

# Mongo Data Modelling

---

# Embed vs References

---

- The key consideration for the structure of documents is the decision to :
  - Embed objects to encapsulate relationships

**OR**

- Use object references to encapsulate relationships

# Embedded Data Models

---

- Embed related data in a single structure or document.
- These schema are generally known as “denormalized” models
- Embedded data models allow applications to store related pieces of information in the same database record.
- As a result, applications may need to issue fewer queries and updates to complete common operations.

```
{  
  _id: <ObjectId>,  
  username: "123xyz",  
  contact: {  
    phone: "123-456-7890",  
    email: "xyz@example.com"  
  },  
  access: {  
    level: 5,  
    group: "dev"  
  }  
}
```

Embedded sub-document

Embedded sub-document

# When to use Embedded Models?

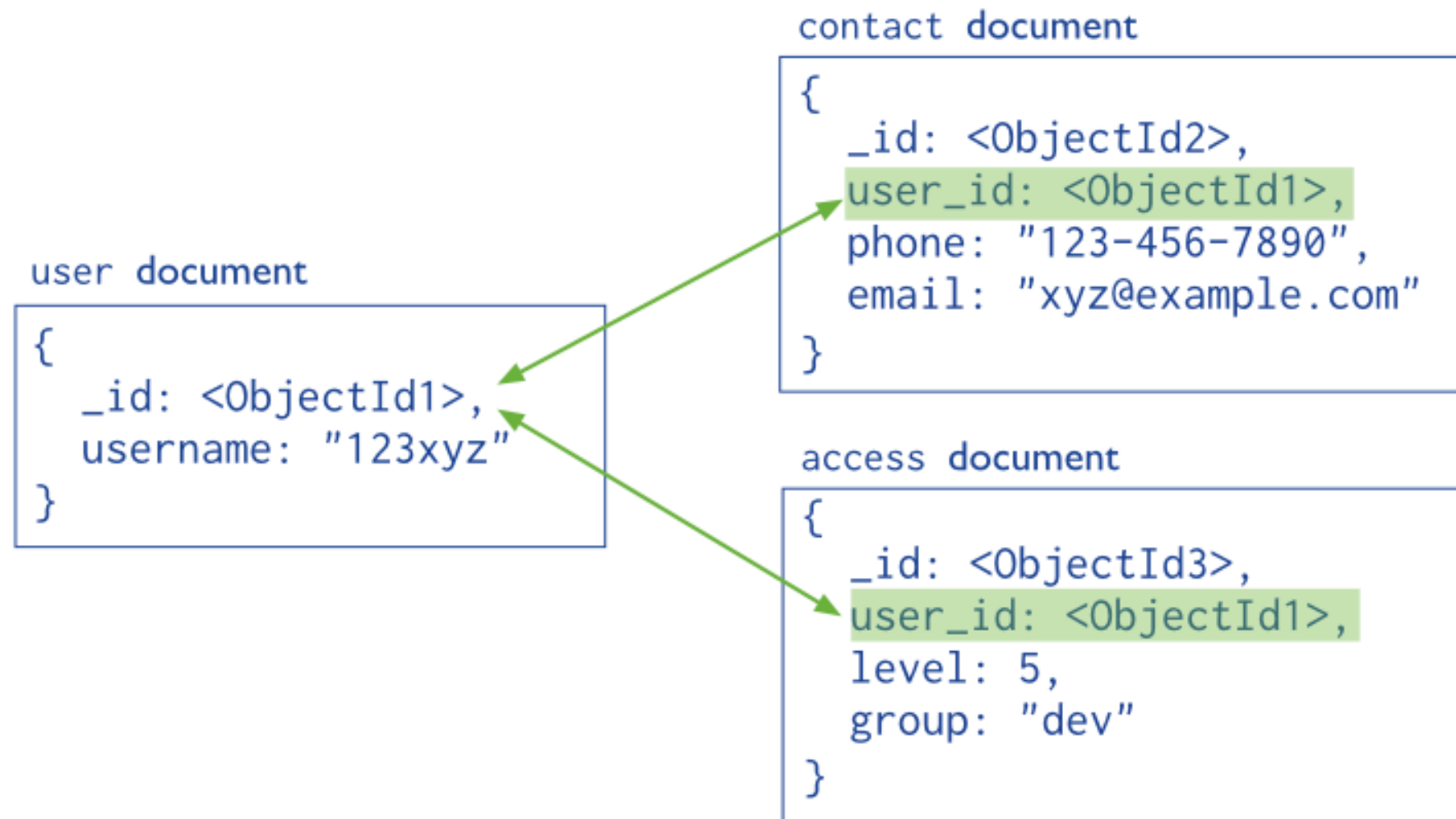
---

- The “contains” relationships between entities (One-to-One Relationship)
- Some one-to-many relationships between entities. In these relationships the “many” or child documents always appear with or are viewed in the context of the “one” or parent documents.
- Embedding provides better performance for read operations, as well as the ability to request and retrieve related data in a single database operation. Embedded data models make it possible to update related data in a single atomic write operation.
- However, embedding related data in documents may lead to situations where documents grow after creation.

# Object References -> 'Normalized' Data Model

---

- Normalized data models describe relationships using references between documents.



# When to use Normalized Data Model?

---

- When embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- To represent more complex many-to-many relationships.
- To model large hierarchical data sets.
- References provides more flexibility than embedding.  
However, client-side applications must issue follow-up queries to resolve the references. In other words, normalized data models can require more round trips to the server.

# Model: One-to-Many

---

- Stories are written by Persons

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

const personSchema = Schema({
  name: String,
  age: Number,
});

const storySchema = Schema({
  creator: { type: mongoose.Schema.Types.ObjectId, ref: 'Person' },
  title: String,
});
```

# Creating the objects

---

```
const Story = mongoose.model('Story', storySchema);
const Person = mongoose.model('Person', personSchema);

var aaron = new Person({ name: 'Aaron', age: 100 });

aaron.save().then(newPerson => {

  const story1 = new Story({
    title: 'Once upon a timex.',
    creator: newPerson._id,
  });

  return story1.save();

}).then(newStory => {

  console.log('Saved!');

});
```

```
{
  "_id" : ObjectId("57ee63b9ded76fc76f903318"),
  "name" : "Aaron",
  "age" : 100,
  "__v" : 0
}

{
  "_id" : ObjectId("57ee63b9ded76fc76f903319"),
  "title" : "Once upon a timex.",
  "creator" : ObjectId("57ee63b9ded76fc76f903318"),
  "__v" : 0
}
```

## Created Objects

- Chaining promises



# Model: One-to-Many, Many-to-One

```
const personSchema = Schema({
  name: String,
  age: Number,
  stories: [{ type: Schema.Types.ObjectId, ref: 'Story' }],
});

const storySchema = Schema({
  creator: { type: mongoose.Schema.Types.ObjectId, ref: 'Person' },
  title: String,
});
```

```
{
  "_id" : ObjectId("57ee64b9f764aac77ea465e0"),
  "name" : "Aaron",
  "age" : 100,
  "stories" : [
    ObjectId("57ee64b9f764aac77ea465e1")
  ],
  "__v" : 1
}

{
  "_id" : ObjectId("57ee64b9f764aac77ea465e1"),
  "title" : "Once upon a timex.",
  "creator" : ObjectId("57ee64b9f764aac77ea465e0"),
  "__v" : 0
}
```

Example  
Documents

```
const Story = mongoose.model('Story', storySchema);
const Person = mongoose.model('Person', personSchema);

var aaron = new Person({ name: 'Aaron', age: 100 });

aaron.save().then(newPerson => {

  const story1 = new Story({
    title: 'Once upon a timex.',
    creator: newPerson._id,
  });

  return story1.save();

}).then(newStory => {

  Person.findOne({ name: 'Aaron' }).then(person => {

    person.stories.push(newStory._id);
    return person.save();
  });

});
```

# Users & Donations

---

```
const userSchema = mongoose.Schema({  
  firstName: String,  
  lastName: String,  
  email: String,  
  password: String,  
});
```

```
const donationSchema = mongoose.Schema({  
  amount: Number,  
  method: String,  
  donor: String,  
});
```

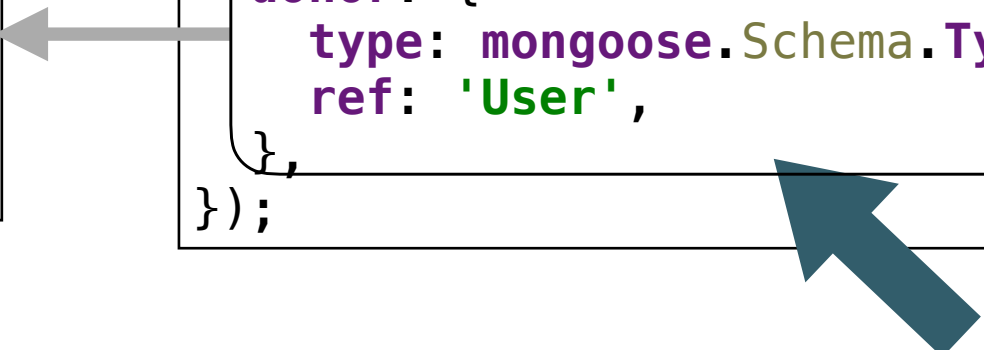
Reference  
encapsulated as  
email of donor

# Normalised Users & Donations

---

```
const userSchema = mongoose.Schema({  
  firstName: String,  
  lastName: String,  
  email: String,  
  password: String,  
});
```

```
const donationSchema = mongoose.Schema({  
  amount: Number,  
  method: String,  
  donor: {  
    type: mongoose.Schema.Types.ObjectId,  
    ref: 'User',  
  },  
});
```



Reference  
encapsulated as  
Object Reference  
to donor object

The diagram illustrates the relationship between the two schemas. A grey arrow points from the `donor` field in the `donationSchema` to the `userSchema`. A blue arrow points from the text 'Reference encapsulated as Object Reference to donor object' to the `donor` field in the `donationSchema`.

# Creating an Object Reference

---

key	value	type
▼ (1) ObjectId("57a4692...	{ 6 fields }	Object
_id	ObjectId("57a4692eb2366806c5eca610")	ObjectId
firstName	homerus	String
lastName	simpson	String
email	homer@simpson.com	String
password	secret	String
_v	0	Int32

▼ (8) ObjectId("57a6eab...	{ 5 fields }	Object
_id	ObjectId("57a6eabe7aed2bfbea3f950c")	ObjectId
donor	ObjectId("57a4692eb2366806c5eca610")	ObjectId
amount	100	Int32
method	direct	String
_v	0	Int32

ID of Homer  
user object

# Creating a Normalised Donation

---

Identify logged in user

Create new donation object

Link to logged in user id


Save the donation object

```
handler: function (request, reply) {  
  var userEmail = request.auth.credentials.loggedInUser;  
  User.findOne({ email: userEmail }).then(user => {  
    let data = request.payload;  
    const donation = new Donation(data);  
    donation.donor = user._id;  
    return donation.save();  
  }).then(newDonation => {  
    reply.redirect('/report');  
  }).catch(err => {  
    reply.redirect('/');  
  });  
},
```

Donations to Date x Eamonn

mainmac.local:4000/report

Donation Donate Report Settings Logout



Amount	Method donated	Donor
100	paypal	
50	direct	
100	paypal	
50	direct	
100	direct	
100	paypal	
1000	paypal	
100	direct	57a4692eb2366806c5eca610

Object IDs rendered in table

# Normalised documents & Population

---

- There are no joins in MongoDB but sometimes we still want references to documents in other collections.
- Population is the process of automatically replacing the specified paths in the document with document(s) from other collection(s).
- We may populate a single document, multiple documents, plain object, multiple plain objects, or all objects returned from a query.



Amount	Method donated	Donor
100	paypal	
50	direct	
100	paypal	
50	direct	
100	direct	
100	paypal	
1000	paypal	
100	direct	57a4692eb2366806c5eca610

- Extend table template to include full name of donor

```
<tbody>
  {{#each donations}}
    <tr>
      <td> {{amount}} </td>
      <td> {{method}} </td>
      <td> {{donor.firstName}} {{donor.lastName}} </td>
    </tr>
  {{/each}}
</tbody>
```

- However, default behaviour is for **find** to return only return ids in place of **donor**

```
handler: function (request, reply) {
  Donation.find({}).then(allDonations => {
    reply.view('report', {
      title: 'Donations to Date',
      donations: allDonations,
    });
  }).catch(err => {
    reply.redirect('/');
  });
},
```

```
{
  "_id" : ObjectId("57ee67fd35821864c10344a5"),
  "donor" : ObjectId("57ed30729b9a6b11bad56dc7"),
  "amount" : 1000,
  "method" : "paypal",
  "__v" : 0
}
```

# Mongoose Populate Method

---

- Populated paths are no longer set to their original `_id`, their value is replaced with the mongoose document returned from the database by performing a separate query before returning the results.

```
handler: function (request, reply) {  
  Donation.find({}).populate('donor').then(allDonations => {  
    reply.view('report', {  
      title: 'Donations to Date',  
      donations: allDonations,  
    });  
  }).catch(err => {  
    reply.redirect('/');  
  });  
},
```

```
{  
  "_id" : ObjectId("57ee67fd35821864c10344a5"),  
  "donor" : {  
    "_id" : ObjectId("57ed30729b9a6b11bad56dc7"),  
    "firstName" : "Homer",  
    "lastName" : "Simpson",  
    "email" : "homer@simpson.com",  
    "password" : "secret",  
    "__v" : 0  
  }  
  
  "amount" : 1000,  
  "method" : "paypal",  
  "__v" : 0  
}
```

[Donation](#)[Donate](#)[Report](#)[Settings](#)[Logout](#)

Amount	Method donated	Donor
1000	paypal	Homer Simpson

```
handler: function (request, reply) {  
  Donation.find({}).populate('donor').then(allDonations => {  
    reply.view('report', {  
      title: 'Donations to Date',  
      donations: allDonations,  
    });  
  }).catch(err => {  
    reply.redirect('/');  
  });  
},
```

```
<tbody>  
  {{#each donations}}  
    <tr>  
      <td> {{amount}} </td>  
      <td> {{method}} </td>  
      <td> {{donor.firstName}} {{donor.lastName}} </td>  
    </tr>  
  {{/each}}  
</tbody>
```