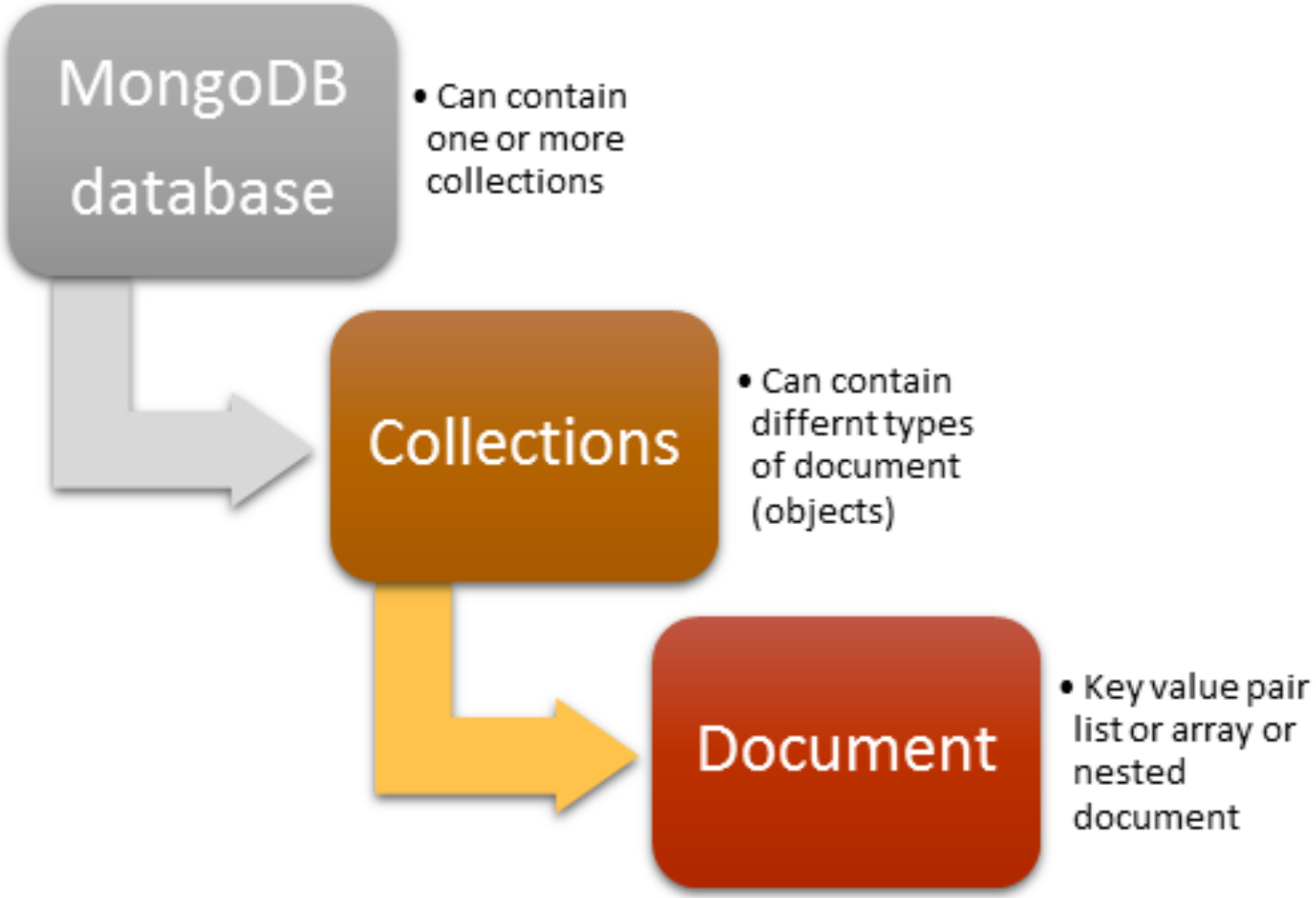
MongoDB

Overview

- MongoDB è un DBMS non relazionale, orientato ai documenti.
- Classificato come un database di tipo NoSQL
- MongoDB si allontana dalla struttura tradizionale basata su tabelle dei database relazionali in favore di documenti in stile JSON con schema dinamico
- Caratteristiche: scalabilità e flessibilità

Structure


MongoDB database
• Can contain one or more collections
Collections
• Can contain differnt types of document (objects)
Document
• Key value pair list or array or nested document

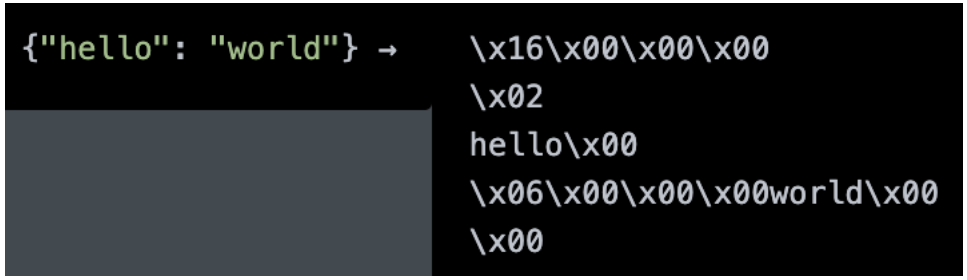
Features

- Document Based
  - strutture chiave valore
- Scalabile
  - adatto per I cluster di macchine
- Flessibile
  - non definiamo uno schema
- Performant
  - Indexing, sharding, duplication
- Open source

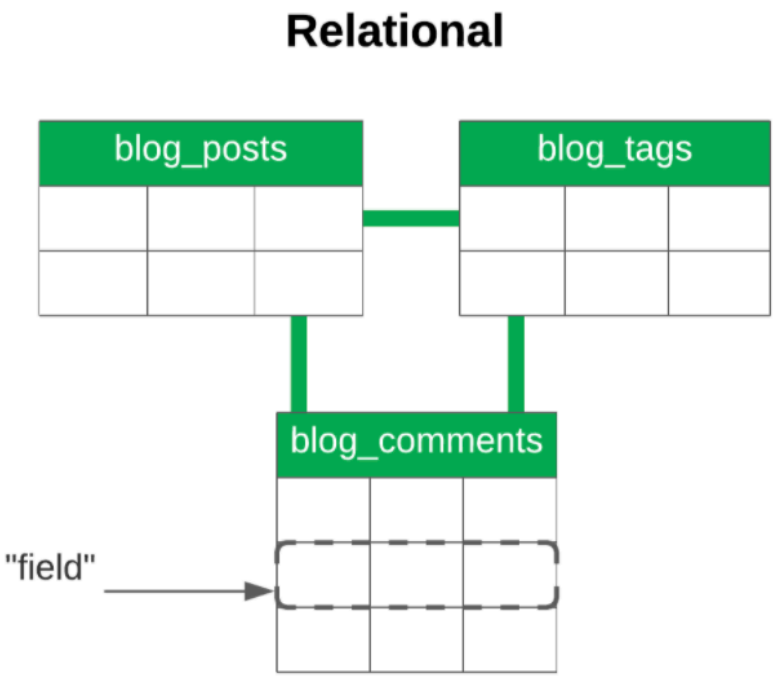
Documenti e BSON

```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"'
  }
  "hobbies": ["surfing", "coding"]
}
```

JSON vs BSON



SQL vs NoSQL



https://www.mongodb.com/databases/types

SQL vs NoSQL

ID	first_name	last_name	cell	city	year_of_birth	location_x	locatio
1	'Mary'	'Jones'	'516-555-2048'	'Long Island'	1986	'-73.9876'	'40.75

ID	user_id	profession
10	1	'Developer'
11	1	'Engineer'

ID	user_id	name	version
20	1	'MyApp'	1.0.4
21	1	'DocFinder'	2.5.7

ID	user_id	make	year
30	1	'Bentley'	1973
31	1	'Rolls Royce'	1965

```
first_name: "Mary",
last_name: "Jones",
cell: "516-555-2048",
city: "Long Island",
year_of_birth: 1986,
location: {
        type: "Point",
        coordinates: [-73.9876, 40.7574]
},
profession: ["Developer", "Engineer"],
apps: [
 { name: "MyApp",
   version: 1.0.4 },
 { name: "DocFinder",
   version: 2.5.7 }
 ],
cars: [
   { make: "Bentley",
     year: 1973 },
   { make: "Rolls Royce",
     year: 1965 }
 ]
```

Embedded Data

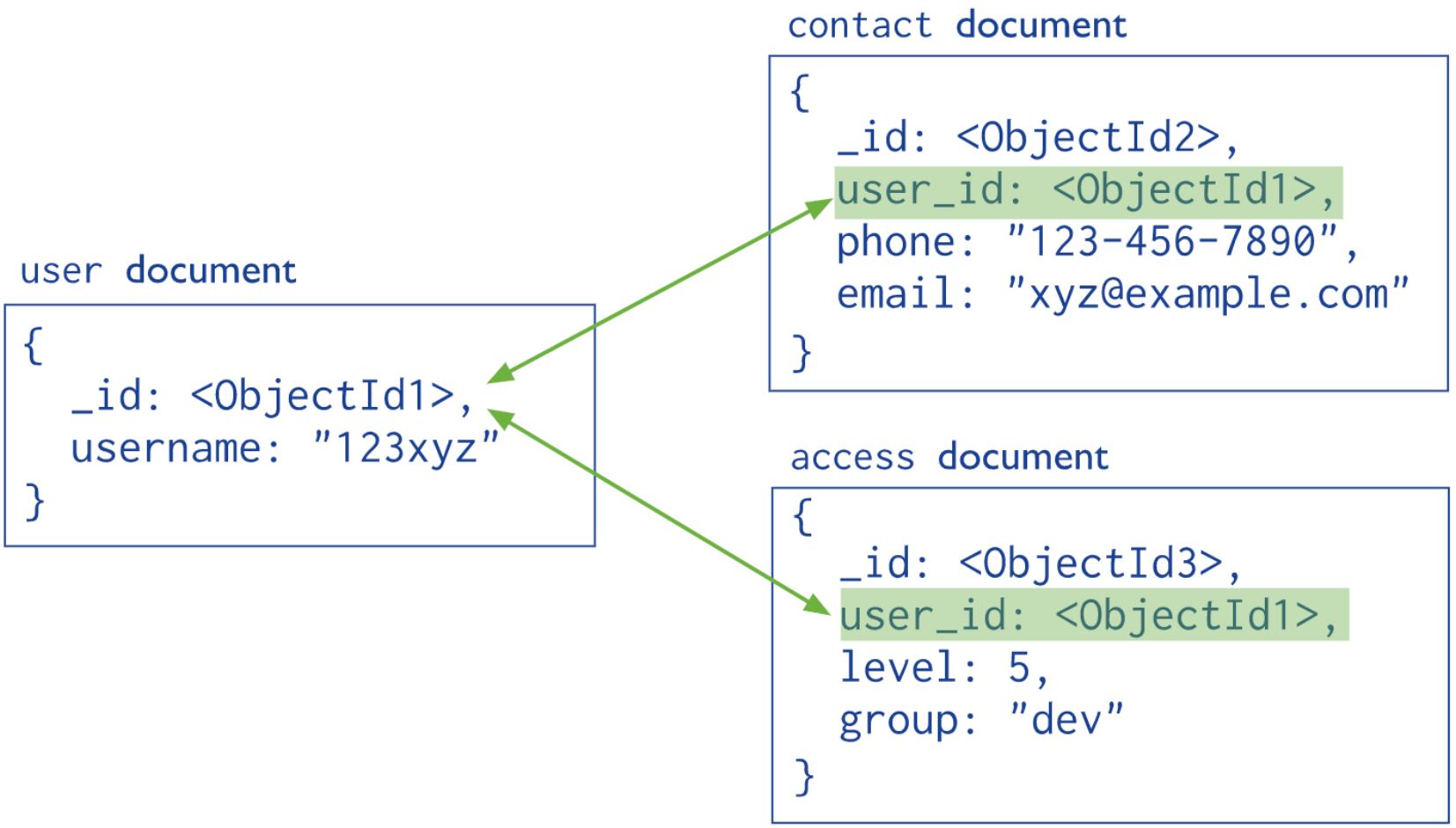
```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
            level: 5,
            group: "dev"
          }
}
```

Embedded sub-document

Embedded sub-document

https://docs.mongodb.com/manual/core/data-modeling-introduction/

Reference


contact document
{
_id: <ObjectId2>,
user_id: <ObjectId1>,
phone: "123-456-7890",
email: "xyz@example.com"
}
user document
{
_id: <ObjectId1>,
username: "123xyz"
}
access document
{
_id: <ObjectId3>,
user_id: <ObjectId1>,
level: 5,
group: "dev"
}

Installazione

- Locale
  - https://docs.mongodb.com/manual/installation/
- Docker
  - https://www.thepolyglotdeveloper.com/2019/01/getting-started-mongodb-docker-container-deployment/
- Cloud
  - https://www.mongodb.com/cloud/atlas

Mongo shell

- mongo --port 28015
- mongo "mongodb://mongodb0.example.com:28015"

- https://docs.mongodb.com/manual/reference/mongo-shell/
  - db
  - use <database>
  - show dbs
  - show collections

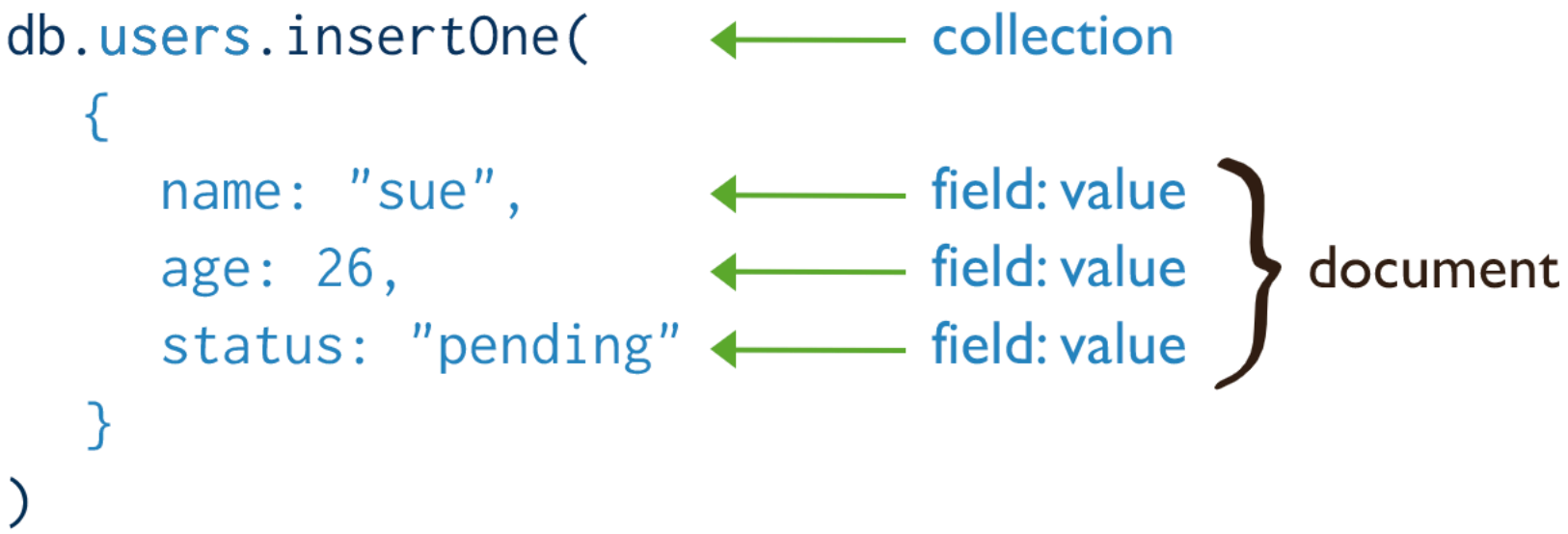
https://mws.mongodb.com/?version=4.4

CRUD OPERATIONS

https://docs.mongodb.com/manual/crud/

create

- db.collection.insertOne()
- db.collection.insertMany()



read

- db.collection.find()

```
db.users.find(
    { age: { $gt: 18 } },
    { name: 1, address: 1 }
).limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

update

- db.collection.updateOne() New in version 3.2
- db.collection.updateMany() New in version 3.2
- db.collection.replaceOne() New in version 3.2

```
db.users.updateMany(
   { age: { $lt: 18 } },
   { $set: { status: "reject" } }
)
```

collection

update filter

update action

delete

- db.collection.deleteOne() New in version 3.2
- db.collection.deleteMany() New in version 3.2

```
db.users.deleteMany(
   { status: "reject" }
)
```

collection

delete filter

Esempi

https://mws.mongodb.com/?version=4.4

- insertMany

- find()
- find() semplice
- find() con $in
- find() con $lt p $gt
- find() AND
- find() OR

- projection and limit

- updateOne
- deleteOne

https://docs.mongodb.com/manual/crud/

Atlas

Schema Validation

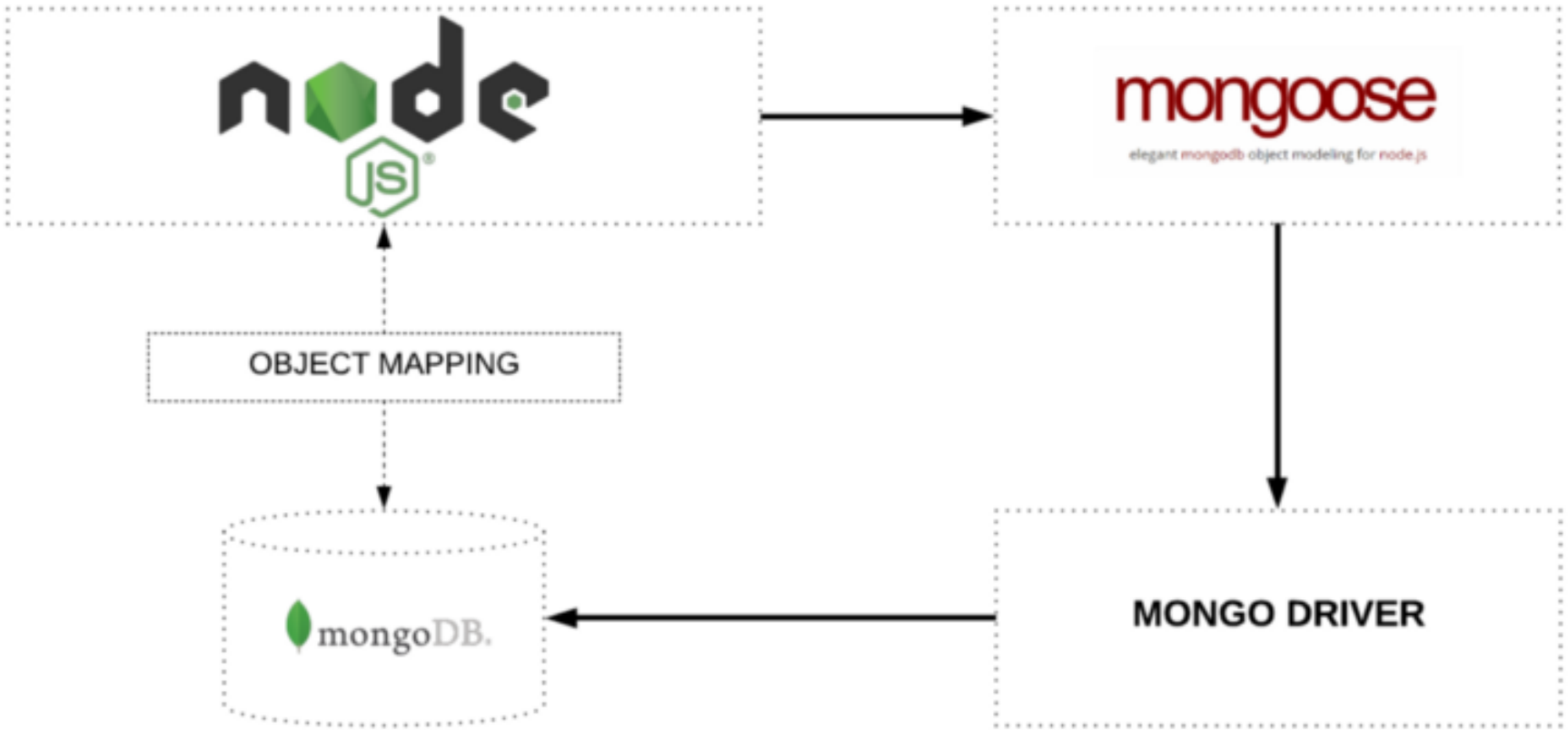
- Validation rules are on a per-collection basis.
- To specify validation rules when creating a new collection, use db.createCollection() with the validator option.
- To add document validation to an existing collection, use collMod command with the validator option.

```
db.runCommand( {
    collMod: "recipes",
    validator: { $jsonSchema: {
         <<Validation Rules>>
        }
    }
} )
```

```
db.createCollection("recipes",
    validator: { $jsonSchema: {
         <<Validation Rules>>
        }
    }
)
```

MONGOOSE

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose


node
JS
mongoose
elegant mongodb object modeling for node.js
OBJECT MAPPING
mongoDB.
MONGO DRIVER

Overview

- Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js.

- Features
  - manages relationships between data
  - provides schema validation
  - translate between objects and the representation in the DB
- Schema:
  - data structure, default values, validation
- Model
  - an interface toward the DB

Connect

```
//Import the mongoose module
var mongoose = require('mongoose');

//Set up default mongoose connection
var mongoDB = 'mongodb://127.0.0.1/my_database';
mongoose.connect(mongoDB, {useNewUrlParser: true, useUnifiedTopology: true});

//Get the default connection
var db = mongoose.connection;
```

Schema and Model

- The Schema allows you to define the fields stored in each document along with their validation requirements and default values

```
// Define schema
var Schema = mongoose.Schema;

var SomeModelSchema = new Schema({
  a_string: String,
  a_date: Date
});

// Compile model from schema
var SomeModel = mongoose.model('SomeModel', SomeModelSchema );
```

Schemi complessi

```
var schema = new Schema(
{
  name: String,
  binary: Buffer,
  living: Boolean,
  updated: { type: Date, default: Date.now() },
  age: { type: Number, min: 18, max: 65, required: true },
  mixed: Schema.Types.Mixed,
  _someId: Schema.Types.ObjectId,
  array: [],
  ofString: [String], // You can also have an array of each of the other types too.
  nested: { stuff: { type: String, lowercase: true, trim: true } }
})
```

Esempio

```
var breakfastSchema = new Schema({
  eggs: {
    type: Number,
    min: [6, 'Too few eggs'],
    max: 12,
    required: [true, 'Why no eggs?']
  },
  drink: {
    type: String,
    enum: ['Coffee', 'Tea', 'Water',]
  }
});
```

Validating

```
const schema = new Schema({ name: String, age: { type: Number, min: 0 } });
const Person = mongoose.model('Person', schema);

let p = new Person({ name: 'foo', age: 'bar' });
// Cast to Number failed for value "bar" at path "age"
await p.validate();

let p2 = new Person({ name: 'foo', age: -1 });
// Path `age` (-1) is less than minimum allowed value (0).
await p2.validate();
```

```
// Cast to number failed for value "bar" at path "age"
await Person.updateOne({}, { age: 'bar' });

// Path `age` (-1) is less than minimum allowed value (0).
await Person.updateOne({}, { age: -1 }, { runValidators: true });
```

Esempio

```
mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
    useUnifiedTopology: true,
  })
  .then((con) => {
    console.log('database connected');
  });
```

```
const prodSchema = new mongoose.Schema({
  name: { type: String, required: [true, 'name missing'] },
  prezzo: Number,
  luogo: String,
});

const prodModel = new mongoose.model('Product', prodSchema);
```

insert

```
const Product = new mongoose.model('Product', prodSchema);

const prod1 = new Product({
  name: 'Vino rosso',
  prezzo: 10,
  luogo: 'Roma',
});

prod1
  .save()
  .then((doc) => {
    console.log('data saved');
  })
  .catch((err) => {
    console.error(err);
  });
```

filtering

```
// With a JSON doc
Person.
  find({
    occupation: /host/,
    'name.last': 'Ghost',
    age: { $gt: 17, $lt: 66 },
    likes: { $in: ['vaporizing', 'talking'] }
  }).
  limit(10).
  sort({ occupation: -1 }).
  select({ name: 1, occupation: 1 }).
  exec(callback);
```

filtering query

```
// Using query builder
Person.
  find({ occupation: /host/ }).
  where('name.last').equals('Ghost').
  where('age').gt(17).lt(66).
  where('likes').in(['vaporizing', 'talking']).
  limit(10).
  sort('-occupation').
  select('name occupation').
  exec(callback);
```

Virtual Properties

```
// define a schema
const personSchema = new Schema({
  name: {
    first: String,
    last: String
  }
});

// compile our model
const Person = mongoose.model('Person', personSchema);

// create a document
const axl = new Person({
  name: { first: 'Axl', last: 'Rose' }
});
```

```
personSchema.virtual('fullName').get(function() {
  return this.name.first + ' ' + this.name.last;
});
```

Middleware

```
const schema = new Schema(..);
schema.pre('save', function(next) {
  // do stuff
  next();
});
```

```
schema.pre('save', async function() {
  await doStuff();
  await doMoreStuff();
});
```

Middleware

```
const schema = new Schema(..);
schema.pre('save', function(next) {
  // do stuff
  next();
});
```

```
schema.pre('save', async function() {
  await doStuff();
  await doMoreStuff();
});
```

```
schema.pre('save', function(next) {
  const err = new Error('something went wrong');
  // If you call `next()` with an argument, that argument is assumed
  // an error.
  next(err);
});

schema.pre('save', function() {
  // You can also return a promise that rejects
  return new Promise((resolve, reject) => {
    reject(new Error('something went wrong'));
  });
});
```