



# BACKEND DATA BASE



# Persistent Data

- Typically our App is going to need persistent data associated with an individual user.
- Client side
  - *Cookies*
  - *LocalStorage*
- Server side - Relational database
  - *Often accessed via SQL*
  - *SQLite, Postgres, Oracle*
  - *Designed to work with and maintain relations*
  - *Well defined tables*
  - *Objects may be spread over multiple tables*

# Persistent Data

- Server side – Document stores
  - *Often stored as XML, JSON*
  - *Each object is a document*
  - *Every object has a unique key*
  - *Structured, but more flexible on attributes.*
    - Loose schema, dynamic shaping
    - Promise of updating of schema without migration
  - *Built in mechanisms to distribute across multiple machines*
  - *Relatively fast writes*
  - *Mongo DB is common, but there are many more.*

# Persistent Data

- Server side – Document stores
  - *Operations on a single document are atomic, but we have race conditions.*
  - *Originally did not have transactions. There are now multi document transactions, but not seamless and takes work to get them right.*
  - *Relationships are not enforced by the DB. The app must enforce them instead.*
  - *Lack of strong schema means maintainer has the responsibility to keep the data clean.*
  - *No SQL. Use bits of JS.*

# DB with Node/Express

- An express app can use any database that is supported by Node
  - *PostgreSQL, MySQL, SQLite, MongoDB, Redis (In-memory key-value)*
  - *And Others [with Node drivers](#)*
- The database can be associated with the app host or can be located elsewhere.

# Selection

- We will make a certain set of choices that will work for us, but should not be assumed to be correct in all situations.
- MongoDB – A pretty popular document store DB.
- Mongoose –object model
- MongoDB Atlas – Cloud based service for a MongoDB.

# Object Modeling

- Even though you can free style with your accesses to a data base it is convenient to add some structure that allows for validation and business logic.
- Mongoose is schema based and it provides:
  - *Type casting*
  - *Validation*
  - *Query Building*
  - *Business logic hooks*

# Schema

- Description of properties for a named collection
- We can nest arrays and objects.
- Type has special meaning – Safer not to use as a property in the schema ([see](#))

- Example:

```
const costumeSchema = new mongoose.Schema({
  costumetype: String,
  size: String,
  cost: Number,
  onSale: Boolean
  locations: [{address: String, quantity: Number}]
  description: { color: String, material: String}
});
```

```
const userSchema = new mongoose.Schema({
  name: String,
  years: Number
});
```



# Schema

- The types we can specify are
  - *String*
  - *Number*
  - *Date*
    - Not part of change tracking, so save after an update requires additional work.
  - *Boolean*
  - *ObjectId*
  - *Array*
    - Automatically initialize to an empty array instead of being undefined.
  - *And others*

# Model – id

- Each document has a unique id. This will be determined when the instance has been saved
  - *We can get the value by using the property `_id`*
  - *We can use it as part of a query*

# Schema Options

- We can specify additional information about the property as options. ([see](#))

```
const userSchema = new mongoose.Schema({
  name: {
    type:String,
    lowercase: true
  }
  age: {
    type:Number,
    set: v => Math.floor(v),
    default: 10
  }
});
```

```
const person = new User()
person.age = 5.34
console.log(person.age)
```

# Schema Function

- We can add functions to the schemas (just like we can add functions to a class.)
- Must be done before compilation.
- Don't use =>. Arrow notation will not bind these

```
userSchema.methods.report =  
  function {  
    age ? `${this.name} is ${this.age} years old` :  
          `${this.name} has undetermined age`  
  }
```

# Model

- We can compile the schema into a class where we associate the store with the schema

```
const Model = mongoose.model('Users', userSchema)
```

- We can then create instances of the class which will allow us to communicate with an associated document in the document store.

```
const fred = new User({name: 'Fred', years: 25})
```

# Model Constructor options

- There are some options that can be handed to the constructor. ([see](#))

```
const user = new User({index: true});
```

# Model - Save

- The instance is local. If we want to save it to the database we invoke the save method and provide a call back.
- ```
fred.save( (err, user) => {  
  if (err) return console.error(err);  
  console.log(`saved user ${user.name}`)  
})
```

# Model – Find all

- We use the class to search the document store for all instances.

```
User.find( (err, results) => {  
  if (err) return console.error(err);  
  // Iterate through the results  
  for user of results {  
    console.log(`found user ${user.name}`)  
  }  
})
```



# Model – Find some

- We can use regular expressions and mathematical operators to restrict the results presented to the call back.

```
User.find( { name:/^f/, age {$lt:'65' } ,  
  (err, results) => {  
    if (err) return console.error(err);  
    // Iterate through the results  
    for user of results {  
      console.log(`found user ${user.name}`)
```

# Model – Find one

- We can chain filters

```
const query = User.findOne( { name: 'johny' })  
  
query.find({age: '$gt 20'})  
  
const found = await query.exec();  
  
query.exec(  
  (err, result) => {  
    if (err) return console.error(err);  
    console.log(`found user ${result.name}`)  
  })
```

# Model – Delete

- The query part of things is the same except we delete the documents from the data base.

```
User.deleteOne( { name:/^f/, age {$lt:'65' } ,  
  (err) => {  
    if (err) return console.error(err);  
  }  
)
```

# Model – Update

- Usually, `save()` is the right way to update a document.
- Guarantees that we have
  - *Casting* – convert the data to what the schema specifies
  - *Validation* – Apply any validation methods
  - *Middleware* - Hooks that are passed control on asynchronous calls.

# Validators

- There are some validators that you get for free
- String
  - *enum, match, minLength, maxLength*
- Number
  - *min, max*
- All
  - *Required*
- Unique is not a validator, it's used for creating indexes.

# Validators

```
const userSchema = new
mongoose.Schema({
  name: {
    type:String,
    lowercase: true
    enum:['Fred', 'Barney']
    required: [true, l

  }
  age: {
    type:Number,
    set: v => Math.floor(v),
    default: 10
    min: [1, 'too small']
    max:25
  }
});
```

# Validators

- Can always build a validator function that returns true/false.

```
age: {  
  type: Number,  
  set: v => Math.floor(v),  
  default: 10  
  validate: {  
    validator: function(v){  
      return v%2  
    },  
    message: "Number must be even"  
  }  
}
```

# More on DB design

- To become proficient in DB design takes theoretical knowledge and experience.
- Know what the goals are for each of the kinds of DB
- Know the performance (memory, speed)
- The following links have references you can read to build up your understanding
  - [Schema](#)
  - [Mongo Schema Types](#)