Monorepo Setup Guide (MongoDB)

Follow these commands in your terminal to build the project structure.

1. Create the Root Directory

First, create a main folder for your project and navigate into it.
mkdir convocation-app
cd convocation-app

2. Initialize the Monorepo Workspace

Now, initialize a bun project at the root. This will create the main package.json file that will manage all your workspaces.
bun init -y

Next, open the package.json file you just created and add the "workspaces" key. This tells bun where to find your different apps and packages.
./package.json

```
{
  "name": "convocation-app",
  "workspaces": [
    "apps/*",
    "packages/*"
  ]
}
```

3. Create the Directory Structure

Create the standard directories for a monorepo.
mkdir apps packages

4. Scaffold the Frontend (Next.js)

We'll use create-next-app to set up the frontend inside the apps directory. bunx is the bun equivalent of npx.
bunx create-next-app@latest apps/web

Follow the prompts. I recommend these settings:

- **Would you like to use TypeScript?** Yes
- **Would you like to use ESLint?** Yes
- **Would you like to use Tailwind CSS?** Yes
- **Would you like to use src/ directory?** No (for simplicity in this example)

- **Would you like to use App Router?** Yes
- **Would you like to customize the default import alias?** No

5. Create the Backend (ElysiaJS)

Now, let's manually create the folder structure for our backend API and initialize it.

```
# Create the directory
mkdir -p apps/api/src

# Navigate into it
cd apps/api

# Initialize its own package.json
bun init -y

# Install Elysia
bun add elysia

# Go back to the root directory
cd ../..
```

Now, create a basic server file at apps/api/src/index.ts:

./apps/api/src/index.ts

```
import { Elysia } from 'elysia';

const app = new Elysia()
  .get('/', () => 'Hello from Elysia API!')
  .listen(3001);

console.log(
  `🦊 Elysia is running at http://${app.server?.hostname}:${app.server?.port}`
);
```

6. Set Up Shared Packages

Create the directories for your shared database configuration and types.

```
# Create directories
mkdir -p packages/db packages/types/src

# Initialize package.json for the db package
cd packages/db
bun init -y
cd ../..
```

Initialize package.json for the types package
cd packages/types
bun init -y
cd ../..

Now, you can install Prisma in your root workspace.
bun add prisma typescript ts-node @types/node -D

Next, we'll initialize Prisma. First, add a script to the package.json inside your db package, specifying mongodb as the datasource provider.
./packages/db/package.json

```
{
  "name": "db",
  "module": "index.ts",
  "type": "module",
  "scripts": {
    "db:init": "prisma init --datasource-provider mongodb"
  }
}
```

Now, run the script from the root of your project.
cd packages/db
bun run db:init

This will create the prisma/schema.prisma file inside packages/db, configured for MongoDB.

7. Install All Dependencies

Finally, from the **root directory** of your project, run the install command. bun will read the package.json file in each of your workspaces (apps/web, apps/api, etc.) and install all dependencies correctly.
bun install

You now have a fully set up, type-safe monorepo! You can run your frontend and backend separately from the root directory:

- **Start Frontend:** bun --cwd apps/web dev
- **Start Backend:** bun --cwd apps/api dev (you'll need to add a dev script to apps/api/package.json that runs bun src/index.ts)
  Adding a dist directory for my production
  Of course. Using a dist (short for distributable) directory is a standard convention for production builds. Here's how you can configure that for both your backend and frontend.
  ## Backend (ElysiaJS)
  For your Elysia API, you just need to change the --outdir flag in your build script.

Open the package.json file in your apps/api directory.
Add or modify your scripts section to look like this:
JSON
"scripts": {
"dev": "bun run src/index.ts",
"build": "bun build ./src/index.ts --outdir ./dist",
"start": "bun ./dist/index.js"
}
Explanation:
"build": This script now tells Bun to compile your TypeScript code and place the optimized output into a ./dist folder.
"start": This script is now configured to run the production-ready application from that new ./dist folder.
Frontend (Next.js)
By default, Next.js uses a .next directory for its production build output. While you can change this, it's generally recommended to stick with the default, as deployment platforms like Vercel are optimized to work with it.
However, if you have a specific reason to change it to dist, here's how:
Open the next.config.js (or next.config.ts) file in your apps/web directory.
Add the distDir property to the configuration object:
JavaScript
/** @type {import('next').NextConfig} */
const nextConfig = {
distDir: 'dist', // Add this line
};
export default nextConfig;
Now, when you run bun run build in your apps/web directory, the output will be placed in a dist folder instead of .next.