Checklist for Configuration

The Configuration category centralizes all system settings, ensuring that Orgo operates consistently and securely across environments (e.g., development, production). This checklist ensures configuration files are modular, validated, and aligned with global standards.

---

1. Structure

[ ] Modularity:

Split configurations into smaller, focused files (e.g., smtp\_config.yaml, db\_config.yaml).

[ ] Base Configuration:

Include a base\_config.yaml for global settings shared across the system.

[ ] Environment Overrides:

Add an environment.yaml file for development, staging, and production-specific settings.

[ ] Sensitive Data:

Store sensitive data (e.g., credentials, API keys) in separate files under /security/.

---

2. Naming Conventions

[ ] Keys:

Use descriptive, snake\_case keys (e.g., smtp\_server, db\_host).

[ ] Placeholders:

Use <ALL\_CAPS> placeholders for sensitive or example values.

Example: <USERNAME>, <PASSWORD>, <DB\_HOST>.

[ ] Consistency:

Ensure keys and placeholders are consistent across all configuration files.

---

3. File Validation

[ ] Required Keys:

Ensure every configuration file includes mandatory keys.

Example for smtp\_config.yaml:

smtp:

server: "<SMTP\_SERVER>" # Mandatory

port: 587 # Mandatory

encryption: "TLS" # Mandatory

[ ] Data Types:

Validate all values match expected types (e.g., strings for server addresses, integers for ports).

[ ] Error Messages:

Provide clear, actionable error messages for missing or invalid keys.

---

4. Metadata

[ ] Versioning:

Include a version field in every configuration file.

Example:

metadata:

version: "1.0"

last\_updated: "2024-11-24"

[ ] Environment:

Add a environment field to indicate the active environment (e.g., development, production).

---

5. Sensitive Data Management

[ ] Separation:

Store sensitive data in /security/credentials.yaml or similar files.

Example:

smtp:

username: "<USERNAME>"

password: "<PASSWORD>"

database:

username: "<DB\_USERNAME>"

password: "<DB\_PASSWORD>"

[ ] Encryption:

Encrypt sensitive configuration files or manage them via environment variables.

---

6. Environment-Specific Overrides

[ ] File Location:

Place overrides in environment.yaml or similarly named files.

[ ] Structure:

Example:

development:

smtp:

server: "smtp.dev.example.com"

database:

host: "localhost"

production:

smtp:

server: "smtp.prod.example.com"

database:

host: "prod-db.example.com"

[ ] Fallback:

Ensure overrides fall back to base\_config.yaml if environment-specific values are missing.

---

7. Shared Configuration Loader

[ ] Centralized Script:

Implement a script (config\_loader.py) to dynamically load and merge configurations.

[ ] Validation:

Validate required keys and data types during the load process.

[ ] Environment Awareness:

Apply environment-specific overrides automatically.

Example Loader Script

import yaml

def load\_config(service, environment="production"):

with open(f"config/global/base\_config.yaml", "r") as base\_file:

config = yaml.safe\_load(base\_file)

with open(f"config/services/{service}/{service}\_config.yaml", "r") as service\_file:

service\_config = yaml.safe\_load(service\_file)

config.update(service\_config)

with open("config/global/environment.yaml", "r") as env\_file:

env\_config = yaml.safe\_load(env\_file)

if environment in env\_config:

config.update(env\_config[environment].get(service, {}))

return config

---

8. Logging Configuration

[ ] File Separation:

Separate logging configurations into general, activity, error, and security YAML files.

[ ] Retention Policies:

Define retention periods for each log type.

Example:

activity:

enabled: true

retention\_period: "6 months"

error:

enabled: true

alert\_on\_critical: true

[ ] Log Directory:

Specify a shared directory for storing logs (e.g., /var/logs/orgo).

---

9. Testing

[ ] Validation Scripts:

Write scripts to test all configuration files for:

Presence of required keys.

Correct data types.

Environment-specific overrides.

[ ] Error Handling:

Simulate scenarios with missing or invalid keys to ensure meaningful error messages.

Example Test Script

def validate\_config(config, required\_keys):

for key in required\_keys:

if key not in config:

raise KeyError(f"Missing required key: {key}")

print("Configuration validated successfully!")

---

10. Scalability

[ ] Adding New Configurations:

New configurations should follow the same structure and validation rules.

[ ] Dynamic Loading:

Ensure the configuration loader can handle new files without modification.

[ ] Documentation:

Include inline comments for every key-value pair to guide future updates.

---

Example Configuration Application

File Structure

/config/

├── global/

│ ├── base\_config.yaml # Shared global settings

│ ├── environment.yaml # Environment-specific overrides

│ └── credentials.yaml # Global credentials

├── services/

│ ├── email/

│ │ ├── smtp\_config.yaml # SMTP server details

│ │ ├── imap\_config.yaml # IMAP server details

│ │ └── overrides.yaml # Email-specific overrides

│ ├── database/

│ │ ├── postgres\_config.yaml # PostgreSQL-specific settings

│ │ ├── sqlite\_config.yaml # SQLite-specific settings

│ │ └── db\_credentials.yaml # Credentials for databases

│ └── logging/

│ ├── general.yaml # General logging settings

│ ├── activity.yaml # Activity log settings

│ ├── error.yaml # Error log settings

│ └── security.yaml # Security log settings

---

Optimized Workflow

1. Start with Base Configurations:

Define global settings and environment overrides.

2. Validate Configuration Files:

Test for required keys and values.

3. Write Shared Loader:

Implement a loader script to dynamically handle all configurations.

4. Integrate Gradually:

Use configurations in small modules (e.g., email parsing) before expanding system-wide.

---

This checklist ensures configuration files are modular, secure, and scalable while adhering to best practices. Let me know if you'd like to apply it to a specific configuration group!