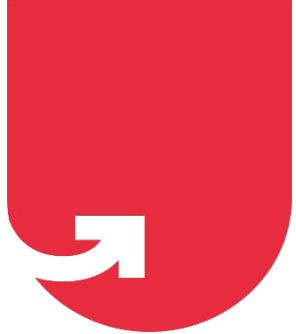
upGrad

#RahoAmbitious



Full Stack Software Development

upGrad

Course: Server-Side Development Using Node.js, Express.js and MongoDB

Lecture on: Mongoose Relationship Models

Instructor: Rocky Jagtiani

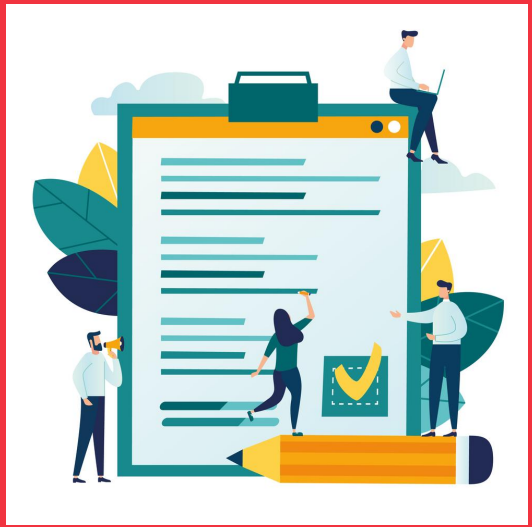
upGrad

In the previous class, we covered...

- Mongoose
- Middleware and Plugins

Today's Agenda

- Mongoose Relationship Models



upGrad

Mongoose Relationship Models

Mongoose Relationship Model

upGrad

- Relationship in NoSQL database, I mean MongoDB, is not the same as traditional SQL database (like Oracle or MySQL).
- We as MERN stack developers have to implement the relationship between various schemas by defining an association between them. Just for example in SQL databases we build this association by defining a foreign key - primary key relationship. For example, a student table has departmentId as a foreign key to DeptId of Department Table. Hence, Student records can only have departmentId from the set {Department.DeptId or null}.
- In this session we would learn, how to build such associations in MongoDB through proper mongoose schema definitions.

Mongoose Relationship Model

upGrad

Types of associations :

1. One-to-One
2. One-to-Many
3. Many-to-Many

Data Models:

1. Reference Data Models (Normalisation)
2. Embedded Data Models (Denormalisation)

upGrad

Poll 1 (45 sec)

In NoSQL database how many types of associations are possible. Choose all correct options :

1. One-to-One
2. One-to-Many
3. Normalisation
4. Many-to-Many
5. Denormalisation

upGrad

Poll 1 (45 sec)

In NoSQL database how many types of associations are possible. Choose all correct options :

1. **One-to-One**
2. **One-to-Many**
3. Normalisation
4. **Many-to-Many**
5. Denormalisation

One-to-One Model Relationship

One-to-One Model Relationship

upGrad

Assume that we have two entities:

1. SchoolAdmission
2. Child.

```
// SchoolAdmission
{
   _id: "12345xyz",
   enrollmentCode: "Sri130421"
}

// Child
{
   _id: "sri123",
   name: "Sristhi",
   age: 4,
   gender: “female"
}
```

One-to-One Model Relationship

upGrad

```
// SchoolAdmission
{
   _id: "12345xyz",
   enrollmentCode: "Sri130421"
}

// Child
{
   _id: "sri123",
   name: "Sristhi",
   age: 4,
   gender: “female"
}
```

We want to map SchoolAdmission and Child relationship in that:

- One **Child** has only one **SchoolAdmission** enrollment.
- One **enrollment** code belongs to only one **Child**.

This is called One-to-One Relationship.

Now we would learn two ways to design schema for this kind of model.

1. Reference Data Models (Normalisation)
2. Embedded Data Models (Denormalisation)

One-to-One Model Relationship

upGrad

- **Normalisation** means using reference data models.
- In this model, an object **A** connects to the object **B** by reference to object **B** id or a unique identification field.
- For example, **SchoolAdmission** has a ***Child_id*** field whose value is equal to Child object's unique ***_id*** value.

```
// SchoolAdmission
{
   _id: "12345xyz",
   enrollmentCode: "Sri130421"
}

// Child
{
   _id: "sri123",
   name: "Sristhi",
   age: 4,
   gender: “female"
}
```

One-to-One Model Relationship

upGrad

- In **normalisation** we create a references , something like a parent-child relation or foreign-primary key relationship.
- The opposite of it is **denormalisation**. Here we embed data instead. Denormalisation makes a embedded data model.
- In 'Embedded' data model, instead of using a reference, object **A** contains the whole object **B**, or object **B** is embedded inside object **A**.

```
// SchoolAdmission
{
   _id: "12345xyz",
   enrollmentCode: "Sri130421",
   child:
   {
      _id: "sri123",
      name: "Sristhi",
      age: 4,
      gender: "female"
   }
}
```

One-to-One Model Relationship

upGrad

- For example *beside*, **SchoolAdmission** has a **nested field child**.

```
// SchoolAdmission
{
   _id: "12345xyz",
   enrollmentCode: "Sri130421",
   child:
   {
      _id: "sri123",
      name: "Sristhi",
      age: 4,
      gender: "female"
   }
}
```

Hands on Exercise 1 (20 mins)

upGrad

- Create a project folder **one-to-one-reference**
- Setup Node.js application.
- Create the below project structure :
  - models folder with two files:
    - schoolAdmission.js
    - Child.js
  - server.js file
- Initialize the project
  npm init
- Install mongoose:
  npm install mongoose

Hands on Exercise 1 (20 mins)

upGrad

- In server.js
  - import mongoose to our app
  - connect to MongoDB database
- Create two schema with **mongoose.Schema()** constructor function.
- Go to **models/child.js**, define Child schema with three fields:
  - name
  - age
  - gender
- Go to **models/SchoolAdmission.js**, define SchoolAdmission schema with two fields
  - enrollmentCode
  - child object
- SchoolAdmission schema would have reference to child object.

Hands on Exercise 1 (20 mins)

upGrad

Hint:

- In the code, we need to add child field and set its type to **ObjectId** and ref to **Child**.
- Now if we save an enrollment to MongoDB database, a document will be added something like this:

```
{
   _id : ObjectId("5da000be062dc522eccaedeb"),
   enrollmentCode : "5DA000BC06",
   child : ObjectId("5da000bc062dc522eccaedea"),
   __v : 0
}
```

Hands on Exercise 1 (20 mins)

upGrad

- In server.js file :
  - Load the models
  - Define a function say createChild(), to create and save child object.
  - Further define a function say createEnrollment(), to create and save an enrollment object.
  - At last call the createChild() function, to add a new child object to the MongoDB.

- Run server.js file :

node server.js

Refer To: [one-to-one-reference]

Poll 2 (45 sec)

In Referenced data model (normalised data model) to fetch entire information about an entity , say about an enrollment, how many queries do you think would be executed internally in MongoDB.

Consider the inter-schema reference resolution as a call.

1. one call when we execute the appropriate find().

2. two calls, first to access the enrollment object and second is the reference call to get the respective child details.

Poll 2 (Answer)

In Referenced data model (normalised data model) to fetch entire information about an entity , say about an enrollment, how many queries do you think would be executed internally in MongoDB.

Consider the inter-schema reference resolution as a call.

1. one call when we execute the appropriate find().
2. **two calls, first to access the enrollment object and second is the reference call to get the respective child details.**

Tip : Embedded Data Model stores related information directly inside the object. Hence it gives us better performance with a single query.

Hands on Exercise 2 (20 mins)

upGrad

- Create a project folder **one-to-one-embedded**
- Setup Node.js application
- Create a project structure :
  - models folder with two files in it, ***schoolAdmission.js*** and ***child.js***
  - server.js file
- Initialize the project
  npm init
- Install mongoose:
  npm install mongoose

Hands on Exercise 2 (20 mins)

upGrad

- In server.js
  - import mongoose to our app
  - connect to MongoDB database
- Create two schema with ***mongoose.Schema()*** constructor function.
- Go to ***models/child.js***, define Child schema with three fields: name, age, gender.
- export Child Schema.
- Go to ***models/SchoolAdmission.js***, define SchoolAdmission schema with two fields : enrollmentCode and child Object. But instead using a reference, we import and assign ChildSchema directly.

Hands on Exercise 2 (20 mins)

upGrad

- Run server.js file :

  node server.js

Refer To: one-to-one-embedded

One-to-Many Model Relationship

One-to-Many Model Relationship

upGrad

- One-to-Many relationship is one of the most important modelling relationship that is used to design the backend applications.

- There are two data models for this kind of relationship as well:
  - Reference Data Models (Normalisation)
  - Embedded Data Models (Denormalisation)

- There are **three criteria to decide whether to choose referencing or embedding**, with the objective to improve application performance
  - Types of relationships
  - Data access patterns
  - Data cohesion

One-to-Many Model Relationship

upGrad

- Consider that we are designing a tutorial blog data model. Some of the relationships that we can think of for a tutorial blog model are:
  - A blog can have some images
  - A blog can have multiple comments
  - Multiple blogs based on single category

- All the above mentioned examples have one-to-many relationship.

One-to-Many Model Relationship

upGrad

- One-to-Many relationships have three categories:
  - One-to-Few
  - One-to-Many
  - One-to-aLot

- Depending on the **types of relationships**, **data access patterns**, or **data cohesion**, one can decide how to implement the data model, in other words, decide if we should denormalise or normalise the data.

One-to-Many Model Relationship

upGrad

- We know that for reference data models (normalisation) in MongoDB, we keep all the documents 'separated'.
- For example, here we have separate documents for Blog and Comments.
- As the two documents are separate , the Blog document needs a way to know which Comments it contains. Thus, each Comment ID is referenced in Blog document.

.

```
// Blog
{
  _id: "5db579f5faf1f8434098f7f5"
  title: "Tutorial #1",
  author: "rocky jagtiani"
  comments: [ "5db57a03faf1f8434098f7f8",
"5db57a04faf1f8434098f7f9" ]
}

// Comments
{
  _id: "5db57a03faf1f8434098f7f8",
  username: "ifrah",
  text: "This is a great tutorial.",
  createdAt: 2021-04-13T11:05:39.898Z
}

{
  _id: "5db57a04faf1f8434098f7f9",
  username: "sreejit",
  text: "Thank you, it helped me.",
  createdAt: 2021-04-13T11:05:40.710Z
}
```

One-to-Many Model Relationship

upGrad

- You can see that in the Blog document, we have an array where we stored the IDs of all the Comments so that when we request Blog data, we can easily identify its Comments.
- This type of referencing is called **Child Referencing**.

```
// Blog
{
  _id: "5db579f5faf1f8434098f7f5"
  title: "Tutorial #1",
  author: "rocky jagtiani"
  comments: [ "5db57a03faf1f8434098f7f8",
"5db57a04faf1f8434098f7f9" ]
}

// Comments
{
  _id: "5db57a03faf1f8434098f7f8",
  username: "ifrah",
  text: "This is a great tutorial.",
  createdAt: 2021-04-13T11:05:39.898Z
}

{
  _id: "5db57a04faf1f8434098f7f9",
  username: "sreejit",
  text: "Thank you, it helped me.",
  createdAt: 2021-04-13T11:05:40.710Z
}
```

One-to-Many Model Relationship

upGrad

- There are few problems associated with child referencing.
    - In the Blog document, we have an array where we stored the IDs of all the Comments.
    - Now, let's think about the array with all the IDs. The problem here is that this array of IDs can become very large if there are lots of comments. This is an **anti-pattern** in MongoDB that we should avoid.
    - To avoid the above given condition, we should practice **parent referencing**

```
// Category
{   _id: "5db66dd1f4892d34f4f4451a",
    name: "Node.js",
    description: "Node.js tutorial"
}

// Blogs
{   _id: "5db66dcdf4892d34f4f44515",
    title: "Tutorial #1",
    author: "rocky",
    category_id: "5db66dd1f4892d34f4f4451a"
}

{   _id: "5db66dd3f4892d34f4f4451b",
    title: "Tutorial #2",
    author: "rocky",
    category_id: "5db66dd1f4892d34f4f4451a"
}
```

One-to-Many Model Relationship

upGrad

- We can also denormalise data into a denormalised form simply by embedding the related documents right into the main document.
- So now we have all the relevant data about Comments right inside one Tutorial document and now, we do not need to separate the documents, collections, and IDs.

```
// Tutorial or Blog
{
  _id: "5db579f5faf1f8434098f7f5"
  title: "Tutorial #1",
  author: "rocky jagtiani"
  comments: [
      {  _id: "5db57a03faf1f8434098f7f8",
         username: "ifrah",
         text: "This is a great tutorial.",
         createdAt: 2021-04-13T11:05:39.898Z
      }

      {   _id: "5db57a04faf1f8434098f7f9",
          username: "sreejit",
          text: "Thank you, it helped me.",
          createdAt: 2021-04-13T11:05:40.710Z
      }
  ]
}
```

One-to-Many Model Relationship

upGrad

Because we can get all the data about Tutorial and Comments at the same time, **our application will need fewer queries to the database which will increase the performance**.

```
// Tutorial or Blog
{
  _id: "5db579f5faf1f8434098f7f5"
  title: "Tutorial #1",
  author: "rocky jagtiani"
  comments: [
      {  _id: "5db57a03faf1f8434098f7f8",
         username: "ifrah",
         text: "This is a great tutorial.",
         createdAt: 2021-04-13T11:05:39.898Z
      }

      {   _id: "5db57a04faf1f8434098f7f9",
          username: "sreejit",
          text: "Thank you, it helped me.",
          createdAt: 2021-04-13T11:05:40.710Z
      }
  ]
}
```

One-to-Many Model Relationship

upGrad

- We can decide how to implement the data model depending :
  - on the types of relationships that exists between collections
  - on data access patterns
  - on data cohesion.
- To actually take a decision, we need to combine all of these three criteria, not just use one of them in isolation.

```
// one-to-few relationship
{
  _id: "5db579f5faf1f8434098f7f5"
  title: "Tutorial #1",
  author: "rocky"
  images: [
            {
              url: "/images/mongodb.png",
              caption: "MongoDB Database"
            },
            {
              url: "/images/one-to-many.png",
              caption: "One to Many
Relationship"
            }
          ]
}
```

One-to-Many Model Relationship

upGrad

Types of Relationships between collections:

- Usually when we have **one-to-few** relationship, we embed the related documents into the parent documents. For example, a Tutorial has some images (10 or less).
- For a **one-to-many** relationship, we can either embed or reference according to the other two criteria.

```
// one-to-few relationship
{
  _id: "5db579f5faf1f8434098f7f5"
  title: "Tutorial #1",
  author: "rocky"
  images: [
            {
              url: "/images/mongodb.png",
              caption: "MongoDB Database"
            },
            {
              url: "/images/one-to-many.png",
              caption: "One to Many
Relationship"
            }
          ]
}
```

One-to-Many Model Relationship

upGrad

- With **one-to-aLot** relationship, we always use data references or normalising because if we embed a lot of documents inside one document, we would quickly make the main document too large. This severely affects performance. For example, imagine a Category having 300 Tutorials.

```
// one-to-few relationship
{
  _id: "5db579f5faf1f8434098f7f5"
  title: "Tutorial #1",
  author: "rocky"
  images: [
            {
              url: "/images/mongodb.png",
              caption: "MongoDB Database"
            },
            {
              url: "/images/one-to-many.png",
              caption: "One to Many
Relationship"
            }
          ]
}
```

One-to-Many Model Relationship

upGrad

Data access patterns

- Data access considers how often data is read and written i.e. find the **read/write ratio**.
- If the collections that we're working is read more often than updated, i.e. there is a lot more reading than writing (a high read/write ratio), then we should embed the data.
- The reason is that by embedding we only need one trip to the database per query while for referencing we need two trips. In each query, we save one trip to the database which makes the entire process way more effective.

One-to-Many Model Relationship

upGrad

- For example, a blog post having about 7-10 Images would be a good candidate for embedding because once these Images are saved to the database they are not really updated.

- On the other hand, if our data is updated a lot then we should consider referencing (normalising) the data. That's because the database engine does more work to update and embed a document than a standalone document, our main goal is performance so we just use referencing for data model.

- For example, each tutorial has many comments. Each time someone posts a comment, we need to update the corresponding tutorial document. The data can change all the time, so this is a great candidate for referencing.

One-to-Many Model Relationship

upGrad

Data cohesion

- The last criterion is just a measure for how much the data is related to each other. If two collections intrinsically belong together then they should be embedded into one another.
- In our example, all tutorials can have many images and every image intrinsically belongs to a tutorial. So images should be embedded into the tutorial document.
- If we frequently need to query both collections, we should normalise the data into two separate collections however closely they are related.

One-to-Many Model Relationship

upGrad

- Imagine that in our tutorial blog, we have a widget called recent images, and images could belong to separate tutorials. This means that we could query images on their own collections without necessarily querying for the tutorials.

- Thus, it's very necessary that **all the three** criteria are looked at simultaneously rather than just one of them in isolation.

Poll 3 (15 sec)

Which factor checks on the intrinsic bonding of data stored in different collections , i.e. are the data in different collections actually related to each other or not?

1. Normalisation
2. Data cohesion
3. Data access pattern
4. Denormalisation

Poll 3 (Answer)

Which factor checks on the intrinsic bonding of data stored in different collections , i.e. are the data in different collections actually related to each other or not?

1. Normalisation
2. **Data cohesion**
3. Data access pattern
4. Denormalisation

Hands on Exercise 3 (20 mins)

upGrad

- Create a project folder **one-to-many**
- Setup Node.js application
- Create a project structure :
  - models folder with two files in it, tutorial.js and index.js
  - server.js file
- Initialize the project
  npm init
- Install mongoose:
  npm install mongoose

Hands on Exercise 3 (20 mins)

upGrad

- In server.js
  - import mongoose to our app
  - connect to MongoDB database
- Create tutorial model with mongoose.Schema() constructor function.
- In models/tutorial.js, define tutorial with three fields: title, author, and images array.
- In server.js implement the following methods :
  - **createTutorial()**
  - **createImage()**
- Calls the above functions

Hands on Exercise 3 (20 mins)

upGrad

Hint : Use async / await , instead of using a chain of promises.

Note : Code after each await expression can be thought of as existing in a **.then** callback.

Useful resource :
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Refer To: one-to-many

Quick Tips

upGrad

- In **One-to-Many relationship** , we don’t have any standard or specific rule for all cases, it depends on the ways your application queries and updates data.
- You should identify the problems in your application’s use case and then model your data to answer the questions in the most efficient way.
- In **one-to-few** and **one-to-many** relationships, prefer embedding.
- In **One-to-aLot** relationships, prefer referencing.

Quick Tips

upGrad

- Use embedding when data is mostly read but rarely updated, and when two models belong intrinsically together.
- Prefer referencing when data is updated a lot, and you need to frequently query a collection on its own.
- Never allow arrays to grow indefinitely. Thus, prefer using **child referencing** for **one-to-many** relationships, and **parent referencing** for **One-to-aLot** relationships.

Hands on Exercise 4 (20 mins)

upGrad

- Improve the previous project (one-to-many) such that one can query images collections without necessarily querying for the tutorials.
- Name this project **one-to-many-version2**
- For the same define a new model image.js under models folder.
- We also need to change **createImage()** function in server.js

```
const createImage = function(tutorialId, image) {
  return db.Image.create(image).then(docImage => {
    console.log("\n>> Created Image:\n", docImage);
    return db.Tutorial.findByIdAndUpdate(
      tutorialId,
      {
        $push: {
          images: {
            _id: docImage._id,
            url: docImage.url,
            caption: docImage.caption
          }
        }
      },
      { new: true, useFindAndModify: false }
    );
  });
};
```

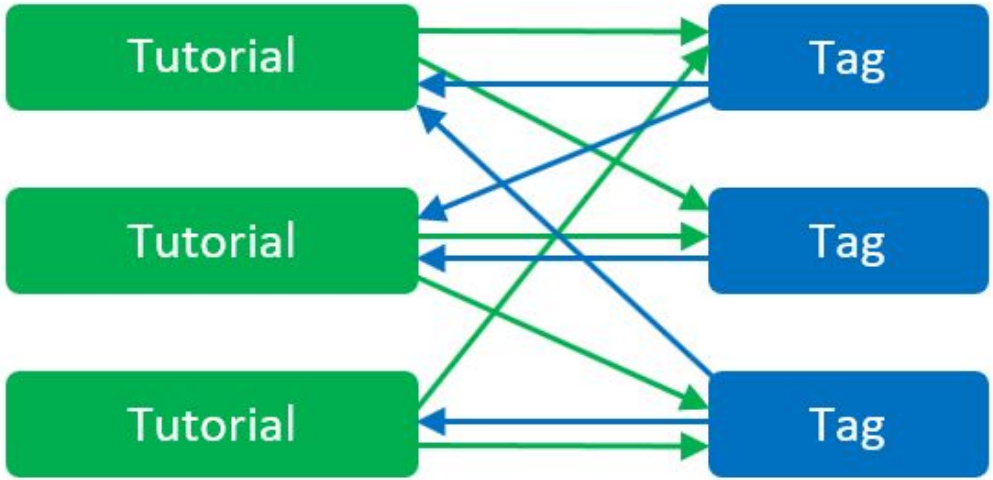
Refer To: [one-to-many-version2]

Many-to-Many Model Relationship

Many-to-Many Model Relationship

upGrad

- Consider a tutorial blog. The relationship between the tutorial and tag goes in both directions.
- A tutorial can have many tags and a tag can point to many tutorials



Many-to-Many Model Relationship

upGrad

- We can denormalise data into a denormalised form simply by embedding the related documents right into the main document.
- Let’s put all the relevant data about tags in tutorial document without separating the documents, collections, and IDs.

Note *: A slug is the part of a URL which identifies a particular page on a website in an easy to read form.*

Many-to-Many Model Relationship

upGrad

```
// Tutorial
{
  _id: "5db579f5faf1f8434098f123"
  title: "Tut #1",
  author: "rocky"
  tags: [
    {
       name: "tagA",
      slug: "tag-a"
    },
    {
       name: "tagB",
        slug: "tag-b"
    }
   ]
}
```

```
// Tag
{
  _id: "5db579f5faf1f84340abf456"
  name: "tagA",
  slug: "tag-a"
  tutorials: [
    {
      title: "Tut #1",
      author: "rocky"
    },
    {
      title: "Tut #2",
      author: "prachi"
    }
  ]
}
```

Many-to-Many Model Relationship

upGrad

- In referenced form i.e. 'normalised' we separate documents, collections, and IDs.
- As tutorials and tags are completely different document, the tutorial needs a way to know which tags it has, similarly a tag needs to know which tutorials contain it.
- To address the above issue, we use IDs to create references.

Many-to-Many Model Relationship

upGrad

```
// Tags
// tagA: [Tut #1, Tut #2]
{
  _id: "5db57a03faf1f8434098ab01",
  name: "tagA",
  slug: "tag-a",
  tutorials: [ "5db579f5faf1f8434098f123", "5db579f5faf1f8434098f456" ]
}

// tagB: [Tut #1]
{
  _id: "5db57a04faf1f8434098ab02",
  name: "tagB",
  slug: "tag-b",
  tutorials: [ "5db579f5faf1f8434098f123" ]
}
```

We can see that in the tag document, we have an array where we stored the IDs of all the tutorials so that when we request a tag data, we can easily identify its Tutorials. This type of referencing is called Child Referencing: the parent (Tag) references its children (Tutorials).

Many-to-Many Model Relationship

upGrad

- In parent referencing each child document keeps a reference to the parent element.
- Tutorials maintain reference to its tags' ids.

```
// Tutorial
// Tut #1: [tagA, tagB]
{
  _id: "5db579f5faf1f8434098f123"
  title: "Tut #1",
  author: "rocky"
  tags: [ "5db57a03faf1f8434098ab01", "5db57a04faf1f8434098ab02" ],
}

// Tut #2: [tagA]
{
  _id: "5db579f5faf1f8434098f456"
  title: "Tut #2",
  author: "prachi"
  tags: [ "5db57a03faf1f8434098ab01" ],
}
```

Many-to-Many Model Relationship

upGrad

Presently we've done **Two-way Referencing** where Tags and Tutorials are connected in both directions:

- In each tag, we keep references to all tutorials that are tagged.
- In each tutorial, we also keep references to its tags.

Many-to-Many Model Relationship

upGrad

- For embedded data models, you can see that we can get all the data about tutorial with its tags (or tag with its tutorials) at the same time, thus our application will need fewer queries. This results in increased performance.

- But when the data becomes larger, duplicates are inevitable. Duplicates increase the risk for updating the document.

- For example, if you want to change the name of tagB, you have to change not only the tag's document but also find the tutorial that contains that tag and update the field.

- Hence, with **Many-to-Many relationship**, we **always use data references or normalising**.

Homework 1

upGrad

Define a proper schema according to the constraints given.

- In a housing society, you are asked to store the details of cars and their respective owners. For the same, you need to create two schemas to store following information.
  - **CarOwner :**
    - name : String, required, minimum length: 3 chars and maximum length: 30 chars.
    - age : number, required, min age: 18 and max age: 100
    - cars : array of references to car objects.
  - **Car:**
    - make: String, required, minimum length: 2 chars and maximum length: 30 chars.
    - model: String, required, minimum length: 2 chars and maximum length: 30 chars.
    - owner: as a reference to a single CarOwner object.

Homework 1

upGrad

- What type of relationship do you think **carOwner - Car** schema is forming?
- How is the data being stored: normalised or denormalised form? Justify it.

Tip: https://mongoosejs.com/docs/validation.html

Refer To: Car-CarOwner-Relation

Homework 2

upGrad

Define a proper schema according to the constraints given.

- You like watching movies on Netflix or Amazon prime. You have noticed that in all these apps listing of movies and their lead actors is usually advertised. Imagine, you wish to hold such information in MongoDB schema. Fo the same you would need to create two schemas - Actor and Movie.
    - Actor :
        - name : String, required, minimum length: 3 chars and maximum length: 30 chars
        - birth_year : Date type , required
        - Movies : An array referencing to movie objects
    - Movie :
        - title : String, required, minimum length: 3 chars and maximum length: 30 chars.
        - year_of_release :Date type, required
        - actors : An array referencing actor objects

Refer To: Actor-Movie-Relation

Doubt Clearance (5 mins)

Key Takeaways

- We learnt about different relationship models and their applications.

The following tasks are to be completed after today's session:

Homework
MCQs
Coding Questions

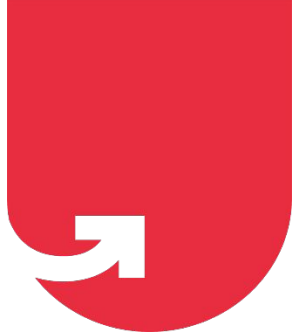
upGrad

In the next class, we will discuss...

- We will get introduced to the concept of Express.js and learn about its applications

upGrad

#RahoAmbitious



Thank you!