Python Sandbox Executor Setup Guide

This guide will help you set up the Python Sandbox Executor for secure code execution in your TaskManager system.

Architecture Overview

The system consists of several components:

1. **Python Sandbox Executor** - A standalone service that executes Python code in isolated Docker containers
2. **TaskManager Integration** - Components that connect the sandbox with the TaskManager and agent system
3. **API Routes** - FastAPI routes for the frontend to interact with the sandbox
4. **UI Components** - React components for executing and visualizing code execution

Prerequisites

- Docker and Docker Compose
- Python 3.10+
- Node.js 16+ (for frontend)
- Redis server
- FastAPI backend

Installation Steps

1. Set Up the Sandbox Executor

First, create a directory for the sandbox executor:

bash

```bash
mkdir -p sandbox-executor
cd sandbox-executor
```

Copy these files to the directory:

- `sandbox_executor.py`
- `Dockerfile`

- `requirements.txt`

Build the Docker image:

bash Copy

```bash
docker build -t sandbox-executor .
```

2. Install Integration Components

In your TaskManager backend directory, create these files:

- `sandbox_agent_integration.py`
- `agent_python_tools.py`
- `routes/sandbox_routes.py`
- `routes/python_routes.py`
- `startup_script.py`

Add these dependencies to your backend's `requirements.txt`:

Copy

```
httpx>=0.24.0
docker>=6.1.2
redis>=4.5.4
```

3. Set Up Frontend Components

In your frontend project, add the React component:

- `components/PythonExecution.tsx` (converted from our React JSX)

Add these routes to your Next.js API routes:

- `pages/api/sandbox/[...route].js` (forwarding to backend)
- `pages/api/python/[...route].js` (forwarding to backend)

4. Update Configuration

Set these environment variables in your `.env` file:

Copy

```
# Sandbox configuration
SANDBOX_API_URL=http://localhost:8001
TOOL_AGENT_NAME=python_sandbox

# Redis configuration
REDIS_URL=redis://localhost:6379

# Security settings
SANDBOX_ALLOW_SUBPROCESS=false  # Set to true only for testing without Docker
```

5. Update Your FastAPI Startup

In your main FastAPI application, add:

python Copy

```python
from startup_script import configure_settings, startup_sandbox_services
from routes.sandbox_routes import register_sandbox_routes
from routes.python_routes import register_python_routes

@app.on_event("startup")
async def startup_event():
    # Existing startup code...

    # Configure sandbox settings
    configure_settings()

    # Start sandbox services
    await startup_sandbox_services()

    # Register API routes
    register_sandbox_routes(app)
    register_python_routes(app)
```

6. Launch with Docker Compose

Update your `docker-compose.yml` to include the sandbox executor service:

yaml

```yaml
services:
  # Existing services...

  sandbox-executor:
    build:
      context: ./sandbox-executor
      dockerfile: Dockerfile
    ports:
      - "8001:8000"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    depends_on:
      - redis
    environment:
      - REDIS_URL=redis://redis:6379
      - LOG_LEVEL=INFO
    networks:
      - app-network
```

Start all services:

bash

```bash
docker-compose up -d
```

Security Considerations

The Python Sandbox Executor is designed with security in mind:

1. **Docker Isolation**: Code is executed in isolated containers with resource limits
2. **Network Restrictions**: Containers run with `network_mode="none"` by default
3. **Read-Only Filesystem**: Containers use a read-only filesystem by default
4. **Resource Limits**: Memory, CPU, and execution time are strictly limited
5. **Dependency Control**: Dependencies are validated before installation

⚠️ **Warning**: The subprocess execution mode is less secure and should only be used for testing. Always use Docker in production.

Usage Examples

From the UI

1. Navigate to your application's Python execution page
2. Enter Python code in the editor
3. Add dependencies if needed
4. Set execution parameters
5. Click "Execute Code"
6. View the results including stdout, stderr, and generated files

From Agent Code

python   Copy

```python
# Make your agent Python-capable
from agent_python_tools import PythonExecutionMixin

class MyAgent(PythonExecutionMixin, BaseAgent):
    # Your agent code...
    pass

# Execute Python code
execution_result = await agent.execute_python_code(
    code="print('Hello, world!')",
    task_id="task_123"
)

# Execute data analysis with common dependencies
result = await agent.execute_data_analysis(
    code="import pandas as pd\nimport matplotlib.pyplot as plt\n...",
    task_id="analysis_task_456",
    include_visualization=True
)
```

Troubleshooting

Docker Issues

If you encounter Docker-related issues:

1. Check if Docker daemon is running: `docker ps`
2. Ensure the Docker socket is accessible: `ls -la /var/run/docker.sock`

3. Check if the sandbox executor has permission to access Docker

Execution Problems

If code execution fails:

1. Check the sandbox executor logs: `docker logs sandbox-executor`
2. Verify Redis connectivity
3. Check if dependencies are valid PyPI packages
4. Ensure resource limits are reasonable for the code you're executing

Integration Issues

If the integration with TaskManager fails:

1. Check if the `SandboxAgentBridge` is initialized correctly
2. Verify Redis connectivity for message passing
3. Check that agent message handling is patched correctly
4. Ensure environment variables are set correctly

Next Steps and Enhancements

Future enhancements could include:

1. **Code Linting & Analysis**: Pre-execution validation
2. **User Permission Levels**: Execution privileges based on user roles
3. **Persistent Storage**: Save notebooks and code for later use
4. **Additional Languages**: Support for JavaScript, Ruby, etc.
5. **GPU Support**: For ML/AI workloads

Support

If you encounter issues, check:

1. Sandbox executor logs: `docker logs sandbox-executor`
2. Backend application logs
3. Redis logs for message passing
4. Docker logs for container execution

For persistent issues, contact the development team.