Solar System Application for Children

This is a full-stack Node.js application designed to educate children about the solar system in an interactive and engaging way. It features a backend API built with Express.js and MongoDB, and a simple frontend using HTML, CSS, and JavaScript to visualize the solar system.

Table of Contents

Features

Interactive Solar System Visualization: A dynamic, rotating solar system displayed on the frontend.

Planet Data API: Backend endpoints to fetch and search for planet information.

MongoDB Integration: Stores planet data persistently in a MongoDB database.

Kubernetes Ready: Displays the pod name if deployed in a Kubernetes environment.

Comprehensive Documentation: Detailed setup and usage instructions.

Test Coverage: Includes unit/integration tests for the backend.

Workflow Explained

The application follows a client-server architecture:

Frontend (Browser):

When the index.html loads, it initializes a canvas for the solar system animation.

JavaScript (public/script.js) makes fetch requests to the Node.js backend to retrieve planet data.

It also handles user interactions, such as viewing planets by range or searching for a specific planet.

The frontend dynamically updates the displayed planet information and visualizes them on the canvas.

It also fetches and displays the Kubernetes pod name from a dedicated backend endpoint.

Backend (Node.js/Express):

The server.js file sets up an Express.js server.

It connects to a MongoDB database (configured via environment variables).

It defines several API endpoints:

/api/planets: Fetches all planets or a specific range of planets from the database.

/api/planets/search: Searches for planets by name using a query parameter.

/api/podname: Returns the name of the Kubernetes pod (if applicable) or a default message.

The backend interacts with MongoDB using Mongoose, defining a Planet schema to structure the data.

MongoDB (Database):

Stores information about each planet (name, description, orbital radius, speed, color).

The application is designed to work with a cloud-hosted MongoDB instance (e.g., MongoDB Atlas). Initial data can be seeded into the database.

Prerequisites
Before you begin, ensure you have the following installed:

Node.js: Download & Install Node.js (LTS version recommended).

npm: Node Package Manager (comes with Node.js).

MongoDB Atlas Account: A free tier account is sufficient for this project. Sign up for MongoDB Atlas.

MongoDB Atlas Setup
Follow these steps to set up your cloud MongoDB database:

Create a MongoDB Atlas Account:

Go to MongoDB Atlas and sign up for a free account.

Follow the prompts to create a new Shared Cluster (the free M0 tier).

Create a Database User:

In your Atlas project, navigate to Database Access under the Security section.

Click Add New Database User.

Choose Password as the Authentication Method.

Enter a strong Username and Password (remember these, you'll need them for your .env file).

Grant Read and write to any database privileges.

Click Add User.

Configure Network Access:

Navigate to Network Access under the Security section.

Click Add IP Address.

For local development, you can choose Add Current IP Address or Allow Access from Anywhere (for testing purposes, but less secure for production).

Click Confirm.

Find Your Connection String:

Go back to Databases in the left navigation.

Click the Connect button for your cluster.

Choose Connect your application.

Select Node.js and copy the connection string. It will look something like this:

mongodb+srv://:@cluster0.abcde.mongodb.net/?retryWrites=true&w=majority

IMPORTANT: Replace and with the database user credentials you created in step 2. Also, ensure the database name in the connection string matches what you intend to use (e.g., /solar_system_db?).

Local Setup Guide
Backend Setup
Clone the repository (or create the files manually):

If you have a git repo

git clone
cd solar-system-app

If you're creating files manually, make sure your project structure matches the Project Structure section.

Navigate to the backend directory:

cd # e.g., cd solar-system-app

Create a .env file:

In the root of your project directory (where server.js is), create a file named .env.

Add your MongoDB Atlas connection string and port:

MONGO_URI=mongodb+srv://<your_username>:
<your_password>@cluster0.abcde.mongodb.net/solar_system_db?retryWrites=true&w=majority
PORT=3000

Remember to replace <your_username> and <your_password> with your actual MongoDB Atlas credentials.

Install backend dependencies:

npm install

Frontend Setup
The frontend is served statically by the Node.js backend, so there are no separate frontend installation steps once the backend is set up.

Running the Application
Start the backend server:

npm start

You should see a message like Server running on port 3000 and MongoDB connected... in your console.

Open your web browser:

Navigate to http://localhost:3000.

You should now see the Solar System application in your browser.

API Endpoints
The backend exposes the following API endpoints:

GET /api/planets

Description: Retrieves a list of all planets.

Query Parameters:

start (optional): Starting index for planets (0-based).

end (optional): Ending index for planets (exclusive).

Example: http://localhost:3000/api/planets?start=0&end=3 (returns the first 3 planets)

Response: [{ planet1_data }, { planet2_data }, ...]

GET /api/planets/search

Description: Searches for planets by name.

Query Parameters:

name (required): The name or part of the name of the planet to search for.

Example: http://localhost:3000/api/planets/search?name=mars

Response: [{ planet_data_matching_search }]

GET /api/podname

Description: Returns the name of the Kubernetes pod the application is running on.

Response: { "podName": "your-pod-name" } or { "podName": "Not running in Kubernetes or POD_NAME environment variable not set." }

Running Tests & Code Coverage
This project uses Jest for testing and nyc for code coverage.

Install dev dependencies (if not already installed):

npm install --save-dev jest supertest nyc

Run tests:

npm test

Generate code coverage report:

npm run coverage

This will generate a coverage/ directory with an HTML report that you can open in your browser (coverage/lcov-report/index.html) to view detailed coverage information.

Kubernetes Pod Name Display
The application attempts to read the POD_NAME environment variable, which is typically set automatically when deployed to Kubernetes.

How it works:

The backend's /api/podname endpoint checks for process.env.POD_NAME.

The frontend fetches this value and displays it at the bottom of the page.

To test locally (simulate Kubernetes):
You can set the POD_NAME environment variable before starting the server:

Linux/macOS:

POD_NAME=my-local-solar-app-pod npm start

Windows (Command Prompt):

set POD_NAME=my-local-solar-app-pod && npm start

Windows (PowerShell):

$env:POD_NAME="my-local-solar-app-pod"; npm start

Enhancing with Planet Images
The current solar system visualization uses colored circles for planets. To make it more attractive, informative, and interactive, you can replace these with actual planet images.

Steps to add images:

Prepare your images:

Find high-quality images for each planet (e.g., sun.png, mercury.png, venus.png, earth.png, mars.png, jupiter.png, saturn.png, uranus.png, neptune.png).

Ensure they have transparent backgrounds (PNG format is ideal).

Place these images in the public/images/ directory (you might need to create this directory).

Modify public/script.js:

Locate the drawPlanet function or the section where planets are drawn.

Instead of drawing ctx.arc, you will load and draw an image.

Here's a conceptual example of how you'd modify the drawPlanet function:

// Inside public/script.js, where planets are defined or drawn

// Add a 'texture' property to your planet data in data/planets.json
// For example:
// {
// "name": "Earth",
// "description": "Our home planet...",
// "radius": 15, // Visual radius on canvas
// "orbitalRadius": 150,
// "orbitalSpeed": 0.01,
// "color": "#007bff",
// "imageSrc": "/images/earth.png" // <-- Add this
// }

// Modify the drawPlanet function in public/script.js
function drawPlanet(ctx, planet, angle) {
const x = SUN_CENTER_X + planet.orbitalRadius * Math.cos(angle);
const y = SUN_CENTER_Y + planet.orbitalRadius * Math.sin(angle);

```
// --- COMMENT OUT THE FOLLOWING LINE TO USE IMAGES INSTEAD OF CIRCLES ---
// ctx.fillStyle = planet.color;
// ctx.beginPath();
// ctx.arc(x, y, planet.radius, 0, Math.PI * 2);
// ctx.fill();
// ctx.closePath();
```

```
    // ---------------------------------------------------------------------

    // --- UNCOMMENT AND MODIFY THE FOLLOWING SECTION TO ADD PLANET IMAGES ---
    if (planet.image) { // Assuming 'planet.image' is an Image object loaded
    beforehand
        const imageSize = planet.radius * 2; // Adjust size as needed
        ctx.drawImage(planet.image, x - planet.radius, y - planet.radius,
    imageSize, imageSize);
    } else {
        // Fallback to drawing a circle if image is not loaded
        ctx.fillStyle = planet.color;
        ctx.beginPath();
        ctx.arc(x, y, planet.radius, 0, Math.PI * 2);
        ctx.fill();
        ctx.closePath();
    }
    // ---------------------------------------------------------------------

    // Draw planet name (optional, adjust position if image is used)
    ctx.fillStyle = '#fff';
    ctx.font = '10px Arial';
    ctx.textAlign = 'center';
    ctx.fillText(planet.name, x, y + planet.radius + 12);
```

}

// You'll also need to pre-load images before drawing.
// In public/script.js, where you fetch planet data:
// After fetching planets, iterate through them and load images:
/*
async function fetchAndRenderPlanets() {
try {
const response = await fetch('/api/planets');
let planetsData = await response.json();

```
    // Load images for each planet
    for (const planet of planetsData) {
        if (planet.imageSrc) {
            const img = new Image();
            img.src = planet.imageSrc;
            await new Promise(resolve => {
                img.onload = () => {
                    planet.image = img; // Store the Image object
                    resolve();
                };
                img.onerror = () => {
                    console.error('Failed to load image:', planet.imageSrc);
                    resolve(); // Resolve even on error to not block
```

```
                };
            });
        }
    }
    // Now planetsData has the loaded image objects
    // Proceed with rendering: animate(planetsData);
} catch (error) {
    console.error('Error fetching or loading planets:', error);
}
```

}
*/

Important Note: Make sure the imageSrc path in your planets.json (or wherever you define planet data) correctly points to the images in public/images/.

Project Structure
solar-system-app/
├── public/
│ ├── index.html # Frontend HTML structure
│ ├── style.css # Frontend CSS for solar system animation
│ ├── script.js # Frontend JavaScript logic
│ └── images/ # Directory for planet images (create this)
│ ├── sun.png
│ ├── mercury.png
│ └── ...
├── data/
│ └── planets.json # Initial planet data
├── models/
│ └── Planet.js # Mongoose schema for Planet
├── tests/
│ └── server.test.js # Backend API tests
├── .env.example # Example environment variables
├── .env # Your actual environment variables (Gitignored)
├── server.js # Main backend application file
├── package.json # Node.js project dependencies and scripts
├── package-lock.json # Node.js dependency lock file
└── README.md # This file

Contributing
Feel free to fork this repository, open issues, or submit pull requests to improve the application.

License
This project is open-source and available under the MIT License.