Journées Perl 2004

- **Agenda**
- **XML**
- **DOM / SAX**
- **Flux + MI + Handle**
- **XML::Parser**
- **XML::Parser, reloaded**
- **XML::SAX**
- **Conclusion**
- **Annexes**

XML[1]

- **XML est un méta-language à balises imbriquées**
  - XML
    - E**X**tended **M**arkup **L**anguage
    - Version *simplifiée* et *normalisée* du SGML
  - Méta-language
    - Défini - par exemple - au travers d'une DTD (**D**ocument **T**ype **D**efinition)
    - Permet une *validation* du document
  - Balises imbriquées
    - <BALISE attribut='valeur'>contenu</BALISE>
    - Le contenu peut être constitué par d'autres balises
    - Ou des données utilisateur
    - Et ainsi de suite de façon hiérarchique

XML²

- **Un document XML doit être bien formé et valide**
  - Bien formé
    - Respecte la syntaxe de la recommandation W3C
    - Déclaration d'en-tête correcte (e.g., <?xml version='1.0' standalone='no'?>)
    - Une balise ouvrante correspond à une balise fermante
    - Pas de recouvrement entre balises
    - Valeurs d'attributs entre guillemets ou cotes
    - Les caractères spéciaux hors balise sont remplacés par leur équivalent &;
  - Valide
    - On vérifie que le document est bien conforme à sa DTD
    - Le validateur peut être un outil externe (i.e., rxp)
    - Mais ce n'est pas exhaustif
    - Utiliser sinon “XML Schema”, plus puissant mais aussi plus complexe

DOM / SAX

- **2 approches principales, radicalement différentes**
  - DOM : représentation “spatiale” / “pull”
    - **D**ocument **O**bjet **M**odel
    - Avaler tout le document (crée une *arborescence*)
    - Se promener sur l'arbre à l'aide d'une *API*
    - Fournit un accès *aléatoire* à un MI “privé” représentant le document
  - SAX : représentation “temporelle” / “push”
    - **S**imple **A**PI for **X**ML
    - Travailler sur un *flux*
    - Gérer des *évènements* à l'aide de “callbacks”
    - Construit un MI “maison” à partir du document *sérialisé*
- **Nous allons étudier plus particulièrement SAX**

Flux + MI + Handle[1]

- **Le document devient un flux d'évènements**
  - Début de balise (+ définition d'attributs)
  - Charactères
  - Fin de balise
- **Déclenchant des fonctions (“callbacks”)**
- **Utilisées pour peupler le modèle d'information**
  - En insérant des données *incidentes* à l'endroit *courant* dans le MI
  - Référencé par une *handle* (la dernière empilée dans le contexte)
  - Pile de contexte mise à jour au fur et à mesure des évènements
  - Les attributs et charactères sont fournis par les parsers
  - Sous forme de hash ou string passées en paramètres aux callbacks

Flux + MI + Handle^2

- **XML**

```
<foo nom='fou'>
  <bar>fred</bar>
</foo>
```

- **Notes**
  - On perd l'ordre (HoH, !LoL)
  - On doit gérer les collisions
  - FILS à la racine dû à META
  - META = pile de handles
  - Hiérarchie ~ à l'identique

- **MI**

```
$id = {
  'FILS' => {
    'foo' => {          # $h1
      'ATTR' => {
        'nom' => 'fou'
      },
      'FILS' => {
        'bar' => {      # $h2
          'CHAR' => 'fred'
        }
      }
    }
  }
};
```

Flux + MI + Handle^3

- **Flux d'évènements**
  - Start (foo, %a)
  - Char(“”)
  - Start (bar)
  - Char (“fred”)
  - End (bar)
  - End (foo)
- **Etats & actions**
  - [].....[$h1]
    peupler $h1->{ATTR}
  - [$h1].....[$h1]
    peupler $h1->{CHAR}
  - [$h1].....[$h1, $h2]
    peupler $h2->{ATTR}
  - [$h1, $h2].....[$h1, $h2]
    peupler $h2->{CHAR}
  - [$h1, $h2].....[$h1]
    remonter
  - [$h1]..... []
    remonter

XML::Parser[1]

- **Référencement des callbacks**
  - `my $p = new XML::Parser ( Handlers => {`
    `Start => \&HStart, Char => \&HChar, End => \&HEnd } );`
- **Paramètres des callbacks**
  - `sub HStart { my( $expat, $element, %attribute ) = @_ }`
    `sub HChar  { my( $expat, $string              ) = @_ }`
    `sub HEnd   { my( $expat                       ) = @_ }`
- **Mise à jour du MI et de la handle par effet de bord**
  - Depuis les callbacks (pas très propre, n'est-ce pas ?)
  - La handle (dernier élément de la pile) pointe le noeud courant
  - On descend sur “start” et on remonte sur “end”

XML::Parser^2

```perl
#!/usr/bin/perl -w
use strict;

use XML::Parser;
my $parser = new XML::Parser(
  Handlers => {
    Start => \&HStart,
    Char  => \&HChar,
    End   => \&HEnd,
  }
);

our $id;
$id->{META} = [];
$parser->parsefile( 'foo.xml' );
delete $id->{META};

use Data::Dumper;
$Data::Dumper::Indent   = 1;
$Data::Dumper::Sortkeys = 1;
print Data::Dumper->Dump( [ $id ], [ 'id' ] );
```

```perl
sub HStart {
  my( $expat, $element, %attribute ) = @_;
  our $id;

  my $h = handle($id)->{FILS}->{$element} = {};
  while ( my( $k, $v ) = each %attribute ) {
    $h->{ATTR}->{$k} = $v;
  }
  push( @{$id->{META}}, $h );
}

sub HChar {
  my( $expat, $string ) = @_;
  our $id;

  unless ( $string =~ /\s+$/ ) {
    handle($id)->{CHAR} = $string;
  }
}

sub HEnd {
  my( $expat, $element ) = @_;
  our $id;

  pop @{$id->{META}};
}

sub handle {
  my( $id ) = @_;

  @{$id->{META}} ? $id->{META}->[-1] : $id;
}
```

XML::Parser, reloaded[1]

- **Le MI doit être un des paramètres des callbacks**
  - Afin de ne plus utiliser de référence “globale” comme MI
  - Ou d'employer le MI de $expat
- **Utilisation d'une closure**
  -
```
Start => sub { Hstart ( $id, @_ ) }
Char  => sub { HChar  ( $id, @_ ) }
End   => sub { HEnd   ( $id, @_ ) }
```
- **Mais ce n'est pas suffisant**
  - L'API n'est pas normalisée, donc dépendante du back-end
  - On ne peut pas facilement faire de pipelines
  - Le back-end (<eXpat/>) ne permet pas la validation (à dessein)

XML::Parser, reloaded²

```perl
#!/usr/bin/perl -w
use strict;

my $id;

use XML::Parser;
my $parser = new XML::Parser(
  Handlers => {
    Start => sub { HStart ( $id, @_ ) },
    Char  => sub { HChar  ( $id, @_ ) },
    End   => sub { HEnd   ( $id, @_ ) },
  }
);

$id->{META} = [];
$parser->parsefile( 'foo.xml' );
delete $id->{META};

use Data::Dumper;
$Data::Dumper::Indent   = 1;
$Data::Dumper::Sortkeys = 1;
print Data::Dumper->Dump( [ $id ], [ 'id' ] );
```

```perl
sub HStart {
  my( $id, $expat, $element, %attribute ) = @_;

  my $h = handle($id)->{FILS}->{$element} = {};
  while ( my( $k, $v ) = each %attribute ) {
    $h->{ATTR}->{$k} = $v;
  }
  push( @{$id->{META}}, $h );
}

sub HChar {
  my( $id, $expat, $string ) = @_;

  unless ( $string =~ /\s+$/ ) {
    handle($id)->{CHAR} = $string;
  }
}

sub HEnd {
  my( $id, $expat, $element ) = @_;

  pop @{$id->{META}};
}

sub handle {
  my( $id ) = @_;

  @{$id->{META}} ? $id->{META}->[-1] : $id;
}
```

XML::SAX[1]

- **Décorrélation du front-end et du back-end**
  - Le front-end est constitué de méthodes d'une classe (e.g., start_element, character, end_element, etc.)
  - Le back-end est au choix : PurePerl, <eXpat/>, libxml2, etc.
- **Approche orientée objet**
  - Le MI est maintenant celui de l'objet appelant la méthode
  - Permet de définir des constructeurs et destructeurs adaptés
  - Et bien d'autres méthodes privées
- **Permet de construire des pipelines**
  - `use XML::SAX::Machines`

XML::SAX^2

```perl
#!/usr/bin/perl -w
use strict;

use XML::SAX;
use MySAXHandler;
my $parser = XML::SAX::ParserFactory->parser(
  Handler => my $id = MySAXHandler->new );

open( my $fh, 'foo.xml' );
$parser->parsefile( $fh );
close $fh;

use Data::Dumper;
$Data::Dumper::Indent   = 1;
$Data::Dumper::Sortkeys = 1;
print Data::Dumper->Dump( [ $id ], [ 'id' ] );

package MySAXHandler;
use base ( 'XML::SAX::Base' );

sub new { bless {} }

sub handle {
  my( $self ) = @_;
  @{$self->{META}} ?
    $self->{META}->[-1] : $self;
}

sub xml_decl {}
```

```perl
sub start_document {
  my( $self ) = @_; $self->{META} = [];
}

sub end_document {
  my( $self ) = @_; delete $self->{META};
}

sub start_element {
  my( $self, $hash ) = @_;
  my $h = $self->handle->{FILS}->{$hash->{Name}} = {};
  for my $v ( values %{$hash->{Attributes}} ) {
    $h->{ATTR}->{$v->{Name}} = $v->{Value};
  }
  push( @{$self->{META}}, $h );
}

sub characters {
  my( $self, $hash ) = @_;
  unless ( $hash->{Data} =~ /\s+$/ ) {
    $self->handle->{CHAR} = $hash->{Data};
  }
}

sub end_element {
  my( $self, $hash ) = @_;
  pop @{$self->{META}};
}

1;
```

Conclusion

- **Parser du XML avec SAX, ce n'est pas compliqué !**
- **Avantages**
  - Normalisé et reconnu par de multiples languages
  - Faible consommation de mémoire
  - Un front-end unique mais plusieurs back-end
  - Permet de filtrer les données à la volée
- **Inconvénients**
  - Pas d'accès aléatoire
  - Pas de requêtes complexes “à la” XPath
  - Pas d'accès à la DTD

Annexes[1]

- **Pourquoi un MI “maison” ?**
  - Compatibilité avec le format de l'ère pré-XML
  - Un nouveau parser est nécessaire
  - L'API est en effet conservée
    - Get / set, itérateurs, etc.
    - Vérifications qualitatives et quantitatives du MI
    - Translation dans d'autres formats propriétaires
  - Qualification du nouveau parser uniquement
- **Utiliser du DOM à la place ?**
  - Réecriture de l'API à l'aide de celle de DOM
  - Requalification de toute l'API

Annexes²

Un exemple d'utilisation de XML::Writer

```
use IO::File;
$fh = IO::File->new( '> foo2.xml' );
use XML::Writer;
my $writer = XML::Writer->new( OUTPUT => $fh, DATA_MODE => 1, DATA_INDENT => 2 );

BuildNode( $writer, $id->{FILS} );

$writer->end();
close $fh;

sub BuildNode {
  my( $writer, $node ) = @_;

  for my $e ( keys %$node ) {
    my $eh = $node->{$e};

    $writer->startTag( $e, map { ( $_, $eh->{ATTR}->{$_} ) } keys %{$eh->{ATTR}} );

    if ( $eh->{CHAR} ) {
      $writer->characters( $eh->{CHAR} );
    }
    if ( $eh->{FILS} ) {
      BuildNode( $writer, $eh->{FILS} );
    }
    $writer->endTag( $e );
  }
}
```

Annexes[3]

- **Liens utiles (non exhaustif)**
  - XML.....http://www.w3.org/XML/
    .....http://www.xml.com/
  - Perl & XML.....http://xml.com/pub/q/perlxml
  - SAX.....http://www.saxproject.org/
  - XML Schemas.....http://www.w3schools.com/schema/
  - XSLT / Xpath.....http://www.w3.org/Style/XSL
  - <eXpat/>.....http://expat.sourceforge.net/
  - Gnome libxml2.....http://xmlsoft.org/
  - RXP.....http://www.cogsci.ed.ac.uk/~richard/rxp.html
  - CPAN.....http://search.cpan.org