Cloudflare Worker Project Structure

TypeScript Worker with Wrangler, environment bindings, and modular route handling.

#cloudflare #worker #edge #typescript #serverless

PNG PDF Copy Prompt

Project Directory

```
my-worker/
  src/ Application sou…
    index.ts Entry point and…
    handlers/ Route handlers
      api.ts API endpoints
      assets.ts Static asset se…
    services/ Business logic
      kv.ts KV storage oper…
      d1.ts D1 database que…
    utils/
      response.ts Response helpers
      auth.ts Auth utilities
    types.ts Env and binding…
  migrations/ D1 database mig…
    0001_init.sql
  test/ Vitest tests
    index.test.ts
    handlers.test.ts
  wrangler.toml Worker configur…
  tsconfig.json
  package.json
  .dev.vars Local secrets
  README.md
```

Why This Structure?

This structure separates handlers (routing) from services (business logic). The `wrangler.toml` defines bindings to KV, D1, R2, and other Cloudflare services. TypeScript types in `types.ts` ensure type-safe access to environment bindings.

Key Directories

src/handlers/ - Route handlers, one file per resource or path group

src/services/ - Business logic and storage abstractions

migrations/ - D1 SQL migrations run via Wrangler

wrangler.toml - Bindings, routes, and deployment config

Worker Entry Point

```
export interface Env {
  MY_KV: KVNamespace;
  DB: D1Database;
  BUCKET: R2Bucket;
  API_KEY: string;
}

export default {
  async fetch(request: Request, env: Env): Promise<Respons
    const url = new URL(request.url);
    if (url.pathname.startsWith("/api/")) {
      return handleApi(request, env);
    }
    return new Response("Not found", { status: 404 });
  },
};
```

Getting Started

1. `npm create cloudflare@latest`
2. Select Worker template with TypeScript
3. `npx wrangler dev` for local development
4. `npx wrangler d1 create my-db` for D1 database
5. `npx wrangler deploy` to publish

Cloudflare Bindings

KV - Key-value storage for config, cache, sessions

D1 - SQLite database at the edge

R2 - Object storage (S3-compatible)

Durable Objects - Stateful edge compute with WebSocket support

Queues - Message queues for async processing

Best Practices

- Define all bindings in `Env` interface for type safety
- Use `.dev.vars` for local secrets (never commit)
- Keep handlers thin, put logic in services
- Use `waitUntil()` for background tasks after response
- Test with `wrangler dev --local` for fast iteration

When To Use This

- Low-latency APIs served from 300+ edge locations
- Auth at the edge (JWT validation, rate limiting)
- Image optimization and transformation
- A/B testing and feature flags
- Serverless APIs with built-in storage (KV, D1, R2)

Trade-offs

CPU limits - 10-50ms CPU time per request (plan dependent)

No Node.js - Web APIs only, some npm packages won't work

Cold starts - Minimal (~0-5ms) but isolate limits apply