THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

# COMPSCI 732

# SOFTENG 750

MongoDB

# Agenda



- MongoDB
  - Relational vs Document ("No-SQL") databases
  - Examples
  - MongoDB Compass (DB browser tool)
- Mongoose: MongoDB from JavaScript applications
  - Modelling schema
  - Adding / removing / modifying documents

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING


CS 732 / SE 750 – Module Four

# MongoDB

# Document databases



- Colloquially known as “No-SQL” databases
  - Because you don’t write SQL to interact with them!
- Originally designed to simplify the mapping between object-oriented business logic and the underlying data storage, without requiring ORM (Object Relational Mapping) middleware
- Individual documents (objects) are stored in various collections (similar to RDBMS tables) in a database
- Nested documents are trivial and don’t require separate collections (though sometimes you might want one)
- Relationships between different collections are easy
- Document / object properties can be indexed for increased speed / enforcing uniqueness

# MongoDB



- Stores documents in a JSON-like structure
- Can interact with the DB via the command line using what is essentially JavaScript
- MongoDB Compass – a DB browser tool
- Can be local or cloud-based
- Wide community support

# MongoDB



- Each MongoDB database can have an arbitrary number of **collections**
- Each *collection* can have any number of **documents**
- Each document can be represented as **JSON**
- Each document has a field called `_id`, which is an identifier that's guaranteed to be unique amongst all documents in its collection
- Documents may have any number of other fields

# MongoDB



- There is **no requirement by default** for documents in a collection to conform to the same *schema* (i.e. have the same fields, etc).
- For example, it is valid for a single collection to contain all of these at the same time:

```
{
  _id: ObjectId("5ea10e4846c12e2d40b69a9a")
  name: "Thomas"
  age: 42
}
```

```
{
  _id: ObjectId("5ea11151d7deff616c570d5a")
  line1: "123 Some St"
  city: "Townsville"
}
```

```
{
  _id: ObjectId("5ea11562ccc5ba5378c9d96d")
  date: "2020-04-23T04:11:45.933+00:00"
  content: "This is a note."
}
```

In practice, developers will nearly always store objects of the same type in a single collection

# Schema validation



- If we would like to enforce particular rules on inserted documents (e.g. required fields and data types of those fields, relationships between douments, etc…), we can use **schema validation**
- This can be done from the command line (https://docs.mongodb.com/manual/core/schema-validation/) or from Compass (see later slide)

# MongoDB – data types



- The following are valid datatypes for a field in a MongoDB document:
  - Commonly used: `Binary`, `Boolean`, `Date`, `Decimal128`, `Double`, `Int32`, `Int64`, `ObjectId`, `String`, `Timestamp`
  - Less common: `Code`, `MaxKey`, `MinKey`, `BSONRegexp`, `Symbol`
- In addition, fields can be `null`, `undefined`, or other `objects` (essentially sub-documents)

# MongoDB – downloading



- Download from: https://www.mongodb.com/
- Can create an account using Google credentials
- Free for personal use
- Can download local version (Community Server, from the Software tab) and / or create a free Atlas Cloud database
- Can also download Compass, the DB browser tool

# MongoDB - Usage



- MongoDB installations come with a *shell*
  - The mongo shell is a JavaScript REPL interface to a database
  - https://docs.mongodb.com/manual/mongo/
- We can also use the GUI – MongoDB Compass
- Finally, we can interact with the database through our own node.js code using mongoose.

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

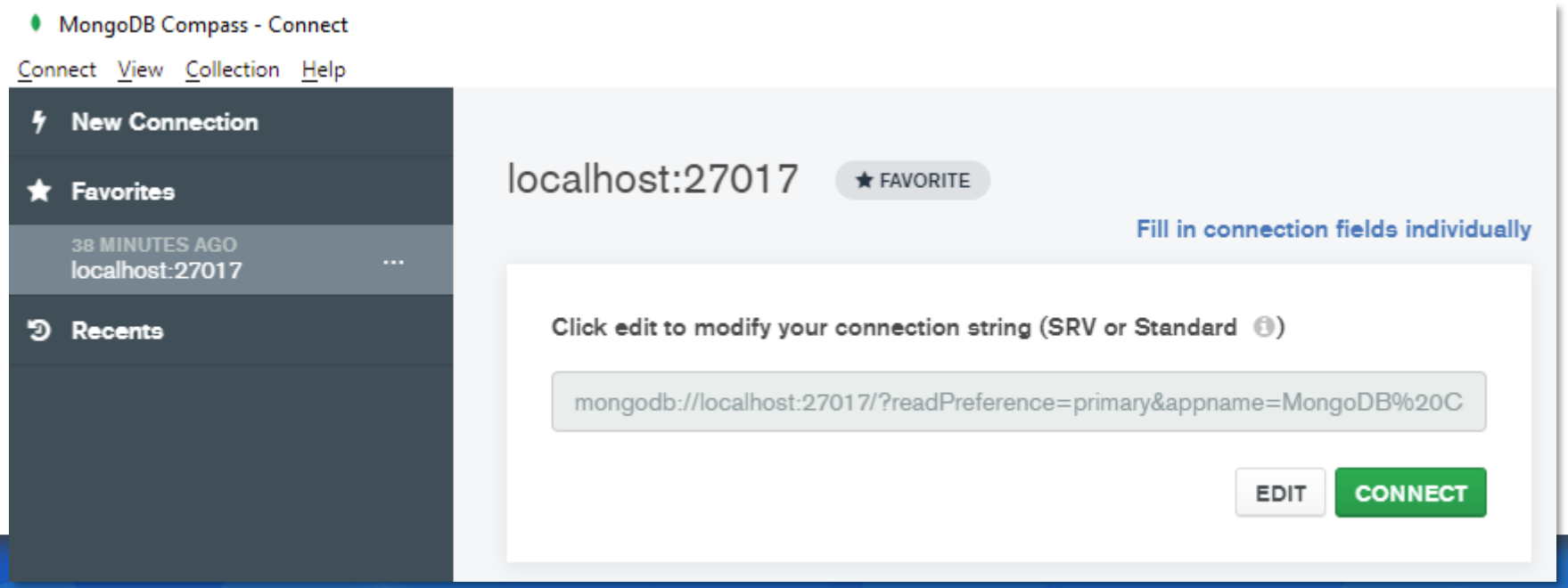
CS 732 / SE 750 – Module Four

# MongoDB Compass

# MongoDB Compass



- Once the tool is open, connect using your *connection string*
  - This is mongodb://localhost:27017/ by default for local installations
  - You can see your cloud connection string on the Atlas website



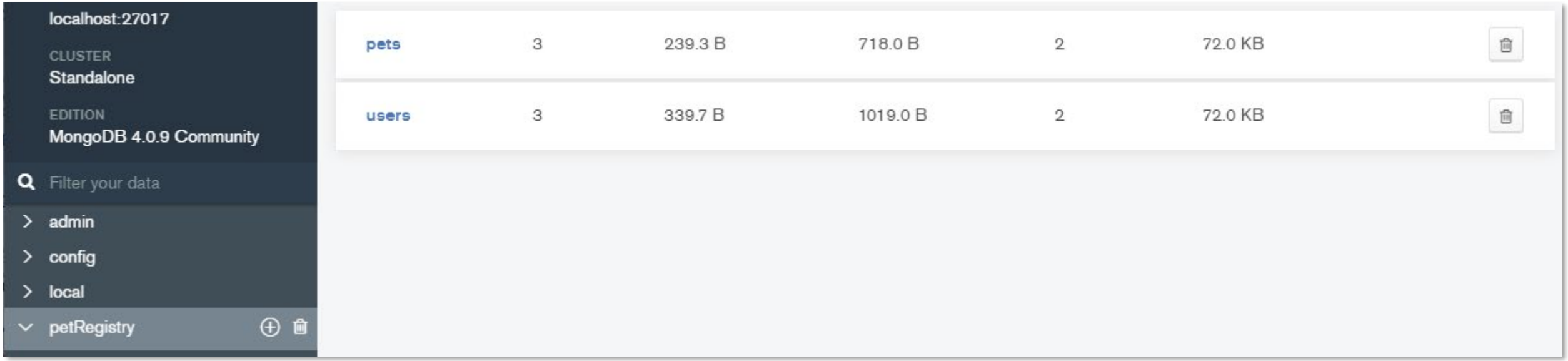
# MongoDB Compass



- Once connected we can see a list of databases on the connected server

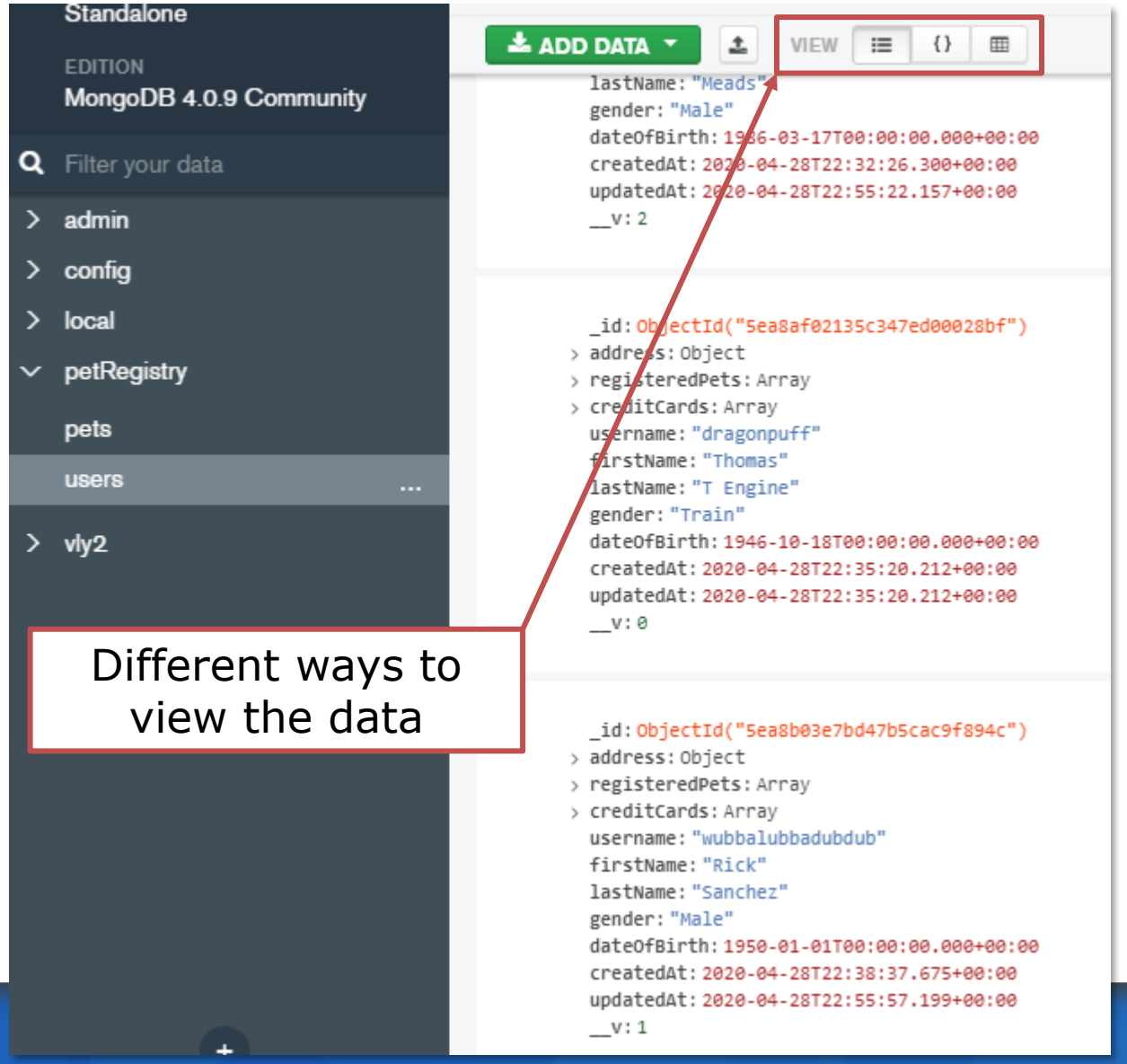
# MongoDB Compass

- Clicking on one will show us the collections in that database



# MongoDB Compass

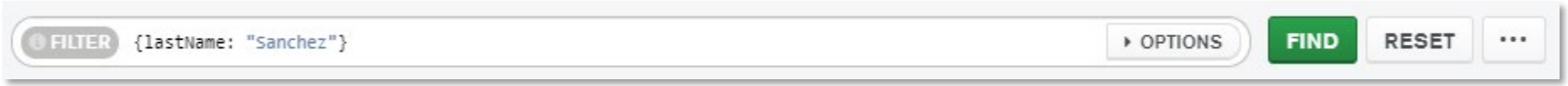
- Clicking on a collection will show us the documents in that collection
- From here we can add arbitrary documents to the collection using "Add Data"



# MongoDB Compass



- We can filter on this page too – i.e. query for documents which match certain criteria




- We can filter by match, exclusion, comparison, date, array contents, …
  - https://docs.mongodb.com/compass/current/query/filter/

# MongoDB Compass

THE UNIVERSITY OF AUCKLAND Te Whare Wānanga o Tāmaki Makaurau NEW ZEALAND | ENGINEERING

- We can edit individual documents
  - When hovering over a document, a context menu will appear showing the edit and delete buttons

# MongoDB & MongoDB Compass



- The database itself and its command-line and GUI tools have extensive documentation:
  - MongoDB
  - Mongo shell
  - Compass
- However, we will focus on interaction from our own JavaScript code…

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Four

# Mongoose

# Mongoose



- Mongoose allows us to interact with MongoDB databases via node.js (including the Express apps that often form the backends of our React webapps)
- Install as follows:

```
npm install mongoose
```

- Import:

```
import mongoose from 'mongoose';
```

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Four

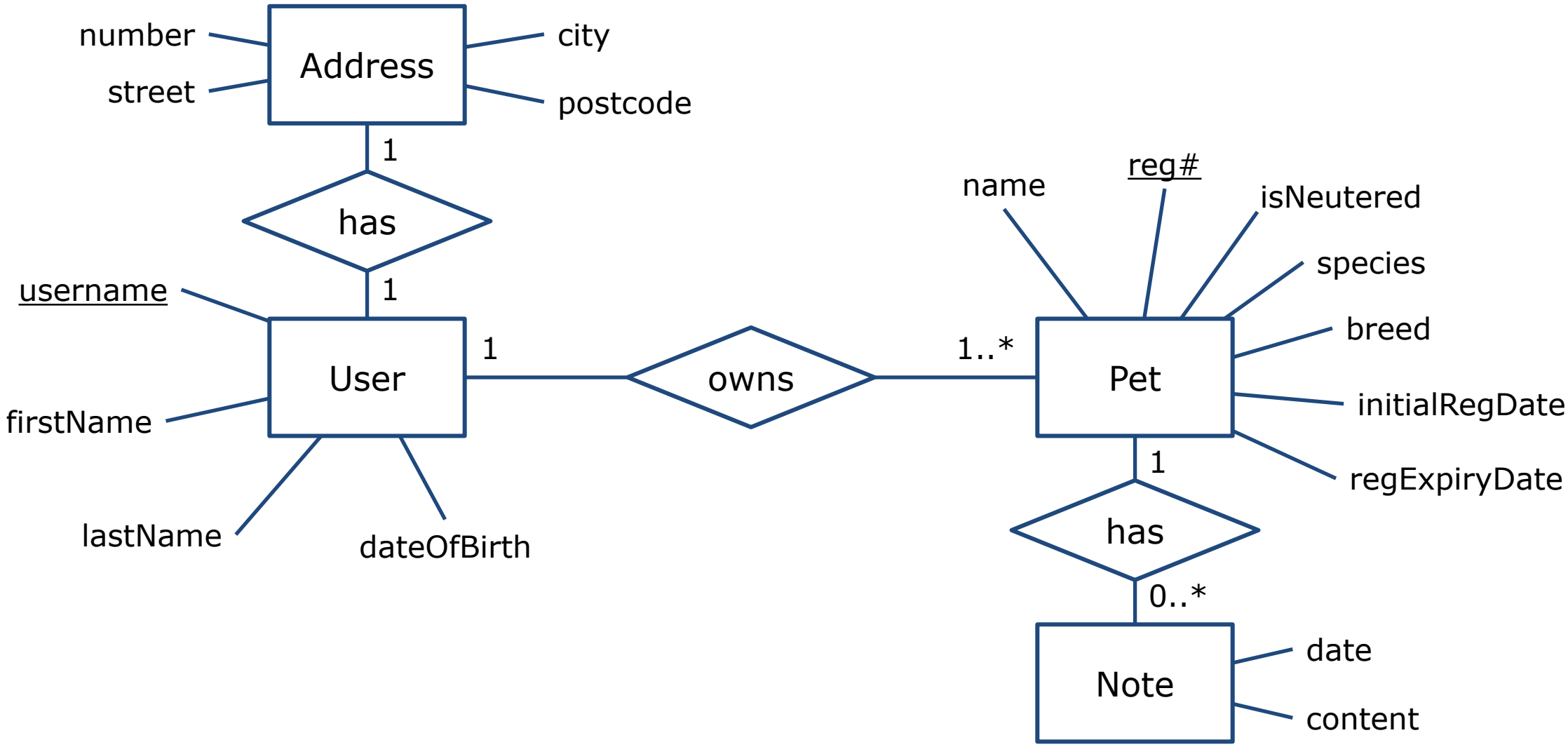
# Mongoose schema definition

# Schema definition



- Let's build a simple database for a pet registry webapp:
  - Users have a username, first & last names, date of birth, gender, and address
    - A user's address has several fields such as street name, suburb, etc.
  - A user may have any number of pets registered
  - Each pet has a registration number, a name, a breed & species, an initial registration date, a registration expiry date, and info on whether the pet has been neutered.
    - Any number of additional notes may be made about any pet. Each note has a date and content (text).

# Schema definition



# Relational model



- If we were building a traditional relational database (e.g. SQLite, MySQL, Postgres…), we might:
  - Form a relational model from our ER diagram
  - Use this to inform our SQL `CREATE TABLE` statements

# Possible relational model



User(username, firstName, lastName, dateOfBirth)

Address(number, street, city, postcode, *ownerUsername*)

Pet(regnum, name, isNeutered, species, breed, initialRegDate, expiryDate, *ownerUsername*)

Note(*petRegnum*, date, content)

# Possible relational model



User(_id, username, firstName, lastName, dateOfBirth)

Address(number, street, city, postcode, *owner_id*)

Pet(_id, regnum, name, isNeutered, species, breed, initialRegDate, expiryDate, *owner_id*)

Note(*pet_id*, date, content)

# Possible relational model



User(_id, username, firstName, lastName, dateOfBirth, address_number, address_street, address_city, address_postcode)

Pet(_id, regnum, name, isNeutered, species, breed, initialRegDate, expiryDate, *owner_id*)

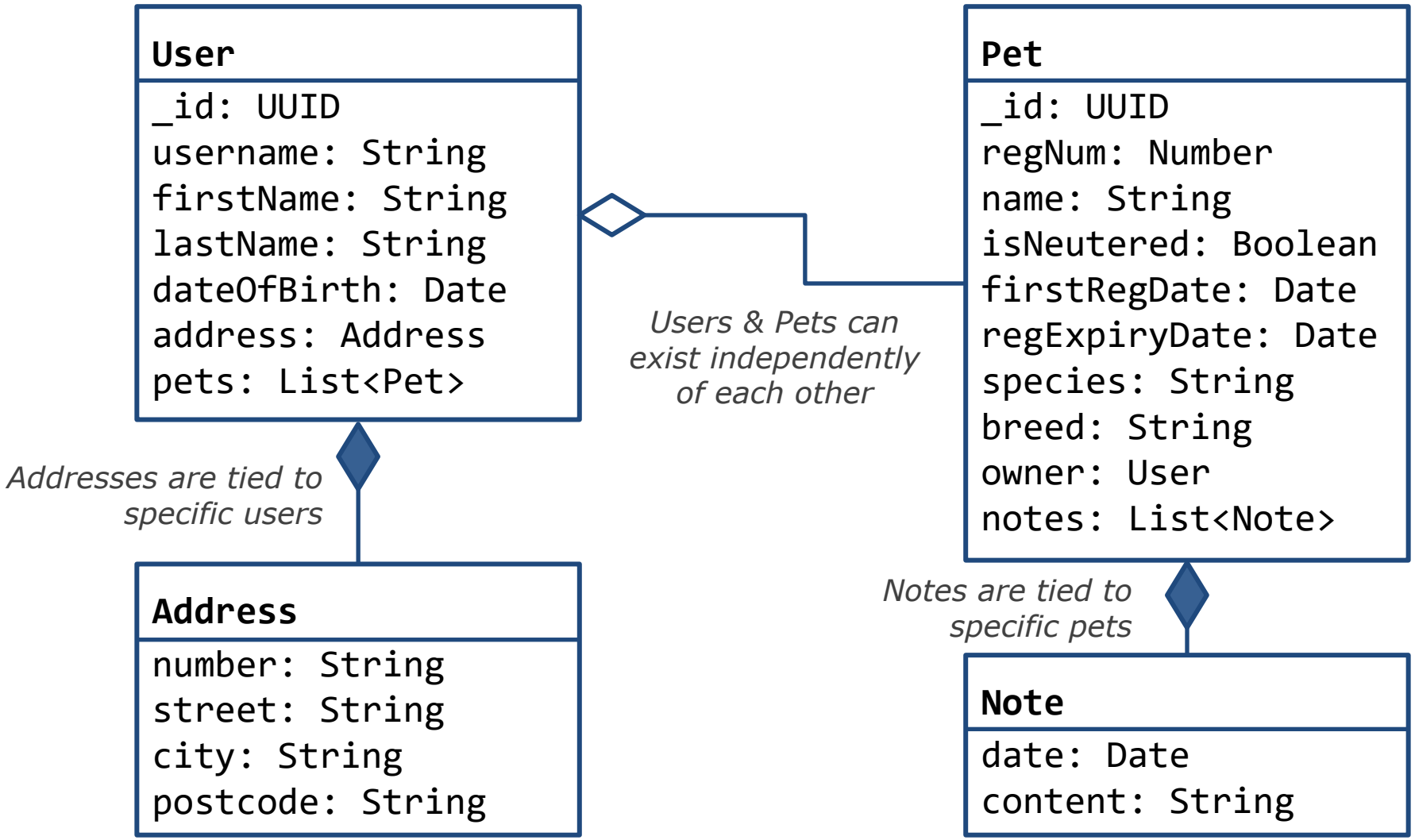
Note(*pet_id*, date, content)

# Object model



- When designing a MongoDB database schema, it might help more to think of our system in terms of **objects** and **classes** – similar to object-oriented software design
- The classes might inform the design of our schema, while individual objects may be documents in our database

# Object model



# Potential queries



- When designing a schema, it can be useful to think of the queries one might ask of the database. In this case:
  - Get all Users
  - Get all Pets
  - Get a User with a particular username
  - Get a Pet with a particular registration number
  - Get all Pets owned by a particular User
  - Get the User who owns a particular Pet
- Our **Users** and **Pets** are important entities which need to be queried independently of each other – they can be stored in their own **collections**
- **Addresses** and **Notes** can't exist independently. They can be **sub-documents**

# Schema definition in mongoose



- To define the schema for a particular type of document, we create an instance of a `Schema` object
- We can then use the `mongoose.model()` function to create a class definition for our document type, which we can then create instances of.

```
import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const userSchema = new Schema({ … });

const User = mongoose.model('User', userSchema);

const user = new User();
```

Create a new `Schema` which defines what a `User` looks like in our system

Create the User class. The name given as a string will also determine the name of the MongoDB collection used (it will use the plural form of the given word, in all lower case – e.g. users in this case).

Create an individual User instance

# Schema definition in mongoose



```
const userSchema = new Schema({

    username: { type: String, unique: true },
    firstName: String,
    lastName: String,
    gender: String,
    dateOfBirth: Date,

    address: {
        line1: String,
        line2: String,
        suburb: String,
        city: String,
        postcode: Number
    },

    creditCards: [{ lastFourDigits: String, encryptedInfo: String }],

    registeredPets: [{ type: Schema.Types.ObjectId, ref: 'Pet' }],

}, {
    timestamps: {}
});
```

Define a field which must be unique amongst all Users. This will be indexed and uniqueness will be enforced in MongoDB.

Standard fields

Address is an "embedded document". It won't be a separate collection. Each User will have an address field which in turn will have these fields.

The array `[]` syntax means that Users will have *an array of* credit cards, each one with the two fields given here

Each element in the `registeredPets` array should be an `ObjectId` referring to the `_id` of a document in the pets collection

# Schema definition in mongoose



```
const userSchema = new Schema({

    username: { type: String, unique: true },
    firstName: String,
    lastName: String,
    gender: String,
    dateOfBirth: Date,

    address: {
        line1: String,
        line2: String,
        suburb: String,
        city: String,
        postcode: Number
    },

    creditCards: [{ lastFourDigits: String, encryptedInfo: String }],

    registeredPets: [{ type: Schema.Types.ObjectId, ref: 'Pet' }],

}, {
    timestamps: {}
});
```

The second object supplied here contains additional config info.

Here, we can specify additional options such as auto-generating createdAt and updatedAt fields (as shown here), removing the auto-generated _id, or many other options.

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

# Schema definition in mongoose



- In addition to fields and sub-documents, we can also add extra:
  - Instance methods

```
const animalSchema = new Schema({ name: String, type: String });

animalSchema.methods.findSimilarTypes = function (cb) {
    return this.model('Animal').find({ type: this.type }, cb);
};

const Animal = mongoose.model('Animal', animalSchema);
const dog = new Animal({ type: 'dog' });

dog.findSimilarTypes(function (err, dogs) {
    console.log(dogs); // woof
});
```

# Schema definition in mongoose



- In addition to fields and sub-documents, we can also add extra:
  - Static methods

```
const animalSchema = new Schema({ name: String, type: String });
```

```
animalSchema.statics.findByName = function (name) {
    return this.find({ name: new RegExp(name, 'i') });
};
```

```
const Animal = mongoose.model('Animal', animalSchema);
```

```
let animals = await Animal.findByName('fido');
```

# Schema definition in mongoose



- In addition to fields and sub-documents, we can also add extra:
  - Virtuals (JavaScript link)

```
const userSchema = new Schema({firstName: String, lastName: String});

userSchema.virtual('fullName')
    .get(function () { return `${this.firstName} ${this.lastName}`; })
    .set(function (value) {
        this.firstName = value.substr(0, value.indexOf(' '));
        this.lastName = value.substr(value.indexOf(' ') + 1);
    });

const User = mongoose.model('User', userSchema);

const axl = new User({ firstName: 'Axl', lastName: 'Rose' });
console.log(axl.fullName);
axl.fullName = 'William Rose';
console.log(axl.firstName);
```

"Axl Rose"

"William"

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Four

# Mongoose schema validation

# Schema validation



- We can further extend our schema with validation logic which will be run when we try to save an object to the database
  - Objects failing the validation check will not be saved
- We can also run the validation logic ourselves in an attempt to pre-empt / fix issues before trying to save

```
const meow = new Schema({
    name: { type: String, required: true }
});
const Cat = db.model('Cat', meow);

const cat = new Cat();
cat.save(function (error) {
    assert.equal(error.errors['name'].message, 'Path `name` is required.');
});


const error = cat.validateSync();
assert.equal(error.errors['name'].message, 'Path `name` is required.');
```

The name field is required.

Trying to save a cat with no name will give us an error

We can use `validateSync()` to run the validation logic whenever we like

# Schema validation



```
const breakfastSchema = new Schema({
    eggs: {
        type: Number,
        min: [6, 'Too few eggs'],
        max: 12
    },
    bacon: {
        type: Number,
        required: [true, 'Why no bacon?']
    },
    drink: {
        type: String,
        enum: ['Coffee', 'Tea'],
        required: function () {
            return this.bacon > 3;
        }
    }
});
```

If we have eggs (it's not a required field), we must have between 6 – 12. "Too few eggs" is a custom error message that we'll get if the min validation fails.

We must have some bacon for breakfast. "Why no bacon?" is a custom error message we'll get if there's no bacon.

If we have a drink, it must be either Coffee or Tea.

We must have a drink if we have more than 3 strips of bacon.

# Schema validation



- We can define custom validator functions for arbitrary validation logic

```
const userSchema = new Schema({
    phone: {
        type: String,
        validate: {
            validator: function (v) {
                return /\d{3}-\d{3}-\d{4}/.test(v);
            },
            message: props => `${props.value} is not a valid phone number!`
        }
    }
});
```

A user's phone number must match this regular expression (XXX-XXX-XXXX, where X's are digits)

This is the custom error message we'll get if this validation fails

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Four

# Working with documents

# Modelling and object (document) creation



- Once we've modelled our schema:

```
const User = mongoose.model('User', userSchema);
```

- We can then create instances of the model class, which will correspond to MongoDB documents:

```
const anne = new User();
anne.fullName = 'Anne Hathaway';
anne.dateOfBirth = new Date('1982-11-12');

const bob = new User({
  firstName: 'Bob', lastName: 'Ross',
  dateOfBirth: new Date('1942-10-29')
});

console.log(bob._id);
```

Can get / set fields on an individual basis…

And / or supply values in the constructor as so

The _id field is auto-generated; we didn't have to define this. If we need to, we can disable this behaviour

(see: https://mongoosejs.com/docs/guide.html#_id)

# Connecting, saving & deleting



- Many of these functions are *async* and thus return promises. We can `await` them as shown here from within our own `async` functions, or use any other promise management syntax.
- To connect to a database:

```
mongoose.connect('mongodb://localhost:27017/petRegistry',
    { useNewUrlParser: true });
```

- To save a document instance:

```
const bob = new User({ … });
await bob.save();
```

- To delete an instance:

```
await User.deleteOne({ username: 'anhydrous' });
```

Deletes the first document in the users collection matching the given criteria. See "querying" for further discussion of valid criteria. There is also a `deleteMany()` function.

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Four

# Querying in mongoose

# Querying in mongoose



- We can conduct queries using the following static methods on our model classes (those generated with `mongoose.model`):
  - `findById()`: finds and returns the document with the given _id, if any
  - `findOne()`: finds and returns the first document matching the given criteria, if any
  - `find()`: finds and returns an array of all documents matching the given criteria
- All such methods are async and thus can be awaited – but **are not promises**.

*Find the user whose _id is 5ea8aede135c347ed00028be*

```
const user = await User.findById('5ea8aede135c347ed00028be');
```

*Find all users*

```
const allUsers = await User.find();
```

# Querying in mongoose



*Find the first user whose username is anhydrous*

```
const user = await User.findOne({ username: 'anhydrous' });
```

*Find all users whose age is greater than 42*

```
const users = await User.find({ age: { $gt: 42 } });
```

*Find all users whose age is greater than 42, and less than or equal to 100*

```
const users = await User.find({ age: { $gt: 42, $lte: 100 } });
```

*Find all users whose city (part of their address) is either Auckland or Hamilton*

```
const users = await User.find({ 'address.city': { $in: ['Auckland', 'Hamilton'] } });
```

*Find all users whose address line 1 contains the text "Some Street", AND who have at least one registered pet*

```
const users = await User.find({
  'address.line1': /Some Street/,
  registeredPets: { $not: { $size: 0 } }
});
```

For more query operators, see: https://docs.mongodb.com/manual/reference/operator/query/

# Populating fields referencing other collections



- When we have a field that's an `ObjectId` or an array of `ObjectIds` referencing another collection, we can populate that field with the data from the other collection using the populate method.
  - This works at query-time:

```
const user = await User
   .findOne({ username: 'anhydrous' })
   .populate('registeredPets');
```

This will make the `registeredPets` field of the returned user be an array of `Pet` objects, rather than an array of `ObjectIds`.

  - Or at any point afterwards.

```
const pets = await Pet.find({ ... });

await Pet.populate(pets, 'owner');
```

This will *convert* the `owner` field of all pets in the given array to the matching `User` object (or null if there's no matching user)

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Four

# Building a REST API for a MongoDB app

# RESTful APIs for MongoDB applications



- Using mongoose, we can easily integrate MongoDB into our node.js apps
  - Including Express web apps / services
- Using the Express knowledge we already have from Week Three, plus the mongoose knowledge from these slides, we can build an API which utilizes MongoDB
- Let's use our running Articles app as an example…

# An Articles REST API backed by MongoDB



- Defining the schema

```
const Schema = mongoose.Schema;

const articleSchema = new Schema({
    title: { type: String, required: true },
    date: Date,
    image: String,
    content: String
}, {
    timestamps: {}
});

const Article = mongoose.model('Article', articleSchema);
```

# An Articles REST API backed by MongoDB



- Create

```
async function createArticle(article) {

    const dbArticle = new Article(article);
    await dbArticle.save();
    return dbArticle;
}

router.post('/', async (req, res) => {
    const newArticle = await createArticle(req.body);

    if (newArticle) return res.status(HTTP_CREATED)
        .header('Location', `/api/articles/${newArticle._id}`)
        .json(newArticle);

    return res.sendStatus(422);
})
```

# An Articles REST API backed by MongoDB



- Retrieve One

```
async function retrieveArticle(id) {
    return await Article.findById(id);
}

router.get('/:id', async (req, res) => {
    const { id } = req.params;

    const article = await retrieveArticle(id);

    if (article) return res.json(article);
    return res.sendStatus(HTTP_NOT_FOUND);
});
```

# An Articles REST API backed by MongoDB



- Retrieve All

```
async function retrieveArticleList() {
    return await Article.find();
}

router.get('/', async (req, res) => {

    res.json(await retrieveArticleList());
});
```

# An Articles REST API backed by MongoDB



- Update

```
async function updateArticle(article) {

    const dbArticle =
        await Article.findOneAndUpdate({ _id: article._id }, article);
    return dbArticle !== undefined;
}

router.put('/:id', async (req, res) => {
    const { id } = req.params;
    const article = req.body;
    article._id = id;
    const success = await updateArticle(article);
    res.sendStatus(success ? HTTP_NO_CONTENT : HTTP_NOT_FOUND);
});
```

# An Articles REST API backed by MongoDB



- Delete

```
async function deleteArticle(id) {
    await Article.deleteOne({ _id: id });
}

router.delete('/:id', async (req, res) => {
    const { id } = req.params;
    await deleteArticle(id);
    res.sendStatus(HTTP_NO_CONTENT);
});
```

# Full stack!



- Combining this REST API, backed by a database, with our React knowledge… we have now mastered the MERN stack!

# References



- All frameworks & libraries covered in these slides are far more comprehensive than we can cover in one lecture:
- All have excellent documentation, which is a great starting point for further reading:
  - MongoDB
  - Mongoose