# Cocoon Forms: Introduction (2.1 legacy document)

## Table of contents

# Table of contents

**Warning:**
This document was copied as is from the Cocoon 2.1 documentation, but has not yet been fully reviewed or moved to its new home.

# 1. Introduction

Cocoon has an advanced form framework. It is simply called "Forms" or "Cocoon Forms", but mostly referred to as **CForms**. Previously it was called "Woody".

Combined with other Cocoon technologies, such as flowscript and the JXTemplate generator, this provides a solid basis for building webapplications in Cocoon.

**WARNING:** CForms is still under development. We are currently working towards its first stable release.

CForms aims to be powerful enough to handle complex use cases while still being simple enough to be used by non-Java-programmers who want to add a form to their site.

Essentially to create a form with CForms you need to define two things:

- the form model
- the form template

The form model is defined by a **form definition**. This is an XML file describing the structure of the form, by declaring the widgets it consists of. This file doesn't contain any presentational information. Based on the form definition, a **form instance** can be created. This is a specific instance of the form that will hold actual data. The form defintion is to the form instance what a Java class is to a Java object, or what an XML Schema is to an XML document.

Since the form definition is simply described in an XML file, this means you can create forms without any knowledge of Java. Unlike some other form frameworks, you don't need to write Java classes to hold form data.

As stated before, a form consists of a number of widgets. A **widget** is an object that knows how to read its state from a Request object, how to validate itself, and can generate an XML representation of itself. A widget can remember its state itself, so unlike e.g. Struts, you do not have to write a form bean for that purpose. A widget can hold strongly typed data. For example, you can indicate that a field should contain a date. If you do this, you can be assured that after the form is successfully validated, and you retrieve its value, you will get a Date object. So your own business logic doesn't need to care about converting strings to other types, and all the locale and formatting related issues of this.

CForms contains a flexible set of widgets that should cover most needs. However like everything in CForms, you can add new ones of your own. One special widget is the repeater widget, which "repeats" a number of other widgets multiple times, as is needed to generate table-like structures. Widgets can thus have child widgets, so a form instance is effectively a **widget tree**.

The presentation of the form is (usually) handled by a form template. The form themplate is an XML file (e.g. an XHTML file, but this could be any markup) and on the places you want a widget to appear, you insert a special tag referencing that widget. After processing by the Forms Template Transformer, these tags will be replaced by the XML representation of the widget, which contains all state information of the widget (its value, validation errors, ...). These bits of XML can then be transformed to plain HTML by an XSLT. Note that this XSLT only has to know how to style certain kinds of widgets, but not individual widget instances itself. Thus one template in this XSLT can style all widgets of a certain type that appear on all forms you might have. Cocoon includes a flexible,

configurable XSLT that covers most needs.

Below a typical scenario is described to explain how things fit together:

Overview of CForms

- Initially, the controller logic asks the FormManager component to create a form instance based on a form definition (form definitions are cached, so creating an instance is very fast).
- The controller can then optionally pre-populate this form object with some data. To fill the form with data from a bean or XML document, a binding framework is available.
- Then the form is shown by calling a pipeline.
- When the form is submitted, the controller will let the form instance object process the request, so that all widgets can read their value from the request. Some might generate events, which will be handled by event handlers. Validation of the widget tree is also triggered, whereby all widgets will validate themselves based on widget validators described in the form definition. The controller can afterwards perform application-specific validation logic.
- If there were validation errors, the form will be redisplayed. Otherwise the controller will decide what's the next step, for example saving the form data back to a bean or calling some backend process using data from the form.

**Next step:** have a look at a concrete sample to get a feel for how things work.

## 1. Comments

add your comments