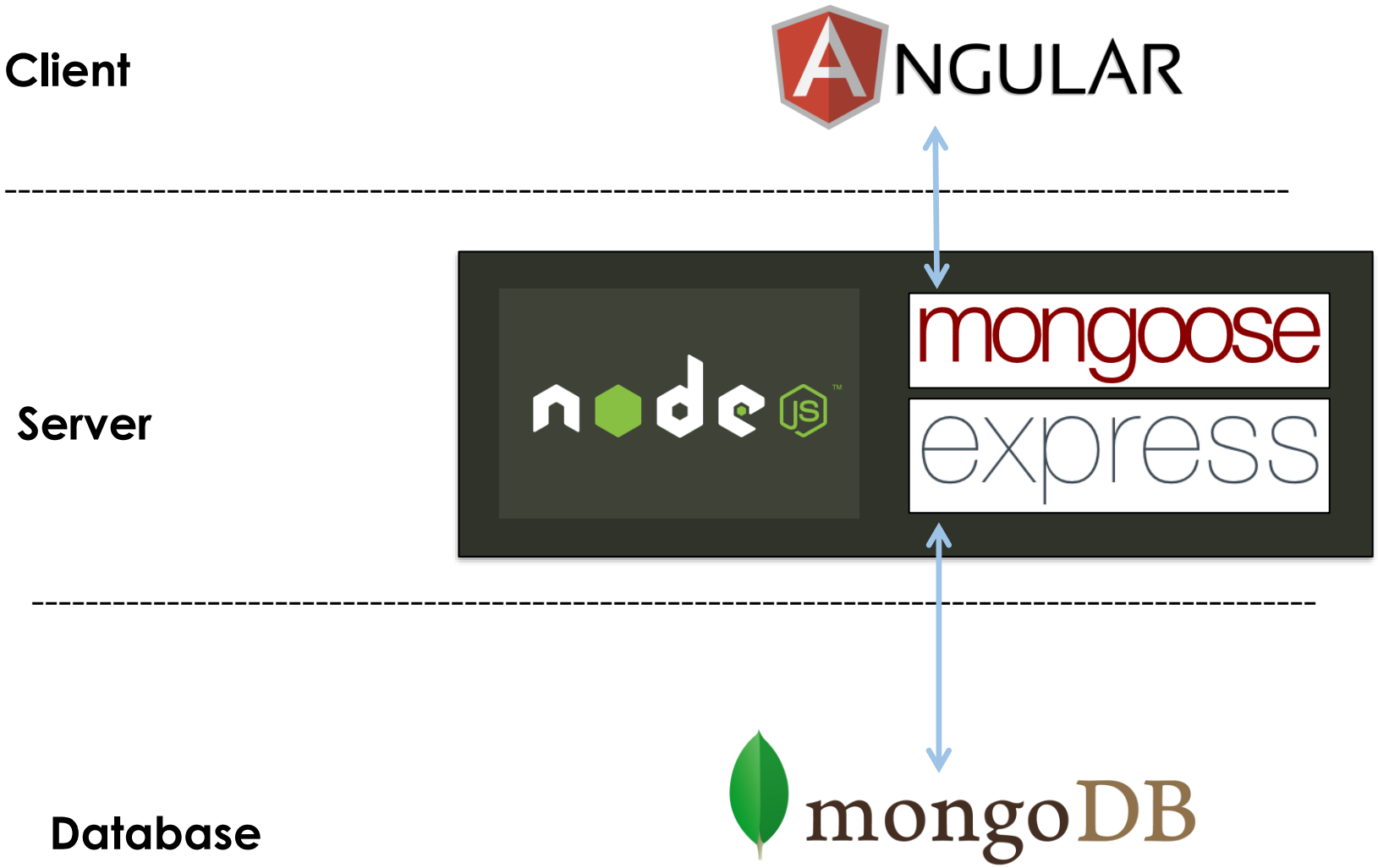
Data Management using mongoDB® & Mongoose

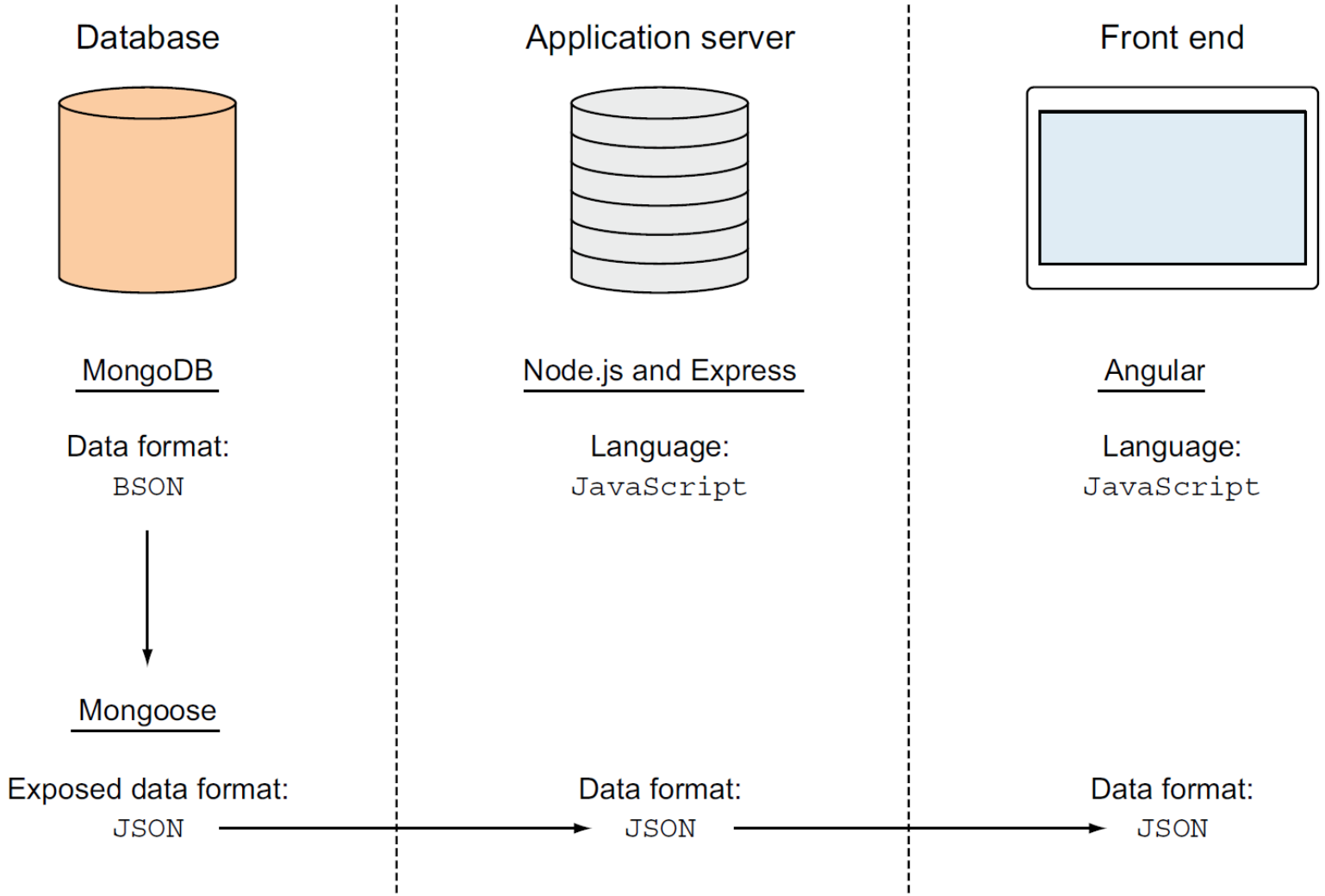
Outline

1. MEAN Stack
2. Introduction to MongoDB
3. Introduction to Mongoose
4. CRUD Operations

MEAN Stack


Client
ANGULAR
Server
node
mongoose
express
Database
mongoDB

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



Node.js **plays such a pivotal role in the MEAN (Mongo, Express, Angular, and Node) stack**

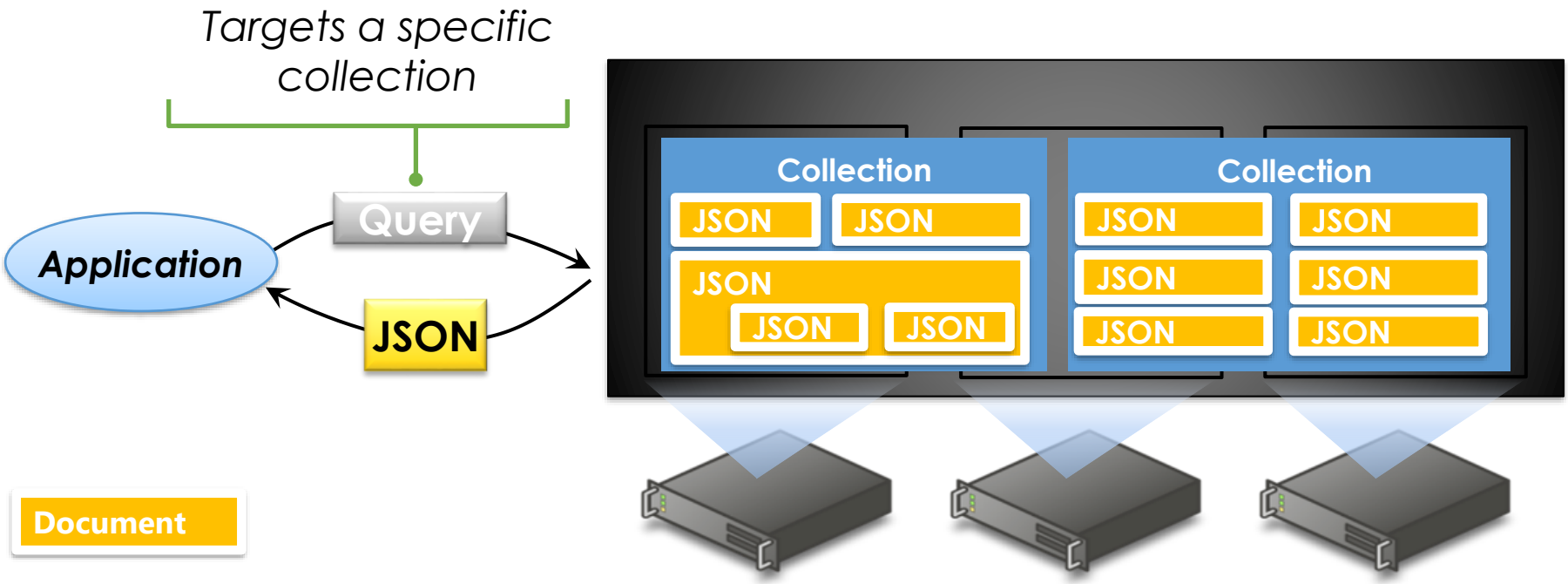
Introduction to

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
- Queries Cheat Sheet

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

MongoDB Architecture



- MongoDB
https://www.mongodb.com/download-center
- IDE
https://robomongo.org/

- MongoDB in the Cloud
https://mongolab.com/ (500MB free)

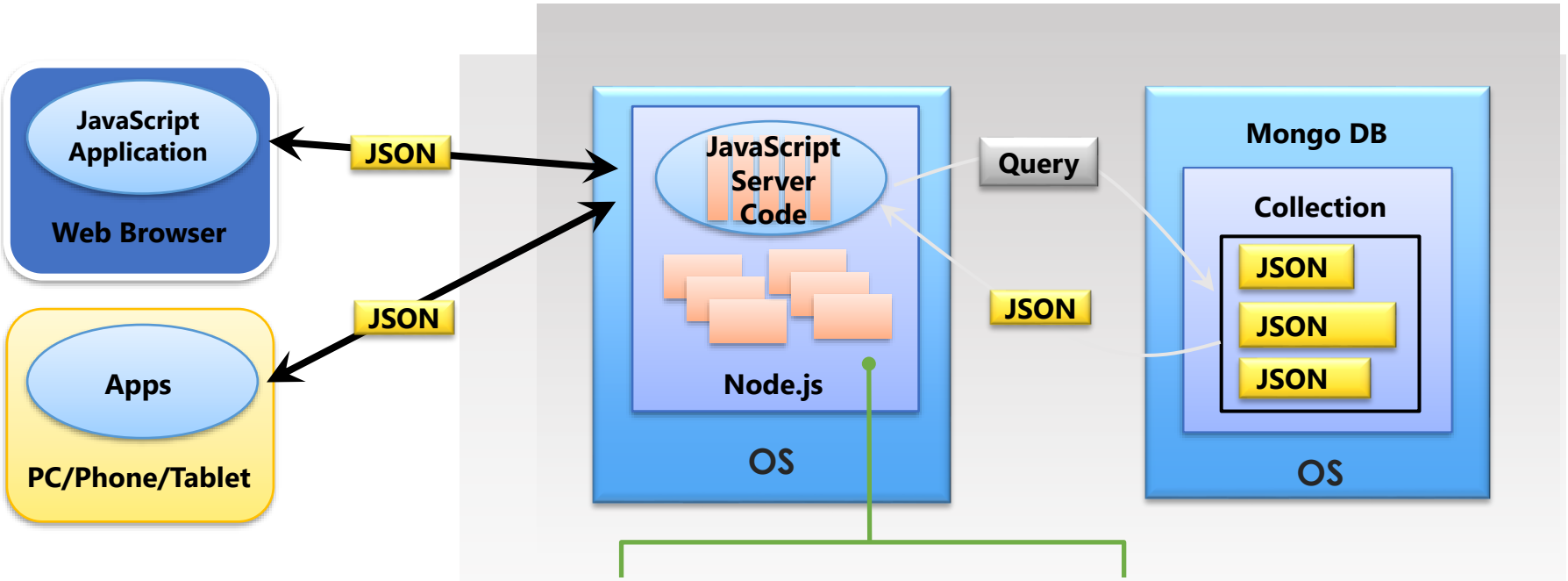
Document Data Model

Relational

MongoDB

```
{
  first_name: 'Paul',
  surname: 'Miller',
  city: 'London',
  location:
[45.123,47.232],
  cars: [
    { model: 'Bentley',
      year: 1973,
      value: 100000, … },
    { model: 'Rolls Royce',
      year: 1965,
      value: 330000, … }
  ]
}
```

JSON Storage for JavaScript Applications
The complete picture



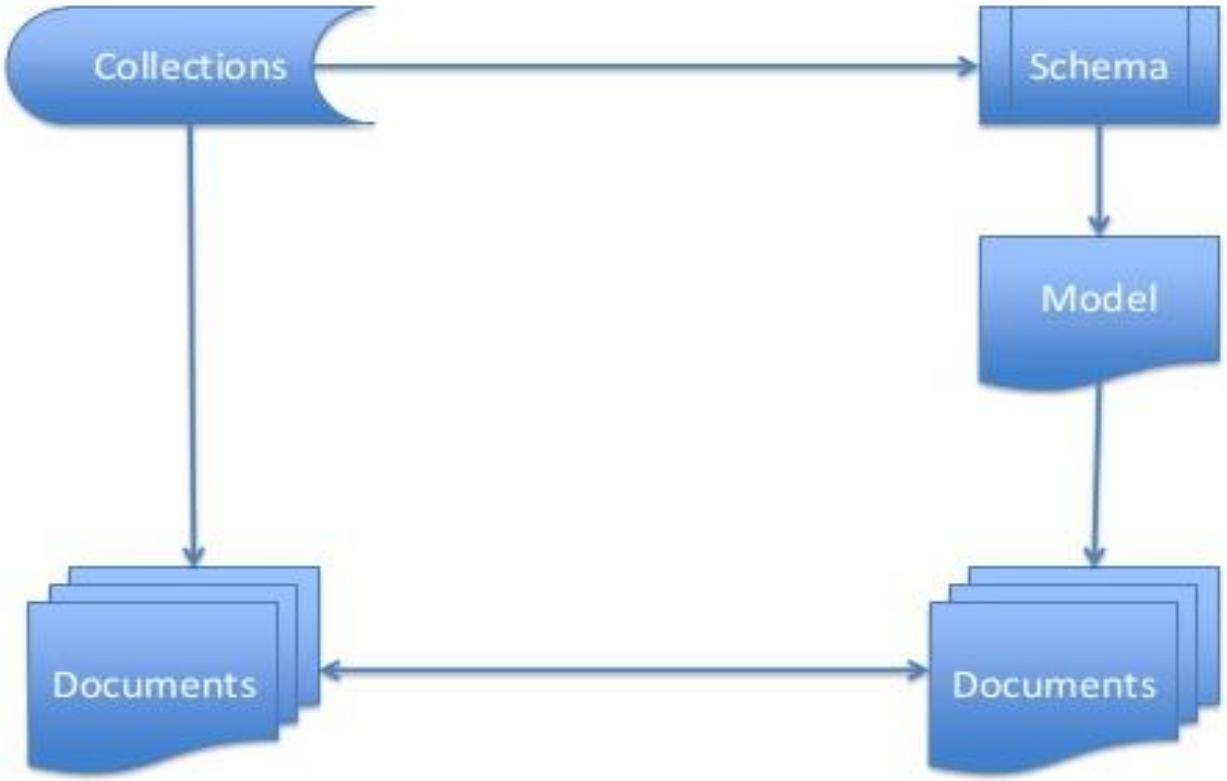
Mongoose supports connecting then reading/writing to MongoDB

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
    - A **schema** maps to a MongoDB collection
    - A **model** is created based on a schema
    - Instances of a model represent documents in MongoDB
  - Supports data validation on save
  - Allow rich querying of documents

MongoDB & ODM



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({}).exec()    //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

Each property must have a type:

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

Example

```
let reviewSchema = new mongoose.Schema({
    author: String,
    rating: {type: Number, required: true, min: 0, max: 5},
    reviewText: String,
    createdOn: {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},
    createdOn : {
        type : Date,
        default : Date.now
    },
    reviews: [reviewSchema],
    store : [{ type : mongoose.Schema.ObjectId, ref : 'Store' }]
})
```

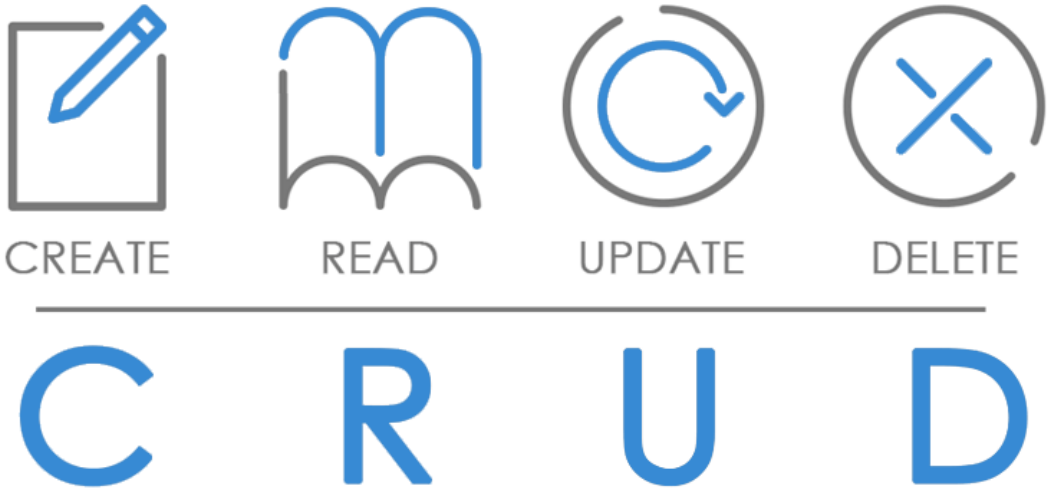
Property Validation

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

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

- Validation happens on save

CRUD Operations



CRUD operations

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

Mongoose Queries

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

```
Book.find({ category: 'Fun', pages : { $gt : 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({ $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

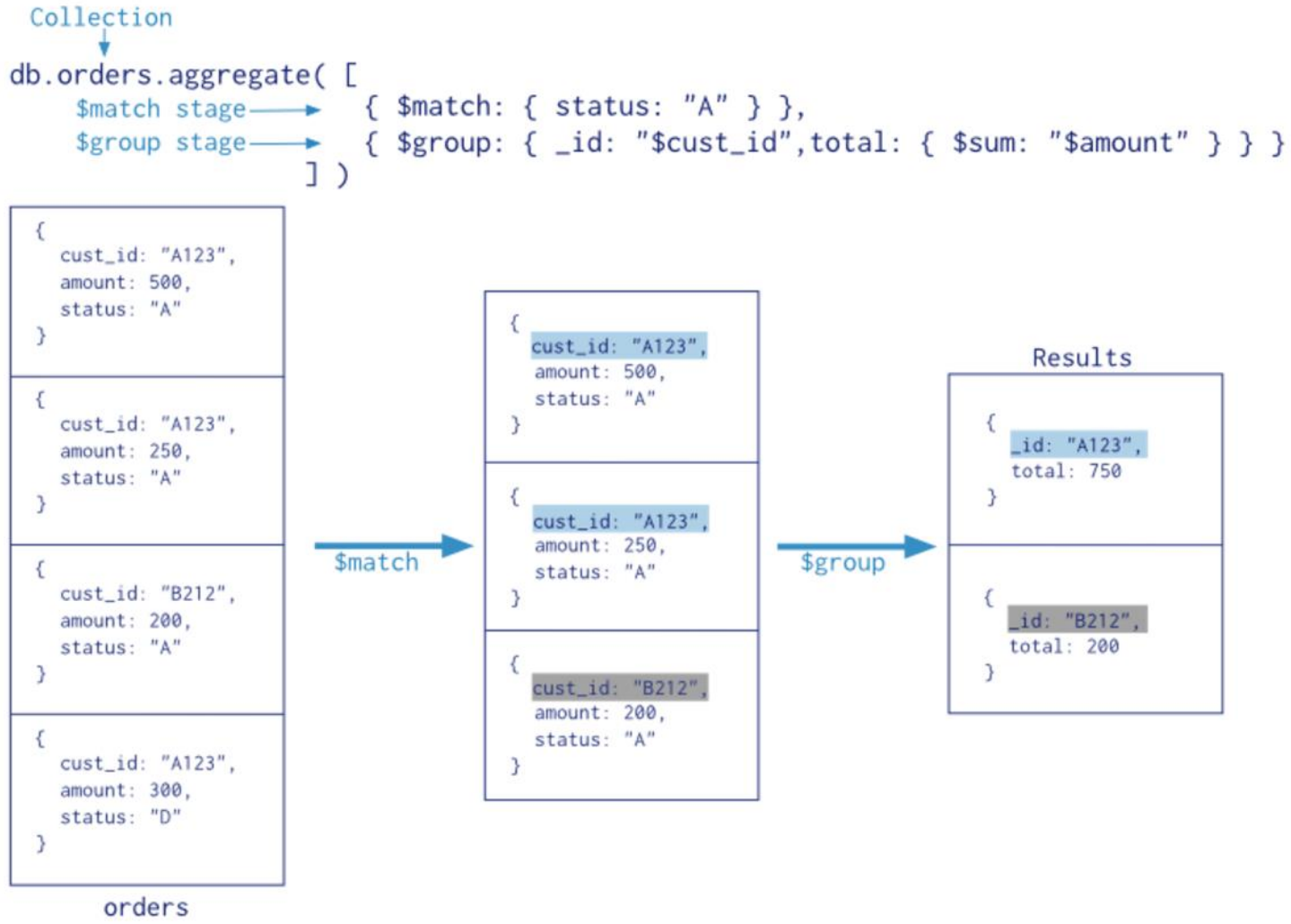
```
School.find({ name: 'Iqraa'})
.where('state').equals('AZ')
.where('licenses').gt(17).lt(100)
.where('district').in(['dist1', 'dist2'])
.limit(10)
.populate ('owner', 'name')
.sort('name')
.select('id name state')
```

Count and Distinct Methods

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

Aggregation Pipeline

- Pipeline operations: filter then grouping documents by specific field or fields



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')
```

Resources

- Queries Cheat Sheet

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

- Mongoose Documentation

http://mongoosejs.com/docs/