User-Agent

Philippe PRADOS

site@philippe.prados.name

Préservez l'environnement,

n'imprimez pas ce document

Avant-propos

Nous allons décrire pas à pas, comment proposer un langage de développement simplifié. Nous étudierons le modèle objet, et nous proposerons une syntaxe au format XML. Nous utiliserons plusieurs techniques pour optimiser le code. Le langage que nous allons réaliser permet de simplifier l'analyse des différentes versions des navigateurs, et permet d'intégrer rapidement de nouvelles versions.

Proposer un site Internet qui fonctionne pour toutes les versions des navigateurs est un chalenge ambitieux. Chaque version possède ses particularités. Par exemple, définir la largeur d'un champ texte pour accueillir tous les caractères voulus, n'est pas chose aisée. Une taille conforme sous IE Windows, n'est pas valide sous IE Macintosh ou Mozilla Firefox sous Unix.

De nouvelles versions sortent régulièrement. Il faut pouvoir adapter l'application aux évolutions, sans devoir revoir toute l'application. Nous vous proposons de rédiger un script en XML, permettant d'adapter l'application aux différentes versions des navigateurs. Cela sera l'occasion d'étudier la modélisation d'un langage de développement simple, d'utiliser un analyseur XML pour le compiler, et d'exploiter le polymorphisme des langages objets pour exécuter le scénario.

Voici un extrait d'un script que nous souhaitons pouvoir exécuter.

```
<if match="^([Ww]in|Microsoft Internet Explorer)">
  <set key="familly.os" value="Windows"/>
</if>
```

Chaque navigateur s'identifie auprès du serveur par l'en-tête `User-Agent`. Celui-ci peut indiquer de nombreuses informations comme la version du navigateur, la version qu'il simule, les extensions qu'il possède, la langue utilisée, etc.

Voici par exemple, les chaînes retournées par certaines versions de Internet Explorer et Firefox sous Windows

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.5) Gecko/2004 1107 Firefox/0.10.1
```

Vous trouverez d'autres possibilités ici : http://www.psychedelix.com/agents.html

La première chose à faire est d'analyser le format de l'en-tête `User-Agent`, émis par les navigateurs. Celui-ci respecte une syntaxe ouverte, composée de trois champs principaux. La première partie indique le nom de l'agent. Pour des raisons historique et de compatibilité, la chaîne de caractère commence généralement par `Mozilla` suivit éventuellement d'un numéro de version. Cela correspond au nom de code de la toute première version du navigateur de Netscape. Ensuite, entre parenthèse, de nombreuses précisions sont indiquées, séparées par des points-virgules. Après la parenthèse fermante, on trouve également des informations complémentaires.

Nous allons découper ces chaînes en champs. Le premier correspond à la première partie avant la parenthèse. Ensuite, nous trouvons les différentes valeurs entre les paramètres, à raison d'un champ par point virgule. La dernière partie constitue également un champ.

```
ArrayList items = new ArrayList(7);
int idx = userAgent.indexOf('(');
if (idx != -1)
{
  items.add(userAgent.substring(0, idx - 1));
  for (StringTokenizer tokens = new StringTokenizer(userAgent
       .substring(idx + 1), ";)");
        tokens.hasMoreTokens();)
    items.add(tokens.nextToken().trim());
}
else
  items.add(userAgent);
```

Ainsi, il devient plus facile d'analyser les champs. Des expressions régulières vont nous permettrent de rechercher des valeurs dans les champs, afin de valoriser des variables.

Nous souhaitons permettre la rédaction d'un script d'analyse à l'aide d'un langage XML. Celui-ci doit nous permettre d'exprimer l'algorithme à utiliser sur l'en-tête `User-Agent`, afin d'identifier les valeurs à indiquer pour différentes variables. Celles-ci seront exploitable par l'application pour adapter le comportant suivant les versions. Elles pourront valoriser les tailles des champs, les polices de caractères à utiliser, les modifications des scripts, les largueurs des colonnes, etc.

Nous choisissons de rédiger un langage impératif. C'est-à-dire que celui-ci nous permet d'exprimer des actions à effectuer, et non des règles à analyser. Java et un langage impératif. Nous avons besoins de deux structures principales : des instructions et des expressions. Les instructions permettent d'indiquer une action à effectuer. Une expression permet d'exprimer un calcul, dont le résultat aura une influence sur l'instruction à exécuter.

Une interface `Statement` permet de porter les différentes instructions. Elle propose une méthode `execute()`. Celle-ci attend deux paramètres : une liste de champs et une liste de propriétés pour y stocker les résultats du traitement.

```
/**
 * A statement.
 *
 * @since 1.0
 * @version 1.0
 * @author: <a href="mailto:securikit@philippe.prados.name">Philippe Prados </a>
 */
interface Statement
{
  /**
   * Execute the statement.
```

```
   *
   * @param items The items presents in the user agent.
   * @param prop The properties to set.
   *
   * @since 1.0
   *
   * @pre items!=null
   * @pre prop!=null
   */
  public void execute(AbstractCollection items, Properties prop);
}
```

Nous utilisons un paramètre de type `AbstractCollection` afin de pouvoir accueillir tout type de conteneur. Il est préférable d'utiliser le type le plus générique possible, afin de laisser plus de libertés au développeur. Les variables à valoriser lors de l'analyse sont mémorisés dans un objet `Properties` sous forme de caractère. Cette classe sert à cela. Elle propose des méthodes d'accès au format caractère et n'a pas l'inconvénient d'un `RessourceBundle`. En effet, il n'est pas nécessaire d'exploiter inutilement les capacités multi langues du `RessourceBundle`.

Quelles sont les instructions dont nous avons besoins ? Il nous faut une instruction se chargeant d'exécuter une liste d'instructions. Cela correspond aux accolades des langages comme Java ou C++. La classe `Body` représente un bloc d'instruction. Elle possède un tableau d'instruction qu'elle se propose d'invoquer dans l'ordre.

```
class Body implements Statement
{
  private Statement[] body_;

  Body(AbstractCollection body)
  {
    body_ = new Statement[body.size()];
    body.toArray(body_);
  }

  public void execute(AbstractCollection items, Properties prop)
  {
    final int s = body_.length;
    for (int i = 0; i < s; ++i)
    {
      body_[i].execute(
        items, prop);
    }
  }
}
```

Après la compilation du programme, une seule instance `Body` est nécessaire. Celle-ci possède alors toutes les autres instructions et expressions du programme. Compiler un programme consiste à construire un arbre d'objet, afin d'obtenir une instance `Body` racine.

Nous avons également besoin d'une instruction `SetProperty` pour valoriser les fameuses variables, objet de l'exécution du programme. La classe `SetProperty` s'occupe de cela. Le constructeur attend une clef et une valeur. L'exécution de cette instruction ne fait qu'ajouter la valeur indiquée sous la clef, dans l'instance `Properties`.

```
class SetProperty implements Statement
{
  private String key_;
  private String value_;

  SetProperty(String key, String value)
  {
    key_ = key;
    value_ = value;
  }

  public void execute(AbstractCollection items, Properties prop)
  {
    prop.put(key_, value_);
  }
}
```

Si le langage ne propose que la valorisation de variable, il ne peut pas faire grand-chose. Il nous manque une instruction de comparaison. Un `If`. Cette instruction regroupe une expression et une instruction à exécuter si celle-ci retourne `true`. La classe `If` exprime cela.

```
class If implements Statement
{
  private Expression expr_;
  private Statement body_;

  If(Expression expr, Statement body)
  {
    expr_ = expr;
    body_ = body;
```

```
  }

  public void execute(AbstractCollection items, Properties prop)
  {
    if (expr_.match(items))
      body_.execute(items, prop);
  }
}
```

La simplicité de notre langage n'exige pas de structures plus complexes comme les boucles, les déclarations de fonctions ou de procédures, de variables, etc. La même approche serait utilisée, en implémentant l'interface `Statement`.

Il nous faut maintenant être capable de proposer des expressions logiques. L'interface `Expression` propose une méthode `match()` retournant un booléen. En effet, dans le cas qui nous concerne, seuls les expressions booléenne nous importent. Nous souhaitons savoir si l'en-tête `User-Agent` correspond à un modèle particulier. Si c'est le cas, nous valorisons des variables.

```
/**
 * An expression.
 *
 * @since 1.0
 * @version 1.0
 * @author: <a href="mailto:securikit@philippe.prados.name">Philippe Prados </a>
 */
interface Expression
{
  /**
   * Check if the items match the expression.
   *
   * @param items The items presents in the user agent.
   * @return true if an items match the expression.
   *
   * @since 1.0
   *
   * @pre items!=null
   */
  public boolean match(AbstractCollection items);
}
```

L'expression la plus importante pour ce langage consiste à rechercher un modèle dans les champs de l'en-tête `User-Agent` à l'aide d'une expression régulière. La classe `MatchRE` implémente l'interface `Expression` pour cela.

```
class MatchRE implements Expression
{
  private Pattern re_;

  MatchRE(Pattern re)
  {
    re_ = re;
  }

  public boolean match(AbstractCollection items)
  {

    for (Iterator i = items.iterator(); i.hasNext();)
    {
      if (re_.matcher((String) i.next()).find())
        return true;
    }
    return false;
  }
}
```

Ainsi, une instruction `If` peut rechercher un modèle de chaîne de caractère, et valoriser ainsi des propriétés à l'aide d'instructions `SetProperty`.

Les expressions sont booléennes. Nous pouvons alors offrir des expressions logiques afin de faciliter la rédaction d'expressions complexes. La classe `Not` permet d'inverser le sens du test. Il est difficile de faire plus simple. Elle attend une autre expression pour inverser le résultat.

```
class Not implements Expression
{
  private Expression expr_;

  Not(Expression expr)
  {
    expr_ = expr;
  }

  public boolean match(AbstractCollection items)
  {
```

```
    return !expr_.match(items);
  }
}
```

Les classes `Or` et `And` permettent de combiner les expressions booléennes. Pour des raisons d'optimisations, les instances mémorisent des tableaux d'expressions et non un simple couple d'expression.

```
class Or implements Expression
{
  private Expression[] nodes_;

  Or(AbstractCollection nodes)
  {
    nodes_ = new Expression[nodes.size()];
    nodes.toArray(nodes_);
  }

  public boolean match(AbstractCollection items)
  {
    for (int i = nodes_.length - 1; i >= 0; --i)
      if (nodes_[i].match(items))
        return true;
    return false;
  }
}

class And implements Expression
{
  private Expression[] nodes_;

  And(AbstractCollection nodes)
  {
    nodes_ = new Expression[nodes.size()];
    nodes.toArray(nodes_);
  }

  public boolean match(AbstractCollection items)
  {
    for (int i = nodes_.length - 1; i >= 0; --i)
      if (!nodes_[i].match(items))
        return false;
    return true;
  }
}
```

Notez les différences entre les deux méthodes `match()`. Dans la version de la classe `Or`, il suffit d'avoir une expression valide pour interrompre l'analyse. Dans la version de la classe `And`, c'est l'inverse. Il faut avoir une expression non valide pour interrompre l'analyse.

Il nous reste à proposer une classe permettant de compiler un fichier XML pour produire des instances `Statement` et `Expression`. Ensuite, lors de l'analyse d'un en-tête `User-Agent`, il suffit d'invoquer le moteur avec les champs extrait.

```
result = new Properties();
engine_.execute(items, result);
```

Le modèle objet est représenté Figure 1.

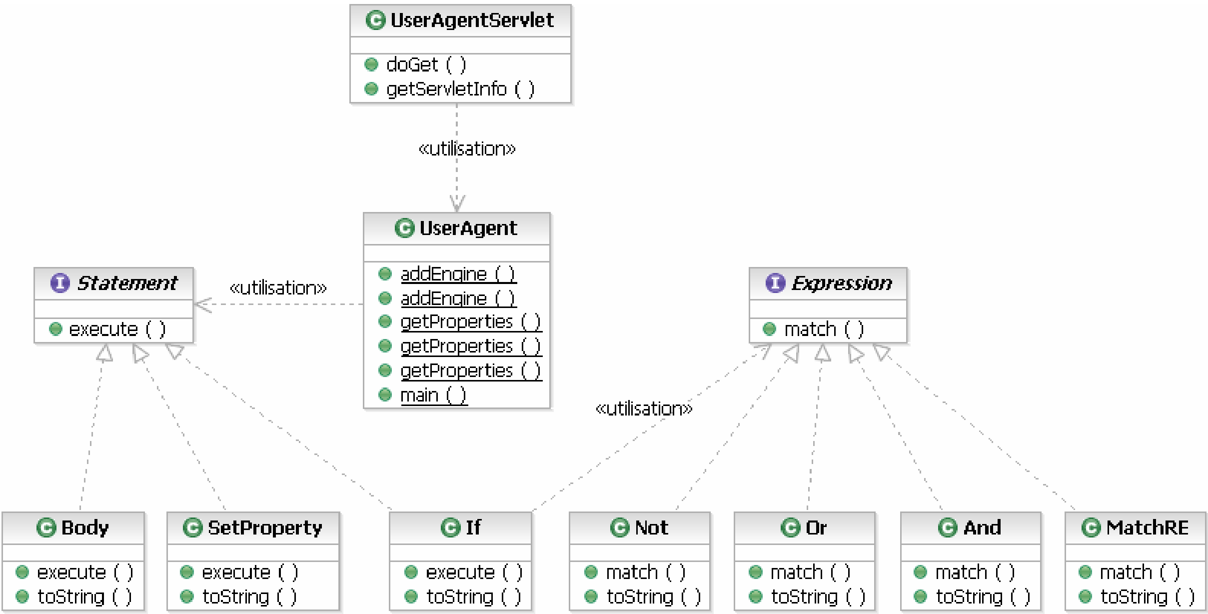

Figure 1

La classe `UserAgent` se charge d'analyser tous cela et d'exécuter le script. Nous devons proposer une syntaxe XML pour décrire les scénarios. Il est facile de décrire un marqueur par classe. Pour des raisons de simplicité syntaxique, certains attributs généreront des objets, et des classes n'auront pas de syntaxe explicite.

Nous avons choisi d'utiliser XML, car les analyseurs sont disponibles et il est aisé de décrire la syntaxe du langage. Nous avons choisi d'exploiter les fonctionnalités des schémas (http://www.w3.org/XML/Schema) car ils sont plus riches que les DTD.

Nous allons commencer par déclarer les expressions. Chacune peut contenir une autre expression. Cela est exprimé dans le schéma de la syntaxe XML comme ceci :

```
<xsd:element name="match" type="xsd:string" />

<xsd:element name="and">
  <xsd:complexType>
    <xsd:choice maxOccurs="unbounded" minOccurs="2">
      <xsd:element ref="match" />
      <xsd:element ref="and" />
      <xsd:element ref="or" />
      <xsd:element ref="not" />
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

<xsd:element name="or">
  <xsd:complexType>
    <xsd:choice maxOccurs="unbounded" minOccurs="2">
      <xsd:element ref="match" />
      <xsd:element ref="and" />
      <xsd:element ref="or" />
      <xsd:element ref="not" />
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

<xsd:element name="not">
  <xsd:complexType>
    <xsd:choice maxOccurs="1" minOccurs="0">
      <xsd:element ref="match" />
      <xsd:element ref="and" />
      <xsd:element ref="or" />
    </xsd:choice>
    <xsd:attribute name="match" type="xsd:string" use="optional" />
  </xsd:complexType>
</xsd:element>
```

Ces syntaxes permettent de rédiger un code comme celui-ci :

```
<if>
  <and>
```

```
    <match>Googlebot</match>
    <not match="Test"/>
  </and>
  <set key="familly.agent" value="Google"/>
</if>
```

Le marqueur `<not/>` propose un attribut `match` afin de simplifier la syntaxe. Une instance `MatchRE` est alors construire dans une instance `Not`. Ainsi, les deux écritures suivantes sont identiques en mémoires.

```
<not>
  <match>[Ww]indows</match>
</not>

<not match="[Ww]indows"/>
```

Il nous faut maintenant s'occuper des instructions. La première syntaxe qui nous intéresse permet de construire une instance `SetProperty`. Nous avons besoins de deux attributs : `key` et `value`. Cela doit être décrit dans un schéma comme ceci :

```
<xsd:element name="set">
  <xsd:complexType>
    <xsd:attribute name="key" type="xsd:normalizedString" use="required" />
    <xsd:attribute name="value" type="xsd:normalizedString" use="required" />
  </xsd:complexType>
</xsd:element>
```

Ainsi, le développeur peut rédiger une expression simplement.

```
<set key="familly.agent" value="MSIE"/>
```

Il nous faut également une syntaxe pour l'instruction `If`.

```
<xsd:element name="if">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:choice maxOccurs="1" minOccurs="0">
        <xsd:element ref="match" />
        <xsd:element ref="and" />
        <xsd:element ref="or" />
        <xsd:element ref="not" />
      </xsd:choice>
      <xsd:choice maxOccurs="unbounded" minOccurs="1">
        <xsd:element ref="if" />
        <xsd:element ref="set" />
      </xsd:choice>
    </xsd:sequence>
    <xsd:attribute name="match" type="xsd:string" use="optional" />
    <xsd:attribute name="notmatch" type="xsd:string" use="optional" />
  </xsd:complexType>
</xsd:element>
```

On retrouve les quatre syntaxes pour les expressions. Deux attributs sont présents pour simplifier la syntaxe : `match` et `notmatch`. Cela génère les instances correspondantes lors de la compilation.

Il n'existe pas de marqueur pour la classe `Body`. Celle-ci est implicite. Lorsque des marqueurs d'instructions sont présents dans d'autres marqueurs d'instructions, une instance `Body` est éventuellement générée pour les mémoriser.

Tous les fichiers XML doivent avoir un marqueur racine. Nous avons choisi de l'intituler `<engine/>`. Il agrège des instructions `<if/>` et `<set/>`.

```
<xsd:element name="engine">
  <xsd:complexType>
    <xsd:choice maxOccurs="unbounded" minOccurs="1">
      <xsd:element ref="if" />
      <xsd:element ref="set" />
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

La syntaxe étant maintenant décrite, il faut compiler un fichier XML pour générer les instances du modèle objet. Deux approches sont possibles : utiliser un analyseur de type SAX ou utiliser un analyseur de type DOM. Le premier émet des évènements lors de la rencontre de chaque marqueur, le deuxième génère un arbre équivalent au fichier XML. Comme le fichier n'est analysé qu'une seule fois, nous avons choisi d'utiliser un analyser DOM, plus simple à manipuler. Le fichier est converti en arbre DOM, puis l'arbre est analysé pour générer les instances qui nous intéressent.

Nous souhaitons bénéficier des vérifications de syntaxes proposées par les analyseurs XML. Pour cela, il faut que le fichier `useragent.xsd` possédant la syntaxe, soit disponible, quel que soit le contexte d'exécution de notre code. Nous pouvons le placer dans un fichier sur le disque, mais alors, il faut pouvoir indiquer son emplacement avec un paramètre de déploiement. Nous pouvons le placer sur Internet, mais l'application doit alors pouvoir sortir du réseau local pour le récupérer. Les pares feux peuvent empêcher cela. La bonne démarche consiste à placer ce fichier dans l'archive java de l'application.

La ressource est récupéré à l'aide d'une méthode getResourceAsStream() afin d'être indépendant du déploiement. Le fichier peut être présent dans un répertoire, une archive, une archive dans une autre archive au format WAR, etc. C'est le chargeur de classe qui se charge d'obtenir le fichier.

```
DocumentBuilder builder = DOMfactory.newDocumentBuilder();
DOMfactory.setSchema(SchemaFactory.newInstance(
  XMLConstants.W3C_XML_SCHEMA_NS_URI).newSchema(
    UserAgent.class.getResource("useragent.xsd")));
Document root = builder.parse(xmlSource);
```

Le code complet est présent sur mon site référencé plus bas.

Nous souhaitons pouvoir utiliser plusieurs fichiers de scripts pour produire qu'un seul moteur. Pourquoi cela ? Car les applications importantes sont construites par plusieurs équipes. Chacune à des besoins spécifiques d'adaptation suivant les navigateurs. Chaque sous projets doit pouvoir proposer son script de paramétrage. L'union des scripts permet de produire une liste complète de variable. La classe UserAgent propose des méthodes addEngine() pour permettre d'ajouter différents scripts. Par défaut, si aucun n'est renseigné, les ressources /useragent.xml sont utilisées. Il peut y en avoir simultanément dans plusieurs archives. Elles seront toutes utilisées.

```
if (engine_ == null)
{
  for (Enumeration e=UserAgent.class.getClassLoader().getResources("useragent.xml");
       e.hasMoreElements();)
  {
    URL url =(URL)e.nextElement();
    if (url != null)
      addEngine(url);
  }
}
```

Ainsi, chaque sous projet peut proposer son fichier et le placer dans une archive qui lui est propre.

Nous pouvons maintenant proposer un fichier d'analyse du champs User-Agent, afin d'identifier sa famille ainsi que le système d'exploitation utilisé par l'internaute (Figure 2).

```
<engine xmlns="http://www.philippe.prados.name/securikit/1.0/UserAgent"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
          "http://www.philippe.prados.name/securikit/1.0/UserAgent
           http://www.philippe.prados.name/securikit/1.0/UserAgent/useragent.xsd"
>
  <!-- default values -->
  <set key="familly.agent" value="unknown"/>
  <set key="familly.os" value="unknown"/>
  <!-- Check OS familly -->
  <if match="^([Ww]in|Microsoft Internet Explorer)">
    <set key="familly.os" value="Windows"/>
    <if match="(win16|windows 3\.)">
      <set key="familly.os.sub" value="3.x"/>
    </if>
    <if match="[Ww]indows 95">
      <set key="familly.os.sub" value="95"/>
    </if>
    <if match="[Ww]indows 98">
      <set key="familly.os.sub" value="98"/>
    </if>
    <if match="[Ww]in 9x">
      <set key="familly.os.sub" value="Me"/>
    </if>
    <if match="[Ww]indows NT">
      <set key="familly.os.sub" value="NT"/>
      <if match="5\.0">
        <set key="familly.os.sub" value="2000"/>
      </if>
      <if match="5\.[^0]">
        <set key="familly.os.sub" value="XP"/>
      </if>
    </if>
  </if>
  <if match="^Mac(intosh)?">
    <set key="familly.os" value="Macintosh"/>
    <if match="68(k|000)">
      <set key="familly.os.sub" value="68k"/>
    </if>
    <if match="(ppc|powerpc)">
      <set key="familly.os.sub" value="ppc"/>
    </if>
  </if>
  <if match="(^X11|inux)">
    <set key="familly.os" value="Unix"/>
```

```
    <set key="familly.os.sub" value="Unknown"/>
    <if match="(sco|unix_sv)">
      <set key="familly.os.sub" value="SCO"/>
    </if>
    <if match="unix_system_v">
      <set key="familly.os.sub" value="Unixware"/>
    </if>
    <if match="ncr">
      <set key="familly.os.sub" value="MPRAS"/>
    </if>
    <if match="reliantunix">
      <set key="familly.os.sub" value="Reliant"/>
    </if>
    <if match="dec">
      <set key="familly.os.sub" value="DEC"/>
    </if>
    <if match="vax">
      <set key="familly.os.sub" value="VMS"/>
    </if>
  </if>
  <if match="sunos">
    <set key="familly.os" value="Unix"/>
    <set key="familly.os.sub" value="sun"/>
  </if>
  <if match="irix">
    <set key="familly.os" value="Unix"/>
    <set key="familly.os.sub" value="Irix"/>
  </if>
  <if match="hp-ux">
    <set key="familly.os" value="Unix"/>
    <set key="familly.os.sub" value="HP-ux"/>
  </if>
  <if match="aix">
    <set key="familly.os" value="Unix"/>
    <set key="familly.os.sub" value="Aix"/>
  </if>
  <if match="^Elaine">
    <set key="familly.os" value="palm"/>
  </if>

  <!-- check agent familly -->
  <if match="^Mozilla">
    <!-- Internet Explorer -->
    <if match="^(MSIE|Microsoft Internet Explorer)">
      <set key="familly.agent" value="MSIE"/>
    </if>

    <!-- Gecko navigator -->
    <if match="^Gecko">
      <set key="familly.agent" value="Gecko"/>
    </if>

    <!-- Opera navigator -->
    <if match="^Opera">
      <set key="familly.agent" value="Opera"/>
    </if>

    <!-- Hot Java -->
    <if match="^hotjava">
      <set key="familly.agent" value="HotJava"/>
    </if>

    <!-- AOL -->
    <if match="^Aol">
      <set key="familly.agent" value="Aol"/>
    </if>

    <!-- WebTV -->
    <if match="^webtv">
      <set key="familly.agent" value="WebTV"/>
    </if>

    <!-- Netscape navigator -->
    <if notmatch="^(MSIE|compatible|Gecko|Firefox|Opera|HotJava)">
      <set key="familly.agent" value="Netscape"/>
    </if>
```

```
  </if>
</engine>
```

Figure 2

Nous pouvons maintenant obtenir facilement les valeurs des variables dans notre application.

```
String userAgent = request.getHeader("User-Agent");
UserAgent.getProperties(userAgent).getProperty("familly.os");
```

Des versions de la méthode `getProperty()` permettent de simplifier ce code en une seule ligne.

```
UserAgent.getProperties(request).getProperty("familly.os");
```

L'exécution du moteur prend un certains temps, même s'il est très rapide. En effet, celui-ci est basé sur des expressions régulières. La recherche d'une simple chaîne de caractère prend trois fois plus de temps avec une expression régulière qu'avec une recherche linéaire. Il n'est pas possible de savoir à l'avance les différents agents rencontrés par l'application. Un cache permet alors de calculer les différentes propriétés lors de la première rencontre d'un agent. La clef du cache est bien entendu, la valeur de l'en-tête `User-Agent`. Ainsi, le calcul n'est effectué qu'une seule fois sur le serveur pour chaque agent rencontré.

Les agents sont très divers. Ils peuvent être différent d'un autre, uniquement par la modification d'un numéro de génération ou l'existence de plug-in, voir l'ordre d'installation des plug-in. Parmi les nombreuses valeurs possibles, des versions différentes des agents vont produire le même résultat en terme de propriété valorisée par les scripts. Afin d'économiser la mémoire, avant de placer le résultat dans le cache, une analyse est effectué pour vérifier qu'il n'existe pas une autre instance `Properties` ayant strictement les mêmes valeurs. Si c'est le cas, l'instance est recyclée pour être référencées par plusieurs agents.

```
for (Iterator i = cache_.values().iterator(); i.hasNext();)
{
  Properties prop = (Properties) i.next();
  if (prop.equals(result))
  {
    // Share the properties with differents user agent
    cache_.put(userAgent, prop);
    return prop;
  }
}

// Put it in the cache
cache_.put(userAgent, result);
return result;
```

Avec ces deux améliorations, nous optimisons les performances sans sacrifier la mémoire.

Nous avons maintenant la possibilité d'enrichir l'analyse des `User-Agent`, sans devoir modifier l'application. Un administrateur de bon niveau peut intégrer une nouvelle version d'un navigateur, en utilisant son éditeur XML préféré. La syntaxe est automatiquement vérifiée par l'analyse du schéma.

L'analyse s'effectue coté serveur. Et coté client ? Il serait sympathique de pouvoir bénéficier des mêmes résultats sur le client et sur le serveur à partir d'un seul script. Pour offrir cela, nous allons proposer une servlet qui se chargera de générer un javascript, avec les valeurs calculées sur le serveur.

```
String userAgent = request.getHeader("User-Agent");
response.setContentType("application/x-javascript");
Properties prop = UserAgent.getProperties(userAgent);
PrintWriter out = response.getWriter();
out.print("var UserAgent=\n" +
  "{ getProperties:function()\n" +
  "  { return {");
if (prop != null)
  for (Enumeration e = prop.keys(); e.hasMoreElements();)
  {
    String key = (String) e.nextElement();
    String value = prop.getProperty(key);
    out.print('"' + key + "\":\"" + value + '"');
    if (e.hasMoreElements())
      out.print(',');
  }
out.print(" };\n" + "   }\n" + "};\n");
```

Ce bout de code produit un javascript comme ceci :

```
var UserAgent=
{
  getProperties:function()
  {
    return
     {
       "familly.os":"Windows",
       "familly.os.sub":"XP",
       "familly.agent":"Gecko"
     };
```

```
  }
};
```

Ce code permet de simuler le code équivalent, présent sur le serveur. La page doit importer le javascript généré et utiliser l'objet UserAgent.

```
<script src="/UserAgentServlet"></script>
<script>
alert(UserAgent.getProperties()["familly.agent"]);
</script>
```

Le code produit est indépendant de l'utilisateur. Nous pouvons alors utiliser toutes les possibilités de caches du protocole http pour éviter de générer le script à chaque page, voir à chaque utilisateur.

```
final long timeout = 86400 * 1000;
boolean etag = userAgent.equals(request.getHeader("If-None-Match"));
long d = request.getDateHeader("If-Modified-Since");
if (etag && ((d != -1) && d < bootstraptime))
  response.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
else
{
  response.addHeader("Vary", "user-agent");
  response.addDateHeader("Last-Modified", System.currentTimeMillis());
  response.addDateHeader("Content-Expire", System.currentTimeMillis()
    + timeout);
  response.addHeader("Cache-Controle", "max-age " + timeout);
  response.addHeader("ETag", userAgent);
  ...
}
```

Lors de la production de la page de script, nous indiquons une date d'expiration de 1000 jours. Nous ajoutons également un en-tête ETag avec le nom de l'agent. Cet en-tête permet d'indiquer que la page est spécifique à cette version d'agent. Ainsi, les proxies utilisés par les internautes sont capable de garder en cache différentes versions de la page, en s'appuyant sur la valeur de l'agent. Si un navigateur ou un proxy demande la page avec l'en-tête If-None-Match, cela indique qu'il connaît déjà une version de la page, et souhaite éventuellement une nouvelle version. Si l'en-tête If-Modified-Since est présent, cela indique que le navigateur ou le proxy possède une ancienne version de la page, et qu'il demande si une nouvelle est présente. Si la date indiquée est antérieure à la date de lancement de l'application, une nouvelle page est générée. Sinon, la requête indique que la page présent dans les caches des navigateurs et des proxies sont toujours valides.

Imaginons quatre utilisateurs A, B, C et D. A et B utilisent la même version de navigateur 1.0 ; C la version 1.1 et D la version 2.0. Tous les utilisateurs utilisent un proxy d'entreprise pour communiquer avec Internet. Lorsque A se connecte, une page pour la version 1.0 est générée. Elle est gardée dans le cache du proxy. A peut redemander la même page. Cette fois-ci, le navigateur ajoute l'en-tête If-None-Match avec la valeur du champ ETag précédemment indiqué pour A, correspondant à la version 1.0. Cela est conforme à la version présente dans le cache du proxy. Le proxy interroge le serveur pour vérifier s'il n'est pas nécessaire d'obtenir une nouvelle version de la page. Celui-ci répond que la page présente dans le cache du proxy est toujours valide à l'aide d'un code retour NOT_MODIFIED. Le proxy peut fournir la page sans télécharger une nouvelle version complète à partir du serveur HTTP.

Puis, C se connecte par l'intermédiaire du proxy. Celui-ci constate que l'en-tête If-None-Match n'est pas présent. Il ne peut livrer la version qu'il possède dans son cache. Une requête est alors généré vers le serveur d'application pour obtenir une nouvelle version de la page. Une nouvelle valeur de ETag est produite avec la valeur 1.1. Le proxy cache la page en y associant cette valeur. A et C peuvent redemander la page, le proxy leurs livre leurs versions.

D se connecte également et une troisième version de la page est mémorisée dans le proxy sous la clef 2.0.

Puis, B se connecte. B possède la même version de navigateur que A. Comme B ne possède pas de valeur d'ETag, le proxy ne connaît pas la version valide parmi les trois présentes dans son cache. Une requête part alors vers le serveur qui livre la version 1.0, la même que lors de la première requête, produite lors de la connexion de A. Le proxy met à jour la version pour l'agent 1.0. Maintenant, A et B partagent la version présente dans le cache du proxy. Si A ou B demande à nouveau la page, le proxy peut la livrer sans interroger le serveur.

En plus de cela, un timeout évite que les navigateurs ou les proxies interrogent trop souvent le serveur. Dès que la page est présente dans le cache du navigateur ou du proxy, elle est réutilisée sans nécessiter de nouvelle connexion.

Dans cet article, nous avons décomposé les différentes étapes pour la réalisation d'un petit langage. Pour ne pas écrire d'analyseur syntaxique, nous avons utilisés une syntaxe XML que nous vérifions à l'aide d'un schéma. Nous avons proposé un modèle objet avec deux interfaces principales, afin de pouvoir bénéficier d'instructions et d'expressions dans notre langage. Puis, nous avons implémentés ces interfaces. Pour des raisons de simplification syntaxique pour le développeur, la syntaxe XML est parfois éloignée de modèle interne. Nous avons utilisés différentes techniques d'optimisations pour améliorer la consommation mémoire et les performances de notre outil. Pour permettre une cohérence entre le client et le serveur, nous avons généré dynamiquement du javascript, et nous avons exploité toutes les techniques d'optimisations du protocole http afin d'améliorer les performances.

Tous les sources sont présent sur mon site : www.philippe.prados.name. Le code est parfois plus complexe qu'exposé ici. Je vous invite à l'étudier et à l'enrichir. Faites moi part de vos améliorations !

Philippe PRADOS

IBM : Sécurité et technologies GRID

press@philippe.prados.name