🐳 DockerManager – Full Code Explained in Small Chunks (Beginner Friendly)

This guide explains your entire DockerManager file in **small code snippets**, followed by **simple explanations**.

Take it slowly. Read one chunk at a time.

🐓 Importing Required Libraries

```
import Docker from "dockerode";
```

What this does:

- Imports the `dockerode` library.
- This library lets your Node.js app talk to Docker.
- Without this, you cannot create or manage containers from code.

```
import { v4 as uuidv4 } from "uuid";
```

What this does:

- Imports a function that creates random unique IDs.
- We use it to generate unique container names.

```
import { pool } from "@/config/pgPool";
```

What this does:

- Imports PostgreSQL connection pool.
- Used to store container session data in database.

```
import tar from "tar-fs";
import path from "path";
import fs from "fs";
```

What these do:

- `tar-fs` → Used to pack files into tar format (needed to build Docker images).
- `path` → Helps work with folder paths safely.
- `fs` → Lets you check if folders/files exist.

🐕 Creating Docker Connection

```
const docker = new Docker();
```

Explanation:

- Connects to your local Docker engine.
- Now your app can create containers.

🐗 Configuration Variables

```
const MAX_CONTAINERS = parseInt(process.env.MAX_CONTAINERS || "3", 10);
```

Explanation:

- Maximum containers allowed at the same time.
- Default is 3 if environment variable not set.

```
const IMAGE_NAME = "custom-node-sandbox:latest";
```

- Name of the Docker image we build.

```
const BASE_IMAGE = "node:18-alpine";
```

- Lightweight Node.js base image.

```
const DOCKERFILE_PATH = path.resolve(__dirname, "../../sandbox-image");
```

- Location of folder containing Dockerfile.

🐚 Defining ContainerInfo Interface

```
interface ContainerInfo {
  containerId: string;
  userId: number;
  createdAt: Date;
}
```

Explanation:

Defines structure of stored container information.

🐜 Active Containers Map

```
const activeContainers = new Map<number, ContainerInfo>();
```

Explanation:

- Stores running containers in memory.
- Key = userId
- Value = container details

🐝 followProgress() – Track Docker Build Progress

```
private static followProgress(stream: NodeJS.ReadableStream) {
  return new Promise<void>((resolve, reject) => {
    docker.modem.followProgress(
      stream,
      (err: any) => (err ? reject(err) : resolve()),
      (event: any) => {
        if (event.status) console.log(`📦 ${event.status}`);
      }
    );
  });
}
```

What it does:

- Watches Docker pull/build process.
- Prints progress messages.

- Resolves when complete.

🐟 pullImage()

```
private static async pullImage(image: string) {
  const stream = await docker.pull(image);
  await this.followProgress(stream);
}
```

What it does:

- Downloads image from Docker Hub.
- Waits until fully downloaded.

🐠 buildImage()

```
private static async buildImage() {
  if (!fs.existsSync(DOCKERFILE_PATH)) {
    throw new Error(`Dockerfile path not found: ${DOCKERFILE_PATH}`);
  }

  const tarStream = tar.pack(DOCKERFILE_PATH);

  const stream = await new Promise<NodeJS.ReadableStream>(
    (resolve, reject) => {
      docker.buildImage(tarStream, { t: IMAGE_NAME }, (err, stream) => {
        if (err) return reject(err);
        if (!stream) return reject(new Error("Build stream undefined"));
        resolve(stream);
      });
    }
  );

  await this.followProgress(stream);
}
```

What it does:

1. Checks if Dockerfile folder exists.
2. Packs folder as tar.
3. Builds Docker image.
4. Waits for build completion.

🐡 ensureImage()

```
static async ensureImage(): Promise<void> {
  const images = await docker.listImages();

  const exists = images.some((img) =>
    img.RepoTags?.some((tag) => tag === IMAGE_NAME)
  );

  if (exists) {
    console.log("🔗 Sandbox image already exists");
    return;
  }

  console.log("🛬 Pulling base image...");
  await this.pullImage(BASE_IMAGE);

  console.log("🔨 Building sandbox image...");
  await this.buildImage();

  console.log("🔗 Sandbox image built successfully");
}
```

What it does:

- Checks if image already built.
- If not, pulls base image.
- Builds custom sandbox image.

🌈 createContainer()

```
static async createContainer(userId: number): Promise<string> {
```

Step 1: Prevent duplicates

```
if (activeContainers.has(userId)) {
  throw new Error("User already has an active container");
}
```

Step 2: Limit maximum containers

```
if (!(await this.canCreateContainer())) {
  throw new Error(`Maximum ${MAX_CONTAINERS} containers reached`);
}
```

Step 3: Ensure image exists

```
await this.ensureImage();
```

Step 4: Create container with restrictions

```
const container = await docker.createContainer({
  Image: IMAGE_NAME,
  Cmd: ["tail", "-f", "/dev/null"],
  WorkingDir: "/workspace",
  HostConfig: {
    Memory: 512 * 1024 * 1024,
    MemorySwap: 512 * 1024 * 1024,
    CpuQuota: 50000,
    NetworkMode: "none",
    PidsLimit: 64,
  },
});
```

Security Limits:

- 512MB RAM
- Limited CPU
- No internet
- Max 64 processes

Step 5: Start container

```
await container.start();
```

Step 6: Save in memory

```
activeContainers.set(userId, {
  containerId,
```

```
  userId,
  createdAt: new Date(),
});
```

Step 7: Save in database

```
await pool.query(
  `INSERT INTO user_sessions (user_id, container_id)
   VALUES ($1, $2)
   ON CONFLICT (user_id)
   DO UPDATE SET container_id = $2, last_activity = CURRENT_TIMESTAMP`,
  [userId, containerId]
);
```

🐓🐓 executeCode()

```
const base64Code = Buffer.from(code, "utf8").toString("base64");
```

Why Base64?

To safely pass multi-line code into shell.

Write file inside container

```
Cmd: [
  "sh",
  "-c",
  `echo '${base64Code}' | base64 -d > /workspace/index.js`,
],
```

Run the file

```
Cmd: ["node", "/workspace/index.js"],
```

🐓🐕 runExec()

Handles execution and timeout.

```
if (timeoutMs > 0) {
  setTimeout(() => {
    stderr += `\nExecution timeout`;
    stream.destroy();
  }, timeoutMs);
}
```

Prevents infinite loops.

🐓🐗 stopContainer()

```
await container.stop({ t: 5 });
await container.remove({ force: true });
```

Stops and deletes container.

Removes from: - Memory map - Database

🐓🐚 cleanupAll()

Loops through all users and stops their containers.

Used during shutdown.

🐓🐜 Graceful Shutdown

```
process.on("SIGINT", async () => {
  await DockerManager.cleanupAll();
  process.exit(0);
});
```

When server stops: - Clean containers - Exit safely

Final Flow

User → Create Container
→ Run Code
→ Get Output
→ Stop Container

🌀 What This System Achieves

- Secure code execution
- Resource isolation
- Database tracking
- Cleanup safety

If you want next, I can: - Add architecture diagram - Add security deep explanation - Add Redis scaling explanation - Convert this into production-ready documentation

You're building something powerful. Keep going ⏳