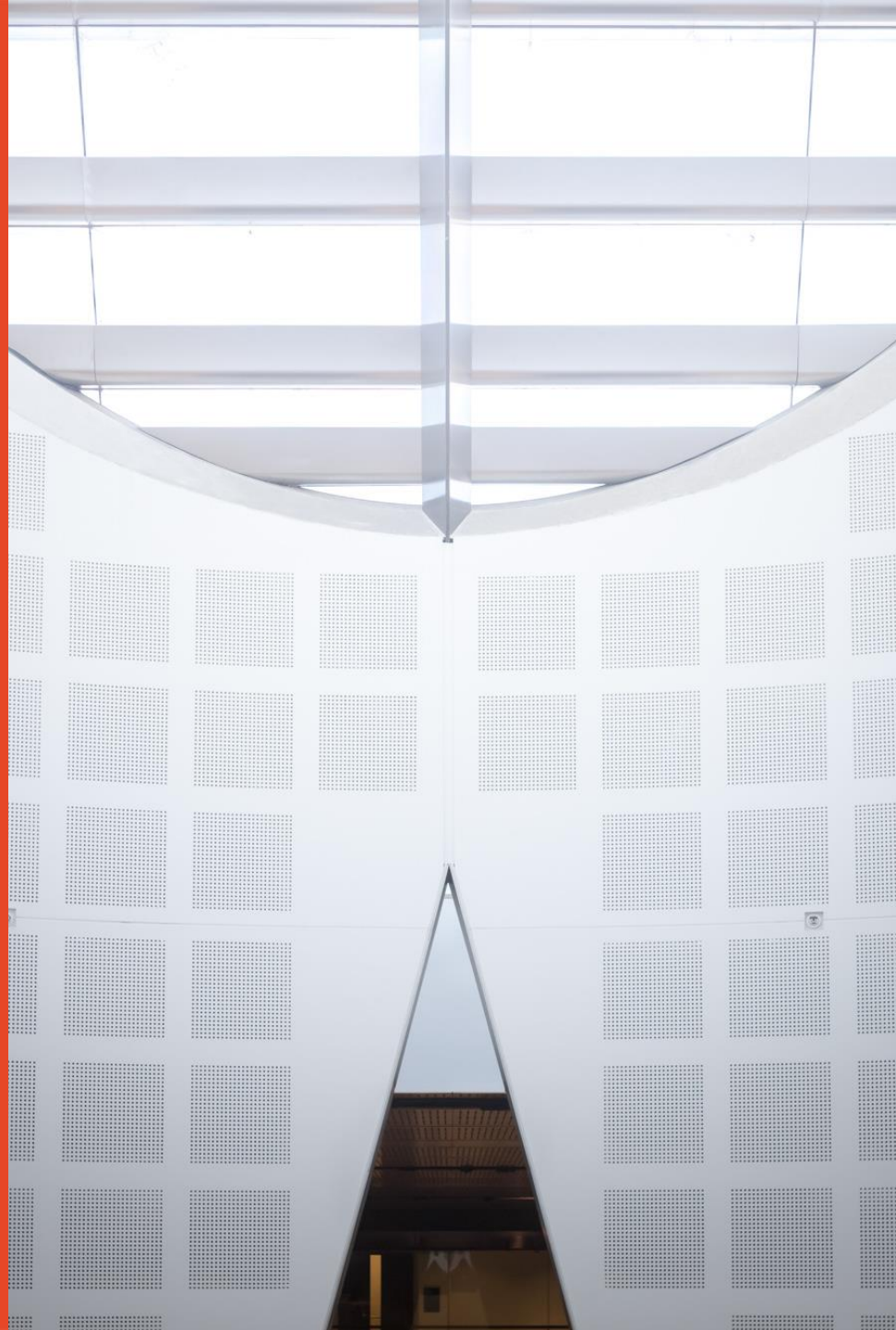
COMP5347 Web Application Development

Connecting to MongoDB

Dr. Basem Suleiman

School of Computer Science



THE UNIVERSITY OF SYDNEY

COMMONWEALTH OF
Copyright Regulations 1969
WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

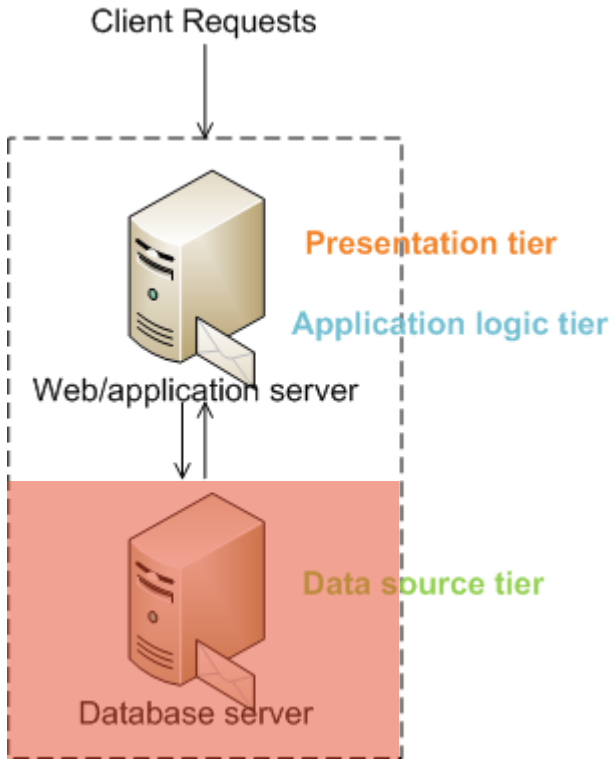
Do not remove this notice.

Outline

- MongoDB indexing
- Database
  - Data layer (MVC)
- Mongoose

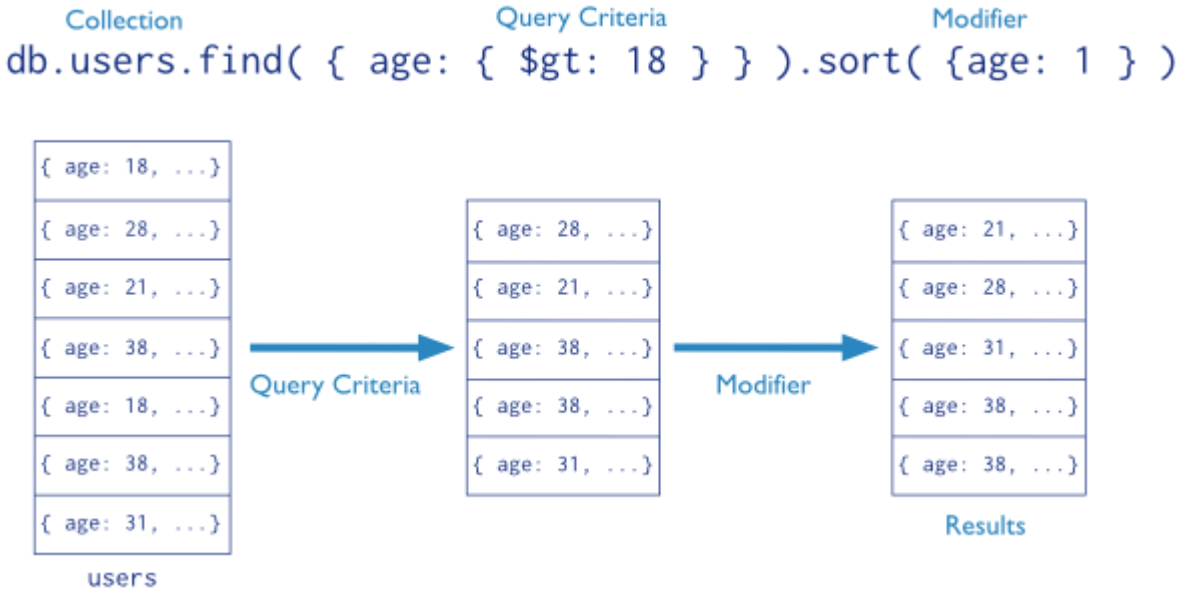
Databases Layer/Tier

- Database tier in Multi-tier (n-tier) Architecture
  - Maintain persistent data of the application
  - CRUD operations (Create, Read, Update, Delete)
- Request/query processing require network communication and server processing
- Many ways to improve performance
  - Hardware
  - Software/application
    - Database level



MongoDB Queries

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



- Creating an appropriate index can help to limit the number of documents it must read

Indexing

- An index is a data structure that makes it efficient to find certain rows/documents in a table/collection
- Indexes support efficient query execution
- Indexing can help to improve database performance if it is done properly
- Most DBMS providers provide facility for indexing

Indexing

- An index consists of records (called *index entries*) each of which has a value for the attribute(s)

- Index files are typically much smaller than the original file

- Most MongoDB indexes are organized as **`B-Tree`** structure

MongoDB Indexes

- The `_id` index
  - `_id` field is automatically indexed for all collections
  - The `_id` index enforces uniqueness for its keys
- The _id index cannot be dropped
- If you do not use the _id as a key, your application must maintain unique values in the _id field

https://docs.mongodb.com/manual/indexes/

MongoDB Indexes – Single Field Index

- Single-field index
  - An index that can be created on a single field of a document
  - Additional properties can be specified for an index:
    - Sparse: an index only contain entries that have the indexed field
    - Unique: MongoDB rejects duplicate values for the indexed field

https://docs.mongodb.com/manual/indexes/

MongoDB – Creating Indexes

- Generic format for creating an index in MongoDB

```
db.<collectionName>.createIndex({<fieldName>:direction})
```

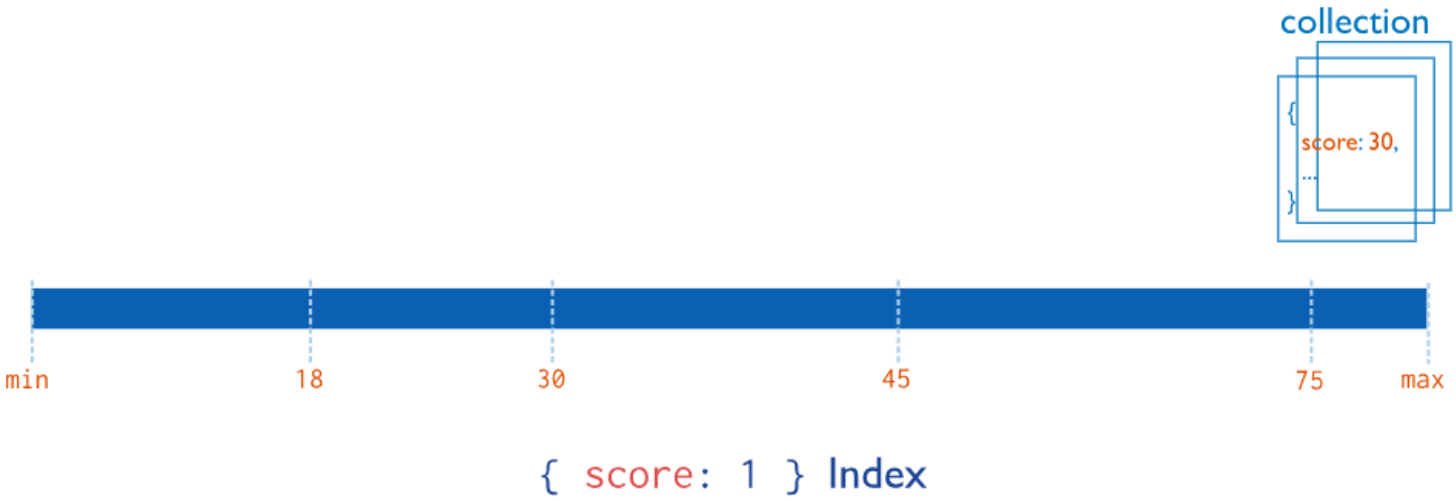
- **fieldName** can be a simple field, array field or field of an embedded document (using dot notation)
- **direction** specifies the direction of the index (1: ascending; -1: descending)

- Examples:
  - db.blog.createIndex({author:1})
  - db.blog.createIndex({tags:-1})
  - db.blog.createIndex({"comments.author":1})

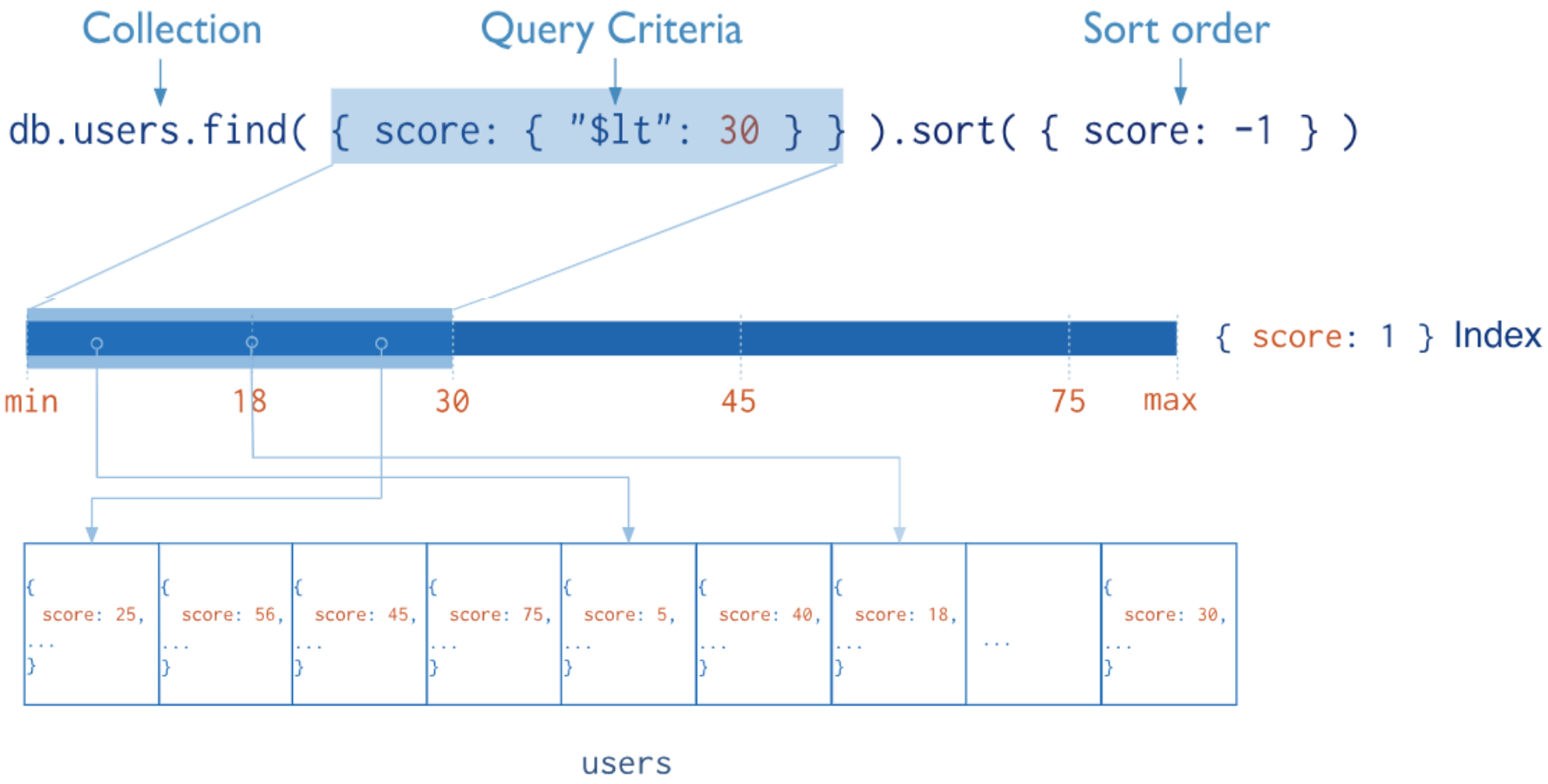
https://docs.mongodb.com/manual/indexes/

Single Field Index – Example

db.*users*.createIndex({score:1})



Single Field Index – Example



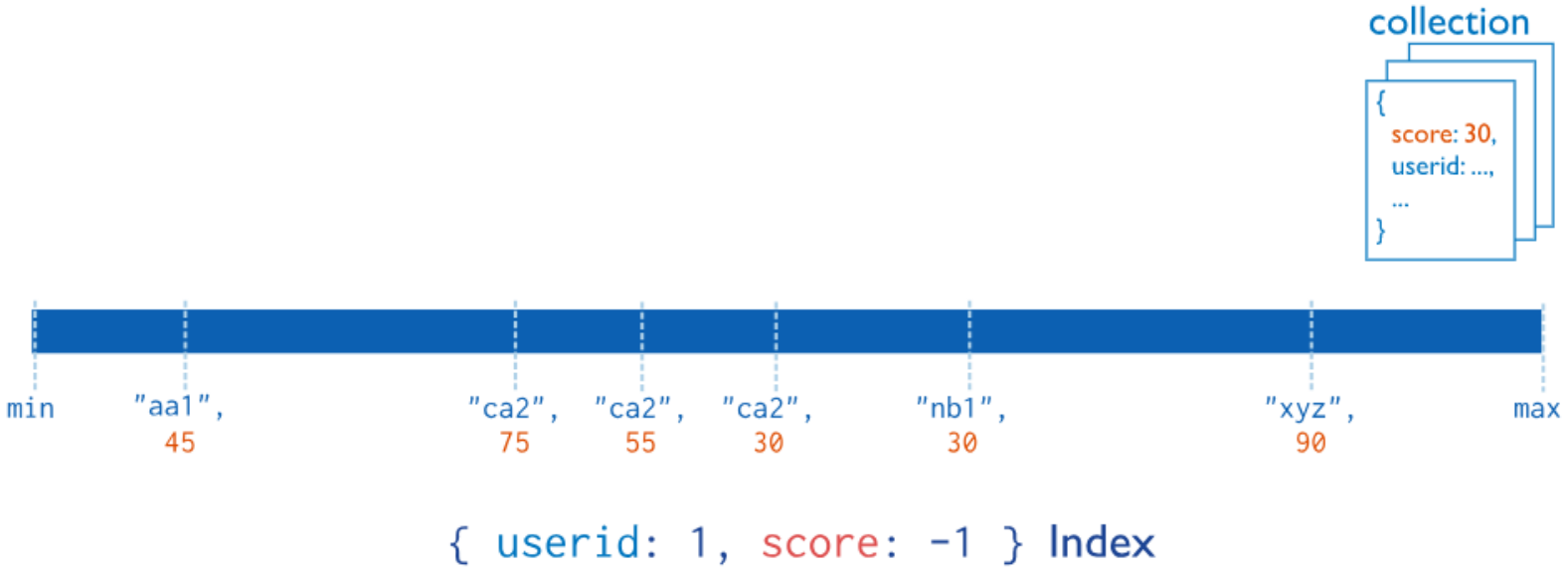
MongoDB – Compound Index

- Compound index is a single index structure that holds references to multiple fields within a collection

- The order of field in a compound index is very important
  - The indexes are sorted by the value of the first field, then second, third…
  - It supports queries like
    - db.users.find({userid: “ca2”, score: {$gt:30} })
    - db.users.find({userid: “ca2”})

Compound Index – Example

db.creatIndex({userid: 1, score: -1})



Designing Indexes

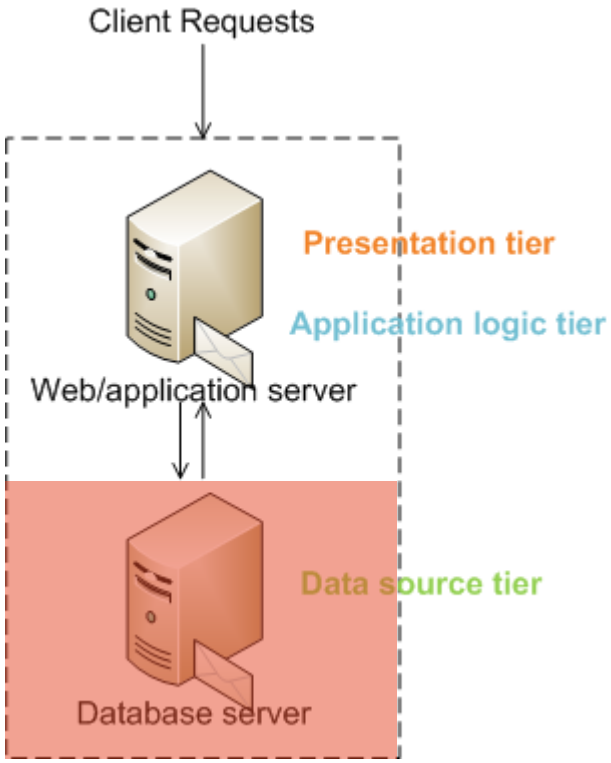
- Understand the application requirements and queries
- Identify types of queries that need to be issued to the database
  - Frequency of key queries
  - Read/write and performance implications
  - Available memory on your server
  - Compare and prioritize – trade-off analysis
- Performance profiling
  - Experiment with a variety of index configurations with data sets
  - Choose the best configuration
- Review indexes on regular basis

Outline

- MongoDB indexing
- Database
  - Data layer
- Mongoose

Web Applications – Database

- Database tier in Multi-tier (n-tier) application Architecture
  - Maintain persistent data of the application
  - CRUD operations (Create, Read, Update, Delete)
- Database Server / DBMS
  - RDBMS (MySQL, PostgreSQL)
  - NoSQL DBMS (MongoDB, Redis)
  - Choice of DBMS is crucial
- Express integrates with many DBMS
  - MySQL, PostgreSQL, MongoDB, Redis, many other*



https://expressjs.com/en/guide/database-integration.html

Database Drivers

- All database management systems work like a "server" application
  - Running on a host and waiting for connections from clients
    - Simple command line shell client
    - GUI shell client
    - Program-based client
  - There are different protocols db server used to communicate with their clients

- All database management systems provide language based drivers to allow developers to write client in various languages
  - Open/close connection to database
  - Translate between language specific construct (functions, methods) and DB queries
  - Translate between language specific data types and database defined data types

- MongoDB provides many native drivers:
  - https://docs.mongodb.com/ecosystem/drivers/

Higher level module/package

- The native DB drivers provide basic supports for client-side programming
  - Powerful, flexible
  - But usually not easy to use

- Higher level modules usually provide more convenient ways to communicate with DB servers
  - *Mongooes* is the node.js module built on top of basic mongodb node.js driver
    - Data structure to match collection "schema"
    - Validation mechanism
    - Connection management
    - Etc.

Object Data Model / Object Relational Model

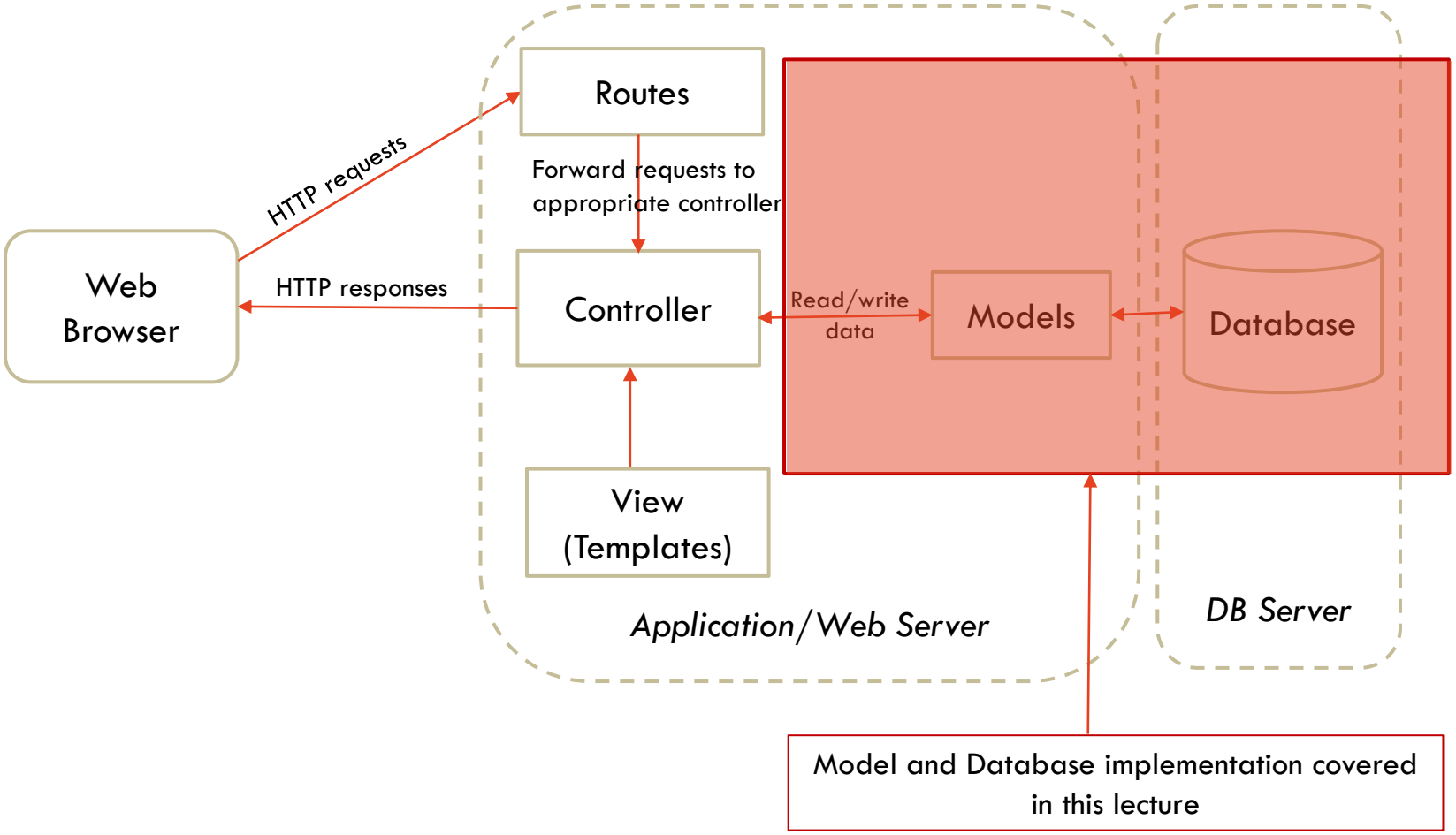
- Approaches to interact with a database
  - Database native query language (e.g., SQL)
  - Object Data Model (ODM) / Object Relational Model (ORM)

- Represents the web application data as objects, to be mapped to the DB
  - Productivity
  - Performance

- Node.js supports many ODM/ORM solutions*
  - Mongoose: a MongoDB object modeling tool for asynchronous environment
  - Others; Sequellize, Objection, Waterline
  - Consider features supported, and the community activity
  - Mongoose will be used to access data from MongoDB database

https://www.npmjs.com/search?q=keywords:odm

MVC Application Architecture



Outline

- MongoDB indexing
- Database
  - Data layer
- Mongoose (MVC)

Mongoose

- All database operations should be implemented using event-driven programming style
  - Start an operation
  - Register a *callback* function to indicate what we want to do when the operation completes
  - Continue processing other parts of the program

Mongoose – Basic Concepts

- Schema
  - Schema is an *abstract* data structure defines the shape of the documents in a collection
  - Each name/value pair is a path
- Model
  - Model is a compiled version of schema, model is the schema binded with a collection
- Document
  - Document is an instance of Model, mapped to the actual document in a collection

Mongoose – Example

- A collection "**movies**" with the example document

```
{    "_id" : 1,
    "Title" : "Sense and Sensibility",
     "Year" : 1995,
     "Genres" : [ "Comedy", "Drama",
"Romance"]
}
```

Mongoose – Schema

- A collection "**movies**" with the example document

```
{    "_id" : 1,
    "Title" : "Sense and Sensibility",
     "Year" : 1995,
      "Genres" : [ "Comedy", "Drama",
"Romance"]
}
```

- Schema definition

```
var movieSchema = new Schema({
  Title: String,
   Year: Number,
  Genres: [String]
})
```

Mongoose – Schema, Model and Document

- A collection "**movies**" with the example document

```
{   "_id" : 1,
   "Title" : "Sense and Sensibility",
    "Year" : 1995,
    "Genres" : [ "Comedy", "Drama",
"Romance"]
}
```

- Schema definition

```
var movieSchema = new Schema({
  Title: String,
   Year: Number,
  Genres: [String]
})
```

- Model definition (collection name, schema, collection name)
- Save a document in a movie collection

```
var Movie = mongooes.model('Movie',
movieSchema, 'movies')

var aMovie = new Movie({
    title="Ride With the Devil"})
```

Mongoose – Queries

- All Mongodb queries run on a model
  - Including **`find`**, **`update`**, **`aggregate`**
  - Very similar syntax to the shell command query
  - A callback function needs to be specified if we want to do something with the query result
  - Two ways to run the callback function
    - Callback function is passed as a *parameter* in the query
      - The operation will be executed immediately with results passed to the callback
    - Callback function is not passed as a parameter in the query
      - An instance of the query is returned which provides a special query builder interface

Queries with Callback Function

```
Movie.find({}, function(err,movies){
   if (err){
    console.log("Query error!")
   }else{
      console.log(movies)
   }
 }
)
```

Call back function

Queries with Callback Function

- The query was executed immediately, and the results passed to the callback
  - Callback syntax in Mongoose: **callback (error, results)**
  - If successful, results will be populated with the query results, error will be null
  - If unsuccessful error will contain error document and the result will be null
  - Result depends on the operations: e.g., find() list of documents, count() number of documents, update() the number of documents affected

Query Instance – No Callback Passed

- A Query instance enables you to build up a query using chaining syntax, rather than specifying JSON object
  - A full list of Query helper functions (http://mongoosejs.com/docs/api.html#query-js)

```
Movie.find({Year: 1996})
.select({Title:1,Year:1})
.exec(function(err,movies){
  if (err){
    console.log("Query error!")
  }else{
    console.log("Movies in year 1996:")
    console.log(movies)
  }
 }
)
```

Query Instance – No Callback Passed

- A Query instance enables you to build up a query using chaining syntax, rather than specifying JSON object
  - A full list of Query helper functions (http://mongoosejs.com/docs/api.html#query-js)

```
Var query = Movie.find({Year: 1996});
query.select({Title:1,Year:1});

query.exec(function(err,movies){
  if (err){
    console.log("Query error!")
  }else{
    console.log("Movies in year 1996:")
    console.log(movies)
  }
 }
)
```

Queries – Insert Documents

- First create a document based on the model

- Use *save()* method to insert the new document
  - The model is linked to the collection, so it knows which collection to save this document to

```
var newMovie = new Movie(
{ MovieID: 292,
  Title: "Outbreak",
  Year: 1995,
  Genres: ['Action','Drama','Sci-Fi','Thriller']}
)
newMovie.save()
```

Queries – Static Methods

- To run certain queries often on some collection, we can implement those queries either as *static methods* or as *instance methods*
- *A static method* is defined on the Model (collection), any standard query/aggregation can be implemented as static method
- Better for reusability and modularity of database related code

Static Methods – Example

```
movieSchema.statics.findByYear = function(year, callback){
    return this
            .find({Year: year})
            .select({Title:1,Year:1})
            .exec(callback)
}
var Movie = mongoose.model('Movie', movieSchema, 'movies')
Movie.findByYear(1995, function(err,movies){
        if (err){
         console.log("Query error!")
        }else{
           console.log("Movies in year 1995:")
           console.log(movies)
        }
})
```

Query – Instance Methods

- Instance methods is defined on document instance
- It is often used to create queries based on a given document

Instance Methods

```
movieSchema.methods.findSimilarYear = function(cb) {
  return this.model('Movie').find({ Year: this.Year }, callback);
};

var newMovie = new Movie(
{MovieID: 292,
 Title: "Outbreak",
 Year: 1995,
 Genres: ['Action','Drama','Sci-Fi','Thriller']}
)
newMovie.findSimilarYear(function(err,movies){
  if (err){
    console.log("Query error!")
  }else{
    console.log("The movies released in the same year as " +
        newMovie.Title + " are:")
    console.log(movies)
 }
}
)
```

Database Connection

- Opening and closing connection to database is time consuming
- Let all requests share a pool of connections and only close them when application shuts down
- Mongoose manages connection pool

http://mongoosejs.com/docs/connections.html

Database Connection

- No application level open or close is required
- Mongoose.connect() prepares a number of connections. The callback can handle the success/error

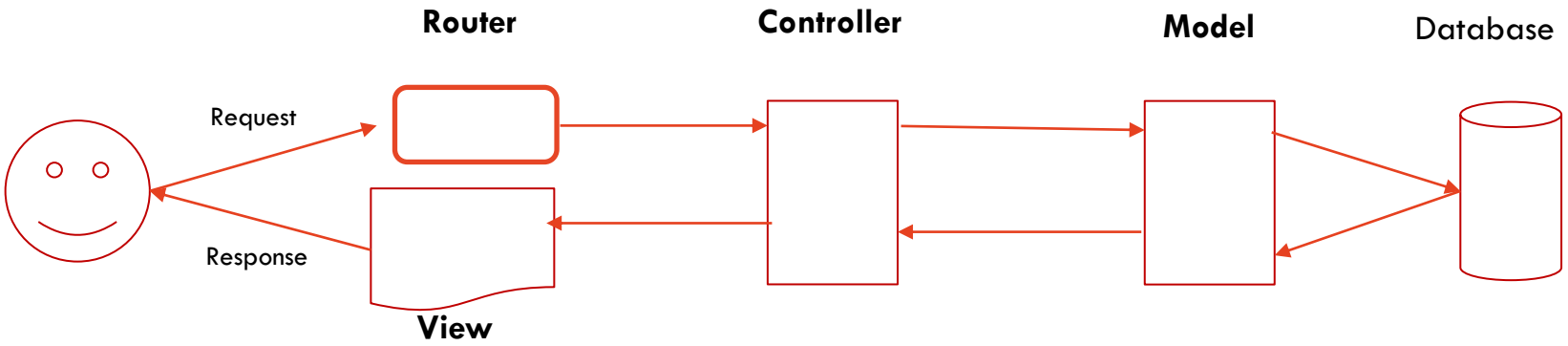
```
var mongoose = require('mongoose')

mongoose.connect('mongodb://localhost/comp5347', function
(err) {
  if (!err)
      console.log('mongodb connected')
})
```

Connection string or database URI

- You can specify more parameters, e.g., mongoose.connect('mongodb://username:password@host:port/database?options...');

http://mongoosejs.com/docs/connections.html

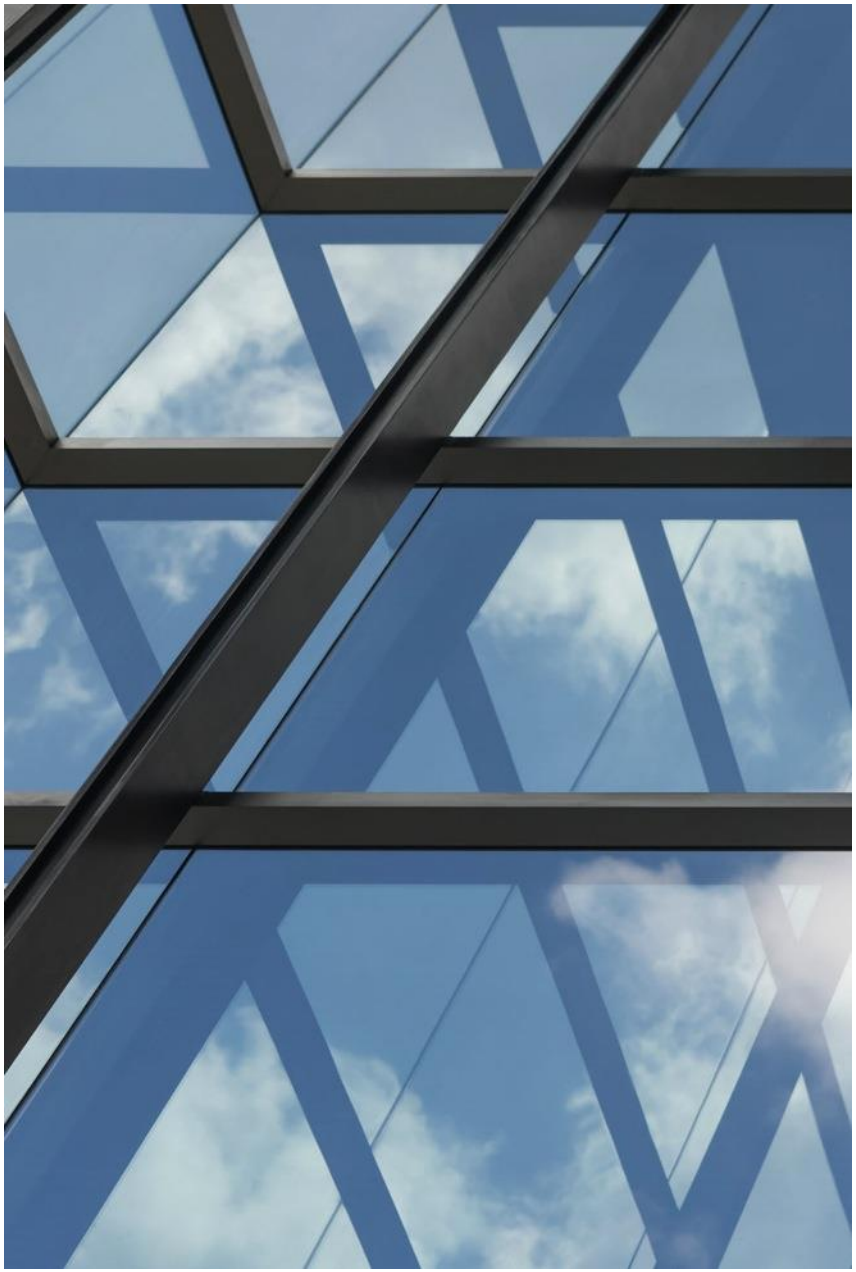
Full MVC Architecture



Resources

- Haviv, Amos Q, MEAN Web Development
- MongoDB online documents:
  - MongoDB CRUD Operations
    - http://docs.mongodb.org/manual/core/crud-introduction/
- Mongooes online documents:
  - Guide: http://mongoosejs.com/docs/guide.html

W7 Tutorial: MongoDB
W8 Tutorial: Mongoose + Promise
W8 Lecture: Client-side Libraries



THE UNIVERSITY OF SYDNEY