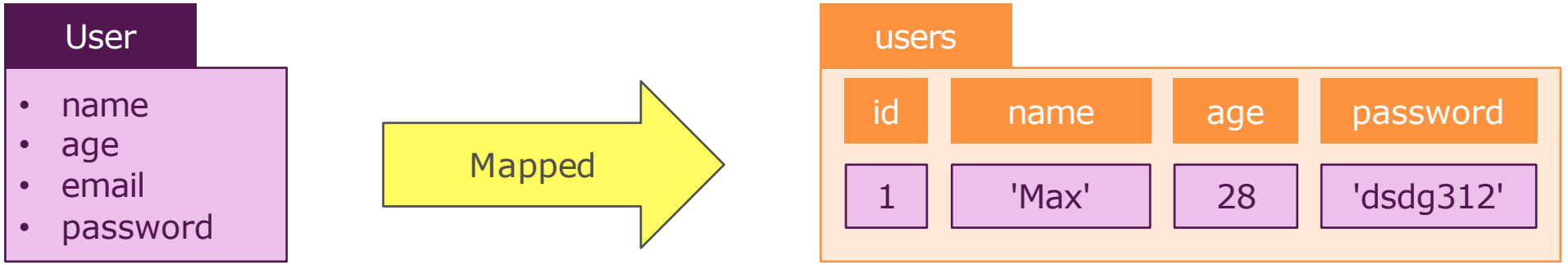
Mongoose

What is Mongoose?

- Mongoose ODM is an object document modeling package for Node that essentially works like an ORM (like Hibernate in Java).
- Mongoose allows us to have access to the MongoDB commands for CRUD simply and easily.




```
db.collection('users').insertO    ame: 'Max', age: 28, password:
                          'ds      ' })
```

```
const user = User.create({ name: 'Max', age: 28, password: 'dsdg312' })
```

Getting Started with Mongoose

```
// install mongoose package
$ npm install mongoose

// import the library in your application
const mongoose = require('mongoose');

// connect to a MongoDB database
mongoose.connect('mongodb://localhost:27017/shopping')
    .then(() => {
        …
    }).catch(err => console.error(err));
```

Defining a Model

- Mongoose Schema is what we use to define attributes for our documents (structure of the document).

- Before we can handle CRUD operations, we will need a mongoose Model. These models are constructors that we define. They represent documents which can be saved and retrieved from our database.

- Mongoose Methods can also be defined on a mongoose schema.

Working with Mongoose

1. **Create schema**

```
const mySchema = new mongoose.Schema({ name: String });
```

2. **Convert schema to model** (collection)

```
const Collection = mongoose.model('Collection', mySchema);
```

3. **Every instance of the model represents a document**

```
const doc = new Collection({ name: 'Josh' });
```

Mongoose - Model Example & Create

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const productSchema = new Schema({
    title: {
        type: String,
        required: true
    },
    price: {
        type: Number,
        required: true,
    },
    description: String,
    imageUrl: String
});

//model name will be used to turn into collection name
//'Product' -> lower case 'product' + 's'
module.exports = mongoose.model('Product', productSchema);
```


https://mongoosejs.com/docs/guide.html

Mongoose – CRUD

```
const Product = require('../models/product');

exports.getProducts = async(req, res, next) =>
{
    const prods = await Product.find();
    res.status(200).json(prods);
}

exports.getProductById = async(req, res, next)
=> {
    const prod = await
Product.findById(req.params.prodId);
    res.status(200).json(prod);
}

exports.save = async(req, res, next) => {
    const savedProd = await new
Product(req.body).save();
    res.status(201).json(savedProd);
}
```

```
exports.update = async(req, res, next) => {
    const result = await Product
        .updateOne({ _id: new
ObjectId(req.params.prodId) }, req.body);
    res.status(200).json(result);
}

exports.deleteById = async(req, res, next) => {
    await Product.deleteOne({ _id:
req.params.prodId });
    res.status(200).end();
}
```

Instance Methods

- Instances of Models are documents. Documents have many of their own built-in instance methods. We may also define our own custom document instance methods too.

```
productSchema.methods.saveWithCheckingTitle = function() {
    if (this.title.length < 4) {
        throw new Error('Product length must be greater than 4');
    } else {
        return this.save();
    }
}
```

Statics

- You can also add static functions to your model. There are 2 equivalent ways to add a static:
  - Add a function property to schema.statics
  - Call the Schema#static() function

```
productSchema.statics.filterByPrice = function(price) {
    return this.find().where('price').gt(price);
};
```

Run a Function Before Saving

- We also want to set `created_at` and `updated_at` fields to know when the record was created or updated. We can use the Schema `pre` method to have operations happen before an object is saved.

```
// on every save, add the date
productSchema.pre(['save', 'saveWithCheckingTitle'], function(next) {
    this.created = new Date();
    this.updated = new Date();
    next();
});
```

- *This is also a great place to hash passwords to be sure that we never save plaintext passwords.*

Working with Relations

```
const mongoose = require('mongoose');
const { Schema } = mongoose;

const categorySchema = new Schema({
    name: { type: String, required: true }
});

module.exports = mongoose.model('Category', categorySchema);
```

Working with Relations (Cont.)

```
const mongoose = require('mongoose');

const { Schema } = mongoose;

const productSchema = new Schema({
    title: { type: String, required: true },
    price: String,
    description: String,
    categories: [{ type: Schema.Types.ObjectId, ref: 'Category' }]
});

module.exports = mongoose.model('Product', productSchema);
```

Populating an existing document

- If we have one or many mongoose documents or even plain objects (like mapReduce output), we may populate them using the Model.populate() method available in mongoose >= 3.6. This is what document#populate() and query#populate() use to populate documents.

```
exports.getProducts = async(req, res, next) => {
    const prods = await Product.find().populate('categories');
    res.status(200).json(prods);
}

exports.getProductById = async(req, res, next) => {
    const prod = await
Product.findById(req.params.prodId).populate('categories');
    res.status(200).json(prod);
}
```

Resources

- Mongoose Resources
  - Mongoose
  - Mongoose Documentation
  - Mongoose Schemas
  - Mongoose Models
  - Mongoose Sub-documents
  - Mongoose-currency