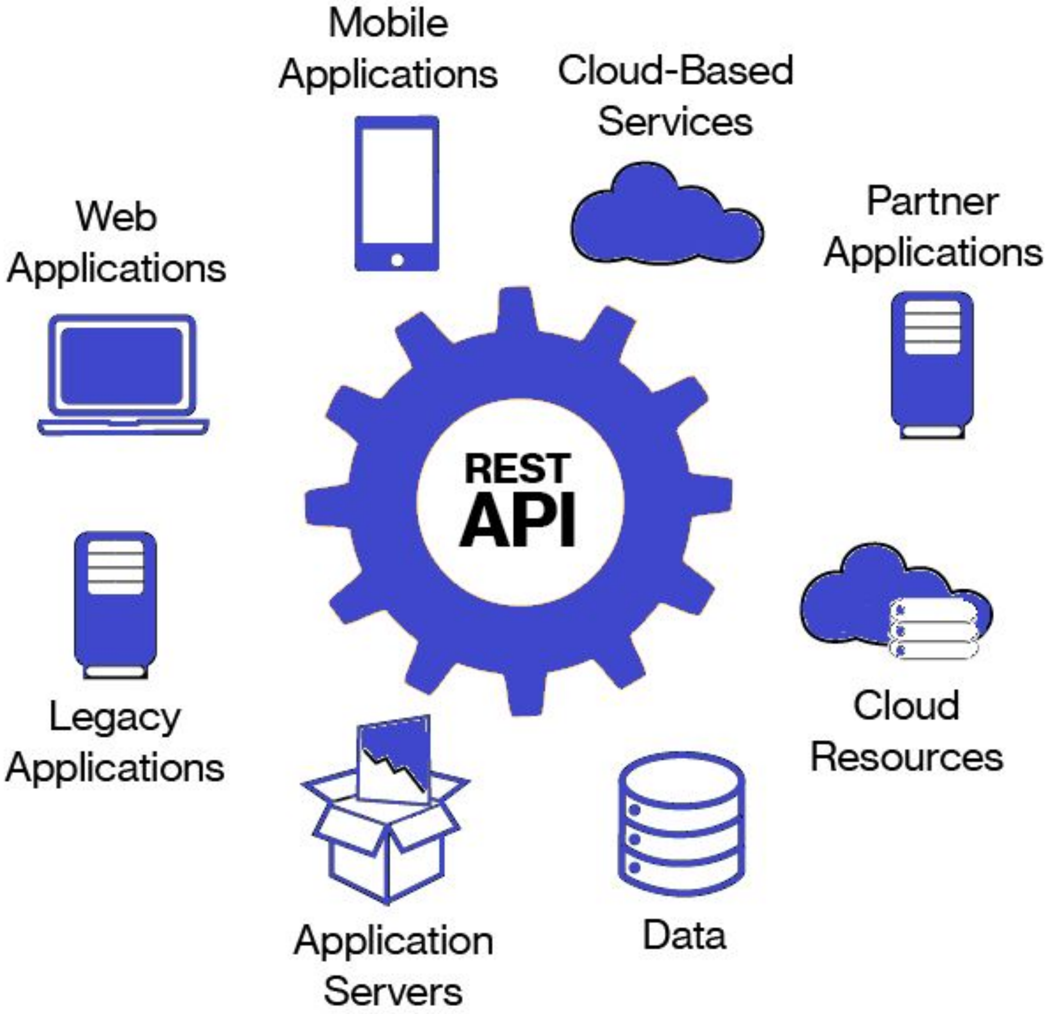
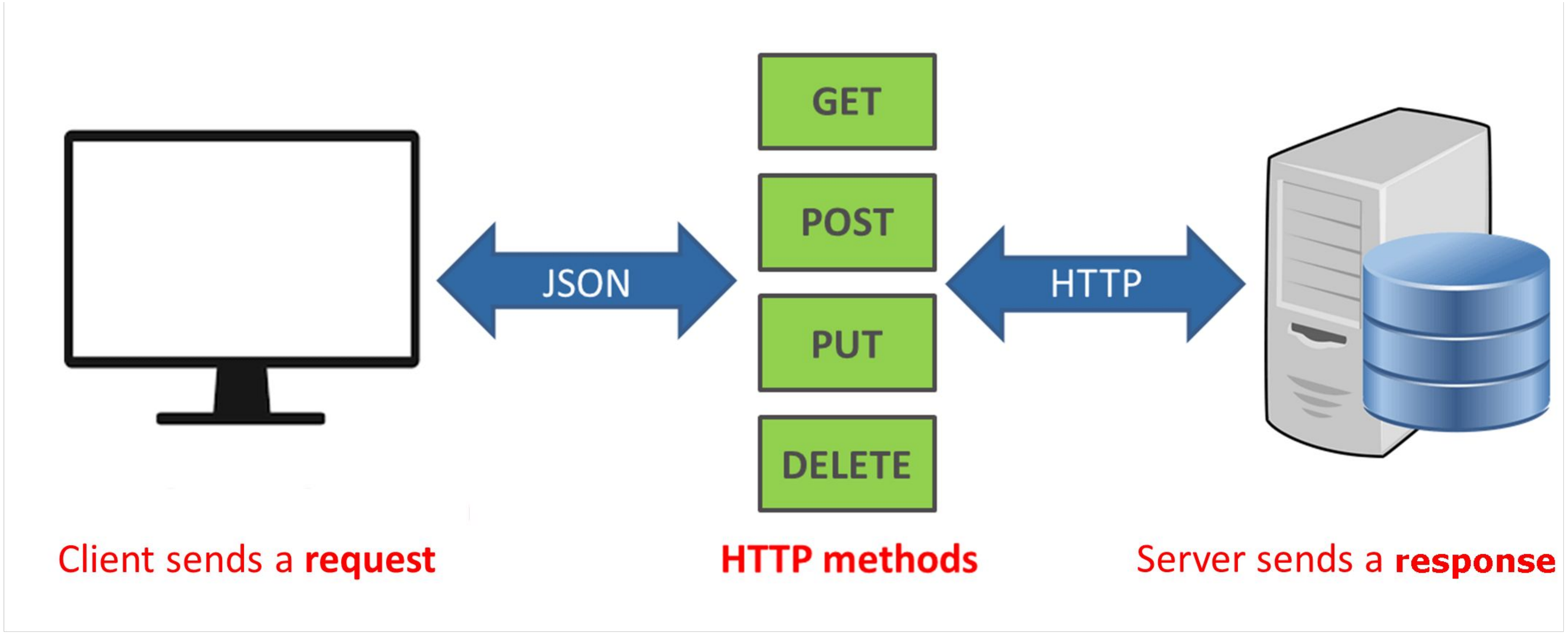
Mobile Backend Development

Restful API (1)

Wan Muzaffar Wan Hashim


Mobile Applications
Cloud-Based Services
Web Applications
Partner Applications
REST API
Legacy Applications
Cloud Resources
Application Servers
Data


JSON
GET
POST
PUT
DELETE
HTTP
Client sends a **request**
HTTP methods
Server sends a **response**

HTTP Methods

HTTP Methods	CRUD Operation	Description
POST	Add (Create)	Add Item inside database
PUT / POST	Edit (Update)	Change an Existing Resource
GET	Retrieve (Read)	Retrieve existing resource
DELETE	Delete (Delete)	Delete existing resource

HTTP Response Status Information

Status code	Meaning
1xx	Informational Code
2xx	Successful Codes
3xx	Redirection Codes
4xx	Client Error Code
5xx	Server Error Code

REST API Routing

GET - http://www.example.com/api/v1**/users**

POST - http://www.example.com/api/v1**/users**

GET - http://www.example.com/api/v1/users/1

PUT - http://www.example.com/api/v1/users/1

DELETE - http://www.example.com/api/v1/users/1

Backend programming languages

Java

php

JS

Backend framework

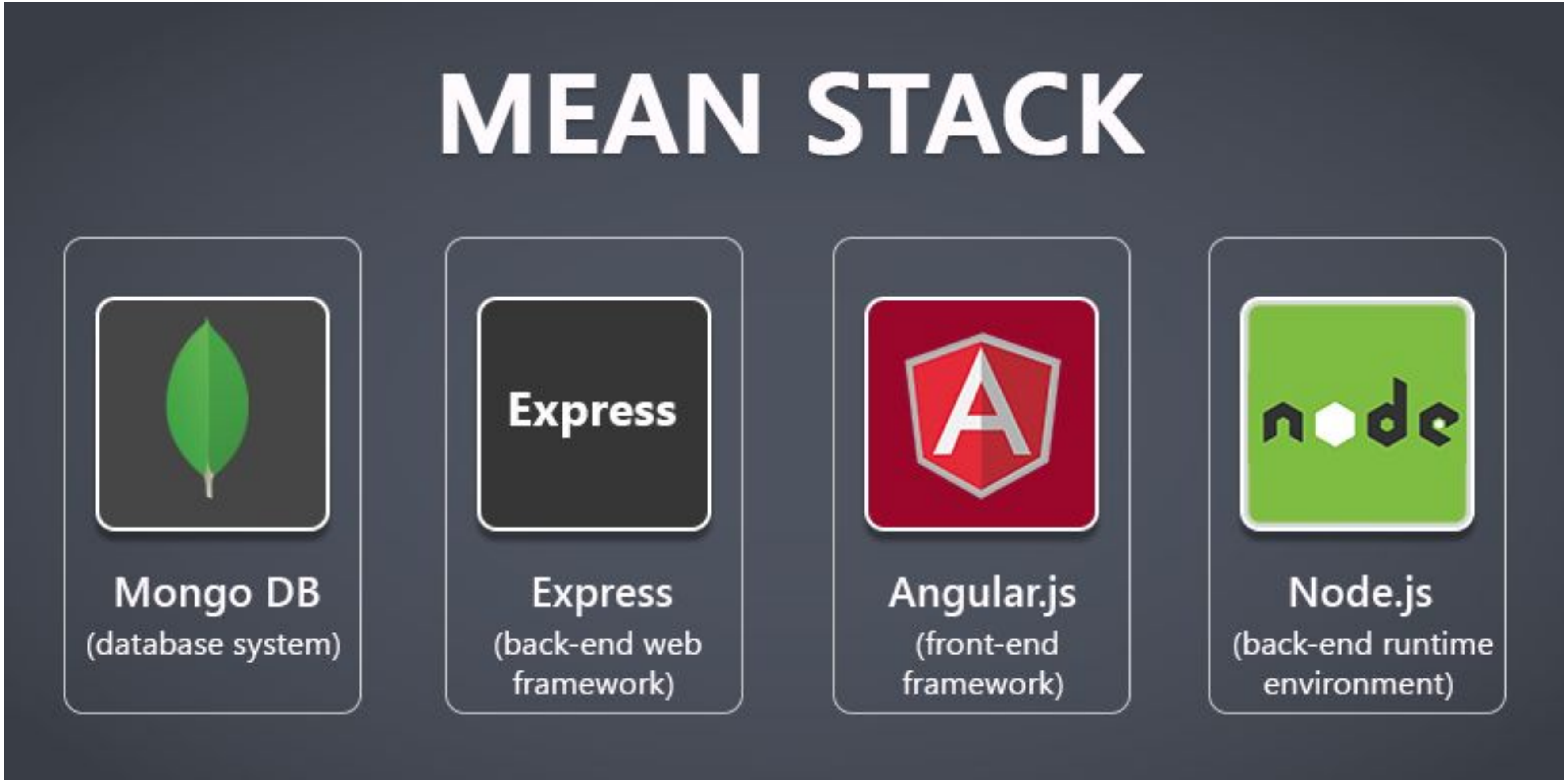
node JS®

Flask

django

Spring Boot

Laravel



MEAN STACK

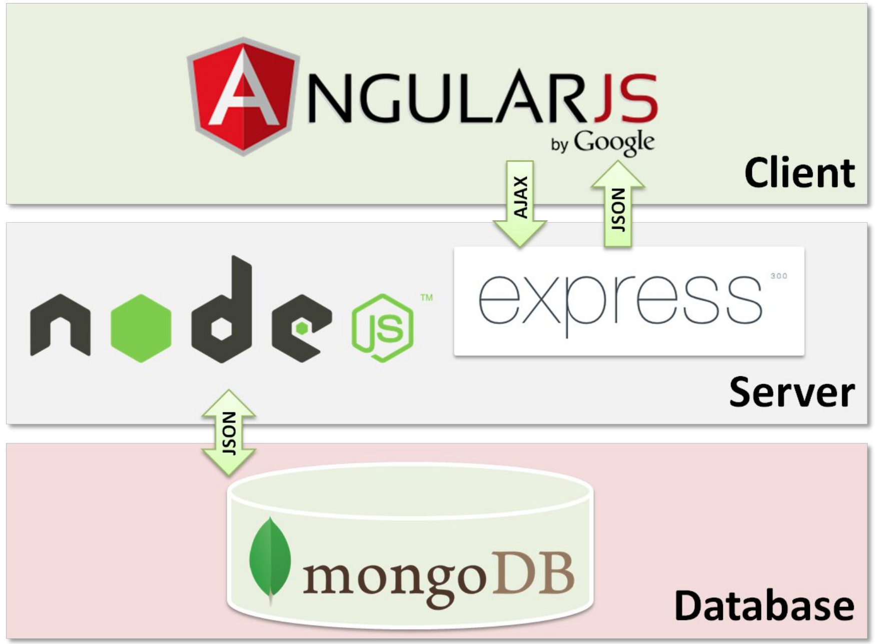
Mongo DB
(database system)

Express
(back-end web framework)

Angular.js
(front-end framework)

Node.js
(back-end runtime environment)

What is MEAN?



MEAN is an opinionated fullstack javascript framework - which simplifies and accelerates web application development.

Why MEAN?

- 100% Free
- 100% Open Source
- 100% (Javascript and HTML)
- Single language throughout the application.
- Adhere to MVC concept.
- Use of JSON as data structure, compared to before where serialization and deserialization of data structure is needed.

Express

JS

What is ExpressJS?

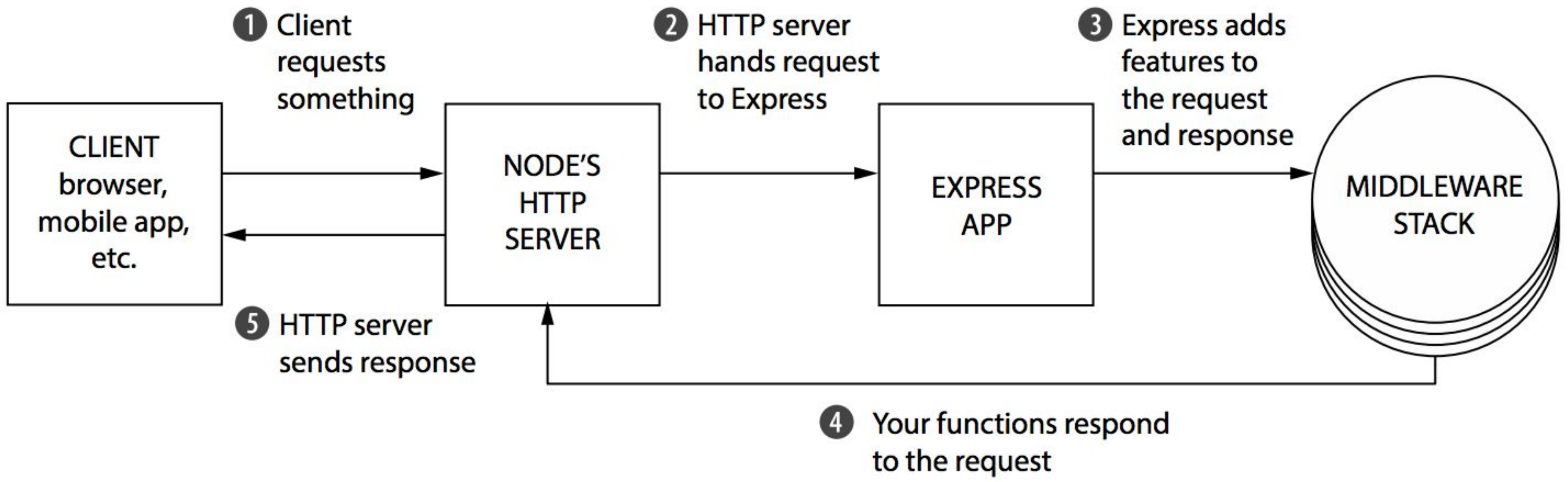
Express JS is a web application framework provides you with a simple API to build websites, webapps and backends. You need not worry about low level protocols, processes, etc.

Express provides a minimal interface to us to build our applications. It is minimal, providing us the absolutely required tools to build our app and flexible, there are numerous modules available on npm for express, which can be directly plugged into express.

How Express helps you?

- Routing
- Add helpful Node.js HTTP objects
- Dynamic HTML Views
- Middleware

Express application flow



1 Client requests something
2 HTTP server hands request to Express
3 Express adds features to the request and response
CLIENT browser, mobile app, etc.
NODE'S HTTP SERVER
EXPRESS APP
MIDDLEWARE STACK
5 HTTP server sends response
4 Your functions respond to the request

Express application flow (2)

- In Express, you write smaller functions or integrate other modules in your application.
- Express has many utilities for partitioning these smaller request handler functions.
- Request handler functions take two arguments: the request and the response.
- Node’s HTTP server provides some functionality; for example, browser’s user agent extraction, sendFile..

Getting started with Express

1) Create a new folder, call it “Hello World.”
2) Initialize the project using ‘**npm init**’
3) Using npm, install Express package inside the project.
4) Install nodemon package so that you can see the change in node without needing to refresh the server. (Refer to nodemon installation guide)

Try It: Hello World in Express JS.

```
var express = require('express');
var app = express();

app.get('/', function(req, res){
    res.send("Hello world!");
});

app.listen(3000);
```

Code Explanation

Code	Explanation
app.get(route, callback)	Define what to be done when a get request at the given route is called. The callback function has 2 parameters, *request(req)* and *response(res)*. The request object(req) represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, etc. Similarly, the response object represents the HTTP response that the express app sends when it receives a HTTP request.
res.send()	This function takes an object as input and it sends this to the requesting client. Here we are sending the string *"Hello World!"*.
app.listen(port, [host], [backlog], [callback]])	This function binds and listens for connections on the specified host and port. Port is the only required parameter here.

HTTP Methods

HTTP Method	Description
GET	The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect.
POST	The POST method requests that the server accept the data enclosed in the request as a new object/entity of the resource identified by the URI.
PUT	The PUT method requests that the server accept the data enclosed in the request as a modification to existing object identified by the URI. If it does not exist then PUT method should create one.
DELETE	The DELETE method requests that the server delete the specified resource.

Revision Exercise

1) We have used Sheetsu in module 1 to transform a Google Doc into an API.
2) Retrieve your Sheetsu API and try the different HTTP Methods:
   a) GET
   b) POST
   c) PUT
   d) DELETE

Routing in Express

When creating our Hello World project, we have created the route for our home page. Here is the syntax for creating route in Express:

app.METHOD(PATH, HANDLER)

- METHOD is any one of the HTTP verbs(get, set, put, delete).
- Path is the route at which the request will run.
- Handler is a callback function that executes when a matching request type is found on the relevant route.

Try it: Routing in Express

```
var express = require('express');
var app = express();

app.get('/hello', function(req, res){
      res.send("Hello World!");
});

app.post('/hello', function(req, res){
      res.send("You just called the post method at '/hello'!\n");
});
app.all('/test', function(req, res){
      res.send("HTTP method doesn't have any effect on this route!");
});


app.listen(3000);
```

Using Routers

1) If we define all the route in index.js, it will become bigger and will be tedious to maintain in the future.
2) The best practice is separating all routes in one file/module, based on Separation of concern principle.
3) We will use Express Router to create the routes.

Try It: Using Routers (1) - api.js

```
var express = require('express');
var router = express.Router();

router.get('/', function(req, res){
    res.send('GET route on things.');
});
router.post('/', function(req, res){
    res.send('POST route on things.');
});
//export this router to use in our index.js
module.exports = router;
```

Try It: Using Routers (2) - server.js

```
var express = require('express');
var app = express();

var routes = require('./api.js');
//both index.js and things.js should be in same
directory
app.use('/api, api);

app.listen(3000);
```

Sending Parameters

In reality, you can send parameters to the API.

Eg: randomuser.me can take **gender** and **nationalities** as parameter.

Once the parameters are retrieved, we can return dynamic content to user.

We will create the dynamic parameters when defining the route, and retrieving it back from the **params** property from the response.

Try It: Route with Parameters (1 param)

```
var express = require('express');
var app = express();

app.get('/:id', function(req, res){
    res.send('The id you specified is ' +
req.params.id);
});

app.listen(3000);
```

Try It: Route with Parameters (More than one)

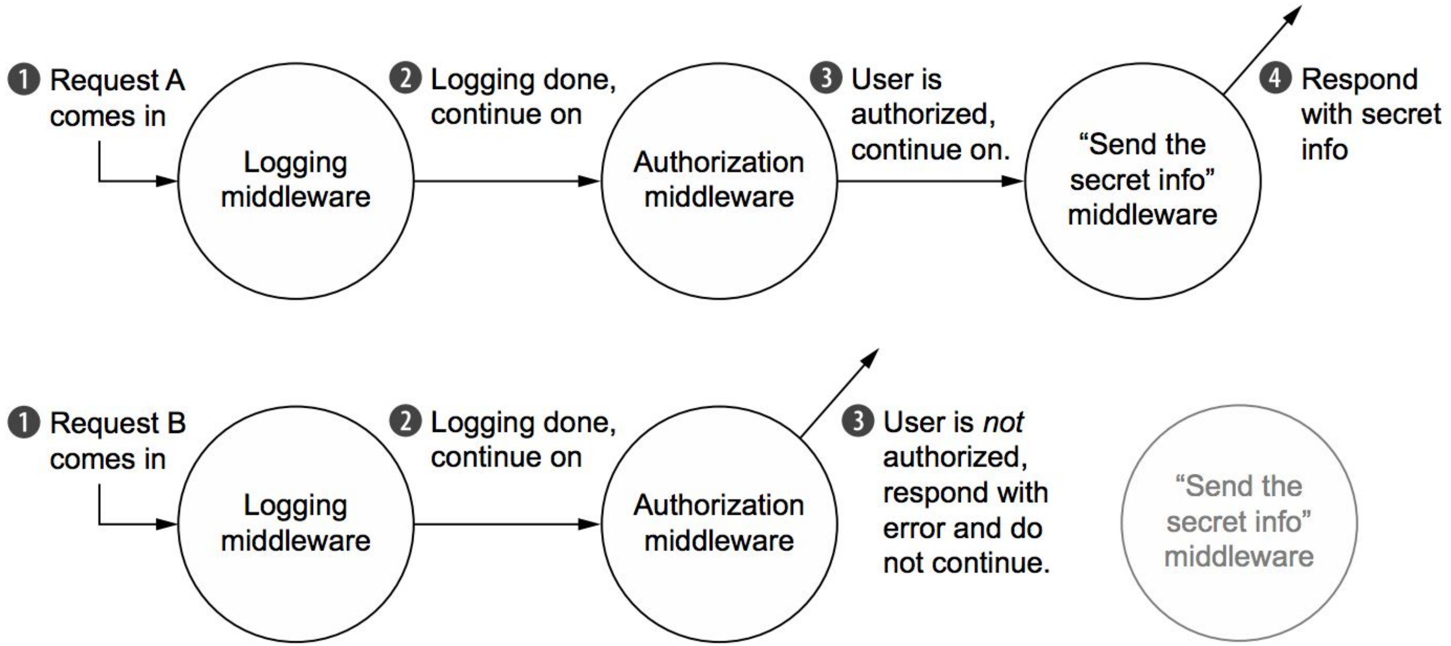
```
var express = require('express');
var app = express();

app.get('/things/:name/:id', function(req, res){
    res.send('id: ' + req.params.id + ' and name: ' +
req.params.name);
});

app.listen(3000);
```

Middleware

Middleware functions are functions that have access to the **request object (req)**, the **response object (res)**, and the next middleware function in the application’s request-response cycle. These functions are used to modify req and res objects for tasks like parsing request bodies, adding response headers, etc.

The middleware will be called for every request on the server, you can use it for example to log a server request, for example

Middleware flow


❶ Request A comes in
Logging middleware
❷ Logging done, continue on
Authorization middleware
❸ User is authorized, continue on.
"Send the secret info" middleware
❹ Respond with secret info
❶ Request B comes in
Logging middleware
❷ Logging done, continue on
Authorization middleware
❸ User is *not* authorized, respond with error and do not continue.
"Send the secret info" middleware

Try It : Middleware

```
var express = require('express');
var app = express();

//Simple request time logger
app.use(function(req, res, next){
      console.log("A new request received at " + Date.now());
      //This function call is very important. It tells that more processing is
      //required for the current request and is in the next middleware function/route
handler.
      next();
});

app.listen(3000);
```

Middleware order of execution

In Express, the order where the middleware is defined is important. It will be executed as how we define it.

Execute the example from next page and observe the log that appears inside console.

Try It : Middleware order of execution

```
var express = require('express');
var app = express();

//First middleware before response is sent
app.use(function(req, res, next){
      console.log("Start");
      next();
});
//Route handler
app.get('/', function(req, res, next){
      res.send("Middle");
      next();
});

app.use('/', function(req, res){
      console.log('End');
});

app.listen(3000);
```

Important Middlewares

These are some of important middlewares that are being used in most of API projects:

Body-parser - To parse the body of requests which have payloads attached to them.

Multer - Used to parse form/data for file upload.

Cookie Parser- To parse *Cookie* header and populate req.cookies with an object keyed by cookie names.

RESTful API

An API is always needed to create mobile applications, single page applications, use AJAX calls and provide data to clients.

An popular architectural style of how to structure and name these APIs and the endpoints is called **REST(Representational Transfer State)**.

RESTful URIs and methods provide us with almost all information we need to process a request.

API Routing - Example

Method	URI	Function
GET	/movies	Gets the list of all movies and their details
GET	/movies/1234	Gets the details of Movie id 1234
POST	/movies	Creates a new movie with the details provided. Response contains the URI for this newly created resource.

API Routing - Example 2

PUT	/movies/1234	Modifies movie id 1234(creates one if it doesn't already exist). Response contains the URI for this newly created resource.
DELETE	/movies/1234	Movie id 1234 should be deleted, if it exists. Response should contain the status of the request.

mongoDB®

Introduction to MongoDB

MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Database :

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Introduction to MongoDB (2)

Collection:

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

Document:

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

Example of document

```
{
    _id: ObjectId(7df78ad8902c)
    title: 'MongoDB Overview',
    description: 'MongoDB is no sql database',
    by: 'tutorials point',
    url: 'http://www.asiadev.academy',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 100,
    comments: [
        {
            user:'user1',
            message: 'My first comment',
            dateCreated: new Date(2011,1,20,2,15),
            like: 0
        },
        {
            user:'user2',
            message: 'My second comments',
            dateCreated: new Date(2017,5,25,7,45),
            like: 5
        }
    ]
}
```

Advantage of MongoDB

- **Schema less** − MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
- Structure of a single object is clear.
- No complex joins.
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- Tuning.
- **Ease of scale-out** − MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.
- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

Why use MongoDB

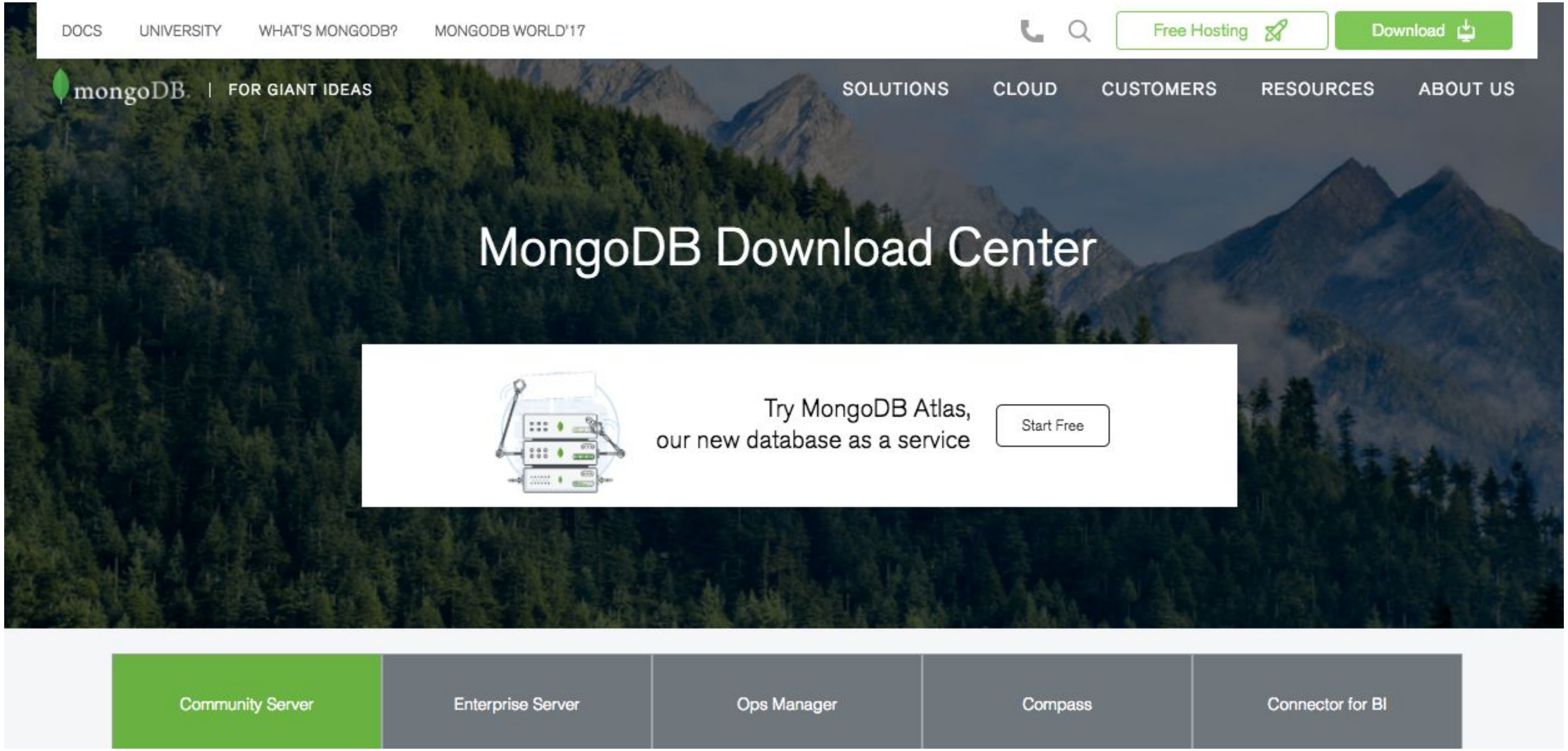
- **Document Oriented Storage** – Data is stored in the form of JSON style documents.
- Index on any attribute
- Replication and high availability
- Auto-sharding
- Rich queries
- Fast in-place updates
- Professional support by MongoDB

Where to use MongoDB

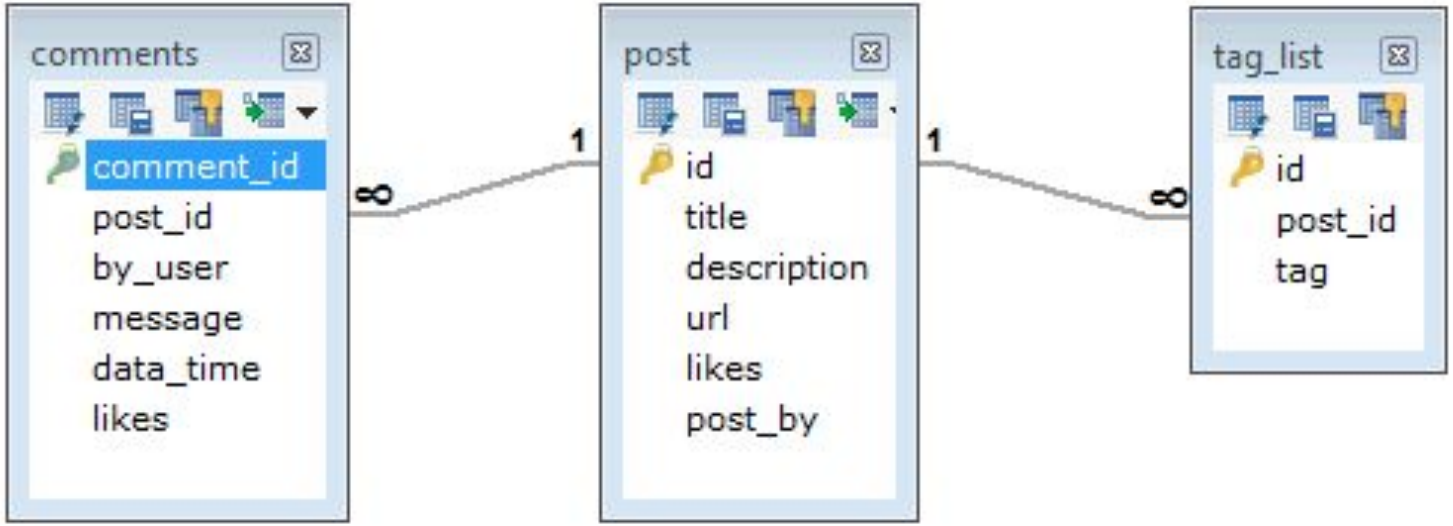
- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

https://www.mongodb.com/

Installing MongoDB locally



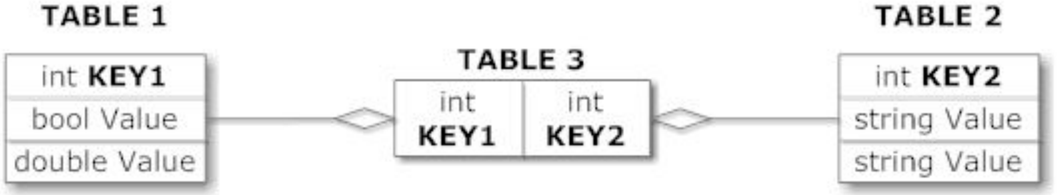
RDBMS


comments
comment_id
post_id
by_user
message
data_time
likes
post
id
title
description
url
likes
post_by
tag_list
id
post_id
tag
1
∞
1
∞

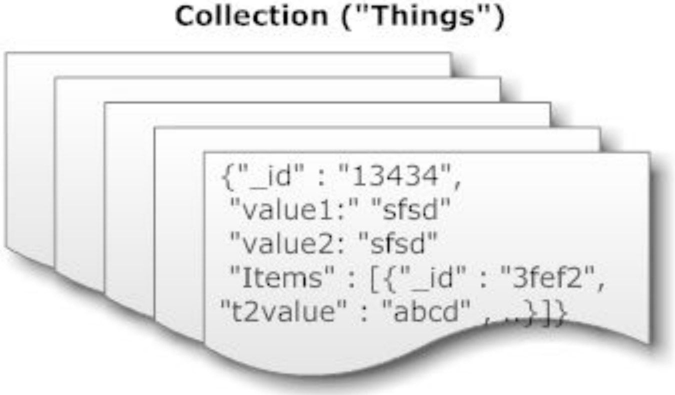
MongoDB

```
{
   _id: POST_ID
   title: TITLE_OF_POST,
   description: POST_DESCRIPTION,
   by: POST_BY,
   url: URL_OF_POST,
   tags: [TAG1, TAG2, TAG3],
   likes: TOTAL_LIKES,
   comments: [
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      },
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      }
   ]
}
```

Relational Model



Document Model



```
{"_id" : "13434",
 "value1:" "sfsd"
 "value2: "sfsd"
 "Items" : [{"_id" : "3fef2",
"t2value" : "abcd" , ..}]}
```

Creating Database

use command to create a database. Example:

```
use mydb
```

db command to check what is the current database.

```
db
```

show dbs command to check the database list

```
show dbs
```

Drop Database

You will use db.dropdatabase() to **drop/delete** the current database.

```
db.dropDatabase()
```

Try It : Play with database

1) Connect to mongodb.
2) Create a new database call it mydb.
3) Insert one of the item inside the db:

```
db.joke.insert({"name":"Knock knock, who’s there?"})
```

4) List down all available dbs.
5) Delete the db.

MongoDB Collections

- A MongoDB collection is a list of MongoDB documents and is the equivalent of a relational database table.
- A collection is created when the first document is being inserted.
- Unlike a table, a collection doesn't enforce any type of schema and can host different structured documents.
- You may use **`show collections`** to see the available collections at the moment.

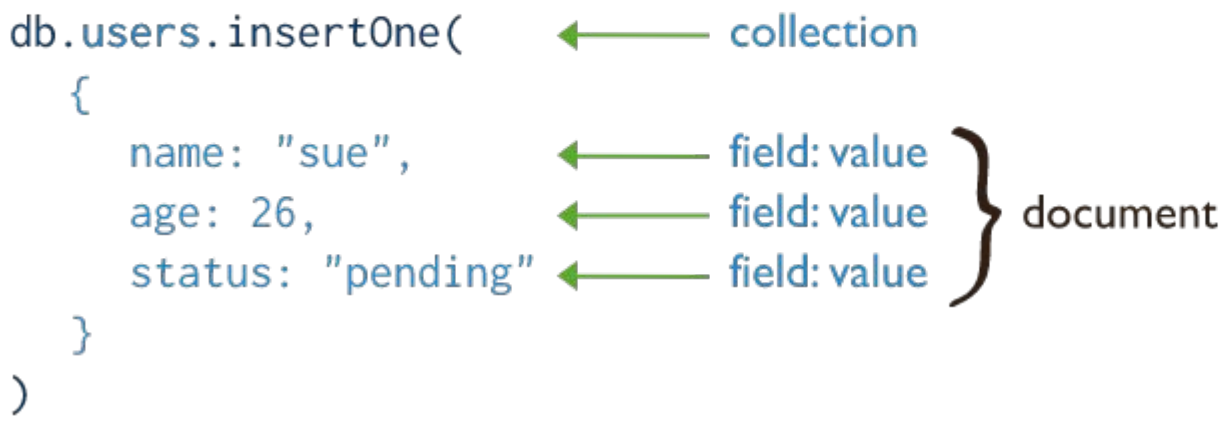
Inserting data in MongoDB Collection

To insert data into MongoDB collection, we will use MongoDB's **insert()** or **save()** method.

```
db.COLLECTION_NAME.insert(document)
```

You may also add more than one data at the time, by sending it as an **array** separated by comma.

Inserting Data in MongoDB Collection



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

Query Document

We use **find()** method to query data from MongoDB collection.

```
db.COLLECTION_NAME.find()
```

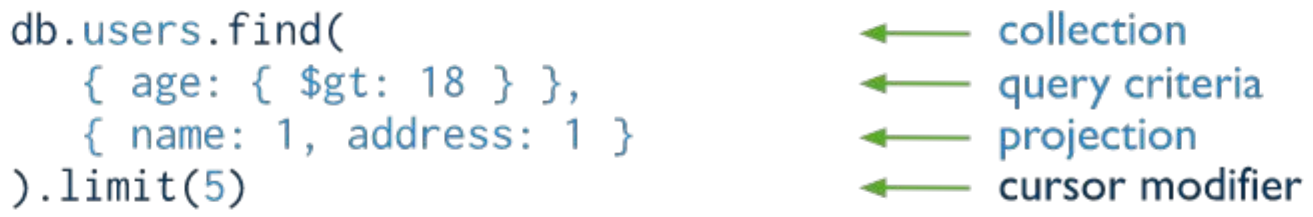
You may use **pretty()** method to format the JSON nicely.

```
db.COLLECTION_NAME.find().pretty()
```

You may also use findOne() method if you only need to return 1 result.

```
db.COLLECTION_NAME.findOne()
```

Query



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

Filtering results.

Operation	**Syntax**	**Example**
Equality	{“<key>”:”<value>”}	db.places.find({"country":"France"})
Less Than	{<key>:{$lt:<value>}}	db.places.find({"likes":{$lt:120} })
Less than equals	{<key>:{$lte:<value>}}	db.places.find({"likes":{$lte:120} })
Greater than	{<key>:{$gt:<value>}}	db.places.find({"likes":{$gt:100} })
Greater than equal	{<key>:{$gte:<value>}}	db.places.find({"likes":{$gte:100} })
Not Equal	{<key>:{$ne:<value>}}	db.places.find({"likes":{$ne:100} })

AND and OR query in MongoDB

In **find()** method, if you pass multiple keys by separating them by ',' then MongoDB treats it as **AND** condition. Eg:

```
db.places.find({"country":"France","likes":{$gt:100}})
```

To query documents based on the OR condition, you need to use **$or** keyword, followed by an array of condition. Following is the basic syntax of **OR** –

```
db.places.find({$or:[{"country":"France","likes":{$gte:100}}]})
```

Limiting records

You use **limit()** method to limit the number of documents to be displayed:

```
db.COLLECTION_NAME.find().limit(NUMBER)
```

Eg:db.places.find().limit(2)

You use **skip()** method to skip the number of documents to be displayed:

Eg: db.places.find({},{"name":1,_id:0}).limit(1).skip(1)

Sorting records

We use **sort()** method to specify the documents in a specific sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

```
db.COLLECTION_NAME.find().sort({KEY:1})
```

Eg: db.places.find().sort({"name":1})

Updating a document

We use update() or save() methods to update a document into collection. You will use:

Update - Update the current document with the updated data.

```
db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA)
 db.places.update({'name':'Lille'},{$set:{'likes':200}})
```

Save - Replace the current document with new data.

```
 db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})
```

Deleting a document

We use **remove()** method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.

Eg:

db.places.remove({'name':'Lille'})

db.places.remove() -> Remove all

Projection

Projection means finding the necessary data rather than selecting whole data of document. For example, if a data has 5 fields and you only need to show 3, then you may only set to show 3 of them.

```
db.COLLECTION_NAME.find({},{KEY:1})
```

Eg:

```
db.mycol.find({},{"title":1,_id:0})
```

Indexing

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. We created index to speed up the query process .

```
db.COLLECTION_NAME.ensureIndex({KEY:1})
```

Eg: db.places.ensureIndex({name:1})

Aggregation

Aggregations operations process data records and return computed results.

Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.

```
db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
Eg:
db.places.aggregate([{$group : {_id: "$country", num_cities :
{$sum : 1}}}] )
```

Aggregation

Expression	Description	Example
$sum	Sums up the defined value from all documents in the collection.	db.places.aggregate([{$group : {_id: "$country", average_like : {$sum : "$likes"}}}])
$avg	Calculates the average of all given values from all documents in the collection.	db.places.aggregate([{$group : {_id: "$country", average_like : {$avg : "$likes"}}}])
$min	Gets the minimum of the corresponding values from all documents in the collection.	db.places.aggregate([{$group : {_id: "$country", average_like : {$min : "$likes"}}}])
$max	Gets the maximum of the corresponding values from all documents in the collection.	db.places.aggregate([{$group : {_id: "$country", average_like : {$max : "$likes"}}}])