heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Data Persistence

Structuring data for mongoDB

heig-vd

Daily menu

- Javascript and Promises training
- Data persistence
  - MongoDB
  - Mongoose
- (Build pipelines)

heig-vd

Javascript basics and Promises...all clear ?

heig-vd

Javascript basics

Are you familiar with arrow functions ?

```
[...].then(function (value) {
  return value + 1;
});
```

```
[...].then((value) => {
  return value + 1;
});
```

```
[...].then(value => value + 1);
```

```
const increment = (v) => v + 1;
[...].then(increment);
```

heig-vd

Where do I store my data ?

heig-vd

Where do I store my data ?

- For a long time, web applications were storing data in **RDBMS** (MySQL, Postgres)
- About 10 years ago, many alternatives started to appear. Today, we can choose between hundreds of NoSQL databases.
- Which one should we look at and how do we use it?

heig-vd

Where do I store my data ?

- There are **different types** of NoSQL databases: key-value stores, graph databases, document stores, etc.
- **Document stores** allow us to store semi-structured information, similar to JSON payloads.
- **MongoDB** was one of the early **popular** document stores. It still is.

heig-vd

How ?

- First, understand **how data is organised** in MongoDB: databases, collections, documents, fields.
- Then, learn how to perform **CRUD** operations via the console.
- Finally, learn how to do the same operations in **Javascript**.
- One more thing: learn about **Mongoose**.

MongoDB

Document oriented NoSQL Database

Definition

MongoDB is one of the most popular NoSQL databases (and one of the first to have been categorized as such).

It is a schema-less document-oriented database:

- The data store is made of several **collections**.
- Every collection contains a set of **documents**, which you can think of as JSON objects.
- The structure of documents is not defined a priori and is not enforced. This means that a collection can contain documents that have different **fields**.

> MongoDB stores JSON documents in a binary representation called BSON (Binary JSON). BSON encoding extends the popular JSON representation to include additional data types such as int, long, and floating point.

Rich Document Data Models

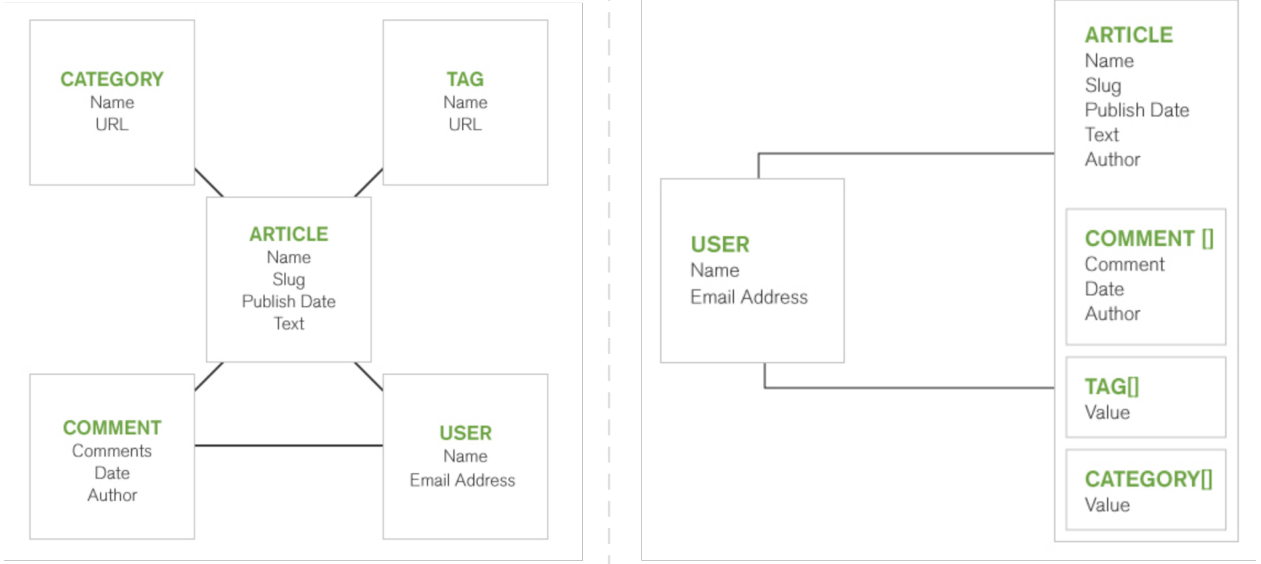
- Relation data models flattens data into rigid 2-dimensional **tabular structure** of rows and columns
- In contrast, rich document data models can have **embedded sub-documents and arrays**

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document
Index	Index
JOIN	Embedded Document or Reference

Table 1: Translating between relational and document data models

MongoDB

Relational data vs Rich document data models



With Rich Document Data Model (on the right), all of the blog data is aggregated within a single document, linked with a single reference to a user document

Data modeling

- Creating a data model with MongoDB does not have to follow the rules that apply for relational databases. Often, they should not.
- However, the data should be organized depending on the **application needs** — how your application queries and updates data.
- Consider the **performance characteristics** of the database engine.
- The key consideration for the structure of your documents is the decision to **embed** or to use **references**.

Embedded Data

Generally known as **denormalized** data model

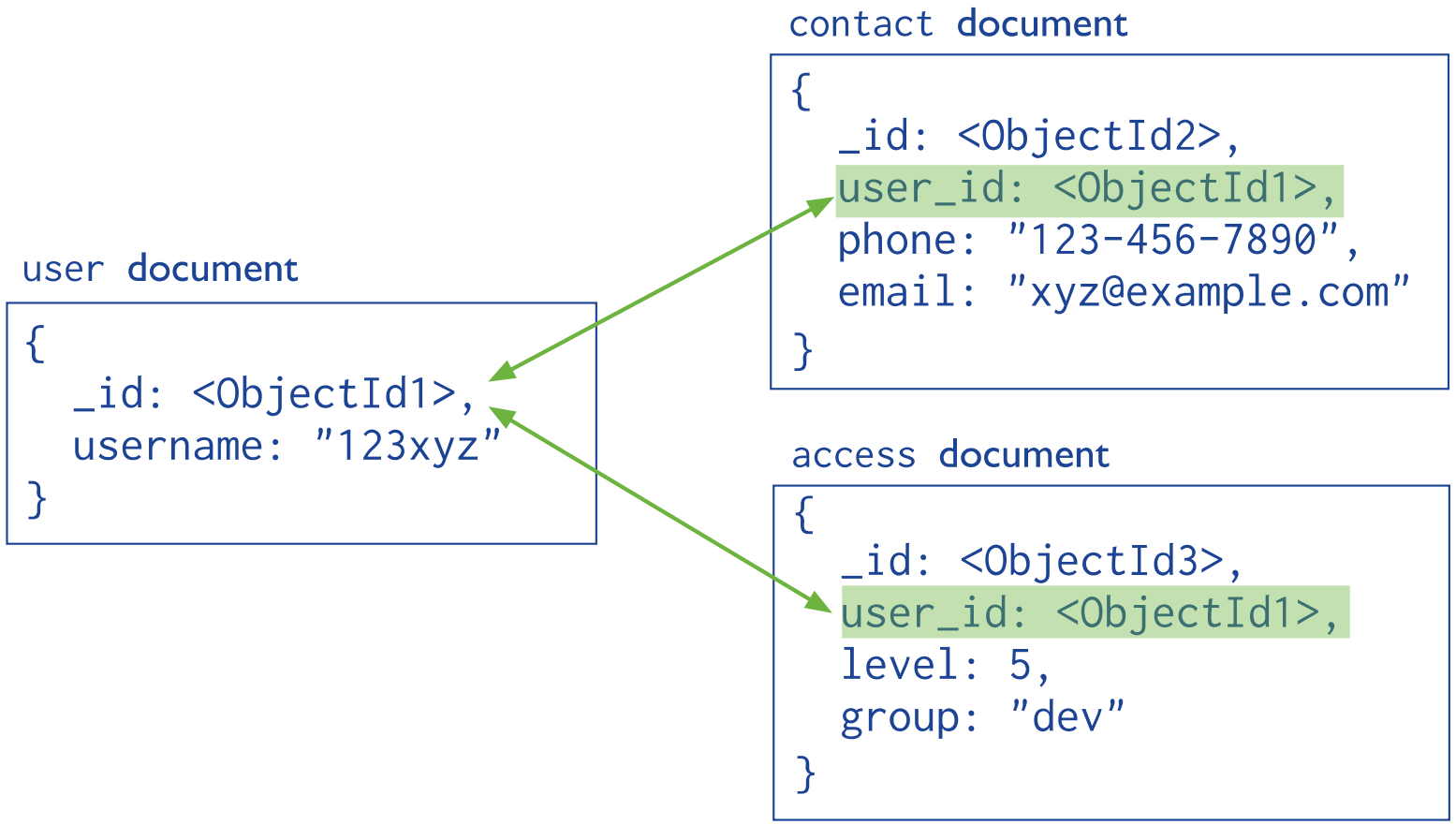
```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

Embedded sub-document

References

known as **normalized** data model — describe relationships using references between documents.



When to embed data ?

Embedding data allows to retrieve information with *fewer queries* and provides *better read performance*. It's also possible to update related data in a single *atomic write* operation.

You should **favor embedding**

- When you have “contains” relationships between entities
- When you have “One-to-Few” relationships between entities

...unless there is a compelling **reason not to**

- When there is a risk to reach the maximum BSON document size (16Mb)
- When embedding would result in unwanted duplication of data (sub-documents)
- When needing to access an object on its own

When to use references ?

Using normalized data models provides *more flexibility* than embedding, but requires an *application-level* joins to get complete information

Use normalized data models:

- When embedding would result in **duplication** of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- to represent more **complex many-to-many** relationships.
- to model large **hierarchical** data sets

> If you index correctly and use the projection specifier then application-level joins are barely more expensive than server-side joins in a relational database.

One-to-one relationships

Normalized data model

2 documents (requires 2 queries to get all of the person data)

```
> db.addresses.findOne()
{
  contact_id: "joe",
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}

> db.contacts.findOne({ _id: "joe" })
{
  _id: "joe",
  name: "Joe Bookreader"
}
```

Embeeded data model

single document (requires only 1 query to get all the person data)

```
> db.contacts.findOne()
{
  _id: "joe",
  name: "Joe Bookreader",
  address: {
    street: "123 Fake Street",
    city: "Faketon",
    state: "MA",
    zip: "12345"
  }
}
```

One-to-N relationships

There is different ways to describe One-to-N relationships:

- One-to-Few
- One-to-Many
- One-to-Squillions

Again, it depends on your application needs. Each methods for structuring has its pros and cons

One-to-Few

The **embedded data model** allows you to retrieve complete information with one query :

```
> db.contacts.findOne()
{
  _id: "joe",
  name: "Joe Bookreader",
  addresses: [
    {
      street: "123 Fake Street",
      city: "Faketon",
      state: "MA",
      zip: "12345"
    },
    {
      street: "1 Some Other Street",
      city: "Boston",
      state: "MA",
      zip: "12345"
    }
  ]
}
```

No way of accessing the embedded details `addresses` as stand-alone entities

One-to-Many

Using **references** provides more flexibility. This method can also be used to describe Many-to-Many relationships without the need of a join table.

```
> db.publishers.findOne()
{
  name: "O'Reilly Media",
  founded: 1980,
  location: "CA",
  books: [123456789, 234567890, ...]
}
```

```
> db.books.find({ _id: { $in: [123456789, 234567890, ...] }})
{
  _id: 123456789,
  title: "MongoDB: The Definitive Guide",
  author: [ "Kristina Chodorow", "Mike Dirolf" ],
  published_date: ISODate("2010-09-24"),
  pages: 216,
  language: "English"
}

{
  _id: 234567890,
  title: "50 Tips and Tricks for MongoDB Developer",
  author: "Kristina Chodorow",
  published_date: ISODate("2011-05-06"),
  pages: 68,
  language: "English"
}
```

One-two-Squillions

```
> db.hosts.findOne()
{
  _id : ObjectID('AAAB'),
  name : 'gaps.heig-vd.ch',
  ipaddr : '127.66.66.66'
}
```

```
> db.logs.find({ host: ObjectID('AAAB') }).sort({time : -1}).limit(5000)
{
  time : ISODate("2014-03-28T09:42:41.382Z"),
  message : 'cpu is on fire!',
  host: ObjectID('AAAB')       // Reference to the Host document
}
{
  time : ISODate("2014-03-28T09:42:40.123Z"),
  message : 'all clear for now',
  host: ObjectID('AAAB')       // Reference to the Host document
}
...
```

Insert data in MongoDB

- To insert data in MongoDB, you simply have to provide a **JSON document** (with an **arbitrary structure**).
- The documents in the collection do not have to all have the same structure (this is why we talk about a **schemaless** database).

```
db.users.insertOne(          ← collection
   {
      name: "sue",           ← field: value
      age: 26,               ← field: value
      status: "pending"      ← field: value
   }
)
```

} document

Insert methods

db.collection.insertOne(Document)	Inserts a single document into a collection.
db.collection.insertMany(Array)	inserts multiple documents into a collection.
db.collection.insert(Array|Document)	inserts a single document or multiple documents into a collection.

Query MongoDB

Read operations retrieves documents from a collection.

```
db.users.find(                          ← collection
   { age: { $gt: 18 } },                ← query criteria
   { name: 1, address: 1 }              ← projection
).limit(5)                              ← cursor modifier
```

- **query criteria** — to filter documents. support query on nested fields, arrays, arrays of embedded documents and operators
- **projection** - to restrict returned fields. see project fields from query results
- **cursor modifier** - to sort, limit, etc. see cursor methods

Query methods

db.collection.find(query, projection)	Selects documents in a collection or view and returns a cursor to the selected documents
db.collection.findOne(query, projection)	Returns one document that satisfies the specified query criteria on the collection or view.

> Insert some data, then practice with queries criteria, projections and cursor methods.

Update and delete data in MongoDB

```
db.users.updateMany(                    <-- collection
   { age: { $lt: 18 } },                <-- update filter
   { $set: { status: "reject" } }       <-- update action
)

db.users.deleteMany(                    <-- collection
   { status: "reject" }                 <-- delete filter
)
```

- **update/delete filters** — use the same syntax as read operations.
- **update action** — contains different operators such as `$set`, `$push`, `$inc`, etc.. see update operators

Update and delete data in MongoDB

db.collection.updateOne()	Updates a single document within the collection based on the filter
db.collection.updateMany()	Updates multiple documents within the collection based on the filter.
db.collection.deleteOne()	Removes a single document from a collection.
db.collection.deleteMany()	Delete all documents that match a specified filter.
db.collection.remove()	Delete a single document or all documents that match a specified filter.

References

- Install MongoDB
- 6 Rules of Thumb for MongoDB Schema Design: Part 1
- MongoDB CRUD Operations

How do I access mongoDB from Node.js

You need a driver

In the **Java ecosystem**, it is possible to interact with a RDBMS by using a JDBC driver:

- The program loads the driver.
- The program establishes a connection with the DB.
- The program sends SQL queries to read and/or update the DB.
- The program manipulates tabular result sets returned by the driver.

With **Node.js and mongoDB**, the process is similar:

- There is a Node.js driver for mongoDB (in fact, there are several).
- A Node.js module can connect to a mongoDB server and issue queries to manipulate collection and documents.

Example 1: connect and insert

```
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017/demo')
  .then((client) => {
    const db = client.db();
    const collection = db.collection('test');

    collection.insertOne({ hello : 'doc1' });
    collection.insertOne({ hello : 'doc2'});
    collection.insertMany([
      { hello: 'doc3' },
      { hello: 'doc4'},
    ]);

  });
```

Example 2: query

```
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017/demo')
  .then((client) => {
    const collection = client.db().collection('test');
    const docs = [{ doc: 1 }, { doc: 2 }, { doc: 3 }];

    collection.insertMany(docs)
      .then(() => {
        // beware of memory consumption!
        collection.find().toArray((err, items) => { });

        // better when many documents are returned
        const stream = collection.find({ doc: { $ne: 2 } })
          .stream();

        stream.on("data", (item) => { });
        stream.on("end", () => {});

        // special case when only one document is expected
        collection.findOne({ doc: 1 }, (err, item) => { });
      });
  });
```

Object Document Mapping with Mongoose

ORM - Object Relational Mapping

In the **Java EE ecosystem**, you may have seen how the **Java Persistence API** (JPA) specifies a standard way to interact with Object-Relational Mapping (ORM) frameworks.

- The developer **first** creates an **object-oriented domain model**, by creating Entity classes and using various annotations (@Entity, @Id, @OneToMany, @Table, etc.)
- He **then** uses an **Entity Manager** to **C**reate, **R**ead, **U**pdate and **D**elete objects in the DB.
- The ORM framework takes care of the details: **it generates the schema** and the **SQL queries**.

Mongoose: an ORM for MongoDB

In the **Javascript ecosystem**, we have similar mechanisms.

- There is **data mapping tools** such as mongoose
- It is more appropriate to talk about an Object-Document Mapping tool, rather than an ORM.

> Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in **type** casting, **validation**, **query building**, business logic **hooks** and more, out of the box.

Mongoose basics

Schemas maps to a MongoDB collection and defines the shape of documents

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/demo');

const { Schema } = mongoose
const catSchema = new Schema({ name: String })
```

Models are fancy constructors compiled from our Schema definitions

```
const Cat = mongoose.model('Cat', catSchema);
```

Mongoose documents represent a one-to-one mapping to documents as stored in MongoDB.

```
const kitty = new Cat({ name: 'Zildjian' });
kitty.save().then(() => console.log('meow'));
```

Mongoose: Schemas

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var blogSchema = new Schema({
  title: String,
  author: { type: String, required: true },
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs:  Number
  },
});
```

- Schemas provides built-in type casting
- Schemas provides built-in validators `required: true`. It is possible to add custom validators
- And more...

Mongoose: Models

An instance of a model is called a document. Models are responsible for...

- creating documents: `new Model()`, `Model.create()`
- reading documents: `Model.find()`, `Model.findById()`, etc..
- updating documents: `Model.updateOne()`, `Model.findOneAndUpdate()`, etc..
- deleting documents: `Model.remove()`, `Model.removeById()`, etc..

...from the underlying MongoDB database.

> Note: When doing `const doc = new Model()`, you get an instance `doc` but nothing is persisted to the database yet. You need to call `doc.save()` or use `Model.create(doc)` to save data to the database

https://mongoosejs.com/docs/api.html#Model

Mongoose: Queries

Here is an example of how can chain queries conditions

```
Person
  .find({ occupation: /host/ })
  .where('name.last').equals('Ghost')
  .where('age').gt(17).lt(66)
  .where('likes').in(['vaporizing', 'talking'])
  .limit(10)  // at most 10 documents
  .sort('-occupation')
  .select('name occupation')  // projection (select some fields)
  .exec(callback);  // run the query
```

> Use `.exec()` or `.then()` to actually **run** the query

Mongoose: Documents

“Mongoose documents represent a one-to-one mapping to documents as stored in MongoDB. Each document is an instance of its Model.”

You can also use documents to perform operations like

- saving: `doc.save()`.
- updating: `doc.field = 4`, `doc.set({ field: 4 })`
- accessing fields: `doc.field`, `doc.toJSON()`, `doc.toObject()`
- and more

https://mongoosejs.com/docs/documents.html

Mongoose and Promises

“Mongoose async operations, like `.save()` and queries, return thenables. This means that you can do things like `MyModel.findOne({}).then()` and `await MyModel.findOne({}).exec()` if you're using async/await.”

```
var gnr = new Band({
  name: "Guns N' Roses",
  members: ['Axl', 'Slash']
});

var promise = gnr.save();
assert.ok(promise instanceof Promise);

promise.then(function (doc) {
  assert.equal(doc.name, "Guns N' Roses");
});
```

https://mongoosejs.com/docs/promises.html#built-in-promises

Mongoose and Promises

"Mongoose queries are not promises. They have a `.then()` function for co and async/await as a convenience. If you need a fully-fledged promise, use the `.exec()` function."

```
var query = Band.findOne({name: "Guns N' Roses"});
assert.ok(!(query instanceof Promise));

// A query is not a fully-fledged promise,
// but it does have a `.then()`.
query.then(function (doc) {
  // use doc
});

// `.exec()` gives you a fully-fledged promise
var promise = query.exec();
assert.ok(promise instanceof Promise);

promise.then(function (doc) {
  // use doc
});
```

https://mongoosejs.com/docs/promises.html#queries-are-not-promises

heig-vd

Webcasts

- Bootcamp 4.1: Intro aux webcasts "MongoDB"
- Bootcamp 4.2: prise en main de MongoDB
- Bootcamp 4.3 (a): identification de la source de données JSON
- Bootcamp 4.3 (b): utilisation de request-promise pour interroger l'API REST
- Bootcamp 4.4: utilisation du driver node.js MongoDB
- Bootcamp 4.5: implémentation de la chaîne de promesses