Task Manager Backend – Line-by-Line Explanation

Files: db.js, routes/tasks.js, models/Task.js
Generated on 2025-09-03 23:10:37

1) db.js (connectDB)

```
001  import mongoose from "mongoose";
002
003  export async function connectDB() {
004    const uri = process.env.MONGODB_URI || "mongodb://127.0.0.1:27017/task_manager";
005    mongoose.set("strictQuery", false);
006    await mongoose.connect(uri);
007    console.log("■ MongoDB connected");
008  }
```

Explanations

```
001  import mongoose from "mongoose";
```
Import the Mongoose library (ES module syntax). Mongoose is an ODM (Object Data Modeling) library used to interact with MongoDB from Node.js.

```
002
```
Blank line for readability.

```
003  export async function connectDB() {
```
Export an async function named `connectDB` so other modules can call it to establish a DB connection.

```
004    const uri = process.env.MONGODB_URI || "mongodb://127.0.0.1:27017/task_manager";
```
Inside the function, read the MongoDB connection URI from the environment variable `MONGODB_URI`. If not present, fall back to a local MongoDB URI pointing at a `task_manager` database on the default port.

```
005    mongoose.set("strictQuery", false);
```
Set Mongoose's `strictQuery` option to `false`. This controls whether unknown fields are allowed in query filters; setting to false avoids deprecation warnings and keeps behavior compatible with older Mongoose versions.

```
006    await mongoose.connect(uri);
```
Call `mongoose.connect(uri)` to open the connection to MongoDB. `await` ensures the function waits until the connection is established (or fails) before proceeding.

```
007    console.log("■ MongoDB connected");
```
Log a confirmation message to the console once connected. The checkmark emoji is decorative and helps quickly spot a successful connection in logs.

```
008  }
```
Closing brace of the async function.

2) routes/tasks.js (Express Router for Tasks)

```
import { Router } from "express";
import Task from "../models/Task.js";

const router = Router();

// CREATE
router.post("/", async (req, res, next) => {
  try {
    const { title, description = "", status } = req.body;
    const task = await Task.create({ title, description, status });
    res.status(201).json(task);
  } catch (err) {
    next(err);
  }
});

// READ (list) with search & status filter
// GET /tasks?q=keyword&status=pending|in-progress|completed
router.get("/", async (req, res, next) => {
  try {
    const { q, status } = req.query;
    const filter = {};

    if (status && ["pending", "in-progress", "completed"].includes(status)) {
      filter.status = status;
    }

    if (q && q.trim()) {
      // Regex search on title OR description (case-insensitive)
      filter.$or = [
        { title: new RegExp(q, "i") },
        { description: new RegExp(q, "i") }
      ];
      // If you prefer text search (requires index above):
      // filter.$text = { $search: q };
    }

    const tasks = await Task.find(filter).sort({ createdAt: -1 });
    res.json(tasks);
  } catch (err) {
    next(err);
  }
});

// READ (single)
router.get("/:id", async (req, res, next) => {
  try {
    const t = await Task.findById(req.params.id);
    if (!t) return res.status(404).json({ error: "Task not found" });
    res.json(t);
  } catch (err) {
    next(err);
  }
});

// UPDATE (full)
router.put("/:id", async (req, res, next) => {
  try {
    const { title, description, status } = req.body;
    const t = await Task.findByIdAndUpdate(
      req.params.id,
      { title, description, status },
      { new: true, runValidators: true }
    );
    if (!t) return res.status(404).json({ error: "Task not found" });
    res.json(t);
  } catch (err) {
    next(err);
069    }
070  });
071
072  // UPDATE status only
073  router.patch("/:id/status", async (req, res, next) => {
074    try {
075      const { status } = req.body;
076      if (!["pending", "in-progress", "completed"].includes(status)) {
077        return res.status(400).json({ error: "Invalid status" });
078      }
079      const t = await Task.findByIdAndUpdate(
080        req.params.id,
081        { status },
082        { new: true, runValidators: true }
083      );
084      if (!t) return res.status(404).json({ error: "Task not found" });
085      res.json(t);
086    } catch (err) {
087      next(err);
088    }
089  });
090
091  // DELETE
092  router.delete("/:id", async (req, res, next) => {
093    try {
094      const t = await Task.findByIdAndDelete(req.params.id);
095      if (!t) return res.status(404).json({ error: "Task not found" });
096      res.json({ message: "Deleted", id: t._id });
097    } catch (err) {
098      next(err);
099    }
100  });
101
102  export default router;
```

Explanations

```
001  import { Router } from "express";
```
Import `Router` from Express using ES module syntax. Router lets you create modular route handlers.

```
002  import Task from "../models/Task.js";
```
Import the Task model (Mongoose model) which provides DB operations for the `tasks` collection.

```
003
```
Create a new router instance. We'll attach request handlers to this router.

```
004  const router = Router();
```
Blank line for readability.

```
005
```
Comment indicating the CREATE route section.

```
006  // CREATE
```
Handle POST requests to root of this router (`/`). This creates a new task. The handler is async so we can `await` DB ops.

```
007  router.post("/", async (req, res, next) => {
```
Start of a try block to catch errors and forward them to the Express error middleware.

```
008    try {
```
Destructure `title`, `description`, and `status` from the incoming JSON body; default `description` to empty string if not provided.

```
009      const { title, description = "", status } = req.body;
```
Create a new Task document using Mongoose's `create` helper. This writes the task to MongoDB.

```
010      const task = await Task.create({ title, description, status });
```
Respond with status 201 (Created) and the newly created task object as JSON.

```
011      res.status(201).json(task);
```
Catch block to forward any error to `next(err)` — Express will pass it to the error handler middleware.

```
  } catch (err) {
```
Close the POST route handler.
```
    next(err);
```
Blank line for readability.
```
  }
```
Comment for READ (list) route which supports search and status filter.
```
});
```
Comment showing expected query parameters for the route.
```

```
Handle GET requests to `/` (list tasks).
```
// READ (list) with search & status filter
```
Open try block.
```
// GET /tasks?q=keyword&status=pending|in-progress|completed
```
Extract query parameters `q` (search) and `status` from the request's query string.
```
router.get("/", async (req, res, next) => {
```
Initialize an empty `filter` object which will be passed to Mongoose `find`.
```
  try {
```
If a valid `status` value is provided (pending, in-progress, completed), set `filter.status` so the query will return only tasks with that status.
```
    const { q, status } = req.query;
```
Blank line for readability.
```
    const filter = {};
```
If `q` exists and is not just whitespace, build a `$or` regex-based filter to search title or description case-insensitively.
```

```
Set `filter.$or` to an array of conditions: title matches the regex or description matches the regex.
```
    if (status && ["pending", "in-progress", "completed"].includes(status)) {
```
Comment noting an alternative approach using MongoDB text indexes and `$text` search, which requires an index on the fields.
```
      filter.status = status;
```
Close the `if (q)` block.
```
    }
```
Execute the Mongoose `find` with the constructed `filter`, sorting results by `createdAt` descending (newest first).
```

```
Send the found tasks back to the client as JSON.
```
    if (q && q.trim()) {
```
Catch block to forward any error to the error middleware.
```
      // Regex search on title OR description (case-insensitive)
```
Close the GET list route handler.
```
      filter.$or = [
```
Blank line for readability.
```
        { title: new RegExp(q, "i") },
```
Comment indicating the READ single-task route.
```
        { description: new RegExp(q, "i") }
```
Handle GET requests to `/:id` to fetch a single task by its MongoDB `_id`.
```
      ];
```
Open try block.
```
      // If you prefer text search (requires index above):
```
Use `Task.findById` with `req.params.id` to retrieve a single document by its ID.
```
      // filter.$text = { $search: q };
```
If no task is found (`t` is falsy), return a 404 response with a JSON error message.
```
    }
```

Otherwise return the found task as JSON.

```
037
```
Catch block to forward errors to the error middleware.

```
038      const tasks = await Task.find(filter).sort({ createdAt: -1 });
```
Close the GET single route.

```
039      res.json(tasks);
```
Blank line for readability.

```
040    } catch (err) {
```
Comment indicating the UPDATE (full) route.

```
041      next(err);
```
Handle PUT requests to `/:id` to update title, description, and status all at once.

```
042    }
```
Open try block.

```
043  });
```
Destructure `title`, `description`, and `status` from the request body.

```
044
```
Call `Task.findByIdAndUpdate` with the provided id and update object. The `new: true` option returns the updated document; `runValidators: true` ensures schema validators run on the updated values.

```
045  // READ (single)
```
If no document was found to update, return 404 with an error message.

```
046  router.get("/:id", async (req, res, next) => {
```
Return the updated task as JSON.

```
047    try {
```
Catch block to forward errors.

```
048      const t = await Task.findById(req.params.id);
```
Close the PUT route handler.

```
049      if (!t) return res.status(404).json({ error: "Task not found" });
```
Blank line for readability.

```
050      res.json(t);
```
Comment indicating a PATCH route that only updates status.

```
051    } catch (err) {
```
Handle PATCH requests to `/:id/status` to change just the status field.

```
052      next(err);
```
Start try block.

```
053    }
```
Extract the `status` field from the request body.

```
054  });
```
Validate the provided status; if it's not one of the allowed values, respond with 400 Bad Request and an error message.

```
055
```
Perform `findByIdAndUpdate` to set only the `status` on the document. Keep `new: true` and `runValidators: true` as before.

```
056  // UPDATE (full)
```
If document not found, return 404.

```
057  router.put("/:id", async (req, res, next) => {
```
Return the updated task object as JSON.

```
058    try {
```
Catch block to forward errors to the error middleware.

```
059      const { title, description, status } = req.body;
```
Close the PATCH route.

```
060      const t = await Task.findByIdAndUpdate(
```
Blank line for readability.

```
      req.params.id,
```
Comment indicating the DELETE route.

```
      { title, description, status },
```
Handle DELETE requests to `/:id` to remove a task from the DB.

```
      { new: true, runValidators: true }
```
Open try block.

```
    );
```
Call `Task.findByIdAndDelete` to remove the document by its id.

```
    if (!t) return res.status(404).json({ error: "Task not found" });
```
If nothing was deleted (document not found), respond 404 with an error.

```
    res.json(t);
```
Otherwise return a JSON message confirming deletion and the deleted document id.

```
  } catch (err) {
```
Catch block to forward errors.

```
    next(err);
```
Close the DELETE route handler.

```
  }
```
Blank line for readability.

```
});
```
Export the configured router as the module default so it can be mounted by the main server (`app.use('/tasks', tasksRouter)`).

```

```
(No further explanation provided for this line.)

```
// UPDATE status only
```
(No further explanation provided for this line.)

```
router.patch("/:id/status", async (req, res, next) => {
```
(No further explanation provided for this line.)

```
  try {
```
(No further explanation provided for this line.)

```
    const { status } = req.body;
```
(No further explanation provided for this line.)

```
    if (!["pending", "in-progress", "completed"].includes(status)) {
```
(No further explanation provided for this line.)

```
      return res.status(400).json({ error: "Invalid status" });
```
(No further explanation provided for this line.)

```
    }
```
(No further explanation provided for this line.)

```
    const t = await Task.findByIdAndUpdate(
```
(No further explanation provided for this line.)

```
      req.params.id,
```
(No further explanation provided for this line.)

```
      { status },
```
(No further explanation provided for this line.)

```
      { new: true, runValidators: true }
```
(No further explanation provided for this line.)

```
    );
```
(No further explanation provided for this line.)

```
    if (!t) return res.status(404).json({ error: "Task not found" });
```
(No further explanation provided for this line.)

```
    res.json(t);
```

(No further explanation provided for this line.)

```
  } catch (err) {
```
(No further explanation provided for this line.)

```
    next(err);
```
(No further explanation provided for this line.)

```
  }
```
(No further explanation provided for this line.)

```
});
```
(No further explanation provided for this line.)

```

```
(No further explanation provided for this line.)

```
// DELETE
```
(No further explanation provided for this line.)

```
router.delete("/:id", async (req, res, next) => {
```
(No further explanation provided for this line.)

```
  try {
```
(No further explanation provided for this line.)

```
    const t = await Task.findByIdAndDelete(req.params.id);
```
(No further explanation provided for this line.)

```
    if (!t) return res.status(404).json({ error: "Task not found" });
```
(No further explanation provided for this line.)

```
    res.json({ message: "Deleted", id: t._id });
```
(No further explanation provided for this line.)

```
  } catch (err) {
```
(No further explanation provided for this line.)

```
    next(err);
```
(No further explanation provided for this line.)

```
  }
```
(No further explanation provided for this line.)

```
});
```
(No further explanation provided for this line.)

```

```
(No further explanation provided for this line.)

```
export default router;
```
(No further explanation provided for this line.)

3) models/Task.js (Mongoose Task Model)

```
001  import mongoose from "mongoose";
002
003  const TaskSchema = new mongoose.Schema(
004    {
005      title: { type: String, required: true, trim: true },
006      description: { type: String, default: "", trim: true },
007      status: {
008        type: String,
009        enum: ["pending", "in-progress", "completed"],
010        default: "pending",
011        index: true
012      }
013    },
014    { timestamps: true } // adds createdAt & updatedAt
015  );
016
017  // Optional: supports text search via $text (we’ll use regex in routes for simplicity)
018  TaskSchema.index({ title: "text", description: "text" });
019
020  export default mongoose.model("Task", TaskSchema);
```

Explanations

`001  import mongoose from "mongoose";`

Import Mongoose library (ES module syntax). We'll use it to define schemas and models.

`002`

Blank line for readability.

`003  const TaskSchema = new mongoose.Schema(`

Create a new `TaskSchema` using `mongoose.Schema`. The schema defines the shape and validation rules for documents in the `tasks` collection.

`004    {`

Open the schema `fields` object.

`005      title: { type: String, required: true, trim: true },`

Define `title` as a string, required, and `trim: true` to remove whitespace at the ends.

`006      description: { type: String, default: "", trim: true },`

Define `description` as a string with default empty string and `trim: true` to keep data tidy.

`007      status: {`

Define `status` field with nested configuration:

`008        type: String,`

Set the type to `String` for `status`.

`009        enum: ["pending", "in-progress", "completed"],`

Use `enum` to restrict values to only the allowed strings: pending, in-progress, completed.

`010        default: "pending",`

Set the default status to `'pending'` when not provided.

`011        index: true`

Add `index: true` on status to make queries filtering by status faster.

`012      }`

Close the `status` field object.

`013    },`

Close the fields object and add schema options: `{ timestamps: true }` instructs Mongoose to automatically add `createdAt` and `updatedAt` timestamps.

`014    { timestamps: true } // adds createdAt & updatedAt`

Close the `new mongoose.Schema(...)` call.

`015  );`

Comment noting that a text index is optional but useful for `$text` searches; in this code we use RegExp for search instead.

```
016
```
Create a text index on `title` and `description` so MongoDB's `$text` operator can be used for text search if desired.

```
017  // Optional: supports text search via $text (we’ll use regex in routes for simplicity)
```
Export the compiled model named `Task` using `mongoose.model`. This creates (or references) a collection named `tasks` in MongoDB.

```
018  TaskSchema.index({ title: "text", description: "text" });
```
(No further explanation provided for this line.)

```
019
```
(No further explanation provided for this line.)

```
020  export default mongoose.model("Task", TaskSchema);
```
(No further explanation provided for this line.)