task-management-system/

│

├── user-service/

│ ├── Dockerfile

│ ├── index.js

│ ├── package.json

│ └── routes/

│ └── userRoutes.js

│

├── task-service/

│ ├── Dockerfile

│ ├── index.js

│ ├── package.json

│ └── routes/

│ └── taskRoutes.js

│

├── assignment-service/

│ ├── Dockerfile

│ ├── index.js

│ ├── package.json

│ └── routes/

│ └── assignmentRoutes.js

│

├── notification-service/

│ ├── Dockerfile

│ ├── index.js

│ ├── package.json

│ └── routes/

│ └── notificationRoutes.js

│

├── dashboard-service/

│ ├── Dockerfile

│ ├── index.js

│ ├── package.json

│ └── routes/

│ └── dashboardRoutes.js

│

├── docker-compose.yml

└── README.md

Task 1: Define Microservices

User Management - Handles user registration, authentication, and profiles.

Task Creation - Manages task creation.

Task Assignment - Assigns tasks to users.

Notifications - Sends notifications about task updates.

Task Dashboard - Displays tasks assigned to users.

npm i express body-parser jsonwebtoken

Here's a step-by-step guide to creating a `user-service` module for your Task Management System using Node.js, Express, and MongoDB. This service will handle user registration, authentication, and profile management.

### Step 1: Create the Project Structure

Create a folder named `user-service` and navigate into it:

```bash

mkdir user-service

cd user-service

```

### Step 2: Initialize Node.js Project

Run the following command to create a `package.json` file:

```bash

npm init -y

```

### Step 3: Install Required Packages

Install the necessary packages:

```bash

npm install express mongoose bcryptjs jsonwebtoken body-parser dotenv

```

### Step 4: Create Folder Structure

Create the following directories and files:

```bash

mkdir models routes config

touch index.js .env

```

### Step 5: Set Up Environment Variables

Open the `.env` file and add your environment variables:

```plaintext

PORT=3000

MONGODB\_URI=mongodb://localhost:27017/user-service

JWT\_SECRET=your\_jwt\_secret

```

### Step 6: Create the User Model

Create a file named `User.js` in the `models` directory:

#### `models/User.js`

```javascript

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

username: {

type: String,

required: true,

unique: true,

},

password: {

type: String,

required: true,

},

});

module.exports = mongoose.model('User', userSchema);

```

### Step 7: Create Routes for User Management

Create a file named `userRoutes.js` in the `routes` directory:

#### `routes/userRoutes.js`

```javascript

const express = require('express');

const bcrypt = require('bcryptjs');

const jwt = require('jsonwebtoken');

const User = require('../models/User');

const router = express.Router();

// Register user

router.post('/register', async (req, res) => {

const { username, password } = req.body;

try {

const hashedPassword = await bcrypt.hash(password, 10);

const newUser = new User({ username, password: hashedPassword });

await newUser.save();

res.status(201).json({ message: 'User registered successfully' });

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// User login

router.post('/login', async (req, res) => {

const { username, password } = req.body;

try {

const user = await User.findOne({ username });

if (!user) return res.status(404).json({ message: 'User not found' });

const isPasswordValid = await bcrypt.compare(password, user.password);

if (!isPasswordValid) return res.status(401).json({ message: 'Invalid credentials' });

const token = jwt.sign({ id: user.\_id }, process.env.JWT\_SECRET, { expiresIn: '1h' });

res.json({ token });

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Get user profile

router.get('/profile', async (req, res) => {

const token = req.headers.authorization?.split(' ')[1];

if (!token) return res.status(403).json({ message: 'No token provided' });

try {

const decoded = jwt.verify(token, process.env.JWT\_SECRET);

const user = await User.findById(decoded.id, '-password');

if (!user) return res.status(404).json({ message: 'User not found' });

res.json(user);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

module.exports = router;

```

### Step 8: Set Up Express Server

Now, open the `index.js` file and set up the Express server:

#### `index.js`

```javascript

const express = require('express');

const mongoose = require('mongoose');

const bodyParser = require('body-parser');

const userRoutes = require('./routes/userRoutes');

require('dotenv').config();

const app = express();

app.use(bodyParser.json());

app.use('/api/users', userRoutes);

const PORT = process.env.PORT || 3000;

// Connect to MongoDB

mongoose.connect(process.env.MONGODB\_URI, { useNewUrlParser: true, useUnifiedTopology: true })

.then(() => {

console.log('MongoDB connected');

app.listen(PORT, () => {

console.log(`User service running on port ${PORT}`);

});

})

.catch(err => console.error(err));

```

### Step 9: Create Dockerfile

Create a file named `Dockerfile` in the `user-service` directory:

#### `Dockerfile`

```dockerfile

FROM node:14

WORKDIR /usr/src/app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "index.js"]

```

### Step 10: Create Docker Compose Configuration (Optional)

If you want to run this service with MongoDB using Docker Compose, create a `docker-compose.yml` file in the root project directory:

#### `docker-compose.yml`

```yaml

version: '3'

services:

user-service:

build: ./user-service

ports:

- "3000:3000"

environment:

- MONGODB\_URI=mongodb://mongo:27017/user-service

- JWT\_SECRET=your\_jwt\_secret

depends\_on:

- mongo

mongo:

image: mongo

ports:

- "27017:27017"

```

### Step 11: Build and Run the Service

If you're using Docker Compose, run the following command:

```bash

docker-compose up --build

```

If you are not using Docker, run the service directly:

```bash

node index.js

```

### Step 12: Test the User Service

You can test the API endpoints using tools like Postman or curl.

1. \*\*Register a User:\*\*

```http

POST http://localhost:3000/api/users/register

Content-Type: application/json

{

"username": "testuser",

"password": "password123"

}

```

2. \*\*Login a User:\*\*

```http

POST http://localhost:3000/api/users/login

Content-Type: application/json

{

"username": "testuser",

"password": "password123"

}

```

3. \*\*Get User Profile:\*\*

After logging in, use the token received in the response to access the user's profile:

```http

GET http://localhost:3000/api/users/profile

Authorization: Bearer <your\_token\_here>

```

### Conclusion

You now have a fully functional `user-service` module for your Task Management System. This service can handle user registration, authentication, and profile retrieval. You can expand upon this foundation by adding more features, such as password reset functionality or email verification, as needed.

Here's a step-by-step guide to creating a `task-service` module for your Task Management System using Node.js, Express, and MongoDB. This service will handle task creation, retrieval, assignment, and management.

### Step 1: Create the Project Structure

Create a folder named `task-service` and navigate into it:

```bash

mkdir task-service

cd task-service

```

### Step 2: Initialize Node.js Project

Run the following command to create a `package.json` file:

```bash

npm init -y

```

### Step 3: Install Required Packages

Install the necessary packages:

```bash

npm install express mongoose body-parser dotenv

```

### Step 4: Create Folder Structure

Create the following directories and files:

```bash

mkdir models routes config

touch index.js .env

```

### Step 5: Set Up Environment Variables

Open the `.env` file and add your environment variables:

```plaintext

PORT=3000

MONGODB\_URI=mongodb://localhost:27017/task-service

```

### Step 6: Create the Task Model

Create a file named `Task.js` in the `models` directory:

#### `models/Task.js`

```javascript

const mongoose = require('mongoose');

const taskSchema = new mongoose.Schema({

title: {

type: String,

required: true,

},

description: {

type: String,

required: true,

},

assignedTo: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User',

required: true,

},

status: {

type: String,

enum: ['Pending', 'In Progress', 'Completed'],

default: 'Pending',

},

createdAt: {

type: Date,

default: Date.now,

},

});

module.exports = mongoose.model('Task', taskSchema);

```

### Step 7: Create Routes for Task Management

Create a file named `taskRoutes.js` in the `routes` directory:

#### `routes/taskRoutes.js`

```javascript

const express = require('express');

const Task = require('../models/Task');

const router = express.Router();

// Create a new task

router.post('/', async (req, res) => {

const { title, description, assignedTo } = req.body;

try {

const newTask = new Task({ title, description, assignedTo });

await newTask.save();

res.status(201).json(newTask);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Get all tasks

router.get('/', async (req, res) => {

try {

const tasks = await Task.find().populate('assignedTo', 'username');

res.json(tasks);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Get task by ID

router.get('/:id', async (req, res) => {

try {

const task = await Task.findById(req.params.id).populate('assignedTo', 'username');

if (!task) return res.status(404).json({ message: 'Task not found' });

res.json(task);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Update task

router.put('/:id', async (req, res) => {

const { title, description, assignedTo, status } = req.body;

try {

const updatedTask = await Task.findByIdAndUpdate(

req.params.id,

{ title, description, assignedTo, status },

{ new: true }

);

if (!updatedTask) return res.status(404).json({ message: 'Task not found' });

res.json(updatedTask);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Delete task

router.delete('/:id', async (req, res) => {

try {

const deletedTask = await Task.findByIdAndDelete(req.params.id);

if (!deletedTask) return res.status(404).json({ message: 'Task not found' });

res.json({ message: 'Task deleted successfully' });

} catch (error) {

res.status(500).json({ error: error.message });

}

});

module.exports = router;

```

### Step 8: Set Up Express Server

Now, open the `index.js` file and set up the Express server:

#### `index.js`

```javascript

const express = require('express');

const mongoose = require('mongoose');

const bodyParser = require('body-parser');

const taskRoutes = require('./routes/taskRoutes');

require('dotenv').config();

const app = express();

app.use(bodyParser.json());

app.use('/api/tasks', taskRoutes);

const PORT = process.env.PORT || 3000;

// Connect to MongoDB

mongoose.connect(process.env.MONGODB\_URI, { useNewUrlParser: true, useUnifiedTopology: true })

.then(() => {

console.log('MongoDB connected');

app.listen(PORT, () => {

console.log(`Task service running on port ${PORT}`);

});

})

.catch(err => console.error(err));

```

### Step 9: Create Dockerfile

Create a file named `Dockerfile` in the `task-service` directory:

#### `Dockerfile`

```dockerfile

FROM node:14

WORKDIR /usr/src/app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "index.js"]

```

### Step 10: Create Docker Compose Configuration (Optional)

If you want to run this service with MongoDB using Docker Compose, create a `docker-compose.yml` file in the root project directory:

#### `docker-compose.yml`

```yaml

version: '3'

services:

task-service:

build: ./task-service

ports:

- "3000:3000"

environment:

- MONGODB\_URI=mongodb://mongo:27017/task-service

depends\_on:

- mongo

mongo:

image: mongo

ports:

- "27017:27017"

```

### Step 11: Build and Run the Service

If you're using Docker Compose, run the following command:

```bash

docker-compose up --build

```

If you are not using Docker, run the service directly:

```bash

node index.js

```

### Step 12: Test the Task Service

You can test the API endpoints using tools like Postman or curl.

1. \*\*Create a New Task:\*\*

```http

POST http://localhost:3000/api/tasks

Content-Type: application/json

{

"title": "Test Task",

"description": "This is a test task",

"assignedTo": "<user\_id\_here>"

}

```

2. \*\*Get All Tasks:\*\*

```http

GET http://localhost:3000/api/tasks

```

3. \*\*Get Task by ID:\*\*

```http

GET http://localhost:3000/api/tasks/<task\_id\_here>

```

4. \*\*Update a Task:\*\*

```http

PUT http://localhost:3000/api/tasks/<task\_id\_here>

Content-Type: application/json

{

"title": "Updated Task Title",

"description": "Updated description",

"assignedTo": "<user\_id\_here>",

"status": "In Progress"

}

```

5. \*\*Delete a Task:\*\*

```http

DELETE http://localhost:3000/api/tasks/<task\_id\_here>

```

### Conclusion

You now have a fully functional `task-service` module for your Task Management System. This service can handle task creation, retrieval, updating, and deletion. You can expand upon this foundation by adding more features, such as task filtering by user or status, as needed.

Here's a step-by-step guide to creating an `assignment-service` module for your Task Management System using Node.js, Express, and MongoDB. This service will manage the assignment of tasks to users.

### Step 1: Create the Project Structure

Create a folder named `assignment-service` and navigate into it:

```bash

mkdir assignment-service

cd assignment-service

```

### Step 2: Initialize Node.js Project

Run the following command to create a `package.json` file:

```bash

npm init -y

```

### Step 3: Install Required Packages

Install the necessary packages:

```bash

npm install express mongoose body-parser dotenv

```

### Step 4: Create Folder Structure

Create the following directories and files:

```bash

mkdir models routes config

touch index.js .env

```

### Step 5: Set Up Environment Variables

Open the `.env` file and add your environment variables:

```plaintext

PORT=3000

MONGODB\_URI=mongodb://localhost:27017/assignment-service

```

### Step 6: Create the Assignment Model

Create a file named `Assignment.js` in the `models` directory:

#### `models/Assignment.js`

```javascript

const mongoose = require('mongoose');

const assignmentSchema = new mongoose.Schema({

taskId: {

type: mongoose.Schema.Types.ObjectId,

ref: 'Task',

required: true,

},

userId: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User',

required: true,

},

assignedAt: {

type: Date,

default: Date.now,

},

});

module.exports = mongoose.model('Assignment', assignmentSchema);

```

### Step 7: Create Routes for Assignment Management

Create a file named `assignmentRoutes.js` in the `routes` directory:

#### `routes/assignmentRoutes.js`

```javascript

const express = require('express');

const Assignment = require('../models/Assignment');

const router = express.Router();

// Create a new assignment

router.post('/', async (req, res) => {

const { taskId, userId } = req.body;

try {

const newAssignment = new Assignment({ taskId, userId });

await newAssignment.save();

res.status(201).json(newAssignment);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Get all assignments

router.get('/', async (req, res) => {

try {

const assignments = await Assignment.find().populate('taskId', 'title').populate('userId', 'username');

res.json(assignments);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Get assignment by ID

router.get('/:id', async (req, res) => {

try {

const assignment = await Assignment.findById(req.params.id).populate('taskId', 'title').populate('userId', 'username');

if (!assignment) return res.status(404).json({ message: 'Assignment not found' });

res.json(assignment);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Update assignment

router.put('/:id', async (req, res) => {

const { taskId, userId } = req.body;

try {

const updatedAssignment = await Assignment.findByIdAndUpdate(

req.params.id,

{ taskId, userId },

{ new: true }

);

if (!updatedAssignment) return res.status(404).json({ message: 'Assignment not found' });

res.json(updatedAssignment);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Delete assignment

router.delete('/:id', async (req, res) => {

try {

const deletedAssignment = await Assignment.findByIdAndDelete(req.params.id);

if (!deletedAssignment) return res.status(404).json({ message: 'Assignment not found' });

res.json({ message: 'Assignment deleted successfully' });

} catch (error) {

res.status(500).json({ error: error.message });

}

});

module.exports = router;

```

### Step 8: Set Up Express Server

Now, open the `index.js` file and set up the Express server:

#### `index.js`

```javascript

const express = require('express');

const mongoose = require('mongoose');

const bodyParser = require('body-parser');

const assignmentRoutes = require('./routes/assignmentRoutes');

require('dotenv').config();

const app = express();

app.use(bodyParser.json());

app.use('/api/assignments', assignmentRoutes);

const PORT = process.env.PORT || 3000;

// Connect to MongoDB

mongoose.connect(process.env.MONGODB\_URI, { useNewUrlParser: true, useUnifiedTopology: true })

.then(() => {

console.log('MongoDB connected');

app.listen(PORT, () => {

console.log(`Assignment service running on port ${PORT}`);

});

})

.catch(err => console.error(err));

```

### Step 9: Create Dockerfile

Create a file named `Dockerfile` in the `assignment-service` directory:

#### `Dockerfile`

```dockerfile

FROM node:14

WORKDIR /usr/src/app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "index.js"]

```

### Step 10: Create Docker Compose Configuration (Optional)

If you want to run this service with MongoDB using Docker Compose, create a `docker-compose.yml` file in the root project directory:

#### `docker-compose.yml`

```yaml

version: '3'

services:

assignment-service:

build: ./assignment-service

ports:

- "3000:3000"

environment:

- MONGODB\_URI=mongodb://mongo:27017/assignment-service

depends\_on:

- mongo

mongo:

image: mongo

ports:

- "27017:27017"

```

### Step 11: Build and Run the Service

If you're using Docker Compose, run the following command:

```bash

docker-compose up --build

```

If you are not using Docker, run the service directly:

```bash

node index.js

```

### Step 12: Test the Assignment Service

You can test the API endpoints using tools like Postman or curl.

1. \*\*Create a New Assignment:\*\*

```http

POST http://localhost:3000/api/assignments

Content-Type: application/json

{

"taskId": "<task\_id\_here>",

"userId": "<user\_id\_here>"

}

```

2. \*\*Get All Assignments:\*\*

```http

GET http://localhost:3000/api/assignments

```

3. \*\*Get Assignment by ID:\*\*

```http

GET http://localhost:3000/api/assignments/<assignment\_id\_here>

```

4. \*\*Update an Assignment:\*\*

```http

PUT http://localhost:3000/api/assignments/<assignment\_id\_here>

Content-Type: application/json

{

"taskId": "<new\_task\_id\_here>",

"userId": "<new\_user\_id\_here>"

}

```

5. \*\*Delete an Assignment:\*\*

```http

DELETE http://localhost:3000/api/assignments/<assignment\_id\_here>

```

### Conclusion

You now have a fully functional `assignment-service` module for your Task Management System. This service can handle the creation, retrieval, updating, and deletion of task assignments. You can expand upon this foundation by adding more features, such as filtering assignments by user or task, as needed.

Here’s a step-by-step guide to creating a `notification-service` module for your Task Management System using Node.js, Express, and MongoDB. This service will handle sending notifications to users about task assignments, updates, or any other relevant events.

### Step 1: Create the Project Structure

Create a folder named `notification-service` and navigate into it:

```bash

mkdir notification-service

cd notification-service

```

### Step 2: Initialize Node.js Project

Run the following command to create a `package.json` file:

```bash

npm init -y

```

### Step 3: Install Required Packages

Install the necessary packages:

```bash

npm install express mongoose body-parser dotenv nodemailer

```

### Step 4: Create Folder Structure

Create the following directories and files:

```bash

mkdir models routes config

touch index.js .env

```

### Step 5: Set Up Environment Variables

Open the `.env` file and add your environment variables:

```plaintext

PORT=3000

MONGODB\_URI=mongodb://localhost:27017/notification-service

EMAIL\_USER=your\_email@example.com

EMAIL\_PASS=your\_email\_password

```

### Step 6: Create the Notification Model

Create a file named `Notification.js` in the `models` directory:

#### `models/Notification.js`

```javascript

const mongoose = require('mongoose');

const notificationSchema = new mongoose.Schema({

userId: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User',

required: true,

},

message: {

type: String,

required: true,

},

isRead: {

type: Boolean,

default: false,

},

createdAt: {

type: Date,

default: Date.now,

},

});

module.exports = mongoose.model('Notification', notificationSchema);

```

### Step 7: Create Routes for Notification Management

Create a file named `notificationRoutes.js` in the `routes` directory:

#### `routes/notificationRoutes.js`

```javascript

const express = require('express');

const Notification = require('../models/Notification');

const nodemailer = require('nodemailer');

const router = express.Router();

// Setup Nodemailer transport

const transporter = nodemailer.createTransport({

service: 'gmail',

auth: {

user: process.env.EMAIL\_USER,

pass: process.env.EMAIL\_PASS,

},

});

// Create a new notification

router.post('/', async (req, res) => {

const { userId, message } = req.body;

try {

const newNotification = new Notification({ userId, message });

await newNotification.save();

// Send email notification

const mailOptions = {

from: process.env.EMAIL\_USER,

to: userId, // Assuming userId is the email address

subject: 'New Notification',

text: message,

};

transporter.sendMail(mailOptions, (error, info) => {

if (error) {

console.log(error);

return res.status(500).json({ error: 'Failed to send email' });

}

res.status(201).json(newNotification);

});

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Get all notifications for a user

router.get('/:userId', async (req, res) => {

try {

const notifications = await Notification.find({ userId: req.params.userId });

res.json(notifications);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Mark notification as read

router.put('/:id/read', async (req, res) => {

try {

const updatedNotification = await Notification.findByIdAndUpdate(

req.params.id,

{ isRead: true },

{ new: true }

);

if (!updatedNotification) return res.status(404).json({ message: 'Notification not found' });

res.json(updatedNotification);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Delete notification

router.delete('/:id', async (req, res) => {

try {

const deletedNotification = await Notification.findByIdAndDelete(req.params.id);

if (!deletedNotification) return res.status(404).json({ message: 'Notification not found' });

res.json({ message: 'Notification deleted successfully' });

} catch (error) {

res.status(500).json({ error: error.message });

}

});

module.exports = router;

```

### Step 8: Set Up Express Server

Now, open the `index.js` file and set up the Express server:

#### `index.js`

```javascript

const express = require('express');

const mongoose = require('mongoose');

const bodyParser = require('body-parser');

const notificationRoutes = require('./routes/notificationRoutes');

require('dotenv').config();

const app = express();

app.use(bodyParser.json());

app.use('/api/notifications', notificationRoutes);

const PORT = process.env.PORT || 3000;

// Connect to MongoDB

mongoose.connect(process.env.MONGODB\_URI, { useNewUrlParser: true, useUnifiedTopology: true })

.then(() => {

console.log('MongoDB connected');

app.listen(PORT, () => {

console.log(`Notification service running on port ${PORT}`);

});

})

.catch(err => console.error(err));

```

### Step 9: Create Dockerfile

Create a file named `Dockerfile` in the `notification-service` directory:

#### `Dockerfile`

```dockerfile

FROM node:14

WORKDIR /usr/src/app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "index.js"]

```

### Step 10: Create Docker Compose Configuration (Optional)

If you want to run this service with MongoDB using Docker Compose, create a `docker-compose.yml` file in the root project directory:

#### `docker-compose.yml`

```yaml

version: '3'

services:

notification-service:

build: ./notification-service

ports:

- "3000:3000"

environment:

- MONGODB\_URI=mongodb://mongo:27017/notification-service

- EMAIL\_USER=your\_email@example.com

- EMAIL\_PASS=your\_email\_password

depends\_on:

- mongo

mongo:

image: mongo

ports:

- "27017:27017"

```

### Step 11: Build and Run the Service

If you're using Docker Compose, run the following command:

```bash

docker-compose up --build

```

If you are not using Docker, run the service directly:

```bash

node index.js

```

### Step 12: Test the Notification Service

You can test the API endpoints using tools like Postman or curl.

1. \*\*Create a New Notification:\*\*

```http

POST http://localhost:3000/api/notifications

Content-Type: application/json

{

"userId": "user\_email@example.com",

"message": "You have been assigned a new task."

}

```

2. \*\*Get All Notifications for a User:\*\*

```http

GET http://localhost:3000/api/notifications/user\_email@example.com

```

3. \*\*Mark Notification as Read:\*\*

```http

PUT http://localhost:3000/api/notifications/<notification\_id>/read

```

4. \*\*Delete a Notification:\*\*

```http

DELETE http://localhost:3000/api/notifications/<notification\_id>

```

### Conclusion

You now have a fully functional `notification-service` module for your Task Management System. This service can handle creating, retrieving, updating, and deleting notifications. Additionally, it can send email notifications to users when tasks are assigned or updated. You can expand upon this foundation by adding more features, such as different types of notifications or integration with other messaging services.

Creating a `dashboard-service` module for your Task Management System involves building an API to fetch and display relevant data, such as tasks, assignments, and notifications. This service can aggregate information from other services and provide a dashboard overview.

Here's a step-by-step guide to creating a `dashboard-service` module using Node.js, Express, and MongoDB.

### Step 1: Create the Project Structure

Create a folder named `dashboard-service` and navigate into it:

```bash

mkdir dashboard-service

cd dashboard-service

```

### Step 2: Initialize Node.js Project

Run the following command to create a `package.json` file:

```bash

npm init -y

```

### Step 3: Install Required Packages

Install the necessary packages:

```bash

npm install express mongoose axios body-parser dotenv

```

### Step 4: Create Folder Structure

Create the following directories and files:

```bash

mkdir routes config

touch index.js .env

```

### Step 5: Set Up Environment Variables

Open the `.env` file and add your environment variables:

```plaintext

PORT=3000

TASK\_SERVICE\_URL=http://localhost:3001/api/tasks

ASSIGNMENT\_SERVICE\_URL=http://localhost:3002/api/assignments

NOTIFICATION\_SERVICE\_URL=http://localhost:3003/api/notifications

```

### Step 6: Create Dashboard Routes

Create a file named `dashboardRoutes.js` in the `routes` directory:

#### `routes/dashboardRoutes.js`

```javascript

const express = require('express');

const axios = require('axios');

const router = express.Router();

// Get dashboard data

router.get('/', async (req, res) => {

try {

const [tasks, assignments, notifications] = await Promise.all([

axios.get(process.env.TASK\_SERVICE\_URL),

axios.get(process.env.ASSIGNMENT\_SERVICE\_URL),

axios.get(process.env.NOTIFICATION\_SERVICE\_URL),

]);

const dashboardData = {

tasks: tasks.data,

assignments: assignments.data,

notifications: notifications.data,

};

res.json(dashboardData);

} catch (error) {

console.error(error);

res.status(500).json({ error: 'Error fetching dashboard data' });

}

});

module.exports = router;

```

### Step 7: Set Up Express Server

Now, open the `index.js` file and set up the Express server:

#### `index.js`

```javascript

const express = require('express');

const mongoose = require('mongoose');

const bodyParser = require('body-parser');

const dashboardRoutes = require('./routes/dashboardRoutes');

require('dotenv').config();

const app = express();

app.use(bodyParser.json());

app.use('/api/dashboard', dashboardRoutes);

const PORT = process.env.PORT || 3000;

// Connect to MongoDB (if needed for local storage or caching)

mongoose.connect(process.env.MONGODB\_URI || 'mongodb://localhost:27017/dashboard-service', {

useNewUrlParser: true,

useUnifiedTopology: true

})

.then(() => {

console.log('MongoDB connected');

app.listen(PORT, () => {

console.log(`Dashboard service running on port ${PORT}`);

});

})

.catch(err => console.error(err));

```

### Step 8: Create Dockerfile

Create a file named `Dockerfile` in the `dashboard-service` directory:

#### `Dockerfile`

```dockerfile

FROM node:14

WORKDIR /usr/src/app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "index.js"]

```

### Step 9: Create Docker Compose Configuration (Optional)

If you want to run this service with other services using Docker Compose, create a `docker-compose.yml` file in the root project directory:

#### `docker-compose.yml`

```yaml

version: '3'

services:

dashboard-service:

build: ./dashboard-service

ports:

- "3000:3000"

environment:

- TASK\_SERVICE\_URL=http://task-service:3001/api/tasks

- ASSIGNMENT\_SERVICE\_URL=http://assignment-service:3002/api/assignments

- NOTIFICATION\_SERVICE\_URL=http://notification-service:3003/api/notifications

depends\_on:

- task-service

- assignment-service

- notification-service

task-service:

image: task-service-image # Replace with the actual image name if using Docker

ports:

- "3001:3001"

assignment-service:

image: assignment-service-image # Replace with the actual image name if using Docker

ports:

- "3002:3002"

notification-service:

image: notification-service-image # Replace with the actual image name if using Docker

ports:

- "3003:3003"

```

### Step 10: Build and Run the Service

If you're using Docker Compose, run the following command:

```bash

docker-compose up --build

```

If you are not using Docker, run the service directly:

```bash

node index.js

```

### Step 11: Test the Dashboard Service

You can test the API endpoint using tools like Postman or curl.

1. \*\*Get Dashboard Data:\*\*

```http

GET http://localhost:3000/api/dashboard

```

### Conclusion

You now have a fully functional `dashboard-service` module for your Task Management System. This service aggregates data from the task, assignment, and notification services, providing a unified view of the system's state. You can expand upon this foundation by adding more features, such as filtering or aggregating data further, based on user requirements.