# Web Application Development

COMP4347
COMP5347

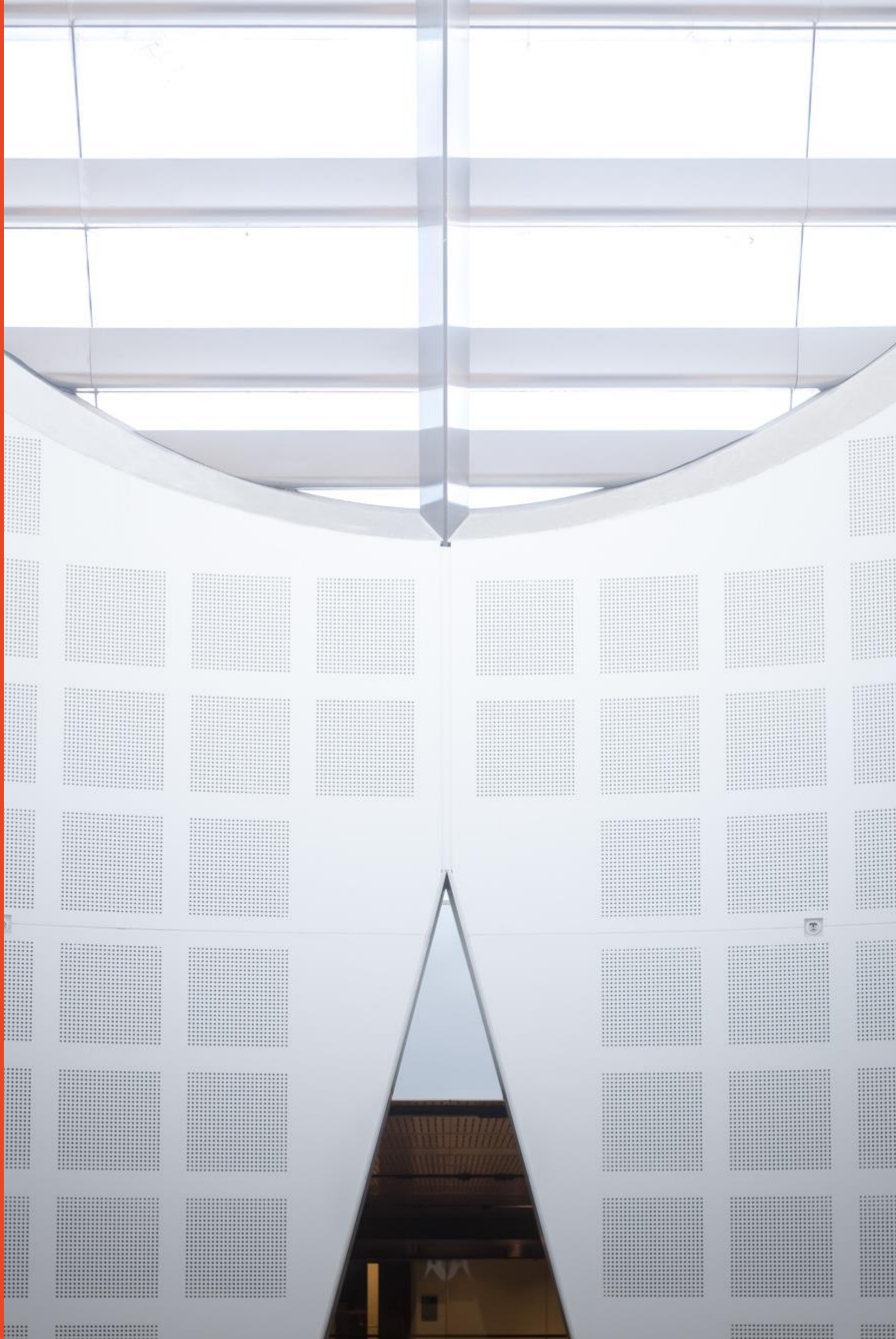
Node.js
MVC Architecture
Introduction to MongoDB

Week 6
Semester 1, 2025

Dr. Mehdi Nobakht

School of Computer Science

THE UNIVERSITY OF SYDNEY



## Outline

- Implementing MVC basics
  - Application Folder Structure
  - CommonJS modules
  - Controller and Routers
- Session Management
- Database Layer
  - Introduction to NoSQL
  - Introduction to MongoDB

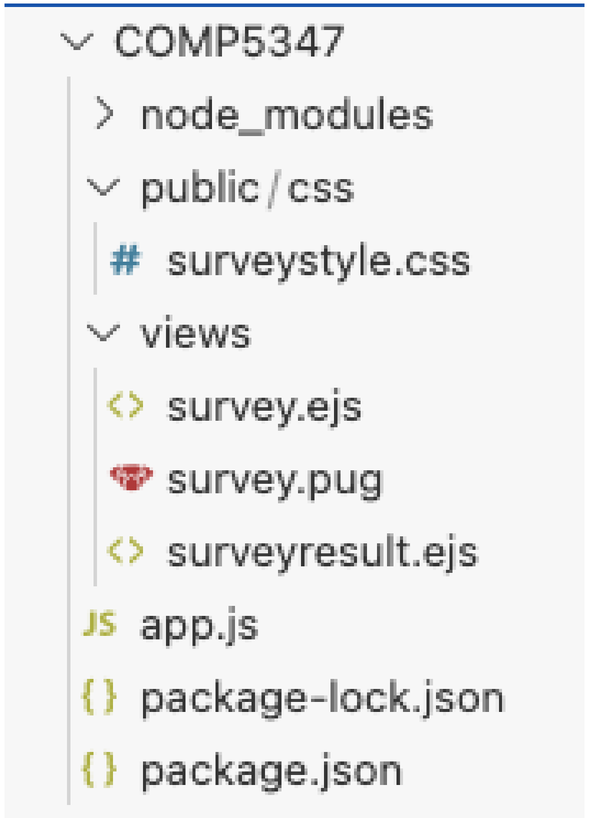
# The Survey app (Week 5 lab)

```
var express = require('express')
var path = require('path’)


var app = express()
var products=['iphone 7', 'huawei p9', 'Pixel XL', 'Samsung S7']
var surveyresults = { fp:[0,0,0,0],mp:[0,0,0,0]}

app.use(express.json());
app.use(express.urlencoded({extended: true}));
app.use(express.static(path.join(__dirname, 'public')));
app.set('views', path.join(__dirname,'views'));

app.get('/', function(req,res){
    res.render('survey.pug',{products:products})
});

app.post('/survey', function(req,res){
    console.log(req.body);
    cast vote = req.body;
    …
    res.render('surveyresult.ejs', {products: products, surveyresults: surveyresults})
});
app.listen(3000, function () {
  console.log('survey app listening on port 3000!')
})
```

Application scope variables

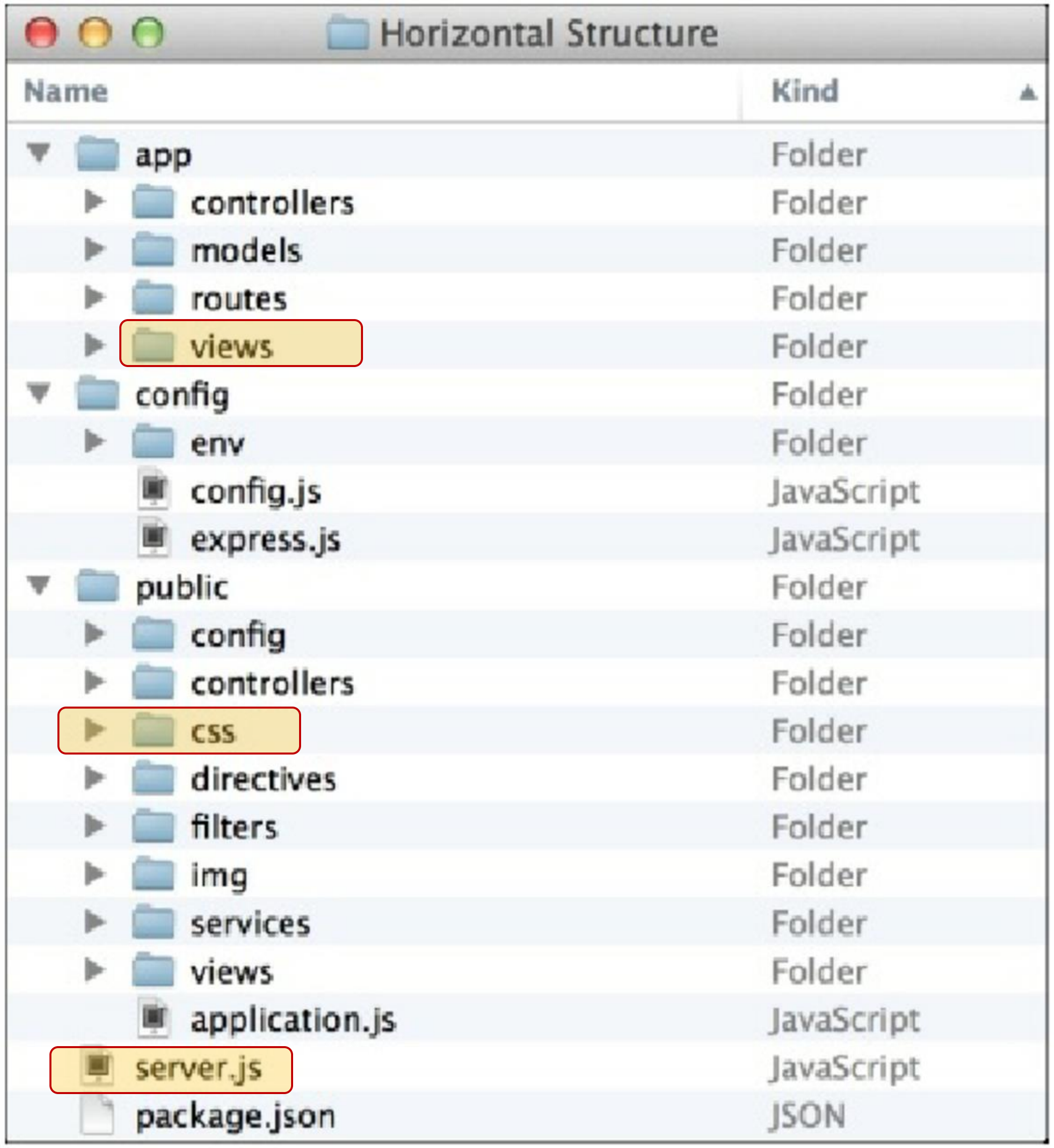
Request scope variables

## Several Issues


COMP5347
node_modules
public / css
surveystyle.css
views
survey.ejs
survey.pug
surveyresult.ejs
app.js
package-lock.json
package.json

- There is a *single* JS file with all settings and route methods
- Large application will have many route methods and some are related
  - Modular controller and route mappings
- Data sharing
  - Application scope variable
  - Request scope variable
  - Session scope variable
- We don't have separated model yet!

## Application folder structure



Horizontal Structure

| Name | Kind |
|---|---|
| app | Folder |
| controllers | Folder |
| models | Folder |
| routes | Folder |
| views | Folder |
| config | Folder |
| env | Folder |
| config.js | JavaScript |
| express.js | JavaScript |
| public | Folder |
| config | Folder |
| controllers | Folder |
| css | Folder |
| directives | Folder |
| filters | Folder |
| img | Folder |
| services | Folder |
| views | Folder |
| application.js | JavaScript |
| server.js | JavaScript |
| package.json | JSON |

## CommonJS Module in Node.js

- A file-based module system to solve JavaScript single global namespace issue
  - Each file is its own module
- Three key components:
  - **requires():** this method is used to load the module into the current code
    - Eg: require(express)
  - **exports**: this object is contained in each module and allows you to expose piece of your code when the module is loaded.
  - **module**: refers to the current module definition (metadata).

## Writing our own module

hello.js

```
var message = 'Hello';

module.exports.sayHello=function(){
      console.log(message);
}
exports.sayBye=function(){
      console.log("Bye")
}
```

module.exports and exports are equivalent, both referring to the object exposed by the module

We can expose many methods by defining them as properties of the **module.exports** object.

Hello_client.js

```
var hello = require('./hello');
hello.sayHello()
hello.sayBye()
```

Calling **require(...)** in the client code would return the **modules.exports** object. Our hello which has exposed two methods

## Writing Controller(s)

survey.server.controller.js

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

module.exports.showForm = function(req,res){
    products = req.app.locals.products
    res.render('survey.pug',{products:products})
}

module.exports.showResult = function(req,res){
    console.log(req.body);
    gender = req.body.gender
    productidx = req.body.vote;
    products = req.app.locals.products;
    surveyresults = req.app.locals.surveyresults;
    if (gender == 0)
        surveyresults.mp[productidx]++;
    else
        surveyresults.fp[productidx]++;

    res.render('surveyresult.ejs ', {products: products,
          surveyresults: surveyresults})
}
```

This controller module exposes two methods: **showForm** is used for displaying the form; **showReulsult** is used for showing the results

The methods are not mapped to URL yet

**req.app.locals** is used to share application scope variables

Each request object has a reference to the current running express application: **req.app**

**app.locals** is used to store properties that are local variables within the application (application scope data)

## Mapping Controller to URL

survey.server.routes.js

```
var express = require('express')
var controller = require('../controllers/survey.server.controller')
var router = express.Router()

router.get('/', controller.showForm)
router.post('/survey', controller.showResult)
module.exports = router
```

- **Express Router** middleware used to group  the route handlers for particular part of a web application (e.g., user account functions)
- Router can be used to define the routes (HTTP request method, URL path/pattern, and callback function to handle that pattern)
- Using Router is similar to defining routes directly on the Express application object
  - Router supports application modularity

# Mapping Controller to URL

survey.server.routes.js

```
var express = require('express')
var controller = require('../controllers/survey.server.controller')
var router = express.Router()

router.get('/', controller.showForm)
router.post('/survey', controller.showResult)
module.exports = router
```

server.js

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

var survey = require('../routes/survey.server.routes')

var app = express()
app.locals.products=['iphone 7', 'huawei p9', 'Pixel XL', 'Samsung S7']
app.locals.surveyresults = {
    fp:[0,0,0,0], mp:[0,0,0,0]
}

app.use(express.json());   % to parse json data from the request
app.use(express.urlencoded({ extended: true}));
app.set('views', path.join(__dirname,'/app/views'));
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', survey)
app.listen(3000, function () {
  console.log('survey app listening on port 3000!')
})
```

# Mapping Controller to URL

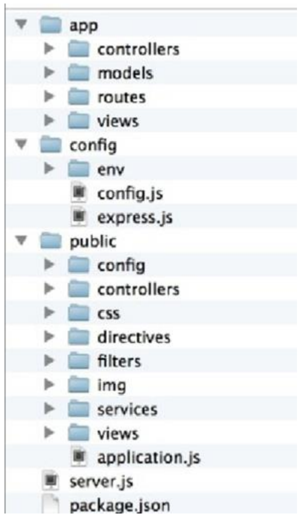
survey.server.routes.js

```
var express = require('express')
var controller = require('../controllers/survey.server.controller')
var router = express.Router()

router.get('/', controller.showForm)
router.post('/survey', controller.showResult)
module.exports = router
```

server.js

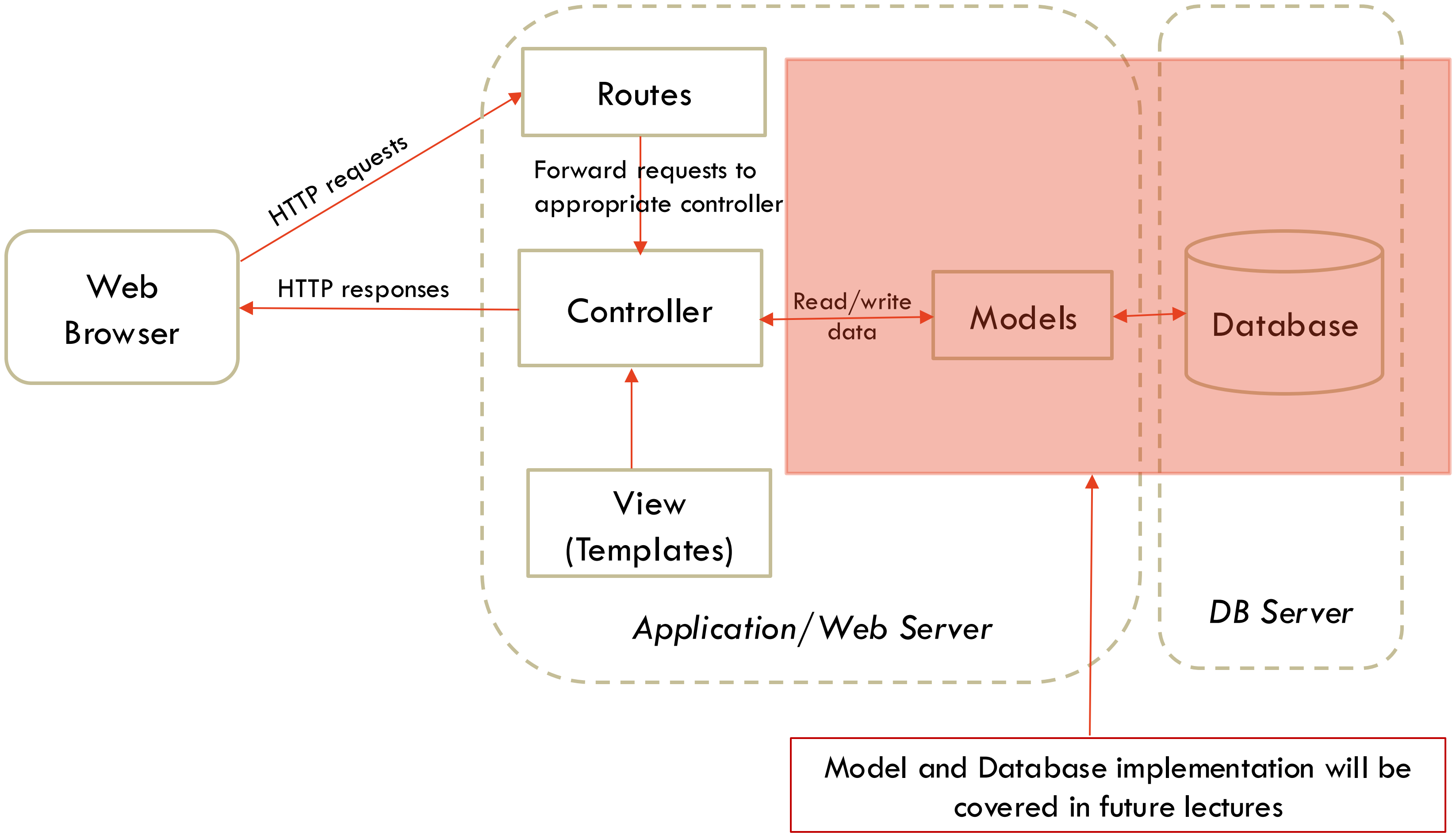
```
var express = require('express');
var path = require('path’)


var survey = require('../routes/survey.server.routes')

var app = express()
app.locals.products=['iphone 7', 'huawei p9', 'Pixel XL', 'Samsung S7']
app.locals.surveyresults = {
    fp:[0,0,0,0], mp:[0,0,0,0]
}

app.use(express.json());   % to parse json data from the request
app.use(express.urlencoded({ extended: true}));
app.set('views', path.join(__dirname,‘/app/views'));
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', survey)
app.listen(3000, function () {
  console.log('survey app listening on port 3000!')
})
```

Set the two application scope variables



- Get request send to /survey will display the form
- Post request send to /survey/survey will show the result

# Survey Web Application – MVC



Model and Database implementation will be covered in future lectures

## Outline

- Implementing MVC basics
  - Application Folder Structure
  - CommonsJS modules
  - Controller and Routers
- Session Management
- Database Layer
  - Introduction to NoSQL
  - Introduction to MongoDB

## Variable Scopes

- *Application scope* data is available through out the application
- *Request scope* data is available just to components handling the current request (controller, view, model)

- **Session scope** data is available across multiple related requests
  - Login to a web mail server
  - e-commerce website with default session until inactivity
  - Session management - implemented as middleware function in module **express-session**

## Session

- Session is a mechanism to associate a series of requests coming from a client
  - A conversational state between the client and the server

- HTTP is stateless
  - By default, each request is a session!
  - To maintain a conversational state
    - A <u>server</u> needs to remember what has been going on for EACH client
    - A <u>client</u> needs to send some data to identify self
    - Also a mechanism to control the start/end of a session

## How does session work in general

- Client sends the first request
- The server creates an ID for the client and a session object to store session data
  - A server can maintain many sessions simultaneously (identified by ID)
- The server executes predefined business logic for that request and sends back the response together with the ID to the client
- The client stores the ID and associates it with the server
  - A client may maintain many sessions with different servers simultaneously, hence it is important to remember which ID belongs to which server
- When the client sends a second request to this server, it attaches the ID with the request
- The server extracts the ID and use it to find the session data associated with this particular client and make it available to the current request

## On the server side

- How can server remembers client state?
  - An object to hold conversational state across multiple requests from the same client identified by a key or ID.
  - It stays for an entire session with a specific client
  - We can use it to store everything about a particular client.
  - Stay in memory mostly but can be persisted in a database

## Where does clients stores the ID?

- A cookie is a small piece of information stored on a client's computer by the **browser**
- Each browser has its own way to store cookies either in a **text file** or in a **lightweight database**
- Each browser manages its own cookies.
- Since a browser stores cookies from various websites, it also needs a way to identify cookie for a particular site.
- Cookies are identified by {name, domain, path}
- The following are two cookies from different domains

cookie 1
name = **connect.id**
value = s%3AKTObttJqW0k6aVrHB
domain = **localhost**
path = ***/***

cookie 2
name = **name**
value = Joe
domain = **web.it.usyd.edu.au**
path = ***/~comp5347/doc/***

## Associate web sites/pages and Cookies

– Browser would associate/send all cookies in the URL scope:
  - cookie-domain is domain-suffix of URL-domain, and
  - cookie-path is prefix of URL-path
– An example
  - Page http://web.cs.usyd.edu.au/~comp5347/doc/cookie.html will have cookie B and C, cookie A is not associated with this page

| cookie A | cookie B | cookie C |
|---|---|---|
| name = **name** | name = **name** | name = _utma |
| value = Paul | value = Joe | value = 223117855… |
| domain = web.it.usyd.edu.au | domain = web.cs.usyd.edu.au | domain = usyd.edu.au |
| path = **/~info5010/comp5347/** | path = /~comp5347/doc/ | path = **/** |

## express-session

- Express application’s session management is implemented as middleware function in module **express-session**
- Module **express-session** uses cookie based session management where a small cookie is created to store session id.

## Session-aware survey

- Requirements: to ensure that a user cannot vote more than once in a certain period of time 'using the same browser'.

- Simple solution:
  - Add a session scope variable **vote** to store a user's previous vote
  - Submit bottom → check if variable **vote** exists, if true, discard the current vote; else, set the variable to the current vote in session scope and update the results
  - Session scope (object) is accessible to all request as: **req.session**

## Session-aware survey

surveysession.server.controller.js

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

module.exports.showForm=function(req,res){
    products = req.app.locals.products
    res.render('surveysession.pug',{products:products})
}

module.exports.showResult=function(req,res){
    gender = req.body.gender
    productidx = req.body.vote;
    products = req.app.locals.products;
    surveyresults = req.app.locals.surveyresults;
    sess=req.session;
    if ("vote" in sess)
        res.render('surveysessionresult.pug', {products: products, surveyresults: 
surveyresults})
    else{
        sess.vote = productidx;
        gender = req.body.gender
        productidx = req.body.vote;
        if (gender == 0)
            surveyresults.mp[productidx]++;
        else
            surveyresults.fp[productidx]++;
        res.render('surveysessionresult.pug', {products: products, surveyresults: 
surveyresults})
    }
}
```

# Routes and server.js

surveysession.server.routes.js

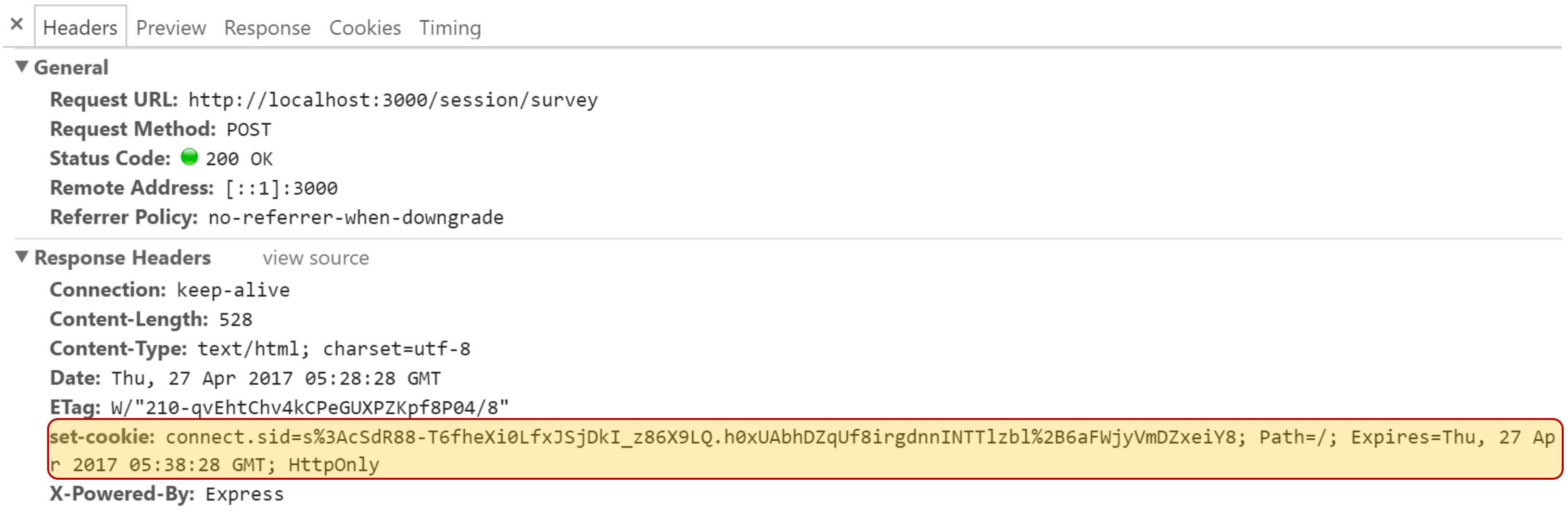
```
var express = require('express')
var router = express.Router()
var controller = require('../controllers/surveysession.server.controller')

router.get('/', controller.showForm)
router.post('/survey', controller.showResult)
module.exports = router
```

server.js

```
var express = require('express');
var path = require('path')
var bodyParser = require('body-parser');
var session = require('express-session');

var surveysession = require('./routes/surveysession.server.routes')

var app = express()
app.locals.products=['iphone 7', 'huawei p9', 'Pixel XL', 'Samsung S7']
app.locals.surveyresults = {fp:[0,0,0,0],mp:[0,0,0,0]}

app.set('views', path.join(__dirname,'views'));
app.use(express.static(path.join(__dirname, 'public')));
app.use(bodyParser.json())
app.use(bodyParser.urlencoded())
app.use(session({secret: 'ssshhhhh',cookie:{maxAge:600000}}));
app.use('/session',surveysession)
app.listen(3000, function () {
  console.log('survey app listening on port 3000!')
})
```

The session will expire in 10 minutes

# Cookies sent by server



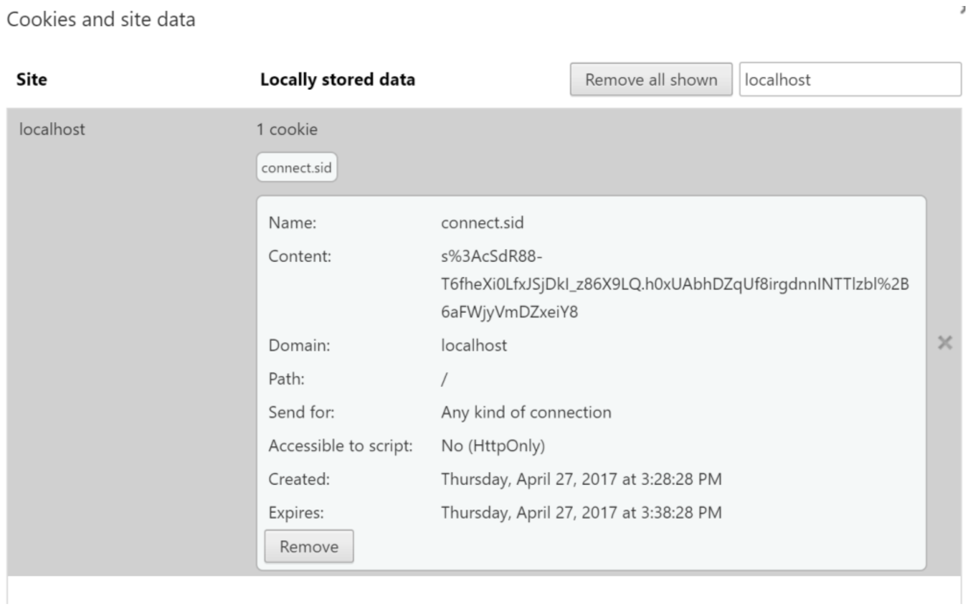
× Headers Preview Response Cookies Timing

▼ General

```
Request URL: http://localhost:3000/session/survey
Request Method: POST
Status Code: ● 200 OK
Remote Address: [::1]:3000
Referrer Policy: no-referrer-when-downgrade
```

▼ Response Headers view source

```
Connection: keep-alive
Content-Length: 528
Content-Type: text/html; charset=utf-8
Date: Thu, 27 Apr 2017 05:28:28 GMT
ETag: W/"210-qvEhtChv4kCPeGUXPZKpf8P04/8"
set-cookie: connect.sid=s%3AcSdR88-T6fheXi0LfxJSjDkI_z86X9LQ.h0xUAbhDZqUf8irgdnnINTTlzbl%2B6aFWjyVmDZxeiY8; Path=/; Expires=Thu, 27 Ap
r 2017 05:38:28 GMT; HttpOnly
X-Powered-By: Express
```



Cookies and site data

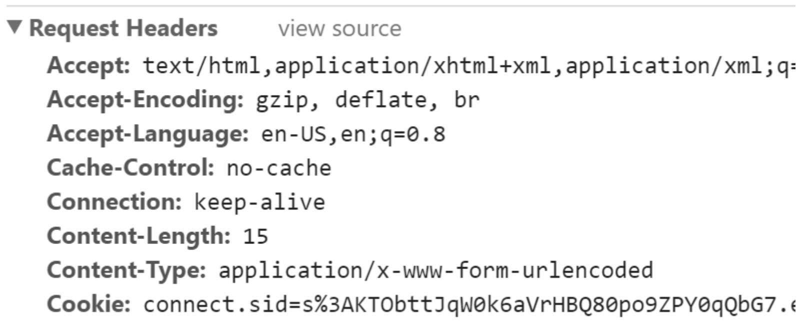
| Site | Locally stored data |
|---|---|
| localhost | 1 cookie |

Remove all shown | localhost

connect.sid

| | |
|---|---|
| Name: | connect.sid |
| Content: | s%3AcSdR88-T6fheXi0LfxJSjDkI_z86X9LQ.h0xUAbhDZqUf8irgdnnINTTlzbl%2B6aFWjyVmDZxeiY8 |
| Domain: | localhost |
| Path: | / |
| Send for: | Any kind of connection |
| Accessible to script: | No (HttpOnly) |
| Created: | Thursday, April 27, 2017 at 3:28:28 PM |
| Expires: | Thursday, April 27, 2017 at 3:38:28 PM |

Remove



▼ Request Headers view source

```
Accept: text/html,application/xhtml+xml,application/xml;q=
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.8
Cache-Control: no-cache
Connection: keep-alive
Content-Length: 15
Content-Type: application/x-www-form-urlencoded
Cookie: connect.sid=s%3AKTObttJqW0k6aVrHBQ80po9ZPY0qQbG7.
```

## Outline

- Implementing MVC basics
  - Application Folder Structure
  - CommonsJS modules
  - Controller and Routers
- Session Management
- Database Layer
  - Introduction to NoSQL
  - Introduction to MongoDB

## NoSQL Brief Introduction

- NoSQL encompasses the general trend of a new generation of database servers
  - Challenges of traditional SQL databases
  - Scalability, flexible schema, object relational mismatch, etc

- Broad categories of NoSQL systems
  - Document Storage (e.g. MongoDB)
  - Key-Value Storage
  - Column based Storage
  - Graph database

## Document Storage and MongoDB

- Document storage system based on *self describing* documents
  - Entity stored as a document (record (row) in typical SQL0
  - Two dominant formats
    - XML
    - JSON (JavaScript Object Notation)

```
Invoice _1= {   customer: {name: "John", address: "Sydney"},
           product: { code: "123", quantity: 2}
       }
```

```
Invoice _2= {   customer: {name: "Smith", address: "Melbourne"},
              product: { code: "xyz", quantity: 20},
              delivery: "express"
        }
```

## JSON Data Format

- JSON (**J**ava**S**cript **O**bject **N**otation): represent JavaScript objects as strings

- Introduced in 1999 as an alternative to XML for data exchange

- JSON object: a list of property names and values contained in curly braces:

```
{ propertyName1 : value1, propertyName2 : value2 }
```

- Arrays represented with square brackets

```
[ value1, value2, value3 ]
```

## Matching Terms in SQL and MongoDB

| **SQL** | **MongoDB** |
|---|---|
| Database | Database |
| Table | Collection |
| Index | Index |
| Row | BSON* document |
| Column | BSON field |
| Primary key | _id field |
| Join | Embedding and referencing<br>$lookup in aggregation (since 3.2) |

*Binary JSON

## MongoDB Document Model – Example

| TFN | Name | Email | age |
|---|---|---|---|
| 12345 | Joe Smith | joe@gmail.com | 30 |
| 54321 | Mary Sharp | mary@gmail.com | 27 |

```
{ _id: 12345,
  name: “Joe Smith”,
  email: “joe@gmail.com”,
  age: 30
}
```

```
{ _id: 54321,
  name: “Mary Sharp”,
  email: “mary@gmail.com”,
  age: 27
}
```

## MongoDB Document Model - Example

`users` table in RDBMS

Column name is part of schema

| TFN | Name | Email | age |
|---|---|---|---|
| 12345 | Joe Smith | joe@gmail.com | 30 |
| 54321 | Mary Sharp | mary@gmail.com | 27 |

two rows

```
{ _id: 12345,
  name: "Joe Smith",
  email: "joe@gmail.com",
  age: 30
}
{ _id: 54321,
  name: "Mary Sharp",
  email: "mary@gmail.com",
  age: 27
}
```

two documents

Field name is part of data

## Native Support for Array

```
{ _id: 12345,
  name: “Joe Smith”,
  emails: [“joe@gmail.com”, “joe@ibm.com”],
  age: 30
}
{ _id: 54321,
  name: “Mary Sharp”,
  email: “mary@gmail.com”,
  age: 27
}
```

| TFN | Name | Email | age |
|---|---|---|---|
| 12345 | Joe Smith | joe@gmail.com , joe@ibm.com ?? | 30 |
| 54321 | Mary Sharp | mary@gmail.com | 27 |

## Native Support for Embedded Document

```
{ _id: 12345,
  name: “Joe Smith”,
  email: [“joe@gmail.com”, “joe@ibm.com”],
  age: 30
}
```

```
{ _id: 54321,
  name: “Mary Sharp”,
  email: “mary@gmail.com”,
  age: 27,
  address: { number: 1,
             name: “cleveland street”,
             suburb: “chippendale”,
             zip: 2008
           }
}
```

| TFN | Name | Email | age |
|---|---|---|---|
| 12345 | Joe Smith | joe@gmail.com | 30 |
| 54321 | Mary Sharp | mary@gmail.com | 27 |

## Native Support for Embedded Document

```
{ _id: 12345,
  name: “Joe Smith”,
  email: [“joe@gmail.com”, “joe@ibm.com”],
  age: 30
}
```

```
{ _id: 54321,
  name: “Mary Sharp”,
  email: “mary@gmail.com”,
  age: 27,
  address: { number: 1,
             name: “cleveland street”,
             suburb: “chippendale”,
             zip: 2008
           }
}
```

| TFN | Name | Email | age | address |
|---|---|---|---|---|
| 12345 | Joe Smith | joe@gmail.com | 30 | |
| 54321 | Mary Sharp | mary@gmail.com | 27 | 1 cleveland street, chippendale, NSW 2008 |

## MongoDB data types

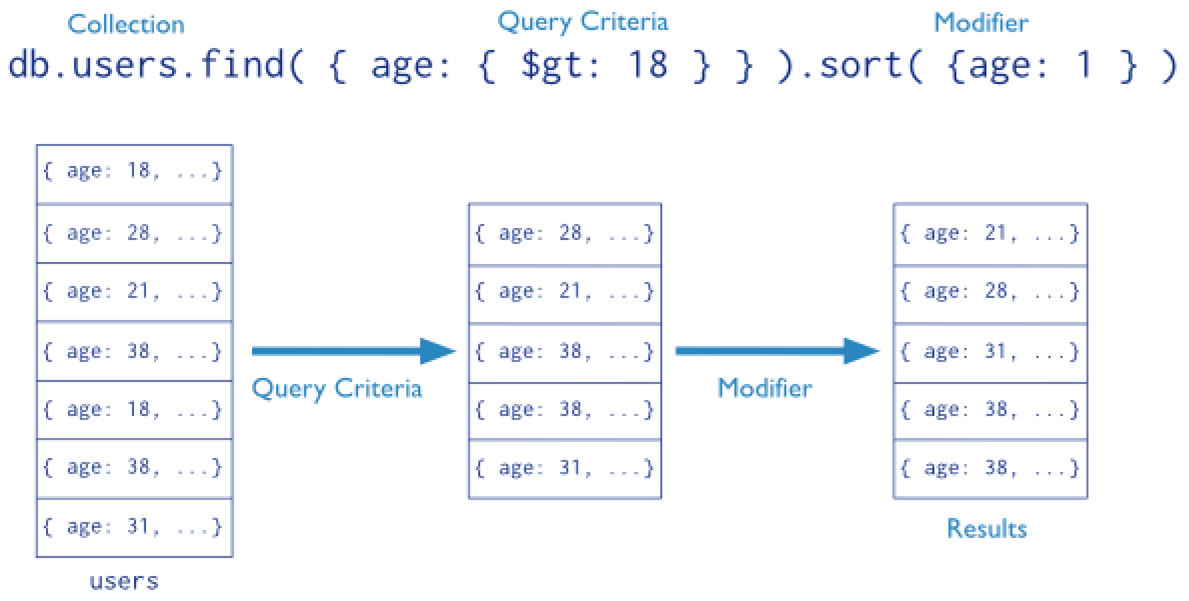
- Primitive types
  - String, integer, boolean (true/false), double, null
- Predefined special types
  - Date, object id, binary data, regular expression, timestamp,
  - DB Drivers implement them in language-specific way
  - The interactive shell provides constructors for all
- Array and object
- Field name is of **string** type with certain restrictions
  - "_id" is reserved for primary key
  - cannot start with "$", cannot contain "." or null

## MongoDB Queries

- A ***read*** query targets a specific collection
  - **criteria**
  - may include a **projection** to specify fields from the matching documents
  - may include **modifier** to limit, skip, or sort the results

- A ***write*** query may *create*, *update* or *delete* data
  - One query modifies the data of a single collection
  - Update and delete query can specify query **criteria**

## Read Query Example



Find documents in the **users** collection with **age** field greater than 18, sort the results in ascending order by **age**

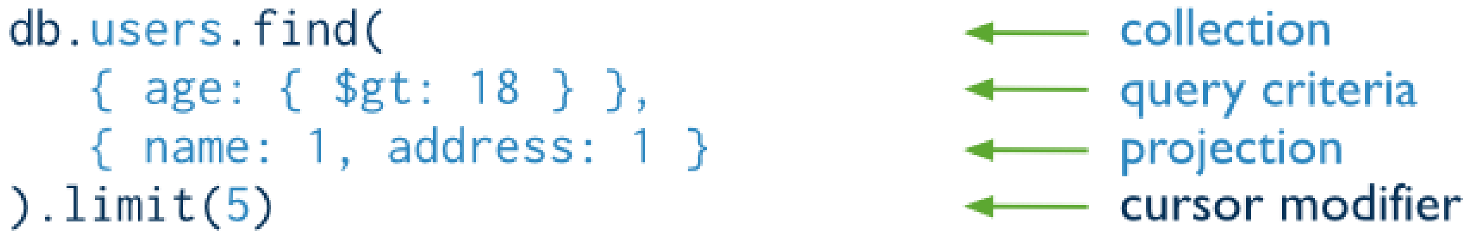
## Read Query Interface

– db.collection.find()

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

## Read Query Interface

– db.collection.find()

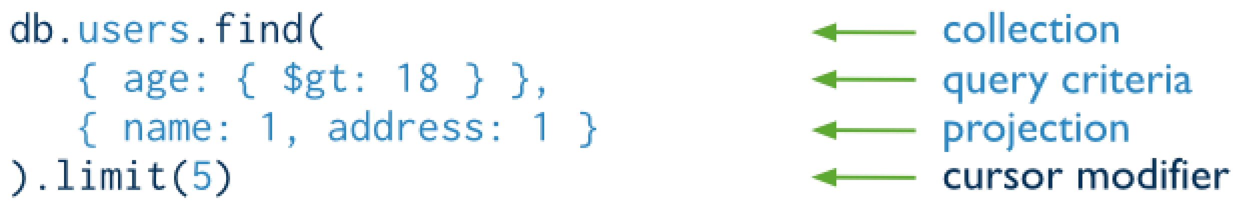


```
db.users.find(
   { age: { $gt: 18 } },
   { name: 1, address: 1 }
).limit(5)
```

Find at most 5 documents in the **users** collection with **age** field greater than 18, return only the name and address field of each document.

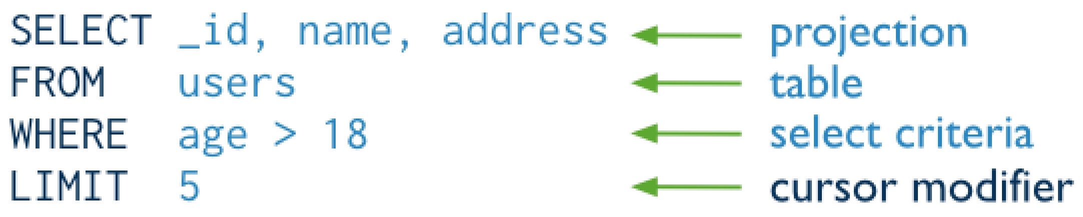
## Read Query Interface

– db.collection.find()



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

Find at most 5 documents in the **users** collection with **age** field greater than 18, return only the name and address field of each document.



```
SELECT _id, name, address
FROM   users
WHERE  age > 18
LIMIT  5
```

## Read Query Features

- Find data using any criteria
  - Does not require indexing
  - Indexing can improve performance
  - JOIN is not supported!!

- Query **criteria** expressed as BSON document (query object)
  - Individual expressed using predefined selection operator
    - Eg. **$lt** is the operator for “less than”

- Query **projection** are expressed as BSON document

## Read Query Features

| SQL | MongoDB Query in Shell |
|---|---|
| select * from user | db.user.find() or db.user.find({}) |
| select name, age from user | db.user.find({},{name:1,age:1,_id:0}) |
| select * from user<br>where name = “Joe Smith” | db.user.find({name: “Joe Smith”}) |
| select * from user<br>where age < 30 | db.user.find({age: {$lt:30}}) |

## Querying Array field

- Like querying simple type field
  - db.user.find({emails: “joe@gmail.com”})
  - db.user.find({“emails.0”: “joe@gmail.com”})

```
{ _id: 12345,
  name: “Joe Smith”,
  emails: [“joe@gmail.com”, “joe@ibm.com”],
  age: 30}
```

```
{ _id: 54321,
  name: “Mary Sharp”,
  email: “mary@gmail.com”,
  age: 27}
```

http://docs.mongodb.org/manual/tutorial/query-documents/#arrays

## Querying Embedded Document

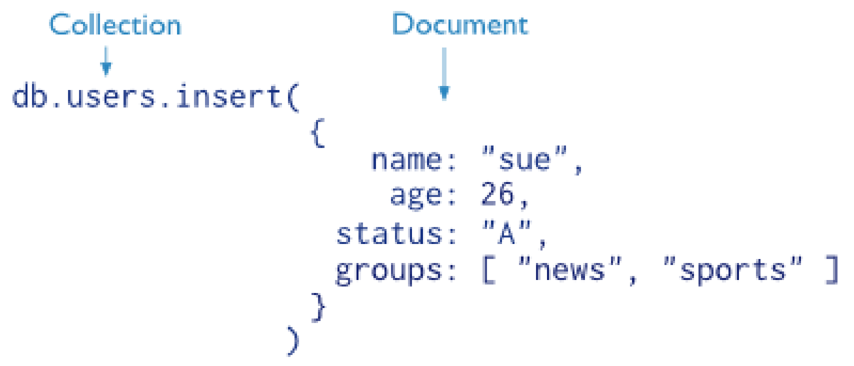
- Queried as a **whole**, by **individual field**, or by **combination of individual fields**
  - db.user.find({**address**: {number: 1, name: "pine street", suburb: "chippendale", zip: 2008}})
  - db.user.find({**"address.suburb"**: "chippendale"})
  - db.user.find({**address**: {**$elemMatch**: {name: "pine street", suburb: "chippendale"}}})

```
{ _id: 12345,
  name: "Joe Smith", email: ["joe@gmail.com", "joe@ibm.com"], age: 30,
  address: {number: 1, name: "pine street", suburb: "chippendale", zip: 2008 }
}
```

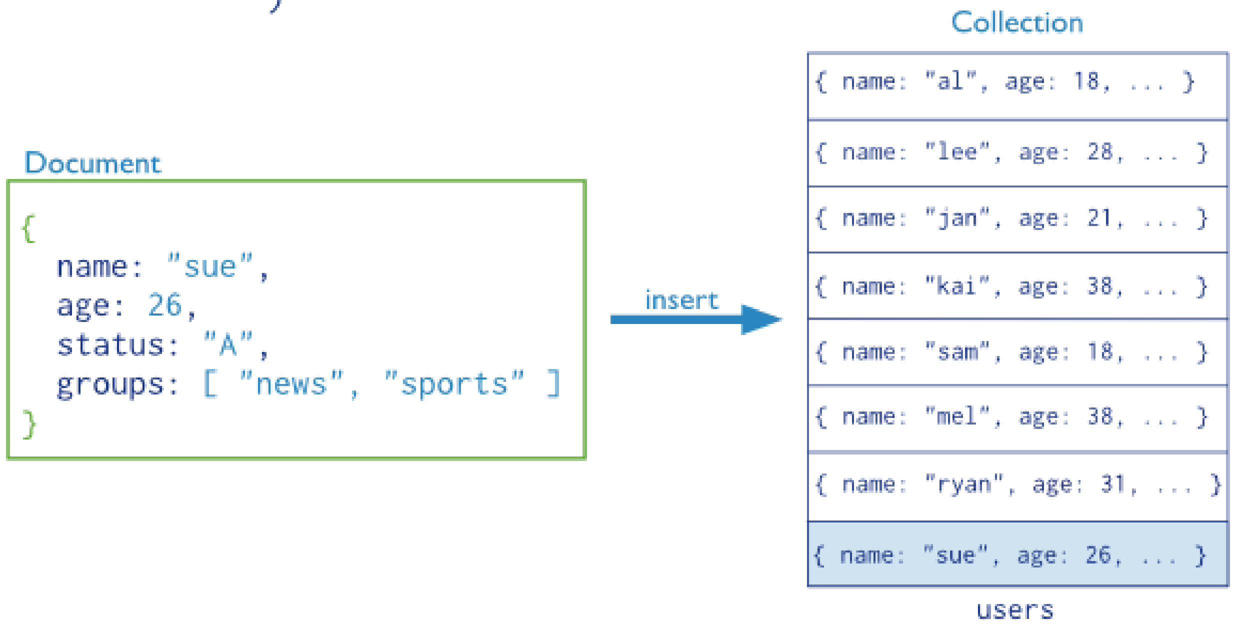
```
{ _id: 54321,
  name: "Mary Sharp", email: "mary@gmail.com",age: 27,
  address: { number: 1, name: "cleveland street",suburb: "chippendale",zip: 2008 }
}
```

http://docs.mongodb.org/manual/tutorial/query-documents/#embedded-documents

## Write Query- Insert



```
db.users.insert(
                {
                  name: "sue",
                   age: 26,
                status: "A",
                groups: [ "news", "sports" ]
                }
               )
```



```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```

```
{ name: "al", age: 18, ... }
{ name: "lee", age: 28, ... }
{ name: "jan", age: 21, ... }
{ name: "kai", age: 38, ... }
{ name: "sam", age: 18, ... }
{ name: "mel", age: 38, ... }
{ name: "ryan", age: 31, ... }
{ name: "sue", age: 26, ... }
```

Insert a new document in **users** collection.

## Insert Example

– db.user.insert(**{_id: 12345**, name: “Joe Smith”, emails: [“joe@gmail.com”, “joe@ibm.com”],age: 30})

– db.user.insert({ _id: 54321, name: “Mary Sharp”, email: “mary@gmail.com”, age: 27,
 address: { number: 1, name: “cleveland street”, suburb: “chippendale”, zip: 2008}})

## Insert Example

- db.user.insert({**_id: 12345**,  name: “Joe Smith”, emails: [“joe@gmail.com”, “joe@ibm.com”],age: 30})

- db.user.insert({ _id: 54321,  name: “Mary Sharp”, email: “mary@gmail.com”, age: 27,
  address: { number: 1, name: “cleveland street”, suburb: “chippendale”, zip: 2008}})

user collection

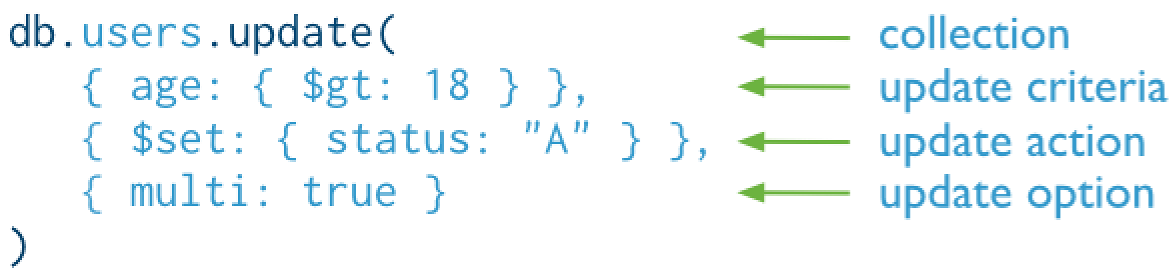
```
{ _id: 12345, name: “Joe Smith”,
  emails: [“joe@gmail.com”, “joe@ibm.com”],
age: 30
}
```

```
{ _id: 54321,
  name: “Mary Sharp”, email: “mary@gmail.com”, age: 27,
  address: { number: 1,
             name: “cleveland street”,
             suburb: “chippendale”,
             zip: 2008
           }
}
```

## Insert Behaviour

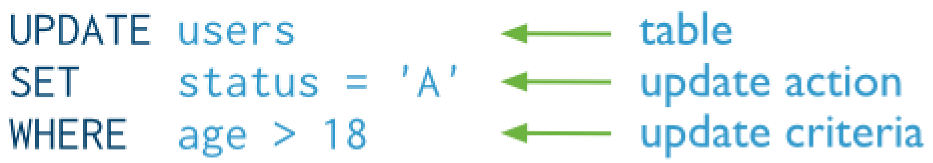
- If the new document does not contain an “**_id**” field, the system will adds an “**_id**” field and assign a unique value to it
- If the new document does contain an “**_id**” field, it should have a unique value

## Write Operation - Update



```
db.users.update(
   { age: { $gt: 18 } },
   { $set: { status: "A" } },
   { multi: true }
)
```

Has the same effect as the following SQL:



```
UPDATE users
SET    status = 'A'
WHERE  age > 18
```

## Updates operators

- Modifying simple field: $set, $unset
  - db.user.update({_id: 12345}, {$set: {age: 29}})
  - db.user.update({_id:54321}, {$unset: {email:1}}) // remove the field

```
{ _id: 12345,
  name: "Joe Smith",
  emails: ["joe@gmail.com", "joe@ibm.com"],
  age: 29}
```

```
{ _id: 54321,
  name: "Mary Sharp",
  email: "mary@gmail.com",
  age: 27}
```

http://www.mongodb.org/display/DOCS/Updating

## Updates operators

- Modifying array elements: $push, $pushAll, $pull, $pullAll
  - db.user.update({_id: 12345}, {$push: {emails: “joe@hotmail.com”}})
  - db.user.update({_id: 54321},
    {$pushAll: {emails: [“mary@gmail.com”, “mary@microsoft.com”]}})
  - db.user.update({_id: 12345}, {$pull: {emails: “joe@ibm.com”}})

```
{ _id: 12345,
  name: “Joe Smith”,
  emails: [“joe@gmail.com”, “joe@ibm.com”],
  age: 30}
```

```
{ _id: 54321,
  name: “Mary Sharp”,
  email: “mary@gmail.com”,
  age: 27}
```

http://www.mongodb.org/display/DOCS/Updating

## Updates operators

- Modifying simple field: $set, $unset
  - db.user.update({_id: 12345}, {$set: {age: 29}})
  - db.user.update({_id:54321}, {$unset: {email:1}}) // remove the field
- Modifying array elements: $push, $pushAll, $pull, $pullAll
  - db.user.update({_id: 12345}, {$push: {emails: “joe@hotmail.com”}})
  - db.user.update({_id: 54321},
    {$pushAll: {emails: [“mary@gmail.com”, “mary@microsoft.com”]}})
  - db.user.update({_id: 12345}, {$pull: {emails: “joe@ibm.com”}})

| | |
|---|---|
| { **_id: 12345**, name: “Joe Smith”, emails: [“joe@gmail.com”, “joe@ibm.com”], age: 30} | { **_id: 12345**, name: “Joe Smith”, emails: [“joe@gmail.com”, “joe@hotmail.com”], age: 29} |
| { **_id: 54321**, name: “Mary Sharp”, email: “mary@gmail.com”, age: 27} | { **_id: 54321**, name: “Mary Sharp”, emails: [“mary@gmail.com”, “mary@microsoft.com”] age: 27} |

http://www.mongodb.org/display/DOCS/Updating

## Write Operation - Delete

- db.user.remove();
  - Remove all documents in user collection

- db.user.remove({_id: 12345})
  - Remove document with a particular id from user collection

## Aggregation

- Simple and relatively standard data analytics can be achieved by **aggregation**
  - Grouping, summing up value, counting, sorting, etc
  - Running on the DB engine instead of application layer

- Several options
  - Aggregation Pipeline
  - MapReduce
    - JavaScript functions
    - Performance
    - Customized aggregations

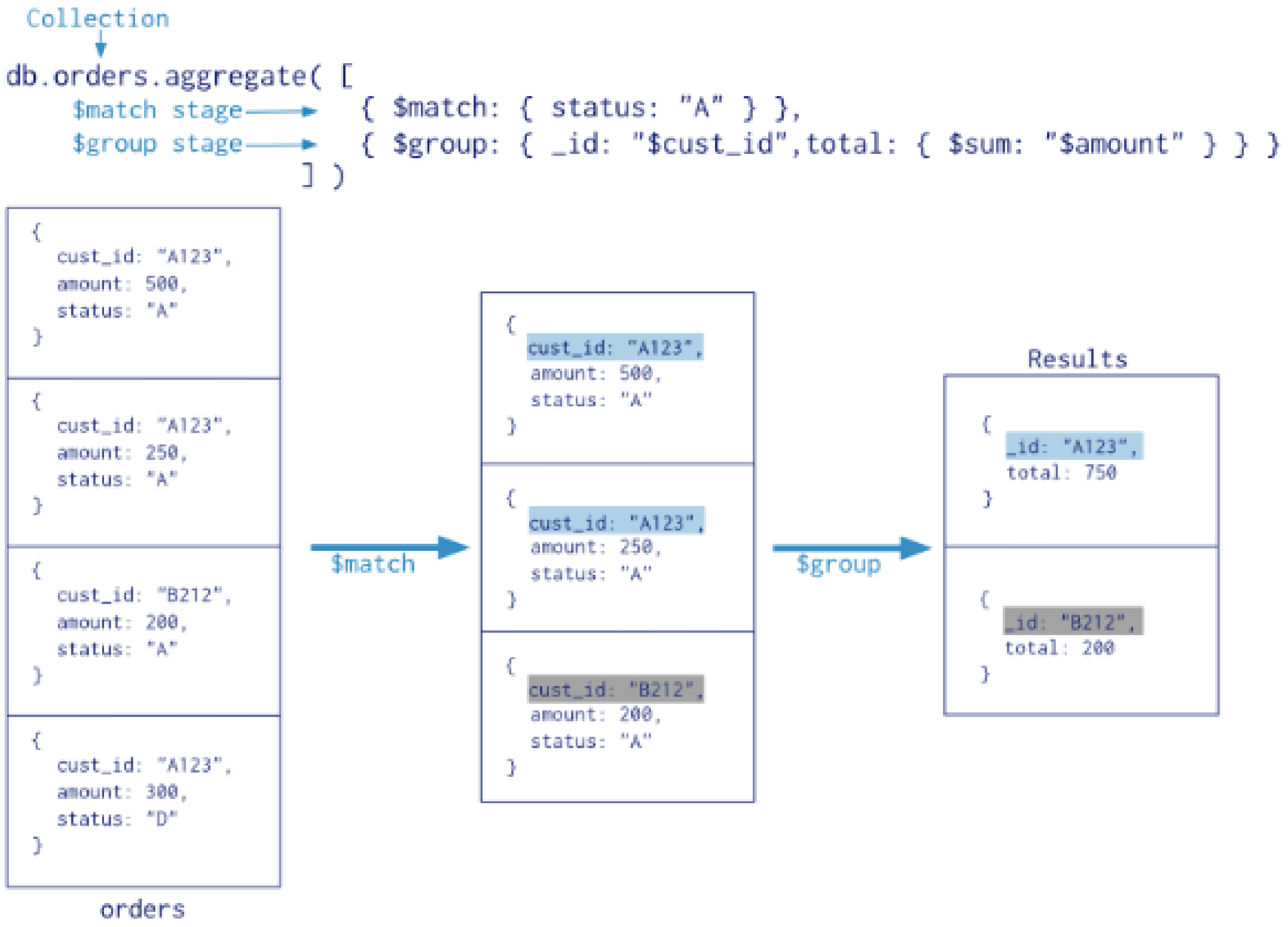
## Aggregation Pipeline

- Consists of multiple stages
  - Specified using **pipeline operators** **(e.g., $match, $group, $sort** and so on)
    - Similar to SQL's WHERE, GROUP BY, SORT BY etc
    - Each stage is expressed as an object enclosed by curly bracket
  - Various **expressions** can be specified in each stage
    - To filter documents or to perform simple calculation on a document
      - `$substr`, `$size`, etc, …
  - **`$group`** stage can specify **accumulators** to perform calculation on documents with the same group key

## Aggregation Pipeline – Format

```
db.collection.aggregate( [
          { pipeline operator: {expression/accumulator,…, expression/accumulator} },
          { pipeline operator: {expression/accumulator,…, expression/accumulator} },
          ...
          ] )
```

## Aggregation Example



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

orders:

```
{
   cust_id: "A123",
   amount: 500,
   status: "A"
}
{
   cust_id: "A123",
   amount: 250,
   status: "A"
}
{
   cust_id: "B212",
   amount: 200,
   status: "A"
}
{
   cust_id: "A123",
   amount: 300,
   status: "D"
}
```

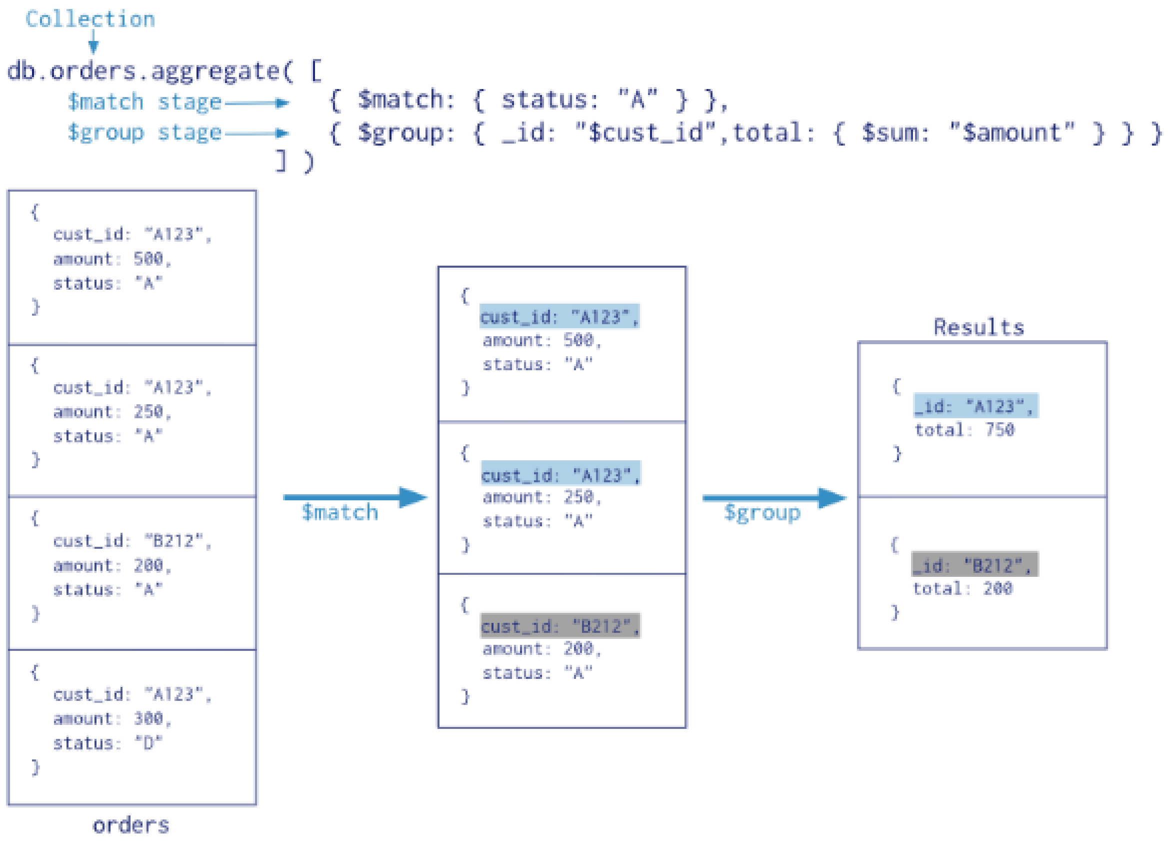
After $match:

```
{
   cust_id: "A123",
   amount: 500,
   status: "A"
}
{
   cust_id: "A123",
   amount: 250,
   status: "A"
}
{
   cust_id: "B212",
   amount: 200,
   status: "A"
}
```

Results:

```
{
   _id: "A123",
   total: 750
}
{
   _id: "B212",
   total: 200
}
```

# Aggregation Example



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

Collection
$match stage
$group stage
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
$match
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
$group
Results
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

```
select  cust_id as _id, SUM(amount) as total
    from orders
    where  status = "A"
    group by cust_id
```

## Aggregation Behaviour

- Operates on a single collection (before 3.2)
  - Join can be performed using a particular operator **$lookup**
- Logically passes the *entire* collection into the pipeline
- Early filtering can improve the performance
- **$match** and **$sort** operators are able to use index if placed at the beginning of the pipeline

## Resources

- Haviv, Amos Q, MEAN Web Development
  - E-book, accessible from USYD library
  - Chapter 4 and 5

- MongoDB online documents:
  - MongoDB CRUD Operations
    - http://docs.mongodb.org/manual/core/crud-introduction/
  - MongoDB Aggregation
    - http://docs.mongodb.org/manual/core/aggregation-introduction/

## Week 6 Tutorial: Node.js MVC Application

## Week 7 Lecture: Connecting to MongoDB

THE UNIVERSITY OF SYDNEY