

MusiXML - XML music notation file format

1. [Why XML Schema?](#)
2. [The MusiXML DTD](#)
3. [The MusiXML Schema](#)
4. [The top level structure of MusiXML](#)

1. Why XML Schema?

When we define a new data format, there are some tasks that have to be solved again and again:

- * Which character set do we use?
- * How do we handle i18n?
- * How do we define data types across programming languages?
- * How do we express hierarchies?
- * Which mechanism do we use to make the format extensible?
- * We have to define and agree in a formal language in which we express the format definition (and we have to learn it).
- * How do we express references/pointers?

This is much work to do in each case and I didn't even mention the concrete problem to be solved.

XML 1.0 solves almost all of these general problems. You probably haven't even to learn the XML syntax because it's a simplified syntax you know from HTML. The additional benefit is that the formal definition is stored in a language independant way in a separate document (the DTD). A programmer who wants to use the format doesn't even have to implement the constraints since they are tested by a parser with well defined APIs.

When programmers who work with e-commerce or music notation saw it, they immediately knew and agreed that this is what they needed. The main problem is, that XML 1.0 emerged from a (text) document oriented domain. This is the reason why there is no way to define data types, to use inheritance, define keys or reuse general modules in different formats. With XML 1.0 you have to learn a Syntax for XML and a different for DTDs.

These problems are addressed by XML Schema.

2. The MusiXML DTD

In spring 1998 I started to develop the MusiXML *DTD* ([view the MusiXML DTD in your browser](#)) with the experience of developing an unpublished binary format and the knowledge of some [more notation formats](#) . Here is an [example](#) that uses the MusiXML DTD.

3. The MusiXML Schema

The MusiXML DTD works fine, but there are many constraints that are only checked by the application. As explained in [Why XML Schema?](#), this is a problem when you want others to understand and implement these constraints.

[MusiXML Schema](#) is fully compliant with [XML Schema](#). The [example](#) and the [MusiXML Schema](#) are part of my [unit tests](#), using [Xerces2-J](#).

You can look at a [simplified browser fiendly view](#) of MusiXML to get an overview. (I know that the stylesheet that generates the overview needs some improvement :-). Anyway it helps.

4. The top level structure of MusiXML

One main goal is to store each data only once instead of holding them consistent. And we consider separating content from style. Assume we represent musical sheets for a symphony orchestra. To see the problem, we need only the score and the violin 1 part. Using a hierarchical representation, the score looks like this:

```
work -> page -> system -> staff -> measure -> <content> and the violin 1
sheets look like this: work -> page -> system -> staff -> measure ->
<content>
```

Both have the same structure. They have different instances of page, system and staff, but they share parts of <content>. The structure of the graphical hierarchy makes it necessary to store two copies of <content>. The simple idea is to store <content> in a separate place, the logical domain, and to refer to it. Now our structure looks like this:

```
work -> page -> system -> staff -> measure -> reference to part of <content>
work -> page -> system -> staff -> measure -> reference to part of <content>
```

But the work embeds everything, so we change the structure to

```
<work> <body> <content> </body> <filter> <extract> (rendering information for
score with reference to <content>) </extract> <extract> (rendering
information for violin 1 with reference to <content>) </extract> </filter>
</work>
```

Since <extract> contains declarative instructions, how to process <content>, I saved some hierarchies there, to make it easier to process. The instructions in <extract> have to be declarative and not something like do this and then do that, since procesing goes in both directions: <content> has to be

rendered using these instructions and <content> has to be changed using the instructions if the user changes something on the screen. The logical domain <body> contains almost all the the knowledge a music notation program can have about music. It's about the same information that is in the linear input mode of some music notation programs.

Then we have a graphical domain that is contained in the <filter> element. It contains <extract> elements. Each <extract> element defines a different printout: the score and each part that is to be printed (In relational databases we would use the term view insted of extract, but we need the term view in the object oriented context).