DITACMS

Integrating DTDs into the DITA CMS

Last revised: May 26, 2014

Table of contents

Creating a shell DTD

Creating a shell DTD in the system configuration

Create your own shell DTD

The DITA CMS uses the file **IxiaDitabase.dtd**, which is found in *Repository/system/dtd/ixia*. This file is essentially a copy of the DITA 1.2 ditabase DTD with some additional items that are specific to the DITA CMS. This is the starting point for integrating any other DTDs into the CMS, but rather than make changes to this file, IXIASOFT strongly recommends that you create your own custom copy of it. While this is more trouble up front than simply editing **IxiaDitabase.dtd**, in the long run it will save you time and really is a best practice.

If you opt to create your own shell DTD, you should do so before you begin production work in the DITA CMS. If you begin production work using **IxiaDitabase.dtd** and later want to create your own shell DTD, you will need to change all of your topics' DOCTYPEs to reference your DTD rather than **IxiaDitabase.dtd**. This is not trivial to do and should be avoided by as much prior planning as possible.

1. **Copy IxiaDitabase.dtd to the** *client* **folder in** *Repository/system/dtd***.**

   To copy the file, you might first have to export it to your desktop and then reimport it to the *client* folder.

2. **Rename the copy CompanyDitabase.dtd (where “Company” is the actual name of your company).**
3. **If you use XMetaL, copy the XMetaL configuration files to the** *client* **subfolder.**

   In the *ixia* subfolder, there are three other IxiaDitabase files: **IxiaDitabase.css**, **IxiaDitabase.ctm**, and **IxiaDitabase.ctr**. These are configuration files for XMetaL, so if you are using XMetaL, you also need to copy those three files to the *client* subfolder and rename them to match the name you gave your custom DTD.

   There is also a file named **macros.dtd**, which is needed for XMetaL as well. You also need to copy it to the *client* subfolder, but do not change its name. If you are not using XMetaL, you do not need to copy these files to the *client* subfolder, but it doesn't hurt anything to do so.

4. **Check out and open CompanyDitabase.dtd.**

5. **At the beginning of the file, edit the header to reflect an appropriate name, version, and date for the DTD:**

```
<!-- ============================================================= -->
<!--                    HEADER                                     -->
<!-- ============================================================= -->
<!--  MODULE:    CUSTOMER DITA BASE DTD                            -->
<!--  VERSION:   1.0                                               -->
<!--  DATE:      April 2014                                        -->
```

6. **Just after that, edit the public identifier to be something unique.**

```
<!-- ============================================================= -->
<!--                    PUBLIC DOCUMENT TYPE DEFINITION            -->
<!--                    TYPICAL INVOCATION                         -->
<!--                                                               -->
<!--  Refer to this file by the following public identfier or an
      appropriate system identifier
PUBLIC "-//CUSTOMER//DTD DITA Composite//EN"
      Delivered as file "CompanyDitabase.dtd"                  -->
```

Remember that public identifiers are resolved through the catalog. You don't want a catalog to have duplicate identifiers pointing to different resources, as that will create confusion and problems with validation. You want to be sure that your DTD has a completely unique identifier. This section of the DTD does not connect the DTD and its public identifier in any way, but it's good to document here what the identifier used in the catalog is, just for reference.

7. **(Optional) If you want, you can delete the following section altogether, or use it to keep a revision history of the file:**

```
<!-- ============================================================= -->
<!-- SYSTEM:     Darwin Information Typing Architecture (DITA)     -->
<!--                                                               -->
<!-- PURPOSE:    Base DTD holding all the information types        -->
<!--             used by IXIASOFT                                  -->
<!--                                                               -->
<!-- ORIGINAL CREATION DATE:                                       -->
<!--             January 2006                                      -->
<!--                                                               -->
<!-- ============================================================= -->
```

8. **Take a look through the topic entity declarations in the next section.**

As you recall, the public identifier for each of these is still valid, because it points to the same catalog as **IxiaDitabase.dtd**. In addition, the system identifier relative path for each is still valid as well, because the *ixia* subfolder that **IxiaDitabase.dtd** lives in and the *client* folder that **CompanyDitabase.dtd** lives in are at the same level in the folder hierarchy. You don't need to change any of these entity declarations.

9. **Scroll forward to find the Domain Attribute Declarations section and make the same change as shown:**

```
<!-- ============================================================= -->
<!--                    DOMAIN ATTRIBUTE DECLARATIONS              -->
<!-- ============================================================= -->

<!ENTITY % localization-loc-d-dec
                      PUBLIC
"-//IXIA//ENTITIES DITA Localization Domain//EN"
"../ixia/localizationDomain.ent"                                   >
%localization-loc-d-dec;
```

10. **Scroll forward to find the Topic Element Integration section and make the system identifier path change as shown:**

```
<!ENTITY % referable-content-typemod
                      PUBLIC
"-//IXIA//ELEMENTS DITA Referable-Content//EN"
"../ixia/referable-content.mod"                                    >
%referable-content-typemod;
```

11. **Save and check in CompanyDitabase.dtd.**

You now have your own shell DTD. Going forward, you'll do all your integrations here instead of in **IxiaDitabase.dtd**.

Next, you need to create your own catalog.

Create a custom catalog

After you create your custom shell DTD, you need to create a catalog to ensure the DTD is recognized by the DITA CMS. Catalogs live in *Repository/system/catalogs***.**

1. **In** *Repository/system/catalogs***, make a copy of catalog-dita-ixia.xml and rename it appropriately.**

   Again, you might need to export **catalog-dita-ixia.xml** to your desktop, rename it, and reimport it. For this example, we'll use the name **catalog-dita-company.xml** (again, where “company” is the actual name of your company).

2. **Very near the beginning of the file, change the line that begins `<group xml:base` as follows:**

```
<group xml:base="../dtd/client/">
```

   so that it references DTDs in the *dtd/client* folder rather than the *dtd/ixia* folder.

3. **Delete everything inside the <group></group> tags except for the line that points to IxiaDitabase.dtd.**

   You should be left with only the following in **catalog-dita-company.xml**:

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog" prefer="public">
   <group xml:base="../dtd/client/">
      <public publicId="-//IXIA//DTD IXIA DITA Composite//EN" uri="IxiaDitabase.dtd"/>
   </group>
</catalog>
```

4. **Change the remaining publicId to match the public identifier of CompanyDitabase.dtd and the uri value to CompanyDitabase.xml:**

```
<public publicId="-//CUSTOMER//DTD DITA Composite//EN" uri="CompanyDitabase.dtd"/>
```

5. **After the </group> tag, add the following line:**

```
<nextCatalog catalog="catalog-dita-ixia.xml"/>
```

   By default, **catalog-dita-ixia.xml** is called first. Instead, you now want your catalog, **catalog-dita-company.xml**, to be called first. This nextCatalog ensures that your **catalog-dita-company.xml** calls **catalog-dita-ixia.xml**.

6. **Save and check in catalog-dita-company.xml.**

You now have your own catalog.

Next, you need to edit the system ids that the DITA CMS uses to identify which DTDs are recognized.

Point oXygen to your catalog

If you are using oXygen, you need to point it to your custom catalog, **catalog-dita-cust.xml**.

1. **In your CMS preferences, go to oXygen XML Author (or Editor).**
2. **Select XML, then XML catalog.**
3. **Click the + icon below the Catalogs list.**
4. **In the Choose Catalog dialog, click the folder icon and select Browse Workspace.**
5. **Browse to the Repository folder where catalog-dita-company.xml is stored and select it.**

   The catalog appears in the Catalogs list.

6. **Below the Catalogs list, click the up arrow to move catalog-dita-company.xml to the top of the list.**
7. **Click Apply, then OK to close the Preferences window.**

Related Links

Create a custom catalog on page 6

Point XMAX to your catalog

Because IXIASOFT developed the Eclipse plugin to integrate XMAX, the plugin automatically loads all the catalogs from the system configuration. You do not need to configure anything manually.

Related Links

Create a custom catalog on page 6

Edit system ids

After you create a custom catalog, you need to edit the system ids that tell the DITA CMS which DTDs to recognize when content is imported into the CMS.

1. **In *Repository/system/conf*, check out and open systemid.xml.**

   When you import DITA content that you've created outside of the DITA CMS, the CMS looks at the DOCTYPE reference. If the reference is not one that is recognized by the DITA CMS (according to **systemid.xml**), the DITA CMS automatically remaps it to whatever the topicdefault is (or mapdefault, if you're importing a map). By default, this file defines three public ids that are accepted and a default to map to. The default out of the box is the **IxiaDitabase.dtd**.

   If you want your imported content to be remapped to your custom shell DTD, it's critical to edit **systemids.xml** accordingly.

2. **In the first line, change the value of topicdefault to the public identifier of CompanyDitabase.dtd, as shown:**

```
<id mapdefault="-//CUSTOMER//DTD DITA Map//EN" topicdefault="-//CUSTOMER//DTD DITA Composite//EN">
```

   You'll notice that in addition to the topicdefault attribute that you just changed, there is also a mapdefault attribute. The DITA CMS uses one shell DTD for topics (**IxiaDitabase.dtd**) and another for maps (**IxiaMap.dtd**). If you want to create a custom map shell DTD, follow the

same process you've seen for **IxiaDitabase.dtd**. (Do not copy **IxiaMap.mod** to the client subfolder.)

3. **In the first reference line, change the public identifier to the public identifier of CompanyDitabase.dtd, as shown:**

```
<reference public="-//CUSTOMER//DTD DITA Composite//EN"
system="../../system/dtd/ixia/IxiaDitabase.dtd"/>
```

You must have a reference for the DTD that you are using as the default.

> **Important:** Note that you are replacing references to **IxiaDitabase.dtd** with references to your custom shell DTD. Your shell completely takes the place of **IxiaDitabase.dtd**. This is unlike your custom catalog **catalog-dita-company.xml**, which does not replace, but adds to the functionality of the default **catalog-dita-ixia.xml**.

4. **In that same line, change the system identifier path to reference CompanyDitabase.dtd, as shown:**

```
<reference public="-//CUSTOMER//DTD DITA Composite//EN"
system="../../system/dtd/client/CompanyDitabase.dtd"/>
```

5. **Save and check in systemids.xml.**

The DITA CMS is now set up to recognize **CompanyDitabase.dtd** as the default when importing or creating content.

Next, you need to edit the topic templates to include a DOCTYPE that references **CompanyDitabase.dtd**.

Edit the DOCTYPE in topic templates

After you set up the DITA CMS to recognize **CompanyDitabase.dtd** when importing or creating content, you need to edit the topic templates to include a DOCTYPE that references **CompanyDitabase.dtd**. These templates live in *Repository/system/templates/topics*. Depending on your configuration, there may be additional subfolders.

1. **Check out and open concept.xml.**
2. **Edit the DOCTYPE declaration so that it references the public and system identifiers of CompanyDitabase.dtd, as shown:**

```
<!DOCTYPE concept PUBLIC "-//CUSTOMER//DTD DITA Composite//EN"
"../../system/dtd/client/CompanyDitabase.dtd">
```

3. **Save and check in concept.xml.**

4. **Repeat the same steps for the remaining templates: task.xml, reference.xml, topic.xml, glossentry.xml, referable-content.xml, and any other templates you might have created.**

The DITA CMS is now set up to include the correct DOCTYPE when importing or creating content.

Test each of your topic types (and maps, if you created a shell map DTD) to be sure that you can create all of them without errors. You should be able to create a new topic, edit it, save it, and release it.

If you are using the Output Generator, you now need to integrate your custom DTDs into it.

Adding the shell DTD to the Output Generator

Create a DTD plugin

After you have successfully created a custom shell DTD and catalog, and everything is working as expected in the DITA CMS, you next need to make sure it will work in the Output Generator. The Output Generator has its own DITA Open Toolkit and set of DITA DTDs. Without some work, they will not recognize the new DOCTYPES you're using for your topics.

You'll start by making a copy of an existing plugin. Within the Output Generator, plugins live in */data/%OT_Dir%/plugins* (where %OT_Dir% stands for the name of the folder where the DITA Open Toolkit is installed, such as DITA-OT1.6.2.)

> **Note:** There are any number of ways you might want to create your plugin. The following is only one example.

1. **Copy the** *com.ixiasoft.dita* **folder.**
2. **Rename the folder appropriately.**

   For this example, we'll rename the folder *com.cust.dtd*.

3. **If they are present, delete the files catalog-ant-dita.xml and catalog-ant-ixia.xml.**
4. **Rename the file catalog-dita-ixia.xml to something more appropriate.**

   For this example, we'll rename it to **catalog.xml**, as that is a convention many plugins use. The name doesn't matter, but it's a good idea to be consistent in your naming.

5. **Open catalog.xml.**

   This file should look familiar, if you recall creating a custom catalog for integrating custom DTDs into the DITA CMS (**Create a custom catalog** on page 6).

6. **Delete everything inside the <group></group> tags except for the line that points to IxiaDitabase.dtd.**

   You should be left with only the following in **catalog.xml**:

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog" prefer="public">

 <group>
  <public publicId="-//IXIA//DTD IXIA DITA Composite//EN" uri="dtd/IxiaDitabase.dtd"/>
 </group>

</catalog>
```

   Important: Notice that unlike the <group> element in **catalog-dita-company.xml**, the <group> element here does not have an xml:base attribute. All public identifier uri paths must be explicitly relative to the catalog in your plugin. Therefore, notice that this uri path contains **dtd/**, because **IxiaDitabase.dtd** lives in the dtd subfolder of the plugin. All uri paths in this file must be relative to the catalog file itself.

7. **Change the remaining publicId to match the public identifier of CompanyDitabase.dtd and the uri value to dtd/CompanyDitabase.xml:**

```
<public publicId="-//CUSTOMER//DTD DITA Composite//EN" uri="dtd/CompanyDitabase.dtd"/>
```

8. **Delete everything in the *dtd* subfolder of this plugin folder.**
9. **Copy CompanyDitabase.dtd into the *dtd* subfolder of the plugin.**

   Note: If you recall, when you created **CompanyDitabase.dtd**, you edited several sections to provide exact relative paths, such as `"../ixia/localizationDomain.ent"`. Those paths are no longer correct, given this location of **CompanyDitabase.dtd**, but it doesn't matter. The Open Toolkit is entirely catalog-based, meaning that it only ever considers public identifiers. It will never consider those relative paths and so there is no harm in leaving them as-is.

10. **Open the plugin.xml file.**
11. **Change the plugin's id as shown:**

```
<plugin id="com.cust.dtd.ot.plugin.dtd">
```

    The first part of the id (bolded above) should match the name of the plugin folder.

12. **Edit the feature extension to point to the name of the catalog you're providing; in this case, catalog.xml.**

```
<feature extension="dita.specialization.catalog.relative" value="catalog.xml" type="file"/>
```

Feature extensions are mechanisms that allow you to extend the DITA Open Toolkit in various ways. This particular feature extension indicates that you are extending the base DITA Open Toolkit catalog with a custom catalog.

13. **In %OT_Dir%, run the integrator.**
14. **In the root of the %OT_Dir% folder, open catalog-dita.xml and verify that your custom catalog is now included.**

    The reference should be near the beginning of the file. If the reference is present, then the DITA Open Toolkit and the Output Generator now know about your custom DTD.

You have successfully integrated your custom DTD into the Output Generator and you can create content using topics (and maps) whose DOCTYPES reference your shell DTDs.

If you are using WebAuthor, you now need to integrate your custom DTDs into that application.

Customizing your DTD for your content model

Select the task model

By default, the DITA CMS uses the strict task model (rather than general task model). If you prefer to use the general task model, which offers a more flexible structure and some additional elements, you need to make that specification within your shell DTD.

> **Caution:** If you switch to the general task model, it will be very difficult to switch back to the strict task model at a later date. Because the general task model allows structures which the strict model does not allow, there is a risk that some tasks might become invalid when you switch back to the strict model.

1. **Check out and open CompanyDitabase.dtd.**
2. **Find the Content Constraint Integration section.**

   In this section is an entity named strictTaskbody-c.def.:

```
<!ENTITY % strictTaskbody-c-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Strict Taskbody Constraint//EN"
  "../technicalContent/dtd/strictTaskbodyConstraint.mod">
%strictTaskbody-c-def;
```

   This is the entity that defines that constraint that specifies the strict task model is used.

3. **To use the general task model, comment out the `strictTaskbody-c.def` entity by surrounding it with `<!--` and `-->`, as shown:**

```
<!--
<!ENTITY % strictTaskbody-c-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Strict Taskbody Constraint//EN"
  "../technicalContent/dtd/strictTaskbodyConstraint.mod">
%strictTaskbody-c-def;
-->
```

4. **Scroll back to the Domains Attribute Override section and find the following section:**

```
<!ENTITY included-domains
                          "&concept-att;
                           &glossentry-att;
                           &glossgroup-att;
                           &reference-att;
                           &task-att;
                           &hi-d-att;
                           &ut-d-att;
                           &indexing-d-att;
                           &hazard-d-att;
                           &abbrev-d-att;
                           &pr-d-att;
                           &sw-d-att;
```

```
                          &ui-d-att;
                          &taskbody-constraints;
                          &localization-loc-d-att;
  "
>
```

5. **Delete `&taskbody-constraints;`.**

   Because you commented out the strictTaskbody constraint, the `&taskbody-constraints;` entity is no longer defined and must not be referenced.

6. **Save, close, and check in CustomerDitabase.dtd.**

Select the topic nesting model

While the DITA model does not encourage nested topics, it does support them by editing your document shells to allow, say, a concept to contain another concept. By default, the DITA CMS is set up to allow nested topics. However, if your content model does not include nested topics, you can configure the CMS to disallow them.

1. **Check out and open CustomerDitabase.dtd.**
2. **Find the Topic Nesting Override section.**

```
<!-- ============================================================= -->
<!--                    TOPIC NESTING OVERRIDE                     -->
<!-- ============================================================= -->

<!--                    Redefine the infotype entity to exclude
                        other topic types and disallow nesting     -->
<!ENTITY % glossentry-info-types
                        "no-topic-nesting"                          >
<!ENTITY % info-types   "topic | concept | task | reference |
                         glossentry | glossgroup"                   >
```

   In this section, there are two nestings defined: glossentry-info-types and info-types. For info-types, the default nesting is either one of topic, concept, task, reference, glossentry or glossgroup. This nesting applies to all topic types (including specializations such as referable-content). In this section, there is also a specific nesting defined (glossentry-info-types) that specifies that a glossentry cannot have nested topics. (no-topic-nesting is a special element defined for this purpose.) The type-specific nesting (in this case, glossentry-info-types) always overrides the info-types definition. You can define a type-specific nesting for any topic type(s).

3. **To disallow all topic nesting, continue to step 4. To disallow specific nesting, continue to step 5.**

4. **Disallow all topic nesting: edit the info-types entity as shown:**

```
<!ENTITY % info-types
                        "no-topic-nesting"                          >
```

Continue to step 6.

5. **Disallow specific topic nesting: add additional exception entities as appropriate.**

In this example, we will disallow task nesting but leave it enabled for all other topic types (except glossentry).

```
<!-- ============================================================= -->
<!--                    TOPIC NESTING OVERRIDE                     -->
<!-- ============================================================= -->

<!--                    Redefine the infotype entity to exclude
                        other topic types and disallow nesting     -->
<!ENTITY % glossentry-info-types
                        "no-topic-nesting"                           >
<!ENTITY % task-info-types
                        "no-topic-nesting"                           >
<!ENTITY % info-types   "topic | concept | task | reference |
                         glossentry | glossgroup"                    >
```

Or, we can turn the scenario around and disallow all topic nesting but enable it for just one topic type, such as task. In this example, we disallow topic nesting for the general info-types entity but we then allow it for the specific task-info-types entity. (We can delete the glossentry-info-types entity altogether because it is redundant in this scenario.) Here, the task-info-types entity allows nesting a task within a task but no other nesting is allowed:

```
<!-- ============================================================= -->
<!--                    TOPIC NESTING OVERRIDE                     -->
<!-- ============================================================= -->

<!--                    Redefine the infotype entity to exclude
                        other topic types and disallow nesting     -->
<!ENTITY % task-info-types
                        "task            "                           >
<!ENTITY % info-types   "
                         no-topic-nesting"                           >
```

6. **Save, close, and check in CustomerDitabase.dtd.**

Select the domains to use

DITA has a number of “standard” domains, such as the programming domain, the user interface domain, the software domain, and so on. Each of these domains includes a set of elements that are usually used inline--that is, they mark up specific words or phrases within a block of text. (There are some domain elements that are block elements in their own right.) You might not need

all of the standard domains. For example, if you are not documenting a computer application, you might not need the user interface or software domains. If you want to prevent authors from using formatting markup with no semantic significance, such as bold (<b>) or italics (<i>), then you don't want to use the highlighting domain.

For these steps, we'll remove the highlighting domain.

1. **Check out and open CustomerDitabase.dtd.**
2. **Find the Domain Entity Declarations section.**

   This section includes entities that correspond to each of the standard DITA domains.

3. **Comment out each entity that corresponds to a domain you do not want to be available to authors.**

   For example, to prevent authors from using the highlighting domain elements, comment out this section:

```
<!ENTITY % hi-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 1.2 Highlight Domain//EN"
         "../base/dtd/highlightDomain.ent"
>%hi-d-dec;
```

4. **Find the Domains Extension section.**
5. **Remove all references to the domain you are not using.**

   For this example, remove the one reference to the highlighting domain (hi-d-ph):

```
<!ENTITY % ph           "ph |
                         %hi-d-ph; |
                         %pr-d-ph; |
                         %sw-d-ph; |
                         %ui-d-ph;
                        ">
```

6. **Find the Domains Attribute Override section.**
7. **Delete *&hi-d-att;* from the list.**
8. **Find the Domain Element Integration section.**
9. **Comment out the following section:**

```
<!ENTITY % hi-d-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Highlight Domain//EN"
         "../base/dtd/highlightDomain.mod"
>%hi-d-def;
```

10. **Save, close and check in CustomerDitabase.dtd.**

Apply additional constraints

Constraints allow you to limit the elements or attributes that can be used in certain contexts. For example, you might want to prevent the use of refsyn and related elements in reference topics. If you don't need to use those elements, you can constrain to simplify the authoring experience for authors. Constraints enable you to use a subset of the structure allowed by the DITA standard in a given context; you cannot use constraints to enable use of structure that the DITA standard itself does not allow.

> **Caution:** Be very careful with this change. If you activate the constraints in CompanyDitabase.dtd after you already have content that references it, that content might becomes invalid if it uses elements that are no longer allowed due to the new constraints. You must either edit the content so that it is no longer invalid or you must edit it so that it no longer references CompanyDitabase.dtd. The former is obviously the recommended option. To avoid creating this situation, IXIASOFT strongly recommends making these decisions before you accumulate a large amount of content.

1. **Check out and open CustomerDitabase.dtd.**
2. **Find the IXIASOFT Additional Constraints section:**

```
<!-- ================== IXIASOFT Additional Constraints ========== -->
<!--                    Strict Paragraph prevents the use of dl, fig,
                        lines, lq, note, object, ol, pre, simpletable,
                        sl, table and ul inside a <p>              -->
<!--
<!ENTITY % strictPara-c-def
  PUBLIC "-//IXIA//ELEMENTS IXIA DITA 1.2 Strict Paragraph Constraint//EN"
  "strictParaConstraint.mod">
%strictPara-c-def;
-->
<!--                    Strict Section prevents the insertion of text
                        directly inside a section                  -->
<!--
<!ENTITY % strictSection-c-def
  PUBLIC "-//IXIA//ELEMENTS IXIA DITA 1.2 Strict Section Constraint//EN"
  "strictSectionConstraint.mod">
%strictSection-c-def;
-->
```

This section contains two entities, `strictPara-c-def` and `strictSection-c-def`. The first entity, as you can read in the comment, prohibits the use of certain elements within a <p> element that would normally be allowed. The second entity does the same for <section>. Both of these entities are commented out by default.

3. **If you want to activate either or both of these constraints, comment the appropriate entity back in.**

As you can see, these entities reference two files, **strictParaConstraint.mod**, and **strictSectionConstraint.mod**. These two files also live in *REPOSITORY/system/dtd/ixia*. Now that you are referencing them from **CustomerDitabase.dtd**, which does not live in the same folder as these two .mod file, you must change the system identifier paths to correctly point to the files:

4. **Add the additional path levels as shown:**

```
<!-- ================= IXIASOFT Additional Constraints ========== -->
<!--                   Strict Paragraph prevents the use of dl, fig,
                       lines, lq, note, object, ol, pre, simpletable,
                       sl, table and ul inside a <p>              -->
<!ENTITY % strictPara-c-def
  PUBLIC "-//IXIA//ELEMENTS IXIA DITA 1.2 Strict Paragraph Constraint//EN"
  "../ixia/strictParaConstraint.mod">
%strictPara-c-def;
<!--                   Strict Section prevents the insertion of text
                       directly inside a section                  -->
<!ENTITY % strictSection-c-def
  PUBLIC "-//IXIA//ELEMENTS IXIA DITA 1.2 Strict Section Constraint//EN"
  "../ixia/strictSectionConstraint.mod">
%strictSection-c-def;
```

5. **Scroll back in the file to find the following section, just above the Content Constraint Integration section:**

```
<!--                   Add the following domains if paragraph and
                       section constraints are enabled
                          &para-constraints;
                          &section-constraints;
-->
```

6. **Comment back in the constraint(s) that you enabled.**
7. **Save, close, and check in CustomerDitabase.dtd.**

If you find that you need to edit either of these constraints, IXIASOFT recommends that you create your own modules rather than edit **strictParaConstraint.mod** or **strictSectionConstraint.mod**. You can copy those two files to *REPOSITORY/system/dtd/client*, rename them, and edit them as needed. Then, in **CustomerDitabase.dtd**, use those modules as the definitions of the strictPara-c-def and strictSection-c-def entities. You can also follow this model to create additional constraints of your own.

Integrating specializations

Integrating a topic specialization

Integrate the FAQ specialization into the DITA CMS

Before integrating the FAQ (Frequently Asked Questions) specialization—or any other specialization—you should create a custom shell DTD and catalog. While you can integrate using **IxiaDitabase.xml**, it's a much better practice to use your own shell. These instructions assume you are using your own shell DTD and catalog.

These steps describe how to integrate the FAQ specialization. To integrate any other DTDs, you will follow a similar process, though each specialization is slightly different. These steps use the same files names as in the previous instructions (**CompanyDitabase.dtd** and **catalog-dita-cust.xml**).

1. **If you do not already have a copy of the FAQ specialization, download it from the DITA-OT plugins repository on GitHub.**
2. **Add the *faq* folder to *Repository/system/dtd*.**
3. **In the *faq* folder, open catalog.xml and copy the following two lines:**

```
<public publicId="-//IBM//DTD DITA FAQ//EN" uri="faq_shell.dtd"/>
<public publicId="-//IBM//ELEMENTS DITA FAQ//EN" uri="faq.mod"/>
```

4. **Check out and open *Repository/system/catalogs/*catalog-dita-company.xml.**
5. **After the existing <group> add a second <group>, as shown:**

```
<group xml:base="../dtd/client/">
   <public publicId="-//CUSTOMER//DTD DITA Composite//EN" uri="CompanyDitabase.dtd"/>
</group>

<group xml:base="../dtd/faq/">
</group>
```

   Make sure the xml:base is set as shown. This setting specifies that the starting point of the relative path in the uri of the system identifiers in this group will start in the *dtd/faq* subfolder. Starting here keeps the uri paths as simple as possible.

6. **Within the new <group>, paste the two lines you copied from catalog-dita-company.xml, as shown:**

```
<group xml:base="../dtd/faq/">
  <!--Public identifiers for FAQ specialization-->
```

```
  <public publicId="-//IBM//DTD DITA FAQ//EN" uri="faq_shell.dtd"/>
  <public publicId="-//IBM//ELEMENTS DITA FAQ//EN" uri="faq.mod"/>
</group>
```

These lines add the public and system identifiers for the faq specialization DTD and .mod files to your catalog.

7. **Save, close and check in catalog-dita.company.xml.**
8. **Check out and open CustDitabase.dtd.**
9. **Add the following within the Topic Element Integration section:**

```
<!ENTITY % faq-typemod
                        PUBLIC
"-//IBM//ELEMENTS DITA FAQ//EN"
"../faq/faq.mod"                                                    >
%faq-typemod;
```

The FAQ specialization is quite simple and has only a .mod file; it doesn't have any .ent files. If your specialization has .ent files, you'll need to add them to the appropriate places in **CompanyDitabase.dtd**.

10. **Check out and open** *Repository/system/conf/***equivalence.xml.**
11. **In the <equivalence type="topic"> section, add the following line:**

```
<object type="faq" icon="/system/conf/icons/reference.png"/>
```

As shown, this line specifies that the CMS should use the same icon for faq topics that it does for reference topics. If you want to specify a unique icon for faq topics, you can add it in the location shown and edit the line accordingly.

12. **Save, close and check in equivalence.xml.**
13. **Using an editor such as Notepad++, create a template for faq topics.**

As a starter, you can copy and paste the following into the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faq PUBLIC "-//CUSTOMER//DTD DITA Composite//EN"
"../../system/dtd/client/CompanyDitabase.dtd">
<faq id="faq">
   <title></title>
   <faqbody>
      <faqgroup>
         <title></title>
         <faqlist>
            <faqitem>
               <faqquest></faqquest>
               <faqans></faqans>
            </faqitem>
         </faqlist>
      </faqgroup>
```

```
   </faqbody>
</faq>
```

Be sure to use the correct DOCTYPE here, especially if you already have templates for your specialized topic type. This example uses the “CUSTOMER” DOCTYPE because that is the DOCTYPE defined by the specialization.

14. **Name the template faq.xml and add it to** *Repository/system/templates/topics***.**

    If you have subfolders within the *topics* folder, add it to the appropriate subfolder.

Related Links

Integrate the FAQ specialization into the Output Generator on page 21
Create your own shell DTD on page 4
Create a custom catalog on page 6

Integrate the FAQ specialization into the Output Generator

After you integrate a specialization into the DITA CMS, you must also integrate it into the Output Generator so that the DITA Open Toolkit instance within the Output Generator can process topics that use the specialized elements or attributes.

1. **Copy the** *faq* **folder into the** *data/%OT_Dir%/plugins* **folder.**

   Depending on your Output Generator configuration, this path might vary.

2. **Copy** *Repository/system/dtd/client/***CompanyDitabase.dtd to** *data/%OT_Dir%/plugins/com.cust.dtd* **and overwrite the existing file.**

   This step ensures that your two **CompanyDitabase.dtd** files, the one in the DITA CMS and the one in the Output Generator, are identical.

   Because this specialization has its own catalog file and is itself a plugin, you don't need to edit your plugin's catalog file. The FAQ plugin is self-integrating.

3. **Run the integrator.**

The FAQ specialization is now integrated into the Output Generator and you can succesfully generate output that includes FAQ topics.

Related Links

Integrate the FAQ specialization into the DITA CMS on page 19
Create a DTD plugin on page 10

Integrating a domain specialization

Integrate the XML Mention specialization into the DITA CMS

DITA lacks elements for tagging XML components such as element, attribute, attribute value, and entity. In addition, it lacks an element for tagging file names. Inspired by Eliot Kimber's tutorials at **http://xiruss.org/tutorials/dita-specialization/**, let's assume you have created a domain specialization, XML Mention, that includes these elements. This specialization includes the two files **xmlDomain.mod** and **xmlDomain.ent**. (Examples of both of these files are included here.) The specialization does not include a DTD file because it is a domain specialization and a domain specialization never includes a DTD because it cannot be used on its own, contrary to a topic type specialization.

1. **Add xmlDomain.mod and xmlDomain.ent to** *Repository/system/dtd/client*.
2. **Check out and open CompanyDitabase.dtd.**
3. **In the Domain Entity Declarations section, add the following:**

```
<!ENTITY % xml-d-dec
  PUBLIC "-//IXIA//ENTITIES XML Mention Domain//EN"
         "xmlDomain.ent"
>
%xml-d-dec;
```

4. **In the Domain Extensions section, edit the keyword entity to include xml-d-keyword:**

```
<!ENTITY % keyword      "keyword |
                         %pr-d-keyword; |
                         %sw-d-keyword; |
                         %ui-d-keyword; |
                         %xml-d-keyword;
                        ">
```

5. **In the Domains Attribute Override section, edit the included-domains entity to reference xml-d-att:**

```
<!ENTITY included-domains
                          "&concept-att;
                           &glossentry-att;
                           &glossgroup-att;
                           &reference-att;
                           &task-att;
                           &hi-d-att;
                           &ut-d-att;
                           &indexing-d-att;
                           &hazard-d-att;
                           &abbrev-d-att;
                           &pr-d-att;
                           &sw-d-att;
                           &ui-d-att;
```

```
                              &taskbody-constraints;
                              &localization-loc-d-att;
                              &color-d-att;
                              &xml-d-att;
  "
>
```

6. **In the Domain Element Integration section, add the following:**

```
<!ENTITY % xml-d-def
  PUBLIC "-//IXIA//ELEMENTS XML Mention Domain//EN"
         "xmlDomain.mod"
>
%xml-d-def;
```

7. **Save, close, and check in CompanyDitabase.dtd.**
8. **Check out and open catalog-dita-company.xml.**
9. **Within the group whose base is /dtd/client/, add the public identifiers for the xmlDomain.mod and xmlDomain.ent files.**

```
<group xml:base="../dtd/client/">

   <public publicId="-//CUSTOMER//DTD DITA Composite//EN" uri="CompanyDitabase.dtd"/>

   <!--Public identifiers for XML Mention specialization-->
   <public publicId="-//IXIA//ELEMENTS XML Mention Domain//EN" uri="xmlDomain.mod"/>
   <public publicId="-//IXIA//ENTITIES XML Mention Domain//EN" uri="xmlDomain.ent"/>

</group>
```

10. **Save, close, and check in catalog-dita-company.xml**

Related Links

Integrate the XML Mention specialization into the Output Generator on page 23
Create your own shell DTD on page 4
Create a custom catalog on page 6
Example xmlDomain.ent file on page 33
Example xmlDomain.mod file on page 31

Integrate the XML Mention specialization into the Output Generator

After you integrate a specialization into the DITA CMS, you must also integrate it into the Output Generator so that the DITA Open Toolkit instance within the Output Generator can process topics that use the specialized elements or attributes.

1. **Copy xmlDomain.ent and xmlDomain.mod into the** *data/%OT_Dir%/plugins* **folder.**

   Depending on your Output Generator configuration, this path might vary.

2. **Copy** *Repository/system/dtd/client/***CompanyDitabase.dtd to** *data/%OT_Dir%/plugins/com.cust.dtd* **and overwrite the existing file.**

   This step ensures that your two **CompanyDitabase.dtd** files, the one in the DITA CMS and the one in the Output Generator, are identical.

3. **Open catalog.xml in** *data/%OT_Dir%/plugins/com.cust.dtd***.**
4. **Add the following three lines:**

```
<!--Public identifiers for XML Mention specialization-->
<public publicId="-//IXIA//ELEMENTS XML Mention Domain//EN" uri="dtd/xmlDomain.mod"/>
<public publicId="-//IXIA//ENTITIES XML Mention Domain//EN" uri="dtd/xmlDomain.ent"/>
```

5. **Save and close catalog.xml.**
6. **Run the integrator.**

The XML Mention specialization is now integrated into the Output Generator and you can succesfully generate output that includes the new elements.

Related Links

Integrate the XML Mention specialization into the DITA CMS on page 22
Create a DTD plugin on page 10

Integrating an attribute specialization

Integrate an attribute specialization into the DITA CMS

Before integrating an attribute specialization—or any other specialization—you should create a custom shell DTD and catalog. While you can integrate using **IxiaDitabase.xml**, it's a much better practice to use your own shell. These instructions assume you are using your own shell DTD.

You might have specialized attributes that you need to integrate into the DITA CMS. For example, you might need to color some of your code samples and therefore you have specialized a *color* attribute from the existing *base* attribute. In these steps, we assume you have created the specialization in a file named companyAttDomain.ent. These steps use the same CompanyDitabase.dtd file name as in the previous instructions.

1. **Copy companyAttDomain.ent into** *REPOSITORY/system/dtd/client***.**
2. **Check out and open CompanyDitabase.dtd.**
3. **Locate the Domain Attribute Declarations section.**

4. **Add an entity definition similar to the following to that section:**

```
<!ENTITY % company-d-dec
                        PUBLIC
"-//COMPANY//ENTITIES DITA Company Attribute Domain//EN"
"companyAttDomain.ent"                                          >
%company-d-dec;
```

   Where the public identifier matches the one you assigned in **companyAttDomain.ent** and the system identifier matches the file name. Note that because **companyAttDomain.ent** lives in the same folder as **CompanyDitabase.dtd**, the system identifier path is flat.

5. **Locate the Domain Attribute Extensions section.**
6. **Edit the *props-attribute-extensions* entity to include the entity (*color-d-attribute*) that defines your new attribute in companyAttDomain.ent, similar to the following example:**

```
<!ENTITY % base-attribute-extensions
          "%color-d-attribute;"                                 >
```

7. **Locate the Domains Attribute Override section.**
8. **Add the *color-d-att* entity to the list of included domains.**

```
<!ENTITY included-domains
                          "&concept-att;
                           &glossentry-att;
                           &glossgroup-att;
                           &reference-att;
                           &task-att;
                           &hi-d-att;
                           &ut-d-att;
                           &indexing-d-att;
                           &hazard-d-att;
                           &abbrev-d-att;
                           &pr-d-att;
                           &sw-d-att;
                           &ui-d-att;
                           &taskbody-constraints;
                           &localization-loc-d-att;
                           &color-d-att;
  "
>
```

9. **Save, close, and check in CustomerDitabase.dtd.**

After integrating the attribute specialization into the DITA CMS, the next step is to integrate it into the Output Generator, to ensure you can successfully generate output from content that includes the specialized attributes.

Related Links

Example of custom attribute definition on page 34
Integrate an attribute specialization into the Output Generator on page 26
Create your own shell DTD on page 4

Integrate an attribute specialization into the Output Generator

These steps assume you have created a custom plugin which you are using to integrate your shell DTD into the DITA Open Toolkit within the Output Generator. You use that same plugin to integrate your attribute specialization.

1. **Copy** *Repository/system/dtd/client/***CompanyDitabase.dtd to** *data/%OT_Dir%/plugins/com.cust.dtd* **and overwrite the existing file.**
2. **Copy companyAttDomain.ent to the same location.**
3. **Open catalog.xml in** *data/%OT_Dir%/plugins/com.cust.dtd***.**
4. **Add the following two lines:**

```
<!--Public identifiers for attribute specialization-->
<public publicId="-//COMPANY//ENTITIES DITA Company Attribute Domain//EN"
uri="dtd/companyAttDomain.ent"/>
```

5. **Save and close catalog.xml.**
6. **Run the integrator.**

The specialized attribute is now integrated into the Output Generator and you can succesfully generate output that includes topics that use it.

Related Links

Integrate an attribute specialization into the DITA CMS on page 24
Create a DTD plugin on page 10

Adding your specializations to the Web Author

By default, the Web Author supports the DITA 1.2 standard as well as IXIASOFT specializations. If you have added your own specializations to the DITA CMS, you also need to add them to the Web Author.

The DTDs required by the Web Author are packaged in a file called *frameworks.zip*, which is included in the *webauthor/webauthor.war* file provided with the installation package. To add your specializations to the Web Author, two steps are required:

1. Create a custom *frameworks.zip* file.
2. Add the custom *frameworks.zip* file to your deployment.

These procedures assume that you are familiar with creating DTDs and catalog files for DITA.

> **Note:** This feature requires Web Author version 1.0.9.138 and up. Please upgrade to the latest version if necessary.

Create a custom frameworks.zip file

By default, the Web Author supports the DITA 1.2 standard as well as IXIASOFT specializations. If you added your own specializations to the DITA CMS, you also need to add them to the Web Author.

To add your specializations to the Web Author, you:

- Unzip the *frameworks.zip* file
- Add your DTDs
- Add your catalogs
- Point to your catalogs in the main *catalog.xml* file
- Rezip the updated *frameworks.zip* file

This procedure is described below.

DTDs and catalogs location

The relative path between the catalogs directory and the DTDs directory should be the same in the DITA CMS Eclipse Client and in the custom frameworks.zip file. While this is not mandatory, this step will make it easier for troubleshooting issues.

The following table lists the recommended locations for your custom DTDs and catalogs:

Document	CMS location	Web Author location

Customer catalog	`Repository/system/catalogs`	`Repository/system/cms.webauthor/frameworks/dita/DITA-OT`
		Your catalogs must be referenced by an entry in the following catalog:
		`Repository/system/cms.webauthor/frameworks/dita/catalog.xml`

Customer DTDs	`Repository/system/dtd/client`	`Repository/system/cms.webauthor/frameworks/dita/DITA-OT/dtd/client`

The procedure below assumes that you have used the recommended locations.

To create a custom *frameworks.zip* file:

1. **Go to the following directory:**

   ```
   %tomcat%/webapps/webauthor/resources
   ```

2. **Extract the** *frameworks.zip* **file to a temporary directory (referred to as** *%CustomFrameworks%* **in this procedure).**

   The *%CustomFrameworks%* directory now contains two subdirectories, `samples` and `dita`.

3. **Add your DTD files to the** *%CustomFrameworks%/dita/DITA-OT/dtd/client* **directory.**
4. **Add your catalog file(s) to the** *%CustomFrameworks%/dita/DITA-OT/* **directory to map your public IDs to the DTD files.**

   Note: Your catalog files *must* specify public IDs for your map. The CMS Application Server requires public IDs to locate the DTD files. It cannot used system IDs.

5. **Open the** *%CustomFrameworks%/dita/catalog.xml* **file.**
6. **Add a new `<nextCatalog>` element to point to the catalog file(s) you added to the** *%CustomFrameworks%/dita/DITA-OT/* **directory.**
7. **Save and close the file.**
8. **Zip the contents of the** *%CustomFrameworks%* **directory to create a new** *frameworks.zip* **file.**

Add the custom frameworks.zip file to your deployment

To add the custom *frameworks.zip* file to your deployment:

1. **Open the TEXTML Administration view.**
2. **Connect to your server and docbase.**
3. **Navigate to */system/cms.webauthor*.**
4. **Right-click *cms.webauthor* and select Create Collection.**
5. **In the Add Collection dialog, enter `frameworks`.**

   You can use another name for the collection, but IXIASOFT recommends that you use `frameworks`.

6. **Right-click the `frameworks` collection and select Insert Documents.**

   The **Import Dialog** window appears.

7. **Click Add File and browse to the location of your custom *frameworks.zip* file.**
   a) **Select the file.**
   b) **Click Open.**

   The file path and name appear under **Import As**.

8. **In the Set options area, click Add and Replace.**
9. **Click OK.**

   The next step is to configure the Web Author to provide the location of the custom *frameworks.zip* file.

10. **Open the following file in an XML editor:**

```
%TomcatDir%/conf/webauthor/webconfig.xml
```

    For example:

```
C:\Program Files\Apache Software Foundation\Tomcat
6.0\conf\webauthor\webconfig.xml (Windows)
/opt/tomcat6/conf/webauthor/webconfig.xml (Linux)
```

    Note: The actual path may vary according to your deployment.

11. **Locate the following line:**

```
<entry key="cms.webauthor.frameworks.path"></entry>
```

12. **Set the `cms.webauthor.frameworks.path` key to the location in the docbase where you stored the *frameworks.zip* file.**

For example:

```
<entry key="cms.webauthor.frameworks.path">
       /system/cms.webauthor/frameworks/frameworks.zip
</entry>
```

13. **Save the file and restart the Web Author.**

Appendix A: Sample files

Example xmlDomain.mod file

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ============================================================= -->
<!--                    HEADER                                     -->
<!-- ============================================================= -->
<!--  MODULE:    XML Mention Domain                                 -->
<!--  VERSION:   1.0                                               -->
<!--  DATE:      May 2014                                          -->
<!--                                                               -->
<!-- ============================================================= -->

<!-- ============================================================= -->
<!--                    PUBLIC DOCUMENT TYPE DEFINITION            -->
<!--                    TYPICAL INVOCATION                         -->
<!--                                                               -->
<!--  Refer to this file by the following public identifier or an
      appropriate system identifier
PUBLIC "-//IXIA//ELEMENTS XML Mention Domain//EN"
      Delivered as file "xmlDomain.mod"                            -->

<!-- ============================================================= -->
<!--                   ELEMENT NAME ENTITIES                       -->
<!-- ============================================================= -->

<!ENTITY % xmlelem "xmlelem" >
<!ENTITY % xmlatt  "xmlatt" >
<!ENTITY % xmlfile  "xmlfile" >
<!ENTITY % xmlenty  "xmlenty" >
<!ENTITY % xmlattval  "xmlattval" >

<!-- ============================================================= -->
<!--                    ELEMENT DECLARATIONS                       -->
<!-- ============================================================= -->

<!--                    LONG NAME: XML Element                     -->
<!ENTITY % xmlelem.content
  "
  (#PCDATA)*
  "
>
<!ENTITY % xmlelem.attributes
  '
  %univ-atts;
  keyref
    CDATA
    #IMPLIED
  outputclass
    CDATA
    #IMPLIED
  '
>
<!ELEMENT xmlelem %xmlelem.content; >
<!ATTLIST xmlelem %xmlelem.attributes; >

<!--                    LONG NAME: XML Attribute                   -->
<!ENTITY % xmlatt.content
  "
  (#PCDATA)*
  "
```

```
>
<!ENTITY % xmlatt.attributes
  '
  %univ-atts;
  keyref
    CDATA
    #IMPLIED
  outputclass
    CDATA
    #IMPLIED
  '
>
<!ELEMENT xmlatt %xmlatt.content; >
<!ATTLIST xmlatt %xmlatt.attributes; >

<!--                    LONG NAME: XML File                        -->
<!ENTITY % xmlfile.content
  "
  (#PCDATA)*
  "
>
<!ENTITY % xmlfile.attributes
  '
  %univ-atts;
  keyref
    CDATA
    #IMPLIED
  outputclass
    CDATA
    #IMPLIED
  '
>
<!ELEMENT xmlfile %xmlfile.content; >
<!ATTLIST xmlfile %xmlfile.attributes; >

<!--                    LONG NAME: XML Entity                      -->
<!ENTITY % xmlenty.content
  "
  (#PCDATA)*
  "
>
<!ENTITY % xmlenty.attributes
  '
  %univ-atts;
  keyref
    CDATA
    #IMPLIED
  outputclass
    CDATA
    #IMPLIED
  '
>
<!ELEMENT xmlenty %xmlenty.content; >
<!ATTLIST xmlenty %xmlenty.attributes; >

<!--                    LONG NAME: XML Attribute Value             -->
<!ENTITY % xmlattval.content
  "
  (#PCDATA)*
  "
>
<!ENTITY % xmlattval.attributes
  '
  %univ-atts;
  keyref
    CDATA
```

```
    #IMPLIED
  outputclass
    CDATA
    #IMPLIED
  '
>
<!ELEMENT xmlattval %xmlattval.content; >
<!ATTLIST xmlattval %xmlattval.attributes; >

<!-- ============================================================= -->
<!--                    SPECIALIZATION ATTRIBUTE DECLARATIONS      -->
<!-- ============================================================= -->

<!ATTLIST xmlelem     %global-atts;
                      class CDATA "+ topic/keyword xml-d/xmlelem "   >
<!ATTLIST xmlatt      %global-atts;
                      class CDATA "+ topic/keyword xml-d/xmlatt "    >

<!-- ====================== End XML Domain ======================== -->
```

Example xmlDomain.ent file

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ============================================================= -->
<!--                    HEADER                                     -->
<!-- ============================================================= -->
<!--  MODULE:    XML Mention Domain                                 -->
<!--  VERSION:   1.0                                               -->
<!--  DATE:      May 2014                                          -->
<!--                                                               -->
<!-- ============================================================= -->

<!-- ============================================================= -->
<!--                    PUBLIC DOCUMENT TYPE DEFINITION            -->
<!--                    TYPICAL INVOCATION                         -->
<!--                                                               -->
<!--  Refer to this file by the following public identifier or an
      appropriate system identifier
PUBLIC "-//IXIA//ENTITIES XML Mention Domain//EN"
      Delivered as file "xmlDomain.ent"                            -->

<!ENTITY % xml-d-keyword
  "xmlelem |
   xmlatt |
   xmlfile |
   xmlenty |
   xmlattval
  "
>

<!ENTITY   xml-d-att     "(topic xml-d)"
>

<!-- ================== End XML Mention Domain Entities ========== -->
```

Example of custom attribute definition

The following is an example of a custom attribute specialized from the existing *base* attribute.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- ============================================================= -->
<!--                    HEADER                                     -->
<!-- ============================================================= -->
<!--  MODULE:    Company Attributes Domain                         -->
<!--  VERSION:   1.0                                               -->
<!--  DATE:      May 2014                                          -->
<!--                                                               -->
<!-- ============================================================= -->

<!-- ============================================================= -->
<!--                    PUBLIC DOCUMENT TYPE DEFINITION            -->
<!--                    TYPICAL INVOCATION                         -->
<!--                                                               -->
<!--  Refer to this file by the following public identifier or an
      appropriate system identifier
PUBLIC "-//COMPANY//ENTITIES DITA Company Attribute Domain//EN"
      Delivered as file "companyAttDomain.ent"                     -->
<!-- ============================================================= -->

<!-- ============================================================= -->
<!--                    ELEMENT EXTENSION ENTITY DECLARATIONS      -->
<!-- ============================================================= -->

<!ENTITY % color-d-attribute
"color
(blue | brown | cyan | dkgray | gold | green | lime | ltgray | mdgray
| orange | purple | red | yellow)
#IMPLIED
"
>

<!-- ============================================================= -->
<!--                    DOMAIN ENTITY DECLARATION                  -->
<!-- ============================================================= -->

<!ENTITY color-d-att
"a(base color)"
>
```

Appendix B: DTD basics

What is a DTD?

DTD stands for Document Type Declaration. A DTD defines the elements, attributes, and attribute values that an XML document can use. It also defines the order in which those elements can occur and their relationships to each other. Here is an example of a simple DTD:

```
1  <!DOCTYPE addressbook [
2  <!ELEMENT addressbook (entry+) >
3  <!ELEMENT entry   (lname, fname, address+, (phone* | email*)*, bdate?) >
4  <!ELEMENT fname   (#PCDATA) >
5  <!ELEMENT lname   (#PCDATA) >
6  <!ELEMENT address (street+, city, state, zip) >
7  <!ATTLIST address type  (home|work|mailing|other)  #REQUIRED >
8  <!ELEMENT phone   (#PCDATA) >
9  <!ATTLIST phone   type  (home|cell|work|fax|other)  #REQUIRED >
10  <!ELEMENT email   (#PCDATA) >
11  <!ATTLIST email   type  (home|work)  #REQUIRED >
12  <!ELEMENT bdate   (#PCDATA) >
]>
```

> **Note:** The numbers on the left are not part of the DTD. They are just for reference in the following explanation.

1. This line defines the document type as addressbook. Therefore, <addressbook> will be the highest-level element in the XML document.
2. This line specifies that the <addressbook> element can contain one or more <entry> elements. The + specifies that <entry> is required and repeatable.
3. This line defines the <entry> element and specifies that each <entry> element must contain only one <lname> element followed by only one <fname> element followed by one or more <address> elements, followed optionally by one or more <phone> elements, and/or one or more optional <email> elements, followed optionally by only one <bdate> element—always in that order. The * specifies that <phone> and <email> are optional and repeatable. You can have none or any number of these elements. The ? indicates that <bdate> is optional but non-repeatable. A comma between two elements indicates they must occur in that order, while a pipe (|) between two elements indicates they can occur in any order.
4. This line defines the <fname> element and specifies that it contains text. #PCDATA stands for Parsed Character Data. The simplest explanation of what that means is that #PCDATA specifies that an element contains only text; it has no child elements.
5. Same as line 4.
6. This line defines the <address> element and specifies that it must contain one or more <street> elements, followed by one <city> element, followed by one <state> element, followed by one <zip> element, always in that order.

7. This line specifies that the <address> element has a required attribute, *type*. The allowed values for type are *home*, *work*, *mailing*, or *other*.
8. This line defines the <phone> element and specifies that it contains text.
9. This line specifies that the <phone> element has a required attribute, *type*. The allowed values for type are *home*, *cell*, *work*, *fax*, or *other*.
10. This line defines the <email> element and specifies that it contains text.
11. This line specifies that the <email> element has a required attribute, *type*. The allowed values for type are *home* and *work*.
12. This line defines the <bdate> element and specifies that it contains text.

Here is an example of a valid XML document based on this DTD:

```
<addressbook>
   <entry>
      <lname>Element</lname>
      <fname>Edna</fname>
      <address type = "home">
         <street>100 Elm St.</street>
         <street>#453</street>
         <city>Springfield</city>
         <state>GA</state>
         <zip>31000</zip>
      </address>
      <phone type = "home">222-555-1212</phone>
      <phone type = "cell">222-555-5555</phone>
      <email type = "home">ednaelement@email.com</email>
   </entry>
   <entry>
      <lname>Attribute</lname>
      <fname>Arthur</fname>
      <address type = "home">
         <street>200 Main St.</street>
         <city>Oakville</city>
         <state>WI</state>
         <zip>53000</zip>
      </address>
      <address type = "work">
         <street>1000 Commerce St.</street>
         <city>Oakville</city>
         <state>WI</state>
         <zip>53000</zip>
      </address>
      <email type = "work">aatribute@bigcorp.com</email>
   </entry>
</addressbook>
```

Look at the difference between the two <entry> elements. The first has one <address> element that contains two <street> elements. This is allowed by the DTD. It also has two <phone> elements, also allowed by the DRD.

On the other hand, the second <entry> element has two <address> elements, as allowed by the DTD—one for Jack's home and another for his work. However, Jack's entry does not have any

<phone> elements. Because these are defined as optional for the <entry> element, it's okay not to have them.

Here is an <entry> element that is not valid, based on the addressbook DTD. Can you spot the problems?

```
<entry>
   <fname>Doris</fname>
   <lname>Declaration</lname>
   <address type = "home">
      <street>750 Grant St.</street>
      <city>Anytown</city>
      <state>MA</state>
   </address>
   <email type="personal">justplaindoris@email.com</email>
   <phone>444-555-3434</phone>
   <bdate>03/27</bdate>
</entry>
```

There are some other things you can specify in a DTD, but these are the basics—enough to get you oriented.

XML documents can either include the DTD directly within them, or reference an external DTD. If you are familiar with HTML files and CSS (Cascading Style Sheets), this is the same idea as either including the CSS styling information directly inside each HTML file, or keeping it in a separate CSS file that each HTML file references.

> **Note:** An XML document does not have to use a DTD, either internal or referenced. However, it's strongly advised so that you can be sure your documents all adhere to the structure you have defined, to avoid any processing errors down the road. If you are authoring using any standard such as DITA, DocBook, S100D, you must reference that standard's DTDs.

DITA XML documents must always reference either the standard DITA DTDs or your own shell DTDs that you have specialized from the standards. They cannot include the DTD information internally.

Here is an example of a typical DOCTYPE reference to a DITA DTD—in this case, the DTD that defines the concept topic:

```
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
```

With DITA, the DTDs are already defined. When you create DITA content, you only have to reference them. All DITA-aware editors will automatically add the correct DOCTYPE and reference when you create a DITA topic. This line (or something similar, if you have specialized) will always be the second line in any DITA concept file.

You can see that this line specifies that this reference is PUBLIC. DTD references can be either SYSTEM, which is the default, or PUBLIC. What's the difference? Essentially, a SYSTEM DTD is on a private system—it's a DTD you've created yourself and that is not accessible to anyone outside of your organization.

Looking again at the addressbook example, say the DTD is one you created for you own personal addressbook, to validate the XML file you keep all your contact in. You've named this DTD “addressbook.dtd”. In your addressbook XML file, you'd add the following:

```
<!DOCTYPE addressbook SYSTEM "addressbook.dtd">
```

which specifies that your addressbook XML file should look to the addressbook.dtd file to define its structure. In this line, `"addressbook.dtd"` is a *system identifier*and it's always enclosed in double quotes. Note that the system identifier actually represents a relative path from the XML document to the DTD. In this example, the assumption is that addressbook.dtd lives in the same folder as the addressbook XML file. If that is not the case, you need to adjust the system identifier accordingly.

A PUBLIC DTD, as the name suggests, is one that is open to the public for general use. Because the DITA is an open standard, its DTDs are free for anyone to use and therefore the DOCTYPE references to them are always PUBLIC.

If the reference is PUBLIC, then the keyword is followed by a restricted form of "public identifier" called Formal Public Identifier (FPI) enclosed in double quote marks. In the previous DITA concept example, the FPI is `"-//OASIS//DTD DITA Concept//EN"`. Following the public identifier is the system identifier. In the DITA concept example, the system identifier is `"concept.dtd"`.

When you are using a PUBLIC DTD reference, you need to use a catalog to resolve the reference. Unlike the system identifier in a SYSTEM DTD reference, the FPI in a PUBLIC DTD reference does not point to an actual file. Instead, it's just a pointer. The catalog being used contains the FPI and maps it to a path. For example, the DITA Open Toolkit includes a catalog.xml file at its root. That catalog.xml file includes the following line:

```
<public publicId="-//OASIS//DTD DITA Concept//EN" uri="concept.dtd"
xml:base="dtd/technicalContent/dtd/"></public>
```

Notice that the publicID here is exactly the same as the FPI in the DOCTYPE declaration for concept, shown above. Now look at the *XML:base* value. It's a path to the actual DTD that's being referenced. In this case, if you follow that path—looking in the *dtd* subfolder of the DITA Open Toolkit folder, and then looking in the *technicalContent* subfolder of the *dtd* subfolder, and then looking in the *dtd* subfolder of the *technicalContent* subfolder, you find the file specified by the *uri* attribute; namely, the concept.dtd file.

Now the question is: how does a DITA file know where the catalog is? There's nothing in the DOCTYPE declaration of a DITA file that specifies a catalog location. If you are using a robust XML editor such as oXygen or XMetaL, those editors automatically point to the DITA catalog. Other, less robust XML editors can usually be set up manually to point to the DITA catalog. One way you can tell the connection is being made is that if you insert an element at a point where it is not allowed, you will usually get a validation error right away, meaning that the editor is constantly comparing the DITA file against its referenced DTD file by way of the catalog.

If you are not using an XML editor at all, but instead are simply using a text editor to edit DITA, there is no connection between the DITA file and the catalog. Your content is not validated against any DTD.

When you use the DITA Open Toolkit to build output, the Toolkit itself takes over and provides the connection between the DITA files and the DTDs using its own catalog.

The advantage of PUBLIC DTD references over SYSTEM ones is that there is no requirement that the DITA file and the DTD be in the same relative location to each other. They can be anywhere and it's up to the XML editor or build tool to know where the catalog file is that matches them up.

So, to sum up, here's a quick recap of what happens with a DITA file—say, a reference.

You open a DITA reference topic in your XML editor. The XML editor reads the DOCTYPE declaration and DTD reference at the top of the file. It will be similar or identical to this:

```
<!DOCTYPE reference PUBLIC "-//OASIS//DTD DITA Reference//EN" "reference.dtd">
```

The XML editor goes out to the catalog it has been set up to use and searches that catalog for a pointer that includes `"-//OASIS//DTD DITA Reference//EN"`. It finds this one:

```
<public publicId="-//OASIS//DTD DITA Reference//EN" uri="reference.dtd"
xml:base="dtd/technicalContent/dtd/"></public>
```

The XML editor then follows the relative path given in the xml:base attribute and in that location looks for a file named reference.dtd. It finds one. The XML editor then compares the current DITA reference topic against the structure defined in reference.dtd. If there are no structural violations, the XML editor is happy. If there are structural violations, the XML editor displays a message.

One more thing about catalogs. A catalog can reference another catalog. Some catalogs don't contain any actual pointers from FPIs to DTDs. They only contain pointers to other catalogs. This is a convenient way of consolidating everything. Instead of the XML editor needing to point to dozens of different catalogs, one for each XML standard writers might use, the XML editor only

needs to point to a single catalog and that single catalog in turn points to the dozens of different catalogs. If you look in a catalog file and you see something like:

```
<nextCatalog catalog="foo/catalog-foo.xml"/>
```

that means the catalog is pointing to another catalog named catalog-foo.xml in the *foo* subfolder. You might have to go to that catalog to find the FPI you are looking for.

Now that you have a basic understand of DTDs and catalogs, it's time to look specifically at DITA DTDs. Most of what you learned applies to DITA DTDs, but they are built quite differently from DTDs that you might have created yourself or used previously.

In addition to the <!ELEMENT> and <!ATTLIST> elements you saw earlier, in the addressbook DTD for example, there are two other basic DTD elements:

- <!ENTITY>
- <!ENTITY %>

These elements are used heavily in the DITA DTDs.

The first, <!ENTITY>, is a general entity. It defines a value that can be used within an XML document that references the DTD. For example, you might add lines 13 and 14 and edit line 6 within your addressbook DTD:

```
1  <!DOCTYPE addressbook [
2  <!ELEMENT addressbook (entry+) >
3  <!ELEMENT entry   (lname, fname, address+, (phone* | email*)*, bdate?) >
4  <!ELEMENT fname   (#PCDATA) >
5  <!ELEMENT lname   (#PCDATA) >
6  <!ELEMENT address (street+, city, state, zip, country) >
7  <!ATTLIST address type  (home|work|mailing|other)  #REQUIRED >
8  <!ELEMENT phone   (#PCDATA) >
9  <!ATTLIST phone   type  (home|cell|work|fax|other)  #REQUIRED >
10  <!ELEMENT email   (#PCDATA) >
11  <!ATTLIST email   type  (home|work)  #REQUIRED >
12  <!ELEMENT bdate   (#PCDATA) >
13  <!ELEMENT country   (#PCDATA) >
14  <!ENTITY nation "USA" >
]>
```

Line 6 now specifies that there must also be a <country> element within <address>. Line 13 defines the <country> element. Line 14 creates an entity named *nation* with a value of *USA*.

In your addressbook XML file, you can now use this general entity. You can use a general entity almost anywhere #PCDATA is allowed. In this example, it makes sense to use it in the <country> element:

```
<entry>
   <lname>Smith</lname>
   <fname>Sally</fname>
   <address type = "home">
```

```
      <street>100 Elm St.</street>
      <street>#453</street>
      <city>Springfield</city>
      <state>GA</state>
      <zip>31000</zip>
      <country>&nation;</country>
   </address>
   <phone type = "home">222-555-1212</phone>
   <phone type = "cell">222-555-5555</phone>
   <email type = "home">sallysmith@email.com</email>
</entry>
```

The ampersand followed by the entity name followed by a semi-colon (`&nation;`) is the standard way to reference a general entity.

Most XML editors will resolve the entity so that you see the actual text (in this case, “USA”) in your editor rather than the entity reference. And of course, when you build output from the XML file, the processor resolves it as well.

In this example, the text is so short that you might wonder what's the point of using the entity—it would be just as fast to type “USA”. That's true in this case, but imagine a case where the entity text is quite lengthy and not easy to type, or when the value of the entity is likely to change over time. In that kind of situation, an entity can be a real time-saver.

You can also use a general entity to refer to a file:

```
<!ENTITY legal SYSTEM "legal_2014.xml">
```

In this case, the entire contents of legal_2014.xml will be inserted into your XML document at the point where the entity is used, including the XML and DOCTYPE declarations and all markup. This use of general entities is not very common because the presence of the declarations severely limits the contexts in which the content is valid.

More common uses of general entities that refer to files are references to an image:

```
<!ENTITY logo SYSTEM "logo2014.png" NDATA png>
<!NOTATION png SYSTEM "image/png">
```

or references to a non-XML file:

```
<!ENTITY legal SYSTEM "legal_2014.txt">
```

This second usage is a lot more flexible because there is no internal structure in the text fle that has to be validated at the point of insertion. This usage is handy for paragraphs or chunks of text that, while you could add them directly to the DTD, are better kept in separate files.

Notice that all of these general entities that refer to external files use a SYSTEM reference, which means they imply a relative path between the DTD and the resource which must be kept intact.

The second entity element, <!ENTITY %>, is more complicated. This kind of entity is called a *parameter entity* and it can only be used within DTDs, unlike general entities, which as you just saw, can be used within XML files.

Parameter entities are used primarily to build DTDs and reduce redundancy. For example, imagine that you are defining an attribute and that attribute has 10 values. Now imagine that the attribute is used on 20 different elements. Typing this:

```
<!ATTLIST element1 type (v1|v2|v3|v4|v5|v6|v7|v8|v9|v10)  #REQUIRED >
<!ATTLIST element2 type (v1|v2|v3|v4|v5|v6|v7|v8|v9|v10)  #REQUIRED >
<!ATTLIST element3 type (v1|v2|v3|v4|v5|v6|v7|v8|v9|v10)  #REQUIRED >
<!ATTLIST element4 type (v1|v2|v3|v4|v5|v6|v7|v8|v9|v10)  #REQUIRED >
```

in your DTD 16 more times is going to get tedious.

Instead, it's a lot easier to define

```
(v1|v2|v3|v4|v5|v6|v7|v8|v9|v10)
```

as a parameter entity and simply reference the entity for each element that uses the type attribute. This is the way the DITA DTDs are built, because DITA is extremely repetitive. For example, the select attributes (audence, platform, product, props, otherprops, importance, rev, status, base) can be used on almost every DITA element. No one wants to type that list over and over again in the <!ATTLIST> for each DITA element that uses @audience. Instead, they are defined as a group in an entity and that entity is referenced throughout the DITA DTDs.

Here's an example of a simple DTD that uses parameter entities:

```
<?xml version="1.0" encoding="UTF-8"?>
1  <!ENTITY % status "(essential | useless | silly)" >
2  <!ENTITY % info "(name, category, location, owner)" >
3  <!ELEMENT stuff ((gadget* | widget* | thingamabob*))+ >
4  <!ELEMENT gadget %info; >
5  <!ELEMENT widget %info; >
6  <!ELEMENT thingamabob %info; >
7  <!ELEMENT name (#PCDATA) >
8  <!ELEMENT category (#PCDATA) >
9  <!ELEMENT location (#PCDATA) >
10  <!ELEMENT owner (#PCDATA) >
11  <!ATTLIST gadget importance %status;  #REQUIRED >
12  <!ATTLIST widget importance %status;  #REQUIRED >
13  <!ATTLIST thingamabob importance %status;  #REQUIRED >
```

> **Note:** Again, the numbers on the left are not part of the DTD. They are just for reference in the following explanation.

1. This line defines a parameter entity named status. Its value is (essential | useless | silly).
2. This line defines a parameter entity named info. Its value is (name, category, location, owner).
3. This line defines an element named <stuff> and its child elements.
4. This line defines an element named <gadget>. Its contents are the value of the info entity.

5. This line defines an element named <widget>. Its contents are the value of the info entity.
6. This line defines an element named <thingamabob>. Its contents are the value of the info entity.
7. This line defines an element named <name>. It contains PCDATA.
8. This line defines an element named <category>. It contains PCDATA.
9. This line defines an element named <location>. It contains PCDATA.
10. This line defines an element named <owner>. It contains PCDATA.
11. This line defines an attribute for the <gadget> element named *importance*. Its allowed values are the value of the status entity.
12. This line defines an attribute for the <widget> element named *importance*. Its allowed values are the value of the status entity.
13. This line defines an attribute for the <thingamabob> element named *importance*. Its allowed values are the value of the status entity.

Unlike general entities, which can be defined anywhere in a DTD, parameter entities must be defined before they are used.

If it's still not clear how parameter entities are used, think of it this way: in the DTD above, everywhere you see `%info;`, substitute the value of the info entity, which is (name, category, location, owner). So line 4 is parsed into

```
<!ELEMENT gadget (name, category, location, owner) >
```

just as if you had typed those elements there manually.

How does the parsing happen? Again, an XML editor will do the job, or a build tool like the DITA Open Toolkit. Some internet browsers can also parse DTDs. It all happens behind the scenes, so you don't need to worry about the mechanics.

Now that you have a basic understanding of DTDs and entities, it's time to look specifically at the DITA DTDs.

Related Links

Understanding the DITA DTDs on page 43

Understanding the DITA DTDs

Now that you understand the basics of DTD's, it's time to apply that understanding to the DITA DTDs. Seeing how they use entities to avoid repetition should make the idea clearer to you.

Take a look at the topic.dtd file (found in {DITA-OT}/dtd/technicalContent/dtd for Open Toolkit versions 1.5.4 and later).

If you look at that DTD, you see that it has almost no declarations of its own. It's almost all entities. For example, at the beginning of the file are a number of sections similar to this one:

```
<!ENTITY % hi-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 1.2 Highlight Domain//EN"
         "../../base/dtd/highlightDomain.ent"
>%hi-d-dec;
```

From the introduction, you recognize this section as a definition for a parameter entity named hi-d-dec, with a PUBLIC identifier of `-//OASIS//ENTITIES DITA 1.2 Highlight Domain//EN` and a SYSTEM identifier of `../../base/dtd/highlightDomain.ent`.

The PUBLIC identifier, if you look at the catalog, points to the file highlightDomain.ent in *dtd/base/dtd*, which ends up being the same file and location as the SYTEM identifier. If you then have a look at the highlightDomain.ent file, you see that the entity hi-d-dec is defined as

```
<!ENTITY % hi-d-ph
  "b |
   i |
   sup |
   sub |
   tt |
   u
  "
>
```

> **Note:** The name of this entity follows a DITA convention. “hi” indicates the name of the domain, “d” indicates that it's a domain, and “ph” is the element that is extended. In other words, all of the elements in the highlight domain can be used anywhere that <ph> can be used.

Back in topic.dtd, notice that the entity is used immediately after it's declared. The last line of that code (`%hi-d-dec;`) is actually calling the just-declared entity.

Farther down in topic.dtd is this section:

```
<!ENTITY % ph           "ph |
                         %hi-d-ph; |
                         %pr-d-ph; |
                         %sw-d-ph; |
                         %ui-d-ph;
                        ">
```

Where for the <topic> element, we are stating (in a somewhat roundabout, entity-based way) that the <ph> element is, of course, itself, but is also extended to the elements in the highlight domain (those defined in the hi-d-ph entity) as well as the elements in the programming domain

(pr-d-ph entity), the software domain (sw-d-ph entity), and the user interface domain (ui-d-ph entity).

Still farther down in topic.dtd, there is a section named TOPIC ELEMENT INTEGRATION. This section calls the topic-type entity which is defined in topic.mod. Skip that section for now.

Just below that section is a section named DOMAIN ELEMENT INTEGRATION. You can see that section includes a number of sections like this one:

```
<!ENTITY % hi-d-def
  PUBLIC "-//OASIS//ELEMENTS DITA 1.2 Highlight Domain//EN"
         "../../base/dtd/highlightDomain.mod"
>%hi-d-def;
```

Where again, an entity named hi-d-def is being defined and immediately resolved. This entity's identifiers point you to a file named highlightDomain.mod.

> **Note:** The file extensions .ent have no special meaning within DITA or XML in general. They are simply a convention that has been adopted. The understanding is that .ent files define entities and .mod files use those entities to build elements.

Taking a look inside of that file, you see several sections similar to this one:

```
<!--                    LONG NAME: Bold                                 -->
<!ENTITY % b.content
                       "(#PCDATA |
                         %basic.ph; |
                         %data.elements.incl; |
                         %foreign.unknown.incl;)*"
>
<!ENTITY % b.attributes
             "%univ-atts;
              outputclass
                        CDATA
                                  #IMPLIED"
>
<!ELEMENT b    %b.content;>
<!ATTLIST b    %b.attributes;>
```

There is a similar section for each of the six highlight domain elements.

The first part of the section defines an entity named b.content, which in turn can contain #PCDATA and three other entities: basic.ph, data.elements.incl, and foreign.unknown.incl. These three entities are defined in commonElements.mod.

The second part of the section defines an entity named b.attributes, which in turn can contain another entity: univ-atts. This entity is defined in commonElements.mod.

If you look at commonElements.mod, you see that the entities basic.ph, data.elements.incl,foreign.unknown.incl, and univ-atts are in turn defined by other entities. The DITA DTDs are extremely layered in this respect.

Finally, the last two lines of the section are the more familiar ELEMENT and ATTLIST definitions, which declare the elements valid in <topic> to be the resolved value of the b.content entity and the attributes valid for <topic> to be the resolved value of the b.attributes entity. Here are two examples of where the actual DTD is built.

As you've briefly seen, fully resolving those entities can involve digging through a number of different .ent and .mod files within the DITA DTD set.

You can imagine that trying to fully envision this DTD with all of the entities resolved is difficult. You have to refer to many other files to see what each parameter entity ultimately resolves to. There is, unfortunately, no magic bullet solution for this. The good news is that it's not necessary to fully understand the DITA DTD architecture to be able to extend DITA by creating and integrating your own custom DTDs.

An excellent place to start in understanding specialization and extension is Eliot Kimber's tutorial at **xiruss.org**. Jarno Eloverta (one of the DITA Open Toolkit developers) also has an online DITA DTD generator at the **Google App Engine**.

Related Links

What is a DTD? on page 35