Structured Form Submission

Why?

The data model of HTML form submission is a map — each form field has a name, and when submit it, the server retrieves the values by names. This model has several problems.

One is the lack of structure; if you have a repeated structure in a form, which in turn contains a nested repeated structure, just giving it a name doesn't allow you to distinguish the nesting structure. IOW, you can't tell ((A,B),(C,D)) from ((A,B,C),(D)) because the map just gives you a single list of 4 items. A similar related issue is when you put a checkbox in a repeated part of a form. If some are checked by some are not, the server can never figure out which ones are checked because that information is lost when a form is submitted.

Another issue is the scope of the name. In HTML, all the form fields need to have unique names. This is akin to have all your variables as global in a programming language, and in a modular extensible system like Jenkins, this is not preferable. Scoping of names also prevent a part of the form to be reused elsewhere in the same form.

To avoid these issues and further simplify the server side, in Jenkins the notion of the "structured form submission" is introduced.

What is structured form submission?

Structured form submission is a form submission where the data model is a JSON object tree, not a map. This is done by having a client compute the JSON representation at the form submission time, and send that to the server as a hidden text field (along with all the other fields.)

Structure is determined by several factors. First, the name of the form field becomes the JSON property name. For compatibility reasons, a name can have any "abc.def." kind of prefix, and the prefix portion will be ignored. So for example, in the simplest case, the following form will produce the JSON result on the right.

```
<form>
  <input type="text" name="my.name"/>
  <input type="checkbox" name="my.optin"/> Send me e-mails
</form>
```

```
{ name: "Kohsuke", optin: true }
```

Any intermediate tag can have the 'name' attribute, and that would group the descendants into an intermediate object. Consider the following example:

```
<form>
  <div name="first">
    <input type="text" name="my.name"/>
    <input type="checkbox" name="my.optin"/> Send me e-mails
  </div>
  <div name="second">
    <input type="text" name="my.name"/>
    <input type="checkbox" name="my.optin"/> Send me e-mails
  </div>
  ...
  <div>
    <input type="password" name="my.password" />
  </div>
</form>
```

```
{
  first: { name: "Kohsuke", optin: true },
  second: { name: "Jesse", optin: false },
  password:"secret"
}
```

If there are multiple elements with the same name, their values are aggregated into an array. Consider the following example:

```
<form>
  <div name="person">
    <input type="text" name="my.name"/>
    <input type="checkbox" name="my.optin"/> Send me e-mails
  </div>
  <div name="person">
    <input type="text" name="my.name"/>
    <input type="checkbox" name="my.optin"/> Send me e-mails
  </div>
  ...
  <div>
    <input type="password" name="my.password" />
  </div>
</form>
```

```
{
  person: [
    { name: "Kohsuke", optin: true },
    { name: "Jesse", optin: false }
  ],
  password:"secret"
}
```

The nesting can be arbitrarily deep.

File uploads

This section applies to Stapler 1.81/Hudson 1.246 and onward.

File uploads in the structured form are renamed to unique names, and the JSON tree will have the name of this unique form name. To remain backward compatible, this processing requires that the file INPUT element contains `jsonAware="yes"` attribute:

```
<form>
  <div name="person">
    <input type="text" name="my.name"/>
    <input type="file" name="my.key" jsonAware="yes"/>
  </div>
  <div name="person">
    <input type="text" name="my.name"/>
    <input type="file" name="my.key" jsonAware="yes"/>
  </div>
</form>
```

```
{
  person: [
    { name: "Kohsuke", key: "randomId1234567" },
    { name: "Jesse", key: "randomIdabcdefg" }
  ]
}
```

`staplerRequest.getFileItem("randomId1234567")` would return the file uploaded for Kohsuke, and `staplerRequest.getFileItem("randomIdabcdefg")` would return the file uploaded for Jesse.

Advanced topics

Sometimes, the mark up and the layout of the form makes it impossible for a grouped form fields to have a single common ancestor element (for example, often in a grouped form elements are spread across multiple table rows.) In such a case, you can put the nameref attribute to merge multiple tree of elements into one. This mechanism is mostly used inside Jenkins form layout tags behind the scene, so it's not something plugin developers would need to worry about.
Consider the following example:

```
<form>
  <table>
    <tr name="cvs" id="abc"><td>
      <input type="text" name="CVSROOT" />
    </td></tr>
    <tr nameref="abc"><td>
      <input type="text" name="module" />
    </td></tr>
    <tr name="svn" id="def"><td>
      <input type="text" name="URL" />
    </td></tr>
    <tr nameref="def"><td>
      <input type="text" name="module" />
    </td></tr>
  </table>
</form>
```

```
{
  cvs: { CVSROOT:"...", module:"..." },
  svn: { URL:"...", module:"..." }
}
```

If the nameref attribute points to a check box or a radio button INPUT element, the subordinate structure is only submitted when the INPUT element is selected/checked. This is convenient when such an INPUT element is used to control the visibility of nested form parts.

Writing server code

You can access the entire JSON tree by calling `StructuredForm.get(request)`, but such code is generally only necessary when you are in charge of the entire form submission. `Descriptor.configure()` and `Descriptor.newInstance()` take `JSONObject`, which corresponds to the form fragment that you contributed via `config.jelly`/`global.jelly`.

See some of the Descriptor implementations in Jenkins core as an example.

Databinding

`StaplerRequest` provides several data-binding methods from `JSONObject`, which greatly simplifies the object instantiaion from form data. See the javadoc of the `StaplerRequest.bindJSONXXX` methods for details.