CSCC09F
Programming on the Web

Mongo DB

A document-oriented Database,

mongoose for Node.js, DB operations

What's Different in MongoDB?

- You should already know about traditional relational DB's (C43, D43?)

RDBMS		MongoDB
Database	⇒	Database
Table, View	⇒	Collection
Row	⇒	Document (JSON, BSON)
Column	⇒	Field
Index	⇒	Index
Join	⇒	Embedded Document
Foreign Key	⇒	Reference
Partition	⇒	Shard

What's Different in MongoDB?

- Query results returned in JavaScript-friendly format

RDBMS		MongoDB
Database	→	Database
Table, View	→	Collection
Row	→	Document (
Column	→	Field
Index	→	Index
Join	→	Embedded
Foreign Key	→	Reference
Partition	→	Shard

```
> db.user.findOne({age:21})
{
    "_id" : ObjectId("5327f8ea17…"),
     "first" : "Jeff",
     "last" : "Lee",
     "age" : 21,
    "interests" : [
          "photography",
          "gaming" ]
    "favorites": {
          "color": "red",
          "sport": "tennis"}
}
```

CRUD (Mongo native API)

- Create
  - `db.collection.insert( <document> )`
  - `db.collection.save( <document> )`
  - `db.collection.update( <query>, <update>, { upsert: true } )`
- Read
  - `db.collection.find( <query>, <projection> )`
  - `db.collection.findOne( <query>, <projection> )`
- Update
  - `db.collection.update( <query>, <update>, <options> )`
- Delete
  - `db.collection.remove( <query>, <justOne> )`

CRUD example

```
>db.user.insert({
    first: "Jeff",
    last : "Lee",
    age: 21
})
```

```
> db.user.find ()
{
  "_id" : ObjectId("51…"),
  "first" : "Jeff",
  "last" : "Lee",
  "age" : 21
}
```

```
> db.user.update(
   {"_id": ObjectId("51…")},
    {
       $set: {
          age: 22,
          salary: 77000}
    }
)
```

```
> db.user.remove({
    "first": /^J/
})
```

Mongo Features

- Document-Oriented storage
- Full Index Support
- Replication + High Availability
- Auto-Sharding
- Flexible Querying including Geospatial
- Map/Reduce for processing large datasets

Agile

Scalable

Mongo: a Document DB

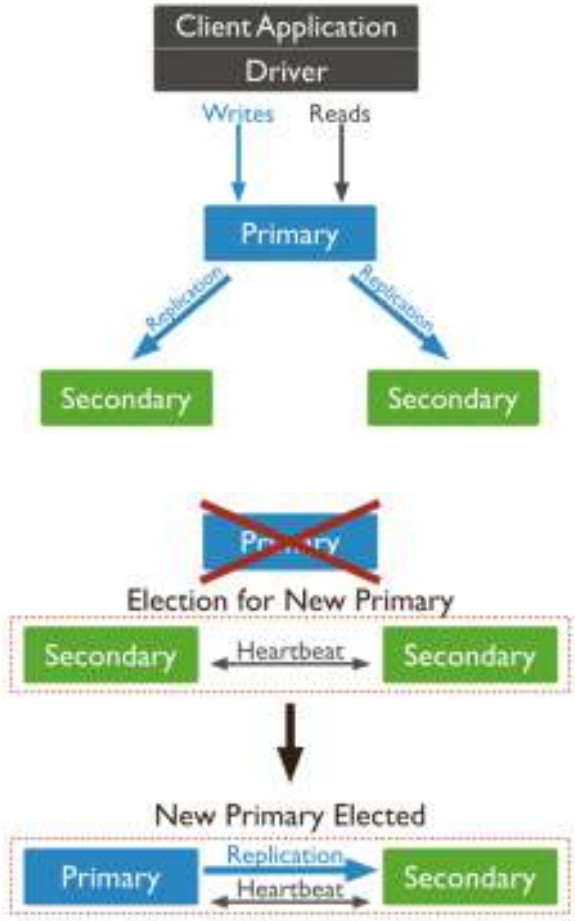
- Mongo is a document-oriented datastore, as compared with a set of related tables in a relational DB
  - Document-oriented store maps more easily to domain models, e.g. User, Dish
- Mongo represents documents as JSON, making it very easy to work with for Web apps, especially when the app server-side is implemented in JavaScript (e.g. Node.js)
  - for efficiency DB stores a binary form of JSON named BSON
- Unlike a traditional relational DB, Mongo does not rely on a rigid table structure with columns and types (although can define schemas with types, as in Mongoose)

Mongo: a Document DB

- A Mongo DB is “schemaless”
  - although we will define schemas in Mongoose which get evaluated every time you restart your Node.js server
- This means that like JavaScript’s collections, the models in a MongoDB collection can have different field sets!
- This is advantageous if you need to change the structure of your data, as in an agile environment
  - Changing structure of relational database tables is a delicate operation – get it wrong and your code is broken
  - In a Mongo environment, change schema, e.g. add new field, restart. Any new models created will have the new structure, old models will have null value for new field (beware!)

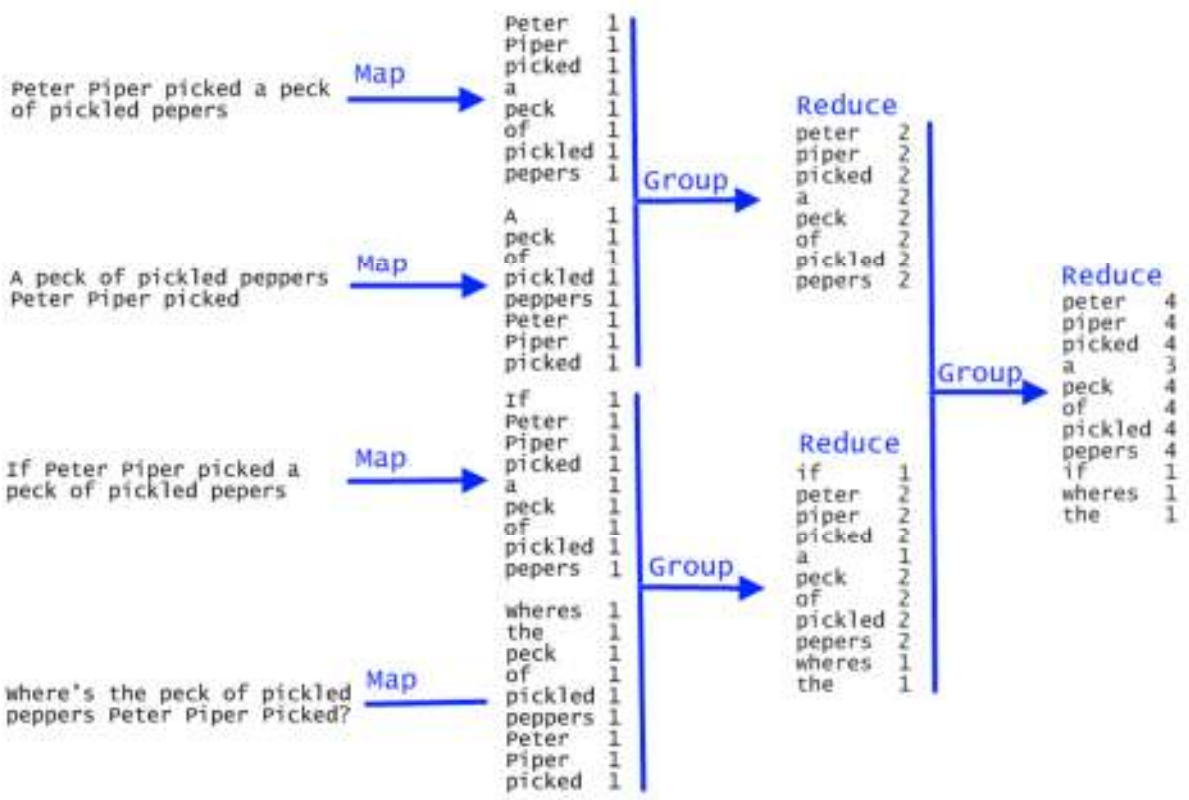
Mongo: performance features

- Any document field can be designated as an index, for faster index-based querying
- Mongo supports DB replication, so that if your primary datastore fails, a replica datastore takes over
- Mongo supports sharding for load balancing, with set of documents in a DB distributed across multiple servers, based on user-selected shard-key, e.g. “city” field



Mongo: map-reduce

- map-reduce is supported for processing large datasets using distributed map workers and reducers



Is Mongo Mainstream?

- Mongo isn't just an isolated open-source project used for tinkering
- As of Oct 2014, is ranked as the 5th most used DB overall and 1st place among NoSQL DB's!

Rank	Last Month	DBMS
1.	1.	Oracle
2.	2.	MySQL
3.	3.	Microsoft SQL Server
4.	4.	PostgreSQL
5.	5.	MongoDB
6.	6.	DB2
7.	7.	Microsoft Access
8.	8.	SQLite
9.	↑ 10.	Cassandra
10.	↓ 9.	Sybase ASE

Mongo with Node.js: Mongoose

- Mongoose module provides object modeling for Node.js
  - similar idea as Object-Relational-Mapping (ORM) for traditional relational databases
  - goal is to reduce the "friction" or "gap" between the objects/models used in the app programming language and the data stored in the database
- Mongoose translates DB data to JavaScript objects and vice versa
- Advantage is that as a developer you can think of the database as storing the same models used in your client-side code (Backbone Models)

Mongo with Node.js: Mongoose

- Mongoose module provides object modeling for Node.js

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');

var Dog = mongoose.model('Dog',{name: String});

var pup = new Dog({ name: 'Boof' });

pup.save(function (err) {
  if (err) // ...
  console.log('Woof!');
});
```

Mongo with Node.js: Mongoose

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');

var Dog = mongoose.model('Dog',{name: String});

var pup = new Dog({ name: 'Boof' });

pup.save(function (err) {
  if (err) // ...
  console.log('Woof!');
});
```

- In **`mongoose.model('Dog',{…})`**, **`Dog`** is compiled to a constructor function that is used to create instances of the model, which in turn are documents persisted by Mongo. **`pup`** is one such instance
- See save() example below for Asst 2 related model

Mongoose

- When using an authenticated database need to specify extra details at connect time (note the ":port" part can be dropped when using the standard port, as on mathlab):

```
var uri = 'mongodb://username:password
          @hostname:port/database_name;
mongoose.connect(uri);
```

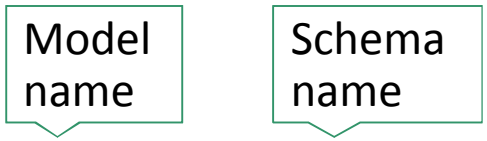
- Should specify username, password, database_name in your **`options.js`** module

Mongoose Schemas

- A mongoose schema is associated with the model for a collection – same idea as a collection model in Backbone

```
// Schemas
var DishSchema = new mongoose.Schema({
    name: { type: String, required: true },
    venue: { type: String, required: true },
    // ADD CODE for other Dish attributes
});
```

- To be able to modify Dish, we need:




```
var Dish = mongoose.model('Dish', DishSchema);
```

Queries

- To retrieve all Dish instances:

```
Dish.find({}, function(error, dishes){
    if (!error)
        res.send(dishes);
});
```

- You can omit the `{}` parameter, as a syntactic abbreviation
- In general, the first parameter of `.find()` can be a JSON object such as:

```
{ username: 'wen' }
```

- which behaves like SQL command:

```
WHERE username="wen"
```

Queries

- A special case of find is used when we have a model's id:

```
Dish.findById(req.params.id,
    function(error, data){
                res.json(data);
});
```

- Another special case of find is used when you want just a single result:

```
Dish.findOne(req.params.id,
    function(error, data){
                res.json(data);
});
```

More Queries

- Mongo queries are quite flexible:
  - by field value, e.g. dish venue
    - `db.dish.find({venue: "Shamrock Burgers"})`
  - by ranges, e.g. price value in range 10-30
    - `db.dish.find({price: {$gt:9.99, $lt:30.01} })`
  - by regular expression, comparable to SQL's LIKE
    - `db.dish.find({info: /takeout/})`
- Result of a query can be one or more documents (models) or fields of those documents, e.g. could return just the venue (and id) field from all dishes matching query:
  - `db.dish.find({info: /takeout/}, { venue: 1} )`

Saving New Model

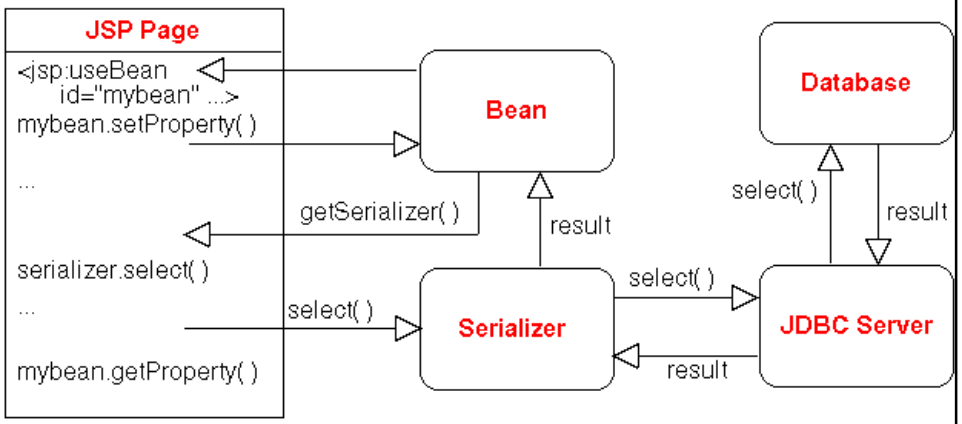
```
// create new dish instance
var dish = new Dish(… attributes …);

// save to the database, with callback
// to return result and handle errors
dish.save(function(error, dish){
    if(!error){
        res.send(dish);  // return model
    else { ... handle errors }
});
```

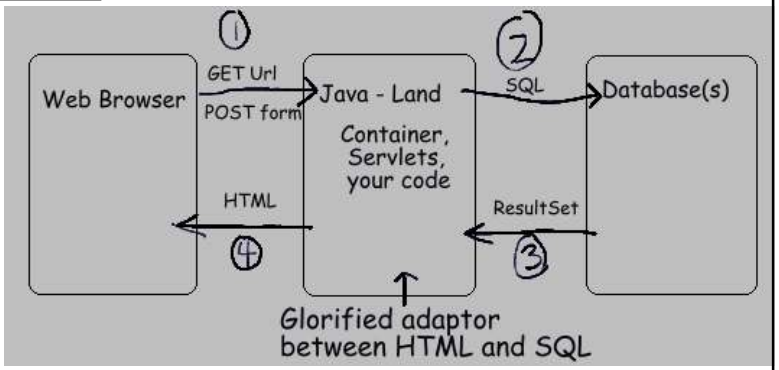
Updating Existing Model

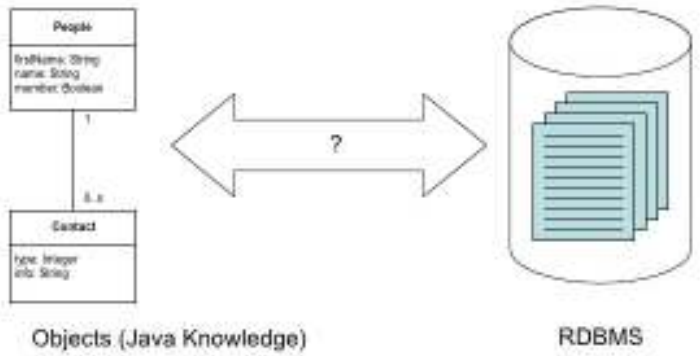
```
Dish.findById(req.params.id,
  function(findErr, dish){
    if(!findErr){
      // update dish attributes from req.body
      dish.save(function(saveErr, dish) {
        if (!saveErr) {
          res.send(dish);  // return model
        else { ... handle error }
    else { ... handle error }
});
```

Classical Apps

- When implementing an app-server using a traditional OO-language such as Java we face a mismatch between client-side JavaScript models and server-side Java objects
- Typically use a server Bean to map client model to server object
- Beans can be populated from forms

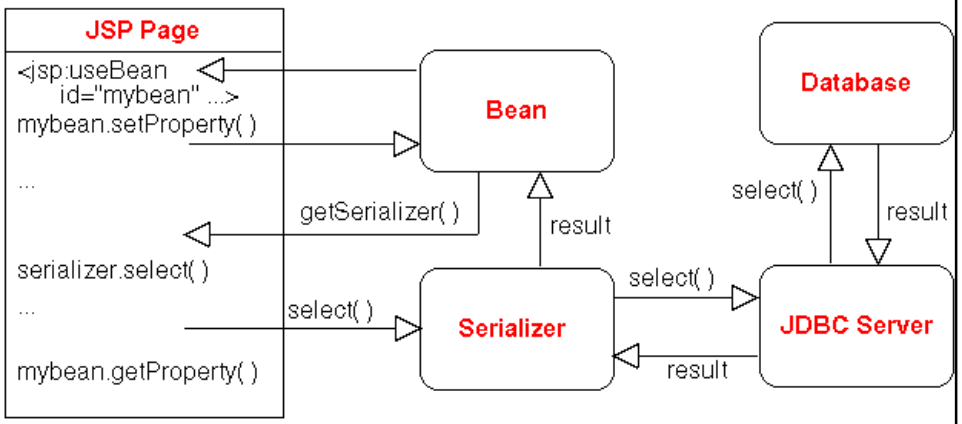


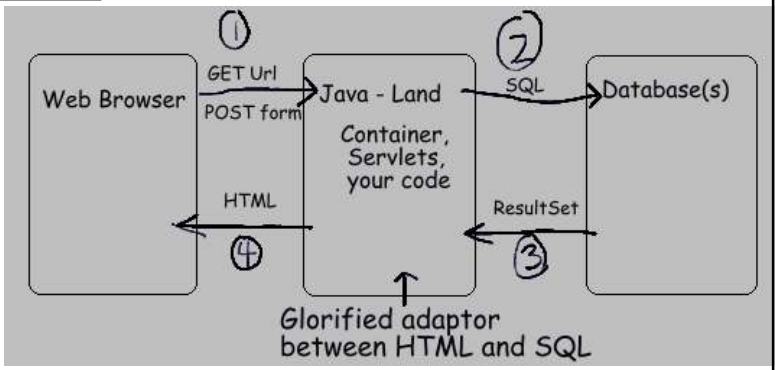


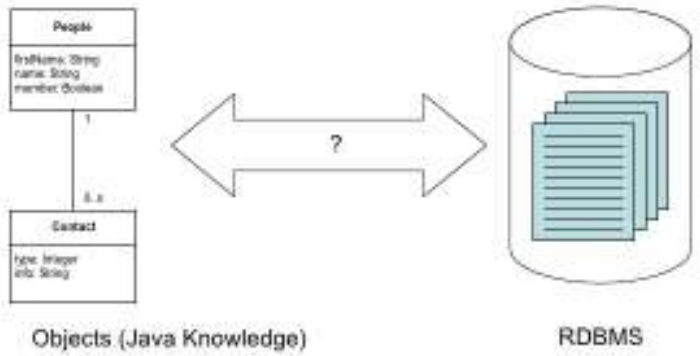


Classical Apps

- When implementing app-server using a language like Java with relational DB we face "object-relational" mismatch
- In the language, domain models represented as objects, in the DB they are represented as rows in tables
- Controller typically has result-set iterator to create beans from DB results

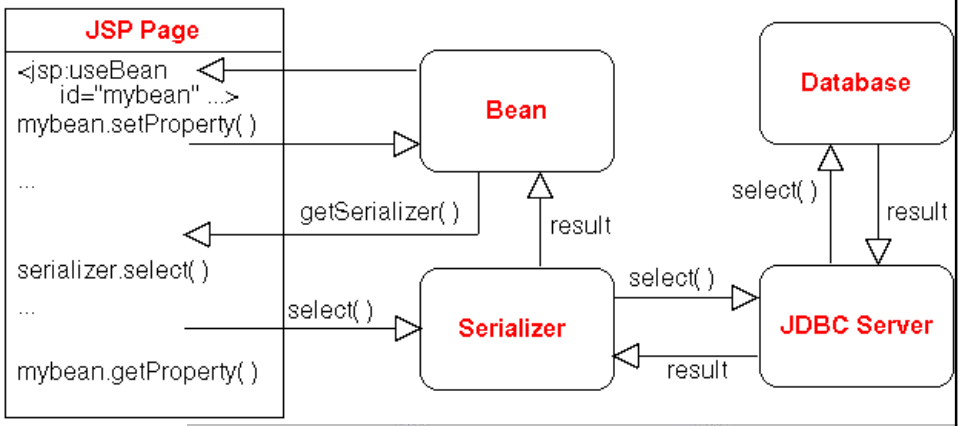


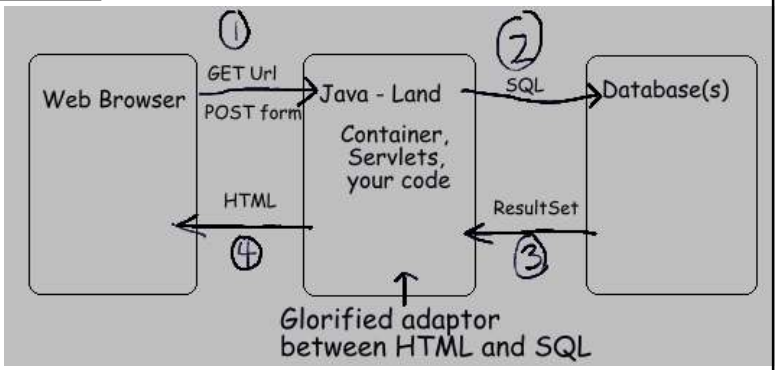


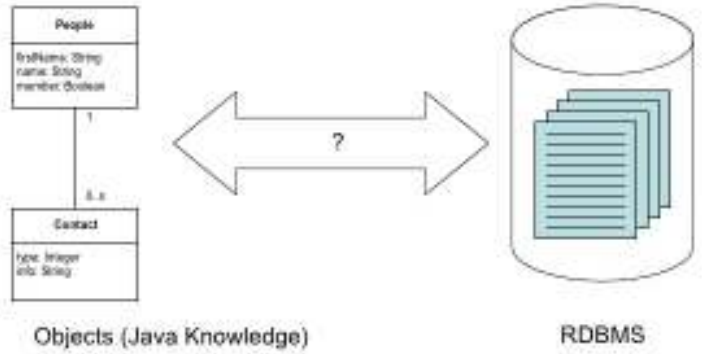


Classical Apps

- When implementing app-server using a language like Java need to map bean results back to client-side objects
- Typically have JSP "view" that renders beans in JSON format
- To get JSON to render on client, use Java tag library to iterate over JSON structure







Full-Stack JavaScript

- Full-Stack JavaScript refers to use of JavaScript end-to-end in building an app …
- from the client-side running with a framework like backbone …
- to the server side, running a JavaScript server like Node.js …
- to a database serving up JavaScript data (JSON)