Telegram Bot TypeScript Project Structure

TypeScript Telegram bot using grammY. Composer-based modules, sessions, and middleware architecture.

#telegram #typescript #bot #grammy #deno #nodejs #webhooks

PNG PDF Copy Prompt

Project Directory

```
telegram-bot/
  src/ Bot source code
    index.ts Entry point, bo…
    bot.ts Bot instance an…
    commands/ Command modules
      index.ts Composer exports
      start.ts
      help.ts
      settings.ts
    handlers/ Message and cal…
      index.ts
      messages.ts
      callbacks.ts
      inline.ts Inline queries
    middleware/ Bot middleware
      index.ts
      auth.ts User authorizat…
      logger.ts
      rateLimit.ts
    keyboards/
      index.ts
      main.ts
      settings.ts
    services/
      user.ts
      database.ts
    types/
      context.ts Extended context
      session.ts
    config.ts Environment con…
  tests/
    commands.test.ts
    handlers.test.ts
  package.json
  tsconfig.json
  .env.example
  .gitignore
  README.md
```

Why This Structure?

grammY uses Composers to organize handlers into reusable modules. Each command or feature is its own Composer that gets merged into the main bot. This pattern scales well and enables easy testing of individual modules.

Key Directories

src/commands/ - Command handlers as Composers

src/handlers/ - Message, callback, and inline handlers

src/middleware/ - Auth, logging, rate limiting

src/types/ - Extended context and session types

Getting Started

1. `npm init -y && npm install grammy dotenv`
2. `npm install -D typescript @types/node tsx`
3. `cp .env.example .env` and add your bot token
4. `npx tsx src/index.ts`

Composer Pattern

```typescript
// src/commands/start.ts
import { Composer } from "grammy";
import type { MyContext } from "../types/context";

const composer = new Composer<MyContext>();

composer.command("start", async (ctx) => {
  await ctx.reply("Welcome! Use /help for commands.");
});

export default composer;
```

Custom Context Types

Extend the default Context type to add session data, user info, or custom methods. Define your context in `types/context.ts` and use it throughout the bot for full type safety.

When To Use This

- Teams preferring TypeScript and type safety
- Projects needing middleware patterns
- Bots with complex session management
- Deno or Node.js environments

Best Practices

- Use Composers for modular command groups
- Define custom context type for type safety
- Use `session` plugin for persistent user data
- Middleware order matters—add auth before handlers
- Use `conversations` plugin for multi-step flows

Trade-offs

Newer ecosystem - Fewer tutorials than python-telegram-bot

Build step - TypeScript requires compilation (or tsx)

Plugin ecosystem - Some features need separate plugin packages