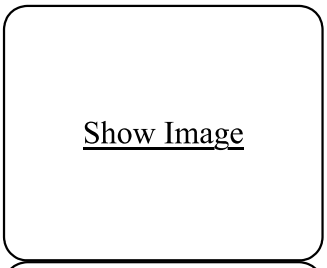
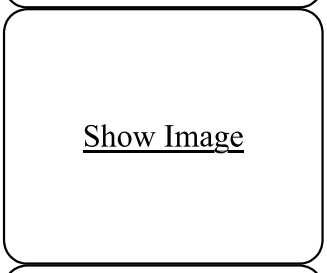
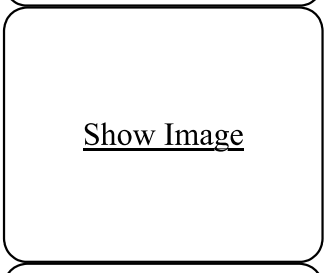
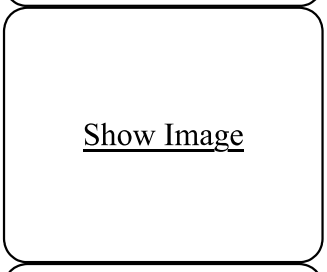
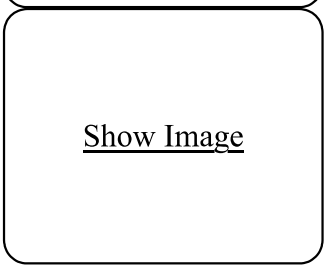
Professional Python Microservice Template











A production-ready microservice template built with FastAPI, PostgreSQL, Docker, and deployed on WSL2. Designed for the **Professional Development Environment** course.

📋 Table of Contents

✨ Features

- **FastAPI Framework -** High-performance async Python web framework
- **PostgreSQL Database -** Robust relational database with asyncpg
- **Docker Containerization -** Consistent development and production environments
- **VS Code Dev Containers -** Develop inside containers with full IDE support
- **Automated Testing -** pytest with >80% code coverage
- **Code Quality Tools -** Black, isort, pylint, mypy
- **Database Migrations -** Automatic schema initialization
- **API Documentation -** Auto-generated Swagger UI and ReDoc
- **Health Checks -** Built-in monitoring endpoints
- **CORS Support -** Configurable cross-origin resource sharing
- **Logging -** Structured logging with configurable levels
- **Error Handling -** Comprehensive exception handling

📦 Prerequisites

Required Software

- **Windows 10/11** (Build 19041 or higher) or **Linux/macOS**
- **WSL2** (Windows only) - Version 0.67.6+

- **Docker Desktop** - Latest version
- **VS Code** with extensions:
  - Remote - WSL
  - Docker
  - Python
  - Dev Containers

Hardware Requirements

- **RAM:** 12GB minimum (8GB for WSL2 + 4GB for system)
- **CPU:** 4 cores recommended
- **Storage:** 50GB free space

Check Your Setup

```bash
# WSL version (Windows)
wsl --version

# Docker
docker --version
docker compose version

# Python (inside container)
python --version
```

🚀 Quick Start

1. Clone Repository

```bash
# Clone to WSL2 filesystem (IMPORTANT!)
cd ~/projects
git clone <repository-url> my-microservice
cd my-microservice
```

2. Configure Environment

```bash
# Copy environment template
cp .env.example .env

# Edit with your values
nano .env
```

3. Start Services

```bash
# Start all containers
docker compose up -d

# Check status
docker compose ps

# View logs
docker compose logs -f web
```

4. Verify Installation

```bash
# Health check
curl http://localhost:8000/health

# API documentation
open http://localhost:8000/docs
```

5. Open in VS Code

```bash
# From WSL terminal
code .
```

Then:

1. Press Ctrl+Shift+P

2. Select "Dev Containers: Reopen in Container"
3. Wait for container to build
4. Start coding!

📁 Project Structure

```
my-microservice/
│
├── .devcontainer/              # VS Code dev container configuration
│   └── devcontainer.json       # Container settings
│
├── .github/                    # GitHub workflows (CI/CD)
│   └── workflows/
│       └── test.yml            # Automated testing
│
├── .vscode/                    # VS Code workspace settings
│   ├── launch.json             # Debug configurations
│   ├── settings.json           # Editor settings
│   └── tasks.json              # Build tasks
│
├── docker/                     # Docker-related files
│   └── init-db/                # Database initialization scripts
│       └── 01-init.sql         # Schema and seed data
│
├── src/                        # Application source code
│   ├── __init__.py             # Package initialization
│   └── main.py                 # FastAPI application
│
├── tests/                      # Test suite
│   ├── __init__.py             # Test package
│   └── test_main.py            # API endpoint tests
│
├── .dockerignore               # Files to exclude from Docker build
├── .gitignore                  # Files to exclude from Git
├── .env.example                # Environment variables template
├── docker-compose.yml          # Service orchestration
├── Dockerfile                  # Container image definition
├── requirements.txt            # Python dependencies
└── README.md                   # This file
```

Key Directories

- **src/** - All application code lives here
- **tests/** - Unit and integration tests
- **docker/** - Docker-specific configurations
- **.vscode/** - Development environment settings
- **.devcontainer/** - Container development setup

💻 Development

Running the Application

```bash
# Start services
docker compose up -d

# Start with rebuild
docker compose up -d --build

# Stop services
docker compose down

# Stop and remove volumes (WARNING: deletes data)
docker compose down -v

# Restart specific service
docker compose restart web
```

Viewing Logs

```bash
```

```
# All services
docker compose logs -f

# Specific service
docker compose logs -f web

# Last 50 lines
docker compose logs --tail=50 web
```

Database Access

```bash
# Connect to PostgreSQL
docker exec -it <project>-db psql -U postgres -d myapp

# Run SQL file
docker exec -i <project>-db psql -U postgres -d myapp < backup.sql

# Database shell
docker compose exec db psql -U postgres -d myapp
```

Making Code Changes

1. **Edit files** in VS Code (inside dev container)
2. **Save** - FastAPI auto-reloads with `--reload` flag
3. **Test** your changes: `curl http://localhost:8000/<endpoint>`
4. **Commit**: `git add . && git commit -m "feat: description"`

Adding Dependencies

```bash
```

```
# Inside container or with Docker
docker compose exec web pip install <package>

# Update requirements.txt
docker compose exec web pip freeze > requirements.txt

# Rebuild container
docker compose build web
docker compose up -d
```

🧪 Testing

Running Tests

bash

```
# Run all tests
pytest

# Run with coverage
pytest --cov=src --cov-report=html

# Run specific test file
pytest tests/test_main.py

# Run specific test
pytest tests/test_main.py::TestUserEndpoints::test_list_users

# Run with verbose output
pytest -v

# Run with print statements
pytest -s

# Run marked tests only
pytest -m integration
```

Test Coverage

bash

```
# Generate HTML coverage report
pytest --cov=src --cov-report=html

# Open report
open htmlcov/index.html  # macOS
xdg-open htmlcov/index.html  # Linux
```

Writing Tests

Tests go in `tests/` directory:

```python
# tests/test_my_feature.py
def test_my_endpoint(client):
    """Test my new endpoint"""
    response = client.get("/my-endpoint")
    assert response.status_code == 200
    assert "expected_field" in response.json()
```

📚 API Documentation

Interactive Documentation

Once running, visit:

- **Swagger UI:** http://localhost:8000/docs
- **ReDoc:** http://localhost:8000/redoc

Available Endpoints

Health Check

```http
GET /health
```

Returns API health status and database connectivity.

Users

```http
GET    /users              # List all users (paginated)
GET    /users/{id}         # Get user by ID
POST   /users              # Create new user
PATCH  /users/{id}         # Update user
DELETE /users/{id}         # Delete user (soft delete)
```

Projects

```http
GET    /projects           # List all projects
GET    /projects/{id}      # Get project by ID
POST   /projects           # Create new project
PATCH  /projects/{id}      # Update project
DELETE /projects/{id}      # Delete project
```

Example Requests

Create User:

```bash
curl -X POST http://localhost:8000/users \
  -H "Content-Type: application/json" \
  -d '{
    "username": "johndoe",
    "email": "john@example.com",
    "password": "securepass123",
    "full_name": "John Doe"
  }'
```

List Users:

```bash
curl http://localhost:8000/users?limit=10&offset=0
```

Get User:

```bash
```

```
curl http://localhost:8000/users/{user_id}
```

🗄️ Database

Schema

The database includes the following tables:

- **users** - User accounts
- **projects** - User projects
- **tasks** - Project tasks
- **audit_log** - Change tracking

See `docker/init-db/01-init.sql` for complete schema.

Migrations

Database is initialized automatically on first startup using scripts in `docker/init-db/`.

Connecting to Database

From host machine:

```bash
psql -h localhost -p 5432 -U postgres -d myapp
```

From container:

```bash
docker compose exec db psql -U postgres -d myapp
```

Backup & Restore

Backup:

```bash
docker compose exec db pg_dump -U postgres myapp > backup.sql
```

Restore:

```bash
docker compose exec -T db psql -U postgres myapp < backup.sql
```

🚢 Deployment

Production Configuration

1. **Create production .env:**

```bash
cp .env.example .env.production
nano .env.production
```

2. **Set production values:**

```bash
ENVIRONMENT=production
DEBUG=false
LOG_LEVEL=WARNING
DATABASE_URL=postgresql://user:pass@prod-db:5432/myapp
SECRET_KEY=<generate-strong-key>
```

3. **Build for production:**

```bash
docker compose -f docker-compose.yml -f docker-compose.prod.yml build
```

Environment Variables

Variable	Description	Default	Required
`ENVIRONMENT`	Environment name	`development`	No
`LOG_LEVEL`	Logging level	`INFO`	No

Variable	Description	Default	Required
`DATABASE_URL`	PostgreSQL connection string	See .env.example	Yes
`SECRET_KEY`	Encryption key	-	Yes (prod)
`CORS_ORIGINS`	Allowed origins	`*`	No

Security Checklist

Before deploying to production:

- [ ] Change all default passwords
- [ ] Generate strong SECRET_KEY
- [ ] Configure specific CORS_ORIGINS
- [ ] Disable DEBUG mode
- [ ] Set LOG_LEVEL to WARNING or ERROR
- [ ] Use HTTPS/TLS certificates
- [ ] Enable firewall rules
- [ ] Set up monitoring and alerts
- [ ] Configure backup strategy
- [ ] Review database permissions

🔧 Troubleshooting

Common Issues

Port Already in Use

Error: `address already in use`

Solution:

```bash
```

```
# Find process using port
sudo lsof -i :8000

# Stop conflicting container
docker compose down

# Or change port in docker-compose.yml
ports:
  - "8001:8000"
```

Database Connection Failed

Error: `could not connect to server`

Solution:

```
bash

# Check database is running
docker compose ps

# Check database logs
docker compose logs db

# Restart database
docker compose restart db
```

Out of Memory

Error: `Container killed (OOMKilled)`

Solution:

1. Increase Docker memory limit (Docker Desktop → Settings → Resources)
2. Reduce WSL2 memory in `.wslconfig`
3. Stop unused containers: `docker stop $(docker ps -q)`

Permission Denied

Error: `Permission denied` when accessing files

Solution:

```bash
# Fix file ownership (inside container)
chown -R appuser:appuser /app

# Or rebuild container
docker compose build --no-cache
```

Getting Help

1. **Check logs:** `docker compose logs -f`
2. **Verify health:** `curl http://localhost:8000/health`
3. **Test database:** `docker compose exec db psql -U postgres -d myapp -c "SELECT 1"`
4. **Rebuild containers:** `docker compose down && docker compose build && docker compose up -d`

Debug Mode

Enable debug logging:

```bash
# In .env
LOG_LEVEL=DEBUG
DEBUG=true

# Restart
docker compose restart web
```

🤝 Contributing

Development Workflow

1. **Create feature branch:**

```bash
git checkout -b feature/my-feature
```

2. **Make changes and test:**

```bash
# Make changes
# Run tests
pytest
# Format code
black src/ tests/
isort src/ tests/
```

3. **Commit with conventional commits:**

```bash
git add .
git commit -m "feat: add user authentication"
```

4. **Push and create PR:**

```bash
git push origin feature/my-feature
# Create pull request on GitHub
```

Commit Message Convention

- `feat:` New feature
- `fix:` Bug fix
- `docs:` Documentation only
- `style:` Formatting, missing semicolons, etc.
- `refactor:` Code restructuring
- `test:` Adding tests
- `chore:` Maintenance tasks

Code Style

This project uses:

- **black** for code formatting
- **isort** for import sorting

- **pylint** for linting
- **mypy** for type checking

Run before committing:

```bash
black src/ tests/
isort src/ tests/
pylint src/
mypy src/
```

📄 License

This project is licensed under the MIT License - see LICENSE file for details.

📞 Support

- **Documentation:** See `docs/` directory
- **Issues:** GitHub Issues
- **Questions:** GitHub Discussions
- **Email:** support@example.com

🙏 Acknowledgments

Built with:

- FastAPI - Web framework
- PostgreSQL - Database
- Docker - Containerization
- pytest - Testing framework

Course materials adapted from industry best practices and professional development standards.

Happy Coding! 🚀