На платформе Alfresco ECM

Формы в Citeck EcoS

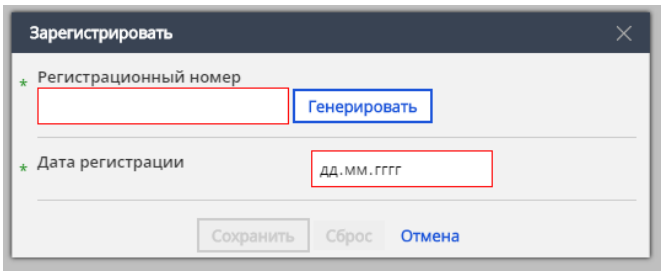
2016

Оглавление

Что такое формы....4
Структура форм....4
Шаблоны....5
Идентификация элементов....7
Параметры элементов....8
Файлы форм....10
Загрузка форм....11
Правила....11
Процедура заполнения формы....13
Некоторые варианты использования форм....14
Поддержка инвариантов....16
Страницы с формами....17
Советы по быстрой разработке....18

Что такое формы

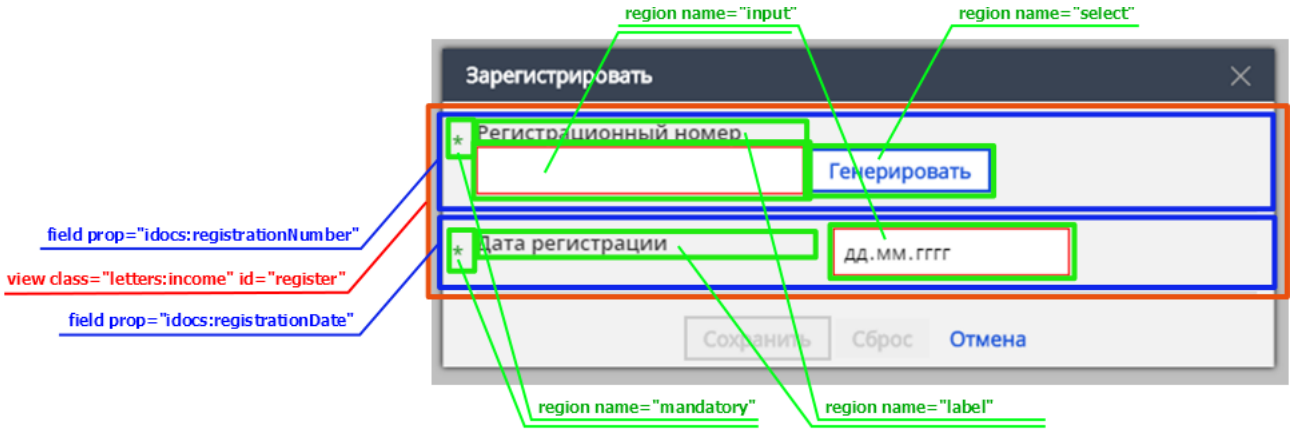
Форма – это графическое представление объекта репозитория в виде набора полей, соответствующих атрибутам объекта. С помощью форм можно создавать новые объекты, просматривать и редактировать существующие. Пример формы редактирования с двумя полями показан на иллюстрации ниже:



Структура форм

Формы состоят из блоков трех различных видов:

- `field` – поле, соответствует некоторому атрибуту;
- `view` – набор полей, вся форма – это тоже `view`;
- `region` – именованная часть поля, например, надпись (`label`), поле для ввода (`input`) или кнопка выбора (`select`).



На рисунке красным обозначен `view` (он же соответствует всей форме), синим – `field` (два поля – для атрибутов *Регистрационный номер* и *Дата регистрации*), зеленым – `region` (у первого поля регионы `mandatory, label, input, select`, у второго – `mandatory, label, input`).

Описание форм делается на языке XML, в частности для приведенной формы описание может быть следующим:

```
<view class="letters:income" id="register">
    <field prop="idocs:registrationNumber">
        <region name="select" template="number-generate">
            <param name="template">idocs-income-number-template</param>
        </region>
        <invariant on="mandatory" language="explicit">true</invariant>
    </field>
    <field prop="idocs:registrationDate">
        <invariant on="mandatory" language="explicit">true</invariant>
    </field>
</view>
```

Из этого описания видно, что `view`, `field` и `region` вложены друг в друга в XML. Структура формы, заданная в XML, и определяет то, как она будет выглядеть в интерфейсе пользователя.

Правила вложенности следующие:

- в элементы `view` могут быть вложены другие элементы `view`, `field` и `region`;
- в элементы `field` могут быть вложены элементы `region`;
- в элементы `region` ничего не может быть вложено.

Также видно, что в описании приведены не все регионы. Остальные регионы заполняются по умолчанию. Язык описания форм позволяет очень гибко управлять этими умолчаниями.

Наконец, из приведенного примера XML-описания формы видно, что для полей можно задавать дополнительные инварианты. Эти инварианты будут действовать только для данной формы. Подробнее об инвариантах можно узнать из документа «***Моделирование контента в Alfresco ECM и Citeck EcoS***».

Шаблоны

У каждого элемента (`view`, `field`, `region`) должен быть определенный шаблон (`template`). Шаблон определяет внешний вид элемента в интерфейсе. Шаблон можно задать явно в разметке формы, или он может быть назначен по умолчанию через правила (см. раздел Правила).

Для явного задания шаблона его указывают в атрибуте `template`:

```
<region name="select" template="number-generate" />
```

Для каждого типа элементов формы (`view`, `field`, `region`) свой набор шаблонов:

- `view`:
    - `table` – табличная верстка – одинаковые регионы разных полей располагаются друг под другом: надпись под надписью, поле ввода под полем ввода, подсказка под подсказкой и т.п. Все вложенные `view` имеют шаблон по умолчанию `rowset`, а все вложенные `field` – шаблон `row`.
    - `rowset` – набор строк в табличной верстке. Все вложенные view – шаблон `rowset`, `field` – `row`.

- `field`:
    - `row` – строка в табличной верстке. Выводит в одном и том же порядке регионы: `label`, `mandatory`, `input`, `select`, `help`, `message`.

  - row-checkbox – строка с галкой в табличной верстке. В отличие от row, выводит сначала галку (checkbox в регионе input), а затем – надпись (label). Порядок регионов: input, label, help, message
  - half-width – половина строки для табличной верстки. Вывод регионов аналогичен row.
  - single - полностью занимает строку и содержит в себе всего один region, занимающий все пространство.
  - none – не выводит поле.
- region:

  шаблоны, предназначенные для регионов input:
  - view – выводит текущее значение поля
  - view-table – выводит текущее значение поля в виде таблицы
  - text – однострочное текстовое поле
  - textarea – многострочное текстовое поле
  - number – поле для ввода числа
  - checkbox – выводит галочку
  - date – ввод даты
  - datetime – ввод даты и времени
  - select – выбор из списка значений (drop-down list)
  - autocomplete – выбор из списка значений с авто-дополнением

  шаблоны, предназначенные для регионов select:
  - select-orgstruct – выбор из оргструктуры
  - select-journal – выбор из журнала
  - create-object – создание нового объекта
  - number-generate – генерация номера по шаблону автонумерации (шаблон автонумерации указывается в параметре template)

  шаблоны, предназначенные для других регионов
  - mandatory – выводит звездочку (*), если поле обязательно. В режиме просмотра выводит восклицательный знак (!), если поле обязательно и не заполнено.
  - label – выводит надпись для поля
  - help – выводит вопросительный знак (?) с текстом подсказки для поля. В режиме просмотра не выводится
  - message – выводит сообщение о причине ошибке. В режиме просмотра не выводится.
  - free-cotnent - позволяющий разместить на форме произвольную информацию.
  - none – не выводит регион.

В шаблоны могут передаваться параметры, например, в шаблоне number-generate параметр template указывает, каким шаблоном автонумерации пользоваться:

```
<region name="select" template="number-generate">
    <param name="template">idocs-income-number-template</param>
</region>
```

Идентификация элементов

Большинство элементов формы имеют идентификаторы.

Поле (`field`) всегда соответствует какому-то атрибуту (свойству или ассоциации). Для того чтобы задать атрибут, которому соответствует поле, используется следующая нотация:

```
<field prop="idocs:registrationNumber" />
<field property="idocs:registrationDate" />
<field assoc="letters:reporterOrganization" />
<field association="letters:subdivision" />
```

Набор полей (`view`) может представлять собой либо всю форму, либо часть формы. В случае если `view` определяет всю форму, он должен соответствовать какому-то типу, для которого эта форма предназначена.

```
<view class="letters:income">
    ...
</view>
```

Если же `view` – это просто набор полей внутри формы, то у него обычно нет никакого идентификатора.

```
<view>
    ...
</view>
```

Регион (`region`) соответствует какой-то характеристике атрибута, но идентифицируется с помощью имени (`name`). Набор регионов для поля задается по умолчанию, поэтому регионы имеет смысл помещать в поле только если для них есть какая-то дополнительная конфигурация, например, шаблон (`template`) или параметры шаблона (`param`).

```
<region name="select" template="select-orgstruct" />
```

Если на форме указано несколько элементов с одной и той же идентификационной информацией, то они считаются одним и тем же элементом, их конфигурация совмещается и на форму выводится только один элемент. Если по каким-то причинам требуется вывести один и тот же элемент несколько раз, то для их различия используется атрибут `id`. Например, можно вывести несколько различных кнопок для выбора значения различными способами:

```
<region name="select" id="select" template="select-journal" />
<region name="select" id="create" template="create-variants" />
```

Аналогично можно добавить несколько раз одно и то же поле, или сделать несколько различных форм для одного типа. Например, можно определить общую форму для входящих писем и сделать отдельные формы для указания дополнительных атрибутов, например, форму регистрации входящего, как в примере выше.

```
<view class="letters:income"> ... </view>
<view class="letters:income" id="register"> ... </view>
```

Параметры элементов

Шаблоны для элементов `region` представляют из себя контролы для отображения и добавления данных на форму. Разберем уникальные параметры, которыми обладают некоторые из них.

autocomplete

`minQueryLength` — минимальная длинна поискового запроса, при которой будет сделан запрос. По умолчанию минимальная длинная равна 3.

`searchScript` — название поискового скрипта, с помощью которого, будет сделан запрос. На выбор предоставляется 2 скрипта:

- `criteria-search` имеет широкий список параметров для поиска элементов. Критерии поиска соединяются через И, тем самым, увеличивая точность поиска. Этот скрипт используется по умолчанию.
- `light-search` более легковесный и предназначен для более широкого поиска элементов. Критерии соединяются через ИЛИ, таким образом, увеличивается количество предположительно подходящих элементов.

`criteria` — через этот параметр передаются критерии, которые будут использованы для поиска элементов. Если параметр не задан, то поиск будет происходить по всем полям.

Задать критерии поиска можно 2 вариантами:

- использовать параметры: `attribute` и `predicate`. Это устаревший вариант. Минус такого подхода в том, что задать можно только один критерий.
- Использовать параметр `criteria`. В качестве значения этот параметр принимает массив объектов с ключами: `attribute`, `predicate`.

create-object

`source` — метод, с помощью которого будут найдены варианты создания. Значение может быть либо пустым, либо содержать строку `create-views`.

`buttonTitle` - название кнопки. Принимает либо текст, либо ключ локализации. Если значение не задано, то используется стандартное название «Создать».

`constraint` — дополнительное условие. Параметр задается в виде анонимной JavaScript функции, которая принимает объект с 2 свойствами: `oldValue`, `newValue`. Заданная функция должна возвращать значение типа `boolean`.

`constraintMessage` — позволяет задать сообщение об ошибке, в случае, когда дополнительное условие возвращает `false`.

journal

`journalType` — тип журнала. На основе данных, полученных по типу журнала, будут сконфигурированы параметры контрола.

`searchBar` — логический параметр, с помощью которого можно отобразить или скрыть поле поиска в шапке диалогового окна.

`defaultVisibleAttributes` — перечень атрибутов, которые должны быть отображены. Остальные атрибуты, относящиеся к объекту, будут скрыты.

`defaultSearchableAttributes` — список атрибутов, которые будут доступны для расширенного поиска.

`filterMode` — стиль отображения фильтра. Контрол имеет 2 режима: `collapse` и `full-page`. При первом варианте блок с фильтрами будет появляться сверху окна, при втором — появится отдельная вкладка в диалоговом окне. По умолчанию активен режим `collapse`.

Label

`text` — текст заголовка

`key` — ключ локализации

number-generate

`template` — название шаблона автонумерации. Шаблоны задаются в файле `enumeration-template.xml`

number

`step` — шаг увеличения значение

orgstruct

`allowedAuthorityType` — тип объекта. Доступны для выбора 2 значения: `USER` и `GROUP`. Так же есть вариант использование и первого, и второго. Нужно указать значение через запятую.

`allowedGroupType` — тип группы. Значение будет активно только в том случае, если в свойстве `allowedAuthorityType` имеется значение `GROUP`. Параметр принимает перечень типов групп, которые будут доступны для выбора. (например `branch`, `role` и т.д.)

textarea

`width` — первоначальная длинная.

`height` — первоначальная ширина.

view-table

Есть 2 варианта определить отображаемые столбцы для таблицы с элементами:

- задать параметр `journalType`. По типу журнала будут определены настройки журнала.
- Использовать параметр `columns` и явно перечислить список атрибутов, соответствующих столбцам, которые необходимо показать.

`maxheight` — максимальная высота контейнера с элементами.

`highlightedColumnMarker` — в качестве значения принимает название атрибута. Атрибут должен быть логическим. Если значение `true`, то вся строка будет помечена стилем `hightlighted` и выделена специальным светом.

`highlightedAdditionalClass` — атрибут принимает строку с названием класса стиля, который будет добавлен к выделенному элементу в дополнение к `highlighted`.

Что бы использовать индивидуальные стили для таблицы и всех вложенных в нее элементов, можно воспользоваться параметром `style` и задать собственные `CSS` стили.

free-content

`html` — задает отображаемую информацию в формате `html`. При верстке можно использовать функции `knockout` для отображения данных о доступном поле. Разметка должна быть заключена в `<![CDATA[  ]]>`

`function` — работает практически аналогично атрибуту `html`, за исключением, что контент моделируется с использованием `JavaScript` функции. На выходе функция должна отдать либо `HTMLElement`, либо строку с `html` кодом. В коде функции так же можно пользоваться возможностями `knockout`.

`css` — стили для верстке, заданной через `html` или `function`

Файлы форм

Формы определяются в XML-файлах следующей структуры:

```
<?xml version="1.0" encoding="UTF-8"?>
<views xmlns="http://www.citeck.ru/ecos/views/1.0">
    <imports>...</imports>
    <!-- определения форм -->
</views>
```

Вместо комментария `<!-- определения форм -->` в файлах располагаются определения форм, как описано выше:

```
<view class="letters:income>
    ...
</view>
```

Все имена типов и атрибутов, указанные в `class`, `prop` и `assoc` должны быть из пространств имен, перечисленных в разделе `imports`. В этом разделе перечисляют пространства имен аналогично тому, как это делается в моделях:

```
<import uri="http://www.alfresco.org/model/dictionary/1.0" prefix="d" />
<import uri="http://www.alfresco.org/model/content/1.0" prefix="cm" />
<import uri="http://www.citeck.ru/model/content/idocs/1.0" prefix="idocs" />
<import uri="http://www.citeck.ru/model/letters/1.0" prefix="letters" />
```

Загрузка форм

Загружать формы можно при старте сервера через механизм bootstrap (аналогично тому, как загружаются модели и инварианты). Для этого в файле `*-context.xml` нужно указать следующие строки:

```
<bean parent="nodeViewsDeployer">
    <property name="location"
              value="alfresco/extension/views/letters-views.xml" />
</bean>
```

В свойстве `location` указывается, где располагается XML-файл с формами относительно корня classpath (папка `tomcat/shared/classes`).

Правила

Помимо самих форм в файлах форм можно перечислять правила.

Правила отличаются тем, что в них можно указывать разметку не для конкретных элементов (`view`, `field`, `region`), а для множеств элементов. Эти множества обозначаются специальными элементами `views`, `fields`, `regions`.

В элементах `views`, `fields`, `regions` необязательно указывать идентификационную информацию (то есть `view class`, `field prop`, `field assoc`, `region name`), хотя это не запрещается. Если в определении формы указать, например:

```
<fields prop="idocs:registrationDate" template="half-width" />
```

то это не приведет к добавлению поля для свойства `idocs:registrationDate`, однако все поля для этого свойства получат шаблон `half-width`.

Указывая атрибут `prop` для правила `fields`, мы ограничиваем область действия этого правила до всех полей с этим значением `prop`. Также область действия можно ограничивать другими ключами:

- `views: class + id + mode, class + id, id, mode, kind, template`
- `fields: prop + id, assoc + id, id, kind, nodetype, datatype, template`
- `regions: name + id, id, kind, template`

Рассмотрим эти ключи.

Для `view` (`views`) мы можем указать режим формы (`mode`), он может принимать одно из значений: `create`, `edit`, `view` (создание, редактирование и просмотр). При этом поддерживаются ключи `class + id + mode` и `mode`, т.е. можно задать правила, например, для конкретной формы в режиме просмотра и для всех форм в режиме просмотра.

Для `fields` мы можем указать `nodetype` и `datatype`. Ключ `datatype` – это тип свойства, указываемый в модели в характеристике `type`, например, `d:text`. Для ассоциаций `datatype` равен `d:noderef`. Таким образом, можно задавать стандартные шаблоны для полей различного типа, например:

```
<fields datatype="d:text">
    <regions name="input" template="text" />
</fields>
```

Ключ `nodetype` – это тип объекта, на который ссылается ассоциация, это указывается в модели в характеристике `target.class`. Для свойств с `datatype = d:category` ключ `nodetype` равен `cm:category`, для остальных свойств он не задан. С помощью `nodetype` можно задавать правила для ассоциаций с определенными типами данных.

Для всех видов элементов можно указать ключ `kind`. Ключ `kind` – это произвольный «вид» элемента, с которым можно связывать набор настроек через правила. Например, можно определить вид `autocomplete-groups`, который будет устанавливать для поля шаблон `autocomplete` с настройками для поиска групп:

```
<regions kind="autocomplete-groups" template="autocomplete">
    <param name="attribute">cm:authorityDisplayName</param>
    <param name="predicate">string-contains</param>
</regions>
<fields kind="autocomplete-groups">
    <regions name="input" kind="autocomplete-groups" />
</fields>
```

Далее можно просто добавить несколько полей с этим видом:

```
<field assoc="group1" kind="autocomplete-groups" />
<field assoc="group2" kind="autocomplete-groups" />
<field assoc="group3" kind="autocomplete-groups" />
```

Далее можно изменить правило для вида autocomplete-groups и все регионы с этим видом изменятся соответствующим образом.

Также можно создавать правила по шаблону. Например, можно указать для шаблона настройки по умолчанию:

```
<regions template="autocomplete">
    <param name="attribute">all</param>
    <param name="predicate">string-starts-with</param>
</regions>
```

При обработке правил перечисленные выше ключи перебираются по одному, и **первый подходящий** атрибут (или сочетание атрибутов), который задан в правиле, является ключом. Например, в правиле

```
<views mode="view" kind="custom" template="table" />
```

ключом является атрибут `mode`, а `kind` и `template` ключами не являются. Другими словами, **Правильная** интерпретация этого правила всего одна:

- для всех `view` с режимом `view` должен быть вид `custom` и шаблон `table`

Неправильные интерпретации:

- для всех `view` с видом `custom` должен быть режим `view` и шаблон `table`
- для всех `view` с шаблоном `table` должен быть режим `view` и вид `custom`

- для всех `view` с режимом `view` и видом `custom` должен быть шаблон `table`
- для всех `view` с режимом `view` и шаблоном `table` должен быть вид `custom`
- …

Более того, указанные неправильные интерпретации не могут быть записаны на языке форм.

В XML порядок атрибутов элемента не влияет на смысл, поэтому следующие правила являются эквивалентными и обозначают то же: для всех `view` с режимом `view` должен быть вид `custom` и шаблон `table`:

```
<views mode="view" kind="custom" template="table" />
<views kind="custom" mode="view" template="table" />
<views template="table" mode="view" kind="custom" />
```

Поэтому во избежание путаницы рекомендуется записывать атрибуты в порядке следования ключей (см. выше), и тогда первый записанный атрибут (в данном случае `mode`) и будет ключом.

Если требуется, чтобы правило выполнялось вообще для всех элементов (например, для всех элементов view), то необходимо написать в правиле атрибут `any="true"`. Например, можно указать, что все поля на форме должны иметь шаблон `half-width`:

```
<view class="…">
    <fields any="true" template="half-width" />
    ...
</view>
```

Процедура заполнения формы

Описанные выше определения форм и правила используются для того, чтобы получать структуру формы по ее идентификатору. Допустим, требуется нарисовать форму просмотра для типа Входящий. Изначально о форме известен только тип документа (`letters:income`) и режим (`view`):

```
<view class="letters:income" mode="view" />
```

Далее начинается просмотр подходящих правил. Сначала ищутся определения и правила по ключу `class + id + mode`, то есть определения вида:

```
<view class="letters:income" mode="view" …> … </view>
<views class="letters:income" mode="view" …> … </views>
```

Затем ищутся определения и правила по ключу `class + id`, т.е. определения вида:

```
<view class="letters:income" …> … </view>
<views class="letters:income" …> … </views>
```

Затем ищутся только правила: по ключам `mode`, `template` и т.п.

В результате форма «обрастает» содержимым. Например, если было найдено правило

```
<views mode="view" template="table" />
```

то форма будет уже следующей:

```
<view class="letters:income" mode="view" template="table" />
```

После этого к форме уже будет подходить, например, такое правило:

```
<views template="table">
    <views any="true" template="rowset" />
    <fields any="true" template="row" />
</views>
```

После того как все подходящие определения и правила применены, начинается аналогичная процедура для дочерних объектов. Например, если в определении формы было задано поле

```
<field prop="idocs:registrationNumber" />
```

то благодаря предыдущему правилу (`<fields any="true" template="row" />`) к полю добавится шаблон `row`:

```
<field prop="idocs:registrationNumber" template="row" />
```

Процедура заполнения формы продолжается до тех пор, пока форма не будет полностью заполнена.

Некоторые варианты использования форм

Механизм форм позволяет определять для одного и того же типа различные формы в зависимости от режима (`mode`) или просто несколько разных форм, различающихся идентификатором (`id`). Например, если требуется, чтобы форма в режиме просмотра имела дополнительный набор полей (поля только для просмотра), то можно записать это следующим образом:

```
<view class="letters:income">
    <!-- определение формы для всех режимов -->
</view>
<view class="letters:income" mode="view">
    <!-- дополнительные поля для режима просмотра -->
</view>
```

В данном случае для режимов создания и редактирования будет подходить только первое определение формы, а для режима просмотра – оба определения, причем сначала будет найдено и применено второе, а затем первое.

Если требуется, чтобы форма просмотра не содержала какого-то поля, то можно указать этому полю шаблон `none`:

```
<field prop="…" template="none" />
```

Если же требуется, чтобы формы редактирования и просмотра вообще отличались, то это можно записать следующим образом:

```
<view class="letters:income" mode="create">
    <!-- определение формы для режима создания -->
</view>
<view class="letters:income" mode="edit">
    <!-- определение формы для режима редактирования -->
</view>
<view class="letters:income" mode="view">
    <!-- определение формы для режима просмотра -->
</view>
```

Если при этом требуется, чтобы режимы редактирования и создания были одинаковыми, то можно выделить общую часть, задав для нее отдельный идентификатор (`id`):

```
<view class="letters:income" id="create-edit">
    <!-- определение формы для режимов создания и редактирования-->
</view>
<view class="letters:income" mode="create">
    <view class="letters:income" id="create-edit" />
</view>
<view class="letters:income" mode="edit">
    <view class="letters:income" id="create-edit" />
</view>
```

Другой вариант разделения полей по режимам – использование разных видов полей (kind). Например, можно задать такие виды полей как mode-create, mode-not-create, mode-edit, mode-not-edit и т.п. со следующими правилами:

```
<views mode="create">
    <fields kind="mode-edit" template="none" />
    <fields kind="mode-view" template="none" />
    <fields kind="mode-not-create" template="none" />
</views>
...
```

Далее можно определить всего одну форму и используя различные виды ограничивать видимость полей в разных режимах:

```
<view class="letters:income">
    <field prop="…" kind="mode-view" />
    <field prop="…" kind="mode-not-create" />
    ...
</view>
```

Недостатком такого подхода является то, что для скрытия поля используется атрибут template, поэтому если задать и kind и template, то поле не будет скрыто:

```
<!-- это поле не будет скрыто в режиме просмотра -->
<field prop="…" kind="mode-not-view" template="text" />
```

Можно задать просто несколько разных форм для объекта, различающихся идентификаторами, например, обычная форма и форма регистрации:

```
<view class="letters:income">
  <!-- определение обычной формы →
</view>
<view class="letters:income" id="register">
    <!-- определение формы регистрации -->
</view>
```

Далее обратиться к форме по идентификатору можно на странице с формой, для этого достаточно корректно сформировать ссылку (см. ниже).

Форма для одного и того же типа может быть распределена на несколько файлов. Например, для входящего документа существует стандартная форма в системе. Допустим, при внедрении понадобилось добавить во входящий документ одно поле. В модели это поле было добавлено в аспект, а для формы это можно указать следующим образом:

```
<view class="letters:income">
    <field prop="custom:incomeField" />
</view>
```

Возможно, при внедрении пошли другим путем и сделали подтип для входящего. Можно определить форму для нового типа на основе существующей формы просто включив ее:

```
<view class="custom:income">
    <view class="letters:income" />
</view>
```

При этом если требуется убрать какие-либо поля, то можно передать им шаблон `none`, а если требуется добавить, их можно добавить перед или после включения стандартной формы:

```
<view class="custom:income">
    <field prop="custom:newFieldBefore" />
    <view class="letters:income">
        <field prop="letters:fieldToRemove" template="none" />
    </view>
    <field prop="custom:newFieldAfter" />
</view>
```

Поддержка инвариантов

Для полей формы можно задавать инварианты:

```
<field name="…">
    <!-- поле обязательно на форме -->
    <invariant on="mandatory" language="explicit">true</invariant>
</field>
```

Эти инварианты будут действовать только на данной форме (можно определить формы, на которых они не будут действовать).

С помощью инвариантов можно поддержать, например, следующие сценарии работы формы:

Обязательность поля на форме:

```
<invariant on="mandatory" language="explicit">true</invariant>
```

Условная обязательность поля (поле обязательно, если статус документа не «Новый»):

```
<invariant on="mandatory" language="javascript">
    node.properties["idocs:documentStatus"] != "new"
</invariant>
```

Значение по умолчанию:

```
<invariant on="default" language="explicit">1</invariant>
```

Поле, защищённое от редактирования:

```
<invariant on="protected" language="explicit">true</invariant>
```

Можно задавать любые другие инварианты для любых поддерживаемых характеристик (`mandatory`, `default`, `protected`, `valid`, `...`) и они будут работать на форме.

Поскольку инварианты можно задать не только для формы, но и вообще для типов/аспектов и свойств/ассоциаций, то при задании инварианта необходимо ответить на вопрос: этот инвариант действителен только для этой формы или вообще для всего типа? Ответ на этот вопрос и определяет, какую область действия задавать для инварианта.

Страницы с формами

В EcoS доступно две стандартных страницы с формами: для создания новых объектов и для редактирования существующих.

Страница `node-create` предназначена для создания новых объектов. Она принимает следующие аргументы:

- `type` – тип объекта для создания
- `viewId` – идентификатор формы
- `destination` – папка, в которой будет создан объект
- `param_*` – передача произвольных параметров на форму

Например, страница создания входящих документов:

```
node-create?type=letters:income&destination=workspace://SpacesStore/...
```

Параметр `destination` может быть не указан, тогда объект будет создан в папке по умолчанию.

Страница `node-edit` предназначена для редактирования существующих объектов. Она принимает следующие аргументы:

- `nodeRef` – идентификатор объекта для редактирования
- `viewId` – идентификатор формы
- `param_*` – передача произвольных параметров на форму

Например, страница редактирования входящего документа:

```
node-edit?nodeRef=workspace://SpacesStore/...
```

и страница регистрации входящего документа:

```
node-edit?nodeRef=workspace://SpacesStore/...&viewId=register
```

С помощью параметров можно передать произвольные объекты на форму. Далее их можно использовать в инвариантах полей формы. Например, если передан параметр `param_outcome`, то в инвариантах можно на него сослаться следующим образом: `view.params.outcome`:

```
<field assoc="letters:outcome">
  <invariant on="value" language="javascript">
         view.params.outcome
   </invariant>
</field>
```

В этом случае если этот параметр задан, то поле для ассоциации `letters:outcome` (исходящий документ) будет автоматически заполнено и недоступно для редактирования (если нужно сделать поле доступным для редактирования, то можно использовать характеристику `default` вместо `value`). Например, страница создания входящего на основе исходящего:

```
node-create?type=letters:income&param_outcome=workspace://SpacesStore/...
```

Далее можно сделать в карточке исходящего документа действие «Создать входящий», которое будет переводить на эту страницу.

Советы по быстрой разработке

Возможна перезагрузка конфигурации формы без необходимости перезапуска всей `alfresco`. Обязательным требованием для таких манипуляций является установленный на `alfresco` модуль `javascript-console`.

Для перезапуска конфигурации форм следует в javascript-console выполнить следующую команду:

```
services.get('nodeViews.autoDeployer').load('alfresco/module/{module-name}-repo/views/{module-name}-views.xml');
```

Для перезапуска конфигурации инвариантов следует в javascript-console выполнить следующую компанду:

```
services.get('invariants.autoDeployer.module').load('alfresco/module/{module-name}-repo/invariants/{module-name}-invariants.xml');
```