Waterline Reference

작성자 : 양동호 (Rex)

https://github.com/balderdashy/waterline

<워터라인이란 무엇인가? >

- Waterline은 저장 및 검색 엔진의 새로운 종류이다.

- Waterline은 다른 종류의 DB, protocol, 3rd party APIs 에 access할수 있는 일관된 API를 제공한다.

- 그것이 의미하는 바는 동일한 코드를 작성함으로써 얻을 수 있다는 것이다. Redis 던지 mySQL이던지 LDAP이던지 MongoDB던지 Postgres 던지 말이다.

- 워터라인은 ActiveRecord, Hibernate, 그리고 Mongoose 같은 최고의 ORM들을 상속하기 위해 노력하지만, 새로운 관점과 모듈, 테스트 용이성 및 어댑터를 통한 일관성을 강조한다.

< Sails.js 와 함께 사용하기 >

워터라인은 sails.js 프레임워크 (http://sailsjs.org/)에서 추출되었고, 그것은 sails.js에서 사용하던 기본 ORM(객체 관계 맵핑)이다.

워터라인을 사용함에 있어 더 많은 정보는, 당신의 Sails App 에서 Sails Docs를 보아라.

< 독립된 사용법 >

```
var Waterline = require('waterline');

// Define your collection (aka model)
var User = Waterline.Collection.extend({

  attributes: {

    firstName: {
      type: 'string',
      required: true
    },

    lastName: {
      type: 'string',
      required: true
    }
  }
});
```

위의 소스코드는 Collection(Table)을 생성하는 소스코드로 보여진다. Collection을 생성하고, Attributes 라는 key에 Object를 value로 넣어, 각 컬럼들을 설정한다.

그 컬럼은 firstName과 lastName이라는 컬럼이며, 해당 컬럼에는 type(varchar2등)과 required(not null) 을 설정할 수 있는 옵션 Object를 value로서 집어넣는다.

기본적으로 Waterline.Collection.extend({ }); 라는 사용법으로 보아, MySQL 이나 그런 RDBMS를 NoSQL의 방법으로 접근할 수 있도록 해주는 듯 하다.

< 개요 (Overview) >

- 어댑터의 개념

워터라인은 DataBase 에 의해 각 DB가 이해하는 문법(질의)를 변환할 수 있는 어댑터의 개념을 사용한다.

어댑터는 데이터작업을 위해 다양한 저장소들 (MySQL, MongoDB 등) 에 대해서 명확하고 동일한 API를 사용할 수 있다.

그리고 반드시 워터라인에서 기본적으로 정의된 CRUD 방법이 아니라, 자신의 방법을 정의하는 어댑터를 사용할 수 있다. 그러나, 사용자 정의 어댑터(메소드)를 정의하면, 워터라인은 단순히 어댑터까지 함수 인자를 전달할 뿐이며, 더 이상 워터라인의 고유 기능이 사용되지 않는다. 더 이상 Lifecycle CallBacks 그리고 Validations 를 얻을 수는 없다.

  * 여기서 Lifecycle CallBacks 란? 안드로이드의 Lifecycle Callbacks와 같은 의미이다. onActivitPaused() 등 before와 after로 나누어지는데, Validate, Update, Create, Destroy 이렇게 총 8가지의 콜백이 존재한다. beforeValidate / afterValidate …

그리고 어댑터의 배열을 제공할 수 있다. 어댑터의 배열의 마지막 항목이, 앞의 어댑터의 모든 메소드를 오버라이딩(overriding) 을 하는 것과 유사한 작동을 한다.

커뮤니티 어댑터가 존재한다.

< Collection >

컬렉션은 워터라인에서 사용되는 “주 객체” 이다. 그것은 데이터의 validations와 인스턴스 생성 방법과 함께 데이터의 레이아웃/스키마를 정의한다.

새 컬렉션을 만들려면, Waterline.Collection 을 확장(extend)하고, 당신이 필요로 하는 어떤 속성이든지 추가하면 된다.

 * 여기에서 말하는 Collection이란, RDBMS의 Table을 말하는 것이라고 생각한다. (아니면 나중에 고치겠음)

사용할 수 있는 옵션은 다음과 같다.

 - tableName : 컬렉션 정의에 정의되지 않은 경우에 사용

 - adapter

 - schema

 - attributes
        - string
        - text
        - integer
        - float
        - date

    - time
    - datetime
    - Boolean
    - binary
    - array
    - json
- autoCreatedAt, autoUpdateAt, createdAt, updatedAt
- autoPK
- lifecycle callbacks
- 당신이 정의한 다른 분류(class)의 메소드.

< 기존의 RDBMS 와 MongoDB의 용어 비교 (SQL to MongoDB Mapping Chart) >

출처 : https://docs.mongodb.org/manual/reference/sql-comparison/

Terminology and Concepts

The following table presents the various SQL terminology and concepts and the corresponding MongoDB terminology and concepts.

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index
table joins	embedded documents and linking
primary key Specify any unique column or column combination as primary key.	primary key In MongoDB, the primary key is automatically set to the _id field.
aggregation (e.g. group by)	aggregation pipeline See the SQL to Aggregation Mapping Chart.

또한, 인스턴스 메소드를 아래와 같이 정의할 수 있다고 한다.

```
    // You can also define instance methods here
    fullName: function() {
      return this.firstName + ' ' + this.lastName

    }
```

< Model >

워터라인 쿼리를 통해 나온 각각의 결과는 Model의 인스턴스가 된다.

이는 일부 CRUD 헬퍼 메소드와 함께, 컬렉션에 정의 된 모든 인스턴스 메소드를 추가한다.

기본적인 CRUD 인스턴스 메소드

- save
- destroy
- toObject
- toJSON

그렇지만, 특정 속성을 제거하고 싶은 경우에는, 다음과 같이 toJSON 메소드를 오버라이드 할 수도 있다.

```
var user = Waterline.Collection.extend({

  attributes: {
    name: 'string',
    password: 'string',

    // Override toJSON instance method
    toJSON: function() {
      var obj = this.toObject();
      delete obj.password;
      return obj;
    }
  }
});

// Then on an instantiated user:
user.find({ id: 1 }).exec(function(err, model) {
  return model.toJSON(); // Will return only the name
});
```

< 쿼리 메소드 >

쿼리는 하나의 Callback Interface 또는 연기중인 객체를 실행할 수 있다.

복잡한 쿼리를 구축하기 위한, 연기중인 객체의 메소드(deferred object method) 는 최고의 선택이다.

편의를 위해, Promise 는 기본적으로 지원된다.

Callback Method

```
User.findOne({ id: 1 }, function(err, user) {
  // Do stuff here
});
```

Deferred Object Method

```
User.find()
.where({ id: { '>': 100 }})
.where({ age: 21 })
.limit(100)
.sort('name')
.exec(function(err, users) {
  // Do stuff here
});
```

*** 강추하는 방법이라 하지만, 나는 최악의 퍼포먼스를 예상해본다. *** 만약 find() 메소드만으로 모든 documents를 select하는 것은 아니라면, 위의 방법은 나쁘지는 않다고 본다.

Promises

```
User.findOne()
.where({ id: 2 })
.then(function(user){
    var comments = Comment.find({userId: user.id}).then(function(comments){
        return comments;
    });
    return [user.id, user.friendsList, comments];
}).spread(function(userId, friendsList, comments){
    // Promises are awesome!
}).catch(function(err){
    // An error occurred
})
```

Promises 는 Bluebird 라이브러리를 사용한다. 그래서 첫번째 call또는 spread 또는 catch가 이루어진 후에 어떤것이든 할 수 있다. 완전한 Bluebird Promise 객체가 될 것이다.

반드시 알아두어야 할 것은, 데이터베이스의 요청을 완료하기 위해, 어떻게든 쿼리(calling이나 또 다른 function 등)를 종료해야만 한다.

다음과 같이 각 컬렉션 인스턴스에서 기본적으로 basic Methods 를 사용할 수 있다.

findOne (R)
find (R)
create (C)
update (U)
destroy (D)
count (row계산)

또는 createEach
findOrCreateEach
findOrCreate
findOneLike
findLike
startsWith
endsWith
contains

컬렉션을 기반으로 다이나믹 파인더를 사용할 수 있다.

< 페이지네이션 Pagination >

페이지네이션은 복잡하게 할 필요없이

skip(2)

limit(2)

paginate({page: 0, limit: 0})

메소드를 사용하면 된다.

< 정렬, Sorting >

정렬 또한 어렵지 않다.

```
User.find()
.sort('roleId asc')
.sort({ createdAt: 'desc' })
.exec(function(err, users) {
  // Do stuff here
});
```

asc 또는 desc 를 사용하면 된다.

< Validations, 유효성 >

유효성 검증은 Node Validate를 기반으로 하고 node-validate의 등록정보의 대부분을 지원하며, 앵커에 의해 처리된다.

검증에 대해서는 개발자의 Collection attributes안에 직접 정의하도록 한다.

당신은 Anchor 타입에 속성 유형을 설정할 수 있으며, 워터라인은 유효성검사를 구축하고, 문자열로 스키마 유형을 설정한다.

유형성 검사 규칙은, 테스트에 반환되는 간단한 값이나 함수로 정의될 수 있다. (동기, 비동기 모두)

```
    age: {
      type: 'integer',
      after: '12/12/2001'
    },

    website: {
      type: 'string',
      // Validation rule may be defined as a function. Here, an async function is mimicked.
      contains: function(cb) {
        setTimeout(function() {
          cb('http://');
        }, 1);
      }
    }
  }
});
```

```
var Event = Waterline.Collection.extend({

  attributes: {

    startDate: {
      type: 'date',
      // Validation rule functions allow you to validate values against other attributes
      before: function() {
        return this.endDate;
      }
    },

    endDate: {
```

< Custom Types, 사용자 정의 자료형 >

워터라인에서는 types 에 대한 해시값을 개발자 본인의 취향에 맞게 설정할 수 있다.

그것은 다른 attributes 의 값을 비교하여 액세스할 수도 있다.

```
var User = Waterline.Collection.extend({
  types: {
    point: function(latlng){
     return latlng.x && latlng.y
    },

    password: function(password) {
      return password === this.passwordConfirmation;
    });
  },

  attributes: {
    firstName: {
      type: 'string',
      required: true,
      minLength: 5,
      maxLength: 15
    },

    location: {
      // Note, that the base type (json) still has to be defined
      type: 'json',
      point: true
    },

    password: {
      type: 'string',
      password: true
    },

    passwordConfirmation: {
      type: 'string'
    }
  }
});
```

< Indexing, 인덱싱 >

Adapter가 지원하는 경우에는, 인덱스를 생성하는 모든 속성에 인덱스 속성을 추가 할 수 있다.

Key에 대한 반복쿼리를 수행할 때에 유용하다.

```
var User = Waterline.Collection.extend({

  attributes: {

    serviceID: {
      type: 'integer',
      index: true
    }
  }
});
```

현재 워터라인은 속성정의에 멀티 컬럼 인덱스를 지원하지 않는다. 특수 인덱스의 어떤 종류를 구축하려는 경우, 당신은 여전히 수동으로 구축해야한다. 또한 속성을 추가할 때, 인덱스가 자동으로 해당 속성에 대해 생성된다.

현재 문자열 필드에 인덱스를 추가하는 문제가 있다. 워터라인은 대소문자를 구분하는 방식의 쿼리를 수행하기 때문에, 우리는 문자열 속성에 인덱스를 사용할 수 없다. 이것은 문자열의 인덱스를 완벽히 지원하기위해 조만간 업데이트 될 예정이다.

< Lifecycle Callbacks >

위에서도 적었지만, 안드로이드의 Lifecycle Callbacks 와 의미가 완전히 동일하다.

라이프싸이클 콜백은, 쿼리의 특정 시간에 실행되도록 정의할 수 있는 기능(함수)이다.

Creating 하기전에, 또는 속성을 생성하기 전에, 또는 데이터를 검증하기 전에, 데이터를 변조하는데 유용하게 쓰인다.

- Create 관련 콜백
  beforeValidate / function(values, cb)
  afterValidate / function(values, cb)
  beforeCreate / function(values, cb)
  afterCreate / function(newlyInsertedRecord, cb)

- Update 관련 콜백
  beforeValidate / function(valuesToUpdate, cb)
  afterValidate / function(valuesToUpdate, cb)
  beforeUpdate / function(valuesToUpdate, cb)
  afterUpdate / function(updatedRecord, cb)

- Destroy 관련 콜백
  beforeDestroy / function(criteria, cb)
  afterDestroy / function(cb)

< Test 방법 >

모든 테스트는 mocha와 함께 작성하고 npm으로 실행해야 한다.

```
npm test
```

< Coverage, 적용 범위 >

코드 커버리지 보고서를 생성하려면

```
npm run coverage
```

를 타이핑하고, coverage/lcov-report/index.html 을 열어보면 된다.

< NodeJS + Express + MongoDB 와 연동해보기 >

1. `npm install waterline --save`

   아래와 같이 3개의 waterline모듈과 30개의 의존모듈이 생성되는 듯 하다. 이 워터라인을 **Model** 이라고 한다.

anchor	2016-01-27 오후 7:33
switchback	2016-01-27 오후 7:33
waterline	2016-01-27 오후 7:33
waterline-criteria	2016-01-27 오후 7:33
waterline-schema	2016-01-27 오후 7:33
lodash	2016-01-27 오후 7:33
bluebird	2016-01-27 오후 7:33
winston	2016-01-27 오후 7:33
balanced-match	2016-01-27 오후 7:33
colors	2016-01-27 오후 7:33
deep-diff	2016-01-27 오후 7:33
deep-equal	2016-01-27 오후 7:33
i	2016-01-27 오후 7:33
ncp	2016-01-27 오후 7:33
pkginfo	2016-01-27 오후 7:33
prompt	2016-01-27 오후 7:33
revalidator	2016-01-27 오후 7:33
utile	2016-01-27 오후 7:33
brace-expansion	2016-01-27 오후 7:33
concat-map	2016-01-27 오후 7:33
cycle	2016-01-27 오후 7:33
eyes	2016-01-27 오후 7:33
glob	2016-01-27 오후 7:33
inflight	2016-01-27 오후 7:33
isstream	2016-01-27 오후 7:33
minimatch	2016-01-27 오후 7:33
mute-stream	2016-01-27 오후 7:33
once	2016-01-27 오후 7:33
path-is-absolute	2016-01-27 오후 7:33
read	2016-01-27 오후 7:33
rimraf	2016-01-27 오후 7:33
stack-trace	2016-01-27 오후 7:33
wrappy	2016-01-27 오후 7:33

2. `npm install sails-mongo --save`

   내가 사용할 DB의 의존모듈을 설치해준다. 이를 **Adapter** 라고 한다. (mySQL이라면, sails-mysql 을 설치)

sails-mongo	2016-01-28 오후 12:13
waterline-cursor	2016-01-28 오후 12:13
validator	2016-01-28 오후 12:13
waterline-errors	2016-01-28 오후 12:13

3. app.js 에 아래의 코드를 입력한다.

```
var Waterline = require('waterline');  //워터라인 모듈
```

```
//워터라인 Collection 정의하기, 이를 model 이라고 하는듯 하다.
var userCollection = Waterline.Collection.extend({
  connection: "default",
  identity: "usertable",  //대문자를 써도 소문자로만 인식함. waterline.initialize()메소드에서 ontology.collections.usertable 로 사용
  tableName: 'user',    // Define a custom table name (테이블명 정의)
  schema: true,         // Set schema true/false for adapters that support schemaless
  adapter: 'mongodb',  // Define an adapter to use (사용할 데이터베이스 정의)
  attributes: {         // Define attributes for this collection (컬럼을 정의)
    id    : { type: 'string', required: true, unique: true, primaryKey: true },   // also accepts any validations
    pw    : { type: 'string', required: true, maxLength: 10 },
    email : { type: 'email', required: true },  //문자열이 어댑터로 전달이 될 때, 특수한 type은 검증과 설정에 사용할 수 있습니다.
    like  : { type: "string", required: false },  //type: 'integer', min: 19
    좋아하는것: function() {  //여기에 인스턴스의 메소드도 정의할 수 있습니다.
      return "저는 " + this.like + "를 좋아합니다.";
    },
    pets  : { collection: "pettable", via: "주인" },  //pet테이블에서 "주인"컬럼을 참조하도록 함 (?) "1대 다" 관계를 만드는 중.
    seqNo : { type: "integer", autoIncrement: true, unique: true },
    가입일 : new Date(),
    autoPK : false,
    autoCreatedAt: true,
    autoUpdatedAt: true
  },
  // Lifecycle Callbacks
  // beforeValidate / afterValidate // beforeUpdate / afterUpdate // beforeCreate / afterCreate // beforeDestroy / afterDestroy
  beforeCreate: function(values, next) { console.log("User 도큐먼트 생성을 시작합니다."); next(); },  //비밀번호 암호화로직을 추가할수 있다.
  afterCreate: function(values, next) { console.log("User 도큐먼트 생성완료"); next(); },
  afterDestroy: function(values, next) { console.log("User 도큐먼트 삭제완료"); next(); },
  doSomething: function() {  // Class Method
    // Do something here
    console.log("두 썸띵 히어?");
  }
});
```

```
var petCollection = Waterline.Collection.extend({
  connection: "default",
  identity: "pettable",
  tableName: "pet",
  schema: true,
  adapter: "mongodb",
  attributes: {
   "펫 이름": { type: "string", required: true },
    주인: { model: "usertable" }  //유저테이블과 조인함으로써, "many-to-many 관계" 성립됨.
  },
  beforeCreate: function(values, next) { console.log("Pet 도큐먼트 생성을 시작합니다."); next(); },
  afterDestroy: function(values, next) { console.log("Pet 도큐먼트 삭제완료."); next(); },
  bark: function() { console.log(this["펫 이름"] + " : 왈왈"); }
});
```

```
var waterline = new Waterline();
waterline.loadCollection(userCollection);  //여기서 loadCollection() 메소드를 실행하면, 아래의 ontology로 들어옴.
waterline.loadCollection(petCollection);
```

```
var config = {
  adapters: {
    //'default': 'mongo',
    'mongo': require('sails-mongo')  //sails-몽고 모듈을 어댑터에 꽂음
  },
  connections: {
    default: {
      adapter: 'mongo',
      url: "mongodb://localhost:27017/mongoDBtest"  //기본적으로는 mongodb://url/sails 로 들어감.
    }
  }
};
```

```
waterline.initialize(config, function(err, ontology) {  //ontology변수는 컬렉션 객체를 가지고있음.
  if(err) { throw err; }
  //완전히 모델을 초기화해서 가지런하게 하자,
  var User = ontology.collections.usertable;  //쉽게말해서 바로가기를 추가하는것.
  var Pet  = ontology.collections.pettable;  //서버를 올릴때 에러는 안나는데, pettable이라는 tableName을 정의하지않았으면, 사용도 불가능하다.

  //이곳은 위에서 아래로 순차적으로 실행하지 않는다. 비동기로 실행하기때문에, Promise로서 어떤 순서대로 실행할지 약속은 되어있지만,
  //User.create를 실행한뒤에 .then을 한 뒤에 그 다음 .then()을 호출한다는 보장이 없다. 갑자기 User.destroy()를 호출한다.
  //그리고 CRUD로직 이외에는 모두 순차적으로 실행하는듯 하다. CRUD로직이 가장 나중에 실행되는 것 같다.

  User.create({  //테스트로 한번 생성해봄.
    id: "test00",
    pw: "00",
    email: "test@test.com",
    like: "오호라"
  }).then(function (user) { //첫번째 then 메소드의 인자로는, 생성된 해당 document인듯 하다.
    return Pet.create({
        "펫 이름": "testPet",
        주인: user.id
    });

  }).then(function (pet) { // Then we grab all users and their pets
    return User.find().populate('pets');  //각 사용자에 대한 애완 동물의 기록을 찾아갑니다...???

  }).then(function(users){ //이전의 then에서 return한 객체는 다음에 전달됨.
    console.dir(users);  //최초로 create한 테이블(유저테이블)의 모든 documents를 Array로 출력함.

  }).catch(function(err){ //오류가 발생하면, 바로 catch블록으로 이동함.
    console.log("오류발생!!!");
    console.error(err);
  });
```

```
  User.destroy({ id: "test00" })
    .exec(function (err) {  //destroy() 를 할때는 반드시 exec() 함수를 체인으로 호출해줘야한다.
      Pet.destroy({ "펫 이름": "testPet" })
        .exec(function (err) {  //destroy() 를 할때는 반드시 exec() 함수를 체인으로 호출해줘야한다. 그렇지않으면 삭제가 되지않는다.
          console.log("펫 삭제 콜백");
        });
    });
  app.models = ontology.collections;  //앱 전체에서 쓸수있도록 컬렉션과 커넥션을 삽입함.
  app.connections = ontology.connections;

  console.log("모든 initialize() 셋팅이 완료되었습니다.");
  //이 안에서는 트랜젝션이 걸리기 때문에, 중간에 에러가 나면 create했던것들은 모두 사라진다.
});
```

[Sails.js] Ruby on Rails 가 있다면 Node on Sails 도 있다 - 윤영식 블로그
http://mobicon.tistory.com/321

벨리데이션에 관하여. type선정
https://github.com/balderdashy/waterline-docs/blob/master/models/validations.md

워터라인과 몽고디비 결합하기
https://gist.github.com/wangyangkobe/70933684f2de6e24e037