DEX-OS Development Guide

Prerequisites

1. Install Rust toolchain:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

2. Install wasm-pack for WebAssembly builds:

```
cargo install wasm-pack
```

3. Install PostgreSQL for database functionality (optional):
   - Download from https://www.postgresql.org/download/
   - Or use Docker: `docker run -d -p 5432:5432 -e POSTGRES_PASSWORD=postgres postgres`

Project Structure

The DEX-OS is organized as a Cargo workspace with multiple crates:

- `dex-core`: Core DEX engine logic
- `dex-wasm`: WebAssembly bindings
- `dex-db`: Database layer
- `dex-api`: HTTP API layer

Building

Build the entire workspace:

```
cargo build
```

Build in release mode:

```
cargo build --release
```

Build the WASM module:

```
# Unix-like systems
./build-wasm.sh

# Windows
build-wasm.bat
```

Testing

Run tests for all crates:

```
cargo test
```

Run tests for a specific crate:

```
cargo test -p dex-core
```

Running

Start the API server:

```
cargo run -p dex-api
```

The server will start on `http://localhost:3030`

Environment Configuration

The API reads its configuration from environment variables (a `.env` file in the repository root is also supported):

- `DATABASE_URL` (required) — Postgres connection string used by `dex-db`.
- `JWT_SECRET` (required) — HMAC signing key for authenticating API requests.
- `SERVER_PORT` (optional) — Override the default `3030` HTTP port.

API Endpoints

1. `POST /orderbook/orders` - Create a new order
2. `GET /orderbook/prices` - Get best bid and ask prices

Database Setup

To use the database functionality:

1. Start PostgreSQL server
2. Create a database:

```
CREATE DATABASE dex_os;
```

3. The database schema will be automatically initialized when the application starts

WebAssembly Usage

After building the WASM module, you can use it in a web application:

```
import init, { WasmOrderBook, WasmAMM } from './pkg/dex_wasm.js';

async function run() {
    await init();

    const orderbook = new WasmOrderBook();
    const amm = new WasmAMM(30); // 0.3% fee

    // Use the DEX functionality in the browser
}
```

Development Workflow

1. Make changes to the Rust code
2. Run tests: `cargo test`
3. Build: `cargo build`
4. For WASM changes: rebuild with `./build-wasm.sh`
5. Test API: `cargo run -p dex-api`

Code Organization

Core Engine (`dex-core`)

The core engine implements the fundamental DEX functionality:

- Orderbook management with BTreeMap-based storage
- Price-time priority matching algorithm
- Automated Market Maker with constant product formula
- Common types and data structures

WebAssembly Interface (`dex-wasm`)

The WASM interface provides bindings to use the core engine in web browsers:

- wasm-bindgen wrappers for OrderBook and AMM

- JavaScript-compatible APIs
- Serialization/deserialization between JS and Rust

Database Layer (`dex-db`)

The database layer handles persistence:

- SQLx-based database interactions
- Order storage and retrieval
- Trade history management

API Layer (`dex-api`)

The API layer provides HTTP endpoints:

- Warp-based web server
- RESTful endpoints for order management
- Real-time price feeds

Extending the DEX

To add new features:

1. Implement core logic in `dex-core`
2. Add database support in `dex-db` if needed
3. Expose via API in `dex-api`
4. Add WASM bindings in `dex-wasm` for web support

Performance Considerations

- Use `cargo build --release` for production builds
- Consider using jemallocator for better memory allocation:

```
[dependencies]
jemallocator = "0.5"
```

- For WASM, use the standard allocator; the optional `wee_alloc` feature has been removed due to maintenance concerns