upGrad

#RahoAmbitious

Full Stack Software Development

upGrad

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

Lecture on: Mongoose ODM

Instructor: Rocky Jagtiani

In the previous class, we covered...

- Basics of MongoDB

upGrad

Poll 1 (15 Sec)

Which MongoDB command is used to **create** a new document?

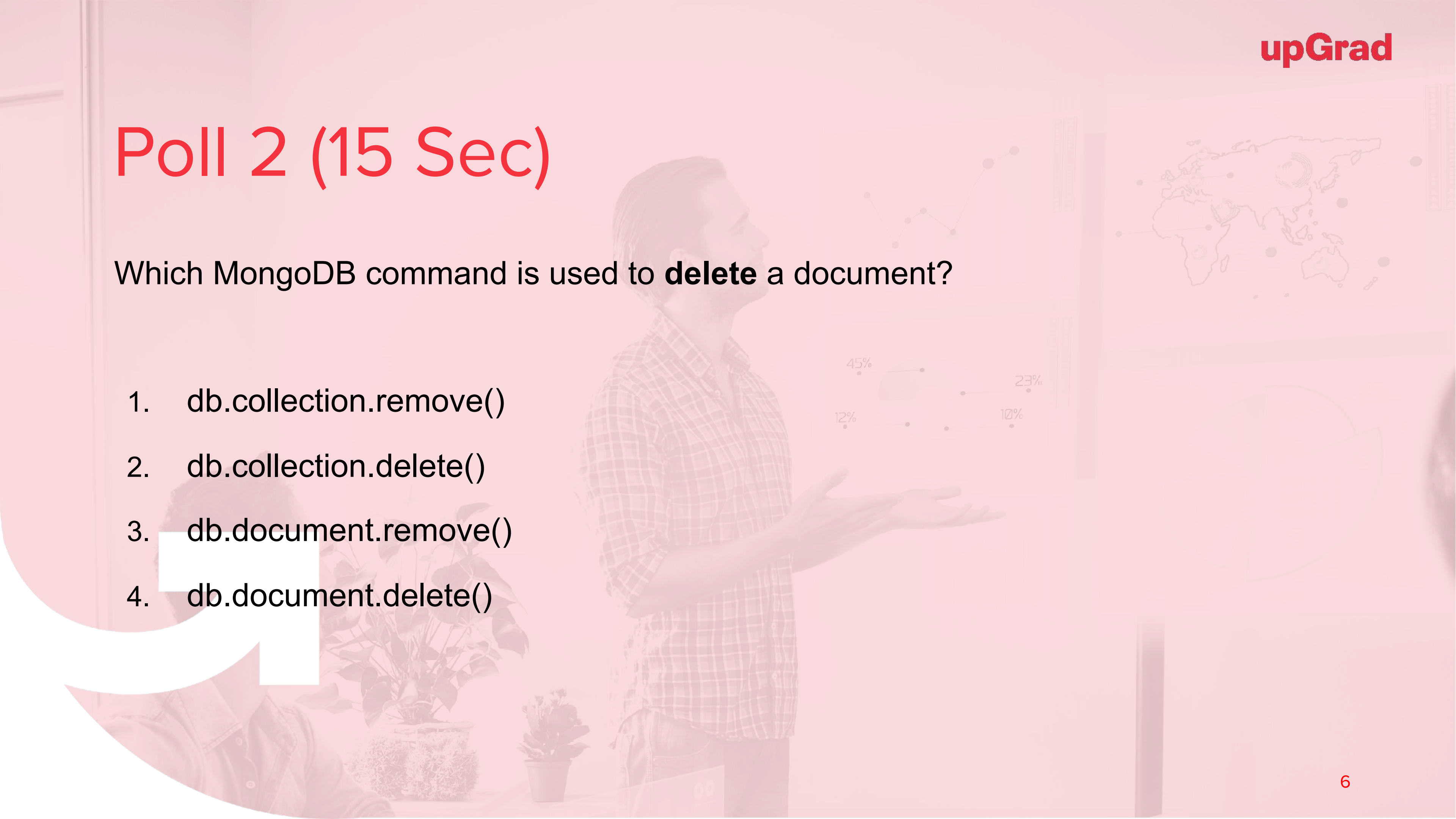
1. db.collection.update()
2. db.collection.create()
3. db.collection.new()
4. db.collection.insert()

upGrad

Poll 1 (Answer)

Which MongoDB command is used to **create** a new document?

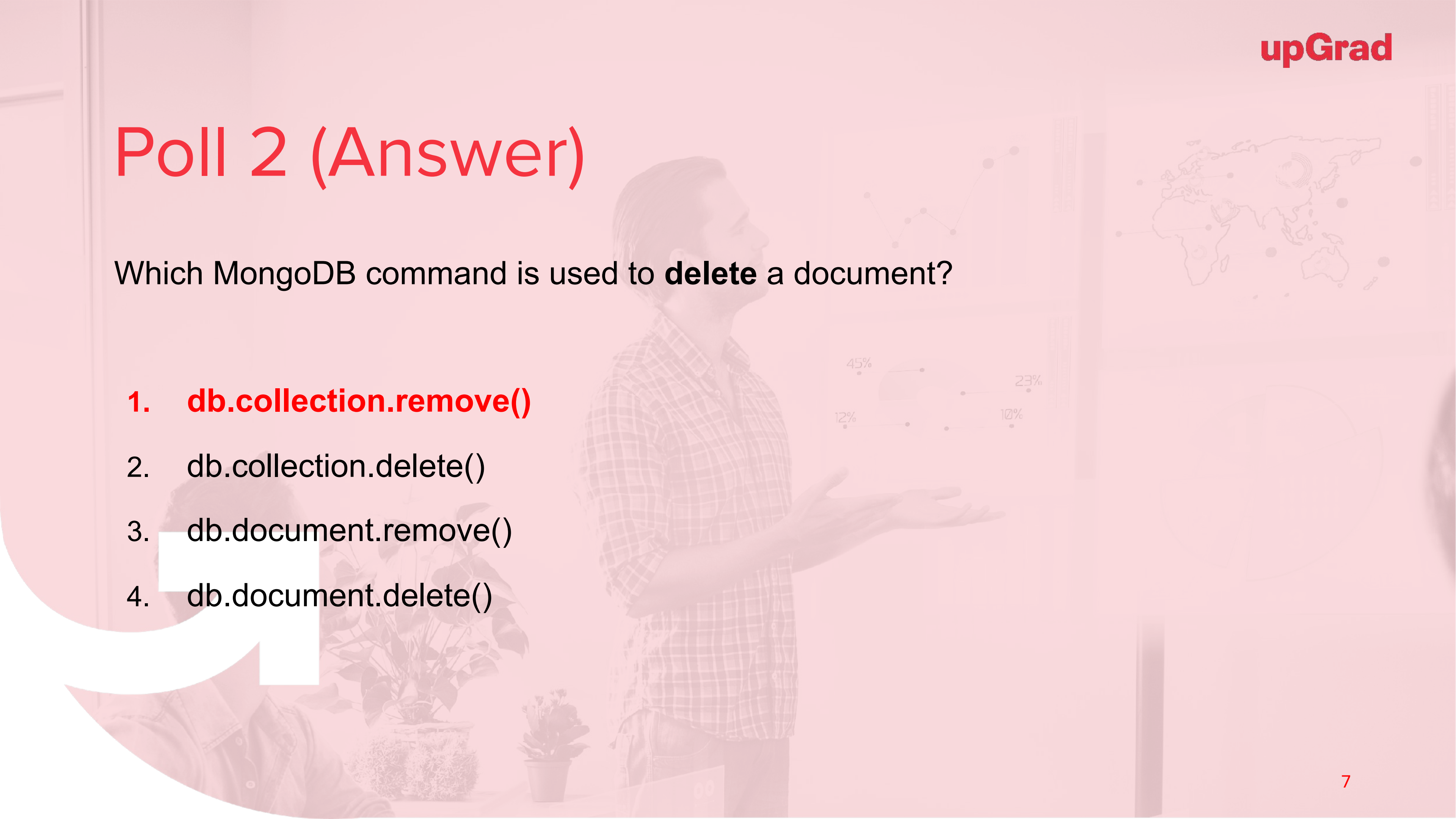
1. db.collection.update()
2. db.collection.create()
3. db.collection.new()
4. **db.collection.insert()**



Poll 2 (15 Sec)

Which MongoDB command is used to **delete** a document?

1. db.collection.remove()
2. db.collection.delete()
3. db.document.remove()
4. db.document.delete()



Poll 2 (Answer)

Which MongoDB command is used to **delete** a document?

1. **db.collection.remove()**
2. db.collection.delete()
3. db.document.remove()
4. db.document.delete()

Today’s Agenda

- JavaScript Promises
- Mongoose ODM
- Helper Methods

upGrad

JavaScript Promises

```
let promise = new Promise((resolve, reject) => {
    if(1 >= 0) {
       resolve("This is working!"); }
    else {
       reject(Error("It failed!"));
    }
})

promise.then(

   (result) => console.log(result),

   (error) => console.log(error)
);

//This is working!
```

JavaScript Promises

- Let us quickly revisit the concept of promises in JavaScript because Mongoose which is a MongoDB object modelling tool is designed to work in an asynchronous environment and thus, **Mongoose supports both promises and callbacks.**

```
let myPromise = new Promise(function(myResolve, myReject) {
   // "Producing Code" (May take some time)
   myResolve(); // when successful
   myReject(); // when error
});

  // "Consuming Code" (Must wait for a fulfilled Promise)
  myPromise.then(
     function(value) { /* code if successful */ },
     function(error) { /* code if some error */ }
  );
```

JavaScript Promises

- A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it’s not resolved.

- A promise may be in one of 3 possible states:
  - fulfilled
  - rejected
  - pending

```
let myPromise = new Promise(function(myResolve, myReject) {
   // "Producing Code" (May take some time)
   myResolve(); // when successful
   myReject(); // when error
});

  // "Consuming Code" (Must wait for a fulfilled Promise)
  myPromise.then(
     function(value) { /* code if successful */ },
     function(error) { /* code if some error */ }
  );
```

JavaScript Promises

upGrad

When the executing code obtains the result, it should call one of the two callbacks:

Result	Call
Success	myResolve(result value)
Error	myReject(error object)

```
let myPromise = new Promise(function(myResolve, myReject) {
   // "Producing Code" (May take some time)
   myResolve(); // when successful
   myReject(); // when error
});

  // "Consuming Code" (Must wait for a fulfilled Promise)
  myPromise.then(
     function(value) { /* code if successful */ },
     function(error) { /* code if some error */ }
  );
```

JavaScript Promises

The Promise object supports two properties: **state and result**.

- While a Promise object is "pending" (working), the result is undefined.
- When a Promise object is "fulfilled", the result is a value.
- When a Promise object is "rejected", the result is an error object.

myPromise.state	**myPromise.result**
"pending"	undefined
"fulfilled"	a result value
"rejected"	an error object

upGrad

Mongoose

Introduction to Mongoose

upGrad

- We know, MongoDB is a schema-less NoSQL document database. It means you can store JSON documents in it, and the structure of these documents can vary as it is not enforced like SQL databases.

- Using NoSQL database speeds up the process of application development and reduces the complexity of deployments.

- Mongoose is an ***Object Document Modeling (ODM)*** layer that sits on top of the Node.js MongoDB API. If you're coming from an SQL background, then Mongoose is similar to an ***Object Relational Mapping (ORM)***.

Why choose Mongoose over MongoDB?

- **Schemas**: Schemas allow to give a structure to the collection.

- **Built-in validation**: This means you don’t have to write the extra code that you had to write with the MongoDB driver. By simply including things like ***required: true*** in your schema definitions, Mongoose provides in-house validations for your collections (including data types).

- **Instance methods**: You can define custom methods on a document with minimal code. While it’s possible to do the same in MongoDB, Mongoose makes it easier to create and organise such methods within your schema definition.

Why choose Mongoose over MongoDB?

- **Returning results**: Returning documents generated by queries is generally easier in Mongoose. One example is the update queries. In MongoDB, this query only returns an object with a success flag and the number of documents modified. Mongoose, on the other hand, provides you the updated document itself so you can easily process the results.

Introduction to Mongoose

upGrad

Schemas:
Everything in Mongoose starts with a Schema.

Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

It has information about properties/field types of documents. Schemas can also store information about validation and default values, and whether a particular property is required. In simple words, they're blueprints for documents.

Model:
A model is a class with which we construct documents.

Install mongoose

Simply install the mongoose module through npm.

```
npm install mongoose
```

Note *: Make a folder* ***mongoose-pracs*** *and then do*

```
npm init

npm install mongoose --S
```

Introduction to Mongoose

- Create a file **./database.js** under the project root.

- Add a simple class with a method that connects to the database.
  Note: The connection string will vary based on your installation.

```
const mongoose = require('mongoose')
mongoose.connect('mongodb://localhost/test', {useNewUrlParser:
true,  useUnifiedTopology: true });
```

- In the given example, we're connecting to a database called test. The first parameter is the URL and the second parameter is options.

Note *: We would be using a Object Oriented Style of coding in the practical.*

Refer To: mongoose-pracs/database.js

Mongoose Schema and Model

- A Mongoose model is a wrapper on the Mongoose schema.
- Schema defines the structure of the document, default values, validators, etc., whereas a model provides an interface to the database for creating, querying, updating, deleting records.
- Creating a Mongoose model comprises primarily of three parts:
  - **Referencing Mongoose**

```
let mongoose = require('mongoose')
```

Mongoose Schema and Model

upGrad

- **Defining the Schema**
  A schema defines document properties through an object where the key name corresponds to the property name in the collection.

```
let emailSchema = new mongoose.Schema({
  email: String
})
```

Here we define a **property called email with a schema type String** which maps to an internal validator that will be triggered when the model is saved to the database. It will fail if the data type of the value is not a string type.

Mongoose Schema and Model

- **Exporting a Model**
  We need to call the model constructor on the Mongoose instance and pass it the name of the collection and a reference to the schema definition.

```
module.exports = mongoose.model('Email', emailSchema)
```

We can create an instance of the model we defined above and populate it using the following syntax:

```
let EmailModel = require('./email_model_v1.js')
let msg = new EmailModel({
  email: 'rockyjagtiani@gmail.com'
})
```

Mongoose Schema and Model

The following schema types are permitted:

- Array
- Boolean
- Buffer
- Date
- Mixed (A generic/flexible data type)
- Number
- ObjectId
- String

Mixed and ObjectId are defined under require('mongoose').Schema.Types.

Refer To: email_model_v1.js and email.js

Hands on Exercise (20 mins)

Enhance the email schema as follows :

- make the email property unique.
- mark email as required field such that it cannot be null.
- convert the email value to lowercase before saving it.
- add a validation function that will ensure that the value is a valid email address. (*hint: use mongoose built-in library called validator library*)

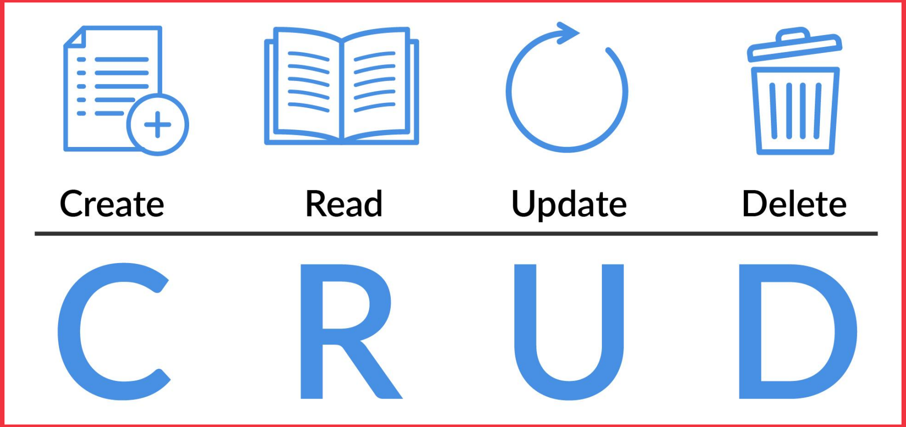
Steps :

1. Reuse the email_model_v1.js file. Edit it to accommodate above properties and save the file as **email_model_v2.js**
2. To use the mongoose validator library, install it as ***npm install validator –S***
3. Try running the same code twice. On the second run, we should get **duplicate key** error**.**

Refer To: hands-on-dataset.js and mongo-hands-on.js

upGrad

CRUD Operations with Mongoose



CRUD Operations - Create Record

To create an instance of the email model and save it to the database use .save method.

```
let EmailModel = require('./email_model_v2.js')
let msg = new EmailModel({
  email: 'SREEJITnair@GMAIL.COM'
})

msg.save()
   .then(doc => {
     console.log(doc)
   })
   .catch(err => {
     console.error(err)
   })
```

CRUD Operations - Create Record

upGrad

The following fields are returned (internal fields are prefixed with an underscore) after a CREATE operation , i.e. when using **.save()** :

1. The **_id** field is auto-generated by Mongo and is a primary key of the collection. Its value is a unique identifier for the document.

2. The value of the email field is returned. Notice that it is lower-cased because we specified the **lowercase:true** attribute in the schema.

3. **__v** is the **versionKey** property set on each document when first created by Mongoose. Its value contains the internal revision of the document.

CRUD Operations - Fetch Record

To retrieve the record use find method with some search query.

The model class exposes several static methods and instance methods to perform operations on the database. We will now try to find the record that we created previously using the find method and pass the email as the search term.

```
EmailModel.find({
    email: 'rockyjagtiani@gmail.com'
    // search query
  })
  .then(doc => {
    console.log(doc)
  })
  .catch(err => {
    console.error(err)
  })
```

Refer To: fetch-records.js

CRUD Operations - Update Record

- To find and modify a record use ***findOneAndUpdate()*** method.
- As the name implies, ***findOneAndUpdate()*** finds the first document that matches a given filter, applies an update, and returns the document.
- By default, ***findOneAndUpdate()*** returns the original document.
- For the program to return the updated document, the new option should be set to true.

For performance reasons, Mongoose won’t return the updated document so we need to pass an additional parameter to ask for it.

CRUD Operations - Update Record

upGrad

```
EmailModel.findOneAndUpdate({
   email: 'akshaykumar@gmail.com'
   // search query
 },
 {
   email: 'superhero@gmail.com'
   // field:values to update
 },
 {
   new: true,                  // return updated doc
   runValidators: true       // validate before update
 }
)
```

Refer To: update-one-record.js

CRUD Operations - Update Record

- The ***updateMany()*** function is same as ***update()***, in MongoDB.

- It will update all documents that match the given condition. It is often used when the user wishes to update all the documents with specific condition.

- This function has 4 parameters
  filter, update, options, callback.

- For example: find all documents matching the condition**(marks >= 99)** and update remark as 'excellent' in all the documents.

CRUD Operations - Update Record

upGrad

```
StudentTestSchema.updateMany(
   {marks : {$gte:99}},
   {remark : "excellent"}
).then(doc => {
    console.log(doc)
 })
 .catch(err => {
    console.error(err)
  })
```

Refer To: [update-many-records.js]

CRUD Operations - Delete Record

upGrad

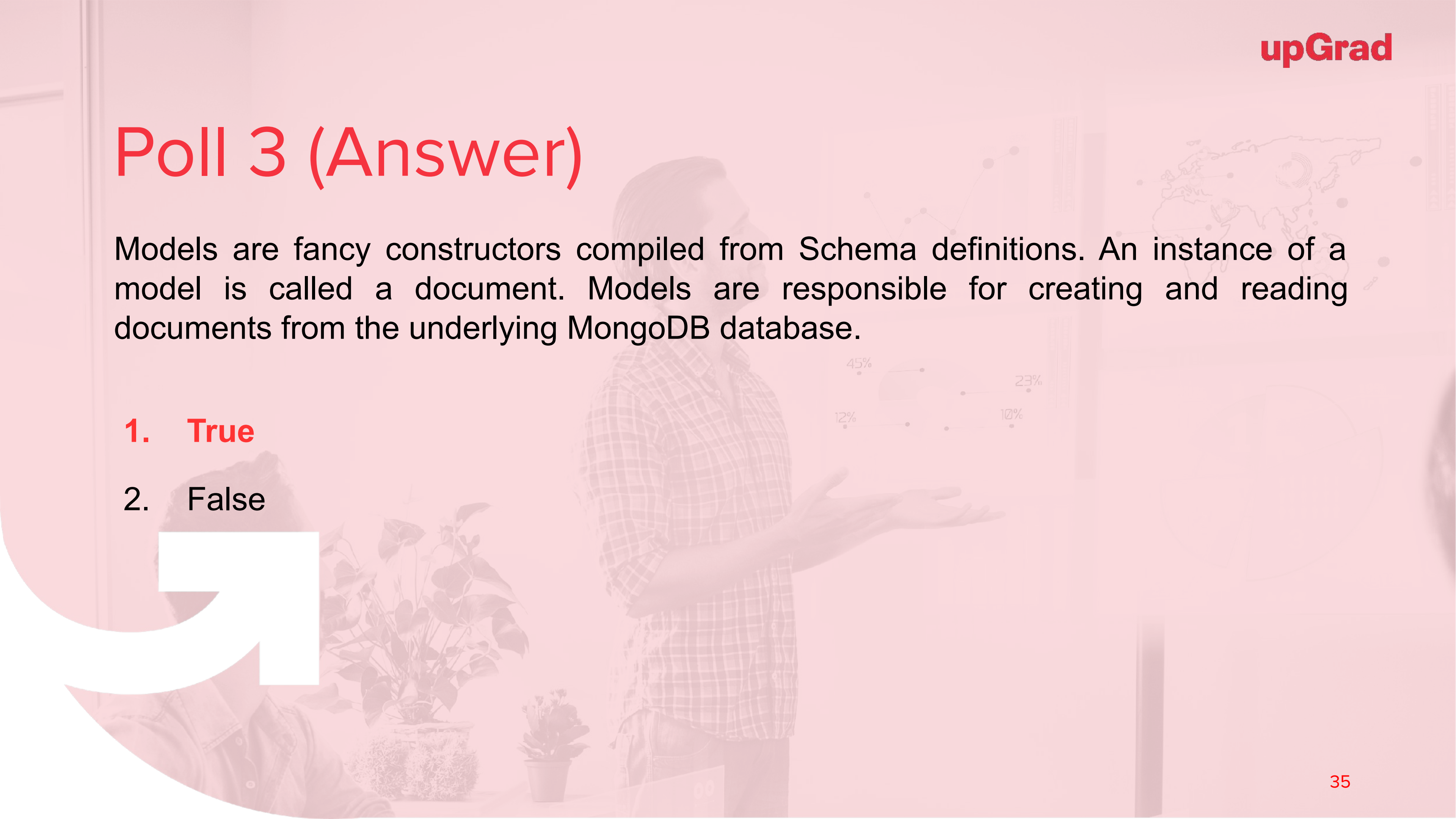
- We can search for documents based on certain conditions and remove it.
- The ***findOneAndRemove()*** or ***findOneAndDelete()*** function is used to find the element according to the condition and then remove the first matched element.

```
StudentTestSchema.findOneAndRemove({
   marks:{$lt:99}
})
.then(doc => {
  console.log(doc)
})
.catch(err => {
  console.error(err)
})
```

Poll 3 (15 Sec)

Models are fancy constructors compiled from Schema definitions. An instance of a model is called a document. Models are responsible for creating and reading documents from the underlying MongoDB database.

1. True
2. False



Poll 3 (Answer)

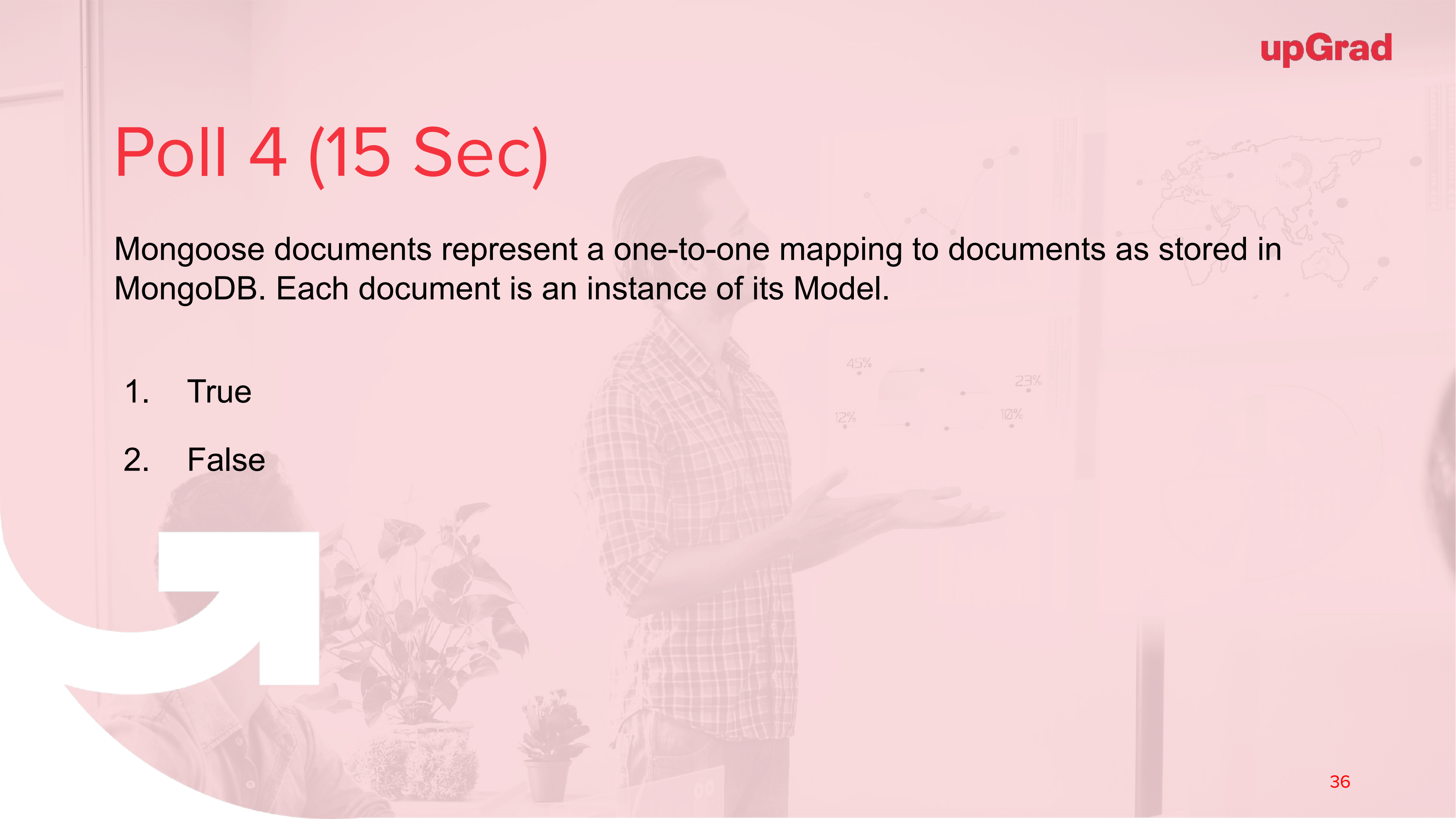
Models are fancy constructors compiled from Schema definitions. An instance of a model is called a document. Models are responsible for creating and reading documents from the underlying MongoDB database.

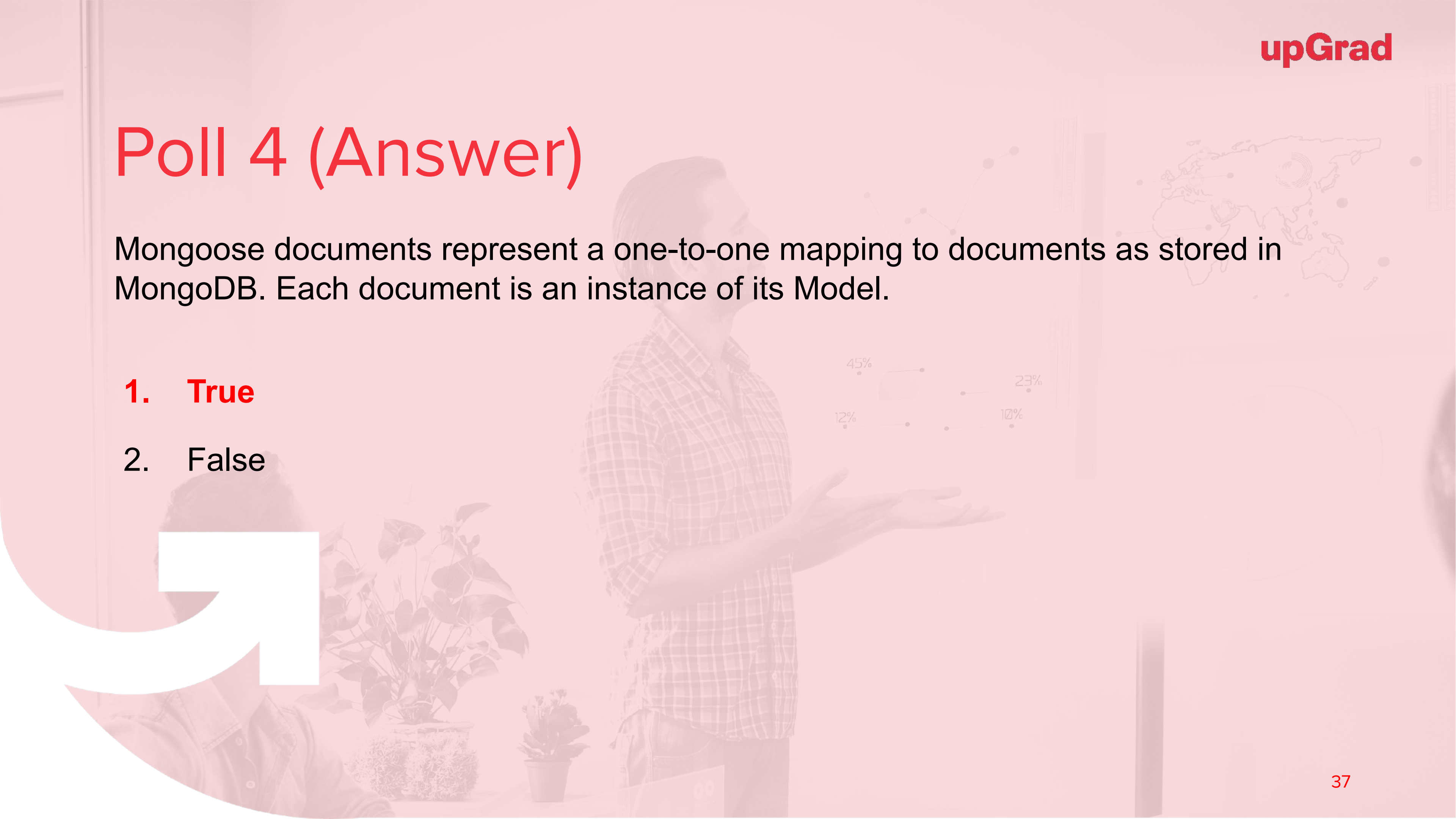
1. **True**
2. False

Poll 4 (15 Sec)

Mongoose documents represent a one-to-one mapping to documents as stored in MongoDB. Each document is an instance of its Model.

1. True
2. False





Poll 4 (Answer)

Mongoose documents represent a one-to-one mapping to documents as stored in MongoDB. Each document is an instance of its Model.

1. **True**
2. False

Hands on Exercise (20 mins)

- Create a book Schema with following properties :
  - name
  - author
  - pages
  - Bestprice
  - kindleversion

- Define the Model.

- Insert four book records. (use ***insertMany()***)

- Update one of the documents. Like the one which has a **bestprice** more than Rs 2000. Change it to Rs 1999. (use **updateOne()**)

- Delete one document which has no kindle version. (use **deleteOne()**)

- Find all books having kindleVersion as true.

Refer To: books-CRUD.js

Helper Methods

Helper Methods

upGrad

- We have looked at some of the basic functionality known as CRUD (Create, Read, Update, Delete) operations, but Mongoose also provides the ability to configure several types of helper methods and properties.

- To understand power of helper methods, we would code the following :
  - Instance methods
  - Static methods
  - Hooks (or middleware functions)

Instance Methods

upGrad

- The documents derived from their respective schemas can use these methods.
- For example, with respect to the **UserSchema** we would create a instance method that will provide us with the initials name of the people.

  Remember : Instance methods work on the objects and not on the model

Instance Methods

```
let userSchema = new mongoose.Schema({
  firstName: String,
  lastName: String
})

userSchema.methods.getInitials = function() {
    return this.firstName[0] + this.lastName[0]
}

// making a Model from the Schema
let UserModel = mongoose.model('User', userSchema)

// Making objects of the Model
let userobject = new UserModel({
    firstName: 'Rocky',
    lastName: 'Jagtiani'
})

let initials = userobject.getInitials()
console.log(initials)
```

Refer To: instance-method.js

Static Methods

upGrad

- These are used whenever we don’t have a particular object or we don’t need it.
- The models derived from their respective schemas can use these methods.
- For example, we would add a static method to find someone by its first-name.

```
userSchema.statics.findByName = function(value){
    this.find( {firstName : value}, (err, document)=>{
        console.log(document);
    })
}
```

Refer To: static-method.js

Middleware

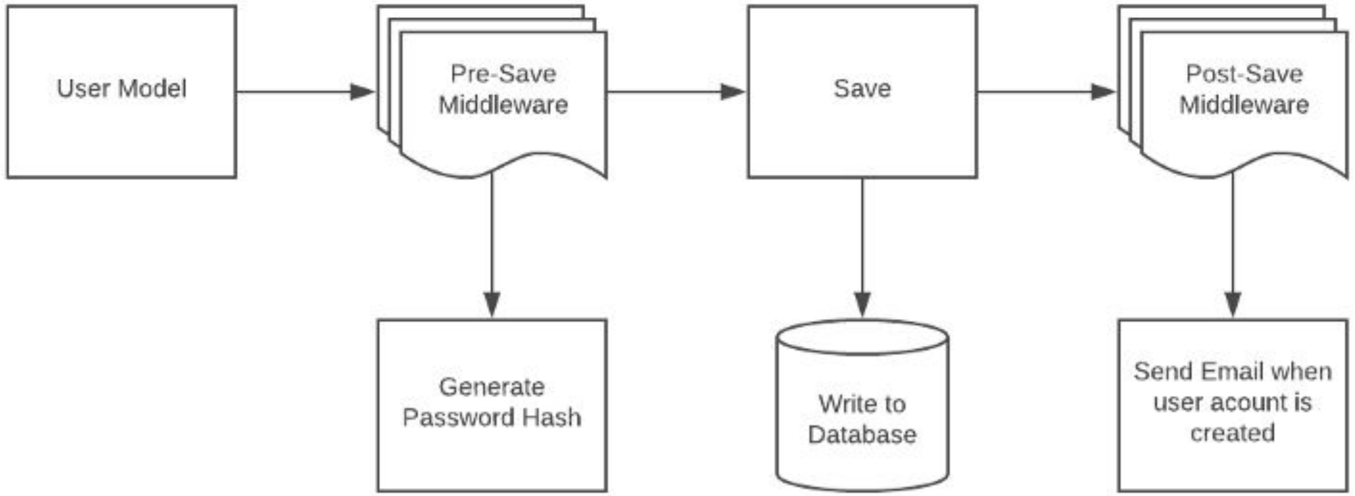
upGrad

- Middleware are functions that run at specific stages of a pipeline.

- Mongoose supports middleware for the following operations:
  - Aggregate
  - Document
  - Model
  - Query

- For instance, models have **pre and post functions** that take two parameters:
  - Type of event ('init', 'validate', 'save', 'remove')
  - A callback that is executed with ***this*** referencing the model instance.

Middleware



Refer To: [middleware.js]

Middleware

upGrad

- When ***model.save()*** is called, there is a pre(‘save’, …) and post(‘save’, …) event that is triggered.

- For the second parameter to pre() and post(), you can pass a function that is called when the event is triggered. These functions take a parameter to the next function in the middleware chain.

- For example: **pre-save hook** set's values for **createdAt** and **updatedAt** properties.

```
userSchema.pre('save', function (next) {
  let now = Date.now()

  this.updatedAt = now
  // Set a value for createdAt only if it is null
  if (!this.createdAt) {
    this.createdAt = now
  }

  // Call the next function in the pre-save chain
  next()
})
```

Refer To: [middleware.js]()

Plugins

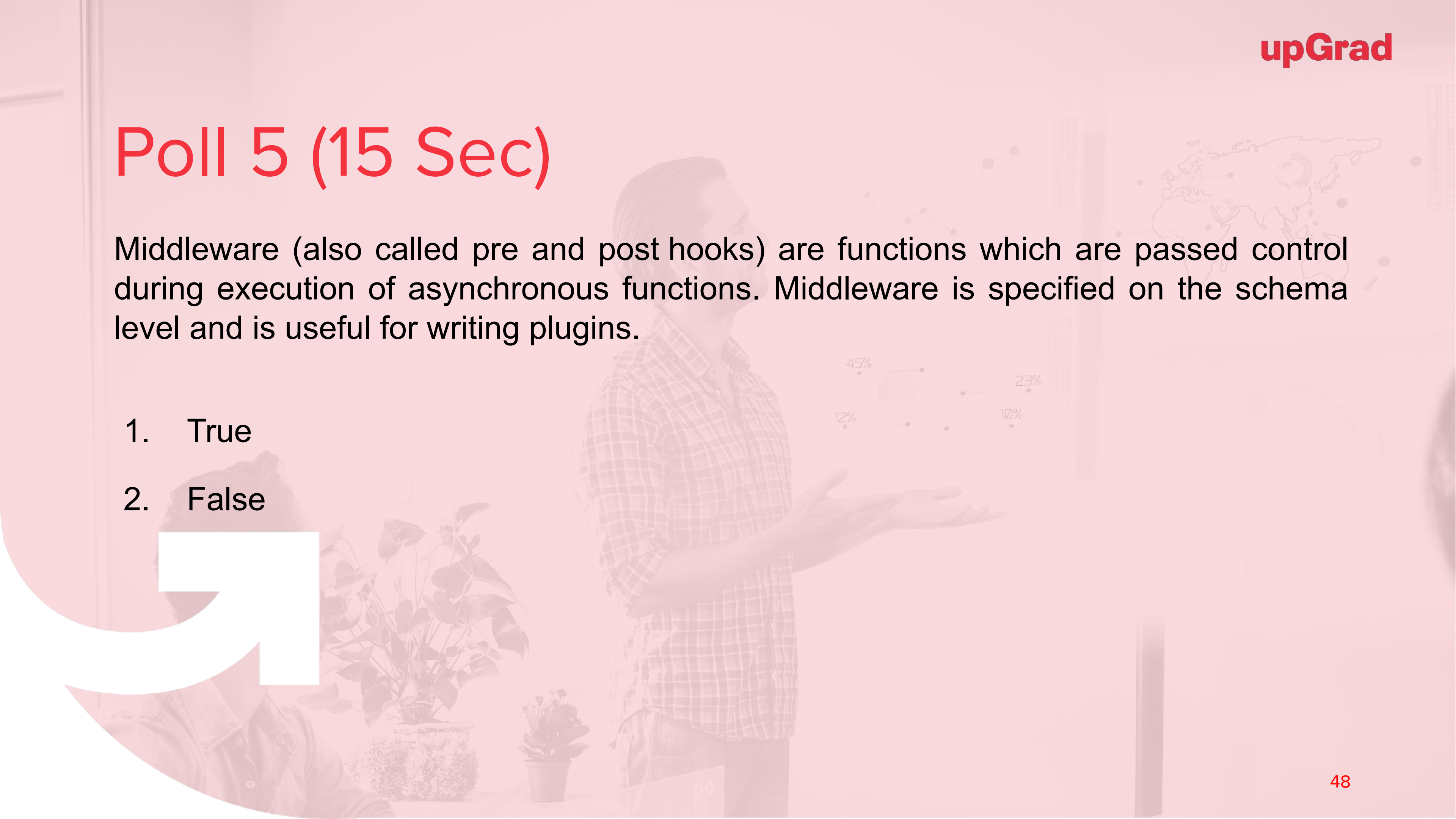
upGrad

- Suppose that we want to track when a record was created and last updated on every collection in our database.
- Instead of repeating the middleware process i.e. defining a pre("save", ...) method on each schema; we can create a plugin and apply it to every schema.

```
let timestampPlugin = require('./plugins/timestamp')

emailSchema.plugin(timestampPlugin)
userSchema.plugin(timestampPlugin)
```

Refer To: useplugins.js and plugins/timestamp.js



Poll 5 (15 Sec)

Middleware (also called pre and post hooks) are functions which are passed control during execution of asynchronous functions. Middleware is specified on the schema level and is useful for writing plugins.

1. True
2. False

Poll 5 (Answer)

Middleware (also called pre and post hooks) are functions which are passed control during execution of asynchronous functions. Middleware is specified on the schema level and is useful for writing plugins.

1. **True**
2. False

upGrad

Project Work - Checkpoint 4

Checkpoint 4 - Define the Course Schema

upGrad

- Define the Schema according to client requirements. (refer **app/models/tutorial.model.js**)
    - title: String
    - description: String
    - published: Boolean
    - skills : String Array
    - Chapters : String Array

- priceInRupees: Number
  keep the default as 5000. Don't allow negative values, means minimum as 0 and maximum as 30000.

- priceAfterDiscount: Number
  Don't allow negative values, means minimum as 0 and maximum as 30000.

Checkpoint 4 - Define the Course Schema

- category: String
  category would gets its value through a drop down from React frontend

- imageURL: String
  Keep the default as : https://ik.imagekit.io/upgrad1/marketing-platform-assets/meta-images/home.jpg

- videoURL: String
  Keep the default as : https://www.youtube.com/watch?v=MTdpHs6HWwM

- notesURL: String
  Keep the default as : https://www.mongodb.com/mern-stack

Checkpoint 4 - Define the Course Schema

- duration in mins: Number
  Set default to 60 and keep minimum as 0 and maximum as 1200 minutes

- popularity: Number
  Keep the default as : 4.0

Refer To: [app/models/tutorial.model.js]()

Checkpoint 4 - Define the User Schema

upGrad

- Define a simple generic USER Schema.
  (refer **app/models/user.model.js**)
  - email: String
  - password: String
  - firstName : String
  - lastName : String
  - role: String

- Keep the default as 'user'. There would be only two kinds of user
  - user
  - admin

Refer To: **app/models/user.model.js**

Checkpoint 4 - Define the Enrollment Schema

upGrad

- Define a Enrollment Schema.
    - userId: String
    - courseId: String

Refer To: app/models/enrollment.model.js

Checkpoint 4 - Index File

upGrad

- Under the app/models folder we would need to define a index.js file. This would be executed when we load the app/models folder.
- The app/models/index.js file would load the db.config file, load the mongoose library and make a mongoose object, to be used in the entire project.
- In the server.js when we load /app/models
  const db = require("./app/models");
- index.js runs and initializes all the model parameters, needed for backend.
- In the server.js , we would later add more code. For backend programming only below code is needed in **server.js.**

Checkpoint 4 - Index File

```
const db = require("./app/models");
db.mongoose
  .connect(db.url, {
    useNewUrlParser: true,
    useUnifiedTopology: true
  })
  .then(() => {
    console.log("Connected to the database!");
  })
  .catch(err => {
    console.log("Cannot connect to the database!", err);
    process.exit(1);
  });
```

Refer To: server.js and app/models/index.js

Homework

upGrad

- Create a Movie Schema with following properties :
  - title (define options for required field and unique field)
  - year (define option for required field)
  - actors
- Define the Model.
- Insert 2-3 movie records. (use ***insertMany()*** method)
- Check at the backend using compass.

Refer: mongoose_homework_1.js

upGrad

Doubt Clearance (5 mins)

Key Takeaways

- We understood the advantages of using Mongoose ODM in our applications.

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

Homework
MCQs
Coding Questions
Course Project - Checkpoint 5

upGrad

In the next class, we will discuss...

- Different types of relationships in Mongoose.

upGrad
#RahoAmbitious

Thank you!