École Polytechnique de l'Université de Tours

64, Avenue Jean Portalis

37200 TOURS, FRANCE

Tél. +33 (0)2 47 36 14 14

www.polytech.univ-tours.fr

Département Informatique

Système interactif de représentation cartographique

Étudiant :

Marc VAILLANT : marc.vaillant@etu.univ-tours.fr

Encadrants :

M.Néron

M.Monmarché

M.Palka

Année Étude : 2011 – 2012

Sommaire

Remerciements

Je remercie M.Néron, M.Monmarché et Gaëtan Palka pour l'encadrement et les décisions prises.

Je remercie également l'étudiant Yuanqing Li qui a participé à l'avancée du projet dans le cadre du projet d'option web.

Enfin, je remercie Tristan Mersch et Fabien Touchard, étudiants, qui m'ont fait part de leur expérience dans le domaine de la programmation géographique avec GéoVélo.

Introduction

L'objet de mon projet de fin d'étude (PFE) est la création d'une application géographique permettant de visualiser les inondations sur Tours et sa région. Cette application devra être utilisée sur internet par les utilisateurs.

Dans un premier temps, je présenterai le contexte et les objectifs du projet. J'indiquerai quelles technologies ont été utilisées pour le projet et les raisons de ces choix. Je préciserai les modifications apportées au cahier de spécification initial.

Dans un second temps, je présenterai le développement du projet, objet du PFE. Cette présentation se découpe en trois parties. Dans la première, j'exposerai ce qui a été réalisé pour la base de données. La seconde partie sera consacrée au fonctionnement du webserveur. Pour conclure, je présenterai la partie client de l'application.

Enfin, je ferai la synthèse du projet.

1 Contexte de la réalisation

1.1 Contexte

Je dois réaliser une application web avec une Interface Homme Machine (IHM) conviviale et complète, suivant les souhaits du client. Le client a participé à plusieurs programmes régionaux, nationaux et européens tels que :

- ACCEL : Évaluation spatio-temporelle de l'ACCessibilité d'Enjeux Localisés en situation d'inondation -ACCEL », de sept 2007 à sept 2010. Mise en œuvre d'une chaine de méthodes multicritères pour déterminer la vulnérabilité dite interdite d'enjeux urbains face aux risques d'inondation. Le but consiste à réaliser des cartes plus pertinentes pour une meilleure gestion du risque d'inondation
- ESPON : European Spatial Network Planning
- RISKMAP

L'application doit permettre une visualisation satisfaisante de l'agglomération Tourangelle en cas d'inondation. Le choix de couleurs des endroits critiques ou des points de hauts risques est modifiable selon les besoins et préférences de la personne qui utilise l'application.

En effet, l'application intégrera un module intelligent qui proposera des patterns différents en fonction du profil de l'utilisateur, permettant une utilisation et visualisation optimisées des cartes produites du logiciel. L'application pourra être utilisée dans plusieurs optiques et devra proposer des layers adéquats. Pour réaliser cette tâche, un algorithme génétique sera mis en place.

1.2 Objectifs

Je reprends une question que l'équipe IPAPE doit traiter pour mon sujet :

« Quels modèles de cartes du risque d'inondation spécifique aux besoins, aux préférences graphiques et aux perceptions visuelles de différentes catégories d'utilisateurs peut-on développer ? »

Mon objectif est de créer une application qui, dans premier temps, tentera de répondre à la question et dans un second temps permettra de satisfaire les besoins de Gaëtan et des membres de l'équipe IPAPE.

Je vais tout d'abord créer l'application qui permettra de visualiser Tours en cas d'inondation. Une fois que la partie serveur/client sera développée, j'intégrerai un algorithme génétique permettant de proposer des choix de layers (sur la carte de Tours) aux différentes personnes amenées à utiliser l'application.

L'utilisateur cliquera sur une proposition d'associations de couleurs qu'il trouve performantes et de nouvelles propositions lui seront soumises en fonction de sa précédente sélection. Il pourra garder son dernier choix ou en sélectionner un nouveau encore plus précis (précis dans le sens que le pattern intégrera ses choix précédents plus les nouveaux) en fonction de ses souhaits. Les choix seront sauvegardés et répertoriés par métiers pour les re-proposer aux personnes intervenant sur les mêmes tâches.

1.3 Hypothèses

Le projet devait au départ être un ajout de plugin dans un logiciel libre nommé QGIS. Ce logiciel était développé avec QT.

L'équipe IPAPE ayant de nouveaux besoins, des changements sont intervenus dans le cahier des charges initial de mon projet. En effet, afin de présenter facilement l'application aux élus locaux, pompiers, etc... il est évident qu'une application web est plus pratique que l'installation d'un logiciel. Cette solution permettra, de plus, de réaliser une enquête à grande échelle.

1.4 Bases méthodologiques

Pour les besoins de l'application web, beaucoup d'outils vont être utilisés. Voici la liste des langages et outils :

Client	Serveur	Communication	Base de données	Web Service
jscolor	Java	Json	PostGreSQL	API MapQuest
OpenLayers	Hibernate		PostGIS	OpenStreetMap
jQuery	Jersey JAX-RS			
HTML/CSS	TomCat			
Json2.js	Maven			

1.4.1 jQuery

Le code côté client est écrit en JavaScript. Pour des raisons de simplicité et de sécurité, il est toujours préférable d'utiliser un framework. Jquery est un framework JavaScript pour les interactions utilisateur/interface. Il sera donc utile à la création de l'IHM. Il s'agit d'une librairie libre qui est très utilisée dans le milieu de la programmation web.

1.4.2 OpenLayers

OpenLayers est un projet complet écrit en JavaScript qui permet de gérer l'affichage d'une carte dynamique dans une interface Web. OpenLayers s'intègre donc parfaitement dans le projet et plus précisément pour la partie client. Il est sous licence BSD et peut, par conséquent, être utilisé librement.

Il permet principalement :

- D'afficher des fonds de cartes disponibles sous forme de tuiles (voir partie OpenStreetMap)
- De dessiner et placer sur ce fond de carte des marqueurs et toutes sortes d'objets géométriques
- De créer des objets à partir de sources de données variées : XML, GeoJSON etc...
- De gérer toutes les projections géographiques existantes pour afficher ces objets
- De générer des événements sur des actions de l'utilisateur : clic sur un objet, passage de souris sur objet, etc...
- De modifier le style d'un objet en fonction d'un événement

1.4.3 Json

Le format JSON, pour JavaScript Object Notation est un format normalisé de représentation d'objets sous une forme textuelle et structurée. Il est très utilisé comme langage de communication dans les applications de type AJAX entre le navigateur et le serveur.

Un message JSON est composé uniquement de l'un des deux éléments suivants :

- Un ensemble de paires clé/valeur, que l'on peut comparer à un objet de type Map en Java
- Une liste de valeurs, que l'on peut comparer à un objet

1.4.4 Json2.js

La librairie Json2.js est une librairie JavaScript permettant de sérialiser un objet JavaScript en une chaîne JSON.

1.4.5 Gson

Gson suit le même principe que Json2.js mais du coté serveur avec le java. Il transforme les objets java en JSON, permettant ainsi l'échange d'informations entre le java et le javaScript. On peut créer des objets Java à partir d'un message JSON envoyé dans le corps de la requête http par le client.

http://code.google.com/p/google-gson

1.4.6 Java et Servlets

Le serveur sera développé en java, plus particulièrement avec des servlets. Ce sont des classes java qui utilisent le package javax.servlet. Elles sont programmées pour répondre à une requête de la part du client. Elles peuvent envoyer en réponse différents types de données : une page HTML, une chaîne au format JSON. L'application serveur sera donc composée d'un ensemble de servlets permettant d'effectuer des tâches, telles que l'interrogation de la base de données.

L'utilisation de servlets présente plusieurs avantages :

- Le découpage modulaire. Chaque servlet fait office de mini web service. En effet, elles fournissent des réponses à des requêtes envoyées par le client. Elles seront indépendantes les unes des autres.
- Les servlets sont très répandues sur internet et la technologie est régulièrement mise à jour.
- Elles ne sont créées qu'une seule fois lors du lancement du serveur ou à la première requête reçue.

1.4.7 PostGIS

PostGis est un plugin de PostgreSQL. Une fois installé, il transforme notre système d'information en un système d'informations géographiques.

C'est à dire qu'il active :

- De nouveaux types de données : point, lignes, polygone. Ces données sont normalement représentées par des coordonnées ce qui rendrait leur stockage coûteux dans un champ de type texte par exemple. PostGIS permet un stockage optimisé de ces données.
- Des fonctions pour extraire ces objets : projection dans un autre système de coordonnées, sortie au format texte, au format GeoJSON, latitude, longitude, etc...
- Des fonctions géométriques : union/intersection de deux lignes/polygones, etc... Cette extension va nous permettre de stocker les contours des zones d'inondation dans TOURS

1.4.8 OpenStreetMap

OpenStreetMap est une application cartographique libre et prône même le travail communautaire du contenu par tout individu susceptible d'apporter des améliorations. OpenLayers utilise les fonds de carte générés à partir des données OpenStreetMap. La carte affichée à l'écran est composée de plusieurs petites images (256*256) « tuiles » qui sont assemblées les unes aux autres sous forme de grille. Chaque déplacement demande une nouvelle requête à OpenStreetMap qui renvoie des nouvelles tuiles de la position souhaitée.

1.4.9 MapQuest

MapQuest est un service de cartes et d'itinéraires en ligne, lancé en 1996. C'est le principal service de cartes en ligne open source qui s'associe avec OpenStreetMap (le fond de cartes est OpenStreetMap). MapQuest ne limite pas le nombre de requêtes par jour comme c'est le cas avec Google Map, il est donc plus adapté pour l'application.

1.4.10 Maven

Maven est un outil libre de gestion et d'automatisation pour la production d'un projet Java. Il permet de télécharger et rassembler les librairies, frameworks, architectures d'un projet dans un fichier .pom ce qui rend le redéploiement du projet beaucoup plus facile. Quand un pom est configuré, il suffit de builder le projet pour que Maven se charge de télécharger tous les objets associés au projet, au lieu de le faire normalement manuellement.

1.4.11 Hibernate

Hibernate est un framework faisant partie du package jBoss. Il permet de créer et gérer des bases de données en utilisant exclusivement du java. Hibernate dispose d'outils de mapping pour gérer les relations objets java et tables de bases de données. Ce framework puissant permet un gain de temps s'il est bien maitrisé ; il permet également d'augmenter la sécurité des échanges de données. Hibernate propose différents types de gestion de la base de données, soit en utilisant des annotations, soit des fichiers xml.

1.4.12 Jersey JAX-RS

Cette librairie permet de gérer des servlets java en utilisant des annotations. Par exemple, lorsque l'annotation @GET est présente au dessus d'une classe, cela signifie que les données seront reçues et envoyées au format GET (html).

2 Description générale

2.1 Environnement du projet

Le projet intègre les données géographiques du Département Aménagement pour connaître les zones d'inondations.

2.2 Caractéristiques des utilisateurs

Comme indiqué précédemment, plusieurs types d'utilisateurs sont amenés à travailler avec l'application. La disposition des objets dans l'IHM ne changera pas en fonction de leur activité. Néanmoins, la représentation cartographique sera modifiable selon le métier exercé.

2.3 Fonctionnalités et structure générale du système

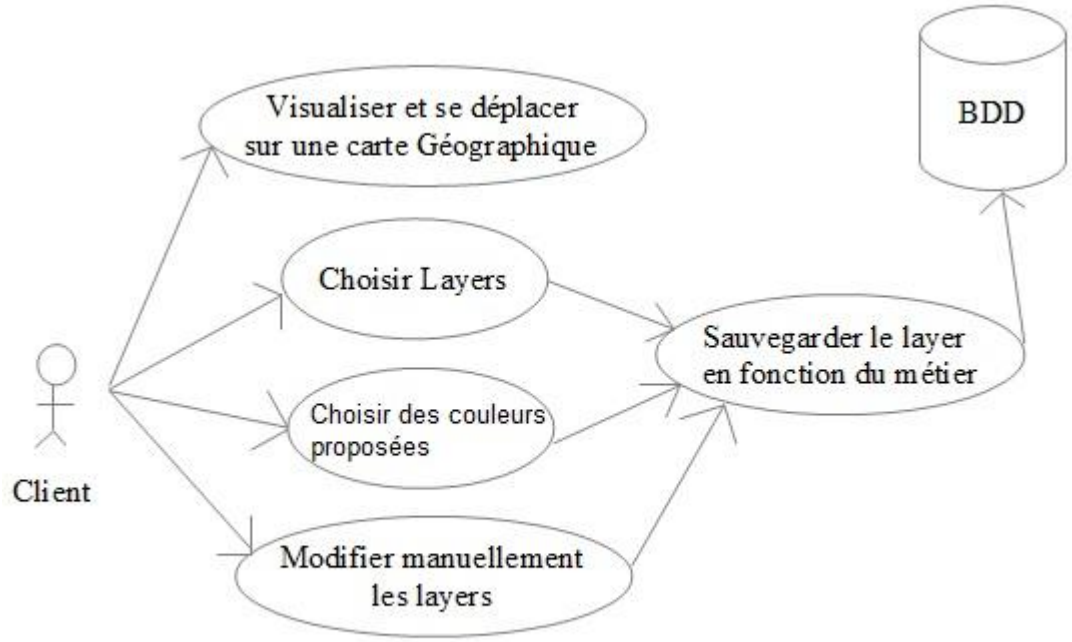

Figure 1 : Diagramme de cas d'utilisation

3 Description des interfaces externes du logiciel

3.1 Architecture générale du système

Voici le diagramme d'environnement représentant les liaisons entre chaque outil qui sera utilisé :

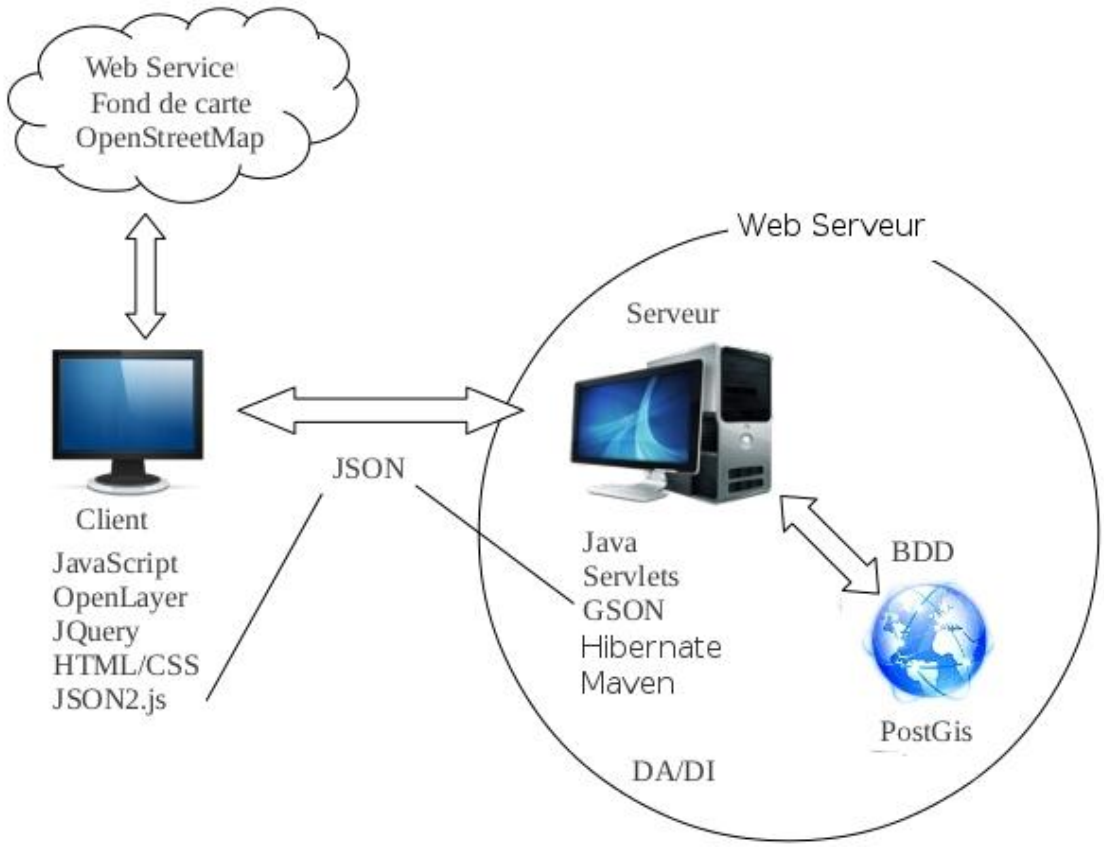

Figure 2 : Architecture générale du système

3.2 Interfaces homme/machine

L'application va être utilisée par des personnes n'ayant pas spécialement de compétences en informatique. L'IHM devra donc être claire et facile d'utilisation. Le fond de map proviendra d'OpenStreetMap qui ressemble fortement à Google Map ; l'utilisateur ne sera donc pas perdu lors de sa navigation.

3.3 Interfaces logiciel/logiciel

En observant l'architecture de l'application, et comme décrit précédemment lors de la présentation des langages et outils utilisés, on constate que l'application utilise des langages différents qui ne sont pas compatibles mais qui communiquent entre eux (par exemple le javascript et le java). Il faut donc passer par une voie intermédiaire de communication qui va être le langage JSON.

Le serveur java va gérer les relations avec la base de données pour toute requête souhaitée par le client.

Enfin, le client va communiquer avec l'API d'OpenStreetMap pour obtenir un fond de map qui servira de base au framework.

4 Description des fonctionnalités

Voici les fonctionnalités redéfinies par rapport au cahier de spécification.

4.1 Navigation sur fond de carte

4.1.1 Présentation de la fonction

Nom de la fonction : init() dans ImportFondCarte.js

- Indispensable pour le reste de l'application car elle importe le fond de carte qui sert de base à openLayer
- Cette fonction contient les appels aux servlets pour importer les données
- Elle contient tous les styles pour les données ainsi que les fonctions permettant de modifier les couleurs

4.1.2 Description de la fonction

- Langage : JavaScript (client)
- Intégrée dans l'interface homme/machine
- Utilise openLayer et jQuery
- Entrée : openStreetMap
- Sortie : fond de carte importé dans openLayer

4.1.3 Conditions de fonctionnement

Le chargement du fond de carte doit être rapide pour éviter les effets indésirables tels que le « saccadage » du navigateur et de l'application.

4.2 Création Interface Homme/Machine

4.2.1 Présentation de la fonction

Nom de la fonction : menu.js et index.html

- L'interface homme machine servira de communication entre l'application et l'utilisateur
- La présentation de cette interface doit être « user friendly »

4.2.2 Description de la fonction

- Langage : JavaScript et JQUERY (client)
- Utilise openLayer
- Entrée : importFondCarte et popupMiniatures
- Sortie : ImportFondCarte avec une interface

4.3 Importation des données géographiques

4.3.1 Présentation de la fonction

Nom de la fonction : nomDuTypeDeDonnéesRessource (Ex : communesRessource)

- Importe les points/lieux d'inondations de la base de données
- Cette fonction est une des bases de l'application car son but premier est de représenter une inondation sur Tours.

4.3.2 Description de la fonction

- Langage : Java (serveur)
- Communique avec la base de données postGIS
- Entrée : Coordonnées de points
- Sortie : Servlet qui renvoie l'objet contenant les points/lieux d'inondations

4.4 IMPORT des couches sur client

4.4.1 Présentation de la fonction

Nom de la fonction : Ces fonctions sont intégrées dans la fonction init(). Elles sont de la forme " jQuery.getJSON(url de la servlet,, function(data)) "

- Importe les couches, telles que celles d'inondation, du serveur
- Fonction transition entre le client et le serveur

4.4.2 Description de la fonction

- Langage : JSON (client)
- Entrée : objetNominatifCoucheX
- Sortie : Couche importée sur le client

4.5 Affichage des Couches

4.5.1 Présentation de la fonction

Nom de la fonction : ces fonctions sont intégrées dans la fonction init().

- Affiche les couches telles que celles d'inondation etc...
- Fonction générale permettant d'afficher dans l'interface les couches désirées

4.5.2 Description de la fonction

- Langage : JavaScript et JQUERY (client)
- Utilise openLayers
- Choix des layers à appliquer
- Entrée : importCouches
- Sortie : Couche appliquée sur la carte

4.6 Modification des couches

4.6.1 Présentation de la fonction

Nom de la fonction : ces fonctions sont intégrées dans la fonction ini().

- Modifie les couleurs des couches

4.6.2 Description de la fonction

- Langage : JavaScript et JQUERY (client)
- Utilise openLayer
- Modification des layers appliqués
- Entrée : les couches
- Sortie : les couches modifiées

4.7 Sauvegarde du schéma de couleur

4.7.1 Présentation de la fonction

C'est un popup jQuery qui s'ouvre et qui demande les informations sur l'utilisateur.

- Ajouter les couleurs que l'utilisateur aura choisies ainsi que ses informations.

4.7.2 Description de la fonction

- Langage : Java (serveur)
- Entrée : Les champs complétés par l'utilisateur
- Sortie : le schéma de couleur sauvegardé avec ses informations

4.8 Algorithme génétique

4.8.1 Présentation de la fonction

Nom de la fonction : popupMiniatures

- Algorithme génétique permettant de proposer plusieurs couches en fonction d'un pattern

4.8.2 Description de la fonction

- Langage : JavaScript et jQuery
- Entrée : 6 miniatures générées aléatoirement
- Sortie : les couleurs de la miniature sélectionnée

5 La base de données

Créer la base de données a été la première étape du PFE. C'est la base de tout traitement de données dans une application. Les données qui sont à ma disposition sont d'un type particulier ; j'ai donc dans un premier temps fait des recherches pour savoir comment les manipuler.

5.1 Manipulation des objets géographiques

On appelle objet géographique un point, une ligne, un polygone… Ces objets sont représentés sur une carte par un couple de coordonnées pour un point, ou une liste de couples de coordonnées pour les objets plus complexes comme une ligne ou un polygone.

Afin de manipuler ces objets et de les afficher sur notre carte, il faut tout d'abord comprendre le fonctionnement, car il existe différents systèmes de coordonnées.

Dans mon projet, j'ai travaillé avec ce type d'objets ; je vais donc expliquer leur fonctionnement global. Si les coordonnées contenues dans la base de données n'utilisent pas le même système que la carte qui les affiche (ce qui était mon cas), les résultats à l'écran seront erronés. On retrouvera, par exemple, des données normalement correspondantes à Tours, en Afrique centrale.

5.1.1 Les coordonnées géographiques

Les coordonnées géographiques permettent de se repérer à la surface de la terre. Il existe trois coordonnées géographiques : la latitude, la longitude et le niveau de la mer.

Les coordonnées géographiques se déduisent à partir d'un système géodésique.

Un géoïde est un objet géométrique qui permet de représenter fidèlement la forme de la Terre. Il s'agit d'une sorte de sphère ou ellipsoïde irrégulière, car la Terre n'est pas une surface régulière.

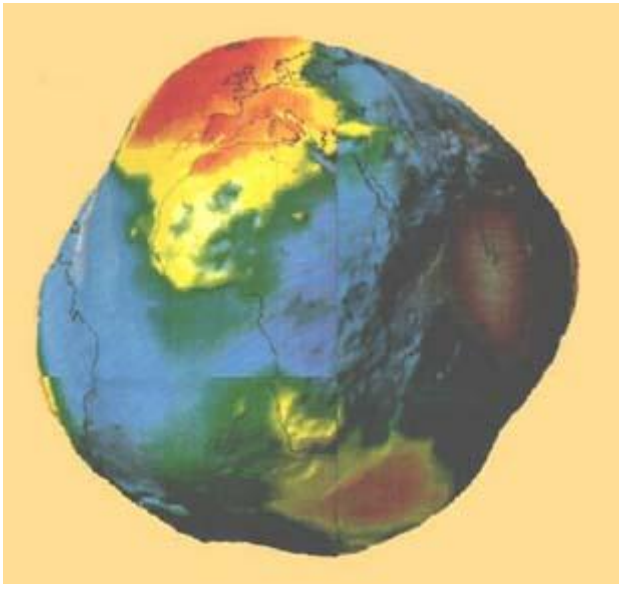

Figure 3 : Terre Géoïde

En réalité, la plupart des systèmes géodésiques utilisent comme référence une ellipsoïde (ou ellipse de révolution) à la place d'un géoïde. Cela simplifie les calculs mais insère une erreur dans ces calculs. Les applications telles que Google Map, OpenLayers, etc... utilisent les ellipsoïdes pour les rapidités de calculs. J'ai donc dû adapter les données qui étaient à ma disposition et les transformer au format qui convenait.

Le système géodésique le plus connu est le WGS 84 (World Geodetic System 1984), qui est le système géodésique mondial. Il est surtout associé à la technologie GPS. Les positions des villes par longitude, latitude que nous utilisons la plupart du temps sont donc associées à ce système.

5.1.2 Les coordonnées cartographiques

Les objets définis par des coordonnées géographiques définies à partir d'un système géodésique ne sont pas affichables sur une carte en deux dimensions. Il faudrait une sphère (3 dimensions) comme le propose Google Earth.

Les cartes en deux dimensions proposées par Google Map ou Open Street Map par exemple sont des projections cartographiques.

Les projections cartographiques sont des techniques un peu complexes avec le principe de base suivant :

- On choisit un objet géométrique : en général un cylindre ou un cône
- On place l'ellipsoïde qui représente la Terre à l'intérieur de cet objet
- On projette l'ellipsoïde sur l'objet
- On déroule l'objet pour obtenir une carte plane

On obtient ainsi de nouvelles coordonnées pour les points sur la carte selon deux axes : abscisse et ordonnée. Ces coordonnées varient en fonction de la technique de projection utilisée mais aussi selon le point de référence utilisé pour dérouler l'objet.

Ces projections entraînent inévitablement des incohérences dans la représentation de la Terre. Il existe de nombreuses projections, chaque pays ayant généralement une projection officielle (parfois partagée par plusieurs). Nous allons nous intéresser à celles que nous avons rencontrées durant notre projet : la projection de Mercator et la projection Lambert II étendu.

5.1.3 Projection de Mercator

Il s'agit d'une projection cylindrique, le cylindre englobant la Terre et étant tangent au niveau de l'équateur.

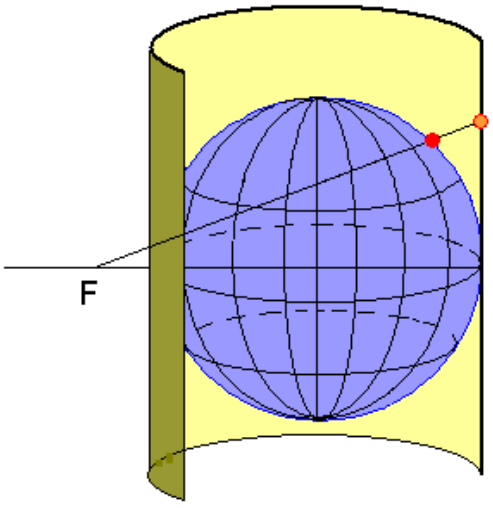

Figure 4 : Projection cylindrique

C’est une projection uniforme c’est-à-dire que les angles observés sur ce planisphère son équivalents à ceux mesurés dans la réalité.

La projection sur un cylindre produit un étirement dans le sens Est-Ouest, mais aussi dans le sens Nord-Sud. Ces deux étirements sont proportionnels, ce qui explique la conservation des angles.

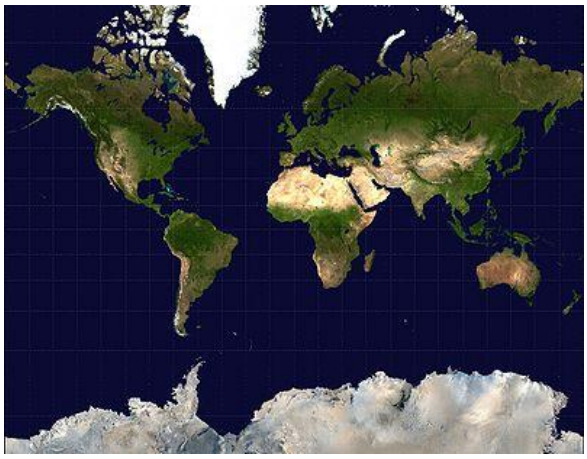

Figure 5 : Projection avec Mercator

A titre d’exemple, l’Amérique du Sud parait plus petite que le Groenland, alors qu’elle est neuf fois plus grande en réalité.

C’est la projection utilisée par Google Map, Open Street Map et par beaucoup d’autres pour représenter le monde sur une surface plane.

5.1.4 Projection de Lambert

Il s’agit d’une projection conique, qui s’effectue de deux façons différentes :

1 Soit le cône englobe la Terre et est tangent en un parallèle

2 Soit il est sécant à la Terre en deux parallèles

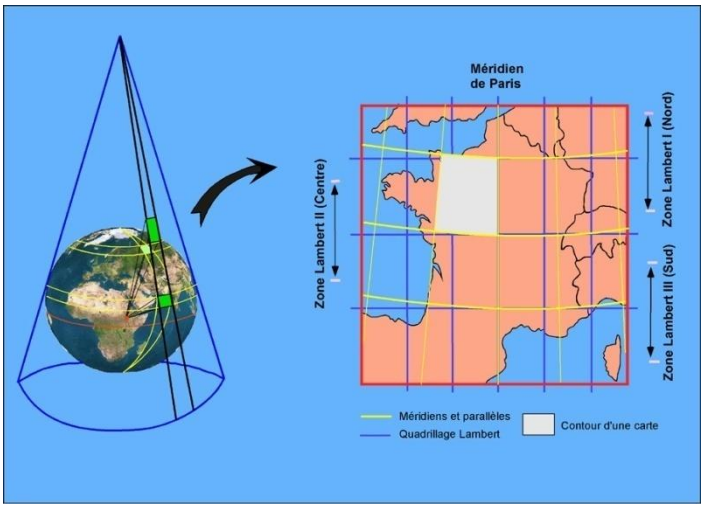

Figure 6 : Projection conique

On obtient une carte qui conserve les angles, dont les méridiens (longitude constante) sont des droites et les parallèles des arcs de cercles. Cela entraîne une grande déformation quand on s'éloigne du parallèle qui a servi de référence. Elle est donc utilisée sur de petites échelles (nationale par exemple).

Pour avoir un résultat correct à l'écran, il faut donc :

- Projeter les contours des objets géographiques selon la projection de Mercator
- Transformer les coordonnées Lambert II en coordonnées Mercator

Il existe des outils dans PostGis qui permettent de transformer un système en un autre.

- Dans le module PostGIS via la fonction **ST_Transform(geometry,integer)**
- Dans la bibliothèque OpenLayers via la méthode : **transform(new Projection(String), new Projection(String))** disponible pour tous les objets de type Geometry

J'ai utilisé la 1ère méthode pour transformer les données car elle est beaucoup plus rapide. Les données sont directement adaptées par la base de données puis envoyées au webservice.

```
SELECT ST_Transform(ST_SetSRID(donnée,27572),900913) FROM table
```

Remarque : la fonction ST_SetSRID(geometry,integer) permet de spécifier à PostgreSQL quel est le système de coordonnées de départ de l'objet géométrique.

5.2 La base de données PostGis et OpenGeo Suite

Lors de mes recherches pour trouver les outils adéquats, j'ai découvert une suite nommée " OpenGeo Community Edition " qui regroupe tous les outils nécessaires à la création, gestion, importation des données géographiques. Elle installe également postGreSQL et postGis, ce qui permet un gain de temps (BDD déjà configurées, etc...).

5.3 L'importation des objets géographiques

Les données géographiques sur lesquelles j'ai travaillé durant mon PFE sont contenues dans des fichiers SHP (ShapeFile). Ces fichiers peuvent comporter plusieurs informations selon la personne qui a récolté les données. On peut par exemple retrouver des noms, profondeurs, etc... Le plus important pour moi était le type dédié aux données géométriques qui pouvaient ensuite être utilisées pour afficher les données sur des cartes. Ces données sont généralement stockées dans une variable nommée " the_geom ". Elle peut être de plusieurs types, tels que des points, des lignes, des polygones.

J'ai importé ces fichiers SHP dans ma Base de données PostGis pour que les données soient ensuite envoyées au webserver. Dans la documentation fournie en annexe, j'explique comment importer un fichier SHP dans la base de données grâce à la suite " OpenGeo ".

Lors de mon PFE, j'ai importé trois fichiers SHP. Un fichier pour les limites des communes, un fichier pour les tronçons d'inondations et un fichier pour les aléas d'inondations. Lorsque j'ai importé les fichiers pour la première fois, les tables se sont créées avec des noms en majuscule, ce qui a posé des problèmes avec Hibernate par la suite. Il faut bien vérifier que les tables soient en minuscule. J'ai donc réalisé ces manipulations plusieurs fois.

5.4 Schéma de la base de donnée

Voici un schéma simplifié de la base de données que j'ai créée durant le PFE. Elle peut grossir très rapidement car chaque fichier SHP ajouté est une nouvelle table créée. Dans le futur, on devrait avoisiner une vingtaine de tables. Pour commencer l'application, je devais d'abord travailler sur peu de données pour vérifier que tout fonctionnait correctement.

Figure 7 : Base de données

La table des Aléas contient quatre niveaux de profondeurs. Cette donnée sera utile pour afficher les profondeurs sur l'application avec des couleurs différentes.

6 Le webserveur

La deuxième étape du PFE, réalisée en parallèle avec la première phase, consiste à créer un webserveur en JAVA EE et à envoyer les informations en JSON au client grâce aux servlets. Pourquoi avoir choisi une telle architecture ?

La librairie géographique utilisée coté client, que nous verrons plus en détail par la suite, permet la gestion des objets en JSON. Envoyer les données géographiques au format JSON à partir du serveur facilite donc grandement la tâche et améliore les vitesses de traitements. Pour programmer, j'ai utilisé l'IDE Eclipse qui, avec ses dernières versions, offre un outil performant (Eclipse Market Place) pour installer des librairies, frameworks, etc...

6.1 Configurer Hibernate

La première étape a consisté à configurer Hibernate pour le projet. J'ai téléchargé le package jBoss qui contient Maven, Hibernate, etc...

Pour configurer Hibernate, il faut créer au moins deux fichiers. Le premier fichier est le fichier **hibernate.cfg.xml**. Il permet de définir les protocoles de communications, les usernames, passwords, drivers, etc... avec la base de données. Voici un exemple de configuration d'Hibernate que j'ai utilisée pour le projet :

```
<hibernate-configuration>

<session-factory>
    <property name="hbm2ddl.auto">update</property>
    <property name="dialect">
        org.hibernate.dialect.PostgreSQLDialect
    </property>
    <property name="hibernate.dialect">
        org.hibernatespatial.postgis.PostgisDialect
    </property>
    <property name="connection.url">
        jdbc:postgresql://localhost:54321/infoGeo
    </property>
    <property name="connection.username">postgres</property>
    <property name="connection.password">postgres</property>
    <property name="connection.driver_class">
        org.postgresql.Driver
    </property>
    <property name="use_sql_comments">true</property>
    <property name="hibernate.generate_statistics">true</property>

    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>

    <mapping resource="troncon_hydrographique.hbm.xml" />
    <mapping resource="InondationNNP.hbm.xml" />
    <mapping resource="Communes.hbm.xml" />



</session-factory>

</hibernate-configuration>
```

Figure 8 : Fichier de config d'Hibernate

On remarque que les dernières lignes du fichier sont dédiées au " mapping " que nous allons voir dans la partie " Les modèles ".

Le deuxième fichier pour configurer Hibernate est un fichier Java qui permet de créer les principales fonctions qui seront utilisées par la suite.

```java
package util;
import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateUtil {

    private static SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new AnnotationConfiguration().configure()
                    .buildSessionFactory();
        } catch (Throwable ex) {
            // Log exception!
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static Session getSession() throws HibernateException {
        return sessionFactory.openSession();
    }

    public static SessionFactory getSessionFactory(){
        return sessionFactory;
    }

    public static void shutdown(){
        getSessionFactory().close();
    }
}
```

Figure 9 : HibernateUtil

Avec ces deux fichiers, on peut alors utiliser Hibernate dans un projet. Nous allons maintenant étudier l'utilisation d'Hibernate avec notre projet.

6.2 Les modèles

La deuxième étape de la création du webserveur consiste à créer les modèles dans le cadre d'une architecture MVC (Modèle-Vue-Contrôleur). Normalement, avec Hibernate, on est capable de créer la base de données à partir des modèles. Néanmoins, dans mon cas, les tables de la base de données sont issues des fichiers SHP et ont un format bien particulier, impossible à deviner lors de la création des modèles. A ce stade, j'ai décidé de créer des modèles " miroirs " des tables de la BDD. Attention, lors de la création des modèles avec

Hibernate, il faut impérativement créer les Getters ET les Setters, même s'il n'est pas prévu de modifier les données avec les setters. Sans Setters, Hibernate génèrera une erreur et l'application ne fonctionnera pas. Une fois les modèles java terminés, j'ai réalisé la mapping des classes avec la base de données. Le mapping sert à créer la liaison entre les modèles et la base de données. Il existe plusieurs méthodes pour créer les mapping, soit en annotations, soit dans des fichiers xml. Ne pouvant pas créer ma base de données avec Hibernate directement, j'ai opté pour les fichiers xml. Voici un exemple de mapping xml :

```
<hibernate-mapping>
 <class name="communes.Communes" table="commune_region">
 <id name="gid" column="gid">
 <generator class="native"/>
 </id>
 <property name="idBdCarto" column="id_bdcarto" />
 <property name="nom_comm" column="nom_comm" />
 <property name="insee_comm" column="insee_comm" />
 <property name="status" column="status" />
 <property name="x_commune" column="x_commune" />
 <property name="y_commune" column="y_commune" />
 <property name="superficie" column="superficie" />
 <property name="population" column="population" />
 <property name="insee_cant" column="insee_cant" />
 <property name="insee_arr" column="insee_arr" />
 <property name="nom_dept" column="nom_dept" />
 <property name="insee_dept" column="insee_dept" />
 <property name="nom_region" column="nom_region" />
 <property name="insee_reg" column="insee_reg" />
 <property name="theGeom" type="org.hibernatespatial.GeometryUserType" column="the_geom" />
 </class>
</hibernate-mapping>
```

Figure 10 : Mapping XML

6.3 Les contrôleurs

Les contrôleurs dans mon webserveur sont les servlets. Une servlet est créée pour chaque envoi spécifique de données. Par exemple, la servlet dédiée à l'envoi des données des limites communales se nomme " CommunesRessource.java ". J'utilise la librairie Jersey JAX-RS pour gérer les servlets. Jersey permet de définir les url, les types d'envoi, les paramètres etc... de chaque servlet via des annotations. Toutes les servlets sont contenues dans un dossier nommé " services ". Par exemple, lorsque l'on veut appeler une servlet, on utilisera une url de ce type :

http://localhost:8081/WebServeurInfoGeo/services/ communesImport

où *communesImport* est l'url associé à une servlet grâce à Jersey. Pour associer une URL à une servlet, on utilise l'annotation suivante :

```
@Path("/communesImport")
@Singleton
public class CommunesRessource
```

Figure 11 : Annotation Path

On peut maintenant appeler la servlet grâce à cette annotation ; il reste à définir ce que la servlet va renvoyer au client. Pour cela, on utilise une nouvelle annotation :

```
public class CommunesRessource {

    @GET
    @Produces("application/json")
    public String getCommunes) {
```

Figure 12 : Annotation Get

Ces deux annotations permettent de définir le type d'envoi des données (GET) et le langage utilisé (JSON).

La dernière étape consiste à ouvrir une session Hibernate, récupérer les données et envoyer les données en JSON. Voici un exemple de code utilisé dans mon application :

```
public String getCommunes) {
    Session session = HibernateUtil.getSession();
    Transaction maTransaction = null;
    List<String> communeGen = null;

    try{
        maTransaction= session.beginTransaction();

        communeGen = session.createSQLQuery("select ST_AsGeoJSON(ST_Transform(ST_Simplify(

        maTransaction.commit();

    }catch(HibernateException e){
        if(maTransaction!=null)
            maTransaction.rollback();
        e.printStackTrace();
    }finally{
        session.close();
    }

    String communegeo= new String();
    communegeo = "{\"type\": \"FeatureCollection\", \"features\": [{\"geometry\":{ \"type\

    return communegeo;

}
```

Figure 13 : Contrôleur Servlet

La ligne contenant le session.createSQLQuery permet d'envoyer une requête SQL à la BDD. Je voulais utiliser des criteria Hibernate pour communiquer avec la BDD (les criteria envoient des requêtes SQL mais sont écrits 100% en java) ; cela n'a pas été possible pour les raisons que nous allons voir par la suite. Voici un exemple de requête SQL que je lance pour récupérer des données.

"select ST_AsGeoJSON(ST_Transform(ST_Simplify(the_geom,600),900913)) from commune_region"

Le problème avec mes requêtes réside dans le fait que j'utilise des types spécifiques à postGis (ST_Transform, etc...) qui ne sont pas gérés par les criteria. Nous avons vu précédemment à quoi servait ST_Transform. Ici, ST_Simplify permet de simplifier les données géographiques pour que le client ait beaucoup moins de points à afficher et permet donc gagner de la fluidité. ST_AsGeoJSON permet de retourner les données sous forme d'objets JSON qui seront ensuite envoyés par la servlet.

Voici un exemple de JSON :

```
{ "type": "FeatureCollection",
  "features":
          [
                  {"geometry":
                          { "type": "GeometryCollection","geometries": "+communeGen+"},
                           "type": "Feature", "properties": {}
                  }
          ]
}
```

CommuneGen correspond à l'objet qui stocke les données retournées par la BDD. Les balises qui entourent cet objet permettent de définir les types de cet objet pour que la librairie openLayers utilisée sur le client puisse mobiliser ces données. Sans la définition des types, features, etc... L'objet communeGen est illisible par openLayers. J'ai réalisé un grand nombre d'essais avant de trouver le format adéquat et lisible par openLayers. J'ai utilisé fireBug pour vérifier que la servlet renvoyait bien du Json et également le site http://jsonlint.com/ qui permet de contrôler l'architecture du JSON. Le site ne livre pas toujours des résultats corrects, ce qui m'a parfois mis sur la mauvaise voie durant mes créations d'objets JSON.

7 Le client de l'application

La partie client est constituée de contrôleurs et de vues. Les contrôleurs servent à récupérer les données et les vues à les afficher.

7.1 La vue

La vue de l'application est l'index.html. Il importe toutes les fonctions et fichiers javascript. Voici un exemple de l'interface de l'application qui a été réalisé dans les débuts du projet. Elle a subi plusieurs modifications. Voici la dernière version :

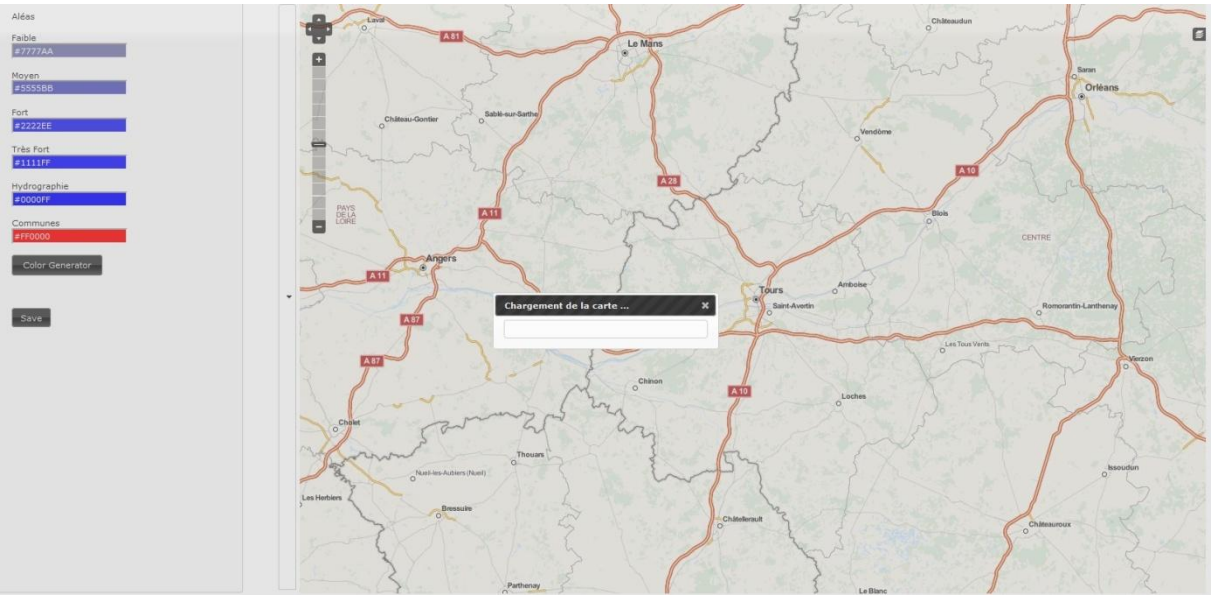

Figure 14 : L'interface

Elle a été réalisée principalement avec jQuery. On remarque que des progressBars ont été implémentés dans l'application pour faire patienter les utilisateurs lors des chargements des données. Néanmoins, ces progressbars ne fonctionnent pas toujours correctement. En effet, à cause du grand nombre de données traitées, dans certains cas, le navigateur n'arrive plus à gérer le transfert des données et l'affichage de la progressBar. J'ai étudié le problème durant de longues heures et je n'ai pas encore trouvé de solutions pour le résoudre. J'ai essayé de faire des wait (affichage de la barre de chargement), etc... Mais rien ne fonctionne pour le moment. La seule barre de chargement qui fonctionne est la barre de chargement des données lors du chargement de l'application. Pour les autres, je poursuis mes recherches.

7.2 Les contrôleurs du client

7.2.1 Importation d'un fond de carte

Le premier travail sur le client a consisté à créer une carte vierge et à importer un fond de carte pour se repérer lors de la navigation sur l'application. J'utilise openLayers pour traiter les données géographiques sur le serveur. On crée une carte de la façon suivante :

map = new OpenLayers.Map (options)

Une fois l'objet carte créé, il suffit d'y insérer du contenu, des textures, des données. Par exemple, lorsque l'on ajoute un fond de carte, voici le résultat dans un navigateur :

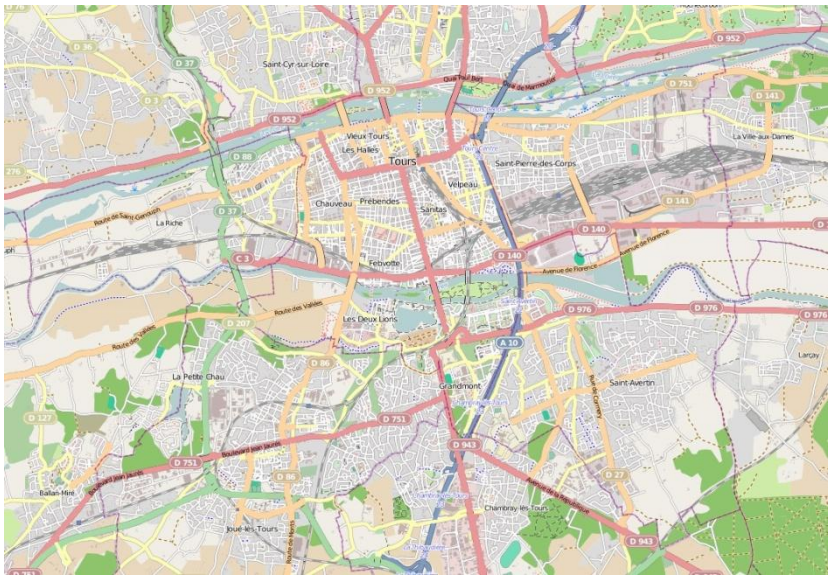

Figure 15 : OpenStreetMap

On utilise map.addLayer(LeLayer) pour ajouter des objets à la carte. La classe Map dispose de plusieurs méthodes pour configurer une carte ; on peut ainsi charger la carte sur la ville de Tours, par exemple, avec la méthode suivante :

map.setCenter (longitudeLatitude, zoom);

La liste des méthodes est disponible à l'adresse suivante :

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

7.2.2 Importer les données du webserveur

La seconde phase consistait à importer des données du webserveur. J'utilise une méthode jQuery, getJSON, qui permet de recevoir les informations envoyées en JSON. Voici un exemple de code permettant de recevoir des données :

```
jQuery.getJSON(urlServletCommunes, '', function(data){
        dataStockCommune= data; //Données brutes
        var geojson_format = new OpenLayers.Format.GeoJSON();
        //Données lisibles après transformation
        dataStockCommune = geojson_format.read(dataStockCommune);

    });
```

Le premier paramètre sert à fournir l'url de la servlet que l'on veut appeler, le second sert à donner des options si besoin et le dernier est le résultat retourné par la servlet. Data contient le résultat de la servlet est peut donc être stocké dans des variables. On peut également observer que j'utilise une méthode d'openLayers qui permet de lire le JSON comme indiqué précédemment.

Maintenant que nous avons nos données, il faut créer un nouveau Layer pour l'intégrer dans la carte.

Pour ce faire, il existe plusieurs méthodes dans openLayers qui permettent de créer des Layers et d'y intégrer des données. Voici un exemple de code :

```
commune = new OpenLayers.Layer.Vector("LimitesCommunes",{styleMap:styleCommune});
map.addLayer(commune);
commune.addFeatures(dataStockCommune);
```

La première étape consiste à créer un Layer de type vector (nos données sont des données géométriques et non des images). Il faut lui donner en paramètre son nom (qui sera affiché dans la vue pour sélectionner ou non ce layer) ainsi que son style (nous reviendrons sur cette dernière option un peu plus tard).

La seconde étape permet d'intégrer nos layers ne contenant pour le moment aucune donnée dans la map que nous avons créée au début.

Pour terminer, nous intégrons les données des limites communales dans le layer commune que nous venons de créer.

Grâce à ce processus, nous avons maintenant un layer contenant les informations envoyées par le webserveur et qui peut être visualisé sur la carte.

Voici le résultat :

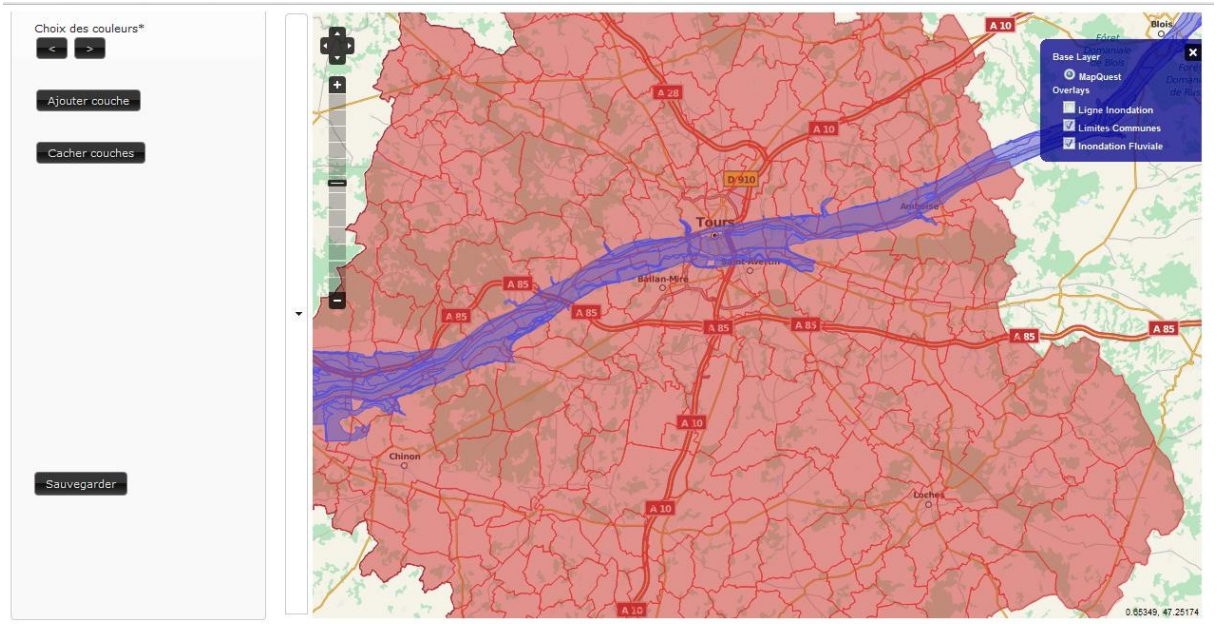

Figure 16 : Importation des couches

Il faut réitérer l'opération pour chaque couche que l'on souhaite afficher.

7.2.3 Les styles des couches (layers)

Nous avons parlé précédemment des styles lors de la création d'un layer. Les styles permettent de donner la couleur, la largeur, la texture, etc... souhaitées. Il faut créer un style par layer (beaucoup plus lorsque l'on veut faire des modifications de couleurs). Voici un exemple de style pour le layer des communes :

```
var styleCommune = new OpenLayers.StyleMap({
    "default": new OpenLayers.Style({
            strokeColor: #FFFFFF,
            strokeWidth:"1",
            fillColor: #FFFFFF,
            fillOpacity:"0"
    })
```

});

La méthode StyleMap prend plusieurs paramètres. Les options " stroke " servent à définir les contours des données. Sur l'image ci-dessus, la couleur des contours est plus rouge que la couleur de remplissage. Les options "fill" sont les couleurs de remplissage. Il existe un grand nombre d'options ; ceci est un petit échantillon.

7.2.4 La modification des layers

La modification des couleurs des couches est l'une des fonctionnalités principales de l'application. Toutes les couches doivent pouvoir être modifiées. Après plusieurs essais et recherches, j'utilise la méthode suivante pour modifier la couleur des layers :

- Sélectionner la couleur
- Supprimer l'ancien layer
- Créer un nouveau layer avec la nouvelle couleur
- Réintégrer les données

Pour sélectionner des couleurs, j'utilise jscolor qui est un plugin javascript permettant d'intégrer une palette de couleur dans un champ html et qui s'affiche lorsque l'on clique dessus.

Pour supprimer, recréer et réintégrer les données, j'utilise les mêmes codes vu précédemment en exemple. Il y a une exception sur la modification des couleurs des 4 aléas liés aux inondations. La couche aléas est divisée en 4 sous couches permettant de donner une couleur différente à chaque couche.

Figure 17 : Les 4 aléas d'inondations

Lorsque l'on doit changer l'une des 4 couleurs, alors les 3 autres s'adaptent à la couleur qui a été choisie en fonction de sa profondeur. Par exemple, l'aléa 1 a la plus faible profondeur et le 4 la plus forte. Si on modifie la couleur de l'aléa 1 avec #FFFFFF, l'aléa 4 prendra la valeur #CCCCCC. Il s'agit d'un exemple ; les vraies valeurs sont calculées avec une soustraction de couleur en fonction de sa profondeur. Ainsi, si l'on considère que l'aléa 1 a un coefficient de modification de 1 (qui ne modifiera pas la couleur choisie) lorsque l'on change sa couleur :

- l'aléa 2 applique filtre de couleur de 1-0,10
- l'aléa 3 applique filtre de couleur de 1-0,15
- l'aléa 2 applique filtre de couleur de 1-0,20

Si l'on modifie l'aléa 2 alors, les coefficients de soustraction seront :

- 1+0,10 pour l'aléa 1
- 1-0,10 pour l'aléa 3
- 1-0,15 pour l'aléa 4

Ainsi de suite pour chacune des modifications de couche. On obtient ainsi un dégradé de couleur constant ; peu importe la couche modifiée.

7.2.5 L'algorithme génétique

Cette partie a été réalisée sur les heures de PFE mais également sur celles du projet d'option web. Elle est expliquée dans le rapport de projet web fourni en annexe.

7.2.6 Sauvegarder le schéma de couleur

La dernière étape consiste à sauvegarder le schéma de couleur avec les informations sur l'utilisateur. J'utilise pour cela un formulaire créée avec jQuery qui permet de rester sur la même page et de récupérer les informations sur les couleurs plus facilement.

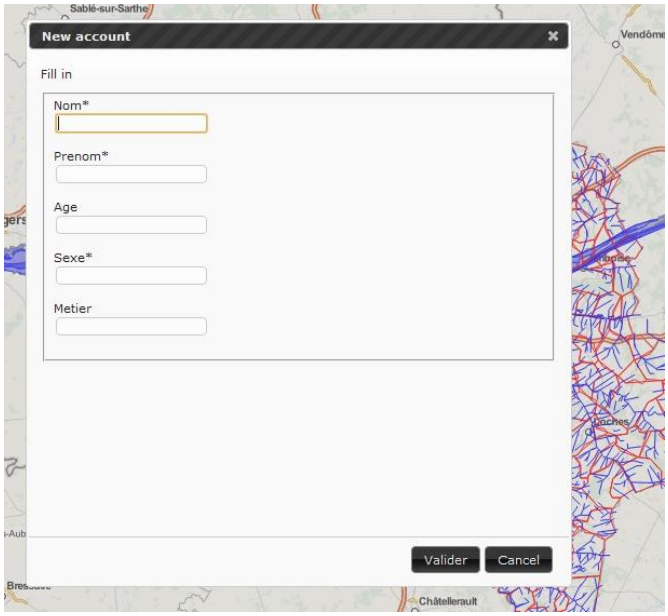

Figure 18 : Sauvegarde du thème

8 Le planning

8.1.1 Planning prévisionnel

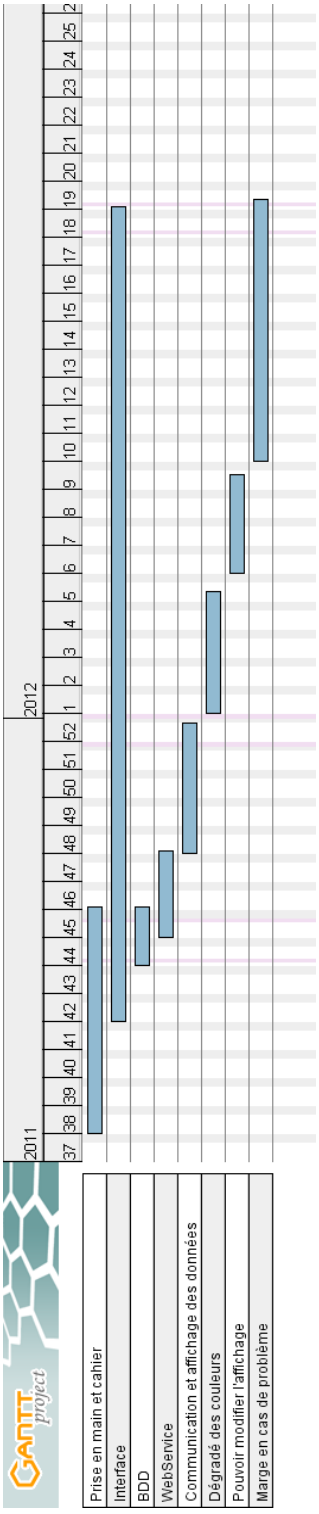

Figure 19 : Planning

8.1.2 Planning réel

Les délais établis lors du planning prévisionnel ont été respectés. Néanmoins, le temps " marge en cas de problème " a été utilisé pour développer l'algorithme génétique et pour la sauvegarde des données qui n'étaient pas prévue dans le début du projet.

Conclusion

Ce projet de fin d'étude m'a beaucoup appris. En effet, durant mes 3 années de formation, je n'avais jamais réalisé de projet aussi important en JEE. Outre le JEE, ce projet m'a permis de découvrir le javascript avec jQuery, le JSON et postGreSQL. Je suis maintenant capable de faire communiquer des langages grâce au JSON. Je peux également réaliser des applications JEE avec des outils utilisés par des grandes SSII tels que Maven et Hibernate.

Par ailleurs, grâce à ce projet, j'ai également acquis de nouvelles connaissances en géographie. Je connais maintenant quelques termes spécifiques utilisés dans ce domaine, les différentes représentations cartographiques, ainsi que les problèmes et contraintes liés au respect des très nombreuses normes.

Table des illustrations

Références

- http://softlibre.gloobe.org/openlayers/workshop/introduction/module1
- http://www.postgis.fr/chrome/site/docs/workshop-foss4g/doc/loading_data.html
- http://blog.le-guevel.com/?p=56
- http://www.hibernatespatial.org/tutorial
- http://www.objis.com/formation-java/tutoriel-web-services-integration-cxf-spring-maven.html
- http://www.hibernatespatial.org/usage.html
- http://www.laliluna.de/articles/java-persistence-hibernate/first-hibernate-example-tutorial.html
- http://java.sun.com/developer/technicalArticles/Programming/serialization/
- http://www.myeclipseide.com/documentation/quickstarts/webservices_rest/
- http://www.vogella.com/articles/REST/article.html
- http://srikanthtechnologies.com/blog/java/rest_service_client.aspx
- http://gis.ibbeck.de/
- http://dev.openlayers.org/releases/OpenLayers-2.11/examples/geojson.html
- http://jsonlint.com/
- http://jscolor.com/try.php#colors
- http://openlayers.org/dev/examples/styles-context.html
- http://dev.openlayers.org/docs/files/OpenLayers/Map-js.html
- http://jqueryui.com/

Résumé

Mon projet consiste à réaliser une application web permettant de visualiser Tours et sa région en cas d'inondation. L'utilisateur doit pouvoir modifier les couleurs des données d'inondations comme il le souhaite. Un algorithme génétique a été implémenté pour faciliter ses choix de couleur. Enfin, l'utilisateur a la possibilité de stocker les couleurs qu'il aura choisies dans la base de données avec ses informations personnelles.

Dans ce rapport, je présente toutes les technologies et l'architecture utilisées. Je décris également les outils qui m'ont été utiles durant mon développement.

Je présente ensuite ce qui a été réalisé durant mon PFE au niveau de la base de données, du webserveur et enfin du client.

Mots clés

Application web, cartographie, java EE, jBoss, Maven, Hibernate, jQuery, openLayers