FBX Assets File Format

Version 1.0 - September 2008

Copyright Notice

Table of Contents

Overview

FBX Assets is a framework that allows you to create, edit, and manage asset templates. An asset template defines the interface of an asset. In other words, it specifies the properties an asset must have in order to comply with a specific asset type.

Asset templates are built in XML files called template files. This document describes the XML template file format.

Goals

The following goals served as guidelines throughout the design of the *FBX Assets* framework, its API, and its file format:

- **Human Readability**: A user should be able to understand and modify the content of a template file.
- **Versioning**: A single template file must be able to describe multiple versions of the same template, ideally the whole template history. Versioning is not described in the current version of this document.
- **Multiple Inheritance**: A template can inherit properties from one or more other templates.
- **Type Definition**: New types can be defined using a set of built-in types. These new types can be simple types, enumerations, templates, or arrays of any of these.
- **Localization**: Text that is used to label or describe elements in the file must be localizable in any language. All localized versions of a label or description must be stored in the template file. The format must allow encoding of a mix of single-byte and multi-byte characters.
- **Packaging**: Template files must belong to a package, allowing template authors to group multiple template files together. Packages may be unnamed if reserved for internal use, but they must be named if published.
- **Namespaces**: Type names must be namespaced using the package name, in order to prevent name collisions across package boundaries. Type names must be unique within a package.
- **Referencing**: A template must be able to reference types and templates defined in other template files.
- **Locking**: Templates must allow locking assets in part or in whole in order to limit the changes a user is allowed to perform.
- **Views**: Different users may want to view a given template differently depending on their workflow.

File Locations

When resolving types, the *FBX Assets* runtime looks for template files using their package names and the active search paths. The following section provides an overview of these concepts.

Search Paths

A search path is a file directory the *FBX Assets* runtime looks for files when attempting to resolve a type name. The *FBX* Assets framework maintains a list of active search paths at all times. For example:

```
c:\myproject\templates
\\server\templates
```

The *FBX Assets* runtime parses all search paths in the order they were added to the configuration. For example, when looking for a template named *Vehicle*, the *FBX Assets* runtime first attempts to open a file named *c:\myproject\templates\Vehicle.template*. If this file doesn't exist, it then attempts to open *\\server\templates\MyTemplate.Vehicle*, and so on. The search stops as soon as the desired template is found.

The search path list can be updated during a session. The *FBX Assets* runtime does not unload all previously loaded template files when the corresponding search path is removed from the list.

Packages

Each template file belongs to a package, whether explicitly or implicitly. Package names have two distinct purposes:

- Package names act as namespaces and prevent type name conflicts across package boundaries. Using a unique package name guaranties that all types within it will have a unique name in the current *FBX Assets* runtime.
- Packages facilitate template distribution, allowing you to group all templates that pertain to a given project under a single directory structure.

The package name is specified in the *templates* XML element:

```
<templates package=”com.mystudio.mygame”>
    <template name=”Vehicle”/>
</templates>
```

If no package name is specified, an empty package name is assumed. All types have their names prefixed with the package name in order to form a unique name. In the example above, the full name of the template is *com.mystudio.mygame.Vehicle*.

The package name determines the physical location of the template file relative to the search path. When attempting to resolve a template named *com.mystudio.mygame.Vehicle*, the *FBX Assets* runtime looks for subdirectories whose names correspond to the package names under all active search paths. Using the search path above, the *FBX Assets* runtime first attempts to open a file named *c:\myproject\templates\com\mystudio\mygame\Vehicle.template*. If this file doesn't exist, it attempts to open *\\server\templates\com\mystudio\mygame\Vehicle.template*, and so on. The search stops when the desired template is found.

Although not a requirement, it is preferable to define one template per template file whose name matches the template name. This allows the *FBX Assets* runtime to locate and load the template automatically using its full name.

All external types referred to within a template file must either be prefixed with their full package name or be located in one of the package names declared at the beginning of the file. One or more *using* declarations can be used to enumerate external package names.

```
<templates package="com.mystudio.mygame">
    <using package="std"/>
    <using package="com.autodesk.hik"/>
    <template name="MyTemplate"/>
</templates>
```

The *FBX Assets* runtime automatically loads all files contained in packages referenced by a *using* declaration. If a large number of template files are located in the same directory (meaning they share the same package), they are all loaded when the package is referenced by a *using* declaration.

The following strategies can help you optimize type referencing, memory footprint, and load time:

- Regroup all template files that define types other than templates in one small package. This package can then be referenced by all other template files without causing the other files to be loaded at the same time.
- When referencing a template, use its full name (*com.mystudio.mygame.MyTemplate*). This allows the *FBX Assets* runtime to resolve its file location without requiring a *using* declaration, and to load only the necessary file.

The *FBX Assets* framework includes a standard package named *std* that contains basic types useful for describing 3D assets.

Types

Types are the basic building blocks used to define asset templates. All types must have unique names within their respective package. A valid name consists of one or more characters within the range [a-z][A-Z][0-9][_].

FBX Assets types can be grouped into the following categories:

- Built-in types
- Simple types
- Custom types
- Enumerations
- Arrays
- Templates

Built-in Types

Built-in types are defined in the core of the *FBX Assets* framework and are always present. They have no package name. The following table provides a complete list of the built-in types, along with the corresponding C++ types:

FBX Assets Type	C++ Type
BOOL	bool
INT8	char
INT16	short
INT32	int
INT64	long long
UINT8	unsigned char
UINT16	unsigned short
UINT32	unsigned int
UINT64	unsigned long long
FLOAT	float
DOUBLE	double
STRING	char * (UTF-8 encoding)
VOID	void
REFERENCE	N/A (Reference to a node in a scene)

Simple Types

Simple types are types that are based on a built-in type or on other simple types. Simple types act as aliases of their base type and the two are interchangeable.

For example:

```
<templates package=”com.mystudio.mygame”>
    <using package=”std”/>
    <type name=”Weight” type=”FLOAT”/>
    <type name=”Speed” type=”DOUBLE”/>
    <type name=”CarVisibility” type=”std.Visibility”/>
</templates>
```

In this example, *CarVisibility* is based on *std.Visibility*, which is defined in the *std* package. This reference requires the *using* declaration at the top of the file, which loads all files contained in the *std* package in order to load the declaration of *std.Visibility*.

Custom types

When the *FBX Assets* runtime runs into an undefined type, it simply issues a warning. A custom type is defined dynamically at load time as a placeholder for the unknown type until the end of the session. Custom types can be saved to the template file without losing information, but they cannot be assigned a value or used in any way.

Due to their short life cycle and other limitations, custom types are only intended to handle special cases or error conditions.

Enumerations

Enumerations are types that contain a list of items. Each item can associate a name with a value. Enumerations also limit the range of valid values for a given type.

The following example shows the days of the week defined by an enumeration type:

```
<templates package=”com.mystudio.mygame”>
    <enumeration name=”DayOfTheWeek” type=”INT32”>
        <item name=”Sunday” value=”1”/>
        <item name=”Monday” value=”2”/>
        <item name=”Tuesday” value=”3”/>
        <item name=”Wednesday” value=”4”/>
        <item name=”Thursday” value=”5”/>
        <item name=”Friday” value=”6”/>
        <item name=”Saturday” value=”7”/>
    </enumeration>
</templates>
```

Arrays

Arrays are types containing a list of elements. All elements contained in an array must share a common base type, which is also the type of the array. Arrays can contain elements of any type, including other arrays or templates.

The number of elements in an array is called the dimension and it is specified using the *dim* attribute. Non-array types have an implied dimension of 1. Dynamic arrays have a dimension of 0.

This example includes three arrays:

```
<templates package="com.mystudio.mygame">
    <using package="std"/>
    <type name="RGB" type="DOUBLE" dim=3/>
    <type name="Windows" type="Window" dim=4/>
    <type name="AnimationClips" type="std.FCurve" dim=0/>
</templates>
```

Templates

Templates are also types. This implies that they can be used whenever a type can be used, such as when defining arrays.

The following section describes in more detail how to define templates.

Templates

A template is a collection of properties. It can also extend other templates, inheriting all their properties.

Attributes

The main property in a template is called an attribute. An attribute has a name, which must be unique within its template, and a type, which must be defined before the attribute declaration.

When a template is assigned to an asset, attributes should be bound to compatible physical attributes of the objects inside the asset.

```
<templates package="com.mystudio.mygame">
    <template name="Vehicle">
        <attribute name="Color" type="std.Color"/>
        <attribute name="Weight" type="FLOAT"/>
    </template>
</templates>
```

Inheritance

A template can extend another template as long as their relationship is non-cyclical. Parent templates are specified using one or more *extend* declarations.

This example shows simple inheritance:

```
<templates package="com.mystudio.mygame">
    <template name="GroundVehicle">
        <extends template="Vehicle"/>
    </template>
    <template name="Car">
        <extends template="GroundVehicle"/>
    </template>
    <template name="Motorcycle">
        <extends template="GroundVehicle"/>
    </template>
</templates>
```

This example shows multiple inheritance:

```
<templates package="com.mystudio.mygame">
    <template name="PublicTransportation"/>
    <template name="Bus">
        <extends template="PublicTransportation"/>
        <extends template="GroundVehicle"/>
    </template>
</templates>
```

Views

The *FBX Assets* framework supports creating multiple views of a given template to allow different users to have a different perspective on the same asset. Views are defined outside the *template* element. They can even be stored in separate XML files if you need to separate them from template definitions.

Views can contain two types of items: view properties and groups. A view property specifies that a given property is present in the view by referring to its name. Properties that are not explicitly mentioned in a view are not visible when the view is active.

The following example defines a view called *Painter* that exposes the *Color* property only.

```
<templates package="com.mystudio.mygame">
    <view name="Painter" template="Vehicle">
        <property name="Color"/>
    </view>
</templates>
```

Groups allow properties to be arranged under a common parent, letting users browse them by category. Groups can be nested.

This example view places the *Weight* property under a group called *PhysicalProperties*.

```
<templates package="com.mystudio.mygame">
    <view name="Physics" template="Vehicle">
        <group name="PhysicalProperties">
            <property name="Weight"/>
        </group>
    </view>
</templates>
```

Labels

Labels let you supply user-friendly names for *FBX Assets* objects. They can be attached to templates, types, attributes, views, properties, groups, or any object defined in a template file. Labels should not exceed a few words and are intended to contain alternate display names for the object. Labels can be used to display tooltips when the mouse hovers over a field and for prefixing a control with localized, user-friendly text in a form.

Labels are localized, which means that a label only makes sense in the context of its language code. *FBX Assets* objects can have one label per language code.

The language of a label is specified using a 2-letter ISO language code, such as *en* for English or *fr* for French. Optionally, language codes can be suffixed with 2-letter country codes to specify a regional variant, such as *en_UK* or *fr_CA*.

When using a language code to retrieve a label, the *FBX Assets* runtime first looks for an exact match. If no label is found using a language code containing a country code, the *FBX Assets* runtime then looks for a label that matches the language code without the country code (the international variant). If still no label is found, the *FBX Assets* runtime looks for a label with no language code. If it exists, the label is considered the default label and is returned for all languages for which no language code is specified. In the event that all these attempts fail, the *FBX Assets* runtime concludes that the given object has no labels for the given language.

Labels are defined using the *label* XML element. The language code is specified using the optional *language* attribute. The label text is contained between the opening and closing XML tags, and is encoded using UTF-8.

The following example template has labels for four language variants:

```
<templates package=”com.mystudio.mygame”>
    <template name=”Drugstore”>
        <label language=”en”>Drugstore</label>
        <label language=”en_UK”>Chemist</label>
        <label language=”en_CA”>Pharmacy</label>
        <label language=”fr”>Pharmacie</label>
    </template>
</templates>
```

Descriptions

Descriptions provide the ability to associate longer, user-friendly text annotations to *FBX Assets* objects. They can be attached to templates, types, attributes, views, properties, groups, or any object defined in a template file. There is no limit to the size of a description. You can use descriptions for example to add instructions on how an object should be used, or to provide examples of its usage.

Descriptions are localized, which means that a description only makes sense in the context of its language code. *FBX Assets* objects can have one description per language code.

Like labels, the language of a description is specified using a 2-letter ISO language code, such as *en* for English or *fr* for French. Optionally, language codes can be suffixed with 2-letter country codes to specify a regional variant, such as *en_UK* or *fr_CA*.

When using a language code to retrieve a description, the *FBX Assets* runtime first looks for an exact match. If no description is found using a language code containing a country code, the *FBX Assets* runtime looks for a description that matches the language code without the country code (the international variant). If still no description is found, the *FBX Assets* runtime looks for a description with no language code. If it exists, the description is considered the default and is returned for all languages for which no language code is specified. If all these attempt have fail, the *FBX Assets* runtime concludes that the given object has no description for the given language.

Descriptions are defined using the *description* XML element. The language code is specified using the optional *language* attribute. The description text is contained between the opening and closing XML tags, and is encoded using UTF-8.

The following example shows a view that has descriptions for two of its items in two languages:

```
<templates package=”com.mystudio.mygame”>
    <view name=”Physics” template=”Vehicle”>
        <group name=”PhysicalProperties”>
            <description language=”en”>This group contains physical properties.</label>
            <description language=”fr”>Ce groupe contient les propriétés physiques.</label>
            <property name=”Weight”>
                <description language=”en”>The object weight in kg.</label>
                <description language=”fr”>Le poids de l’objet en kg.</label>
            </property>
        </group>
    </view>
</templates>
```

NOTE: In the example above, the UTF-8 encoding has been omitted for clarity.

Comments

Comments can be attached to any *FBX Assets* object. These comments are intended for the template author and should not be visible to the template user.

XML comments contained in a template file are attached to the XML element that follows them. The *FBX Assets* runtime reads the comments when loading the file and restores them when saving the file.

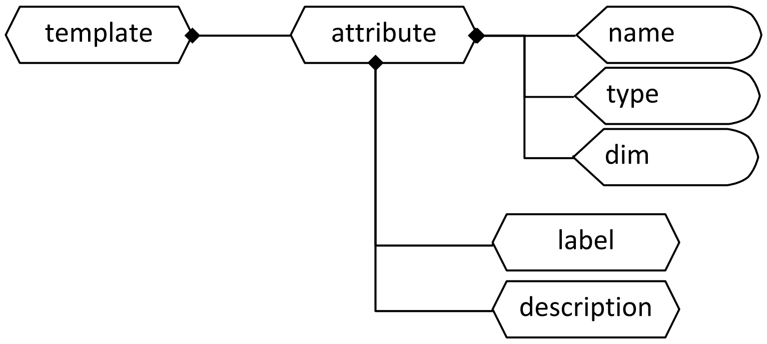
In the following example, the comment is attached to the *Drugstore* template element.

```
<templates package=”com.mystudio.mygame”>
    <!-- TODO: Add translations for Italian and Spanish -->
    <template name=”Drugstore”>
        <label language=”en”>Drugstore</label>
        <label language=”fr”>Pharmacie</label>
    </template>
</templates>
```

Reference

attribute

Use the *attribute* element to define a template property that describes an attribute. Attributes must have a name and a type. In specific cases, they can also override the type dimension.



Attributes

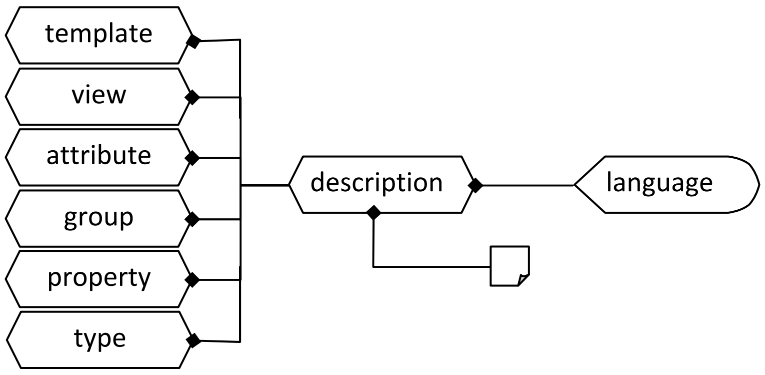
Attribute	Required	Description	Values
name	✓	The name of the attribute. This name is case sensitive and must be unique within the template. The full attribute name is obtained by concatenating the template name, a period, and the attribute name.	String containing alphanumeric characters or periods.
type	✓	The name of the type of the attribute. The type must have been defined before it can be used by an attribute.	String containing alphanumeric characters or periods.
dim	☐	The dimension of the type. Attributes are only allowed to override the dimension of dynamic arrays, that is, types whose dimension is equal to 0.	Integer greater than 0.

Example

```
<templates package=”com.mystudio.mygame”>
    <template name=”Vehicle”>
        <attribute name=”Color” type=”std.Color”/>
        <attribute name=”Weight” type=”FLOAT”/>
    </template>
</templates>
```

description

Use the *description* element to attach long, user-friendly descriptions to *FBX Assets* objects. These descriptions can be attached to templates, types, attributes, views, properties, groups, or any object defined in a template file.



Attributes

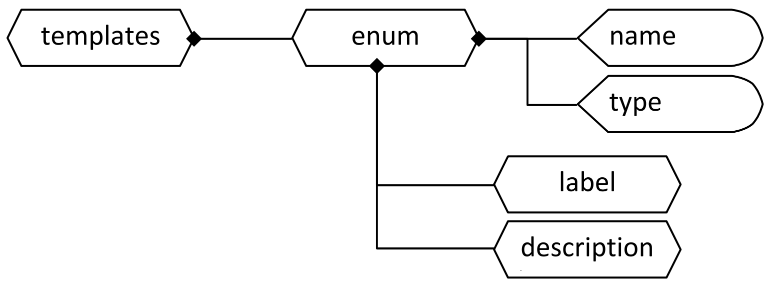
Attribute	Required	Description	Values
language	☐	The 2-letter ISO language code. Optionally, this string can be suffixed with an underscore followed by a 2-letter country code to designate a language variant. If the language is not specified, the description is considered the default description.	String containing alphanumeric characters only.
TEXT	✓	The description text encoded using UTF-8. It can be of any length and contain any number of paragraphs.	UTF-8 characters.

Example

```
<templates package=”com.mystudio.mygame”>
    <view name=”Physics” template=”Vehicle”>
        <group name=”PhysicalProperties”>
            <description language=”en”>This group contains physical properties.</label>
            <description language=”fr”>Ce groupe contient les propriétés physiques.</label>
            <property name=”Weight”>
                <description language=”en”>The object weight in kg.</label>
                <description language=”fr”>Le poids de l’objet en kg.</label>
            </property>
        </group>
    </view>
</templates>
```

enum

Use the *enum* element to define a new enumeration. Enumerations are types that contain a list of items. Each item lets you associate a name with a value.



Attributes

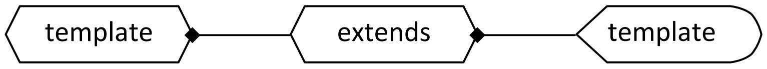
Attribute	Required	Description	Values
name	✓	The name of the enumeration. This name is case sensitive and must be unique within the package. The full enumeration name is obtained by concatenating the package name, a period, and the name of the enumeration.	String containing alphanumeric characters only.
type	✓	The type the enumeration is based on. The base type can be in the same package or in another package.	String containing alphanumeric characters or periods.

Example

```
<templates package=”com.mystudio.mygame”>
    <enumeration name=”DayOfTheWeek” type=”INT32”>
        <item name=”Sunday” value=”1”/>
        <item name=”Monday” value=”2”/>
        <item name=”Tuesday” value=”3”/>
        <item name=”Wednesday” value=”4”/>
        <item name=”Thursday” value=”5”/>
        <item name=”Friday” value=”6”/>
        <item name=”Saturday” value=”7”/>
    </enumeration>
</templates>
```

extends

Use the *extends* element to specify which template a given template inherits from. Templates can extend one or more templates. In the absence of any *extends* declaration, templates implicitly inherit from *Object*.



Attributes

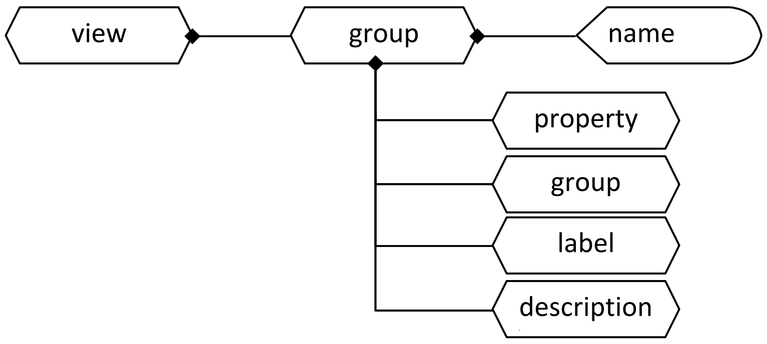
Attribute	Required	Description	Values
template	✓	The name of the template to inherit from. The parent template must be defined before it can be used by another template. The parent template can belong to the same package or to another package.	String containing alphanumeric characters or periods.

Example

```
<templates package=”com.mystudio.mygame”>
    <template name=” GroundVehicle”/>
    <template name=”Car”>
        <extends template=”GroundVehicle”/>
    </template>
</templates>
```

group

Use the *group* element to define a view group. View groups can contain view properties or other groups.



Attributes

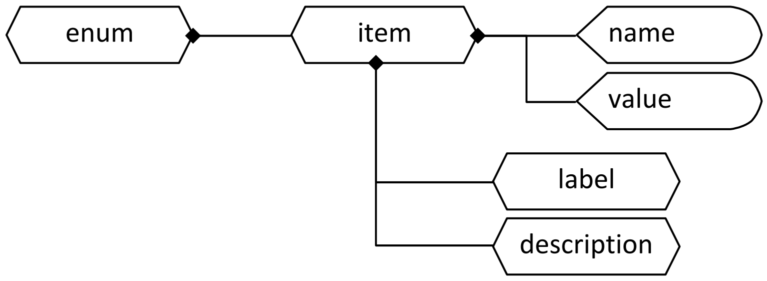
Attribute	Required	Description	Values
name	✓	The name of the group. This name is case sensitive and must be unique within the package. The full template name is obtained by concatenating the package name, a period, and the group name.	String containing alphanumeric characters only.

Example

```
<templates package=”com.mystudio.mygame”>
    <view name=”Painter” template=”Vehicle”>
        <group name=”Appearance”>
            <property name=”Color”/>
        </group>
    </view>
</templates>
```

item

Use the *item* element to define a new item within an enumeration. Enumeration items must have a name and a value.



Attributes

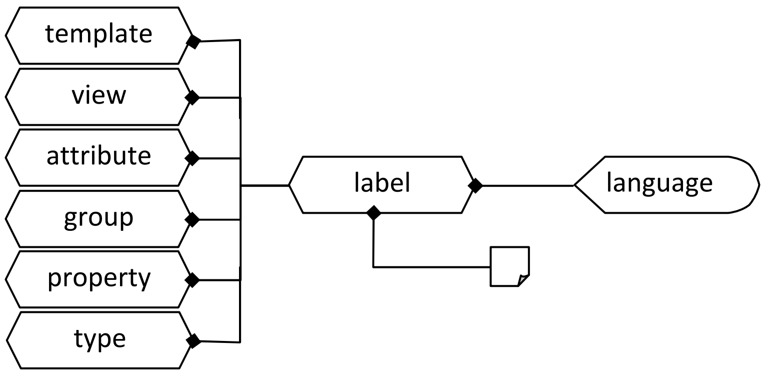
Attribute	Required	Description	Values
name	✓	The name of the enumeration item. This name is case sensitive and must be unique within the enumeration.	String containing alphanumeric characters only.
value	✓	The value of the enumeration item. The value must be compatible with the enumeration type.	Depends on the enumeration type.

Example

```
<templates package=”com.mystudio.mygame”>
    <enumeration name=”DayOfTheWeek” type=”INT32”>
        <item name=”Sunday” value=”1”/>
        <item name=”Monday” value=”2”/>
        <item name=”Tuesday” value=”3”/>
        <item name=”Wednesday” value=”4”/>
        <item name=”Thursday” value=”5”/>
        <item name=”Friday” value=”6”/>
        <item name=”Saturday” value=”7”/>
    </enumeration>
</templates>
```

label

Use the *label* element to give *FBX Assets* objects a user-friendly name. These labels can be attached to templates, types, attributes, views, properties, groups, or any object defined in a template file.



Attributes

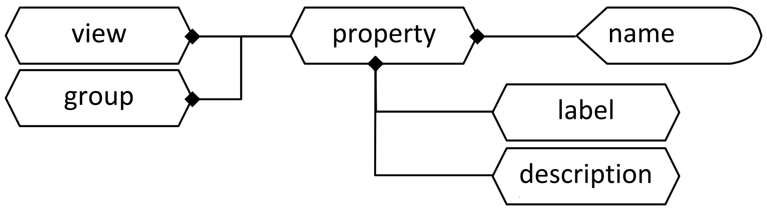
Attribute	Required	Description	Values
language	☐	The 2-letter ISO language code. Optionally, this string can be suffixed with an underscore followed by a 2-letter country code to designate a language variant. If the language is not specified, the label is considered the default label.	String containing alphanumeric characters only.
TEXT	✓	The label text is encoded using UTF-8. It should not exceed a few words and cannot contain any paragraph marks.	UTF-8 characters.

Example

```
<templates package=”com.mystudio.mygame”>
    <template name=”Drugstore”>
        <label language=”en”>Drugstore</label>
        <label language=”en_UK”>Chemist</label>
        <label language=”en_CA”>Pharmacy</label>
        <label language=”fr”>Pharmacie</label>
    </template>
</templates>
```

property

Use the *property* element to define a view property. A view property must refer to an already-defined property within the template the view is based on.



Attributes

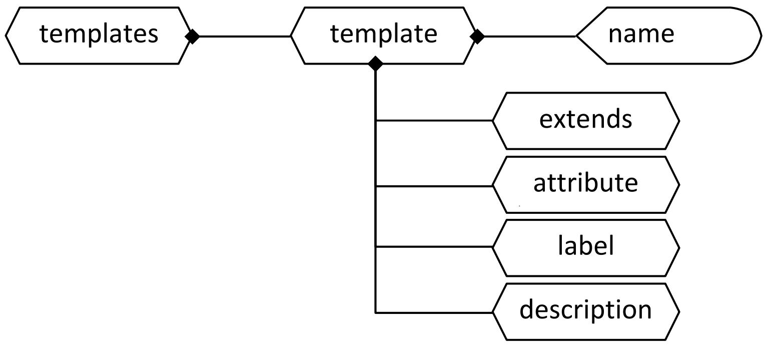
Attribute	Required	Description	Values
name	✓	The name of the property. This name is case sensitive and must exist in the template referenced by the view.	String containing alphanumeric characters or periods.

Example

```
<templates package=”com.mystudio.mygame”>
    <view name=”Painter” template=”Vehicle”>
        <property name=”Color”/>
    </view>
</templates>
```

template

Use the *template* element to define a new template. A template is a complex type that is composed of one or more properties.



Attributes

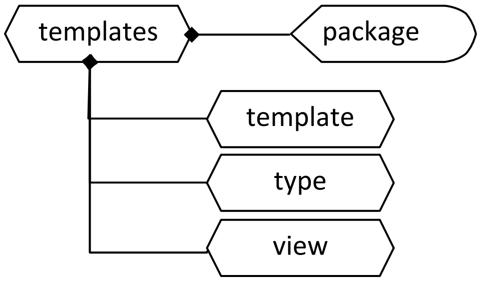
Attribute	Required	Description	Values
name	✓	The name of the template. Template names are case sensitive and must be unique within their package. The full template name is obtained by concatenating the package name, a period, and the template name.	String containing alphanumeric characters only.

Example

```
<templates package=”com.mystudio.mygame”>
    <template name=”Car”>
        <extends template=”GroundVehicle”/>
        <attribute name=”Hood” type=”com.example.CarPart”/>
    </template>
</templates>
```

templates

The *templates* element is the root of the template file. It contains all type, template, and view definitions.



Attributes

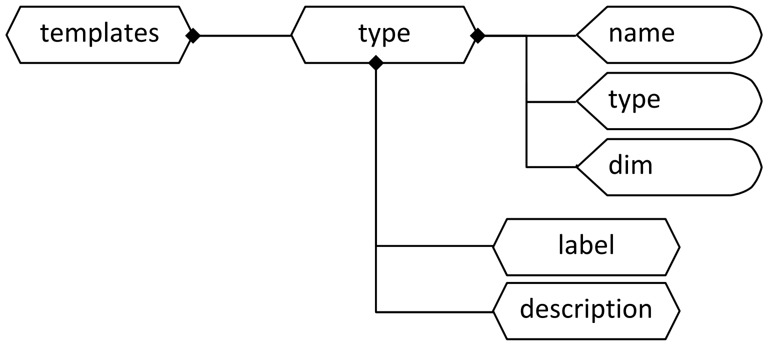
Attribute	Required	Description	Values
package	☐	The name of the package where this template file is located. When not specified, the package name is empty.	String containing alphanumeric characters or periods.

Example

```
<templates package=”com.mystudio.mygame”>
    <template name=”Vehicle”>
    </template>
</templates>
```

type

Use the *type* element to define a new type. A type must have a base type and can also have a dimension.



Attributes

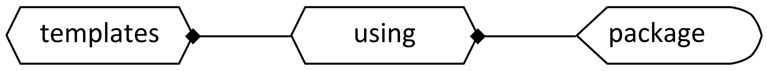
Attribute	Required	Description	Values
name	✓	The name of the type. This name is case sensitive and must be unique within the package. The full type name is obtained by concatenating the package name, a period, and the name of the type.	String containing alphanumeric characters only.
type	✓	The type the new type is based on. The base type can belong to the same package or to another package.	String containing alphanumeric characters or periods.
dim	☐	The dimension of the type. By default, a type has a dimension of 1. A value above 1 defines a static array of the corresponding size. A value of 0 defines a dynamic array whose size is determined at instantiation time.	An integer greater than or equal to zero.

Example

```
<templates package="com.mystudio.mygame">
    <using package="std"/>
    <type name="RGB" type="DOUBLE" dim=3/>
    <type name="Windows" type="Window" dim=4/>
    <type name="AnimationClips" type="std.FCurve" dim=0/>
</templates>
```

using

Use the *using* element to include all template files contained in a given package.



Attributes

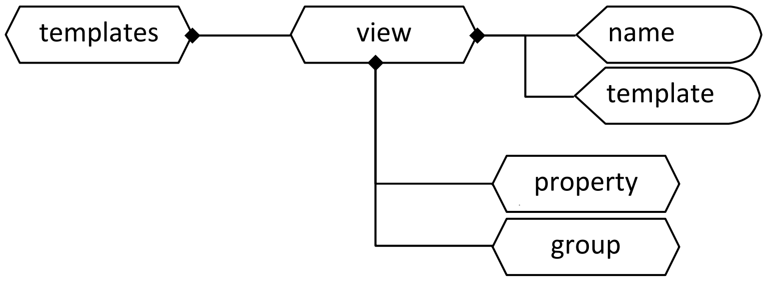
Attribute	Required	Description	Values
package	✓	The name of the package to include. This name is case sensitive and can contain periods if sub-packages are specified.	String containing alphanumeric characters or periods.

Example

```
<templates package=”com.mystudio.mygame”>
    <using package=”std.node”/>
</templates>
```

view

Views allow different users to have a different perspective on the same asset. A template can have multiple views.



Attributes

Attribute	Required	Description	Values
name	✓	The name of the view. This name is case sensitive and must be unique within the template.	String containing alphanumeric characters only.
template	✓	The name of the template the view is based on. The template definition must precede the view definition.	String containing alphanumeric characters or periods.

Example

```
<templates package="com.mystudio.mygame">
    <view name="Physics" template="Vehicle">
        <group name="PhysicalProperties">
            <property name="Weight"/>
        </group>
        <property name="Color"/>
    </view>
</templates>
```