Tutoriel OpenLayers/OGC
Apprendre OpenLayers par les standards OGC
(et vice versa)

Table des matières

Une première carte interactive pour comprendre. 2
Conformité OGC...2
GetCapabilities :...3
GetMap : 3
GetFeatureInfo : 4
OpenLayers.Layer.WMS :..5
Couches cartographiques non-standards.....................6
WMS: personnalisation de style.......................................6
Requête GetMap en POST..8
Requête GetMap multi-couches.................................9
Carte avec plusieurs couches (image overlays). 9
Superposition d'image overlays côté client..................9
Reprojection..10
Couche de référence dite commerciale......................10
Carte avec plusieurs couches (vector overlays)...............11
Etape 1 : création d'un "vector overlay" 11
Sélection avec un serveur conforme OGC WFS.. .12
Utilisation d'un proxy (same origin policy)..........12
Les requêtes WFS au peigne fin..........................13
GetCapabilities : 13
DescribeFeatureType : 13
GetFeature :..14
GetFeature et filtre avec OpenLayers 16
Création depuis un fichier en format standardisé . 16
Utilisation du format GeoJSON 17
Utilisation d'un encodage personnalisé................18
Mise au point sur les projections et OpenLayers...18
Etape 2 : application de style côté client....................19
Utilisation du paramètre de couche style :............19
Propriétés de style :..19
Utilisation du paramètre de couche styleMap :.....20
Utilisation d'un contexte fonctionnel :..................21
Interaction cartographique et gestion des événements.….22
Mise en place de contrôles.......................................23
Personnalisation d'interface.....................................23
Ajout de bouton personnalisé...................................24
Gestion d'événements...25
Utilisation du contrôle GetFeatureInfo......................26
Affichage d'une popup...26
Interaction sur un vector overlay..............................26
Création et stockage d'objets géographiques (DrawFeature)...28
Références supplémentaires...29

Liste des sources

```
Ex1a_wms.html
Ex1b_wms.html
Ex1c_gmaps.html
Ex1d_metacarta.html
Ex2a_internal_styling.html
   sld/giant_polygon.sld
Ex2b_external_styling.html
   sld/pinkWorld.sld
Ex2c_multilayers.html
   sld/capitals.sld
Ex2d_library_styling.html
   sld/library.sld
Ex3a_overlay_wms.html
   sld/redDotCities.sld
Ex3b_overlay_wms+suisse.html
Ex3c_overlay_gmaps+wms.html
Ex4a_overlay_wfs.html
Ex4b_overlay_wfs_filtered.html
Ex5a_overlay_GPX.html
   vector/pyrenees.gpx
   vector/grandraid1.gpx
   vector/grandraid2.gpx
Ex5b_overlay_GPX_gmaps.html
Ex5c_overlay_JSON.html
   vector/4capitals.json
Ex5d_overlay_JSON_db.html
   Ex5d_querydb_JSON.php
Ex5e_overlay_custom.html
   vector/4capitals.txt
Ex6a_overlay_style.html
Ex6b_overlay_styleMap.html
Ex6c_overlay_styleMap.html
Ex6d_overlay_styleMap.html
Ex6e_overlay_styleMap.html
Ex6f_overlay_choropleth.html
   sld/choropleth.sld
Ex6g_overlay_NordEtSud.html
Ex6h_overlay_meteo.html
   images/pluie.png
   images/soleil.png
Ex6i_overlay_readSLD.html
   sld/redDotCities.sld
Ex7a_map_controls.html
   /jslib/OpenLayers/imgogo/
   ogo.css
Ex7b_wms_featureInfoRequest.html
Ex7c_wms_featureInfoControl.html
Ex7d_overlayVector_control.html
Ex7e_overlayVector_drawFeature.html
   insertTrace.php

requests/WFS_GetFeature_filtered+BBOX.xml
requests/WFS_GetFeature_filtered.xml
requests/WMS_GetMap_externalStyle.xml
requests/WMS_GetMap_internalStyle.xml
```

Bonus:
sld/categorize.sld
geoadmin.html (GeoAdmin API)

OpenLayers (http://openlayers.org) est un *mapping framework* conforme OGC (Open Geospatial Consortium, http://www.opengeospatial.org). Ce dernier a pour objectif de définir des standards favorisant l'intéropérabilité des systèmes d'informations géographiques. L'idée de ce tutoriel est d'apprendre à utiliser OpenLayers tout en comprenant la logique d'intéropérabilité par les standards.

Une première carte interactive pour comprendre.

Entrons dans le vif du sujet avec un premier exemple, Ex1a_wms.html. Premier élément à identifier est le chargement du framework OpenLayers.

```
<script src="/jslib/OpenLayers/OpenLayers.js"></script>
```

Celui-ci est composé d'une librairie Javascript et d'un ensemble de répertoires images et de CSS formant un thème. Notez que même compactée, la librairie pèse son poids !

img	24 items
imgogo	26 items
theme	1 item
OpenLayers.js	846.6 KB

Le framework utilisé dans ce tutoriel peut être téléchargé à partir du site de cours.

Voyons la suite dans cet exemple :

- dans tous les exemples, pour faire simple, tout démarre par la fonction *init()*
- une carte est une instance OpenLayers.Map liée à un élément <div>
- une carte possède au moins une couche de référence (base layer)
- OpenLayers gère la navigation, déplacement (pan) et zoom, en organisant le dialogue avec le serveur cartographique

Petite mise au point sur la couche de référence et les codes SRS (Système de Référence Spatial) :

- le SRS (on parle aussi de système de coordonnées) dit WGS84 est utilisé pour les données à l'échelle du monde, en dégrés, les longitudes et latitudes sont simplement considérés comme X et Y du plan 2D projeté.
- sans aucune précision, OpenLayers travaille par défaut avec ce SRS dit WGS84 et sur l'emprise maximale du globe, soit -180,-90,180,90 en degrés
- dans cet exemple, aucune précision n'est donnée à ce sujet, c'est donc tout en WGS84
- lorsqu'il s'agira de préciser un système différent, OpenLayers fait usage des codes dits EPSG définis par l'European Petroleum Survey Group. Quelques codes à enregistrer :
  - EPSG:4326 = SRS WGS84 (système en degrés longitude et latitude)
  - EPSG:21781 = SRS Suisse (projection cartographique, en mètres)
  - EPSG:27572 = SRS France (projection cartographique Lambert 2 étendu, en mètres)
  - EPSG:900913 = SRS Spherical Mercator (projection cartographique popularisée par les APIs Google Maps, Yahoo Maps, Bing, etc)

Conformité OGC.

OpenLayers est donc conforme OGC, c'est-à-dire qu'il est capable de dialoguer avec un serveur cartographique OGC WMS qui implémente la spécification Web Map Service de l'OGC.

Nous allons illustrer comment un *mapping framework* dialogue avec un serveur cartographique par des règles standardisées. Le serveur http://ogo.heig-vd.ch/geoserver/wms est basé sur le logiciel

serveur GeoServer (http://geoserver.org) qui est est conforme WMS. Il délivre des représentations cartographiques de couches géographiques stockées dans des fichiers (ex. Shapefile) ou des bases de données (ex. PostGIS). Un tel serveur est capable de traiter les opérations fondamentales de webmapping (requêtes GetMap et GetFeatureInfo) que nous allons détailler dans la suite.

OpenLayers prend en charge tout le dialogue, c'est-à-dire le formatage des requêtes selon les règles édictées dans la spécification WMS. Même si OpenLayers "fait tout le boulot", voyons comment cela fonctionne en profondeur pour comprendre le fonctionnement d'un serveur cartographique.

GetCapabilities :

Tout serveur conforme OGC doit répondre à la requête dite *GetCapabilities* permettant de connaître ses capacités. Par exemple, saisir http://ogo.heig-vd.ch/geoserver/wms?request=getCapabilities.

Le résultat d'une telle requête est un fichier XML de description :

- une section *<Service>* est la carte d'identité du serveur (titre, keywords, contacts, etc)
- la section *<Request>* liste les opérations possibles sur ce serveur ainsi que les capacités de chaque opération. On y trouve GetCapabilities (évidemment), GetMap avec tous ses formats supportés (png, kml, ...) et GetFeatureInfo (décrits ci-dessous), etc.
- les sections *<Layer>* décrivent les couches "cartographiables"
    - le sous-élément *<Name>* est l'identificateur de couche (notez bien que c'est cet identificateur qui est utilisé dans le paramètre *layers* lors de la construction d'une *OpenLayers.Layer.WMS*)
    - les sous-éléments *<LatLonBoundingBox>*, *<SRS>* et *<BoundingBox>* indiquent l'emprise géographique de la couche complète et son système de référence spatial
    - les sous-éléments *<Style>* indiquent les noms des styles applicables sur cette couche (le premier style est celui par défaut)

C'est donc grâce à ce fichier de description qu'on va pouvoir piloter les deux opérations suivantes.

TODO: lister les couches disponibles sur le serveur ogo et pour chaque couche le style par défaut

GetMap :

L'opération la plus importante pour un serveur cartographique est celle qui permet d'obtenir une représentation cartographique d'unecouche disponible. Pour cela il faut fournir dans la requête GetMap un ensemble de paramètres permettant au serveur de construire une image :

- REQUEST GetMap
    - le type de requête (chaque requête sur le serveur doit préciser l'opération à déclencher)
- LAYERS ogo:world_simple
    - la liste des identificateurs de couches à cartographier (le séparateur est la virgule)
- STYLES polygon
    - la liste des noms de style à appliquer pour chaque couche (le séparateur est la virgule), et si non-indiqué, le serveur utilise le style par défaut (défini dans le GetCapabilities)
- BBOX -180,-90,180,90
    - l'emprise géographique de la zone à représenter (selon l'emprise complète de la couche,

cf. GetCapabilities)

- SRS EPSG:4326
  - le système de référence spatial dans lequel la BBOX est exprimée et définissant la projection de la représentation cartographique produite
- FORMAT image/png
  - le format de l'image produite (notez bien, c'est le paramètre format dans la construction d'un OpenLayers.Layer.WMS)
- HEIGHT 500

  WIDTH 1000
  - la taille de l'image produite

Quelques exemples pour mettre les mains dans le moteur :

```
http://ogo.heig-vd.ch/geoserver/wms?REQUEST=GetMap&LAYERS=ogo:world_simple&BBOX=-180,-90,180,90&SRS=EPSG:4326&STYLES=polygon&FORMAT=image/png&HEIGHT=500&WIDTH=1000
http://ogo.heig-vd.ch/geoserver/wms?REQUEST=GetMap&LAYERS=ogo:world_simple&BBOX=468053,64252,851256,306967&SRS=EPSG:21781&STYLES=polygon&FORMAT=image/png&HEIGHT=500&WIDTH=1000
```

GetFeatureInfo :

Seconde opération que l'on attend d'une application de webmapping concerne la possibilité d'obtenir des informations sur les entités géographiques représentées cartographiquement dans l'image carte produite. Pour cela il faut fournir dans la requête GetFeatureInfo un ensemble de paramètres permettant au serveur de construire un résultat.

La requête exprime des valeurs de paramètres correspondant à un clic d'interrogation à une position x et y sur la carte produite :

- REQUEST GetFeatureInfo
  - bien évidemment le type de requête
- INFO_FORMAT text/plain
  - le format du résultat (cf. getCapabilities, formats listés sous l'élément <GetFeatureInfo>)
- LAYERS ogo:world_adm0
  - idem au GetMap
- QUERY_LAYERS ogo:world_adm0
  - les couches à interroger (le séparateur est la virgule)
- BBOX -180,-90,180,90
  - idem au GetMap
- SRS EPSG:4326
  - idem au GetMap
- WIDTH 1000

  HEIGHT 500

- ○ idem au GetMap
- X 600
  Y 200
  - ○ la position d'interrogation en pixels

Quelques exemples :

- produit un résultat sous forme de tableau HTML

http://ogo.heig-vd.ch/geoserver/wms?INFO_FORMAT=text/html&BBOX=-180,-90,180,90&FORMAT=image/png&LAYERS=ogo:world_simple&QUERY_LAYERS=ogo:world_simple&REQUEST=GetFeatureInfo&SRS=EPSG:4326&STYLES=polygon&WIDTH=1000&HEIGHT=500&X=600&Y=200

- produit un résultat XML au format GML que l'on peut alors transformer à sa guise

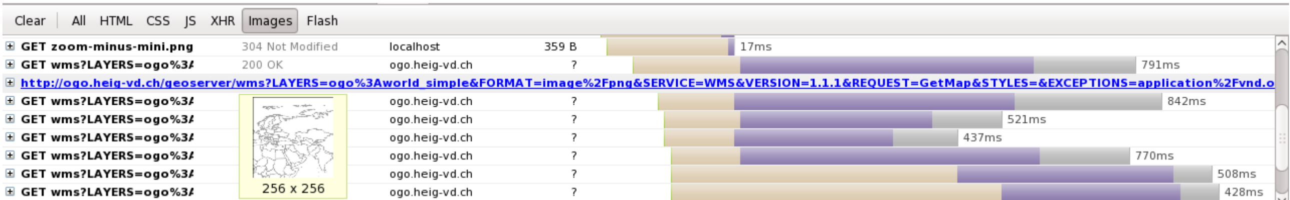
http://ogo.heig-vd.ch/geoserver/wms?INFO_FORMAT=application/vnd.ogc.gml&BBOX=-180,-90,180,90&FORMAT=image/png&LAYERS=ogo:world_simple&QUERY_LAYERS=ogo:world_simple&REQUEST=GetFeatureInfo&SRS=EPSG:4326&STYLES=polygon&WIDTH=1000&HEIGHT=500&X=600&Y=200

OpenLayers.Layer.WMS :

C'est donc OpenLayers qui s'occupe de tout ça, c'est-à-dire de générer toutes ces requêtes et d'en exploiter les réponses :

- au travers de la classe OpenLayers.Layer.WMS dont les paramètres de construction correspondent aux paramètres de requête GetMap ci-dessus
- notez que certains paramètres sont déduits par le framework à partir du contexte de la carte, typiquement c'est le cas des paramètres SRS, BBOX, WIDTH, et HEIGHT.
- dans cette exemple, il suffit de préciser le paramètre layers et OpenLayers s'occupe du reste (le serveur appliquera ici le style par défaut)
- on indique aussi le format d'image car celui par défaut du serveur est JPEG, ce qui peut ne pas être adapté en terme de qualité de rendu

Voyez plutôt Firebug qui montre très bien ce qu'OpenLayers produit comme requête à chaque déplacement et zoom.



Vous remarquerez une particularité par défaut d'OpenLayers qui scinde la demande GetMap d'une zone en tuiles régulières, ici de taille 256x256. Cette technique permet d'améliorer l'expérience utilisateur lors de la navigation :

- lors d'un déplacement, seules les tuiles manquantes sont chargées et complètent le puzzle
- de plus, un certain nombre de tuiles autour de la zone sont aussi pré-chargées

Voyez la différence avec l'exemple Ex1b_wms.html dans lequel un paramètre de constructeur indique qu'il ne faut pas utiliser ce système de tuilage (*singleTile: true*).

Couches cartographiques non-standards.

De même, OpenLayers se charge du dialogue avec d'autres serveurs non-standards offrant des fonds cartographiques comme le libre OpenStreetMap et les propriétaires Google Maps, Bing, ... ou des

technologies propriétaires comme ArcIMS d'ESRI.

Voyez l'exemple Ex1c_gmaps.html :

- on découvre une carte utilisant un fond Google Maps comme couche de base
- préliminaire indispensable pour les sources propriétaires, il faut s'identifier auprès de Google avec une clé API valide

```
<script src='http://maps.google.com/maps?
file=api&amp;v=2&amp;key=ABQIAAAADPWudfErG2kEBVfQJH1_6RRE5u3-
il7140dhX7bjx2Sq1uUC7xTar8Pida8s1EV-ehO37BcWVUjdVQ'></script>
```

- bien évidemment, Google ne suivant pas le standard OGC, l'opération GetCapabilities est remplacée par une information noyée dans la documentation pour connaitre les types de carte fournis au travers de l'API Google Maps :

  http://code.google.com/apis/maps/documentation/reference.html#GMapType.G_NORMAL_MAP

Voyez avec Firebug les requêtes "à la Google Maps" qui ressemblent à ceci :

http://mt1.google.com/vt/v=app.118&hl=en&src=api&x=135&y=90&z=8&s=Galileo

Dans l'exemple, on utilise la méthode setCenter pour centrer et zoomer sur une zone précise :

- la position LonLat est donnée en degrés
- il existe une dizaine de niveaux de zoom

TODO: expérimentez ces positions et niveaux de zoom en les modifiant

Voyez aussi l'exemple Ex1d_metacarta.html.

WMS: personnalisation de style

Nous avons vu que GetMap requiert le paramètre STYLES, il indique un nom de style à associer par couche. Nous avons vu que les noms de styles internes au serveur cartographique sont listés dans le fichier XML GetCapabilities.

Quand le paramètre STYLES n'est pas renseigné dans une requête GetMap, ce sont les styles par défaut qui sont appliqués. On retrouve ce paramètre dans OpenLayers.Layer.WMS.

```
wms = new OpenLayers.Layer.WMS( "OpenLayers WMS", "http://localhost/geoserver/wms", {
       layers: 'ogo:world_adm0',
       format: 'image/png',
       styles: 'un_autre_nom_de_style' // cf. GetCapabilities
 });
```

TODO: reprendre Ex1a_wms.html, et ajouter le paramètre styles avec un autre nom de style applicable (cf. getCapabilities)

Voyez aussi l'exemple Ex2a_internal_styling.html.

Les styles internes sont donc figés, ce qui ne convient pas toujours. Ainsi, GetMap peut recevoir un paramètre spécifique permettant au client de prendre le contrôle total du rendu cartographique.

```
wms = new OpenLayers.Layer.WMS(
         "OpenLayers WMS",
         "http://localhost/geoserver/wms",
         {
            format: 'image/png',
            SLD: 'http://ogo.heig-vd.ch/ologctuto/sld/pinkWorld.sld'
```

```
        }
    );
```

- c'est le paramètre SLD, qui signifie Styled Layer Descriptor
- ce n'est rien d'autre qu'une URI qui pointe sur un fichier de description de style
- ce langage de description SLD est standardisé par l'OGC ainsi que son compagnon SE (Symbology Encoding) qui permet de décrire un style personnalisé

Voyez un exemple simple pour appliquer un nouveau style aux unités administratives de la couche world_simple : Ex2b_external_styling.html et pinkWorld.sld.

Jusqu'à présent, l'association entre une couche et un style interne à appliquer se fait au travers des paramètres *layers* et *styles* de la requête GetMap.

En utilisant une description SLD externe, l'association est décrite dans cette description par l'imbrication des éléments *<NamedLayer>* et *<UserStyle>*

```
<NamedLayer>
  <Name>ogo:world_simple</Name>
  <UserStyle>
    <Name>Pink polygons</Name>
    <FeatureTypeStyle>
      <Rule>
        <PolygonSymbolizer>
           <Fill>
              <CssParameter name="fill">#FFC4FF</CssParameter>
              <CssParameter name="fill-opacity">1.0</CssParameter>
           </Fill>
           <Stroke>
              <CssParameter name="stroke">#000000</CssParameter>
              <CssParameter name="stroke-width">1</CssParameter>
           </Stroke>
        </PolygonSymbolizer>
      </Rule>
    </FeatureTypeStyle>
  </UserStyle>
</NamedLayer>
```

- l'élément *<NamedLayer>* indique la couche à visualiser
- l'élément *<UserStyle>* décrit les règles de représentation à appliquer en se basant sur le langage Symbology Encoding)

Dans l'exemple pinkWorld.sld, il s'agit d'appliquer une règle de symbologie à toutes les entités géographiques à représenter, s'agissant de polygones, cette règle utilise un *<PolygonSymbolizer>* avec ses paramètres *<Fill>* et *<Stroke>*.

Voyez dans les références complémentaires quelques liens de documentation, tutoriels et exemples sur les langages SLD et SE de l'OGC.

Requête GetMap en POST

Remettons un moment les mains dans le moteur. Si la méthode en GET est souvent utilisée pour un GetMap, ce que fait d'ailleurs OpenLayers, il est aussi possible d'exprimer la requête en POST.

Celle-ci s'exprime en XML, voyez l'exemple WMS_GetMap_externalStyle.xml :

- <*GetMap*> est l'élément racine, c'est assez évident, et en trois grandes sections parentes, on retrouve tous les paramètres vus en GET mais ici comme des éléments XML
- notamment, on retrouve intégrer dans cet exemple l'élément <*StyledLayerDescriptor*> qui indique les couches avec les styles souhaités
- <*BoundingBox*> est l'emprise géographique de visualisation (équivalent à BBOX en GET)
- <*Output*> fournit les paramètres graphiques de configuration de la sortie.

TODO: tester l'exemple avec le lanceur de requête fourni par GeoServer

- rendez vous sur http://ogo.heig-vd.ch/geoserver -> Demos -> Demo requests
- saisir dans le champ URL l'adresse du serveur WFS http://localhost:8080/geoserver/wms et copier dans le champ Body le contenu de l'exemple
- dans cet exemple on indique une symbologie personnalisée, analysons-là (elle correspond à la description du fichier capitals.sld qui est aussi utilisée dans l'exemple suivant)
- on découvre une règle de symbologie avec un <*PointSymbolizer*> et un <*TextSymbolizer*>
  - un <*PointSymbolizer*> permet d'afficher l'objet géographique sous forme d'un symbole graphique, il permet de décrire son style (*MarkGraphic, ExternalGraphic, Size, ...*)
  - un <*TextSymbolizer*> permet d'afficher des étiquettes (labels) et de décrire le style des labels à afficher (*LabelPlacement*, ...). Ici on affiche la propriété attributaire WUP_CAPIT comme étiquette (c'est le nom de la ville)
- on découvre aussi que la règle est filtrée
  - un filtre s'effectue sur une propriété, géométrique ou attributaire

```
<ogc:Filter>
  <ogc:PropertyIsEqualTo>
    <ogc:PropertyName>CAPIT_1_0</ogc:PropertyName>
    <ogc:Literal>1</ogc:Literal>
  </ogc:PropertyIsEqualTo>
</ogc:Filter>
```

  - ici, la règle de symbologie ne s'applique donc qu'aux entités dont la propriété attributaire CAPIT_1_0 = 1 (que c'est une capitale de pays)
  - notez bien que, quand une règle est filtrée sur un attribut, le style devient alors très dépendant du modèle d'entité géographique associé (cette symbologie ne fonctionne alors qu'avec cette couche de données)

TODO: modifiez la description en enlevant le filtre (le bloc <ogc:Filter>) et constatez ...

Voyez aussi l'exemple WMS_GetMap_internalStyle.xml, qui montre l'utilisation de styles internes avec l'élément <*NamedStyle*> en lieu et place du <*UserStyle*>.

Chaque imbrication <NamedLayer> + <NamedStyle> correspond à une couche à représenter. Dans cet exemple, il y a deux imbrications, on obtient donc la construction d'une carte par fusion superposée de deux couches.

TODO: tester l'exemple avec le lanceur de requête fourni par GeoServer.

On peut donc dire qu'un serveur cartographique construit une représentation image d'une ou plusieurs couches d'entités géographiques.

Requête GetMap multi-couches

Revenons à OpenLayers avec l'exemple Ex2c_multilayers.html. Ci-dessous les deux lignes à bien comprendre :

```
      layers: 'ogo:worldadm,ogo:cities',
      styles: 'giant_polygon,capitals',
```

En effet, on demande ici la représentation de deux couches avec chacune son style associé. OpenLayers formule une requête GET qui ordonne au serveur de superposer la couche des capitales sur la couche des pays du monde et fusionner le tout en une image. Celui-ci va :

- d'abord appliquer le style interne nommé giant_polygon sur la couche ogo:world_simple
- puis le style interne nommé capitals sur la couche ogo:cities (voir les explications ci-dessus sur le fichier capitals.sld)
- le résultat est une image contenant les deux couches superposées (évidemment, l'ordre est important ici, en inversant, la symbologie des pays cacherait les villes).

Carte avec plusieurs couches (image overlays).

Nous avons vu que le serveur cartographique pouvait gérer la fusion superposée de plusieurs couches en une image. Ceci peut tout à fait être voulu, mais il n'est alors plus possible de les dissocier côté client (on ne peut pas juste désactiver la couche des villes).

Une autre approche consiste alors en une gestion des superpositions côté client OpenLayers (c'est d'ailleurs la première force d'OpenLayers).

Superposition d'image overlays côté client.

Voyez l'exemple Ex3a_overlay_wms.html :

- on ne demande plus au serveur de faire la fusion des deux couches par une requête WMS
- on demande les deux couches séparemment par deux requêtes WMS (deux instances OpenLayers.Layer.WMS)
- en apparence, rien de change, sauf à l'utilisation du sélecteur de couches
  - notez que le sélecteur de couche est activé par la ligne suivante que nous détaillons un peu plus loin dans ce document

    ```
    map.addControl(new OpenLayers.Control.LayerSwitcher());
    ```

  - on anticipe ici sur les contrôles, mais c'est par ce sélecteur qu'on comprend la différence
- en effet, ce sélecteur liste les couches de la carte
  - *Base layer : World admin boundaries*, c'est la couche de référence
  - *Overlays : World cities*, c'est la couche superposée, que l'on peut maintenant activer/désactiver !

Comment est-ce possible ?

OpenLayers gére la superposition par les possibilités HTML et CSS, mais aussi, la seconde requête contient un nouveau paramètre important, *transparent: 'true'*, qui indique au serveur de produire une image transparente pour que la superposition côté client puisse se faire. En effet, le serveur produit deux images indépendantes à superposer côté client, la seconde doit donc être transparente.

TODO: enlevez ce paramètre. Que se passe-t-il ?

Ne pouvant faire jouer la transparence côté client, le sélecteur de couche indique à présent deux couches de référence (Base layer). Les "base layers" sont exclusives, il faut choisir l'une ou l'autre.

Reprojection.

Revenons un instant sur le Système de Référence Spatial, dont nous avons déjà commencé à parler en début de document. Les couches que nous avons manipulé jusqu'à maintenant sont en système EPSG:4326 (voyez l'élément *<SRS>EPSG:4326</SRS>* de la description GetCapabilities).

Rappellons que sans autres précisions, OpenLayers travaille par défaut en EPSG:4326 sur l'emprise maximale du globe -180,-90,180,90 en degrés.

Jusqu'à présent, nous avons travaillé sur le même système de bout en bout du processus

- les entités sont stockés nativement côté serveur en EPSG:4326
- les entités sont représentés côté client en EPSG:4326

Que se passe-t-il si une couche était stockée dans un autre système ? C'est le cas pour la couche géographique *ogo:g4districts98_region* (<SRS>EPSG:21781</SRS> selon le GetCapabilities).

Voyez l'exemple Ex3b_overlay_wms+suisse.html, la Suisse est bien placée sur le globe !

Comment est-ce possible ?

C'est le serveur cartographique WMS qui est capable de construire une image cartographique dans différents systèmes de projection.

- sa force vient de sa capacité à reprojeter les coordonnées d'un système à un autre
- ici le serveur transforme les coordonnées du système EPSG:21781 (Suisse) vers celui de notre carte, EPSG:4326 (WGS84)

Le fichier getCapabilities montre la longue liste des systèmes que le serveur sait gérer (~4000 !).

NB: une librairie bien connue du monde libre permet ce genre de reprojection, il s'agit de PROJ.4 (http://trac.osgeo.org/proj).

Couche de référence dite commerciale.

Voyez l'exemple Ex3c_overlay_gmaps+wms.html. La couche de référence est de type OpenLayers.Layer.Google (ne pas oublier d'indiquer sa clé API).

Or le système d'utilisation d'une telle couche ne correspond pas au système par défaut d'OpenLayers, EPSG:4326. En effet, le système Google est dit "spherical mercator" et correspond au code EPSG:900913 en unité métrique.

Il s'agit donc de reconfigurer la carte en précisant le système de projection et l'emprise maximale. C'est le rôle du paramètre *options* qui permet entre autres options de définir les propriétés *projection* et *maxExtent* :

```
var options = {
        projection: new OpenLayers.Projection("EPSG:900913"),
        maxExtent: new OpenLayers.Bounds(-20037508, -20037508, 20037508, 20037508)
};
map = new OpenLayers.Map('map', options);
```

Pour la projection, on utilise la classe OpenLayers.Projection avec le code EPSG approprié. Pour

l'emprise, on utilise la classe OpenLayers.Bounds avec les bornes minx, miny, maxx, maxy (les bornes de l'emprise sont inhérentes au système de projection).

Pour le système suisse :

- l'emprise maximale est approximativement (485472, 75285, 833838, 295935)
- le point (600000, 200000) est à Bern
- le point (0,0) est aux environs de Bordeaux
- les intervalles X et Y n'ont pas d'intersection, c'est volontaire !

Notez bien :

- une propriété supplémentaire dans la construction de OpenLayers.Layer.Google, il faut ajouter *sphericalMercator: 'true'* (Ex1_gmaps.html est pour cela incomplet*).
- *setCenter* n'utilise donc plus de position LonLat en degrés mais en mètres du nouvel SRS

(*) pour plus d'explications, voir page 32 de la documentation fournie

Carte avec plusieurs couches (vector overlays)

Une des grandes forces d'un mapping framework “moderne”, de l'ère “web 2.0”, c'est la possibilité de manipulation et superposition d'entités géographiques côté client.

Jusqu'à présent, la sélection des objets géographiques et leur rendu graphique se font côté serveur avec un résultat sous forme d'image bitmap. Or, le potentiel des navigateurs est aujourd'hui tel qu'ils permettent le rendu d'objets graphiques vectoriels. Ces objets graphiques peuvent être gérés en VML, en SVG, avec Canvas.

Il est donc possible de faire la sélection côté serveur, de transmettre les objets géographiques au client qui s'occupe alors de les transformer en objets graphique pour le rendu final.

- OpenLayers dénomme ce type de couche "vector overlay"
- selon le navigateur détecté, les objets graphiques sont construits en SVG, en VML, ...
- plus d'informations dans la documentation API d'OpenLayers
  http://dev.openlayers.org/apidocs/files/OpenLayers/Renderer-js.html

Comme le rendu est alors réalisé côté client, c'est aussi ce client qui doit connaître et appliquer le style de représentation pour transformer les objets géographiques en objets graphiques. OpenLayers propose un *Styling Framework* fortement inspiré du standard *Symbology Encoding* de l'OGC.

Etape 1 : création d'un "vector overlay"

Une telle couche contient soit des points, des lignes ou des polygones (on peut les mélanger au sein de la même couche, mais cela est rarement d'intérêt). Retenez le terme en anglais de *"feature"* pour désigner un objet géographique (on dit un "vector overlay" est composé de "features").

Comme indiqué, ces objets proviennent du serveur après sélection, le scénario client/serveur est donc le suivant (cf. aussi OGC portrayal schema) :

sélection/création, encodage [serverside] ► [clientside] décodage, transformation/rendu graphique

La sélection/création peut se faire depuis une source de données spatiales pour être un SGBD (PostGIS, Oracle Spatial, ...), ou une base de fichiers, ...

L'encodage/décodage doit se faire dans un format compréhensible par les deux parties, et on

privilégiera un standard :

- (client) OpenLayers possège déjà des encodeurs/décodeurs capables de traiter des formats standardisés comme OGC GML, GPX, GeoJSON, GeoRSS, OGC KML, ...
- (serveur) de nombreux serveurs sont capables de fournir l'information dans ces formats, notamment ceux qui sont conformes OGC (voir ci-dessous)

Sélection avec un serveur conforme OGC WFS.

L'OGC définit le service dit WFS (Web Feature Server) permettant d'extraire des objets géographiques d'une source de données sous-jacente. Comme WMS, ce service offre plusieurs opérations, mais contrairement à WMS, il ne produit pas une image mais les informations brutes sur les objets géographiques d'une couche interrogée.

Ces informations sont formatées dans un langage XML standardisé d'échange, le GML (Geographic Markup Language).

OpenLayers offre un connecteur conforme WFS capable d'appliquer le protocole de dialogue entre le client et le serveur en respect du standard.

Voyez plutôt l'exemple Ex4a_overlay_wfs.html :

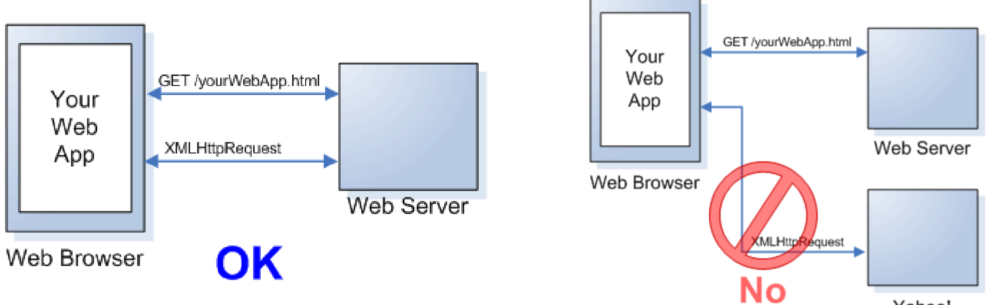
- on découvre un nouveau type de couche, OpenLayers.Layer.Vector
- une telle couche peut donc gérer des “features” (objets géographiques), les manipuler, les "styler", et sur lesquels on peut gérer des événements
- pour ajouter des objets, un protocole peut être utiliser, dans cette exemple OpenLayers.Protocol.WFS.v1_1_0 (précisément dans sa version 1.1.0)
- on indique l'URL sur le serveur conforme WFS et un identificateur de couche (*featureType*)

OpenLayers se charge alors du dialogue client/serveur pour extraire toute une couche depuis ce serveur. Encore une fois c'est lui qui fait tout le boulot. Chaque objet reçu est alors ajouté à notre "vector overlay" qui est le receptacle final côté client avant la dernière étape, le rendu graphique.

Ce rendu graphique se fait en fonction d'un style, ici c'est celui par défaut typique d'OpenLayers (orange transparent) qui est appliqué (cf. Etape 2 pour tous les détails sur le rendu graphique).

Utilisation d'un proxy (same origin policy).

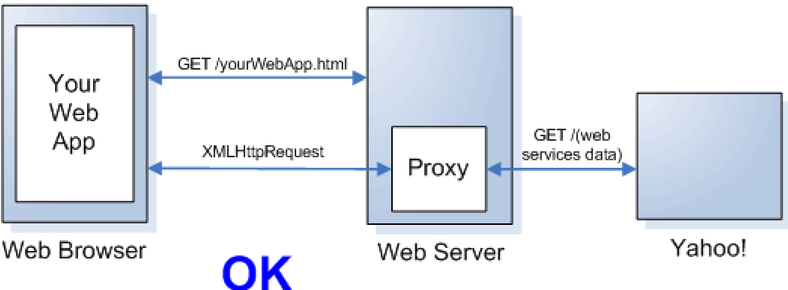
La création du vector overlay requiert la connexion à un serveur distant par une requête XMLHttpRequest. Or les navigateurs interdisent d'effectuer des requêtes asynchrones sur un autre serveur distant que celui qui a transmis la page d'origine. On parle de “Same Origin Policy” qui est une restriction de sécurité visant à limiter les attaques de type phishing.



L'introduction d'un proxy côté serveur d'origine permet de contourner cette restriction légitime. Au

niveau du client, OpenLayers doit alors être configurer pour passer par ce proxy pour toutes ses requêtes XMLHttpRequest, c'est ce qui se fait au travers de l'instruction suivante :

```
OpenLayers.ProxyHost = "/cgi-bin/proxy.cgi?url=";
```



Les requêtes WFS au peigne fin.

Voyons ce qui se passe lors du dialogue client/serveur (utilisons Firebug) :

- notons d'abord les requêtes GET wms pour la couche de référence (*ogo:world_simple*), mais ne concernent pas le vector overlay bien évidemment
- c'est la requête POST wfs qui nous intéresse, gérée par OpenLayers.Protocol.WFS, elle permet d'interroger le serveur pour extraire la couche *cities*
- on découvre le contenu de la requête WFS *<GetFeature>*

```
<wfs:GetFeature xmlns:wfs="http://www.opengis.net/wfs" service="WFS" version="1.1.0" ...
```

- et de la réponse (*<FeatureCollection>* composée de *<featureMember>*)

```
<wfs:FeatureCollection numberOfFeatures="156" ...
```

La suite va nous permettre de décrypter dans le détail la requête et la réponse selon la spécification OGC WFS. On remet encore un peu les mains dans le moteur pour comprendre les différentes opérations liées à l'interface WFS.

GetCapabilities :

Comme pour l'interface WMS, un fichier dit GetCapabilities est bien utile et se construit à partir de la requête suivante : http://ogo.heig-vd.ch/geoserver/wfs?request=getCapabilities

- les éléments *<FeatureType>* indiquent les couches géographiques "extractibles"
  - le sous-élément *<Name>* est l'identificateur de couche
  - le sous-élément *<OutputFormats>* indique les formats d'extraction possibles
  - le sous-élément *<ows:WGS84BoundingBox>* indique l'emprise géographique de la couche complète dans le système WGS84 (en degrés donc)

C'est grâce à ce fichier de description qu'on va pouvoir piloter les deux opérations suivantes.

DescribeFeatureType :

Une requête *DescribeFeatureType* permet de connaître le modèle de données d'une couche. On la désigne par son identificateur au travers du paramètre *typeName*.

Par exemple : http://ogo.heig-vd.ch/geoserver/wfs?request=DescribeFeatureType&typeName=ogo:cities

Sous la forme d'une schéma XML de descripton, on découvre les noms des attributs qui la composent et les types (xsd:string, xsd:double, ...).

```
<xsd:complexType name="citiesType">
    <xsd:complexContent>
      <xsd:extension base="gml:AbstractFeatureType">
        <xsd:sequence>
          <xsd:element maxOccurs="1" minOccurs="0" name="the_geom" nillable="true" type="gml:PointPropertyType"/>
          <xsd:element maxOccurs="1" minOccurs="0" name="IMS_ID" nillable="true" type="xsd:int"/>
          <xsd:element maxOccurs="1" minOccurs="0" name="GEO_CODE" nillable="true" type="xsd:long"/>
          <xsd:element maxOccurs="1" minOccurs="0" name="WUP_AGGL" nillable="true" type="xsd:string"/>
          <xsd:element maxOccurs="1" minOccurs="0" name="CNTRY_NAME" nillable="true" type="xsd:string"/>
                ...
```

Cela est utile pour l'opération *<GetFeature>* décrite ci-dessous, permettant d'exprimer des extractions par filtre sur un attribut (comme quand on exprime un "where" en SQL) ou par des filtres spatiaux comme l'intersection, l'union, le buffer, etc.

GetFeature :

Une requête *GetFeature* permet d'extraire des objets géographiques. Par exemple en GET :

```
http://ogo.heig-vd.ch/geoserver/wfs?request=GetFeature&typeName=ogo:cities&outputFormat=GML3
```

Elle demande de délivrer toute une couche géographique :

- celle-ci est désignée par son identificateur au travers du paramètre *typeName*
- et le paramètre *outputFormat* indique le format de livraison, ici en GML3.

La réponse est donc formatée en GML par une suite d'éléments *<gml:featureMembers>* qui chacun décrit un objet géographique de la couche : sa géométrie (the_geom) et ses valeurs attributaires (WUP_AGGL, CAPIT_1_0, ...). L'entête indique la taille de la réponse.

```
<wfs:FeatureCollection numberOfFeatures="142" ...
  <gml:featureMembers>
        <ogo:cities gml:id="cities.2">
              <ogo:the_geom>
                    <gml:Point>
                          <gml:pos>19.8167 41.3333</gml:pos>
                    </gml:Point>
              </ogo:the_geom>
              <ogo:IMS_ID>2</ogo:IMS_ID>
              <ogo:GEO_CODE>2272</ogo:GEO_CODE>
              <ogo:WUP_AGGL>Tirana</ogo:WUP_AGGL>
              <ogo:CNTRY_NAME>Albania</ogo:CNTRY_NAME> ...
```

TODO: indiquer comme outputFormat=json et utilisez http://jsonformat.com pour y voir clair.

Une requête *GetFeature* peut aussi s'exprimer en POST, c'est d'ailleurs la méthode utilisée par OpenLayers.Protocol.WFS. On peut alors utiliser un riche langage XML d'interrogation. Notez que les opérateurs possibles de ce langage de filtre sont listées dans le fichier GetCapabilities, sous l'élément *<ogc:Filter_Capabilities>*.

Revenons sur l'analyse du dialogue client/serveur de l'exemple Ex4a_overlay_wfs.html avec Firebug et http://xmlindent.com pour y voir clair. On découvre comment OpenLayers formule la requête WFS dans le formalisme XML défini par l'OGC :

- *<wfs:GetFeature>* est l'élément racice, assez évident, avec son élément fils *<wfs:Query>*
- on retrouve un attribut *typeName* pour indiquer la couche concernée par l'interrogation
- suit alors l'élement *<ogc:Filter>* permettant de filtrer les objets (les sélectionner)

- *<ogc:BBOX>* est un opérateur qui permet de formuler un filtre sur des objets géographiques en ne sélectionnant que ceux qui sont dans une enveloppe géographique donnée

```
<ogc:BBOX>
      <ogc:PropertyName>the_geom</ogc:PropertyName>
      <gml:Envelope xmlns:gml="http://www.opengis.net/gml" srsName="EPSG:4326">
            <gml:lowerCorner>-48.546875 31.8388671875</gml:lowerCorner>
            <gml:upperCorner>62.546875 62.1611328125</gml:upperCorner>
      </gml:Envelope>
</ogc:BBOX>
```

Le filtre est appliqué ici sur l'attribut *the_geom* (forcément de type géométrique) et selon une *<gml:Envelope>* dont on définit les coordonnées minx,miny et maxx,maxy.

OpenLayers formule ce filtre à cause de la stratégie OpenLayers.Strategy.BBOX() qui est définie sur notre vector overlay. Cela permet de ne demander au serveur que les objets qui sont strictement nécessaire pour la visualisation en cours. En effet, l'enveloppe est alors définie en fonction de la zone actuellement visible dans la vue cartographique. En conséquence, à chaque déplacement sur la carte, une requête POST wfs sera lancée pour chercher les objets manquants.

TODO: remplacer OpenLayers.Strategy.BBOX() par OpenLayers.Strategy.Fixed() et analysez la réponse avec Firebug.

On constate alors qu'une seule requête POST wfs est lancée pour toute la couche (il n'y a plus de filtre sur la zone visible). Ainsi, le premier chargement est plus long, mais plus de requête à chaque déplacement.

TODO: prenons la place d'OpenLayers et faisons un GetFeature "à la main", en POST

- rendez vous sur http://ogo.heig-vd.ch/geoserver -> Demos -> Demo requests
- saisir dans le champ URL l'adresse du serveur WFS http://localhost:8080/geoserver/wfs
- copier dans le champ Body le contenu de l'exemple WFS_GetFeature_filtered.xml
  - on utilise ici un nouvel opérateur *<ogc:PropertyIsEqualTo>* sur l'attribut *CAPIT_1_0*
  - on sélectionne les objets dont cet attribut est égal à 1 (ce sont les capitales)

```
  <ogc:Filter>
    <ogc:PropertyIsEqualTo>
      <ogc:PropertyName>CAPIT_1_0</ogc:PropertyName>
      <ogc:Literal>1</ogc:Literal>
    </ogc:PropertyIsEqualTo>
  </ogc:Filter>
```

TODO: ajoutons maintenant un second filtre <ogc:BBOX> comme vu ci-dessus

- copier cette fois-ci l'exemple WFS_GetFeature_filtered+BBOX.xml
- on utilise alors un opérateur logique *<ogc:And>* pour exprimer les deux filtres car on ne veut que les capitales dans une enveloppe donnée
- vérifier qu'on obtient bien moins d'objets géographiques (*numberOfFeatures*).

Toute la syntaxe de filtre est explicitée dans le document OGC suivant :

OpenGIS® Filter Encoding Implementation Specification, http://portal.opengeospatial.org/files/?artifact_id=8340

GetFeature et filtre avec OpenLayers

Heureusement nous n'avons pas (la plupart du temps) à produire toute cette syntaxe XML mais aide à la compréhension du mécanisme. De plus, comme OpenLayers s'inspire fortement des standards OGC, les documents de spécification OGC peuvent être considérés comme faisant partie intégrante de la documentation OpenLayers.

Avec OpenLayers et OpenLayers.Protocol.WFS, pour ajouter le filtre sur les capitales équivalent à ci-dessus, on ajoute le paramètre *defaultFilter*. Voyez l'exemple Ex4b_overlay_wfs_filtered.html.

Ce paramètre fait référence à un filtre de comparaison OpenLayers.Filter.Comparison qui se construit de manière très similaire à la syntaxe XML vue ci-dessus.

```
filter_capitals = new OpenLayers.Filter.Comparison({
                                                    type: OpenLayers.Filter.Comparison.EQUAL_TO,
                                                    property: "CAPIT_1_0",
                                                    value: "1"
                                               });
```

Et voilà, c'est OpenLayers qui formule la requête et décode le résultat !

TODO: vérifier que la syntaxe de requête WFS produite correspond bien à ce que l'on attend.

L'utilisation de standards assure l'intéropérabilitié des systèmes d'informations et celle-ci est indispensable dans un cadre multi-acteurs comme au sein d'une infrastructure de données spatiales mais aussi plus globalement pour le développement d'un GeoWeb permettant à tous les acteurs de créer des réponses par composition de services géographiques et cartographiques.

Création depuis un fichier en format standardisé

contrairement au service WFS qui établit un protocole clair avec des opérations précises, dans ce nouveau cas, aucun protocole n'est utilisé, on importe simplement un fichier à “composante spatiale”. Voyez l'exemple Ex5a_overlay_GPX.html.

On découvre un nouveau type de couche, OpenLayers.Layer.GML qui pemet d'ajouter des objets géographiques depuis un fichier distant dans un overlay vector.

Les paramètres du constructeur sont le titre de la couche, l'url du fichier source et les options comme le style à appliquer mais surtout le format d'encodage du fichier qu'il faut préciser.

```
  lgpx = new OpenLayers.Layer.GML("Traversée des Pyrénées",
                        "vector/pyrenees.gpx",
                        {
                            format: OpenLayers.Format.GPX
                        }
                      );
```

OpenLayers supporte plusieurs formats, parmi lesquels le format GPX (GPS eXchange Format), un format de fichier permettant l'échange d'informations géographiques spécifiques au GPS (waypoint, track, route). L'encodeur/décodeur GPX permet de décoder le contenu d'un fichier GPX pour alimenter en objets géographiques un overlay vector.

Notez aussi que le proxy est inutile ici car le fichier source est sur le serveur origine de l'application de webmapping.

Voyez à présent l'exemple Ex5b_overlay_GPX_gmaps.html qui met en oeuvre un overlay GPX sur le fond Google G_PHYSICAL_MAP. On découvre aussi l'utilisation d'un nouveau paramètre,

projection. En effet, si précédemment nous n'apportions aucune information sur le SRS de notre fichier source, c'est que celui-ci était compatible nativement avec la couche de référence et le SRS par défaut d'OpenLayers (un fichier GPX contient en général des coordonnées EPSG:4326).

```
  lgpx = new OpenLayers.Layer.GML("Rocs d'Oche",
                          "http://www.cafleman.fr/caf/gpx/480.gpx",
                          {
                              format: OpenLayers.Format.GPX,
                              projection: new OpenLayers.Projection("EPSG:4326")
                          }
                        );
```

Dans ce nouvel exemple, le fait que la couche de référence soit en projection différente EPSG:900913, il faut préciser le paramètre projection car le fichier source est en projection EPSG:4326.

Notez que le proxy est ici requis car le fichier source est sur un serveur distant différent du serveur origine de l'application de webmapping.

OpenLayers possède bien d'autres formats d'encodage/décodage d'objets géographiques :

- bien évidemment le format GML au travers du protocole WFS
- WKT: un format texte utilisé comme représentation des géométries au sein de requêtes GQL dans un SGBD comme PostGIS, ou avec OrbisGIS
- GeoJSON: un format utilisé essentiellement dans les échanges client/serveur des applications AJAX
- GeoRSS: un format donnant à un flux RSS une composante spatiale
- KML: un format maintenant largement répandu, mais c'est un cas particulier, car il va plus loin que l'encodage d'objets géographiques, il définit également des styles à appliquer et des interactions à activer. Sa philosophie actuelle est celle du HTML du contenu cartographique, mélangeant fond et forme dans un même langage (son avenir est à surveiller ...).

TODO: ajouter sur la carte les fichiers GPX grandraid1.gpx et grandraid2.gpx.

Utilisation du format GeoJSON

Voyez l'exemple Ex5c_overlay_JSON.html qui charge le fichier source 4capitals.json formatté en GeoJSON (JSON Geometry and Feature Description, http://geojson.org).

La structure GeoJSON de ce fichier est simple, elle suit le format Javascript Object Notation :

```
{ "features" :
       [
      { "geometry" : { "coordinates" : [ 2.3332999999999999, 48.866700000000002
              ],
            "type" : "Point"
          },
        "geometry_name" : "the_geom",
        "id" : "cities_capital_pt.247",
        "properties" : { "CAPIT_1_0" : 1,
            "CNTRY_NAME" : "France",
            ...
            "WUP_CAPIT" : "Paris",
            "Y_2003" : 9794
```

```
        },
      "type" : "Feature"
    }, ...
```

C'est donc d'abord un objet, avec deux propriétés :

- type: indiquant le type du jeu de données encodé (FeatureCollection)
- features: un tableau d'objets géogrpahiques, de type Feature (contenant la géométrie et les attributs)

Cas d'utilisation typique : une requête GQL dont le résultat est encodé en GeoJSON et transmis au client pour décodage et représentation. Voyez l'exemple Ex5d_overlay_JSON_db.html.

Dans cet exemple, la source est un script PHP qui procède à une extraction d'objets géographiques pilotée par une requête sur une base PostGIS. Voyons cela Ex5d_querydb_JSON.php.

```
select x(the_geom), y(the_geom), "WUP_AGGL" as name from cities where "GEO_REGION" = 'Europe';
```

On se base sur l'attribut GEO_REGION pour n'extraire que les villes européennes (!).

Les résultats sont formatés en GeoJSON grâce à une fonction d'encodage bien pratique de PHP5. Cependant, il faut respecter la structure du format, ce qui requiert la définition du modèle de classes qui implémente la spécification GeoJSON. Les classes requises sont FeatureCollection, Feature, et dans cet exemple seul le type Point est implémenter.

Pour un support complet de GeoJSON en PHP, le reader/writer du plugin MapFish de Symfony est une piste intéressante (http://www.symfony-project.org/plugins/sfMapFishPlugin). Symfony est un framework PHP5 offrant des fonctionnalités d'ORM (Object-Relational Mapping). Couplé à ce reader/writer GeoJSON, cela devient un bon début de “geoframework” côté serveur ...

Le script joue donc le rôle de service non-standardisé, contrairement à WFS qui est un service standardisé. Quand il s'agit de choisir, tout dépend donc du besoin d'intéropérabilité du système.

Utilisation d'un encodage personnalisé.

Si l'intéropérabilité ne compte pas du tout dans un projet, on peut ni opter pour un service standardisé, ni même un format standard. Voyez l'exemple Ex5e_overlay_custom.html et le fichier source 4capitals.txt.

Le fichier source est chargé par OpenLayers.loadURL (c'est une requête XMLHttpRequest) et comme son contenu est encodé de manière personnalisée, il faut assumer le décodage.

L'encodage est ici très simple à base de séparateurs et c'est par l'utilisation de split qu'on va réaliser un décodage sans difficulté. Notez que nos objets sont ici des points et donc très simples à gérer !

- pour chaque objet géographique décodé, on créé un OpenLayers.Feature.Vector qui se compose d'une géométrie et d'attributs
- la géométrie est construite avec un OpenLayers.Geometry.Point
- les attributs sont stockés dans un tableau associatif
- tous ces objets sont alors ajoutés à un overlay vector (OpenLayers.Layer.Vector)

Mise au point sur les projections et OpenLayers.

Remarquez que dans l'exemple précédent nous n'avons rien géré concernant le SRS. En effet, l'exemple est simplifié, car la source et la carte utilisent le même SRS.

Si ce n'était pas le cas, il faudrait lors du décodage reprojeter les coordonnées des points reçus pour correspondre à la projection de la carte. C'est ce que fait déjà OpenLayers pour tous les formats qu'il supporte au travers du paramètre *projection* prévu à cet effet et vu ci-dessus.

Si nous devions assumer la reprojection côté client, OpenLayers offre des outils de transformation de système grâce à la méthode *transform* des types de OpenLayers.Geometry. Tout dépend de la classe OpenLayers.Projection qui pilote les transformations de systèmes. Nativement, OpenLayers sait gérer les reprojections entre les deux systèmes EPSG:4326 (WGS84) et EPSG:900913 (SphericalMercator des API propriétaires). Pour un support plus étendu d'autres systèmes, il faut charger la librairie *proj4js*.

Pour en savoir plus : http://trac.openlayers.org/wiki/Documentation/Dev/proj4js.

De manière général, il est toujours préférable d'effectuer les reprojections côté serveur de sorte que les objets géographiques soient déjà adaptés à la projection de la représentation cartographique côté client. D'où l'intérêt d'un service comme WFS qui permet la reprojection. Notons aussi que côté serveur, un SGBD spatial comme PostGIS est capable de reprojection par sa fonction *transform* (cf. ci-dessous).

Etape 2 : application de style côté client.

Rappelons que dans le cas d'un vector overlay, la sélection des objets géographiques est faite côté serveur, et ceux-ci sont alors transmis au client qui doit alors les transformer en objets graphiques. Ce rendu graphique des objets doit suivre des règles de style, et OpenLayers propose pour cela un Styling Framework adapté.

Utilisation du paramètre de couche *style* :

Voyez l'exemple Ex6a_overlay_style.html. Le style peut d'abord se définir sous la forme d'un objet Javascript par une suite de propriétés qui est rattaché à une couche par le paramètre dédié *style*, indiquant la symbologie à appliquer. Dans un cet exemple, le style est simple et s'applique à tous les objets géographiques de la couche.

Le styling framework suit la même logique que le langage Symbology Encoding de l'OGC : le style décrit dans cet exemple est équivalent à la description SLD/SE redDotCities.sld.

On voit la correspondance entre les propriétés du styling framework d'OpenLayers et les éléments de la grammaire Symbology Encoding, par exemple :

- les éléments *<Graphic><Mark><WellKnownName>* et la propriété *graphicName*
- l'élément *<Fill>* et la propriété *fillColor*
- l'élément *<Size>* et la propriété *pointRadius*

  (remarquez que *<Size>* = 2 * *pointRadius* !?!)

TODO: comparez l'exemple avec Ex3a_overlay_wms.html qui applique cette description SLD/SE.

Propriétés de style :

La liste des propriétés de style possibles est donnée en page 25 de la documentation fournie (http://docs.openlayers.org/library/feature_styling.html#style-properties). Chaque propriété est adaptée au type géométrie. Pour le type point, deux catégories de style sont applicables :

- graphicName pour les formes connus, circle, star, cross, x, square, triangle, et sur lesquels d'autres propriétés ont un effet :

    - les propriétés stroke... pour le contour de la forme
    - les propriétés fill... pour le remplissage de la forme
- externalGraphic pour des symboles prégénérés dans un format graphique raster comme JPEG et PNG, vectoriel comme SVG, et sur lesquels d'autres propriétés ont un effet :
    - graphicOpacity pour l'opacité du symbole graphique
    - graphicXOffset, graphicYOffset pour définir un déplacement par rapport à l'ancrage
    - graphicWidth, graphicHeight pour définir la taille du symbole en X et Y (notamment utile quand le graphique n'a pas la même dimension initiale en X et Y)
- pointRadius, rotation sont des propriétés qui ont effet sur les deux catégories. Cas particulier, pointRadius est ignoré si graphicWidth et graphicHeight sont définis pour un externalGraphic.

Pour le type ligne, les propriétés d'intérêts sont les propriétés stroke... pour le style de trait.

Pour le type polygone, les propriétés d'intérêts sont les propriétés fill... pour le remplissage de l'objet géographique et les propriétés stroke... pour le style de contour.

Utilisation du paramètre de couche *styleMap* :

Voyez l'exemple Ex6b_overlay_style.html qui montre que les propriétés de style peuvent aussi être encapsulées dans la classe OpenLayers.Style. Dès lors que le style est alors exprimé par une instance OpenLayers.Style et non plus un simple objet Javascript par une suite de propriétés, il faut utiliser le paramètre de couche *styleMap*, et non plus *style*. Cette nouvelle classe dédiée au style offre des fonctionnalités que nous allons développer dans la suite.

```
cities = new OpenLayers.Layer.Vector("WFS - cities", {
           styleMap: redDotCities,
           strategies: [new OpenLayers.Strategy.BBOX()], ...
```

Voyez l'exemple Ex6c_overlay_styleMap.html. Comme nous avons vu qu'il est possible de filtrer une règle de symbologie dans le langage SLD/SE (cf. WMS_GetMap_externalStyle.xml), il est aussi possible avec OpenLayers d'appliquer un filtre de style en exprimant une règle filtrée (notez bien que nous avons déjà défini un tel filtre avec OpenLayers dans l'exemple Ex4b_overlay_wfs_filtered.html et on va s'en inspirer).

On retrouve donc la même notion de règle que dans le langage SLD/SE :

```
redDotCities = {
  graphicName: 'circle',
  pointRadius: 5,
  fillColor: '#ff0000',
  fillOpacity: 0.8
  };
style_capitals = new OpenLayers.Style();
filter_capitals = new OpenLayers.Filter.Comparison({
                      type: OpenLayers.Filter.Comparison.EQUAL_TO,
                      property: "CAPIT_1_0",
                      value: "1"
                 });
rule_capitals = new OpenLayers.Rule({
                    filter: filter_capitals,
```

```
                              symbolizer: redDotCities
                              });
        style_capitals.addRules([rule_capitals]);
```

- pour exprimer un style avec règle filtrée, il faut utiliser la classe OpenLayers.Style qui offre une méthode addRules pour ajouter des règles filtrées
- la classe OpenLayers.Rule permet de créer une règle en spécifiant un filtre et un style
- le filtre est ici une instance OpenLayers.Filter.Comparison exactement comme dans l'exemple Ex4b_overlay_wfs_filtered.html
- et le style (symbolizer) est la suite des propriétés de style

Le moteur de rendu applique ce style qu'aux objets satisfaisants la comparaison, les autres objets restant alors invisible (sans style).

TODO: ajouter une règle filtrée pour visualiser les autres villes avec un style différent

Corrigé: Ex6d_overlay_styleMap.html et Ex6e_overlay_styleMap.html (avec la propriété elseFilter)

Utilisation d'un contexte fonctionnel :

Voyez l'exemple Ex6f_overlay_choropleth.html, on souhaite ici créer une carte choroplèthe, c'est une carte en plage de couleurs comme on voit souvent dans les journaux après une votation.

Tout d'abord, on utilise dans cet exemple le protocole WFS 1.1.0 qui supporte la reprojection. En effet, remarquez que la couche de référence est une couche Google, en projection Spherical Mercator, or l'overlay vector *districts* concerne une couche qui est nativement en EPSG:21781 (projection suisse), il nous faut donc spécifier clairement que l'on souhaite des districts reprojetés par le serveur en projection Spherical Mercator EPSG:900913. Cela se fait avec ce protocole en mentionnant un paramètre supplémentaire *srsName: "EPSG:900913"*.

Concernant le styling, on souhaite ici créer une carte choroplèthe sur la donnée statistique d'un résultat de votation contenu dans l'attribut UEMARS2001 de la couche districts, il représente le pourcentage de non au sujet de l'adhésion à l'UE :

- il s'agit donc d'associer une couleur à chacun des cinq intervalles qui classifient le résultat
- les seuils des classes sont 88.04%, 76.21%, 64.21%, 52.3%
- les couleurs de classes sont ["#ffff00", "#ffcc00", "#ff9900", "#ff6600", "#ff0000"]

Bien évidemment, pour répondre à ce problème, il est possible de suivre la logique prédemment décrite en créant une règle par classe. Cependant, la solution de cette exemple emprunte un raccourci bien pratique du styling framework pour éviter cela, et ce par l'utilisation d'un style piloté par contexte :

```
colors = ["#ffff00", "#ffcc00", "#ff9900", "#ff6600", "#ff0000"];
context = {
    getColor: function(feature) {
        val = parseFloat(feature.attributes.UEMARS2001);
        if (val > 88.04) noClass = 0;
        else if (val > 76.21) noClass = 1;
        else if (val > 64.21) noClass = 2;
        else if (val > 52.3) noClass = 3;
        else noClass = 4
        return colors[noClass];
    }
```

```
};
template = {
    fillOpacity: 0.6,
    fillColor: "${getColor}" // selon la valeur retournée par context.getColor(feature)
};
choropleth = new OpenLayers.Style(template, {context: context});
```

- on utilise un style template dont un (ou plusieurs) élément de style varie selon le contexte
- le contexte définit une fonction qui calcule la classe d'appartenance selon la valeur d'attribut
- ainsi, pour chaque objet à représenter, le style approprié est appliqué selon sa valeur UEMARS2001 et par association du numéro de classe calculé avec le tableau de couleur

TODO: implémentez la solution à une règle par classe en utilisant les opérateurs de comparaison disponibles (cf. http://dev.openlayers.org/apidocs/files/OpenLayers/Filter/Comparison-js.html).

La solution recherchée doit être équivalente à la description de symbologie choropleth.sld.

TODO: testez choropleth.sld au choix en utilisant OpenLayers ou par une requête GET.

```
http://ogo.heig-vd.ch/geoserver/wms?
REQUEST=GetMap&LAYERS=ogo:g4districts98&BBOX=485472,75285,833838,295935&SRS=EPSG:21781&STYLES=choro
pleth&FORMAT=image/png&HEIGHT=600&WIDTH=1000
```

L'utilisation de contexte ouvre un large champ de personnalisation de symbologie capable de varier selon le contexte géométrique et/ou attributaire de l'objet géographique à représenter. En effet, chacune des proriétés de style pouvant varier, on peut imaginer faire varier la propriété pointRadius en fonction d'un attribut, par exemple représentant une quantité de population de ville (on parle alors de carte en symboles proportionnels).

Voyez aussi l'exemple Ex6h_overlay_meteo.html pour une mise en application de la proriété de style *externalGraphic* sur des graphiques raster en format PNG.

TODO: sur la base de Ex6b_overlay_styleMap.html, changez la couche de référence par une couche Google de votre choix et personnalisez le style de la couche des capitales en affichant une symbologie différente selon l'appartenance de la ville à un des deux hémisphères.

Corrigé : voyez Ex6g_overlay_NordEtSud.html

Un dernier exemple sur le styling framework avec Ex6i_overlay_readSLD.html qui introduit le format d'encodage/décodage OpenLayers.Format.SLD. En effet, comme nous avons répété que ce styling framework suit la logique OGC Symbology Encoding, cet exemple montre qu'OpenLayers est donc logiquement capable de transcrire une description SLD/SE vers le modèle de symbologie de ce framework, et donc d'appliquer le style voulu en respect de la description.

TODO: analysez avec Firebug l'objet redDotCities et sa règle de symbologie.

Cepedant, les styling framework n'est pas conforme OGC Symbology Encoding, car notamment il a une lacune, il n'est pas possible de définir plusieurs styles (symbolizer) sur un overlay vector. Ainsi, par exemple, ne peut-on "dessiner" une symbologie d'autoroute avec un gros trait rouge et un trait jaune au milieu. Cela se fait avec deux LineSymbolizer en Symbology Encoding permettant de faire deux passes sur les objets : une au gros crayon rouge, une autre au fin crayon jaune.

Interaction cartographique et gestion des événements

OpenLayers offre des contrôles permettant de gérer les interactions cartographiques comme le zoom, le pan, la visibilité des couches, ... nous les avons déjà utilisé, voyons de plus près comment ils sont mis en oeuvre.

Voyez l'exemple Ex7a_map_controls.html :

- dans les optiosn de carte, nous découvrons le paramètre controls qui est un tableau des contrôles à activer. Sans ce paramètre, seul le contrôle PanZoom est actif
- nous indiquons ici un tableau vide pour l'usage de la méthode map.addControl

Mise en place de contrôles.

Voici quelques contrôles (liste non-exhaustive).

Contrôles de navigation

- interactions par les touches du clavier plus, moins, et flèches de direction

```
map.addControl(new OpenLayers.Control.KeyboardDefaults());
```

- interactions par les mouvements de la souris, pan, zoombox, roulette et double-clic

```
map.addControl(new OpenLayers.Control.Navigation());
```

- interactions par boutons des deux panels par défaut

```
map.addControl(new OpenLayers.Control.PanZoom();
```

- interactions par boutons pan et zoom avec barre de niveaux de zoom

```
map.addControl(new OpenLayers.Control.PanZoomBar());
```

Contrôles d'affichage :

- des coordonnées de la position de souris (selon la projection définie dans les options de carte, map.displayProjection)

```
map.addControl(new OpenLayers.Control.MousePosition());
```

- de l'échelle de carte

```
map.addControl(new OpenLayers.Control.ScaleLine());
```

Contrôle de visibilité des couches :

```
map.addControl(new OpenLayers.Control.LayerSwitcher());
```

Personnalisation d'interface.

Dans cet exemple, nous découvrons que la présentation de l'interface est personnalisée. Tout d'abord, le répertoire source des images est modifié. Ce nouveau répertoire contient une galerie d'images pour personnaliser nos boutons d'interface.

```
OpenLayers.ImgPath = "/jslib/OpenLayers/imgogo/";
```

Notons qu'OpenLayers peut se personnaliser par l'usage de CSS. Sans aucune mention de fichier CSS, OpenLayers charge des styles “en dur”. Pour contourner cela, nous désactivons ce chargement par le paramètre d'option *“theme: null”*, et nous incluons le fichier des styles par défaut *OpenLayers/theme/default/*style.css que nous pouvons modifier. Aussi, nous ajoutons un second fichier de style *ogo.css* pour personnaliser nos contrôles.

```
<link rel="stylesheet" href="/jslib/OpenLayers/theme/default/style.css" type="text/css" />
<link rel="stylesheet" href="ogo.css" type="text/css" />
```

Tous les aspects de l'interface ne sont pas totalement personnalisables par CSS, il est prévu que ce soit le cas dans la prochaine version 3.0 du framework.

Ainsi, certains composants sont encore “stylés en dur”, comme le LayerSwitcher dont nous modifions la propriété de style *roundedCornerColor*.

```
map.addControl(new OpenLayers.Control.LayerSwitcher({"roundedCornerColor":"white"}));
```

Pour en savoir plus sur la personnalisation :

- Documentation (page 10), http://docs.openlayers.org/library/controls.html#styling-panels
- Full CSS suppport, http://trac.openlayers.org/ticket/460
- Personnaliser les icônes et controles d'OpenLayers, http://geotribu.net/node/78)
- et bien d'autres avec votre moteur de recherche préféré ...

Ajout de bouton personnalisé.

Si les controles fournis par le framework couvrent déjà un grand nombre d'interactions de base d'une application de webmapping, cela peut ne pas suffir. Il est alors possible d'ajouter des buttons personnalisés avec le contrôle générique OpenLayers.Control.Button.

```
button1 = new OpenLayers.Control.Button({
                                   trigger: function() {
                                                  map.setBaseLayer(gsat);
                                                  panel.controls[0].active = true;
                                                  panel.controls[1].active = false;
                                          },
                                   title: "Google Satellite",
                                   displayClass: "button1"
                                  });
```

Dans l'exemple, deux boutons sont créés, chacun avec son titre d'infobulle (title), sa représentation (displayClass) et sa fonction de déclenchement (trigger).

Les boutons sont alors ajoutés à un panel qui est lui-même ajouté à la carte.

```
panel = new OpenLayers.Control.Panel({defaultControl: button1});
panel.addControls([button1,button2]);
map.addControl(panel);
```

Mais que font ces deux boutons au juste ?

Nous voyons dans l'exemple la présence de deux couches Google pouvant servir de couche de référence. Ces deux boutons doivent permettre à l'utilisateur de choisir entre l'une ou l'autre (à la façon du LayerSwitcher).

Les fonctions de déclenchement de chaque bouton doivent donc gérer la définition de la couche de référence avec *map.setBaseLayer(...)* et les status des boutons avec la propriété *active* des boutons.

Quid de la représentation personnalisée d'un bouton?

On fait usage de noms de classe CSS basés sur le *displayClass* mentionné. En effet, l'ajout des postfixes ItemActive et ItemInactive à la valeur *displayClass* forme les deux noms de classe à configurer.

Voyez le fichier ogo.css, on découvre que deux classes pilotent la visualisation quand le bouton est actif ou inactif : *.button1ItemInactive .button1ItemActive*

```
.olControlPanel .button1ItemInactive {
    background-image: url("/jslib/OpenLayers/imgogo/satellite.png");
    background-color: white;
    left: 50px;
}
.olControlPanel .button1ItemActive {
    background-image: url("/jslib/OpenLayers/imgogo/satellite.png");
```

```
    background-color: orange;
    left: 50px;
}
```

Nous utilisons ici des images pour visualiser ces boutons.

De manière général, un contrôle a son identifiant CSS formé par concaténation des termes du package de classe en supprimant la notation pointée et en changeant OpenLayers en ol :

exemple : OpenLayers.Control.Panel devient .olControlPanel

Voir aussi page 10 de la documentation fournie.

Gestion d'événements.

Pour une carte, un ensemble d'événements est géré par le framework et il est ainsi possible de déclarer des listeners sur ces événements. Voyez l'exemple Ex7b_wms_featureInfoRequest.html qui permet d'interroger une couche et d'obtenir de l'information sur un pays.

Pour cela, on commence par déclarer un listener sur le type d'événement *click* :

```
map.events.register('click', map, function (e) { ... });
```

Ainsi, au clic sur la carte, notre fonction getFeatureInfo est appelée. Son rôle est alors de lancer au serveur une requête GetFeatureInfo (comme vu ci-dessus) :

```
url =  wms.getFullRequestString({
           REQUEST: "GetFeatureInfo",
           BBOX: wms.map.getExtent().toBBOX(),
           X: e.xy.x,
           Y: e.xy.y,
           INFO_FORMAT: 'text/html',
           QUERY_LAYERS: wms.params.LAYERS,
           WIDTH: wms.map.size.w,
           HEIGHT: wms.map.size.h});
```

On compose la requête par la méthode *getFullRequestString*. On retrouve tous les paramètres pour une telle requête, et certaines des valeurs proviennent bien évidemment de l'état courant de navigation :

- la BBOX est déduite de l'étendue de visualisation courante
- les X et Y du pixel cliqué, récupéré par l'objet événement transmis à notre fonction
- QUERY_LAYERS est déduit de la configuration de l'instance OpenLayers.Layer.WMS (on souhaite interroger toutes les couches serveurs qui y sont configurées).
- WIDTH et HEIGHT sont déduits de la carte actuelle

Cette requête est alors envoyée au serveur par un appel AJAX loadURL dont le résultat est affiché dans le conteneur <div> HTML 'info'.

Pour tout savoir sur les EVENT_TYPES :

http://dev.openlayers.org/docs/files/OpenLayers/Map-js.html#OpenLayers.Map.EVENT_TYPES

TODO: afficher les coordonnées géographiques du pixel cliqué sur la carte (utilisez pour cela la méthode getLonLatFromPixel de la classe OpenLayers.Map).

http://dev.openlayers.org/docs/files/OpenLayers/Map-js.html#OpenLayers.Map.getLonLatFromPixel

Utilisation du contrôle GetFeatureInfo.

La solution décrite précédemment a son équivalent comme contrôle OpenLayers. Voyez l'exemple Ex7c_wms_featureInfoControl.html :

```
info = new OpenLayers.Control.WMSGetFeatureInfo({
    url: 'http://localhost/geoserver/wms',
    title: 'Identify features by clicking',
    queryVisible: true,
    infoFormat: 'text/html'
});
map.addControl(info);
info.events.register('getfeatureinfo', info, onGetFeatureInfo);
info.activate();
```

Doc API : http://dev.openlayers.org/docs/files/OpenLayers/Control/WMSGetFeatureInfo-js.html

Très simple d'utilisation, le contrôle gère la requête GetFeatureInfo pour toutes les couches WMS chargée dans la carte (et interrogeable sur le même serveur). Autre fonctionnement plus avancé, le paramètre *queryVisible* indique de ne considérer que les couches visibles.

Pour finir la mise en oeuvre de ce contrôle, il faut déclarer un listener sur le type d'événement 'getfeatureinfo' et activer le contrôle.

Affichage d'une popup.

On parle aussi de bulle ou nuage d'informations. Dans l'exemple, l'affichage ne se fait plus dans un simple conteneur <div> mais dans une OpenLayers.Popup.FramedCloud.

```
map.addPopup(new OpenLayers.Popup.FramedCloud(
  "popup",
  map.getLonLatFromPixel(event.xy),
  null,
  event.text,
  null,
  true
));
```

Doc API : http://dev.openlayers.org/docs/files/OpenLayers/Popup/FramedCloud-js.html

La méthode addPopup permet d'afficher sur la carte une bulle dont on précise la position géographique (par getLonLatFromPixel) et le contenu en se basant sur l'objet événement transmis au listener.

TODO: gérer l'activation du contrôle par un bouton personnalisé de sorte que l'utilisateur puisse naviguer à sa guise et décider le moment venu de lancer une interrogation.

Interaction sur un vector overlay.

Nous avons vu précédemment l'interaction avec une couche image au travers de la capacité GetFeatureInfo offerte par le standard WMS. Considérons à présent un vector overlay avec lequel l'interaction est plus riche du fait que les objets cartogrpahiques sont présents “vectoriellement” côté client.

Voyez l'exemple Ex7d_overlayVector_control.html qui créer une carte sur la base d'une couche image WMS et d'une couche vector WFS filtrée. L'interaction avec les objets de cette seconde couche (capitals) se fait par le contrôle OpenLayers.Control.SelectFeature.

```
        selectControl = new OpenLayers.Control.SelectFeature(capitals);
        map.addControl(selectControl);
        selectControl.activate();

        capitals.events.register("featureselected", capitals, onFeatureSelect);
        capitals.events.register("featureunselected", capitals, onFeatureUnselect);
```

Ce contrôle est ajouté à la carte et l'interaction activée. De plus, deux listeners sont déclarés pour intercepter les types d'événement *featureselected* et *featureunselected*.

Dès qu'un objet est sélectionné, le listener *onFeatureSelect* est sollicité avec transmission d'un objet événement qui référence l'objet sélectionné (evt.feature). Celui-ci est en charge de créer la popup.

```
          feature = evt.feature;
          popup = new OpenLayers.Popup.FramedCloud("featurePopup",
                                      feature.geometry.getBounds().getCenterLonLat(),
                                      new OpenLayers.Size(100,100),
                                      "<h2>" + feature.attributes.CNTRY_NAME + "</h2>" +
                                      feature.attributes.WUP_CAPIT,
                                      null,
                                      true,
                                      onPopupClose);
```

La position d'origine de la bulle popup est déduite par l'appel :

> *feature.geometry.getBounds().getCenterLonLat()*
>
> on calcule le centroid de l'emprise géographique de l'objet (cela fait peu de sens pour un point, mais est valable pour tout type de géométrie).

Le contenu est composé à partir de deux données attributaires de l'objet :

> *feature.attributes.CNTRY_NAME* et *feature.attributes.WUP_CAPIT*

Notez que le listener *onFeatureUnselect* est mis en place pour détruire la popup lorsque un autre objet est sélectionné. Aussi, contrairement à l'exmple précédent, le bouton de fermeture de popup est géré par le listener *onPopupClose*.

Dernier élément nouveau dans cet exemple et qui complète la compréhension de la classe StyleMap concerne la possibilité de définir un style de sélection. En effet, le paramètre styleMap de la couche WFS montre deux les deux types de rendu possibles (render intents) :

```
styleMap: new OpenLayers.StyleMap({
                    "default": redDotCities,
                    "select": selectedCities
               })
```

Par défaut, le style *default* redDotCities est utilisé pour représenter les villes. Dans le cas d'un objet sélectionné et si le type *select* est défini, il va modifier la représentation de l'objet sélectionné.

Remarquez la manière dont les deux styles sont définis :

```
redDotCities = new OpenLayers.Style({
                                         graphicName: 'circle',
                                         pointRadius: 5,
                                         fillColor: '#ff0000'
                                         });

selectedCities = new OpenLayers.Style({ fillColor: '#ffff00' });
```

Le premier définit le style par défaut et le second n'indique qu'une couleur différente. En effet,

comme ces deux styles sont prévus pour être encapsulés dans un StyleMap, il considère que le second style (type select) étend le premier (type default). Inutile donc de redéfinir les autres proprités de style pour le second.

Création et stockage d'objets géographiques (DrawFeature).

Cet exemple montre un scénario complet “allé et retour” entre client et serveur. Il s'agit d'exploiter les capacités d'édition sur vector overlay. Voyez l'exemple Ex7e_overlayVector_drawFeature.html.

La carte se compose d'une couche de référence Google et d'une couche image WMS. Cette dernière représente des tracés (lignes).

Une troisième couche est définie comme un vector overlay :

```
lineLayer = new OpenLayers.Layer.Vector("Edit line layer");
map.addLayer(lineLayer);

drawControl = new OpenLayers.Control.DrawFeature(lineLayer, OpenLayers.Handler.Path);
map.addControl(drawControl);
drawControl.activate();

drawControl.events.register('featureadded', drawControl, onFeatureAdded);
```

Sur cet overlay vector est ajouté et activé le contrôle OpenLayers.Control.DrawFeature. Il offre des fonctionnalités de dessin sur une couche donnée, ici *lineLayer*. Le contrôle permet de dessiner autant des points (*Point*), des lignes (*Path*), des polygones (*Polygon*) selon configuration. Ici, le second paramètre indique *OpenLayers.Handler.Path* pour le dessin de lignes.

Notez que le listener *onFeatureAdded* est déclaré sur le type d'événement *featureadded* et ce afin de lancer un traitement sur la ligne dessinée. Cette ligne est transmise à ce listener par l'objet événement. Mais quel est ce traitement rempli par ce listener ?

```
function onFeatureAdded(e) {
  do {name = prompt("Saisir un nom de tracé :", "");} while ((name == null)||(name == "") );
  url = "insertTrace.php?geom="+e.feature.geometry+"&name="+name;
  OpenLayers.loadURL(url, '', this, function(response){
    OpenLayers.Util.getElement('info').innerHTML = response.responseText;
    lineLayer.removeFeatures(e.feature);
    traces.redraw(true);
  });
}
```

Il s'agit d'envoyer la ligne dessinée à un service de stockage. Ce service est représenté ici par un script PHP qui reçoit deux paramètres, la géométrie de la ligne et un nom :

- le nom est simplement saisi auprès de l'utilisateur par un démodé prompt Javascript !
- la géométrie est obtenue à partir de l'objet événement par *e.feature.geometry*. Le format de représentation textuelle est une suite des coordonnées composant la ligne, par exemple :

```
	LINESTRING(229922 6026906,900122 5136568,2049735 6144314)
```

Que fait à son tour le service ? Voyez pour cela l'exemple *insertTrace.php*.

Il s'agit d'alimenter une base de tracés à partir des deux informations reçues. Ainsi, une requête d'insertion d'un objet géographique en base de données prend la forme suivante :

```
INSERT INTO traces (the_geom, name) VALUES (
	transform(
```

```
                GeomFromText('LINESTRING(229922 6026906,900122 5136568,2049735 6144314)',900913),
                4326
        ),
        'Test de trace'
);
```

Nous ouvrons ici une parenthèse langage GQL (Geographic Query Language) pour le cartouche spatial PostGIS du SGBD PostgreSQL :

- on repère clairement les bouts de requête qui correspondent aux éléments reçus du client : la ligne (*LINESTRING*) et le nom saisi par l'utilisateur
- la fonction *GeomFromText* permet de transformer la *LINESTRING* en la représentation interne à PostGIS d'une géométrie. Notez bien le second paramètre (900913) qui précise à la fonction que les coordonnées fournies sont en système de projection SphericalMercator

  (PostGIS utilise comme OpenLayers les mêmes codes de projection EPSG)
- la fonction *Transform* permet de transformer cette géométrie dans un autre système de projection, code 4326, car nous ne voulons pas stocker la ligne en système 900913
- le résultat de cette transformation est une géométrie valide pour prendre place dans le champ géométrique (*the_geom*) de la table *traces*.

Nous fermons la parenthèse (voyez en annexe comment créer une table spatiale dans PostGIS).

Revenons côté client à notre listener *onFeatureAdded* qui a lancé le stockage par un appel *loadURL*. En retour de cet appel une fonction est exécutée pour finir le travail.

```
    lineLayer.removeFeatures(e.feature);
    traces.redraw(true);
```

La couche vector overlay *lineLayer* ne servant ici que de “planche à dessin”, il faut l'effacer pour pouvoir saisir une autre ligne.

Dernier point, et non le moindre, il faut rafraichir la couche image WMS des traces. En effet, il faut bien comprendre le scénario de cet exemple : la couche vector sert pour la saisie de la trace et la couche WMS sert à représenter toutes les traces sous forme d'image, car celle-ci est en réalité liée au travers du serveur cartographique à la table *trace* du SGBD*. Comme une trace a été ajoutée, le rafraichissement de la couche permet de montrer la nouvelle trace sur cette couche image.

(*) le serveur cartographique expose sous la forme d'une image le contenu de la table.

Références supplémentaires

OpenLayers, la documentation fournie et remis en page, http://docs.openlayers.org

OpenLayers, de nombreux exemples de mise en application, http://openlayers.org/dev/examples

OpenLayers API documentation, http://dev.openlayers.org/docs

Intéressant thèse faisant un état de l'art du *Smart Map Browsing* http://www.smartmapbrowsing.org/index_en.html

Quelques cours PostGIS, GeoServer, OpenLayers du master SIGMA, http://www.geotests.net/cours/sigma/webmapping

Tutoriel SLD/SE avec GeoServer, http://geoserver.org/display/GEOSDOC/SLD+Intro+Tutorial

Exemple SLD/SE, http://geoserver.org/display/GEOSDOC/SLD+Snippets

MassGIS Web Mapping, exemples SLD/SE, http://lyceum.massgis.state.ma.us/wiki/doku.php?id=wms:sld:home

Petit manuel PostGIS, http://www.postgis.fr/node/156