

Functional approach.

For each user viewing any pages in the system, a list is produced of all visited pages. This list is displayed to the user in the breadcrumb box (.breadcrumb-line). Local navigation is implemented using “CLEAR” and “breadcrumbBack” parameters, to make sure that the breadcrumb faithfully reflects the user's browsing sequence. The system detects when a user has made back / forward and restores the breadcrumb.

Technological approach

Breadcrumb is implemented by a servlet filter (it.cilea.core.breadcrumb.filter.InMemoryBreadcrumbFilter).

This filter saves a list of all visited urls in the user's session.

Passing the url parameter "CLEAR" the list is cleared.

Passing the url parameter "breadcrumbBack" the sytems presents the previously displayed page.

Passing the url parameter "_form = 1" the url is not saved in the breadcrumb (useful when a form is submitted).

The filter identifies if a user has already passed through that url and resets all subsequent breadcrumb.

On initialization, a list of url "excludeUrl" urls can be passed (also as * .ajax or / item / *) to be excluded from the breadcrumb history.

c. The prototype should have a menu or sidebar that contains context-sensitive options.

Functional approach.

Menus are displayed via the "menu / TreeNode" project. It is possible:

- Display menus both in the top bar and the side bar
- Display drop-down menus (father + children)
- Each menu can be hidden / displayed based on
    - user licenses
    - rules on the path (where the user is located)
    - other generic rules (determined by the administrator)
- Menus can be changed at runtime without having to stop / start tomcat (only after the administrator has completed all the changes they are "approved" by displaying them to all. Until then, users see the old menu).
- Menu may have attributes "click" or "href".

The menu UI is done by Londinium graphics. Heavy customization is possible editing the .tag files or via CSS

Technological approach

When starting the tomcat a listener (TreeNode StartupListener Worker) loads table data on the identifierTreeMap map (attribute of ServletContext). This map is used by the tag system to display the menu.

The administrator can edit table data but until the reload menu is invoked (/menu/reload.htm of TreeNodeController) the system continues to use the map in memory to optimize memory and performance (in fact we avoid calling the DB for data that are normally static).

The menu data is stored in "tree_node*" tables represented by the classes:

- TreeNode: contains the individual menus and the relationships between them. Each TreeNode item is characterized by:
    - identifier: is the unique identifier of the menu item. It is used in tags to display the menu item (and / or his sons)
    - treeParentNode: is the parent menu (if root then empty)
    - brotherOrder
    - link/onclick/onmouseover
    - attachedResource: the menu will be displayed only if the logged user has the priviledge for the resource indicated in this column (see "authorization)
    - visibilityPath: the menu will be displayed only if the logged user is in a given path (eg. the "item / new" is displayed only when the user is in / item / *)
    - visibilityRule: it is a rule of javascript/rhino type that allows the administrator to define a particular rule without defining an adhoc Java class (eg. you can view a menu only when there is a certain parameter in the request or the user's username begins with a certain string)
- TreeNode Dictionary: contains any additional dictionaries (eg. the custom label to be assigned to a menu)

Menus are displayed on screen using the tag library http://www.cineca.it/menu. The menu tags (as all the tags of the framework) are written as tagfile "* . tag" version 2.0. This allows easy customization and maintenance. The tagfiles are found in "resources/META-INF/tags. The library definition is in resources/META-INF/tlds

- menu:navbarMenu with attributes
    - item: the instance TreeNode to be displayed (identified by the map identifierTreeMap - eg. $ {identifierTreeMap ['/ module.menu' ']})
    - top: if the menu is on the top menu then set to true
    - divId: id to assign to the menu div parent
    - includeThisNode: whether or not to display the menu "item" or start from children

This tag displays the item and / or direct children and all further branches etc. The menu layout is:

- collapsible (class collapse) if on the side
- downhill / waterfall if on the top
- menu: navbarTopMenu with attributes
    - item: the instance TreeNode to display (identified by the map identifierTreeMap - eg. $ {identifierTreeMap ['/ top.menu' ']}).

This tag displays only the direct children of this item in "button" formtat. Obviously for each item permissions are checked (whether or not to see the menu item for a certain user) and any path rules. Eg. the "new submission" button appears only to holders of the resource.

d. The prototype should allow users to go to the main locations in the system using a url: the browser history, back, and previous buttons, should all work.

Each page in the system is addressable. Each part of the system is subject to a controller and each url is distinct. SpringMVC is configured to answer calls like:

- * .htm: in this case the "default" layout responds
- .ajax: no layout responds

The SpringMVC controllers are made in two modes (ItemController as an example):

- path without parameters "/ get" method getItem responds
- path variables: "/ show / itemId / itemId {}" method showItem responds

In the method signature, you can define (spring intercepts the call and injects their values):

- @PathVariable
- @RequestParam
- HttpServletRequest
- HttpServletResponse
- @ModelAttribute ("Command") (it invokes the method characterized by annotation@ModelAttribute ("command") and the result will be injected in the controller method)
- BindingResult: possible binding errors

e. The prototype should display and function well on screens of all sizes (responsive / mobile friendly, not native mobile).

Sitemash allows to plugin any html/css template. The use of bootstrap and specificallyof the proposed Londinium layout that indeed is a bootstrap template meet the requirements

2. Simple Item View: The prototype should have an equivalent of a simple item view, which displays key metadata about the item, along with links to downloadable files.

The ItemController controller with the two methods listed above allows you to view the item details

The controller returns a ModelAndView:

- model: the instance of the item to display. In particular metadata will be displayed using the "metadataFieldPlaceMap" map
- view: the showItem “page”. When viewing this page, the "ViewBuilder" project intervenes (see dedicated item)

The list of widgets (shown in jsp or viewBuilder) defines the metadata display.

For more details see the "ViewBuilder" section.

A logic for displaying different views based on collection or other parameters can be created.

We defined both

1. ViewBuilder (row in the view_builder table) in state “active”
2. Jsp (it is not shown because the ViewBuilder is in state “active”. Change it to show the jsp instead of the ViewBuilder from DB

3. Community / Collection View: The prototype should have an equivalent to a Community /Collection view page, which displays key metadata about a community or collection

See the Community/Collection controller. Modify the jsp page or load a new ViewBuilder for displaying more information

4. Browse Navigation: The prototype should allow basic browse navigation of communities and collections and items.

Community/Collection controller (this is a simple implementation)

5. Authentication: Users should be able to authenticate themselves via simple password authentication.

The authentication stack is done by Spring Security. A major proposal not implemented in the prototype due to the set time limit is to embed a Central Authentication Service (CAS Jasig) with DSpace. In such way we can decouple DSpace from the backend solution for user's authentication relying on a dedicated community which focuses on such aspects. Another important benefit will be to have a SSO solution integrated with DSpace that allow easy integration with other platforms or external tools developed for specific functionality. The embedded CAS server can be configured also to act as a proxy of an existent CAS server or to support multiple authentication methods. The supported authorization methods for CAS already extend the present capabilities of DSpace.

6. Authorization: A very basic example of authorization operations should be supported. For example, only allow editing of items to authenticated users in the Administrator group.

Functional approach.

The authorization system uses an approach based on functions that aggregate permits on the platform. The system provides the ability to associate these functions to the various user profiles, so to authorize the single action based on the privileges offered by the function. This approach improves and simplify the existing DSpace authorization system based on ACL. The existent approach requires for example that authorization need to be coded when the object is created introducing lot of exceptions (poor code readibility, lot of duplicate code and errors prone for extensions). DSpace ACL approach associates an authorization to an "atomic" action on the object (eg. write on an item), while in the proposed system based on functions this authorization is intrinsic on the operations that the function allows. The current actions managed by the DSpace ACL are often corse-grained or fine-grained respect to the fucntaionlity that the users need. For example, the creation of a new object requires several permission on different objects, both ADD and WRITE instead the functionality to load new file in an item requires an extra WRITE permisison on the item that could result in unliked behaviour from the user. Shifting the focus on the user profile, and the single functionalities as perceived by the user, any further changes on access behavior to various resources and capabilities is more easy and flexible. Each user can belong to multiple profiles, and can be associated with various contexts. The user then has the option, in the same session, to switch between different contexts, and to have distinctly different user profiles based on the role.

The context in terms of the security is the environment where a specific functionality can be executed, for istance an item can have several context

- the item itself: in this way an user can be granted for a specific functionality on a specific item
- the collection (and all the hierarchy) where the item has been submitted
- the submitter, or other users that has been connected to the item (as for example the authors); in this way is possible to grant to the item's authors specific functionalities
- the concept can be extended to allow more sophisticated logic as for istance the support of a hierarchy on the users so that is it possible authorize a director of a deparment to perform some actions on all the items authored by her staff.

The ACL approach is keep in place only for the READ action on the bitstreams where puntual authorization are the best solution to provide embargo, open acccess and restricted access capabilities.

Technological approach

Authorizations exploit the potential of the Spring Security framework. The access logic to the resource is through the Functional approach expressed in the "Functional approach" paragraph. The access logic to resources is injected through IoC (Inversion of Control) via Spring bean with all the benefits that derive (eg. easy implementation and installation of a new access logic)

Example:

What does this approach mean? On the view it is possible to use the Spring Security tag library or a custom tag library to verify access to a certain authorization or function, e.g. menus that can be accessed by a user. There is no longer a need to code behaviors in the Business Logic such as turnoff of the authorization system when an object is accessed or modified. Just check for a single permission tailored on the functionality before to execute the functionality itself, the same check is exposed and shared with the view so that the view can reflect easily and exactly the allowed behaviuor for the user. This approach improve also a lot the performance as the check need to be executed only one time during the execution of the business logic and, as the function_context matrix is much smaller and more "stable" (don't change when new objects are created) can be easly put in a memory cache.

7. Edit/Create Item: The prototype should offer a basic capability to edit or create items via a simple web form. (Please no submission workflows as that is out of scope!) A single page form can serve both edit and create purposes. Not all metadata fields need be included, we are just looking for a simple example of a web form, including the ability to upload a file.

As in Step 2 (how to display the detail of the item) we created two controllers (could be one but for convenience we have created two) to create (ItemNewFormController) and edit (ItemFormController) an item.

No logic related to the metadata is inserted into the controller, as it is all defined in the "view" of the item (/ item/new and /item/form). A logic can be created for displaying different view based on the collection or other parameters.

For more information refer to the "View Builder" section.

Customization Capabilities

A new DSpace UI should allow for the following common UI customization options. Based on your prototype, please either show an example of how these may be achieved in this UI platform or describe them via documentation .

1. Show or describe how an administrator would be able to easily adjust the site wide theme layout based on local design/needs. Specifically, show or describe how the following changes might be achieved:

a. How would someone change the colors, fonts, sizes of the site? (e.g. css changes)

A CMS open source project (https://github.com/Studio-42/elFinder) has been integreated to allow easy management at runtime of any static resources.

Every image and stylesheet are located under dspace-jspui/src/main/webapp/sr.

The decorator page imports the stylesheet using this order:

1. first the css of the framework we leverage on (londinium, ckeditor, etc.)
2. the “/cineca” css that overrides some property of this css. It is better to change this file (adding properties that override londinium behaviour for example) for better compliancy with the framework.

To modify a file you can simple edit it in the structure file, or load it with the CMS.

The system detects the url jspui/sr and finds if the static resources must be loaded from the system or from the assetstore (CMS)

1. go to configuration/CMS
2. override an existing css (eg. load the cineca.css on /root/assetstore/jspui/sr/cineca/css) loading through the interface
3. reload the page

b. How would someone modify the sitewide header/footer? (e.g. to change logo, title, etc)

All the decorators are located under /dspace-jspui/src/main/webapp/decorator.

Simply edit the “default.jsp” page. With the “<sitemesh:write property='body'/>” you can write the “body” contained in the page to be decorated.

c. How would someone adjust the navigation bar to appear on left or right?

In the default.jsp decorator replace

<body class="sidebar-wide">

with

<body class="sidebar-wide sidebar-right">

d. How would someone change the location of the breadcrumb trail (e.g. from header to footer)?

In the default.jsp decoratore move the

<%@ include file="/common/breadcrumb.jsp"%>

in the south.jsp page

e. How would someone display additional metadata fields on the Item view page? For example, displaying “dc.publisher” on the default Item view site wide?

Two ways:

Jsp mode

If the item/get view is not a ViewBuilder (or the given row of the viewbuilder isn’t in “active” state), simple add:

<widget:text propertyPath="command.metadataFieldPlaceMap[dc_publisher_1]" allowMultiple="false"/>

ViewBuilder mode

1. In widget table add a row with discriminator “command-text” and model_attribute_name, page_attribute_name and name equals to “command.metadataFieldPlaceMap[dc_publisher_1]”
2. add a row in view_builder_widget_link with fk_view_builder=ID_OF_GET_VIEW and fk_widget=ID_NEW_WIDGET
3. reload view (from menu) or restart tomcat

f. How would someone add new links or an entire section to the navigation menu (e.g. links to other university pages)?

1. Add rows in tree_node table and link it with fk_tree_node_parent with parent menu.
2. Reload menu with the link on the left (configuration/reload menu)

Modularization Capabilities

A new DSpace UI should allow for / support common modularization needs. Based on your prototype, please describe (via documentation) how you feel this UI platform may (or may not) be able to achieve the modularization examples below:

1. How could this UI platform support optional modules/features?

See also the section related to new extensions/addon.

Optional modules or features can be disabled/enabled using the Spring IoC. Also the registered URL in SpringMVC can be managed at runtime as the only mapping required in the web.xml is related to the spring dispatcher servlet.

a. For example, Embargo is an existing optional feature within DSpace. While it is disabled by default, many sites choose to enable it.

You can add a row in the “configuration” table with key=”embargo.enable” and value=”true” or “false”. The system will read this configuration (that overrides any configuration defined in file) and enable embargo

b. Enabling Embargo functionality requires additional metadata fields to be captured in the deposit form (e.g. embargo date, embargo reason).

You can define a “ifWigdet” that reads the embargo configuration. If true than all the embargo widgets (date, reason) will be shown. This “ifWidget” will be linked in every ViewBuilder so that simply changing the configuration will enable embargo.

c. Does this UI framework support the idea of easily enabling (or disabling) features that require UI-level changes? In other words, in this framework, do you feel enabling Embargo functionality could be automated via a configuration (e.g. embargo.enabled=true)? Or would it require manual editing of the deposit form to add additional metadata fields?

As explained before, you can simply turn it on at runtime by changing a value in the configuration table

2. How could this UI platform support new extensions/add-ons?

The use of SpringMVC simplify the creation of new functionalities and pages allowing an easy use of the Spring IoC and the registration of new URLs using the @Controller annotation. This allow for example additional modules to simply put new controller in a jar file in the classpath of the webapp to have the same available to the end users. The spring bind capability simplify the creation of new functionalities or extending existent ones as for istance it is sufficient to put the html input tag to have such information automatically reflected on the DTO used to save the user entered information. Validation, conversion (from String to Date, to int, etc.) and security on single field can be easily implemented reusing standard component (propertyeditors). When required custom property editors and validators can be written in a way that make its immediately reusable across the whole project

a. Assume that someone has created a new Streaming feature for DSpace which provides a streaming capability for Video/Audio content. How might this UI platform support overriding /overlaying the default Item View to make that streaming feature available?

The Spring MVC capabilities allows filtering and replace at runtime of the registered URLs. The viewbuilder component of the framework-lite allow a full customization at runtime of the item view page so to include preview capabilities or link external tools.

Prototype Documentation

Each prototype MUST be submitted with the following documentation. Some of this documentation aims to stimulate your feelings about of this prototype or how the UI platform might be extended to support more advanced UI features. Please base your answers on what you know (or have learned) about this UI platform during the prototyping process.

1. Describe the design of the prototype (e.g. technologies/platforms used, including version of DSpace, etc.)?

The prototype is based on spring-mvc, spring-security and sitemesh. The template is made with londinium.

Via spring-mvc we make generic controllers in order to carry out activities without changing the java part (using maps that are processed retrospectively by "ViewPage")

2. How do you install the prototype on a new system? (Note: we will be testing the installation of prototypes in order to evaluate the installation of the platform itself)

Described in the git page. Same as dspace installation

3. In your opinion how could i18n (internationalization) support be added to this UI prototype /platform in the future?

The I18n project is already integrated into the system. It comprises two parts:

- static: file properties loaded at the start

- dinamic: data loaded into the db at the start and editable subsequently

Furthermore an administrative view displaying all the editable “keys” is available

4. How would you envision adding theming capabilities to this UI prototype/platform in the future? In other words, how could local or third-party themes be installed or managed? Please refer to theme as a collection of styles, fonts, logo, and page overrides.

By changing the londinum theme with other layout or providing custom view logic selection if the theming is related to specific subpart of the repository (collections, communities)

5. How would you envision supporting common DSpace authentication mechanisms (e.g. LDAP, Shibboleth) in this UI prototype/platform in the future?

Spring-security will ensure all requirements. Move to the embedd CAS solution detailed in the authentication requirements section above will furhter simplify the task, relying on the contribution of a dedicated and specialized community

ViewBuilder

Functional approach.

The idea underlying ViewBuilder is to allow the developer to continue using the most preferred technologies (jsp or xml) and the administrator to make changes at runtime without the support of a developer (thanks to a graphical tool - yet to be developed).

In order to accomplish this,the part of the Model / Controller must be completely generic and no intervention of the developer is required to make changes. All logics are moved in the View that is in the hands of the user. In particular:

- what a specific model should display
- what should editable of a specific model
- which possible validations should be performed on a specific model

Technological approach

When tomcat starts up, a listener (View StartupListener Worker) loads the table data into the ViewConstant static class . This class is used internally by the system to display the viewBuilder "database".

The administrator can edit the table data, the system continues to use the old data (in order to avoid continuos calls to the DB) until the reload of viewBuilder is invoked (/view/reload.htm of ViewController)

When a controller tries to render a view (identified by a unique identifier - identifier), Spring will test the existence of this particular View in the map ViewBuilder.viewBuilderMap.

If it is in place, the displayable view is of a "db" type and it will be displayed by accessing data retrieved from the map. If it is not present, the system will display the physical jsp (that will be in /WEB-INF/jsp/identifier.jsp)

ViewBuilder data are stored in "view_builder *" tables and the " widget" are represented by the following classes:

- ViewBuillder is a particular instance of view and has a unique identifier (identifier). At the startup only views with a "state" value "active" are saved in the map ViewBuilder.viewBuilderMap.The ViewBuilder can be displayed using alternateIdentifiers (comma separated).
- Particular attention should be paid to the attribute "viewName". This will be the "real" JSP page to display. This page is created "for convenience" where some activities are performed by default. A number of generic pages have already been created:
- "ViewBuilder / edit" features the title of the object to be displayed on the top (and sets it as title page) displays any validation errors. Please add the tag "form" with the "action" correct.
- Adds some "input" by default., dispays the content of ViewBuilderWidgetLink by tags It adds buttons to save / cancel at the bottom
- "ViewBuilder / get" like edit without form and buttons
- "ViewBuilder / nothing": does not add anything, it only displays the contents of ViewBuilderWidgetLink (by tag)
- ViewBuilderWidgetLink: These are first level "widgets" display. See "widget"

- ViewBuilderValidator: for the widget type form the check validation of the submitted object can be invoked. Spring performs validation on the "bindings". ViewBuilder instead adds performing logical own assessment on the object to save. Eg. you can perform the following validation types:
- field: for each field to be validated, you can specify a record (required, length, min, max, mailformat). This annotation will trigger an annotation of that particular field. Eg. indicating the field "description" the annotation "notnull" ientails that during the saving of the file if the "description" field is not empty the saving will not occur and the submission form will appear
- bean / xml: are more general validations are, involve writing class java / xml to perform integrity check of the entire object to be saved
- Widgets: Widgets see section
- The View are displayed on the screen using the tag library and http://www.cineca.it/view http://www.cineca.it/widget. Tags are written as tagfile "* . tag" version 2.0. This allows easy customization and maintenance. The tagfile found in "resources / META-INF / tags. The library definition is in resources / META-INF / tlds

- Widgets
- Widgets are "fragments" of the page. Their task is to simplify the writing of normal html tags without even bothering to write all the various "div" / "span" layout as these are placed directly by Tag.
- Eg. writing the tag:
- "<Widget: text propertyPath =" command.ID "/>
- The html code will be generated:
- <Div class = "line-group form" id = "ID__line">
- <Div class = "line-with-label sm-2">
- <Label for = "ID" class = "control-label"> Identier </ span> </ label>
- </ Div>
- <Div class = "line-with content-sm-10" id = "ID__content">
- <Input type = "text" name = "ID" value = "dccd5a60-0061-4981-976e-ee7a8d867c42" id = "ID" maxlength = "255" class = "form-control">
- </ Div>
- </ Div>
- Or:
- A container div with class line and form-group (Londinium layout)
- A div (col-sm-2) containing the label "Identifier". The text "Identifier" is identified by "messages" searching the key label.ID.
- A div (col-sm-10) containing the input tag with name / id = "ID" and value-dccd5a60-0061-4981-976e ee7a8d867c42 calculated on the basis of the "command" istance passed on to the page
- Furthermore if a validation error occurred, you would see an error message below the input tag
- Furthermore if a specific piece of info was added (attribute infokey) it would display the info under the input tag
- The text tag has internally
- some calculated values (id, name, value, class, data type) according to the binding of spring
- saves the value of the "label" using the tag labelCore (see below) (which has passed the values calculated by spring)
- saves the value of the "input" using the tag textCore (see below) (which has passed the values calculated by spring)
- displayed on the screen the html code generated by the tag line that is responsible for the actual display of label and input. Tag line appears in a div container made of two divs side by side (width 2:10)
- There are different types of widgets:
- Container: contain other widgets (widgetLink). Eg. a "fieldset" or a "div" is an example of a widget container.
- form: tags are used to show the type of form (select, input, checkbox, radio button, etc.). The widget-type form are duplicated by the context:
- command: when the page contains a command object / model of spring. In this case you can use the introspection / bindings spring to automatically calculate values. Eg. indicating how propertyPath a text value "command.metadataFieldPlaceMap [dc_title_1]" tag will display an input whose name and id will be calculated according to propertyPath and the "value" will be that contained in the instance command (of type Item) passed on
- Cores: are used when the page is missing an object "command" of spring. In this case it is not possible to invoke spring or introspection to identify the values of the object whose values inputName / Value / Id will be passed on directly to the tag.
- html: code snippets html

E. g. writing the tag:

“<widget:text propertyPath=”command.ID”/>

the html resulting code is:

```
<div class="line form-group" id="ID__line">
 <div class="line-label col-sm-2">
  <label for="ID" class="control-label"><span class="line-label-text">Identier</span></label>
 </div>
 <div class="line-content col-sm-10" id="ID__content">
  <input type="text" name="ID" value="dccd5a60-0061-4981-976e-ee7a8d867c42" id="ID" maxlength="255" class="form-control  ">
 </div>
</div>
```

that is:

- A container div with class line and form-group (Londinium layout)
- A div (col-sm-2) containing the label "Identifier". The text "Identifier" is identified by "messages" looking for the key label.ID.
- A div (col-sm-10) containing the input tag with name / id = "ID" and the value dccd5a60-0061-4981-976e ee7a8d867c42 calculated from the instance "command" passed to the page

- moreover, if there is any validation error, an error message displays below the input tag
- moreover, if there is specified information (infokey attribute), an info displays below the input tag

The text tag internally has:

- some calculated values (id, name, value, class, data type) according to the binding of spring
- saved value of the "label" using the tag labelCore (see below) (to which it has passed the values calculated by spring)
- saved value of the "input" using the tag textCore (see below) (to which it has passed the values calculated by spring)
- displayed on the screen the html code generated by the tag line that is responsible for the actual display of label and input. Tag line appears in a div container made of two divs side by side (width 2 and 10)

There are different types of widgets:

- Container: they contain other widgets (widgetLink). Eg. a "fieldset" or a "div" is an example of a widget container
- form: tags are used to show the type of form (select, input, checkbox, radio button, etc.). The widget-type forms are duplicated according to the context:
    - command: when the page contains a spring command/model object. In this case you can use the spring introspection/bindings to automatically calculate values. E.g. indicating as propertyPath of a text the value "command.metadataFieldPlaceMap[dc_title_1]", the tag will display an input whose name and id will be calculated according to the propertyPath and the "value" will be the one contained in the command instance (of type: Item) passed to the page
    - core: cores are used when in the page a "command" object of spring is missing. In this case it is not possible to invoke spring or introspection to identify the values of the object, so the inputName/Value/Id values will be passed directly to the tag.

- html: fragments of html code
- includes: inclusion of jsp pages
- conditionals: this allows to display the linked widgets only if a particular condition is true.

All widgets can be defined at DB level (through their own class that extends Widget). At rendering the widget type DB will use the "tagfile".

Below is a list of all the widgets of type Class/DB defined. For the discriminator to be used in the dabatase see the "discriminator" section. For the parameters used during init see the “init” method:

- AutocompleteWidgets/CommandAutocompleteWidget: displays an autocomplete input type. During init "autocompleteUrl" is passed which is the url of json type that autocomplete invokes to display the results.
- CheckBoxWidget/CommandCheckBoxWidget: checkbox that extends itemsWidget (to display collection/option)
- DateWidget/CommandDateWidget: displays an input text of "calendar" type
- CommandFileWidget: input of file types
- HiddenWidget/CommandHiddenWidget: input of hidden type
- HtmlWidget/CommandHtmlWidget: displays a html/jsp fragment. The fragment is defined in the "EXPRESSION" parameter and it may also contain expression or other jstl tags. (e.g. Welcome ${userDetail.username})
- RadioWidget/CommandRadioWidget: radio, extends itemsWidget (to display collection/option)
- Select2Widget/CommandSelect2Widget: select of select2JS type, extends itemsWidget (to display collection/option)

- SelectAndMoveWidget/CommandSelectAndMoveWidget: select of selectAndMove type, extends itemsWidget (to display collection/option)
- SelectWidget/CommandSelectWidget: select of selectAndMove type, extends itemsWidget (to display collection/option)
- TextAreaWidget/CommandTextAreaWidget: textarea
- TextWidget/CommandTextWidget: input type text
- DisplayTagTableWidget: used to render table. It is widget container of DisplayTagColumnWidget type
- Div Widgets: renders a div. The widget links will be rendered in the div.
- FieldsetWidget: renders a fieldset. The widget links will be rendered in the fieldset.
- IfWidget: based on the condition "test" (type jstl/expression), it will show or not widgetLink
- IncludeJspWidget: displays the jsp page defined in the "page" attribute
- InfoLineWidget: displays an info
- TabsWidget: renders bootstrap tabs (TabWidget container)

Tagfiles are (for single attributes open the tagfile and see the initial list):

- widget/widgetCore: is the tag used by widgets of “DB” type to show the display according to single tagfiles

- checkbox/checkboxCore
- hidden/hiddenCore
- radio/radioCore
- select/selectCore
- text/textCore
- textarea/textareaCore

Tags for layout:

- line: it displays a div container of:
    - div label
    - div content
    - div error
    - div info
- label/labelCore
- error/errorLine
- warning/warningLine

- info/infoLine
- tabCore/tabsCore

What is missing:

- Authorization with spring-security (we have implemented the proposed approach in our Enterprise solution build on top of DSpace - IRIS, it was a changes too much extend to be included in the prototype so we have preferred document our proposal;
- I18n pages for managing i18n at runtime; as above, the prototype supports the idea of customizable and reloadable labels - edit and managing pages for such aspects can be added easily.
- command-text (and text.tag) is the only widget repeatable by now
- viewBuilder graphical editor
- item by collection page doesn't show any items
- the "model" project that defne a model for everything shown in the ViewBuilder. It's like the input-form xml file.
- search project: with search you can define any "query" (hibernate/sql/solr) with an xml file (or rows in tables) and handle (show/filter) the results with the ViewBuilder projects.

Quick tips:

- after first start, go to browse-community and create a new community and collections
- items page shows any items created in the system
- new item controller set the item "archived" by default
- after editing widget/viewbuilder/treenode table use the links on the left to reload the context
- get/list collection are jsp pages located under dpsace-jpui/…../WEB-INF/jsp/
- list item is a jsp page locate under dspace-jspui/----/WEB-INF/jsp/item/list.jsp
- get/edit item is a ViewBuilder defined in the view_builder_table
- you can "turn off" a view Builder row changing the state column (from active to none). In this way the jsp page will be shown
- even the "list" page can be a ViewBuilder: create your own with the "displaytag-table" widget. (you can find an xml definition of this widget in browseCollection.xml file)