Start the database server: sudo mongod. Use “mongo” to connect to the database server. Use “show db” to list the databases. Create a database using “use whiteboard”

- Use the **`insert()`** command to insert data into a collection. Data is formatted as JSON objects
- Inserting user object into **`user`** collection

```
> db.user.insert({
      username:  'ada',
      firstName: 'Ada',
      lastName:  'Lovelace'});
```

- Inserting data creates collection if none existent

“Show collections” to list existing collections.

- Use the **`find()`** command to query documents from a collection

```
> db.user.find()
{ "_id" :
ObjectId("5aa5ab73ab319d0f94089fc7"),
"username" : "ada", "firstName" : "Ada",
"lastName" : "Lovelace" }
```

- Mongo adds unique identifier field **`_id`**
- Use the **`pretty()`** command to pretty print queries

```
> db.user.find().pretty()
{
   "_id" : ObjectId("5aa5ab73...4089fc7"),
   "username" : "ada",
   "firstName" : "Ada",
   "lastName" : "Lovelace"
}
```

- First parameter filters documents
- Use **`ObjecId`** to filter by primary key **`_id`**

```
> db.user.find(
   {_id: ObjectId('5aa5ab73ab319d0f94089fc7')});
{
   "_id" : ObjectId("5aa5ab73ab319d0f94089fc7"),
   "username" : "ada",
   "firstName" : "Ada",
   "lastName" : "Lovelace"
}
```

- **`find()`** matches all fields in first parameter

```
> db.user.find({username: 'ada'})
{
   "_id" : ObjectId("5aa5ab73ab319d0f94089fc7"),
   "username" : "ada",
   "firstName" : "Ada",
   "lastName" : "Lovelace"
}
```

- Second parameter selects fields

```
> db.user.find(
   {username: 'ada'},
   {lastName: 1});
{
   "_id" : ObjectId("5aa5ab73ab319d0f..."),
   "lastName" : "Lovelace"
}
```

- Add additional fields to query to narrow down matching documents

```
> db.section.find({
      course: 'cs101',
      seats:  12});
{
   "_id" : ObjectId("5aa5bd55ab319d0f..."),
   "section" : "01",
   "seats" : 12,
   "course" : "cs101"
}
```

Finding and filtering: ($and,$not,$nor,$or.) db.users.find({age:{$gt:18}}).sort({age:1}). $eq, $gt, $gte, $in(Matches any of the values specified in an array), $lt, $lte, $ne(not equal to a specified value), $nin(matches none of the values specified in an array).

- Use **`$gt`** and **`$lt`** to filter documents

```
> db.section.find({seats: {$gt: 30}})
{
   "_id" : ObjectId("5aa5b7cca...4089fd1"),
   "section" : "03",
   "seats" : 34
}
{
   "_id" : ObjectId("5aa5b7d2...94089fd2"),
   "section" : "04",
   "seats" : 45
}
```

Use `$and` to combine predicates

```
> db.section.find({
   $and: [
      {seats: {$gt: 30}},
      {seats: {$lt: 40}}
   ]})
{
   "_id" : ObjectId("5aa5bd55ab319d0f..."),
   "section" : "03",
   "seats" : 34,
   "course" : "cs102"
}
```

- Sort ascending modifies query ordering

```
> db.section.find().sort({seats: 1})
{ "section" : "01", "seats" : 12}
{ "section" : "03", "seats" : 34}
{ "section" : "04", "seats" : 45}
```

- Sort descending modifies query ordering

```
> db.section.find().sort({seats: -1})
{ "section" : "04", "seats" : 45}
{ "section" : "03", "seats" : 34}
{ "section" : "01", "seats" : 12}
```

$count	Returns a count of the number of documents at this stage of the aggregation pipeline.
$group	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group.
$limit	Passes the first n documents unmodified to the pipeline where n is the specified limit. For each input document, outputs either one document
$skip	Skips the first n documents where n is the specified skip number and passes the remaining documents unmodified to the pipeline.

```
> db.section.update({name: '02'},
   {$set: {seats: 22}})
WriteResult({"nMatched" : 1,"nModified" : 1})
> db.section.find({name: '02'}, {seats: 1})
{ "_id" : ObjectId("5aa5b..."), "seats" : 22 }
```

```
> db.section.remove({name: '02'})
WriteResult({ "nRemoved" : 1 })
```

db.collection.insertOne()	Inserts a single document into a collection.
db.collection.insertMany()	db.collection.insertMany() inserts multiple documents into a collection.
db.collection.insert()	db.collection.insert() inserts a single document or multiple documents into a collection.
db.collection.insertOne()	Inserts a single document into a collection.
db.collection.updateOne()	Updates at most a single document that match a specified filter even though multiple documents may match the specified filter.
db.collection.updateMany()	Update all documents that match a specified filter.
db.collection.replaceOne()	Replaces at most a single document that match a specified filter even though multiple documents may match the specified filter.
db.collection.update()	Either updates or replaces a single document that match a specified filter or updates all documents that match a specified filter.
db.collection.deleteOne()	Delete at most a single document that match a specified filter even though multiple documents may match the specified filter.
db.collection.deleteMany()	Delete all documents that match a specified filter.
db.collection.remove()	Delete a single document or all documents that match a specified filter.
db.collection.deleteOne()	Delete at most a single document that match a specified filter even though multiple documents may match the specified filter.

Generalization: denormalized strategy

- Creating faculties

```
> db.user.update({username: 'ada'},
   {$set: {type: 'FACULTY', office: '132A'}});
> db.user.insert({username: 'tlee', type:
   'FACULTY', office: '133B'})
```

- Retrieving faculties

```
> db.user.find({type: 'FACULTY'})
```

- Creating students

```
> db.user.insert({username: 'john',
   gpa: 3.8, type: 'STUDENT'})
> db.user.insert({username: 'jane',
   type: 'STUDENT'})
> db.user.update({username: 'jane'},
   {$set: { gpa: 3.9}})
```

Many to Many:

- **Using a map: Enrollment**
  - **enrollment1.section = section1._id**
  - **enrollment1.student = student1._id**
- Creating enrollments

```
> db.enrollment.insert({
   section: '5aa5bd55ab319d0f94089fd3',
   student: '5aa9515ad982cb8e72af70ce'
   })
```

- Using arrays of foreign keys
  - student1.sections = [section1._id, section2._id, …]
  - section1.students = [student1._id, student2._id, …]
- **Students** reference **array of** enrolled **sections**

```
> db.student.update({
   _id: ObjectId('5aa95148d982cb8e72af70cd')},
   {$set: {sections:
      ['5aa5bd55ab319d0f94089fd3']}})

> db.student.update({
   _id: ObjectId('5aa95148d982cb8e72af70cd')},
   {$push: {sections:
      '5aa9515ad982cb8e72af70f4'}})
```

One to Many:

Child references parent (classical)

- section.facultyId = faculty._id
- course.authorId = faculty._id

Parent arrays of child foreign keys

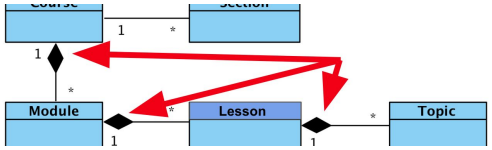
- faculty.sectionsTaught = [section1._id, section2._id, …]
- faculty.coursesAuthored = [course1._id, course2._id, …]

Parent arrays of embedded child object instances

- faculty.sectionsTaught = [section1, section2, …]
- faculty.coursesAuthored = [course1, course2, …]

```
db.section.update({
 _id: ObjectId('5aa5bd55ab319d0f94089fd3')},
 {$set: {
    faculty: '5aa5ab73ab319d0f94089fc7'}
 })
```

Embedded Objects:




```
> db.modules.insert({name: 'module1',
   lessons: [{name: 'lesson1'}]})
> db.modules.find().pretty()
   {
      "_id" : ObjectId("5aa96092d982cb8e7..."),
      "name" : "module1",
      "lessons" : [
         {
            "name" : "lesson1"
```

```
db.modules.update({name: 'module1'},
 {$push: {lessons: {name: 'lesson2'}}});

db.modules.find().pretty()
 {
    "_id" : ObjectId("5aa96092d982cb8e7..."),
    "name" : "module1",
    "lessons" : [
       {     "name" : "lesson1"     },
       {     "name" : "lesson2"     }
    ]
 }
```

Retrieving embedded object

```
> db.modules.find(
     {name: 'module1'}, {lessons: 1})

{
   "_id" : ObjectId("5aa96092d982cb8e72..."),
   "lessons" : [
      {     "name" : "lesson1"     },
      {     "name" : "lesson2"     }
   ]
}
```

Adding objects to nested array

```
> db.modules.update({name: 'module1',
'lessons.name': 'lesson1'}, {$set:
{'lessons.$.topics': [{name: 'topic1'}]}})

> db.modules.find().pretty()
{ "name" : "module1",
  "lessons" : [
    { "name" : "lesson1",
      "topics" : [
        {"name": "topic1"}]
    },
    { "name" : "lesson2"  }]
}
```

The positional $ operator identifies an element in an array to update without explicitly specifying the position of the element in the array

Positional Operator - create a course

```
> db.courses.insert({name: 'course1'})

> db.courses.find().pretty()
{
   "_id" : ObjectId("5aa96463d982cb8e72a..."),
   "name" : "course1"
}
```

Add an empty modules array

```
> db.courses.update({name: 'course1'},
  {$set: {modules: []}});

> db.courses.find({name: 'course1'}).pretty();
{
   "_id" : ObjectId("5aa96463d982cb8e72..."),
   "name" : "course1",
   "modules" : [ ]
}
```

Push an embedded module object

```
> db.courses.update({name: 'course1'},
  {$push: {modules: {name: 'module2'}}});

> db.courses.find({name: 'course1'}).pretty();
{  "_id" : ObjectId("5aa96463d982cb8e72a..."),
   "name" : "course1",
   "modules" : [
      {  "name" : "module1"  },
      {  "name" : "module2"  }
   ]
}
```

Add a module section to the module

```
> db.courses.update({name: 'course1',
   'modules.name': 'module1'}, {$set:
{'modules.$.lessons': [{name:'lesson1'}]}});

> db.courses.find({name: 'course1'}).pretty();
{  "name" : "course1",
   "modules" : [
      {  "name" : "module1",
         "lessons" : [{"name" : "lesson1"}]
      },
      {  "name" : "module2"  }
   ]
}
```

Add a topic to the section

```
> db.courses.update({name: 'course1',
'modules.name': 'module1',
'modules.lessons.name': 'lesson1'}, {$set:
{'modules.$.lessons.0.topics': [{name:
'topic1'}]}});
> db.course.find().pretty()
{ "name" : "course1",
  "modules" :  [{"name" : "module1",
   "lessons" : [{"name" : "lesson1",
    "topics" : [
     { "name" : "topic1" }]
    }],...
```

Mongoose schemas

```
const mongoose = require('mongoose');
const userSchema = mongoose.Schema({
   username: String,
   password: String,
   firstName: String,
   lastName: String
}, {collection: 'user'});
module.exports = userSchema;
```

Mongoose Models

```
const mongoose = require('mongoose');
const userSchema = require
            ('./user.schema.server');
const userModel = mongoose.model
          ('UserModel', userSchema);
module.exports = userModel;
```

Mongoose Test

```
require('./db')();
var userDao = require
      ('./models/user.dao.server');
userDao.find((err, users) => {
   console.log(users);
});
```

userDao.findAllUsers().then(users=>console.log(users)); Mongoose DAO

```
const mongoose = require('mongoose');
const userSchema = ...
const userModel = ...

findAllUsers = () =>
   userModel.find();

module.exports = { findAllUsers }
```

findUserById = userId => userMOdel.find({_id: userId}), findUserByUsername = username => userModel.find({username: username}), createUser = user => userModel.create(user)

```
userDao
   .createUser({
       username: 'paul',
       password: 'muad\'dib',
       firstName: 'Paul',
       lastName: 'Atreides'
   }).then(newUser =>
       console.log(newUser));
```

Dao: update

```
updateUser = (uid, user) =>
  userModel
    .update({_id: uid},
            {$set: user})
```

```
userDao
   .updateUser(
          '5aa9888dae4e5f3818f0b962',
       {firstName: 'Paul Muad\'dib'})
   .then(status => {
       console.log(status);
});
```

Dao: delete

```
deleteUser = userId =>
  userModel
    .remove({_id: userId})
```

```
userDao
   .deleteUser(
     '5aa9888dae4e5f3818f0b962')
   .then(status => {
        console.log(status);
});
```

Many to Many: create section schema

```
const mongoose = require('mongoose')
const sectionSchema =
        mongoose.Schema({
           name: String
        }, {collection: 'section'});
module.exports = sectionSchema;
```

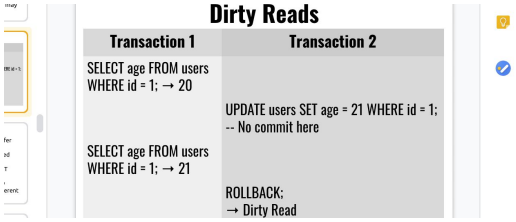
```
const enrollmentSchema = mongoose.Schema({
 student: {ref: 'UserModel',
  type: mongoose.Schema.Types.ObjectId},
 section: {ref: 'SectionModel',
  type: mongoose.Schema.Types.ObjectId}
});
```

Transaction Handling: Transaction is atomic unit of work; All database interaction are treated as unit within transaction; Within transaction, if any SQL command fails, they all fail; All SQL commands must succeed for all to succeed.

Purpose: Recovery from failure; Reliable units of work; Keep database in consistent state
Isolate concurrent processes accessing database.

ACID: **Atomic**–all or nothing
Consistency–transactions ensure database transitions between consistent states
Isolation–concurrent execution must be unaware of each other. **Durability**–committed transactions remain so, despite power loss or crashes.**Transaction Commit and Rollback** Updates in transaction are temporary; Permanent on commits; Other clients don't see temporary changes; Only see permanent changes. JDBC autocommit by default, each statement in own transaction.

Isolation Levels:
Read Uncommitted(Read while others update); **Read Committed**; **Repeatable Reads**(Already read, other updated, change based on previous read); **Serializable(**block other transaction from changing it); **Dirty Reads:** Similar to non-repeatable reads; Transaction 2 changes row, but does not commit;T1 reads uncommitted data; If T2 rolls back or updates different changes; Then T1 view of data may be wrong;



Non-repeatable reads: Row retrieved twice and values differ between reads; Occurs when reading locks not acquired when performing a SELECT; Or locks released as soon as SELECT operating is performed; T2 commits successfully, But T1 already read a different value

Non-repeatable reads

Transaction 1	Transaction 2
SELECT * FROM users WHERE id = 1;	
	UPDATE users SET age = 21 WHERE id = 1; COMMIT;
SELECT * FROM users WHERE id = 1; COMMIT;	

Phantom Reads: When new rows added by another transaction to records being read; When range locks not acquired on performing a SELECT...WHERE; Speical case of Non-repeatable reads; T1 repeats a ranged SELECT … WHERE; T2 inserts new rows fulfilling WHERE clause

Phantom Reads

Transaction 1	Transaction 2
SELECT * FROM users WHERE age BETWEEN 10 AND 30;	
	INSERT INTO users(id,name,age) VALUES (3, 'Bob', 27); COMMIT;
SELECT * FROM users WHERE age BETWEEN 10 AND 30; COMMIT;	

- **Read-Uncommitted** – no isolation at all
- **Read-Committed** – can't read uncommitted data, non-repeatable reads and phantom records still possible
- **Repeatable-Read** – fixes non-repeatable reads (guarantees repeatable reads), phantoms still possible
- **Serializable** – locks everything, no problems, slow

Read Uncommitted Isolation Level: Lowest isolation level; Dirty reads are allowed; One transaction may see changes not-yet-committed by other transactions; Fastest;

Read Committed Isolation Level: Guarantees data read is committed; Restricts reader from reading intermediate, uncommitted, 'dirty' read; Data is free to change after read; If transaction re-issues read, it might have changed; Read & write locks acquired on selected data; Released at end of the transaction; Read locks released as soon as SELECT is performed; No range-locks are managed; Non-repeatable & Phantom reads can occur; Slower;

Repeatable Reads Isolation Level: Read & write locks acquired on selected data; Released at end of the transaction; No range-locks managed; Phantom reads can occur; Even slower;

Serializable Isolation Leve: Highest isolation level; Read & write locks acquired on selected data; Released at the end of the transaction; Range-locks acquired when SELECT uses ranged WHERE clause to avoid phantom reads; Locks everything, no problems, slowest.

Isolation Level/ Inconsistency	Dirty Reads	Non-repeatable Reads	Phantom Reads
Read Uncommitted	Unsafe	Unsafe	Unsafe
Read Committed	Safe	Unsafe	Unsafe
Repeatable Read	Safe	Safe	Unsafe
Serializable	Safe	Safe	Safe

? ?

An index is a smaller, easier to maintain, **copy of original table**; File with index values, **record identifiers(RIDs)** referencing original records; Unique keyword identify index as referring to a key; **Create INDEX <name> ON <table> (<field>);** Queries on **UNIQUE indexes** refer to single record, i.e., searching for SID_IDX for SId=10, refers to single record;

Queries on **NON UNIQUE indexes** may refer to several records, i.e., searching for MAJOR_IDX for MajorId=10, may refer to several records;

Index Efficiency: If we had no indexes, it's **linear search** for Sid. **Disk Block:** It is expensive to read / write to the database, a lot of over head; Instead of reading / writing individual records, databases read / write in blocks; Reading / writing to disk is most expensive operations - I/O is the cost; Optimize on I/O cost;

- I/O costs for indexed and non-indexed approaches:
  - Non-indexed costs: 40,000 / 10 = **4,000** reads, ***worst case***
  - Indexed costs: **couple** of reads to find index and RID, **one more** read to retrieve record from STUDENT with RID
- About **4,000** / **3** ~ 1300 times faster with indexes

Without Indexes

- Without indexes must search linearly
  1. **For each record in STUDENT:**
     If *MajorId-value of record is 20 then:*
     Add *SId-value of record to output list*
  2. **Return the output list.**
- Still ***4,000*** disk reads
- All 10 records in block are compared (at most)

With Indexes

1. **Use the MAJOR_IDX index to find the index records whose *MajorId-value is 20.***
2. **For each such index record:**
   a. Obtain the value of its *RID field.*
   b. Move directly to STUDENT record having RID.
   c. Add the *SId-value of record to output list.*
3. **Return the output list**

Inefficiencies of Indexes

- Non-unique indexes are less efficient than unique indexes, i.e., on primary keys
- The more matching records, the less efficient indexes are, for example:
- Consider **8 departments → 5,000** students / major
- Using indexing, read **1 block for each 5,000** students
- But, there are only **4,000** blocks → redundant reads
- Using **indexes** would be **worse than linear** search

Composite Indexes

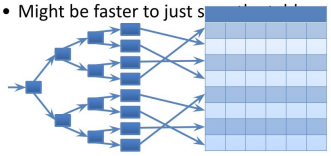
- If most select queries use ***both indexes***, then consider a ***composite index***
  CREATE INDEX MODEL_YEAR ON CARS (MODEL, YEAR)

- Column order defines sort-priority order
- ASC or DESC next to column to define sort order
- Easier to maintain one index on two cols then two separate indexes

Order of columns matters. If both indexes in query, order does not matter WHERE..AND..
If only one index in where, then order matters

To Index or Not To Index

- Indexes great finding specific value / criteria, PKs
- Not great finding many values, ranges, multi rows
- Might be faster to just s… Scan



Indexable Queries

- Indices participate in different selection predicates
- Predicates comparing indexed fields with constants
- Predicates comparing indexed fields with a range
  - compare indexed fields with constants using "<" and ">" operators
  - can be implemented by arranging indexes in sorted order
  - if too many matches just scan table
- Predicates joining indexed fields by comparing fields with other fields

Select SName from Student WHERE Sid > 8

Without an index:1. For each record in STUDENT: {If the SId-value of this record is > 8 then: {Add SName-value of record to output list}}
2. Return the output list. If there are too many students with SId > 8, we might as well scan the table. **Indexed Implementation:** 1. Use the SID_IDX index to find the last index record whose SId-value is 8. 2. For the remaining records in the index (assuming that they are in sorted order): {a. Obtain the value of its RID field
b. Move directly to the STUDENT record having that RID. c. Add the SName-value of that record to the output list} 3. Return the output list.

select s.SName, e.Grade from STUDENT s, ENROLL e where s.SId=e.StudentId.

Non-Indexed: For each record in ENROLL: For each record in STUDENT:{If SId = StudentId then: {Add SName and Grade to the output list}}
2. Return the output list. Scans entire 4,000 blocks of STUDENT table, for each ENROLL record.

Indexed: For each record in ENROLL:
1. Let x be the StudentId-value of that record
2.Use SID_IDX to find index record whose SId-value = x **3.**Obtain the value of its RID field
4. Move directly to STUDENT record having that RID
5. Add values for SName and Grade to the output list
2. Return the output list
Faster since not a nested loop, only cost to loop through ENROLL.

select d.DName from STUDENT s, DEPT d where s.MajorId=d.DId and s.SId=8

Index SID-ID: 1. Use the SID_IDX index to find the index record having SId=8. **2.** Obtain the value of its RID field. **3.** Move directly to the STUDENT record having that RID. Let x be the MajorId-value of that record. **4.** For each record in DEPT: {If the DId-value of that record = x then: {Return the value of DName.}}

Index MAJOR_IDX: For each record in DEPT:{
1.Let y be the DId-value of that record.
2.Use the MAJOR_IDX index to find the index records whose MajorId-value = y. **3.**For each such index record: {**1.**Obtain the value of its RID field. **2.** Move directly to the STUDENT record having that RID. **3.** If SId=8 then:
{Return the value of DName.}}

Indexes have costs and benefits
Benefit = cost savings x how frequent
Cost savings (Total number of records, Number of records per block, Number of distinct values of indexed fields) **Cost (**Disk space to keep indexes, N fields, 1/N index size, if index all fields total size > 2N , Inserts / Deletes / Updates → update indexes, several index records) ///**XML** stands for eXtensible Markup Language. **Document Object Model. Language** – vocabulary, syntax, grammar. **Markup** – annotations providing meta-data, meaning, semantics, context. **Extendable** – can be used in any context. **Elements** – annotate data.

- XML document constraints can be expressed as XML Schema documents
- XML schema describe valid XML document structures, data types, and relations
- Document Type Definitions (DTDs) is a popular XML schema language
- DTDs can be used to validate XML documents

Enrollments – root element, grouping element; **CourseTaken** – one element per course; **Tags**; **Bracketed tags and Empty tags**; **Attributes**; **Semantics** can be captured in tags and/or attributes; **Sibling** meanings can be captured in parent's context; **/** All tags must be closed; **/** Tags can be mixed case, but matching opening and closing tags must be capitalized the same way **/** Tags (or elements) can be arbitrarily nested, describing complex data structures; **/** Tags, as well as nested

tags, must be closed in the correct order

Parent Semantics for Siblings

```
<Enrollments gradyear="2009">
  <CourseTaken sname="ian" title="calculus"
  gradyear="2009"
   yeartaken="2006" grade="A" />
  <CourseTaken sname="ian" title="shakespeare"
   yeartaken="2006" grade="C" />
  <CourseTaken sname="ian" title="pop culture"
   yeartaken="2007" grade="B+" />
  <CourseTaken sname="ben" title="shakespeare"
   yeartaken="2006" grade="A-" />
  <CourseTaken sname="ben" title="databases"
   yeartaken="2007" grade="A" />
</Enrollments>
```

Nested Tags Instead of Attributes

```
<GraduatingStudents>
  <Student>
    <SName>ian</SName>
    <GradYear>2009</GradYear>
    <Courses>
      <Course>
        <Title>calculus</Title>
        <YearTaken>2006</YearTaken>
        <Grade>A</Grade>
      </Course>
      <Course>
        <Title>shakespeare</Title>
        <YearTaken>2006</YearTaken>
        <Grade>C</Grade>
      </Course>…
    </Courses>
  </Student>...
</GraduatingStudents>
```

Normalization: combine elements in the same tag; **Redundancy:** use less tags; Inconsistency:
An XML **element** is everything from (including) the element's start tag to (including) the element's end tag;

```
<person sex="female">   sex is an attribute
  <firstname>Anna</firstname>   firstname is an
  <lastname>Smith</lastname>    element
</person>
```

Children on the same level are called **siblings**;
A **leaf** is a node with no children;
According to the XML DOM, everything in an XML document is a **node**:

- The entire document is a document node
- Every XML element is an element node
- The text in the XML elements are text nodes
- Every attribute is an attribute node
- Comments are comment nodes

Self-closing tag: <element />; **XML tag** opened inside another element must be closed before the outer element is closed.
Java Architecture for XML Binding: @XmlRootElement - maps Java class to XML root element; **@XmlElement** - maps Java field to XML element; **@XmlAttribute** - maps Java field to an XML element's attribute;

```
@XmlRootElement
public class Student {
   @XmlAttribute
   public void setId(String id) {
      this.id = id;
   }
   @XmlElement
   public void setFirstName(String firstName) {
      this.firstName = firstName;
   } ...etc...
}
```

Use **XSLT** to generate / transform XML documents; Stands for **eXtensible Stylesheet Language Transformation; Declarative** programming; Elements can be moved to other parts of document; Nested elements can become siblings;

XML to HTML:

```
<xsl:template match="GraduatingStudents">
 <html><body>
  <table border="1" cellpadding="2">
  <tr> <th>Name</th> <th>Grad Year</th>
       <th>Courses</th> </tr>
    <xsl:apply-templates select="Student"/>
  </table>
 </body></html>
</xsl:template>
```

An **XPath** expression identifies the paths in the element tree that satisfy a given condition

nodename	Selects all nodes w/ name "*nodename*"
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

bookstore	Selects all nodes w/ name "bookstore"
/bookstore	Selects the root element bookstore
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document
bookstore//book	Selects all book elements that are descendant of the bookstore element
//@lang	Selects all attributes named lang

/bookstore/book[1]	Selects the first book element that is the child of the bookstore element
/bookstore/book[last()]	Selects the last book
/bookstore/book[last()-1]	Selects the last but one
/bookstore/book[position()<3]	Selects the first two book
//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='en']	Selects all title elements that have an attribute lang with a value of 'en'
/bookstore/book[price>35.00]	Selects all the book elements that have a price than 35.00
/bookstore/book[price>35.00]/title	Selects all title elements with a value greater than 35.00

- Hierarchy can be flatten

```
<GraduatingStudents>
  <Student>
    <SName>ian</SName>
    <GradYear>…
    <Courses>
      <Course>
  </Student>...
</GraduatingStudents>
```

→ →

```
<Enrollments>
  <CourseTaken…>
  <CourseTaken…>
  <CourseTaken…>
</Enrollments>
```

```
<xsl:stylesheet xmlns:xsl="…" version="2.0">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="GraduatingStudents">
   <Enrollments>
     <xsl:apply-templates
       select="Student/Courses/Course"/>
   </Enrollments>
  </xsl:template>
```

```
<Enrollments>
  <CourseTaken…>
  <CourseTaken…>
  <CourseTaken…>
</Enrollments>
```

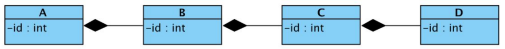
→ →

```
<GraduatingStudents>
  <Student>
    <SName>ian</SName>
    <GradYear>…
    <Courses>
      <Course>…
  </Student>...
</GraduatingStudents>
```

```
<xsl:stylesheet xmlns:xsl="…" version="2.0">
 <xsl:output method="xml" indent="yes"/>
 <xsl:template match="Enrollments">
  <GraduatingStudents>
   <xsl:for-each-group select="CourseTaken"
                group-by="@sname">
    <Student>
     <Sname><xsl:value-of select="@sname"/>
     </Sname>
     <GradYear>
      <xsl:value-of select="@gradyear"/>
     </GradYear>
```

```
<Courses>
 <xsl:for-each select="current-group()">
  <Course>
   <Title><xsl:value-of select="@title"/>
   </Title>
   <YearTaken>
    <xsl:value-of select="@yeartaken"/>
   </YearTaken>
   <Grade><xsl:value-of select="@grade"/>
   </Grade>
  </Course>
 </xsl:for-each>
</Courses>
```

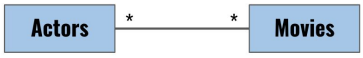
Representational state transfer (REST) is a set of architectural conventions for creating Web services(POST, GET, PUT, DELETE)



Class A represents all instances of type A, referred to as **entities**

Instances can be implemented as **records** in a database or **object instances** in a running program

POST	/A	Create new entity of type A. Data in HTTP body
GET	/A	Retrieve all instances of entity type A, e.g., array
GET	/A/123	Retrieve single instance, whose ID is 123
PUT	/A/234	Update instance, whose ID is 234. Data in body
DELETE	/A/345	Delete existing instance, whose ID is 345
POST	/A/123/B	Create new instance of B related to instance of A whose ID is 123. Data for new B in HTTP body
GET	/A/234/B	Retrieve all instances of B related to instance of A whose ID is 234. As an array or list
GET	/A/123/B/234	Retrieve instance B with ID 234 related to instance A with ID 123. Useful if service needs context of A
		OR
GET	/B/234	Retrieve instance B with ID 234, regardless of A. Useful when context of A is unnecessary



POST	/actors/123/movies	Create new movie for actor 123
GET	/actors/234/movies	Get all movies for actor 234

Web service can be parameterized with variables in the path

```
@GetMapping("/hello/{name}")
public String sayHelloToName
    (@PathVariable("name") String nameVariable) {
  return "Hello " + nameVariable;
}
```

```
@GetMapping("/hello/{name}")
public String sayHelloToName
    (@RequestParam("message") String msg,
     @PathVariable("name") String nameVariable) {
  return "Hello " + nameVariable + ", message = " + msg;
}
```

The @RequestParam is used to extract query parameters while @PathVariable is used to extract data right from the URI.
Fill in the blanks below to constraint role to string values FACULTY and STUDENT. const userSchema = mongoose.Schema ({role:{[type]:[String], [enum]:{'STUDENT', 'FACULTY'}}}). **Fill in the blanks below to give employees an initial salary of 50000:** const employeesSchema = mongoose.Schema ({salary:{[type]:Number, [default]:[50000]}})
returns all documents from the employees collection with only the salary field: db.employees.find ({}, {salary:1, _id:0})/ **retrieves the employee whose first name is Bob and role is engineer:** db.employees.find ({firstName:'Bob', role:'ENGINEER'}). First Normal: (key, **rd, rd**). Second Normal: (**key**, key, **rd**). Third Normal: (key, key, **rd, rd**). [One(x, y), Many(y,..)]; Section(sid, professor, year_offered, cid) Course(cid, title)->Which functional dependency is introduced by the constraint :->{cid, year_offered} ->q professor// Section(sid, professor, year_offered, cid), Course(cid, title)->"A professor can only teach one course"->professor->cid; "Courses are taught at most once a year"->{cid, year_offered}->sid