

## Séance 6

# Interface avec le moteur NoSQL



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

# Objectifs

- Mongoose : un **ODM** pour MongoDB
  - ORM, ODM et schémas Mongoose
  - Validation des données
  - Les middlewares
- Développement d'une **API RESTful**
  - Web et API RESTful
  - Node.js et Express.js

# Mongoose



# ORM : Object-Relational Mapping

- Méthode de programmation consistant à faire correspondre une base de données **relationnelle** avec un modèle **orienté-objet**.
  - Une classe correspond à une table de la DB
  - Une instance de la classe correspond à un record d'une table
  - Un attribut de la classe correspond à un champ dans la table
- Résultat : Une **base de données "orienté-objet" virtuelle**, manipulable facilement depuis le code applicatif.
- **Difficulté** : Les objets sont des entités plus complexes que les records en DB
  - Exemple : Une personne possède une liste de numéros de téléphone*

# ORM : Pros/Cons

- Avantages :
  - L'ORM permet de rester dans le paradigme orienté-objet
  - SQL plus nécessaire pour la manipulation des données
  - Gestion de la connexion à la DB gérée par l'ORM
  - Code structuré de manière cohérente
- Désavantages :
  - Abstraction de haut niveau dissimulant les spécificités de l'implémentation en D
  - Tendance à moins bien concevoir le schéma de la DB
  - Performances moindres pour les requêtes complexes

Problème de l'impédance objet/relation !

# ORM : Que choisir ?

Il existe des frameworks ORMs pour beaucoup de langages de programmation :

- Java : Hibernate, Java Data Objects, Apache Cayenne, Enterprise JavaBeans, ...
- .NET : Entity Framework, XPO, NHibernate, Dapper, DataObjects.NET, Neo, ...
- PHP : Doctrine, CodeIgniter, CakePHP, Laravel, ...
- Python : Django, Storm, SQLAlchemy, ...
- Ruby on Rails : ActiveRecord

# ORM : Exemple (Django)

```
1 class Timesheet(models.Model) :
2     day = models.DateField("Date")
3     num_hours = models.DecimalField("Nombre d'heures prestées",
4                                     max_digits=4, decimal_places=2)
5     study=models.ForeignKey(Study, verbose_name="Etude")
6     operation=models.ForeignKey(Operation, blank=True, null=True,
7                                 verbose_name="Opération")
8     task=models.ForeignKey(Task, blank=True, null=True,
9                             verbose_name="Opération générique")
10    person = models.ForeignKey(Person, verbose_name="par")
11    comment=models.TextField("Commentaires", default="", null=True,
12                              blank=True)
13    def cost(self):
14        return self.num_hours * self.person.cout_horaire_on(self.
15            day)
16    def __str__(self):
17        return str(self.day)+str(" ") +str(self.num_hours)
```



# ORM : Exemple (Django)

```
1 ts=Timesheet.objects.filter(person=person_id, day=form.cleaned_data
  ["day"],
2                                     study=form.cleaned_data["study"], \
3                                     operation=form.cleaned_data["operation"],
4                                     task=form.cleaned_data["task"])
4 if len(ts)>0 :
5     ts[0].num_hours=form.cleaned_data["num_hours"]
6     ts[0].comment=form.cleaned_data["comment"]
7     ts[0].save()
```

# ODM : Object-Document Mapper

- De manière similaire aux ORM, les ODM font le lien entre du code applicatif orienté-objet et une base de données orientée-document.
- A la différence des ORMs, les ODMs ne souffrent pas du problème d'impédance objet-relation, puisque les documents ont une structure très proche de celle des objets.

# ODM : Pros/Cons

- Avantages :
  - Permet d'appliquer un schéma structuré à des collections
  - Permet la conversion automatique des données dans les types natifs du langage
  - Gestion facilitée des opérations CRUD
  - Permet d'ajouter une couche de validation des données avant insertion dans la DB
  - Des middlewares permettent d'appliquer des traitements logiques (business) aux données
  - La couche supplémentaire permet de simplifier la communication avec la DB
- Inconvénients :
  - L'imposition d'un schéma aux documents n'est pas toujours la meilleure option
  - Certaines opérations sont moins efficaces qu'en cas de communication directe avec la DB.

# Mongoose.js : Un ODM pour MongoDB

- Mongoose est une bibliothèque Node.js facilitant l'interaction avec une base de données MongoDB
- Souvent reprise dans les stacks Web Javascript (ex : Mean.js)
- Travaille au dessus du driver Node.js de MongoDB, en ajoutant plusieurs objets :
  - Schema : Permet de définir un schéma structuré pour les documents d'une collection
  - Model : Représentation de tous les documents d'une collection
  - Document : Représentation des documents individuels
  - Query et Aggregate : Permettent de construire des requêtes complexes et réutilisables.

# Mongoose.js : Définition d'un schéma

```
1
2  /**
3   * Measure Schema
4   */
5  var MeasureSchema = new Schema({
6    date : {
7      type : Date,
8      required : 'Please specify a date for this measure',
9      default : Date.now
10   },
11   value : {
12     type : Number,
13     required : 'Please specify the value of your measure',
14     min : '1',
15   },
16   meter : {
17     type : Schema.ObjectId,
18     ref : 'Meter',
19     required : true
20   }
21 }
22 )
23 mongoose.model('Measure', MeasureSchema);
```

# Mongoose.js : Création d'un document

Pour créer un document, deux possibilités :

- Instanciation d'un document puis appel de la méthode `save()`
- Utilisation de `create()` sur le modèle

```
1 var mongoose = require('mongoose'),
2     Measure = mongoose.model('Measure');
3
4 var measure = new Measure();
5 measure.date = Date.now();
6 measure.value = req.body.value;
7 measure.meter = req.body.meter;
8 measure.save(function(err){
9     if(err) console.log("Erreur");
10    });
11 //OU :
12 Measure.create([{'date':Date.now(), value: req.body.value, meter: req
    .body.meter}]);
```

# Mongoose.js : Opérations sur les collections

Les opérations Mongoose de type CRUD sur les modèles sont similaires aux fonctions natives MongoDB sur les collections : `find()`, `findOne()`, `update()`, `remove()`.

Exemples d'utilisation de `find()` sur l'objet Mongoose Model :

```
1  Measure.find({ 'owner':req.user }).sort('-created').populate('
2  owner', 'displayName').exec(function (err, measures) {//Results
3  in var measures});
4  //OU :
5  var query = Measure.find({ 'owner':req.user });
6  query.sort('-created');
7  query.populate('owner', 'displayName');
8  query.exec(function(err, measures){//Results in var measures})
9  ;
10 // Autre requête :
    Measure.findOne({}).where('value').gt(100).sort('value').
    select('date').exec(function(err, measures){//Treatment});
```

# Mongoose.js : Modificateurs de schéma

Lors de la définition du schéma, il est possible de vérifier ou modifier des valeurs avant sauvegarde. Mongoose définit plusieurs modificateurs à cette fin :

- `trim` : permet de retirer les espaces en début et fin de champs String
- `set/get` : permet de définir une fonction appliquée à l'édition ou à la consultation d'un champ

```
1 website: {
2   type: String,
3   set: function(url) {
4     if (!url) {
5       return url;
6     } else {
7       if (url.indexOf('http://') !== 0 && url.indexOf('https://') !== 0) {
8         url = 'http://' + url;
9       }
10      return url;
11    }
12  },
```



# Mongoose.js : Attribut virtuel

Mongoose permet l'ajout d'*attributs virtuels* à un schéma MongoDB. Il s'agit d'attributs calculés dynamiquement à partir des données de la DB.

Dans cet exemple, on crée un champ `fullname`, au départ des attributs `firstname` et `lastname` :

```
1  UserSchema.virtual('fullName')
2    .get(function() {
3      return this.firstName + ' ' + this.lastName;
4    })
5    .set(function(fullName) {
6      var splitName = fullName.split(' ');
7      this.firstName = splitName[0] || '';
8      this.lastName = splitName[1] || '';
9    });
10
11 UserSchema.set('toJSON', { getters: true, virtuals: true });
```

# Mongoose.js : Ajout d'index

Un index primaire peut être défini grâce à l'attribut unique :

```
1  username: {
2    type: String,
3    unique: 'Username already exists',
4    required: 'Please fill in a username',
5    lowercase: true,
6    trim: true
7  },
```

Si d'autres attributs sont destinés à être souvent consultés, on peut créer des index secondaires pour accélérer ces requêtes :

```
1  email: {
2    type: String,
3    index: true
4  },
```

Pour créer des index secondaires sur plusieurs champs :

```
1  MeasureSchema.index({ date: 1, meter: 1},{unique: 'Only one measure
   per day is supported'});
```

# Mongoose.js : Validation des données (1)

Mongoose offre la possibilité de définir des validations sur les champs à la création, lecture ou sauvegarde d'un document.

- Dans la définition du schéma, avec des attributs spécifiques pour les champs visés :
  - `unique` : La valeur du champ doit être unique sur la collection
  - `required` : Champs obligatoire
  - `match` : Spécifie une regex pour contraindre le format d'un string (ex : email)
  - `enum` : Fournit une liste de valeurs acceptables pour un attribut String.

```
1 var MeasureSchema = new Schema({
2   //...
3   value : {
4     type : Number,
5     required : 'Please specify the value of your measure',
6     min : '1',
7   },
8   //...
9 });
```

## Mongoose.js : Validation des données (2)

Mongoose offre la possibilité de définir des validations sur les champs à la création, lecture ou sauvegarde d'un document.

- En définissant l'attribut `validate` sur un champs, en spécifiant un tableau avec une fonction de validation et un message d'erreur :

```
1 var UserSchema = new Schema({
2   ...
3   password: {
4     type: String,
5     validate: [
6       function(password) {
7         return password.length >= 6;
8       },
9       'Password should be longer'
10    ]
11  },
12  ... });
```

# Mongoose.js : Middleware

Dans certains cas, il est utile de pouvoir indiquer un traitement spécifique à appliquer avant ou après une opération sur la base de données. Mongoose fournit pour cela des `middlewares` de deux types : `pre` et `post`. Ils peuvent s'appliquer sur les opérations suivantes :

- `init`
- `validate`
- `save`
- `remove`

# Mongoose.js : Middleware - Exemple

```
1
2 UserSchema.pre('validate', function (next) {
3   if (this.provider === 'local' && this.password && this.isModified
4     ('password')) {
5     var result = owasp.test(this.password);
6     if (result.errors.length) {
7       var error = result.errors.join(' ');
8       this.invalidate('password', error);
9     }
10  }
11  next();
12 });
```

```
1 UserSchema.post('save', function(next) {
2   if(this.isNew) {
3     console.log('A new user was created.');
```

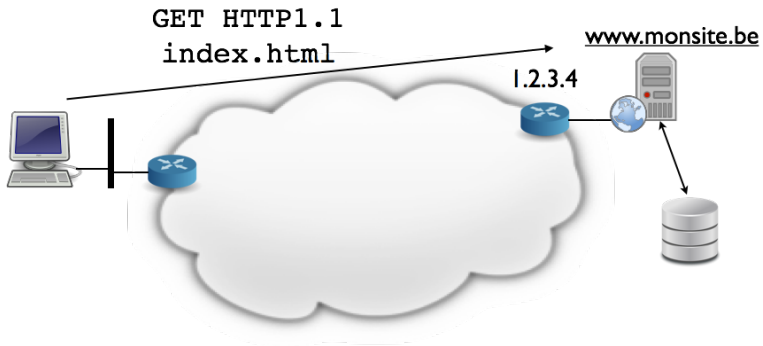
```
4   } else {
5     console.log('A user updated is details.');
```

```
6   }
7 });
```

A close-up photograph of a spider web, illuminated from above by a warm, golden light source. The web's intricate spiral and radial patterns are highlighted, creating a complex, glowing structure against a dark background. The light creates a gradient from bright yellow at the top to deep orange and red towards the bottom. The web is stretched across a dark, possibly black, surface, with some of its threads appearing slightly blurred due to a shallow depth of field.

**Le Web**

# Architecture du Web





# Le protocole HTTP

- HyperText Transport Protocol : Protocole en mode texte orienté client/serveur Style d'architecture pour les systèmes hypermedia distribués
- Requête HTTP en trois parties :
  - 1 Méthode + URI + version
  - 2 En-tête avec paramètres optionnels + ligne vide
  - 3 Document MIME (optionnel)
- Réponse HTTP en trois parties :
  - 1 Ligne de statut (+ code)
  - 2 En-tête avec informations sur la réponse + ligne vide
  - 3 Document MIME (optionnel)

# Un échange HTTP

La requête :

```
GET / HTTP/1.0
User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl
/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
Host: www.ietf.org
```

La réponse :

```
HTTP/1.1 200 OK
Date: Mon, 15 Mar 2010 13:40:38 GMT
Server: Apache/2.2.4 (Linux/SUSE) mod_ssl/2.2.4 OpenSSL/0.9.8
e (truncated)
Last-Modified: Tue, 09 Mar 2010 21:26:53 GMT
Content-Length: 17019
Content-Type: text/html

<!DOCTYPE HTML PUBLIC .../HTML>
```

# API RESTful

- REST : Representational State Transfer
  - Style d'architecture pour les systèmes hypermedia distribués
  - Proposé par R. Fielding dans sa thèse en 2000
    - REST a été utilisé par Fielding pour concevoir HTTP/1.1*
  - Indépendant du protocole utilisé
  - Orienté **ressources**
- REST a été défini dans le but de fournir certaines propriétés aux systèmes concernés
  - Performance, montée en charge, simplicité de l'interface, évolutivité des composants, portabilité, fiabilité, visibilité*
- Définition d'un ensemble de contraintes pour les systèmes RESTful

# Contraintes REST (1)

- Architecture **Client-Server**

*Séparation des préoccupations/rôles*

- **Stateless**

*Toutes les informations nécessaires à la requête sont contenues dans cette dernière, sans stockage d'information*

- Possibilité de **mise en cache**

*L'architecture doit supporter la mise en cache au niveau de systèmes intermédiaires*

- Système en **couches**

*Intégration d'équipements intermédiaires permettant la répartition des rôles et de la charge, de manière transparente pour le client*

# Contraintes REST (2)

- Interface **uniforme**
  - Identification des ressources (ex : URI pour le Web)
  - Manipulation des ressources à travers des représentations
  - Messages auto-descriptifs
  - HATEOAS : Hypermedia As The Engine Of Application State  
*Les réponses contiennent des liens vers les autres ressources ou actions disponibles*
- Code **à la demande** (optionnel)  
*Possibilité de transférer du code au client pour enrichir ses fonctionnalités (ex : applets java, javascript)*

# REST : Nommage des ressources et URI

- Utiliser des noms pour désigner les ressources, pas de verbes
- Dans le cas de collections, utiliser le nom au pluriel
- Utiliser des tirets (-) et non des underscores \_
- Pas de CRUD dans les URI (utiliser les verbes HTTP à la place) *Pour les actions plus complexes : Utiliser les query strings*

Exemples :

```
GET /api/v1/books/  
GET /api/v1/books/harry-potter-2  
PUT /api/v1/books/harry-potter-2  
POST /api/v1/books/harry-potter-2/comments  
DELETE /api/v1/books/harry-potter-2/comments/123  
PUT /api/v1/blogposts/12342?action=like  
GET /api/v1/books?limit=1&sort=created_at
```

Node.js



me's JavaScript runtime for e

# Javascript : les fonctions comme objets

Les fonctions Javascript sont des fonctions d'ordre supérieur :

- Elles peuvent être référencées comme objets
- Elles peuvent être passées en paramètres (callback)
- Elles peuvent être renvoyées

```
1 var add = function(a,b) {  
2     return a+b;  
3 }  
4 add(2,3);
```

```
1 exports.delete = function (req, res) {  
2     var building = req.building;  
3     building.remove(function (err) {  
4         if (err) { return res.status(400).send({  
5             message: errorHandler.getErrorMessage(err)  
6             });  
7         } else { res.json(building); }  
8     });  
};
```



# Javascript : Closure

Une fermeture (closure en anglais) est une fonction associée à l'environnement dans laquelle elle a été définie (variables locales définies dans la portée englobante)

```
1  function creerFonction() {
2      var nom = "Mr. Combéfis";
3      function afficheNom() {
4          alert(nom);
5      }
6      return afficheNom;
7  }
8
9  var maFonction = creerFonction();
10 maFonction();
```

# Javascript : Côté client

A l'origine, Javascript était essentiellement exécuté à l'intérieur des navigateurs Web

- **Manipulation du DOM** (Document Object Model) pour modifier le contenu ou l'apparence d'une page Web et traiter les événements utilisateurs

```
1 $(document).ready(function(){
2     $("div.test").add("p.quote")
3         .addClass("blue")
4 }
```

- AJAX pour les interactions asynchrones avec le serveur
- JQuery, AngularJS, ...

# Javascript : Node.js côté serveur

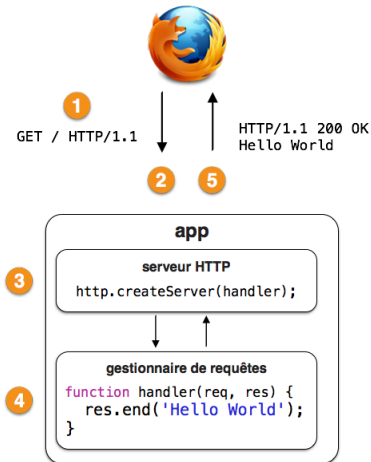
Actuellement, retour de Javascript au niveau des serveurs Web avec Node.js : un environnement de développement multi-plateforme pour développer des applications Web côté serveur

- Architecture event-driven, API I/O non bloquante  
*Programmation asynchrone, utilisation de callbacks. Pas besoin de multi-thread*
- Utilise le moteur javascript v8 de Google
- Gestionnaire de package NPM  
*Beaucoup de librairies Node disponibles*

# Node.js

```
1 var http = require('http')
2 var server = http.createServer(function (request, response){
3     response.writeHead(200, {'Content-type': 'text/plain'});
4     response.write('Hello World!\n');
5     response.end();
6 });
7 server.listen(8000);
8 console.log("Server running at http://localhost:8000");
```

# Node.js : fonctionnement



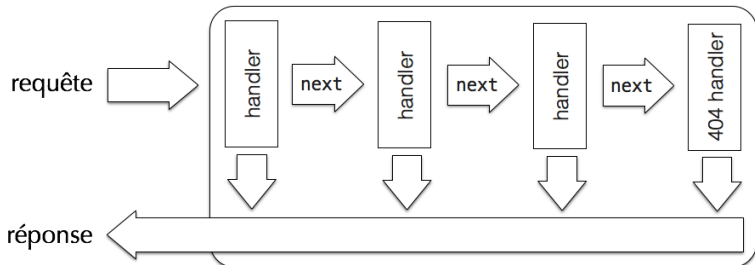
- Librairie permettant d'écrire plus facilement des serveur HTTP
- Objet `request` représentant la requête HTTP enrichi :  
`request.protocol`, `request.host`, `request.port`,  
`request.path`, `request.query`, ...
- Objet `response` représentant la réponse enrichi :  
`response.send()` *calcule le Content-Length automatiquement*

# Express.js : exemple

```
1 var express = require('express');
2
3 var app = express();
4
5 app.get('/',
6   function (request, response) {
7     response.send('Hello, World!');
8   }
9 );
10 app.listen(8080);
11 console.log('Express server listening on port 8080');
```

# Express.js : Middlewares

Pour chaque requête, on peut associer différents handlers, exécutés les uns après les autres. On les appelle des Middlewares.





# Node.js : Middlewares

- Gestion des requêtes en fonction du chemin d'accès
- Logger les requêtes
- Gestion des cookies
- Gestion des pages statiques
- Compression
- Authentification

# Express.js : Middlewares

```
1 var express = require('express');
2
3 var app = express();
4
5 var logger = function(req, res, next) {
6     console.log('%s %s', req.method, req.url);
7     next();
8 };
9
10 app.use(logger);
11 app.use(express.static(__dirname + '/content'));
12
13 app.listen(8080);
14
15 console.log('Express server listening on port 8080');
```

# Express.js : Routage des requêtes

La réponse à une requête doit être produite sur base :

- De la méthode HTTP utilisée

*app.get(), app.put(), app.delete(), app.post()*

- Du path de l'URL

*Utilisation de regex et récupération de marqueurs*

```
1 app.get('/abcd', function(req, res) {  
2   res.send('abcd');  
3 });
```

```
1 app.get('/users/:id/:section', function(req, res) {  
2   res.send(req.params.section + ' of user ' + req.params.id);  
3 });
```

# Express.js : Que renvoyer ?

On sait donc récupérer une requête et y répondre. Mais avec quoi ?

- JSON

*API type Web Service et/ou requête asynchrone d'une page web (AJAX)*

- HTML : Différentes options

- Fichier statique (cfr middleware static)
- HTML entièrement généré par du javascript
- Template HTML (= fichier "à trous") : *View*    *Choix d'un parseur pour remplir les trous : jade, swig, hogan, ...*

```
1 app.get('/abcd', function(req, res) {  
2   res.send('abcd');  
3 });
```

```
1 app.get('/users/:id/:section', function(req, res) {  
2   res.send(req.params.section + ' of user ' + req.params.id);  
3 });
```

# Express.js : Renvoyer une page HTML

## Exemple de réponse contenant une page HTML dynamique

```
1 var express = require('express'),
2   engines = require('consolidate');
3 var app = express();
4 app.set('view engine', 'html');
5 app.set('views', __dirname);
6 app.engine('html', engines.hogan);
7
8 app.get('/', function (req, res) {
9   res.render('express-views', { name: 'LSINF1212' } );
10 });
11 app.get('*', function (req, res) {
12   res.status(404).send('Page not found');
13 });
14
15 app.listen(8080);
16 console.log('Express server listening on port 8080');
```

## Template correspondant :

```
1 <h1>Hello {{name}}!</h1>
```

# Crédits

- <https://www.flickr.com/photos/arcticproductions/16825885078>
- [https://www.flickr.com/photos/tamis\\_haddad/15159148526](https://www.flickr.com/photos/tamis_haddad/15159148526)
- <https://www.flickr.com/photos/peterlozano/5718349563>