Best Practices for Managing Environment Variables in DevOps Workflows

Author: Zayan Ahmed | **Estimated Reading time:** 6 mins

Environment variables play a critical role in DevOps workflows, acting as the medium for passing configuration data between different stages of application development, deployment, and operation. They help decouple sensitive data and dynamic settings from the application code, ensuring better security, flexibility, and maintainability.

This document explores common workflows, best practices, and tools for effectively managing environment variables in DevOps workflows.

1. Understanding Environment Variables

Environment variables are key-value pairs used to configure application behavior without hardcoding values into the application. Typical use cases include:

- **Database credentials** (e.g., `DB_USER`, `DB_PASSWORD`)
- **API keys** (e.g., `API_KEY`, `AUTH_TOKEN`)
- **Service URLs** (e.g., `BASE_URL`, `REDIS_HOST`)
- **Environment-specific settings** (e.g., `NODE_ENV=production`)

These variables are loaded during runtime, enabling seamless transitions between development, staging, and production environments.

---

2. Workflows for Managing Environment Variables

2.1 Using `.env` Files

`.env` files are commonly used to define environment variables during local development. These files contain key-value pairs and can be loaded into the application using libraries like `dotenv` for Node.js or `python-dotenv` for Python.

Workflow:

1. Create a `.env` file in your project root:

```
DB_USER=admin
DB_PASSWORD=securepassword
API_KEY=abc123xyz
```

2. Load the variables in your application:

```
require('dotenv').config();
const dbUser = process.env.DB_USER;
```

3. Exclude `.env` files from version control by adding them to `.gitignore`.

Best Practices:

- Do not commit `.env` files to repositories.
- Use `.env.example` files to document required variables without including sensitive values.

2.2 CI/CD Pipelines

In DevOps workflows, environment variables are often managed within CI/CD tools like Jenkins, GitHub Actions, GitLab CI, or Azure DevOps. These tools allow secure storage and injection of variables during pipeline execution.

Workflow:

1. Define variables in the CI/CD tool's settings (e.g., Jenkins credentials store, GitHub Secrets).
2. Access variables in pipeline scripts:
   - **GitHub Actions:**

```
env:
 NODE_ENV: production
 API_KEY: ${{ secrets.API_KEY }}
```

Jenkins Pipeline:

```
withCredentials([string(credentialsId: 'API_KEY', variable:
'API_KEY')]) {

   sh 'echo $API_KEY'

}
```

3. **Pass variables to the application or deployment scripts during runtime.**

Best Practices:

- **Store secrets securely using built-in tools.**
- **Rotate secrets regularly and revoke unused ones.**

2.3 Configuration Management Tools

Tools like Ansible, Chef, and Puppet enable centralized management of environment variables and secrets, ensuring consistent configuration across environments.

Workflow:

1. **Store variables in inventory files, playbooks, or encrypted vaults:**

```
vars:

 db_user: admin

 db_password: securepassword
```

2. **Use templates or inline substitutions to inject variables into configuration files.**

3. **Deploy the configurations to target systems.**

Best Practices:

- **Use encryption (e.g., Ansible Vault) for sensitive data.**
- **Separate variable definitions by environment for better isolation.**

2.4 Containerization and Orchestration

In containerized environments, environment variables are defined in Dockerfiles, `docker-compose` files, or orchestration tools like Kubernetes.

Workflow:

- **Docker: Define variables in `docker-compose.yml` or pass them during container runtime:**

```
environment:

 - NODE_ENV=production

 - API_KEY=${API_KEY}
```

Kubernetes: Use ConfigMaps and Secrets to manage environment variables:

```
apiVersion: v1

kind: Secret

metadata:

 name: api-keys

data:

 API_KEY: abc123xyz (base64 encoded)
```

Best Practices:

- Avoid hardcoding secrets in Dockerfiles.
- Use Kubernetes Secrets for sensitive data and ensure RBAC policies restrict access.

2.5 Environment Variable Managers

Specialized tools like HashiCorp Vault, AWS Secrets Manager, and Azure Key Vault provide robust mechanisms for managing environment variables and secrets.

Workflow:

1. Store variables securely in the secret management tool.
2. Retrieve variables programmatically or inject them into CI/CD pipelines and runtime environments.
    - **HashiCorp Vault:** Use dynamic secrets and leases to minimize exposure.

    - **AWS Secrets Manager:** Access secrets using AWS SDKs or IAM roles.

Best Practices:

- Monitor and audit access to secrets.
- Use dynamic secrets for ephemeral access to resources.

---

3. General Best Practices for Environment Variables

- **Avoid Hardcoding:** Never hardcode secrets or environment-specific configurations into your codebase.
- **Use Namespacing:** Prefix variable names with the application or service name (e.g., `APP1_DB_USER`) to avoid conflicts.
- **Document Variables:** Maintain clear documentation for required variables, their purposes, and example values.
- **Secure Sensitive Data:** Encrypt secrets in transit and at rest. Use access controls to restrict who can view or modify them.
- **Test Configurations:** Validate environment variable configurations in staging environments before deploying to production.

---

4. Conclusion

Integrating environment variables into DevOps workflows is crucial for building secure, scalable, and maintainable systems. By leveraging the right tools and adhering to best practices, teams can ensure consistent and efficient management of configurations across all stages of development and deployment. Whether using `.env` files for local development or advanced secret managers for production environments, a robust strategy for managing environment variables is essential for successful DevOps operations.

Want more ? 🤔
Follow me on **LinkedIn** 😊