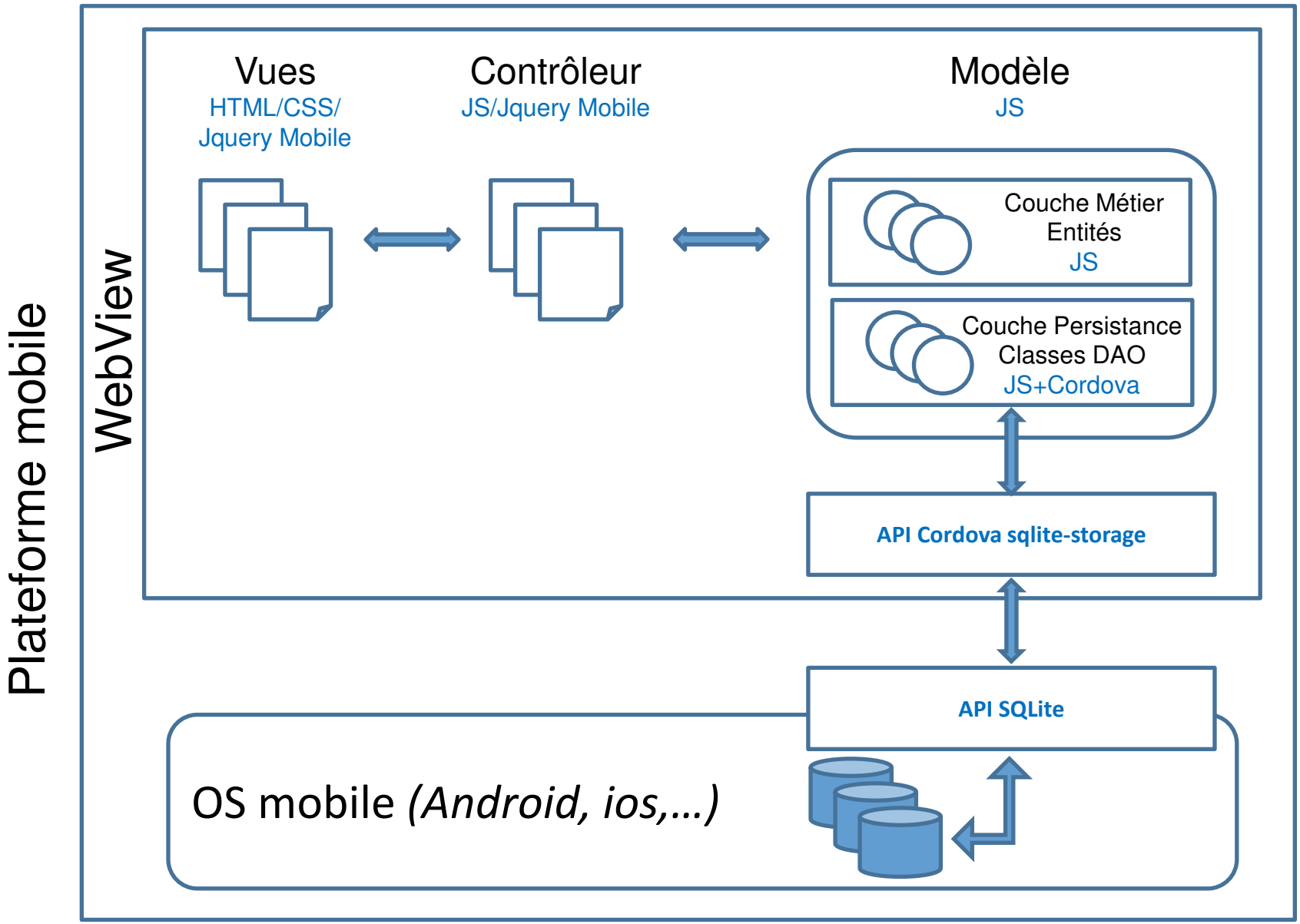
Licence Pro MI-AW

Module INFO-1

Cordova et SQLite

Architecture d'une application MVC avec Données Persistantes



API Cordova SQLite Storage

- **URL** : https://github.com/litehelpers/Cordova-sqlite-storage
- **Install** : **cordova plugin add cordova-sqlite-storage**
- Fournit des fonctions JS pour exécuter des requêtes SQLite
- Les fonctions de l'API SQL sont exécutées de manière **asynchrone :**
  - L'appel d'une fonction SQL dans votre code JS déclenche l'exécution de la requête qui est traitée de manière concurrente (en parallèle)
  - On ne sait pas combien de temps la requête prendra… et pendant ce temps votre code JS continue bien sûr à s'exécuter
  - Il faudra fournir à chaque fonction SQL deux fonctions « **callBack** » :
    - **successCB** : sera exécutée à la fin de la requête SQL si elle s'est achevée avec **succès**
    - **errorCB** : sera exécutée à la fin de la requête SQL si elle s'est terminée sur une **erreur**

SQLite Storage - openDataBase

- Les fonctions de l'API sont disponibles dans un objet global **sqlitePlugin**
- SQLite crée un fichier par base de données, dans le système de fichier de l'OS mobile
- **openDataBase(paramBD, succesCB, errorCB)**
  - À invoquer lorsque le *device* est *ready*
  - Retourne une objet « BD » qu'on utilisera ensuite pour exécuter des requêtes
  - Peut recevoir deux paramètres succesCB et errorCB (voir exemple plus loin)

```
var db = null;

document.addEventListener('deviceready', function() {
  db = window.sqlitePlugin.openDatabase(
         {name: 'demo.db', location: 'default'});
});
```

SQLite Storage - executeSql

- Exécution d'une requête SQLite sur une BD
- Syntaxe SQLite : https://www.sqlite.org/lang.html
- Tester les requêtes avant de les insérer dans le projet : http://sqlfiddle.com/
- **executeSql(requete, paramRequete, succesCB, errorCB)**
  - **requete :** une chaine qui contient une requête SQLite, avec d'éventuels paramètres (caractère ?)
  - **paramRequete** : un tableau contenant les valeurs des paramètres de la requête

```
db.executeSql('SELECT count(*) AS mycount FROM DemoTable', [],
   function(rs) {
     console.log('Record count (expected to be 2): ' + rs.rows.item(0).mycount);
   },
   function(error) {
     console.log('SELECT SQL statement ERROR: ' + error.message);
   });
```

successCB reçoit en paramètre un objet ResultSet (rs) qui contient le résultat de la requête lorsqu'elle est réussie (voir exemple plus loin)

errorCB reçoit en paramètre un objet Error qui contient la description de l'erreur provoquée par la requête

SQLite Storage - transaction

- Il est possible de regrouper plusieurs requêtes au sein d'une transaction
- **transaction(requetesCB, errorCB, successCB)**
  - requetesCB : fonction qui contient les requêtes composant la transaction
  - errorCB : fonction exécutée si la transaction a échoué
  - successCB : fonction exécutée à la fin de la transaction si elle a réussi

```
db.transaction(
  function(tx) {
    tx.executeSql('CREATE TABLE IF NOT EXISTS DemoTable (name,score)');
    tx.executeSql('INSERT INTO DemoTable VALUES (?,?)', ['Alice',101]);
    tx.executeSql('INSERT INTO DemoTable VALUES (?,?)', ['Betty',202]);
  },
  function(error) {
    console.log('Transaction ERROR: ' + error.message);
  },
  function() {
    console.log('Populated database OK');
  }
);
```

SQLite Storage - batch

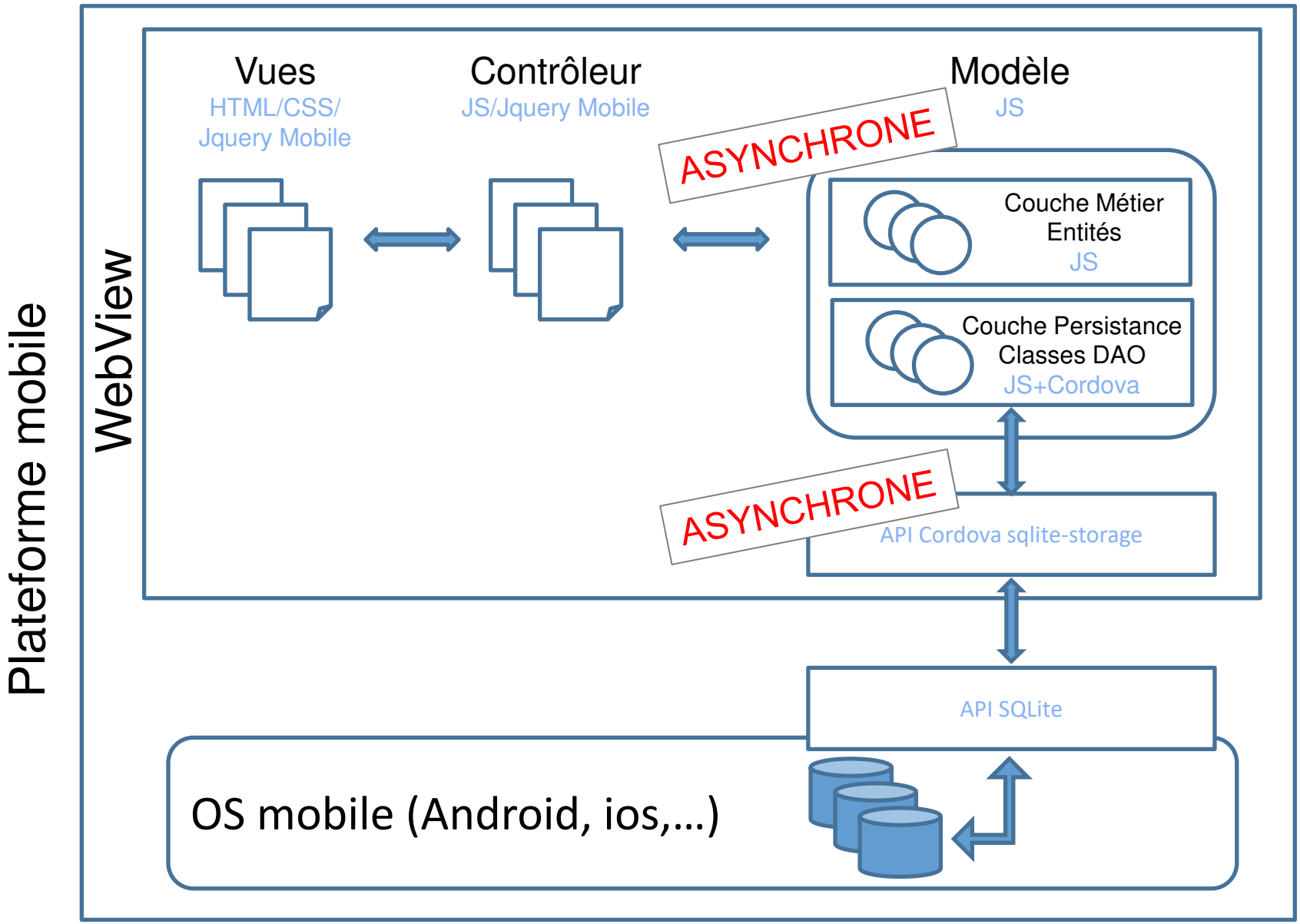
- Il est possible d'enchaîner plusieurs requêtes dans un lot (*batch*)
  - les requêtes seront exécutées les unes après les autres
- **sqlBatch(lesRequetes, successCB, errorCB)**
  - **lesRequetes** : tableau de requêtes avec leurs paramètres
  - successCB, errorCB : usage habituel

```
db.sqlBatch(
   ['CREATE TABLE IF NOT EXISTS DemoTable (name, score)',
     [ 'INSERT INTO DemoTable VALUES (?,?)', ['Alice', 101] ],
     [ 'INSERT INTO DemoTable VALUES (?,?)', ['Betty', 202] ],
   ],
   function() {
     console.log('Populated database OK');
   },
   function(error) {
     console.log('SQL batch ERROR: ' + error.message);
   }
 );
```

Programmation Asynchrone et MVC

- Dans un modèle de programmation **séquentielle** (ex PHP), **le contrôleur :**
  - Invoque une opération sur le modèle
  - Attend que le modèle ait réalisé cette opération
  - Puis poursuit son travail en fonction du résultat de l'opération renvoyé par le modèle
- Dans un modèle de programmation **asynchrone**, le **contrôleur**, lorsqu'il invoque une opération sur le modèle, ne sait pas quand cette opération se terminera…
- Pour conserver l'architecture **MVC**, il faut donc que le contrôleur, lorsqu'il invoque une opération asynchrone sur le modèle, transmette à celui-ci le « code contrôleur » à exécuter lorsque l'opération sera terminée
- La **couche persistance** de notre modèle devra donc elle aussi offrir une **interface asynchrone**, avec des fonctions ayant des paramètres « callBack » :
  - **successCtrlCB** : code du contrôleur à exécuter lorsque l'opération s'est terminée avec succès
  - **errorCtrlCB** : code du contrôleur à exécuter lorsque l'opération s'est terminée sur une erreur

Architecture d'une application MVC avec Données Persistantes



Exemple MVC : Cordova+SQL

- Une application pour saisir/stocker des notes : DemoNoteSql
- Les Vues : 4 pages… code HTML simple (voir projet fourni).
- Utilise le plugin Toast pour afficher des « alertes » :
  cordova plugin add cordova-plugin-x-toast

#accueil

#addNote

#listNote

#detailNote

Le Modèle (1)
model.init, model.Note

```
var model = {
    db: null // objet pour manipuler la BD SQLite
};

// Initialisation du modèle : ouverture de la BD
// successCtrlCB : méthode du contrôleur appelée si BD prête
// errorCtrlCB   : méthode du contrôleur appelée si problème lors de l'ouverture
model.init = function (successCtrlCB, errorCtrlCB) {
    this.db = sqlitePlugin.openDatabase({name: "notes.db", location: "default"},
            function () { // si succes
                var succCB = successCtrlCB; // pbm de visibilité de successCtrlCB ci-dessous
                var errCB  = errorCtrlCB;   // idem
                model.db.executeSql("CREATE TABLE IF NOT EXISTS
                                     note (id integer primary key, titre text, texte text)", [],
                        function () {
                            succCB.call(this); // succesCtrlCB.call(this) n'aurait pas fonctionné
                        },
                        function () {
                            errCB.call(this);
                        });
            },
            function () { // si erreur
                errorCtrlCB.call(this);
            });
};

// Définition de l'Entité Note
model.Note = function (id, titre, texte) {
    this.id = id;
    this.titre = titre;
    this.texte = texte;
};
```

Le Modèle (2)
model.NoteDAO insert, findALL

```
model.NoteDAO = {
  // Requête pour insérer une nouvelle note en BD
  // successCtrlCB : méthode du contrôleur appelée en cas de succès
  // errorCtrlCB   : méthode du contrôleur appelée en cas d'échec
  insert: function (uneNote, successCtrlCB, errorCtrlCB) {
    var laNote = uneNote; // pour rendre la note visible dans la CB ci-dessous
    model.db.executeSql("INSERT INTO note (titre, texte) VALUES (?,?)",
      [uneNote.titre, uneNote.texte],
      function (res) { // succes
        laNote.id = res.insertId; // on met à jour l'id de la note après insertion en BD
        successCtrlCB.call(this); // et on appelle la successCtrlCB en provenance du contrôleur
      },
      function (err) { // erreur
        errorCtrlCB.call(this); // on appelle l'errorCtrlCB en provenance du contrôleur
      }
    );
  },
  // Requête pour récupérer toutes les notes
  // successCtrlCB recevra en paramètre le tableau de toutes les entités Note
  findAll: function (successCtrlCB, errorCtrlCB) {
    model.db.executeSql("SELECT * FROM note", [],
      function (res) { // succes
        var lesNotes = [];
        for (var i = 0; i < res.rows.length; i++) {
          var uneNote = new model.Note(res.rows.item(i).id, res.rows.item(i).titre,
                                                            res.rows.item(i).texte);
          lesNotes.push(uneNote);
        }
        successCtrlCB.call(this, lesNotes);
      },
      function (err) { // erreur
        errorCtrlCB.call(this);
      }
    );
  },
```

Le Modèle (3)
model.NoteDAO findById & removeById

```
  // Requête pour récupérer une note selon son id
  // successCtrlCB recevra en paramètre l'entité Note
  findById: function (id, successCtrlCB, errorCtrlCB) {
    model.db.executeSql("SELECT * FROM note WHERE id = ?", [id],
      function (res) { // success
        var uneNote = new model.Note(res.rows.item(0).id,
                                     res.rows.item(0).titre, res.rows.item(0).texte);
        successCtrlCB.call(this, uneNote);
      },
      function (err) { // erreur
        errorCtrlCB.call(this);
      }
    );
  },

  // Requête pour supprimer une note selon son id
  removeById: function (id, successCtrlCB, errorCtrlCB) {
    model.db.executeSql("DELETE FROM note WHERE id = ?", [id],
      function (res) { //succes
        successCtrlCB.call(this);
      },
      function (err) { // erreur
        errorCtrlCB.call(this);
      }
    );
  }

}; // Fin model.NoteDAO
```

Le Contrôleur (1) controller.init

```
var controller = {};
////////////////////////////////////////////////////////////////////////
// Controleurs : 1 contrôleur par page, qui porte le nom de la page avec le suffixe Controller
////////////////////////////////////////////////////////////////////////

controller.init = function () { // Méthode init appelée au lancement de l'app.
  // Ouverture de la BD. et création si besoin des tables
  model.init(
    function () { // successCB : Si BD prête, on va à la page d'accueil
      // On duplique Header et Footer sur chaque page (sauf la première !)
      $('div[data-role="page"]').each(function (i) {
        if (i)
          $(this).html($('#Header').html() + $(this).html() + $('#Footer').html());
      });
      // On afficher la page d'accueil
      $.mobile.changePage("#accueil");
    },
    function () { // errorCB : Si problème, on va à la page d'erreur
      $.mobile.changePage("#erreur");
    });
};
```

Le Contrôleur (2)
controller.addNote.save

```
////////////////////////////////////////////////////////////////////////////
// Contrôleur de la vue addNote
////////////////////////////////////////////////////////////////////////////
controller.addNote = {
  save: function () {
    // "validation" du formulaire
    var titre = $("#addNoteTitre").val();
    var texte = $("#addNoteTexte").val();
    if (titre === "") {
      plugins.toast.showShortCenter("Entrez un Titre SVP");
      return;
    }
    if (texte === "") {
      plugins.toast.showShortCenter("Entrez un Texte svp");
      return;
    }
    // Interaction avec le modèle
    var newNote = new model.Note(0, titre, texte); // on crée une entité
    model.NoteDAO.insert(newNote, // puis on essaye de la sauver en BD
      function () { // successCB
        plugins.toast.showShortCenter('Note ' + newNote.id + ' Enregistrée');
        $("#addNoteTitre").val("");
        $("#addNoteTexte").val("");
        $.mobile.changePage("#accueil");
      },
      function () { // errorCB
        plugins.toast.showShortCenter("Erreur : note non enregistrée");
      }
    );
  }
};
```

Le Contrôleur (3) controller.listNote.fillListView

```
////////////////////////////////////////////////////////////////////////////
// Contrôleur de la vue listNote
////////////////////////////////////////////////////////////////////////////
controller.listNote = {
  // Définit le contenu de la listView
  fillListView: function () {
    model.NoteDAO.findAll(
      function (lesNotes) { // successCB
        $("#listNoteContenu").empty();
        for (var i = 0; i < lesNotes.length; i++) {
          var liElement = $("<li></li>").data("noteId", lesNotes[i].id);
          var aElement = $("<a></a>")
            .data("noteId", lesNotes[i].id)
            .text(lesNotes[i].id + ". " + lesNotes[i].titre);
          // un click sur une note permet d'en afficher le détail
          liElement.on("click", function () {
            controller.listNote.goToDetailNote($(this));
          });
          // un swipe sur une note permet de la supprimer
          liElement.on("swipe", function () {
            controller.listNote.removeNote($(this));
          });
          $("#listNoteContenu").append(liElement.append(aElement));
        }
        $("#listNoteContenu").listview("refresh");
      },
      function () { // errorCB
        $("#listNoteContenu").html("<li>Pas de Notes</li>");
        $("#listNoteContenu").listview("refresh");
      });
  },
```

Le Contrôleur (4)
controller.listNote goToDetailNote, removeNote

```
  // Définit le contenu de la vue detailNote en fonction de la note cliquée
  goToDetailNote: function (listViewElement) {
    var noteId = listViewElement.data("noteId");
    model.NoteDAO.findById(noteId,
      function (uneNote) { // successCB
        $("#oneNoteId").html(uneNote.id);
        $("#oneNoteTitre").html(uneNote.titre);
        $("#oneNoteTexte").html(uneNote.texte);
        $.mobile.changePage("#detailNote");
      },
      function () { // errorCB
        plugins.toast.showShortCenter("Note non disponible");
      }
    );
  },
  // Supprime la note balayée et met à jour la listView
  removeNote: function (listViewElement) {
    var noteId = listViewElement.data("noteId");
    model.NoteDAO.removeById(noteId,
      function () { // successCB
        plugins.toast.showShortCenter("Note " + noteId + " supprimée");
        listViewElement.remove();
      },
      function () { // errorCB
        plugins.toast.showShortCenter("Note " + noteId + " non supprimée");
      }
    );
  }
}; // fin controller.listNote
// Pour initialiser la listView quand on arrive sur la vue
$(document).on("pagebeforeshow", "#listNote", function () {
  controller.listNote.fillListView();
});
```