1. `users` Collection

#single-collection
#inheritance-pattern

bloated document anti pattern => split into 2collections if good

1. **Student**

```
{
  "_id": "ObjectId", // **
  "name": "String", // **
  "email": "String", // Hashed   **************
  "passwordHashed": "String", // Hashed
  "country": "String", // **
  "phoneNumber": "String", // Hashed   **************
  "isEmailVerified": "Boolean", // **
  "googleId": "String", // Optional // **
  "firstLogin": "Boolean", // **
  "profilePictureUrl" : "String", // **
  "role": "String", // "userType" : "" // **
  "createdAt": "Date", // **
  "updatedAt": "Date", // **
  "lastLogin" : "Date", // **
  "status" : "String" //// ** active, suspended (admin approve), deleted
  "deletionRequestedAt" : "Date",
  "lastActionAt": { "comment": "Date", "favorite": "Date", "note": "Date" } ^^
  "displayCalendar": "String", // Enum: ['gregorian', 'hijri']
  `preferredLanguage` : "String",// (for localization)
  `failedLoginAttempts` : "Number",
  `lastFailedLogin` : "Date",
  "studentProfile": {
```

```
    "dob": "Date",
    "parent": "ObjectId" // Optional
    "points" : 100,
    "dailyUsageLimitInMinutes" : 120,
    "elaspedTimeOfTheDayInMinutes" : 60
  }
  }
```

2. **Parent**

```
{
"_id": "ObjectId",
  "name": "String",
  "emailHashed": "String", // Hashed
  "passwordHashed": "String", // Hashed
  "country": "String",
  "phoneNumberHashed": "String", // Hashed
  "isEmailVerified": "Boolean",
  "googleId": "String", // Optional
  "firstLogin": "Boolean",
  "profilePicture" : "String",
  "role": ["String"],
  "createdAt": "Date",
  "updatedAt": "Date",
  "lastActionAt": { "comment": "Date", "favorite": "Date", "note": "Date" }
  "displayCalendar": "String", // Enum: ['gregorian', 'hijri']
  `preferredLanguage` : "String",// (for localization)
  `loginAttempts` : "Number",
  `lastFailedLogin` : "Date"
  "parentProfile": {
    "points" : 100,
    },
```

TODO

[Single Collection Pattern]

3. **ServiceProvider**

   ContentCreator or Specialist

```
"_id": "ObjectId",
"name": "String",
"emailHashed": "String", // Hashed
"passwordHashed": "String", // Hashed
"country": "String",
"phoneNumberHashed": "String", // Hashed
"isEmailVerified": "Boolean",
"googleId": "String", // Optional
"firstLogin": "Boolean",
"profilePicture" : "String",
"role": ["String"],
"createdAt": "Date",
"updatedAt": "Date",
"lastActionAt": { "comment": "Date", "favorite": "Date", "note": "Date" }
"displayCalendar": "String", // Enum: ['gregorian', 'hijri']
`preferredLanguage` : "String",// (for localization)
`loginAttempts` : "Number",
 `lastFailedLogin` : "Date"
"EarnerProfile": { // Exists only if role is 'contentCreator' or 'specialist'
      "displayName": "String",
      "title": "String",
      "bio" : "String",
      "verificationStatus": (Enum: `['pending', 'verified', 'rejected']`)
      "spokenLanguages" : ["String"],
      "expertiseTags": ["String"],
      "socialLinks" : {
```

```
        "youtube" : "string",
        "facebook" : "string",
        "website" : "string"
    },
    "experience": [
        {
            title : "string",
            institution : "string" ,
            year :"string"
        }
    ],
    "servicesOffered": [
        {
            title: String,
            description: String,
            duration_minutes: Number,
            price: Number ,
            "rating" : {
              "reviewCount" : "number",
              "avgRating" : "Number"
          }
        }
    ],
    "reviews" : // subset pattern [need sync]
    [
    {
        "_id": "ObjectId",
          "user" {
              "userId": "ObjectId",
              "userName" : "string" // duplicated [need sync immediately | later]
          },
      "rating": "Number",
      "comment":"string"
```

```
      // (Optional) Ref: 'appointments'. Required if entityType is 'specialist'.
      "appointmentId": "ObjectId",

         "status": "String", // Approved only

          "isEdited": "Boolean",
          "createdAt": "Date",
          "updatedAt": "Date"
        }
        ],

        "averageRating" : 5, // calculated
        "totalStudents" : 1000 // calculated
        "totalConsultations" : 10 // calculated
    },

  }
```

4. **Admin**

```
"_id": "ObjectId",
  "name": "String",
  "emailHashed": "String", // Hashed
  "passwordHashed": "String", // Hashed
  "country": "String",
  "phoneNumberHashed": "String", // Hashed
  "isEmailVerified": "Boolean",
  "googleId": "String", // Optional
  "firstLogin": "Boolean",
  "profilePicture" : "String",
  "role": ["String"],
  "createdAt": "Date",
```

```
  "updatedAt": "Date",
  "lastActionAt": { "comment": "Date", "favorite": "Date", "note": "Date" }
  "displayCalendar": "String", // Enum: ['gregorian', 'hijri']
  `preferredLanguage` : "String",// (for localization)
  `loginAttempts` : "Number",
  `lastFailedLogin` : "Date"
  "adminProfile" :{
      "isSuperAdmin" : "Boolean",
      "twoFactorEnabled" : "Boolean",
      "permissions" :{
      "users": { "read": true, "write": true, "delete": false },
      "content": { "read": true, "write": false, "publish": false },
      "admin": { "read": true, "write": false, "publish": false },
      }
  }

}
```

Indexing

- `{ "emailHashed": 1 }` (Unique) : For login and preventing duplicate accounts
- `{ "phoneNumberHashed": 1 }` (Unique): Same logic as above
- `{ "googleId": 1 }` (Unique, Sparse): Sparse as the field is optional
- `{ "role": 1 }` : multi-key index to find all users of a specific type
- `{ "EarnerProfile.verificationStatus": 1 }` (Sparse): For the admin dashboard to find all earners with "pending" verification (sparse as it only indexes documents that *have* an `EarnerProfile`).
- `{ "EarnerProfile.expertiseTags": 1 }` (Sparse): A multi-key index essential for the "find a specialist by tag" search feature
- `{ "studentProfile.parent": 1 }` (Sparse): Lets a parent dashboard quickly find all linked student accounts

Validation:

- `role` : Enum: ['student', 'parent', 'contentCreator', 'specialist', 'admin']

- `status`: Enum: ['active', 'suspended', 'deleted']
- `EarnerProfile.verificationStatus`: Enum: ['pending', 'verified', 'rejected']
- `studentProfile.points`: Number, minimum 0
- `studentProfile.dailyUsageLimitInMinutes`: Number, minimum 0
- `studentProfile.elapsedTimeOfTheDayInMinutes`: Number, minimum 0
- `parentProfile.points`: Number, minimum 0
- `EarnerProfile.averageRating`: Number, minimum 0, maximum 5
- `EarnerProfile.totalStudents`: Number, minimum 0
- `EarnerProfile.totalConsultations`: Number, minimum 0
- `EarnerProfile.servicesOffered.duration_minutes`: Number, minimum 0
- `EarnerProfile.servicesOffered.price`: Number, minimum 0
- Required fields: ["name", "emailHashed", "passwordHashed", "country", "phoneNumberHashed", "isEmailVerified", "firstLogin", "profilePicture", "role", "createdAt", "updatedAt", "lastLogin", "status"]
- Conditional required fields:
- If `role` includes 'student': ["studentProfile.dob", "studentProfile.religion", "studentProfile.parent", "studentProfile.points", "studentProfile.dailyUsageLimitInMinutes", "studentProfile.elapsedTimeOfTheDayInMinutes"]
- If `role` includes 'parent': ["parentProfile.religion", "parentProfile.points"]
- If `role` includes 'contentCreator' or 'specialist': ["EarnerProfile.displayName", "EarnerProfile.title", "EarnerProfile.bio", "EarnerProfile.verificationStatus", "EarnerProfile.spokenLanguages", "EarnerProfile.expertiseTags", "EarnerProfile.averageRating", "EarnerProfile.totalStudents", "EarnerProfile.totalConsultations"]
- If `role` includes 'admin': ["adminProfile.isSuperAdmin", "adminProfile.twoFactorEnabled", "adminProfile.permissions"]

```
db.runCommand({
  collMod: "users",
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "emailHashed", "role", "status", "createdAt", "updatedAt"],
      properties: {
```

```
        name: {
          bsonType: "string",
          description: "must be a string and is required"
        },
        status: {
          enum: ["active", "suspended", "deleted"],
          description: "can only be one of the enum values"
        },
        role: {
          bsonType: "array",
          items: {
            enum: ["student", "parent", "contentCreator", "specialist", "admin"]
          }
        },
        // TODO : you validate the email in application *before* hashing
      }
    }
  },
  validationAction: "error"
});
```

2. financeAccounts

Only a specific, secure, authenticated-as-owner endpoint (`/api/me/financials`) is given permission to query this separate collection

```
{
    "_id": "fin_acct_123",
    "userId": "user_alex_id",
    "paymentDetails": {
            "paypalEmailEncrypted": "String",
            "bankAccountEncrypted": "Object"
         },
```

```
    "preferredPayoutMethod" : "",
    "payoutSchedule": "monthly",
    "currentBalance": 5300, // calculated
    "revenueShares": [
    { "type": "courseSale", "percent": 70 },
    { "type": "serviceSale", "percent": 80 }
    ],
    "lastAccessAt": "Date", // Track last access
}
```

`currentBalance` is the result of an aggregation query `db.transactions.aggregate([ { $match: { userId: creatorId, status: 'completed' } }, { $group: { _id: "$userId", balance: { $sum: "$netAmount" } } } ])`.
run the query periodically (or via a database trigger/event) and update the `currentBalance` field on the `financeAccounts`

TODO
Computed Pattern

Indexing

- `{ "userId": 1 }` (Unique): **Critical**. This ensures one finance account per user and provides an instant lookup from a user ID

Validation:

- `payoutSchedule` : Enum: ['monthly', 'quarterly']
- `revenueShares.type` : Enum: ['courseSale', 'serviceSale']
- `revenueShares.percent` : Number, minimum 0, maximum 100
- `currentBalance` : Number, minimum 0
- Required fields: ["userId", "paymentDetails", "payoutSchedule", "currentBalance", "revenueShares"]
- add `minimumPayoutThreshold: Number` to your `financeAccounts` schema and add it to the `$jsonSchema` validator (`{ minimum: 10 }` to enforce a $10 minimum)

3. transactions

The immutable Append-only
The single source of truth for all money.

```
{
  "_id": "ObjectId",
  "earnerId": "ObjectId",
  "payerId": "ObjectId",
  "orderId": "ObjectId",


  // Enum: ['sale', 'refund', 'payout', 'adjustment']
  "type": "String",
  // Enum: ['pending', 'completed', 'cleared', 'failed']
  "status": "String",
  "failureReason" : "String"
  "currency": "String",

  "amount": "Number",
  "platformFee": "Number",
  "netAmount": "Number",

 "relatedCourseId": "ObjectId",
 "relatedAppointmentId": "ObjectId",
 "createdAt": "Date"

 // Ref: 'transactions' Used by a 'refund' to point to the original 'sale'.
 "linkedTransactionId": "ObjectId",
 // Links this 'payout' to a processor batch
 "payoutBatchId": "String",

 "transactionHash" : "String" // Fraud Prevention: to detect duplicate transactions.
```

```
  "processedAt" : "Date" // (for payouts)
}
```

Transaction Status

Transaction Type

Special Fields

Indexing

- `{ "earnerId": 1, "status": 1, "createdAt": -1 }` : A compound index to power the Earner's financial dashboard ("Show me all my 'cleared' transactions, newest first").
- `{ "payerId": 1, "createdAt": -1 }` : To power the Student/Parent "My Purchase History" page.
- `{ "orderId": 1 }` : To find all transactions (e.g., the original `sale` and a subsequent `refund`) associated with a single order.
- `{ "status": 1, "type": 1, "createdAt": 1 }` : For background workers. (e.g., "Find all `pending` `payouts` to process" or "Find all `pending` `sales` to clear").

Validation:

- `type` : Enum: ['sale', 'refund', 'payout', 'adjustment']
- `status` : Enum: ['pending', 'completed', 'cleared', 'failed']
- `amount` : Number
- `platformFee` : Number
- `netAmount` : Number
- Required fields: ["earnerId", "payerId", "orderId", "type", "status", "currency", "amount", "platformFee", "netAmount", "createdAt", "transactionHash"]

4. appointments

Manages the scheduling and details for specialist services

```
{
  "_id": "ObjectId",
  "specialistId": "ObjectId",
  "userId": "ObjectId",
  //to prove this was paid for.
  "orderId": "ObjectId",

  // ['pending_payment', 'scheduled', 'completed', 'cancelled_student', 'cancelled_specialist']
  "status": "String",
  "scheduledTime": "Date",
  "timezone" : "string",
  "durationMinutes": "Number",
  "meetingUrl": "String", // Added after scheduling

  //A snapshot of the service details AT THE TIME OF BOOKING
  // This prevents issues if the specialist later changes their prices/services
  "serviceSnapshot": {
    "title": "String",
    "description": "String",
    "price": "Number"
  },

  "notes": "String", // Notes from the student for the specialist
  "createdAt": "Date",
  "updatedAt": "Date"
  "cancellationDeadline" : "Date",
  "notificationStatus" : "String" // field to track whether reminders have been sent
  `cancelledAt` : "Date",
  "`rescheduledAt`" : "Date"
}
```

Indexing

- `{ "specialistId": 1, "scheduledTime": 1 }`: Powers the Specialist's calendar view.
- `{ "userId": 1, "scheduledTime": 1 }`: Powers the Student/Parent's calendar view.
- `{ "orderId": 1 }` (Unique): Critical link to prove payment. An order should only ever map to one appointment.
- `{ "status": 1, "scheduledTime": 1 }`: For background jobs, like finding all `scheduled` appointments tomorrow to send reminder emails.
- `{ "status": 1, "cancellationDeadline": 1 }`: for background jobs (find appointments still cancelable).

Validation:

- `status`: Enum: ['pending_payment', 'scheduled', 'completed', 'cancelled_student', 'cancelled_specialist']
- `notificationStatus`: Enum: ['pending', 'sent', 'failed']
- `durationMinutes`: Number, minimum 0
- `serviceSnapshot.price`: Number, minimum 0
- Required fields: ["specialistId", "userId", "orderId", "status", "scheduledTime", "timezone", "durationMinutes", "serviceSnapshot", "createdAt", "updatedAt", "cancellationDeadline", "notificationStatus"]
- `serviceSnapshot` required fields: ["title", "description", "price"]

5. adminAuditLog

No one should be able to edit or delete logs from the application layer
Deletion should only be possible at the database super-user level
(for data retention/GDPR compliance).

```
{
  "_id": "ObjectId",
  "adminUserId": "ObjectId", //WHO did the action.
  "ipAddress": "String",
// Enum: ['USER_DELETE', 'CREATOR_APPROVE', 'PAYOUT_ISSUED', 'REFUND_MANUAL', ....]
  "action": "String",

  // WHAT was affected
```

```
  "target": {
    "collection": "String", // 'users', 'courses', 'transactions' , ..
    "documentId": "ObjectId"
  },

  "changes": {
    "field": "String",
    "oldValue": "Mixed",
    "newValue": "Mixed"
  },
  "reason": "String", // Optional
  "createdAt": "Date",
 // `requestId` : "String"
}
```

Indexing

- `{ "adminUserId": 1, "createdAt": -1 }` : To review a specific admin's activity.
- `{ "target.collection": 1, "target.documentId": 1 }` : **Extremely important**. This lets you pull the *entire change history for one specific user, course, or transaction.*

Validation:

- `action` : Enum: ['USER_DELETE', 'CREATOR_APPROVE', 'PAYOUT_ISSUED', 'REFUND_MANUAL']
- Required fields: ["adminUserId", "ipAddress", "action", "target", "createdAt"]
- `target` required fields: ["collection", "documentId"]

6. orders

```
{

  "_id": "ObjectId",
```

```
  "userId": "ObjectId",
  "status": "String", // Enum: ['pending', 'completed', 'failed', 'cancelled']
  "lineItems": [
    {
      "itemType": "String", // Enum: ['course', 'service']
      "itemId": "ObjectId", // Ref: 'courses' or 'services'
      "price": "Number", // Price at time of purchase
      "EarnerId" : "ObjectId" // Helps quickly find "which creator earned from this order" without extra lookups
    }
  ],
  "totalAmount": "Number", // Sum of lineItems.price * quantity
  "paymentMethod": "String", // Enum: ['stripe', 'paypal', 'other']
  "paymentStatus": "String", // Enum: ['pending', 'completed', 'failed']
  "transactionId": "ObjectId", // Ref: 'transactions' (links to the payment record)
  "createdAt": "Date",
  "updatedAt": "Date"
},
```

Indexing

- `{ "userId": 1 }`: For user purchase history queries (e.g., "My Orders" page).
- `{ "transactionId": 1 }` (Unique): Links to the transaction record in the 'transactions' collection.
- `{ "status": 1, "createdAt": -1 }`: For order processing and reporting (e.g., "Find all pending orders, newest first").

Validation:

- `status`: Enum: ['pending', 'completed', 'failed', 'cancelled']
- `paymentMethod`: Enum: ['stripe', 'paypal', 'other']
- `paymentStatus`: Enum: ['pending', 'completed', 'failed']
- `lineItems.itemType`: Enum: ['course', 'service']
- `lineItems.price`: Number, minimum 0
- `lineItems.quantity`: Integer, minimum 1
- Required fields: ["userId", "status", "lineItems", "totalAmount", "paymentMethod", "paymentStatus", "createdAt"]

7.carts

```
{
  "_id": "ObjectId",
  "userId": "ObjectId", // Ref: 'users' (student or parent)
  "items": [
    {
      "itemType": "String", // Enum: ['course', 'service']
```

```
      /// extended reference pattern [duplicated]
      "thumbnailUrl": "String",

      "price": "Number",

      "title" : {
          "en" : "",
          "ar" : ""
      }

       "itemId": "ObjectId", // Ref: 'courses' or 'services'

       "addedAt": "Date"

     }

  ],

  "updatedAt": "Date",
  "`expiresAt`" : "Date"

}
```

Indexing

- `{ "userId": 1 }` (Unique): Ensures one cart per user and enables fast lookup for a user's cart.
- { `items.itemId` : 1 }(optional): if you need analytics (e.g., "how many carts include this course").

Validation:

- `items.itemType` : Enum: ['course', 'service']
- Required fields: ["userId", "items", "updatedAt"]

8. enrollments

```
{

  "_id": "ObjectId",

  "studentId": "ObjectId", // Ref: 'users'
  "courseId": "ObjectId", // Ref: 'courses'
  "orderId": "ObjectId", // Ref: 'orders' (Proof of purchase)


  "enrolledAt": "Date",
  "status": "String", // Enum: ['active', 'completed', 'dropped']


  "progressPercentage": "Number", // 0-100
  "completedLessons": ["ObjectId"],
  "pointsEarned": "Number",

// Certificate link
"certificateId": "ObjectId", // Ref: 'certificates' (Set when status is 'completed')
"completedAt": "Date", // Optional: Set when status is 'completed'

`lastAccessedAt` : "Date" // recently active courses
}
```

Indexing

- `{ "studentId": 1 }`: Find all courses for a student (e.g., "My Courses" page).
- `{ "courseId": 1 }`: Find all students enrolled in a course (e.g., for course analytics).
- `{ "studentId": 1, "courseId": 1 }` (Unique): Prevents duplicate enrollments for the same student and course.

Validation:

- `status`: Enum: ['active', 'completed', 'dropped']
- `progress_percent`: Number, minimum 0, maximum 100
- Required fields: ["studentId", "courseId", "enrolledAt", "progress_percent", "status"]

9. favorites

```
{
  "_id": "ObjectId",
  "userId": "ObjectId", // Ref: 'users' (student or parent)
  "courseId": "ObjectId", // Ref: 'courses'
  "createdAt": "Date",
  "status": "String" // Enum: ['active', 'deleted'] for soft deletes
}
```

Indexes

- `{ "userId": 1 }`: Find all favorite courses for a user (e.g., "My Favorites" page).
  - `{ "courseId": 1 }`: Find all users who favorited a course (e.g., for course popularity analytics).
  - `{ "userId": 1, "courseId": 1 }` (Unique): Prevents a user from favoriting the same course multiple times.

Validation:

- `status`: Enum: ['active', 'deleted']
- Required fields: ["userId", "courseId", "createdAt", "status"]

10. certificates

```
{
  "_id": "ObjectId",


  "studentId": "ObjectId",
  "courseId": "ObjectId",
  "instructorId": "ObjectId",

  "issuedAt": "Date",
  "certificateUrl": "String",
 // "verificationCode": "String",


  // A snapshot of key data at the time of issuing.
  // This prevents data from changing if the course or instructor is updated later.
  "snapshot": {
    "courseTitle": "String",
    "instructorName": "String", // The "displayName" of the Earner
    "studentName": "String"
  },

  "createdAt": "Date"
  "expiresAt": "Date", // Optional
  "status": "String", // Enum: ['active', 'expired']
}
```

Indexes

- `{ "studentId": 1 }`: **Essential**. Lets you quickly find all certificates for a specific student's profile.
- `{ "verificationCode": 1 }` (**Unique**): **Essential**. Powers the public-facing verification page (e.g., `.../verify?code=...`).

- `{ "courseId": 1 }`: Lets you find all students who have completed a specific course.
- `{ "instructorId": 1 }`: Lets an instructor see all certificates they have issued.

Issuer

11. notes

```
{
  "_id": "ObjectId",

  "studentId": "ObjectId",
  "courseId": "ObjectId",

  // Optional: For linking a note to a specific lesson (a video)
  "lessonId": "ObjectId",

  "title": "String",
  "content": "String",

  "videoTimestampInSeconds": "Number",

  "createdAt": "Date",
  "updatedAt": "Date",
  //"isPinned" : "Boolean"
}
```

Indexes

- `{ "studentId": 1, "courseId": 1 }`: **Essential compound index.** This powers the main query: "Show me all my notes for this specific course."

- `{ "studentId": 1, "lessonId": 1 }`: A useful index for showing all notes a student took on a specific lesson.

12. comments

```
{
  "_id": "ObjectId",

  "userId": "ObjectId",
  "courseId": "ObjectId",

  // (Optional) If the comment is on a specific lesson, not just the course.
  "lessonId": "ObjectId",


  // - If 'null', it is a top-level, original comment.
  // - If it contains an ObjectId, it is a *reply* to that comment.
  "parentId": "ObjectId",

  "content": "String",

  // Enum: ['pending', 'approved', 'rejected']
  // This allows Admins or Instructors to validate/approve comments
  "status": "String",

  // (Optional) Allows an Instructor to pin a helpful comment
  "isPinned": "Boolean",   // Default: false

  // (Optional) For 'liking' comments
  "likesCount": "Number",  // Default: 0

  "createdAt": "Date",
  "updatedAt": "Date",
  "replyCount" : "Number"
```

```
}
```

Design Explanation

Indexes

- `{ "courseId": 1, "parentId": 1, "status": 1, "createdAt": -1 }` : **Essential.** This compound index is optimized for the main query: finding all approved comments (or replies) for a specific course, sorted by date.
- `{ "lessonId": 1, "parentId": 1, "status": 1 }` : Speeds up finding comments for a *specific lesson.*
- `{ "status": 1, "createdAt": 1 }` : **For moderation.** Powers the Admin/Instructor "pending comments" dashboard, showing the oldest pending comments first.
- `{ "userId": 1 }` : Lets you find all comments posted by a single user.
- { `likesCount` : 1 } : if you plan "top comments" queries

consider caching likesCount in a separate collection or Redis to reduce write contention.

For very deep threads, consider capping the depth (e.g., 3 levels) to simplify queries.

---

13. courses

bloated document anti pattern => split into 2collections for home & for course details

```
{
  "_id": "ObjectId",
  "title": "String",
  "description": "String",
  "thumbnailUrl": "String",

  "moduleId": "ObjectId",
```

```
 "instructorId": "ObjectId",

  "price": "Number",
  "status": "String", // Enum: ['draft', 'published', 'archived']

 "totalHours": "Number",
 "language": "String",

// (Optional) To sort courses *within* a module
"order": "Number",

"createdAt": "Date",
"updatedAt": "Date"

 "finalExam": "ObjectId", // Ref: 'assessments'

 "ageRange": "String", // "10-14"

 "prerequisites": ["ObjectId"], // Ref: 'courses' collection

 "outcome": ["String"], // List of learning outcomes

 "reviews" : [  // subset pattern [duplicated - need sync]
 {
       "_id": "ObjectId",
   "user" {
       "userId": "ObjectId",
       "userName" : "string" // duplicated [need sync immediately | later]
   },
   "rating": "Number",
   "comment": { // Multilingual review content

     "ar": "String",
```

```
        "en": "String"
      },

      // (Optional) Ref: 'enrollments'. Required if entityType is 'course'.
      "enrollmentId": "ObjectId",

     "status": "String", // approved only

      "isEdited": "Boolean",

      "createdAt": "Date",
      "updatedAt": "Date"
      },
    ]و
    "rating" : {
          "reviewCount" : "number",
          "avgRating" : "Number"
    }

    // newAvg = (avgRating * reviewCount + newRating) / (reviewCount + 1)
    // can apply rollup computations  [one update : daily , monthly , ..]
    // can apply approximation pattern when data escales [users scale]

    "attachments": [
      {
        "name": "String",
        "url": "String"
      }
    ],

    "totalViews": "Number", // Increment on course page load
  },
  "comments" : [{ // subset pattern [need sync]
```

```
        "user" : {  // extended ref pattern [need sync]
            "userId": "ObjectId",
            "userName" : "String",
            "userPicture" : ""
        },
      // - If 'null', it is a top-level, original comment.
      // - If it contains an ObjectId, it is a *reply* to that comment.
      "parentId": "ObjectId",

      "content": "String",

      "status": "String",    // approved only

      // (Optional) Allows an Instructor to pin a helpful comment
      "isPinned": "Boolean",   // Default: false

      // (Optional) For 'liking' comments
      "likesCount": "Number",  // Default: 0

      "createdAt": "Date",
      "updatedAt": "Date",
      "replyCount" : "Number"
  }]
  ```

- **Indexing**:

  - `{ "title": "text", "description": "text" }`: For full-text search.

  - `{ "moduleId": 1 }`: To quickly find all courses in a module.

  - `{ "instructorId": 1 }`: To find all courses by a content creator.

  - `{ "price": 1 }`: For sorting by price.
```
```

14. **Lessons**

```js
{
"_id": "ObjectId",
"courseId": "ObjectId",
"instructorId": "ObjectId",

"title": "String",
"lessonType": "String", // Enum: ['video', 'text', 'audio' , 'quiz' , 'exam']


"videoUrl": "String",
"durationInMinutes": "Number",
"content": "String",

"isFreePreview": "Boolean", // Default: false. Can non-enrolled users see it?

//To sort lessons *within* a course
"order": "Number",
"assessmentId": "ObjectId", // Optional, Ref: 'assessments'. Used if lessonType is 'quiz' or 'exam'.

"createdAt": "Date",
"updatedAt": "Date",
"comments" : [{ // subset pattern [need sync]

  "user" : {  // extended ref pattern [need sync]
      "userId": "ObjectId",
        "userName" : "String",
        "userPicture" : ""
  },
```

```
      // - If 'null', it is a top-level, original comment.
      // - If it contains an ObjectId, it is a *reply* to that comment.
      "parentId": "ObjectId",

      "content": "String",

      "status": "String",    // approved only

      // (Optional) Allows an Instructor to pin a helpful comment
      "isPinned": "Boolean",   // Default: false

      // (Optional) For 'liking' comments
      "likesCount": "Number",  // Default: 0

      "createdAt": "Date",
      "updatedAt": "Date",
      "replyCount" : "Number"
  }]
}
```

Indexes:

- `{ "courseId": 1, "order": 1 }` (Essential for fetching all lessons in a course, in order).

Validation:

```
db.createCollection("lessons", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["courseId", "instructorId", "title", "lessonType", "isFreePreview", "createdAt", "updatedAt"],
      properties: {
        lessonType: { enum: ["video", "quiz", "text", "assignment"] },
```

```
        videoUrl: { bsonType: "string" },
        durationInMinutes: { bsonType: "double", minimum: 0 }
      },
      if: { properties: { lessonType: { const: "video" } } },
      then: { required: ["videoUrl", "durationInMinutes"] }
    }
  }
});
```

---

16. Assessments

Instead of separate collections for `Quiz`, `Question`, and `Exam`

```
 {
"_id": "ObjectId",
"title": "String",
"type": "String", // 'quiz' or 'exam'
"course": "ObjectId",
"lesson": "ObjectId",  // Optional. The specific lesson this quiz is for
"questions": [
  {
    "questionText": "String",
    "questionType": "String", // 'mcq', 'true_false', 'short_answer'
    "points": "Number",
    "options": [
      {
        "text": "String",
        "isCorrect": "Boolean"
      }
    ]
  }
```

```
  ] ,

}
```

Module -> Courses -> lessons

17. Modules

```
{
  "_id": "ObjectId",
  "instructorId": "ObjectId",
  "title": "String",
  "description": "String",
  "thumbnailUrl": "String",
  "status": "String", // Enum: ['draft', 'published', 'archived']

  // (Optional) To sort the modules on the instructor's page
  "order": "Number",

  "createdAt": "Date",
  "updatedAt": "Date"
}
```

Indexes: `{ "instructorId": 1 }`, `{ "status": 1 }`

18. Notifications

```
{

  "_id": "ObjectId",

  "userId": "ObjectId",
```

```
  "type": "String", // Enum: ['appointment_reminder', 'course_update', 'comment_reply', 'transaction_update',
'admin_action', 'certificate_issued', 'general']

  "status": "String", // Enum: ['pending', 'sent', 'failed', 'read']

  "priority": "String", // Enum: ['low', 'medium', 'high']

  "title": {

    "ar": "String",

    "en": "String"

  },

  "message": {

    "ar": "String",

    "en": "String"

  },

  "relatedEntity": {

    "collection": "String", // 'appointments', 'transactions', 'comments', 'certificates'

    "documentId": "ObjectId" // Ref: '_id' of the related document

  },

  "deliveryMethods": [ // How the notification is sent
```

```
    {
      "method": "String", // Enum: ['in_app', 'email', ..]
      "status": "String", // Enum: ['pending', 'sent', 'failed']
      "errorMessage": "String" // Optional: Stores failure reason
    }
  ],
  "createdAt": "Date",
  "updatedAt": "Date",
  "readAt": "Date", // Optional: When the user marked it as read
  "expiresAt": "Date" // Optional: For time-sensitive notifications (ex: reminders)
//"actorId": "ObjectId",
}
```

Indexing

```
db.notifications.createIndex({ "userId": 1, "status": 1, "createdAt": -1 }); // For user notification feed

db.notifications.createIndex({ "relatedEntity.collection": 1, "relatedEntity.documentId": 1 }); // For linking to specific entities

db.notifications.createIndex({ "status": 1, "priority": 1, "createdAt": 1 }); // For background jobs processing notifications
```

```
db.notifications.createIndex({ "expiresAt": 1 }, { expireAfterSeconds: 0 }); // TTL index for auto-deleting expired notifications
```

Validation

```
db.createCollection("notifications", {

  validator: {

    $jsonSchema: {

      bsonType: "object",

      required: ["userId", "type", "status", "priority", "title", "message", "deliveryMethods", "createdAt", "updatedAt"],

      properties: {

        userId: { bsonType: "objectId" },

        type: { enum: ["appointment_reminder", "course_update", "comment_reply", "transaction_update", "admin_action", "certificate_issued", "general"] },

        status: { enum: ["pending", "sent", "failed", "read"] },

        priority: { enum: ["low", "medium", "high"] },

        title: {

          bsonType: "object",

          required: ["ar", "en"],
```

```
      properties: {
        ar: { bsonType: "string" },
        en: { bsonType: "string" }
      }
    },
    message: {
      bsonType: "object",
      required: ["ar", "en"],
      properties: {
        ar: { bsonType: "string" },
        en: { bsonType: "string" }
      }
    },
    relatedEntity: {
      bsonType: "object",
      properties: {
        collection: { bsonType: "string" },
```

```
      documentId: { bsonType: "objectId" }
    }
  },
  deliveryMethods: {
    bsonType: "array",
    items: {
      bsonType: "object",
      required: ["method", "status"],
      properties: {
        method: { enum: ["in_app", "email", "push"] },
        status: { enum: ["pending", "sent", "failed"] },
        errorMessage: { bsonType: "string" }
      }
    }
  },
  createdAt: { bsonType: "date" },
  updatedAt: { bsonType: "date" },
```

```
        readAt: { bsonType: "date" },
        expiresAt: { bsonType: "date" }
      }
    }
  }
});
```

Payment Illustration

---

19. Reviews

use singleCollection pattern (course , review , users) || subset pattern (more suitable)

use archive pattern for reviews from 1 year => scheduled task

```
{
  "_id": "ObjectId",
  "user" {
      "userId": "ObjectId",
      "userName" : "string" // duplicated [need sync immediately | later]
  },
  "rating": "Number",
  "comment": { // Multilingual review content

    "ar": "String",
```

```
    "en": "String"
  },
  // "replies" : [{}] #future => outlier pattern

   "target": { // What is being reviewed

    "collection": "String", // Enum: ['courses', 'users'] (users for specialists/contentCreators)

    "documentId": "ObjectId" // Ref: '_id' of course or user (specialist/contentCreator)

  },

  // (Optional) Ref: 'enrollments'. Required if entityType is 'course'.
  "enrollmentId": "ObjectId",

  // (Optional) Ref: 'appointments'. Required if entityType is 'specialist'.
  "appointmentId": "ObjectId",

 "status": "String", // Enum: ['pending', 'approved', 'rejected', 'deleted'] (for moderation and soft delete)

  "isEdited": "Boolean",
  "approvedAt": "Date", // Optional: When approved by admin/instructor

  "rejectionReason": "String"

  "createdAt": "Date",
  "updatedAt": "Date"
}
```

Indexing

- `{ "entityId": 1, "entityType": 1, "status": 1, "createdAt": -1 }` : **Essential compound index.** This powers the main query: "Show me all *approved* reviews for this specific *course* (or *specialist*), newest first."

- `{ "status": 1, "createdAt": 1 }`: For the admin moderation dashboard ("Show me all *pending* reviews, oldest first").
- `{ "enrollmentId": 1 }` (**Unique**, **Sparse**): Ensures a student can only review a specific course enrollment *once*.
- `{ "appointmentId": 1 }` (**Unique**, **Sparse**): Ensures a user can only review a specific appointment *once*.

Validation

- `rating`: Number, minimum 1, maximum 5.
- `entityType`: Enum: `['course', 'specialist']`.
- `status`: Enum: `['pending', 'approved', 'rejected']`.
- **Required fields**: `["userId", "rating", "entityId", "entityType", "status", "createdAt", "updatedAt"]`.
- **Conditional Logic (Application Layer)**:
  - If `entityType` is `'course'`, `enrollmentId` must be present and valid.
  - If `entityType` is `'specialist'`, `appointmentId` must be present and valid.

20. Views

bucket pattern || time series
options :

1. a document for each course => unbounded array or reach the max size
2. a document for each day or week or ....
3. shard the cluster
4. archive or drop old data
5. watch out of massive number of collections anti pattern

```
{
    "_id":
    {
        book_id: "" ,
```

```
        month: "" // data is read monthly
    },
    "views":
    [
     {
         "timestamp": "",
         "user_id": ""
     },
     {
         "timestamp": "" ,
         "user_id": ""
     }
]
```

TODO

bucket Pattern

```
_id:
{
 book_id: "" ,
 month: ""
 },
 views: [
  { timestamp: "" , user_id: "" } ,
  { timestamp: "" , user_id: "" } ,
  ]
```

```
db.getCollection ("views").find({ "id.book_id": 31459, "id.month": ISODate("2023-09-30T00:00:00.000Z) }, { _id: 0,
views_count: { $size: "$views" } } )
```

Advanced Patterns

1. Approximation Pattern :

for highly revieed courses => use RNG
reduce 90% of writes => 4.76 = 4.8

2. Extended Reference Pattern :

reviews page search => User , Course

Schema Versioning :

schema update with no down time

3. Outlier Pattern

handle some documents differently

```
{
outlier : "true"
}
```

TODO

1. **Enum Validation**
   Fields like role, type (in transactions), and status (in appointments) use enums, which is great for consistency. However, ensure your application layer or database schema validation enforces these enums to prevent invalid data entry.

```
// Apply similar validation for transactions.type, appointments.status, etc.
```

```
db.createCollection("users", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["role"],
      properties: {
        role: {
          bsonType: "array",
          items: {
            bsonType: "string",
            enum: ["student", "parent", "contentCreator", "specialist", "admin"]
          }
        }
      }
    }
  }
})
```

2. **Schema Validation**
   Fields like email (regex for valid email format), phoneNumber (country-specific formats)
3. **Encryption**
   Do not store encryption keys in the same database. Use a dedicated service (like AWS Secrets Manager, Google Secret Manager, or HashiCorp Vault) to manage your keys. Application-level encryption (encrypting data *before* sending it to the DB) is the standard for PII and financial details.
   Hashing `email` and `phoneNumber` for uniqueness is acceptable if done with a consistent, non-salted hash like SHA-256 to allow exact lookups, but any PII should ideally be encrypted at rest with secure key management instead of only hashing. This especially applies to `phoneNumberHashed` which may need reversible decryption for certain features like communication .
4. **Balance Consistency**
   Implement a scheduled job or database trigger to periodically validate currentBalance against the transactions collection to catch discrepancies.

A nightly or hourly scheduled job that runs the aggregation and compares it to the cached `currentBalance` is a perfect reconciliation/audit task.

```
const session = db.getMongo().startSession();
session.startTransaction();
try {
  db.transactions.insertOne({ ...transactionData }, { session });
  db.financeAccounts.updateOne(
    { userId: earnerId },
    { $inc: { currentBalance: netAmount } },
    { session }
  );
  await session.commitTransaction();
} catch (error) {
  await session.abortTransaction();
  throw error;
} finally {
  session.endSession();
}
```

5. **Payout Thresholds**
   Add a minimumPayoutThreshold field to prevent small, frequent payouts, which can incur high transaction fees.
6. **Access Control**
   Implement strict role-based access control (RBAC) at the application and database levels. For example, only admins with canManageFinancials should access transactions.
   This will be handled in your application's API middleware. Before any endpoint logic runs, a middleware must check the authenticated user's token, load their user document, and verify their `role` (and `adminProfile.permissions` if they are an admin).
7. **Audit Log Integrity**
   Enforce write-only access to adminAuditLog at the database level. Use MongoDB's auditing features to log database-level actions.

Use MongoDB's native auditing features or a write-only role to ensure adminAuditLog integrity. Additionally, consider logging failed attempts (e.g., unauthorized access) to detect potential security issues.

8. **Soft Deletes & GDPR Compliance**
   Implement soft deletes (e.g., mark users as deleted with a status field) and anonymize PII for deleted accounts to comply with data deletion requests.
   For full GDPR compliance, you'll also need a background script that runs on "deleted" accounts older than (e.g.) 30 days. This script should **anonymize** PII (replace `name`, `emailHashed`, etc., with "DELETED*USER") but _keep the document* so that financial records (like `payerId`) remain intact for auditing.
9. **Versioning**
   Consider adding a schemaVersion field to users to handle future schema migrations.
10. **Atomicity**
    For both examples, use MongoDB transactions to ensure that orders, transactions, financeAccounts, and other collections (enrollments, appointments) are updated atomically. This prevents partial updates in case of failures.
11. **Arabic Consideration**
    1. Encoding

       The single most important step is to use **UTF-8** (or its variant `utf8mb4`) everywhere
       - **Database:** Your database, tables, and text columns should be set to `utf8mb4`. (e.g., in MySQL: `CREATE TABLE users ( name VARCHAR(100) CHARACTER SET utf8mb4 ... )`).
       - **Database Connection:** The connection from your application (e.g., Node.js, Python, PHP) to the database must be configured to use UTF-8.
       - **Application:** Your application logic should read and write files and strings in UTF-8 (this is the default in most modern languages like Python 3 and Node.js).
       - **Frontend:** Your HTML pages must declare UTF-8 in the header: `<meta charset="UTF-8">`.
    2. Database Collation (Sorting & Comparing)

```
// This tells MongoDB to use Arabic sorting rules by default for this collection.
db.createCollection("categories", {
```

```
  collation: { locale: "ar", strength: 2 }
})
```

3. Text Normalization and Diacritics (Tashkeel)
4. Numbers and Dates
    - **Numbers:** Arabic-speaking countries use standard Western numerals (1, 2, 3) but also Eastern Arabic numerals (١، ٢، ٣). If users can input Eastern numerals in a form (like a phone number), your application should **normalize them to Western numerals (0-9)** before validation and storage. It's best to store all numbers in standard `INT` or `BIGINT` formats.
    - **Dates:** While most systems store dates in the standard Gregorian (UTC) format, be aware that for *display*, some users may expect to see dates in the **Hijri (Islamic) calendar**. This is a frontend conversion task, not a storage one.

12. **Full-Text Search (Text Indexes)**

    For your search functionality (like searching course titles or descriptions), a simple query won't be effective. You need a text index, and that index must understand Arabic.

    For better search performance, consider integrating a dedicated search engine like Elasticsearch or Algolia for course search. Alternatively, enhance the MongoDB text index with weights to prioritize title over description.

```
// This creates a text index that is optimized for the Arabic language.


db.courses.createIndex(
  { title: "text", description: "text" },
  { weights: { title: 10, description: 5 }, default_language: "arabic" }
    );
```

13. **Design for Multilingual Content**

```
{
  "_id": "ObjectId",
  "title": {
    "ar": "دورة البرمجة",
```

```
    "en": "Programming Course"
  },
  "description": {
    "ar": "وصف الدورة...",
    "en": "Course description..."
  },
  // ...other course fields
}
```

14. **Handling Denormalized Data:**
    - **Issue**: Fields like snapshot in certificates and serviceSnapshot in appointments are great for preserving data at a point in time, but they could lead to data duplication and inconsistency if not carefully managed. For example, if an instructor's displayName changes, you need to ensure it doesn't affect historical snapshots.
    - **Suggestion**: Implement a background job or trigger to periodically validate snapshots against their source data (e.g., courses.title, users.displayName). Alternatively, consider using a $lookup in queries to pull fresh data when snapshots are not critical for historical accuracy.

```
db.certificates.aggregate([
  { $lookup: { from: "courses", localField: "courseId", foreignField: "_id", as: "course" } },
  { $match: { "snapshot.courseTitle": { $ne: { $arrayElemAt: ["$course.title", 0] } } } }
])
```

15. **Rate Limiting for Sensitive Endpoints:**
    - **Issue**: The financeAccounts collection is accessible only through a secure endpoint, but there's no mention of rate limiting or abuse prevention for sensitive operations like payouts or balance queries.
    - **Suggestion**: Implement rate limiting at the application layer (e.g., using Redis) for endpoints like /api/me/financials to prevent brute-force attacks or excessive queries. Additionally, consider adding a lastAccessAt field to financeAccounts to track and limit frequent access.
16. **Rate Limiting User Actions**:

- **Issue**: There's no mention of limiting user actions like adding favorites, comments, or notes to prevent spam or abuse.
- **Suggestion**: Add rate-limiting logic at the application layer (e.g., max 10 comments per hour per user) and consider a lastActionAt field in users to track activity frequency.

15. **Backup and Recovery**:

- Ensure a robust backup strategy for critical collections like transactions, financeAccounts, and adminAuditLog. Use MongoDB's point-in-time recovery or incremental backups to minimize data loss in case of failures.
- Consider replicating adminAuditLog to a separate, read-only database for additional security.

15. **Performance Monitoring**:

- Use MongoDB's query profiling and performance monitoring tools to identify slow queries, especially for collections like comments and courses that may experience high read/write traffic.
- Example: Enable the MongoDB profiler to log queries taking longer than 100ms:

```
db.setProfilingLevel(1, { slowms: 100 });
```

16. **Data Retention Policies**:

- Define retention periods for collections like adminAuditLog, transactions, and notes to comply with GDPR and reduce storage costs. For example, anonymize or delete notes older than 5 years for inactive users.

```
db.adminAuditLog.deleteMany({ createdAt: { $lt: new Date("2020-01-01") } });
```

17. **Testing and Validation**:

- Add automated tests for schema validation and index performance. Use tools like mongodump and mongorestore to test migrations and ensure indexes are correctly applied.

  Example test for index coverage:

```
db.users.explain("executionStats").find({ role: "student", "studentProfile.parent": "ObjectId" });
```

18. **Scalability for High-Traffic Collections**:

- Collections like transactions, comments, and userCourseProgress may grow large. Consider sharding these collections (e.g., shard transactions by earnerId, comments by courseId) to distribute load across multiple servers.

```
sh.shardCollection("transactions", { earnerId: "hashed" });
```

19. **Analytics Support**:

- Add fields to support analytics, such as totalViews in courses or lastCommentedAt in comments, to track engagement without complex aggregations.

20. **Event-Driven Architecture**:

- Use MongoDB change streams to trigger real-time updates, such as notifying users of new comments or updating financeAccounts.currentBalance when a transaction is completed.

```
const changeStream = db.transactions.watch([
  { $match: { "fullDocument.status": "completed" } }
]);
changeStream.on("change", (change) => {
  db.financeAccounts.updateOne(
    { userId: change.fullDocument.earnerId },
    { $inc: { currentBalance: change.fullDocument.netAmount } }
  );
});
```

21. **Hijri Calendar**:

- For Hijri calendar support, consider adding a displayCalendar field in users to store user preferences (e.g., gregorian or hijri). Use a library like hijri-converter in the frontend to convert scheduledTime or issuedAt fields for display.

How-to: Never store keys in the database. Use **Application-Level Encryption**.

1. Store your encryption key (e.g., a 32-byte string) in a secret manager (like **HashiCorp Vault**, **AWS Secrets Manager**, or **Google Secret Manager**).
2. In your application, load this key into an environment variable (e.g., `process.env.ENCRYPTION_KEY`) during startup. **Do not** hardcode it.
3. When saving data (like `paypalEmailEncrypted`), encrypt it in your code *before* sending it to MongoDB.

Role-Based Access Control (RBAC)

How-to: This is implemented in your application's **middleware**. Before an API endpoint runs, a middleware checks the user's role and permissions.

Audit Log Integrity

How-to: Create a special database role that can *only* write to the `adminAuditLog` collection and cannot modify or delete from it. Your application (or just the admin-action-handling part) logs in as this user.

Here is a practical guide on how to implement every item from your TODO list, including the cross-cutting concerns like soft deletes, versioning, and localization.

1. Validation (Enums, Regex, and Rules)

This ensures data integrity at the database level, acting as your final line of defense.

How-to: Use MongoDB's `$jsonSchema` validator. You can apply this when creating a collection or modify an existing one with `collMod`.

Example (Applying to your `users` collection):

This command modifies the `users` collection to enforce `status` and `role` enums.

JavaScript

```
db.runCommand({
  collMod: "users",
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "emailHashed", "role", "status", "createdAt", "updatedAt"],
      properties: {
        name: {
          bsonType: "string",
          description: "must be a string and is required"
        },
        status: {
          enum: ["active", "suspended", "deleted"],
          description: "can only be one of the enum values"
        },
        role: {
          bsonType: "array",
          items: {
            enum: ["student", "parent", "contentCreator", "specialist", "admin"]
          }
        },
        // For your TODO on regex, you validate the *raw* email in your
        // application *before* hashing and storing it.
        // A regex on a hash isn't meaningful.
      }
    }
  },
```

```
  validationAction: "error" // Reject documents that fail validation
});
```

Payout Thresholds (TODO 5): This is a simple validation rule. Just add `minimumPayoutThreshold: Number` to your `financeAccounts` schema and add it to the `$jsonSchema` validator (e.g., `{ minimum: 10 }` to enforce a $10 minimum).

---

2. Security (Encryption, RBAC, and Audit Logs)

This section covers protecting your data and controlling access.

Encryption (TODO 3)

How-to: Never store keys in the database. Use **Application-Level Encryption**.

1. Store your encryption key (e.g., a 32-byte string) in a secret manager (like **HashiCorp Vault**, **AWS Secrets Manager**, or **Google Secret Manager**).
2. In your application, load this key into an environment variable (e.g., `process.env.ENCRYPTION_KEY`) during startup. **Do not** hardcode it.
3. When saving data (like `paypalEmailEncrypted`), encrypt it in your code *before* sending it to MongoDB.

Node.js Pseudo-code:

JavaScript

```
import crypto from 'crypto';
const ALGO = 'aes-256-cbc';
const KEY = Buffer.from(process.env.ENCRYPTION_KEY, 'hex'); // Load from env

function encrypt(text) {
  const iv = crypto.randomBytes(16);
```

```
  const cipher = crypto.createCipheriv(ALGO, KEY, iv);
  let encrypted = cipher.update(text, 'utf8', 'hex');
  encrypted += cipher.final('hex');
  return `${iv.toString('hex')}:${encrypted}`; // Store IV with text
}

function decrypt(hash) {
  const [iv, encrypted] = hash.split(':');
  const decipher = crypto.createDecipheriv(ALGO, KEY, Buffer.from(iv, 'hex'));
  let decrypted = decipher.update(encrypted, 'hex', 'utf8');
  decrypted += decipher.final('utf8');
  return decrypted;
}

// Usage:
const encryptedPaypal = encrypt("user@example.com");
// Store encryptedPaypal in db.financeAccounts
```

Role-Based Access Control (RBAC) (TODO 6)

How-to: This is implemented in your application's **middleware**. Before an API endpoint runs, a middleware checks the user's role and permissions.

Express.js Pseudo-code:

JavaScript

```
// Middleware to check if user is an admin with 'canManageUsers' permission
const canManageUsers = (req, res, next) => {
  // 1. 'req.user' is populated by your authentication middleware (e.g., from a JWT)
  if (!req.user) {
    return res.status(401).send("Authentication required.");
  }
```

```
  // 2. Check role and specific permission
  const hasPermission =
    req.user.role.includes("admin") &&
    req.user.adminProfile?.permissions?.canManageUsers;

  if (hasPermission) {
    next(); // User is authorized, proceed to the endpoint
  } else {
    res.status(403).send("Forbidden: You do not have permission.");
  }
};

// 3. Apply the middleware to your routes
app.delete("/api/v1/users/:id", canManageUsers, userController.deleteUser);
```

Audit Log Integrity (TODO 7)

How-to: Create a special database role that can *only* write to the `adminAuditLog` collection and cannot modify or delete from it. Your application (or just the admin-action-handling part) logs in as this user.

MongoDB Shell Command:

JavaScript

```
db.createRole({
  role: "auditLogger",
  privileges: [
    {
      resource: { db: "yourDatabaseName", collection: "adminAuditLog" },
      actions: ["insert"] // ONLY insert
    },
    {
```

```
      // May need 'find' on other collections to populate log details
      resource: { db: "yourDatabaseName", collection: "users" },
      actions: ["find"]
    }
  ],
  roles: []
});
```

---

3. Atomicity & Data Consistency (TODOs 4, 10)

This is the most critical part of your financial system. Use **MongoDB transactions** to ensure operations (like a purchase) are "all or nothing."

How-to: The checkout process is the perfect example. It must:

1. Create the `order`.
2. Create the `transaction` (ledger entry).
3. Create the `enrollment` (giving the student access).
4. Increment the earner's `currentBalance` (your TODO 4).

If any step fails, the entire operation must roll back.

Localization :

```
"description": { "en": "String", "ar": "String" },
"outcome": { "en": ["String"], "ar": ["String"] }
```

This applies to `courses`, `modules`, `lessons`, and `assessments`.

Soft Deletes :

Add `status: "String"` and `deletedAt: "Date"` to all collections that users can "delete" (e.g., `courses`, `comments`, `notes`, `users`).

Your "delete" API endpoint doesn't run `deleteOne()`. It runs `updateOne()`.

```
// User deletes their comment
db.comments.updateOne(
  { _id: commentId, userId: userId }, // Ensure user owns the comment
  { $set: { status: "deleted", deletedAt: new Date() } }
);
```

All your application queries must now *globally* filter out deleted documents.

```
db.comments.find({ courseId: "course_abc", status: { $ne: "deleted" } // GLOBAL FILTER });
```

GDPR Anonymization (The "Hard Delete"): This is the background job from your TODO. It finds documents soft-deleted (e.g., 30 days ago) and permanently anonymizes/scrubs PII to comply with "right to be forgotten" requests. This *preserves* financial integrity (the `payerId` in `transactions` still exists) but *destroys* PII.

```
// A nightly job
const thirtyDaysAgo = new Date(Date.now() - 30 * 24 * 60 * 60 * 1000);

db.users.updateMany(
  {
    status: "deleted",
    deletionRequestedAt: { $lt: thirtyDaysAgo }
  },
  {
    $set: {
```

```
      name: "[DELETED_USER]",
      emailHashed: "[DELETED]",
      phoneNumberHashed: "[DELETED]",
      googleId: "[DELETED]",
      profilePicture: "default_deleted_avatar.png",
      // ... scrub any other PII
    },
    $unset: {
      studentProfile: "", // Remove entire sub-document
      parentProfile: "",
      EarnerProfile: ""
      // ... unset other sensitive data
    }
  }
);
```

APP :

discriminator pattern so the MongoDB C# Driver knows which class to instantiate for a given document based on the `role` field.