



Data Management using mongoDB® & Mongoose



Course Roadmap



Web Client

Request

Response



Web Server

Frontend development

HTML for page structure



CSS for styling



JavaScript for interaction



UI Components



Backend development

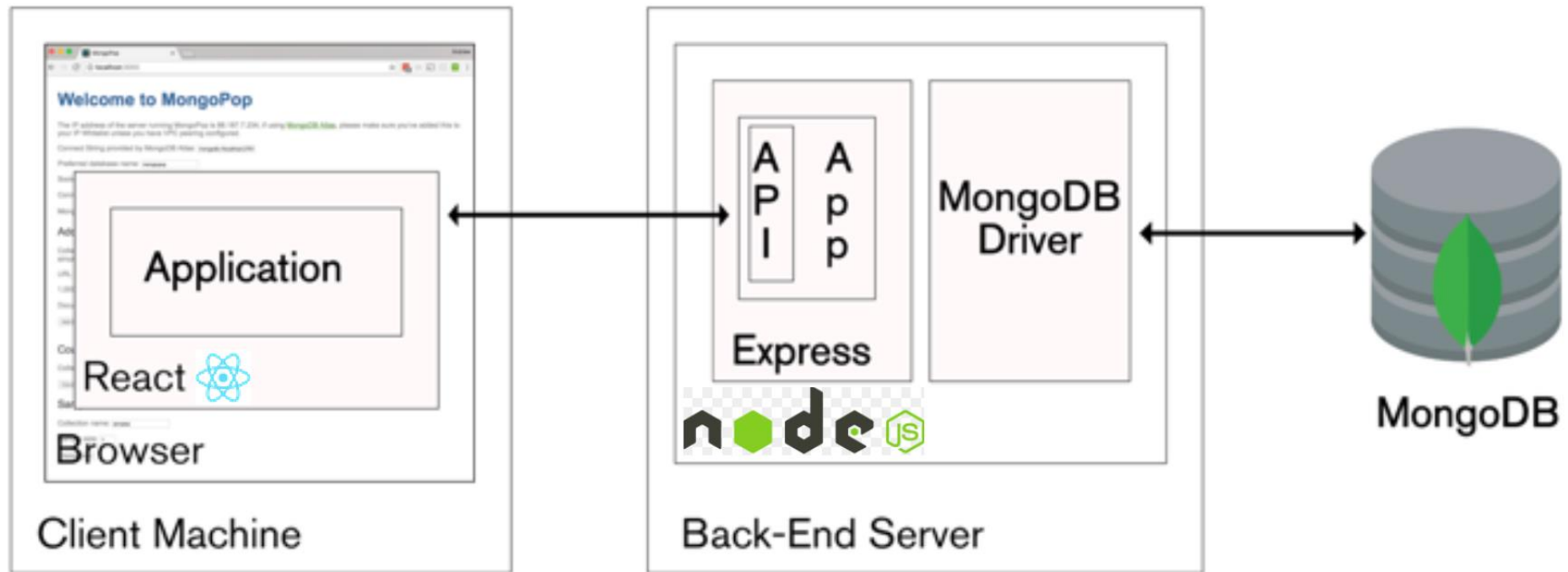
Web API



Data Management



MERN (MongoDB, Express, React, Node.js)



JavaScript is the common language throughout the MERN stack, and **JSON** is the common data format



Outline

1. [Introduction to MongoDB](#)
2. [Document Schema Design](#)
3. [Introduction to Mongoose](#)
4. [CRUD Operations](#)
5. [Aggregation Queries](#)

Introduction to



mongoDB®

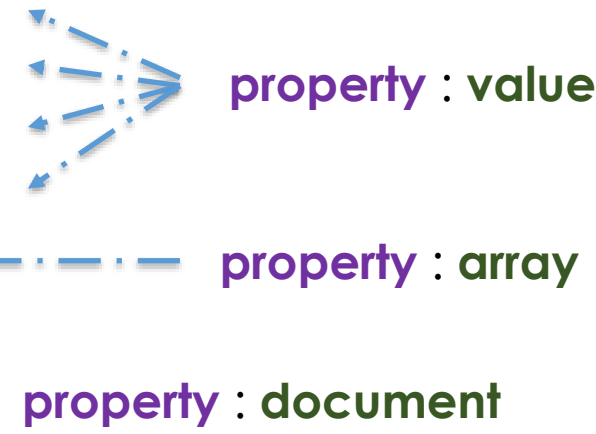


What is MongoDB?

- MongoDB is an open-source **Document Oriented Database**
 - **Uses a document data model**: Stores data as JSON documents (instead of rows and columns as done in a relational database)
 - **Arrange documents in collections** (documents can vary in structure)
 - **API to query and manage documents**
- Better alternative data management solution for Web applications compared to using a Relational Database

Document

```
{
  "isbn" : "123",
  "title": "Mr Bean and the Forty Thieves",
  "category": "Fun",
  "pages": 250
  "authors": ["Mr Bean", "Juha Dahak"],
  "publisher": {
    "name": "MrBeanCo",
    "country": "UK"
  }
}
```



The diagram illustrates the structure of a JSON document with annotations:

- property : value**: Points to the "category": "Fun" pair.
- property : array**: Points to the "authors": ["Mr Bean", "Juha Dahak"] pair.
- property : document**: Points to the "publisher" object.

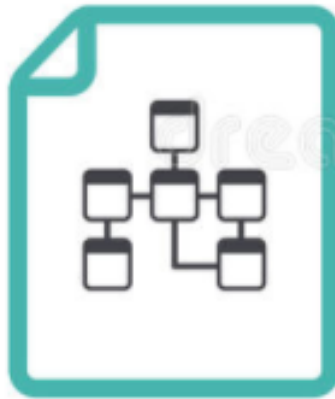
- **Document = JSON object**
- **Document = set of key-value pairs**
- **Basic unit of data** in MongoDB
- Analogous to **row** in a relational database

Collection

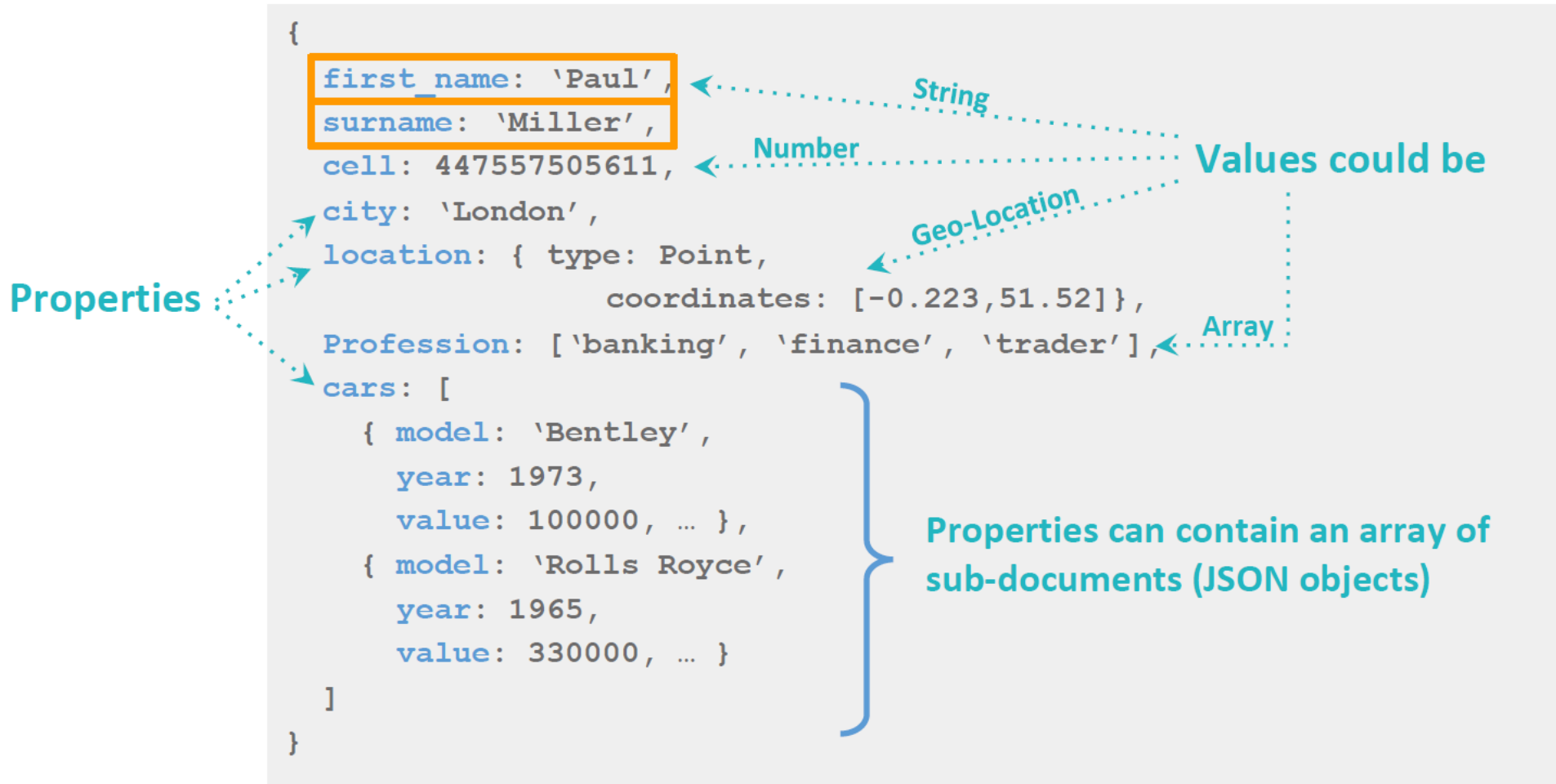
```
{
  "isbn": "123",
  "title": "Mr Bean and the Forty Thieves",
  "authors": ["Mr Bean", "Juha Dahak"],
  "publisher": {"name": "MrBeanCo", "country": "UK"},
  "category": "Fun",
  "pages": 250
}
```

- **Collection = Group** of documents
- Analogous to **table** in a relational database
- **Does not enforce** a schema
- Documents in a collection usually **have similar purpose** but they may have slightly different schema

Document Schema Design



Document can have a Complex Structure



Embedded vs Referenced documents

- Major design decision when designing a Document Schema is to decide **Embedded** vs **Referenced** subdocuments
- Decision should consider:
 - How the data will be used
 - Size of the document

Embedding

- **Advantages**

- Retrieve all relevant information in a single query/document
- Avoid implementing joins in application code => fast data retrieval
- Update related information as a single atomic operation

- **Limitations**

- Large documents mean more overhead if most fields are not relevant
- 16 MB document size limit

Referencing

- **Advantages**


- Smaller documents
- Less likely to reach 16 MB document limit
- Infrequently accessed information not accessed on every query
- No duplication of data

- **Limitations**

- Two queries required to retrieve information
- Cannot update related information atomically (in one atomic operation)

1 to 1 Relationships

Medical Procedures

```
{  
   "_id": 333,  
  "date": "2003-02-09T05:00:00",  
  "hospital": "County Hills",  
  "patient": "John Doe",  
  "physician": "Stephen Smith",  
  "procedure": "Glucose",  
  "result": {  
    "value": 97,  
    "measurement": "mg/dl"  
  }  
}
```

Embed:

- No data duplication
- Data that are read/written together lives together

← Embed – *weak entity*

One to Many Relationships

Patients

Embed

```
{
  _id: 2,
  first: "Joe",
  last: "Patient",
  addr: { ...},
  procedures: [
    {
      id: 12345,
      date: 2015-02-15,
      type: "CAT scan",
      ...},
    {
      id: 12346,
      date: 2015-02-15,
      type: "blood test",
      ...}]
}
```

OR

Patients

Reference

```
{
  _id: 2,
  first: "Joe",
  last: "Patient",
  addr: { ...},
  procedures: [12345, 12346]
}
```

Procedures

```
{
  _id: 12345,
  date: 2015-02-15,
  type: "CAT scan",
  ...}
{
  _id: 12346,
  date: 2015-02-15,
  type: "blood test",
  ...}
```

One to Many : General Recommendations

- **Embed when:**
 - **One-to-few** (e.g. customer - addresses)
 - **Often queried/updated together** in a single query (e.g., book - chapters)
 - No need to access the embedded object outside the context of the parent object (e.g., order – order items)
 - No additional data duplication introduced
- **Reference when:**
 - **1 to a large number of related items** (e.g. customer orders , book – reviews, video - comments)
 - Related data changes frequently (e.g., video – viewCount)
 - Referenced entity that is used by many others (e.g., session - **room**)
 - Document size is > 16 MB
 - Subdocument has a large number of infrequently accessed fields

1 to M Example 1

- *"We need to store user information like name, email and their addresses... yes they can have more than one."*

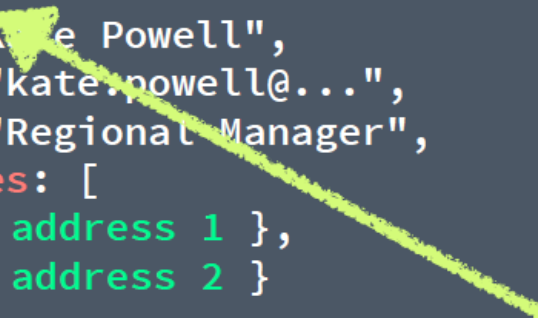
```
{
  _id: 1,
  name: "Kate Powell",
  email: "kate.powell@somedomain.com",
  title: "Regional Manager",
  addresses: [
    { street: "123 Sesame St", city: "Boston" },
    { street: "123 Evergreen St", city: "New York" }
  ]
}
```

One-to-few : embedding is the best design

1 to M Example 2

- "We have to be able to store tasks, assign them to users and track their progress..."*

```
> db.user.findOne({_id: 1})    > db.task.findOne({user_id: 1})
{                                {
  _id: 1,                        _id: 5,
  name: "Kate Powell",          summary: "Contact sellers",
  email: "kate.powell@...",     description: "Contact agents
  title: "Regional Manager",    to specify our ...",
  addresses: [                  due_date: ISODate(),
    { // address 1 },           status: "NOT_STARTED",
    { // address 2 }           user_id: 1
  ]
}
```



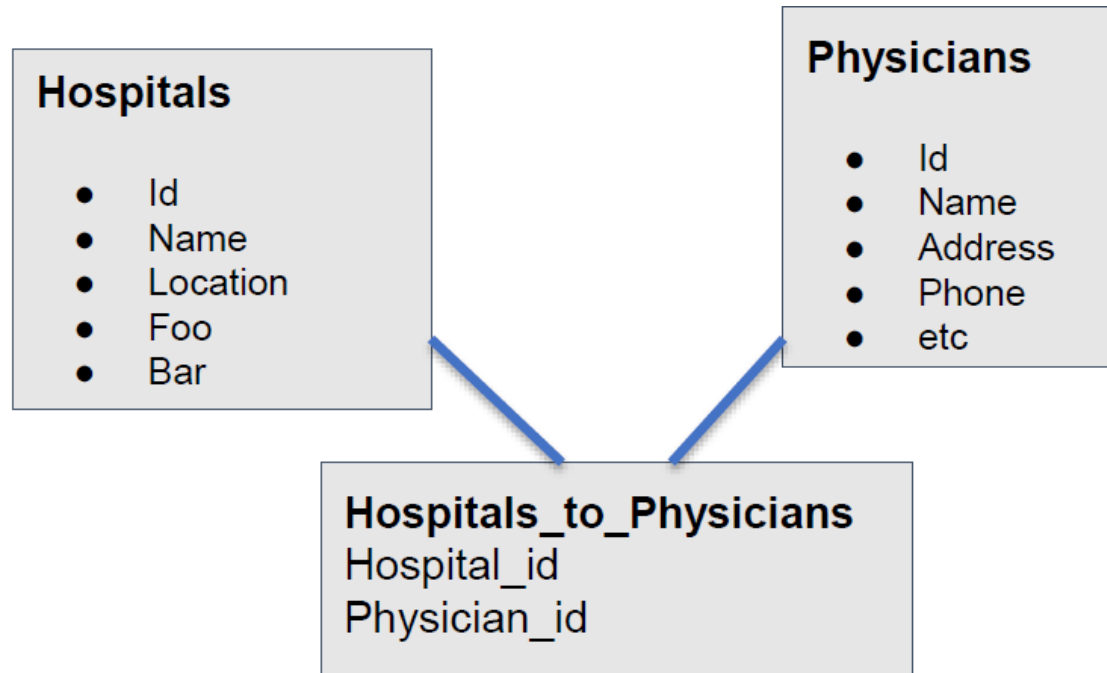
Referencing is the best design:

- Tasks are unbounded items: initially we do not know how many tasks we are going to have
 - A user can end with thousands of tasks
 - Maximum document size in MongoDB: 16 MB !
- Tasks can be retrieved without a user context

Many to Many Relationship

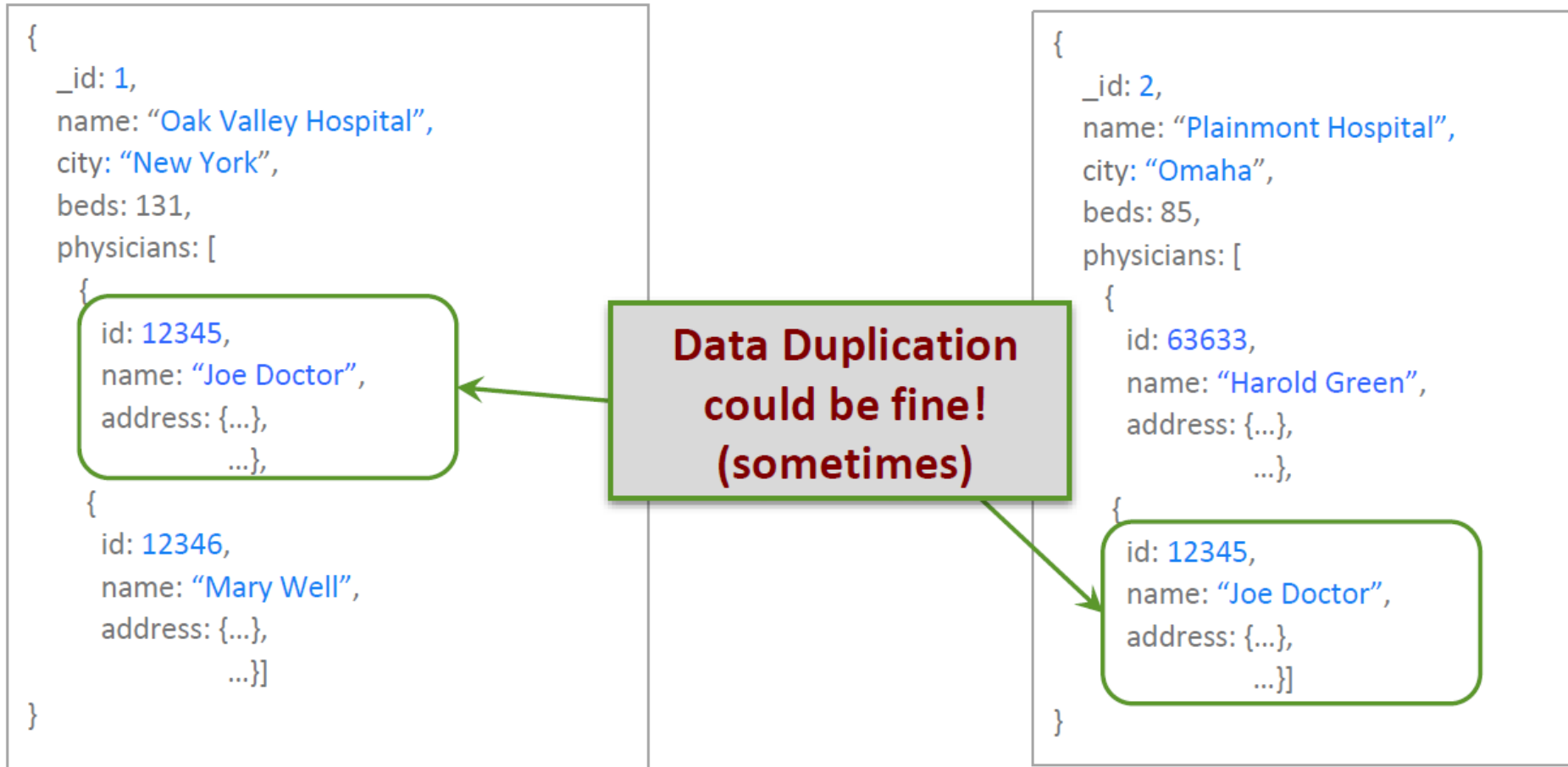
- Like a One-to-Many relationship, you can embed sub-documents or reference them.
- Which approach you take depends on data access patterns and document sizes.

The relational way:



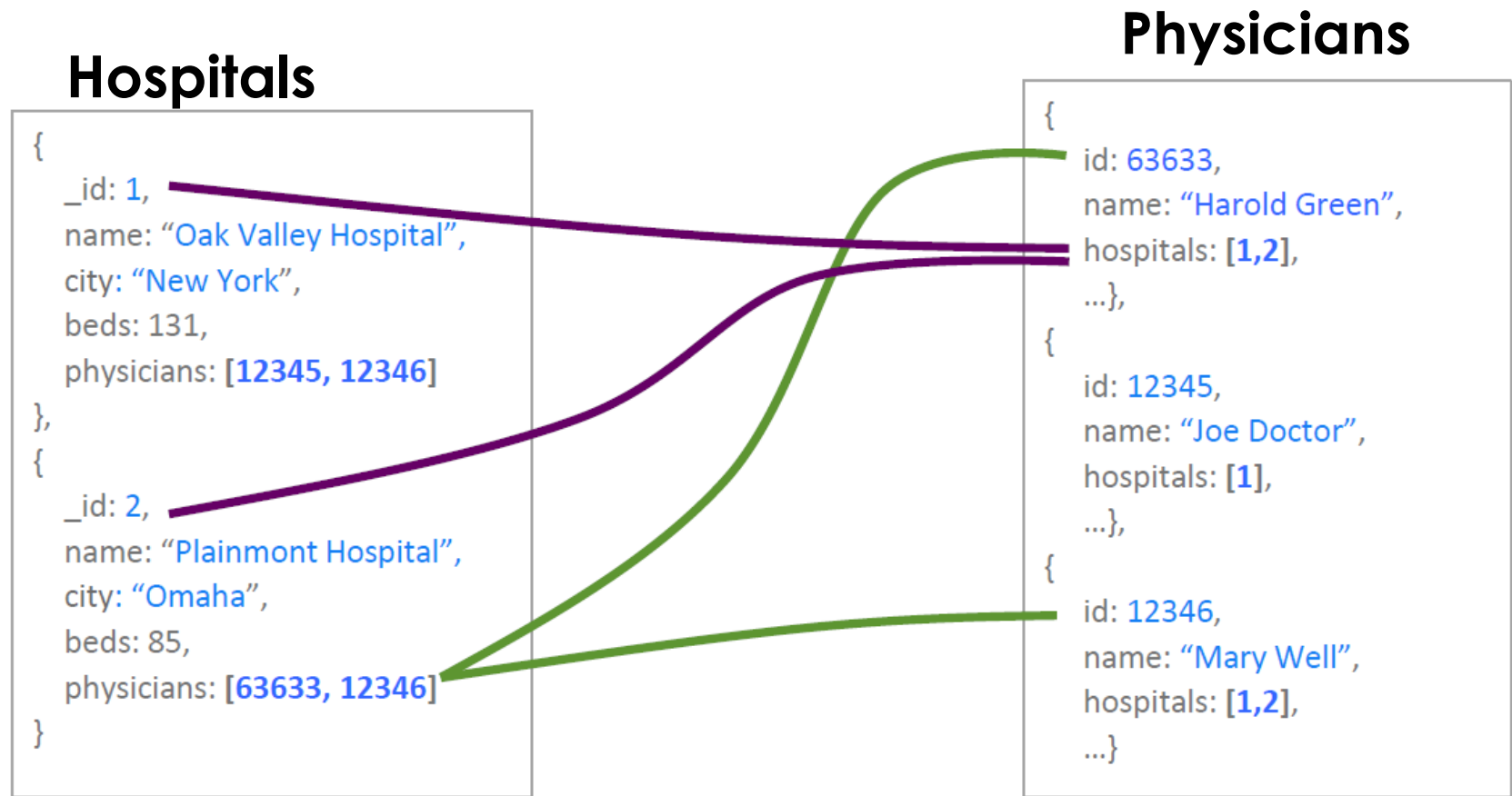
Many to Many Relationship using Embedding

Embedding **Physicians** in **Hospitals** collection



Many to Many Relationship using References

No data duplication. Hence this design is often **recommended**



Note that references can go either direction. There is no primary key/foreign key concept in MongoDB

Introduction to Mongoose



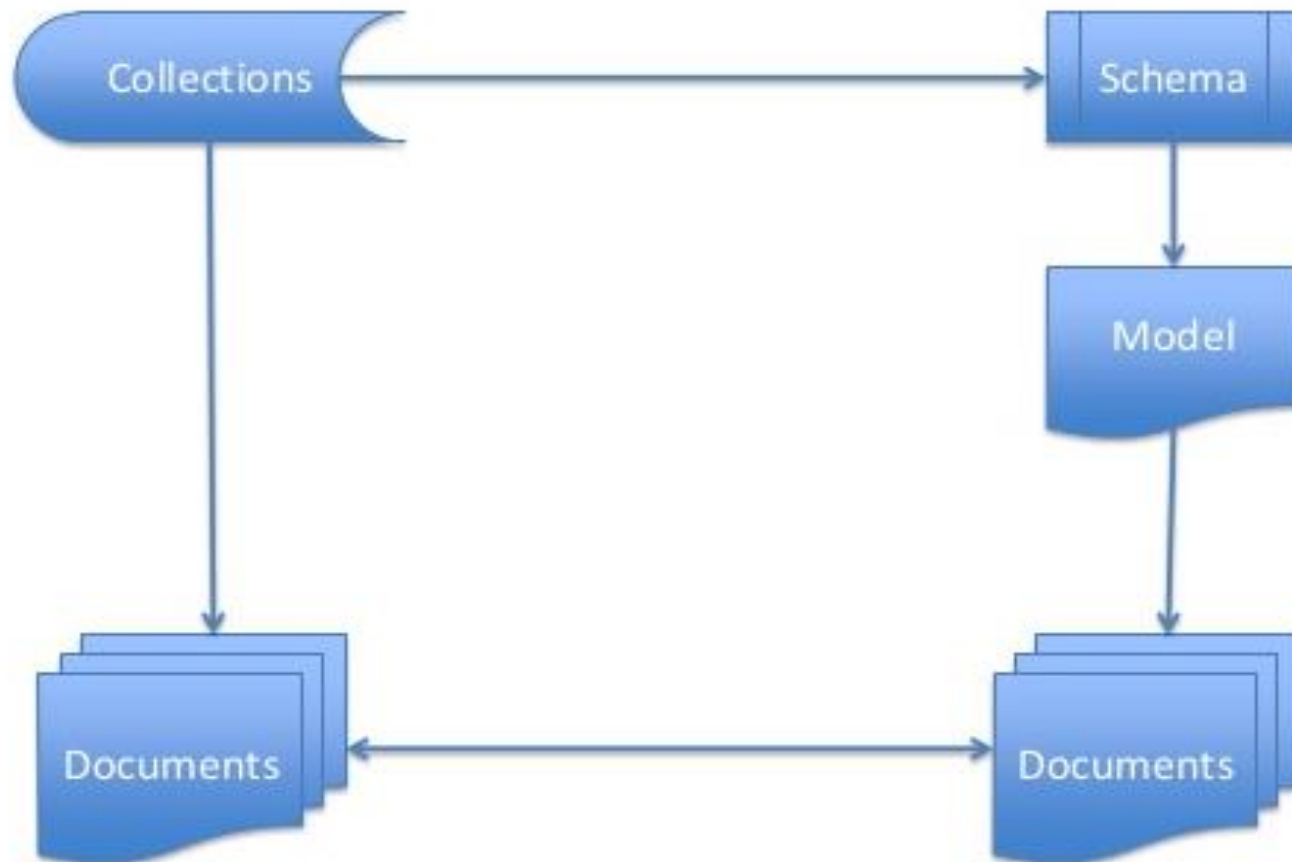
Mongoose Overview

- Mongoose is a Node.js **Object Document Mapper (ODM)** for MongoDB
 - Allows define **schemas to model** documents. Then use the **model** to read/write documents
 - A **schema** describes a document structure in terms of properties and their types.
 - You can add [validation](#), [virtual properties](#)
 - You can establish [references](#) to other collections (models)
 - A **model** is created based on a schema
 - A **model** maps to a MongoDB collection
 - A **model** = class used to run queries against collections
 - Instances of a model represent documents in MongoDB
 - Supports data validation on save
 - Allow rich querying of documents

MongoDB & Mongoose

MongoDB

Mongoose



Programming Steps

1. Require mongoose **module** `let mongoose = require('mongoose')`

2. **Connect** to MongoDB

```
let dbConnection = mongoose.connect('mongodb://localhost/dbName')
```

3. Define a **schema** for each document

```
let storeSchema = new mongoose.Schema({  
  name: String,  
  city: String  
})
```

4. Create a **model** object based

```
let Store = mongoose.model('Store', storeSchema);
```

5. Use the model to **read/write** documents

```
Store.find({})    //get all stores
```

Document Instance vs. Schema

```
{  
  "firstname" : "Simon",  
  "surname" : "Holmes",  
  "_id" : ObjectId("52279effc62ca8b0c1000007")  
}
```

**Example MongoDB
document**

```
{  
  firstname : String,  
  surname : String  
}
```

**Corresponding
Mongoose schema**

Schema Data Types

Example

Each property must have a type:

- String
- Number
- Date
- Boolean
- ObjectId
- Array

```
let reviewSchema = new mongoose.Schema({
  author: String,
  rating: {type: Number, required: true, min: 0, max: 5},
  reviewText: String,
  createdAt: {type: Date, default : Date.now}
})

let bookSchema = new mongoose.Schema({
  isbn: String,
  title: String,
  authors: [String],
  publisher: {name: String, country: String},
  category: String,
  pages: Number,
  read: {type: Boolean, default:false, required: true},
  createdAt : {
    type : Date,
    default : Date.now
  },
  reviews: [reviewSchema],
  store : [{ type : mongoose.Schema.ObjectId, ref : 'Store' }]
})
```

_id

- **_id** is the primary key that uniquely identifies each document in the collection
- It is automatically added when adding a document to the collection
- It is immutable (it cannot be changed)
- It is guaranteed to be unique across the whole database

Property Validation

- Built-in validators: required, min, max
- Can define custom validators

```
bookSchema.path('isbn').validate( value => value.length >= 3 )
```

- Validation happens on save

Virtual Property

- Define a property that won't get persisted to MongoDB

```
//define a fullName property that won't get persisted to MongoDB.  
personSchema.virtual('fullName').get(function () {  
    return this.name.first + ' ' + this.name.last;  
});
```

```
// create a document  
const student1 = new Person({  
    name: { first: 'Ali', last: 'Faleh' }  
});
```

```
console.log(student1.fullName); // Ali Faleh
```

```
/*  
If you use toJSON() mongoose will not include virtuals by default  
unless if you pass { virtuals: true }  
*/  
console.log(student1.toJSON({ virtuals: true }));
```

CRUD Operations



CREATE



READ



UPDATE



DELETE

C

R

U

D



CRUD operations

- Create → `Book.create(newBook)`
- Read → `Book.find({})`
`Book.findById(bookId)`
`Book.findOne({isbn: isbn})`
`Book.find({authors: {$in: [author]}})`
- Update → `Book.update({_id: bookId}, updatedBook)`
- Delete → `Book.remove({_id : bookId});`

Mongoose Queries

- Queries are based on finding documents with any combination of fields in a collection

```
Book.find({ category: 'Fun', pages : { $lt : 200 } })
```

- You sorting and limits the number of returned documents

```
Book.find({}).sort('isbn').limit( 5 )
```

- OR condition is also supported

```
Book.find({}).where({ category: 'Fun' }).or({pages :{ $lt : 100 } })
```

- Filter on the existence of field

```
Book.find( { reviews : { $exists: true } } )
```

QueryBuilder

- The query object allows chaining methods could chained to build a complex query

```
School.find({ name: 'Iqraa'})  
.where('state').equals('AZ')  
.where('licenses').gt(17).lt(100)  
.where('district').in(['dist1', 'dist2'])  
.limit(10)  
.populate ('owner', 'name')  
.sort('owner.name')  
.select('id name state owner.name')
```

Count and Distinct Methods

- `collection.count(query)` - returns the number of documents in the collection that match the query
- `collection.distinct(field, query)` - returns an array of all the unique values found in the passed field for the documents that match the query

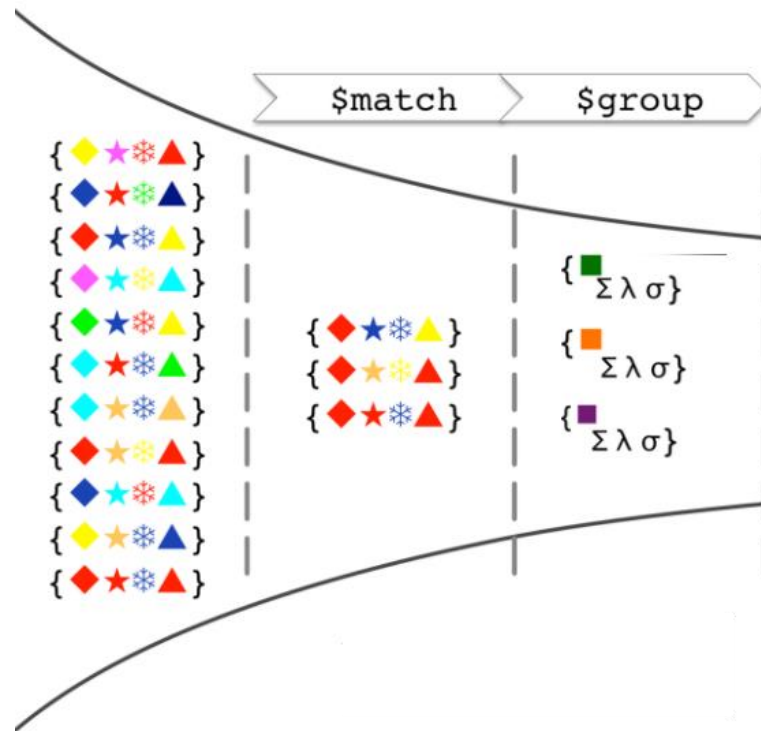
Populating Ref Property

- Population is the process of automatically replacing the specified paths in the document with document(s) from other collection(s)
- Populate sends another query for the related object

```
let bookSchema = new mongoose.Schema({  
  isbn: String,  
  title: String,  
  ...  
  store : [{ type : mongoose.Schema.ObjectId, ref : 'Store' }]  
})
```

//populate('store') will replace the store Id with the corresponding store object
Book.find({}).populate('store')

Aggregation Queries





Aggregation Queries

- Summarize data typically for reports
- How would we solve this in SQL?

SELECT **GROUP BY** HAVING

- What About MongoDB?

=> **Aggregation Pipeline**

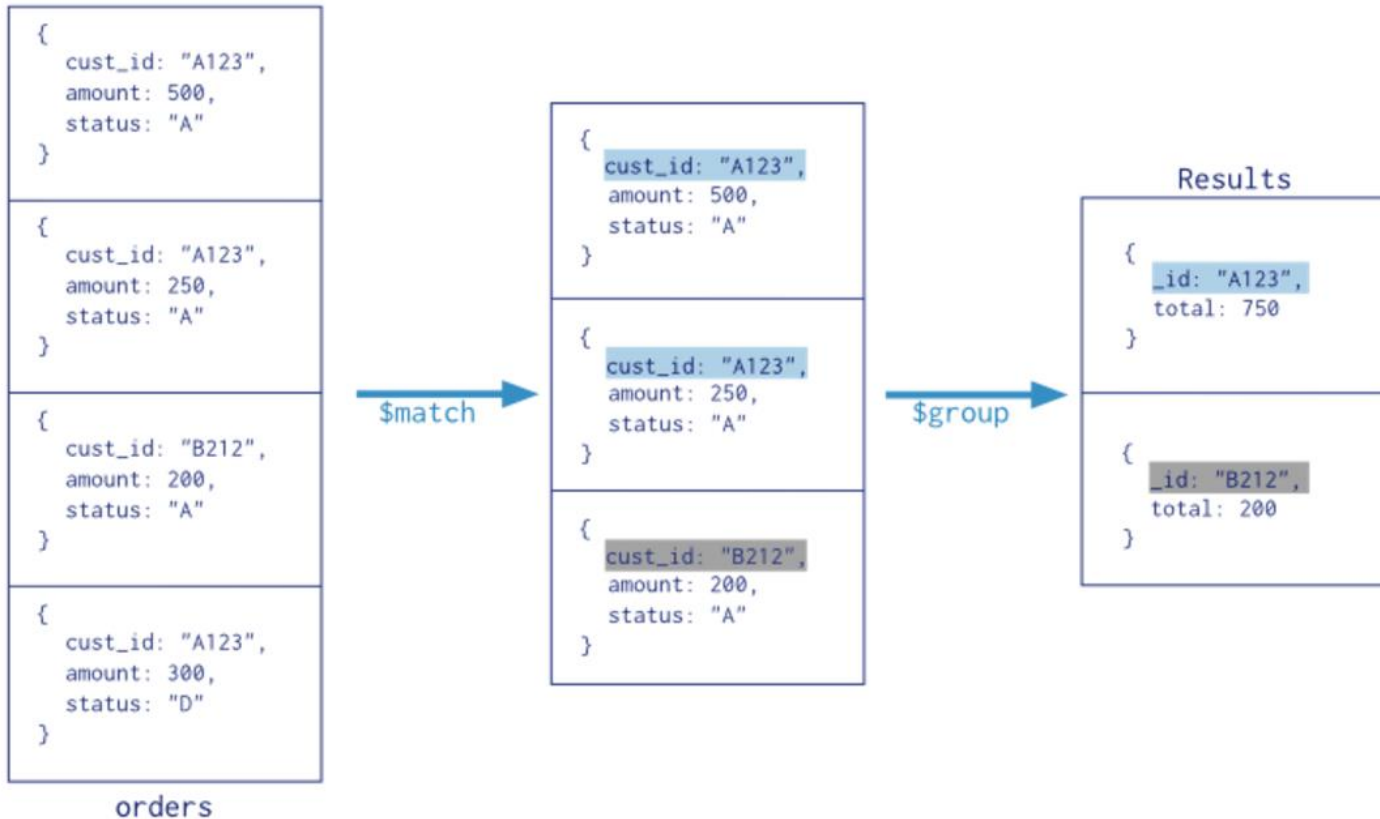


- Pipeline of functions to filter, group, and sort documents
- Operations executed in sequential order
- Output of one stage is used as an input of next

Aggregation Pipeline

- Allows filtering then grouping documents by specific fields

Collection
↓
`db.orders.aggregate([`
 `$match stage → { $match: { status: "A" } },`
 `$group stage → { $group: { _id: "$cust_id", total: { $sum: "$amount" } } }`
 `]`)



Pipeline Operators

- `$match` Filter documents
- `$group` Summarize documents
- `$sort` Order documents
- `$limit` Limit returned results

- **`$group`** specifies:
 - Properties to group by
 - Computed output properties using `$max`, `$min`, `$avg`, `$sum`

\$group Examples

- Return average GPA for all students

```
Student.aggregate([
  {
    "$group" : { "_id" : null,
    "avgGPA" : { $avg : "$gpa" }
  }}])
```

- Return total completed Credit Hours per student

```
StudentCourse.aggregate([
  {
    "$group" : { "_id" : studentId,
    "completedCHs" : { $sum : "$CourseCH" }
  }}])
```

Resources

- Mongoose Documentation

<http://mongoosejs.com/docs/>

- Queries Cheat Sheet

http://s3.amazonaws.com/info-mongodb-com/mongodb_qrc_queries.pdf

- Aggregation Queries

<https://docs.mongodb.com/manual/reference/operator/aggregation/group/>