Summary of MongoDB Advanced

Validators help us to **maintain the quality and accuracy** of the data. By using validators we check if the values in a document follow the rules which we've set for each field.

There are mainly 2 types of Validators in MongoDB.

1. Built-in Validators
2. Custom Validators

🧪 Built-in Validators

- These are validators which are already available in mongoDB.
- For Example:
    - minlength: check minimum length of the string
    - maxlength: check maximum length of the string
    - min: check the minimum value
    - max: check the maximum value
    - match: for comparing value using regular expression
    - enum: for giving the set of values for given field(in array)

🎨 Custom Validators

- This is the validators which we define.
- For Example:
    - required: [true, "Custom Message(Please enter the username)"]

```
// Way to add custom validators
hobbies: { type: [String], validate: {
      validator: (value) => {
            // Custom Validation Logic
            return value.length > 1
        }
    }
}
```

⏳ Async Validators

- In custom validators we can also perform async tasks like checking database or something else. We can do that in the custom validators callback functions.

💼 Other Schema Options

- These are the other schema types for schema fields.
- For Example:
    - **required: true** (make sure field is require or not)
    - **lowercase: true** (store field value in all lowercase)
    - **uppercase: true** (store field value in all uppercase)
    - **trim: true** (trim the unnecessary white spaces from value)
    - **unique: true** (make sure value is unique in collection)

🕸️ Relationship between models

- In the real-world between our two more than two models we might have connection or we can say relationship. By that relationship we can organize data very efficiently.
- So there are 3 approach of relationship in mongoDB:
    1. Reference Approach
    2. Embed Approach
    3. Hybrid Approach

🔍 Reference Approach

- In the reference approach we add reference in our second collection which is the unique object id. By that reference or object id we can access whole data of collection 01.
- For Example:

```
// Users Collection
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number,
});

// Post Collection
const postSchema = new mongoose.Schema({
  content: String,
  date: { type: Date, default: Date.now },
  user: { type: mongoose.Schema.Types.ObjectId, ref: "User" }, // This is refenerce
});

// Access Users Collection Data using Reference
const posts = await Post.find().populate("user", "_id name email");
```

- But Imagine in this approach we need to find multiple users data in single request. Then behind the scene it will run one more query for getting the reference data. This might cause performance issue. But this will never cause data inconsistency.

📦 Embed Approach

- In the embed approach, instead of adding reference in another collection, we have single model and we add all our data in the same model. So our performance issue is gone because it will only run single query.
- Example:

```
const postSchema = new mongoose.Schema({
  content: String,
  date: { type: Date, default: Date.now },
  user: {
      name: { type: String, required: true },
      email: String,
      age: Number,
      // All fields of users collection
    },
});
```

- This is great but imagine one user has 20-40 posts and in all those posts we added whole users data. This will cause storage issue because we are duplicating the users data multiple times.
- Another thing is if our user update it's username, then we have to update that detail in all post documents. This might cause data inconsistency but this will not cause performance issue.

🧬 Hybrid Approach

- Hybrid approach is the mix of reference approach and embed approach.
- In hybrid we use both, we add reference also and some data directly which we are going to need in our regular query.
- Example: for social media posts collection, we need to show username and full name of the user with every post. So we do something like this.

```
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number,
});

const postSchema = new mongoose.Schema({
  content: String,
  date: { type: Date, default: Date.now },
  userSummary: {
    name: String,
    profilePicture: String,
  }, // Embed some data which are needed in regular query
  userRef: { type: mongoose.Schema.Types.ObjectId, ref: "User" }, // Re
ference for getting all data
});
```

⏱️ Indexes

- Index is used to make our database query fast. Even mongoDB says that with indexes our database search query can execute 10x faster than before.
- Without index mongoDB use linear search technique(which is search data collection by collection). With index mongoDB use binary search technique. This technique help to find data directly without going to one by one document and that's why with indexes our query works faster.
- Example:

```
// Create index for email field (1 for accending order, -1 for decending or
der)
userSchema.index({ email: 1 });
```

📈 When we apply index

- When we have large collection
- When searching is frequent
- When sorting is common
- When we use fields in find(), update(), delete() methods

🚫 When we don't apply index

- When our collection is small
- When our data is constantly changing

- When we have too many indexes
- When querying many different fields

The ultimate Node JS Course ~ Code Bless You❤️