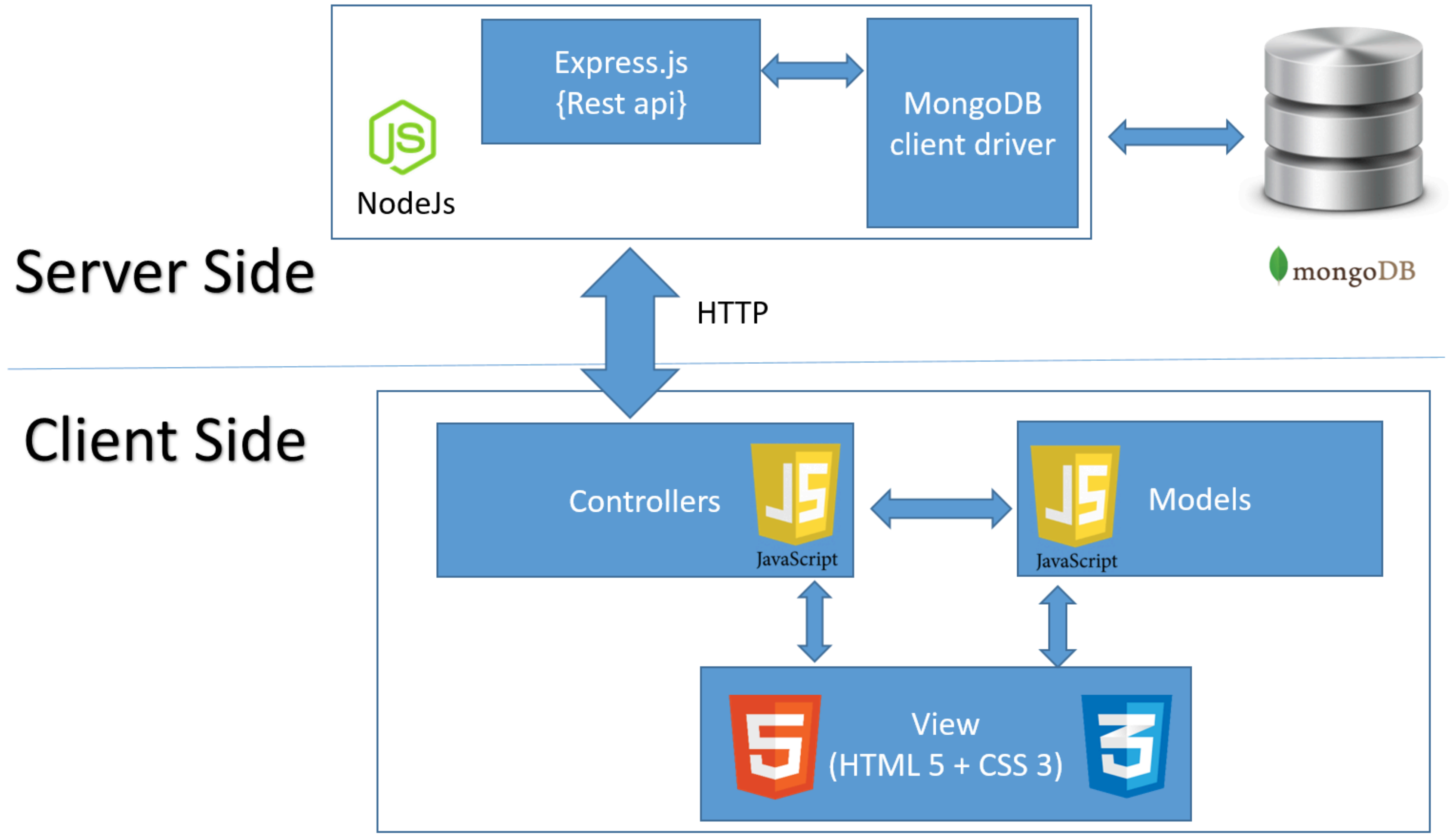
Mongoose: MongoDB object modelling for Node.js

Ashok Dudhat
Germany Startup Jobs
www.germanystartupjobs.com

MongoDB Driver



Main inconveniences of native MongoDB driver

- **No** data **validation**.
- **No** **casting** during inserts.
- **No** **encapsulation**.
- **No** references (**joins**).

Terminologies

Collections

'Collections' in Mongo are equivalent to tables in relational databases. They can hold multiple JSON documents.

Documents

'Documents' are equivalent to records or rows of data in SQL. While a SQL row can reference data in other tables, Mongo documents usually combine that in a document.

Fields

'Fields' or attributes are similar to columns in a SQL table.

Schema

While Mongo is **schema-less**, **SQL defines a schema via the table definition**. A **Mongoose** '**schema**' is a document data structure (or shape of the document) that is enforced via the application layer.

Models

'**Models**' are higher-order constructors that take a **schema** and **create an instance** of a **document** equivalent to **records** in a relational database.

What is Mongoose?

- It's an object data modelling/Mapper (ODM) tool for node.js. This means that Mongoose allows you to define objects with a strongly-typed schema that is mapped to a MongoDB document. It speeds up writing queries and validation code.

- Officially supported by 10gen, Inc.

- Mongoose is a way to make connection with mongodb database. It provide mongodb validation and query in a very simple manner and it makes development fast.

- In simple words, Mongoose acts as an intermediate between mongodb and server side language(like NodeJs)

Features

- Async and sync **validation** of models.
- Model **casting.**
- Object **lifecycle management** (middlewares).
- Pseudo-**joins.**
- **Query builder.**

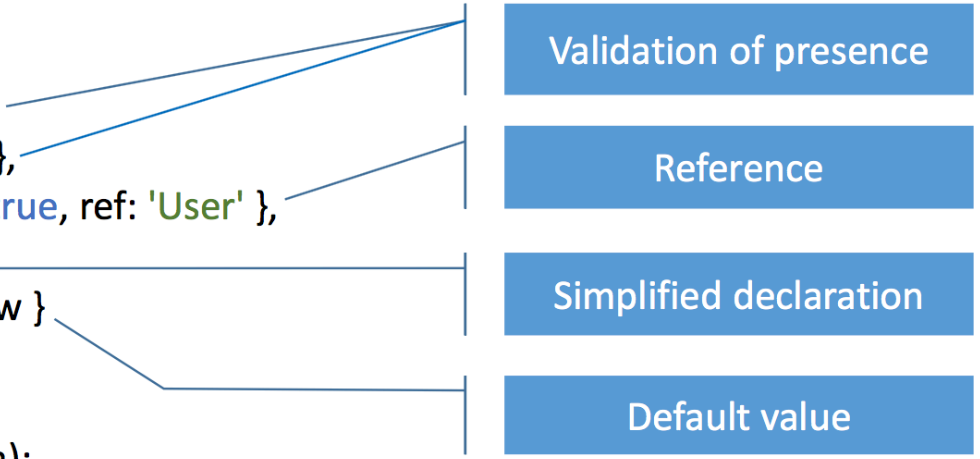
Mongoose setup

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/mycollection');

var Schema = mongoose.Schema;

var PostSchema = new Schema({
  title: { type: String, required: true },
  body: { type: String, required: true },
  author: { type: ObjectId, required: true, ref: 'User' },
  tags: [String],
  date: { type: Date, default: Date.now }
});

mongoose.model('Post', PostSchema);
```



Mongoose features: validation

Simplest validation example: presence

```
var UserSchema = new Schema({ name: { type: String, required: true } });
var UserSchema = new Schema({ name: { type: String, required: 'Oh snap! Name is required.' } });
```

Passing validation function:

```
var UserSchema = new Schema({ name: { type: String, validator: validate } });
var validate = function (value) { /*...*/ }; // synchronous validator
var validateAsync = function (value, respond) { respond(false); }; // async validator
```

Via .path() API call:

```
UserSchema.path('email').validate(function (email, respond) {
  var User = mongoose.model('User');
  User.find({ email: email }).exec(function (err, users) {
    return respond(!err && users.length === 0);
  });
}, 'Such email is already registered');
```

Mongoose features: casting

- Each property is **cast** to its mapped SchemaType in the document, obtained by the query.
- Valid SchemaTypes are:
  - String
  - Number
  - Date
  - Buffer
  - Boolean
  - Mixed
  - ObjectId
  - Array

```
var PostSchema = new Schema({
  _id: ObjectId, // implicitly exists
  title: { type: String, required: true },
  body: { type: String, required: true },
  author: { type: ObjectId, required: true, ref: 'User' },
  tags: [String],
  date: { type: Date, default: Date.now },
  is_featured: { type: Boolean, default: false }
});
```

Mongoose features: encapsulation

Models can have *static methods* and *instance methods:*

```
PostSchema.statics = {
  dropAllPosts: function (areYouSure) {
    // drop the posts!
  }
};
```

```
PostSchema.methods = {
  addComment: function (user, comment,
callback) {
    // add comment here
  },
  removeComment: function (user,
comment, callback) {
    // remove comment here
  }
};
```

Mongoose features: lifecycle management

Models have .pre() and .post() middlewares, which can hook custom functions to *init, save, validate* and *remove* model methods:

```
schema.pre('save', true, function (next, done) {
  // calling next kicks off the next middleware in parallel
  next();
  doAsync(done);
});

schema.post('save', function (doc) {
  console.log('%s has been saved', doc._id);
});
```

Mongoose features: population

Population is the process of automatically replacing the specified paths in the document with document(s) from other collection(s):

```
PostSchema.statics = {
  load: function (permalink, callback) {
    this.findOne({ permalink: permalink })
        .populate('author', 'username avatar_url')
        .populate('comments', 'author body date')
        .exec(callback);
  }
};
```

Mongoose features: query building

Different methods could be stacked one upon the other, but the query itself will be generated and executed only after .exec():

```
var options = {
  perPage: 10,
  page: 1
};

this.find(criteria)
    .populate('author', 'username')
    .sort({'date_published': -1})
    .limit(options.perPage)
    .skip(options.perPage * options.page)
    .exec(callback);
```

Schema-Driven Design for your application

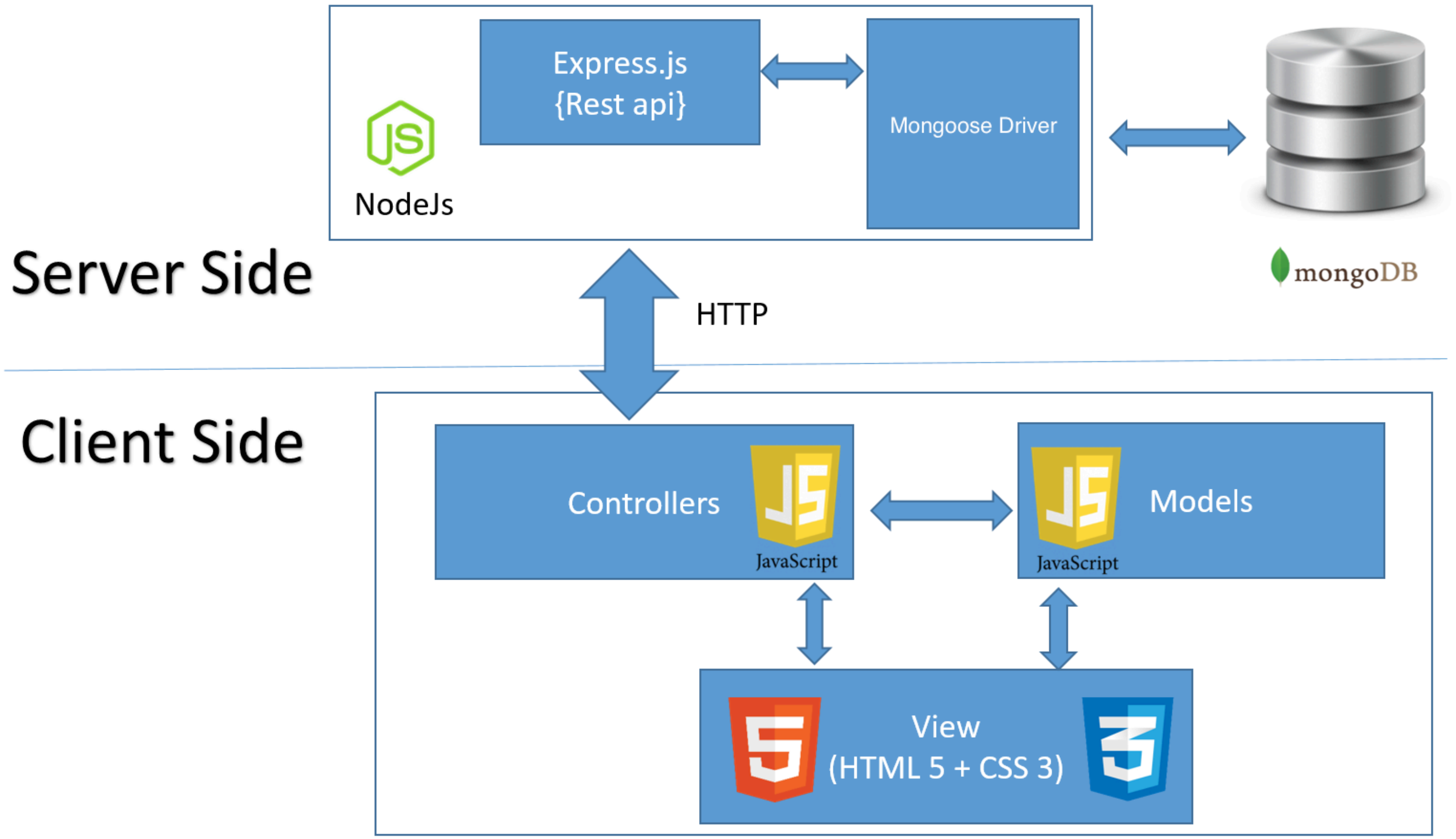
- Schemas should match data-access patterns of your application.

- You should pre-join data where it’s possible (and use Mongoose’s .populate() wisely!).

- You have no constraints and transactions — keep that in mind.

- Using Mongoose for designing your application’s Schemas is similar to OOP design of your code.

Mongoose Driver



Installation : Mongoose

```
$ npm install mongoose
```

OR

```
$ npm install mongoose - -save
```

Installation: Robomongo

- Robomongo is now Robo 3T.
- Robo 3T is the free lightweight GUI for MongoDB.
- MongoDB GUI with embedded shell.
- Download from the below link.
  - https://robomongo.org/download