ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 24 Manageability

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

November 10, 2025

Outline

Manageability and Application Configuration

Manageability in Cloud Systems

Reading Assignment

- This lecture: 10
- Next Lecture: 11

Outline

Manageability and Application Configuration

Manageability in Cloud Systems

Manageability

- Change behaviors without having to recode and redeploy.
  - By yourself or by someone else.
- Manageability allows to make changes from outside.
  - Maintainability allows to make changes from inside, usually by updating code.
- Manageability for complex systems.
  - Make configuration and control options available.
  - Use monitoring, logging, and alerting to identify components that require management, e.g. misconfigured components.
  - Manage deployment by updating, rolling back, and scaling system components.
  - Discover available services.

Application Configuration

- Configuration: anything likely to vary between environments like staging, production, developer, etc.
- Store configuration in the environment.
  - Configuration should be strictly separated from the code.
  - Configurations should be stored in version control – make it possible to inspect, review, rollback, and troubleshoot changes.
- Configuration practices
  - Command-line flags and environment variables: use start-up scripts for version control.
  - Configuration files: use standard format like JSON and YAML.

Configuring with Environment Variables

- Use environment variables

```
name := os.Getenv("NAME")
place := os.Getenv("CITY")
fmt.Printf("%s lives in %s.\n", name, place)
```

- Distinguish between an empty value and an unset value.

```
if val, ok := os.LookupEnv(key); ok {
  fmt.Printf("%s=%s\n", key, val)
} else {
  fmt.Printf("%s not set\n", key)
}
```

Configuring with Command-Line Arguments

```
package main
import (
  "flag"
  "fmt"
)
func main() {
  strp := flag.String("string", "foo", "a string")
  intp := flag.Int("number", 42, "an integer")
  boolp := flag.Bool("boolean", false, "a boolean")
  flag.Parse() // Call flag.Parse() to execute command-line parsing.
  fmt.Println("string:", *strp)
  fmt.Println("integer:", *intp)
  fmt.Println("boolean:", *boolp)
  fmt.Println("args:", flag.Args())
}
```

- Use the `flag` package for command-line flags.
  - Register with types, default values, and short descriptions
  - Map flags to variables.

Configuring with Command-Line Arguments (cont.)

```
$ go run . -help
Usage of /var/folders/go-build618108403/exe/main:
  -boolean
    a boolean
  -number int
    an integer (default 42)
  -string string
    a string (default "foo")
$ go run . -boolean -number 27 -string "A string." Other things.
string: A string.
integer: 27
boolean: true
args: [Other things.]
```

Configuring with JSON Files

```
type Config struct {
  Host string
  Port uint16
  Tags map[string]string
}
func EncodeJson() {
  c := Config{
    Host: "localhost",
    Port: 1313,
    Tags: map[string]string{"env": "dev"},
  }
  bytes, err := json.Marshal(c)
  fmt.Println(string(bytes))
  // {"Host":"localhost","Port":1313,"Tags":{"env":"dev"}}
}
```

- Use `json.Marshal()` to encode any struct as JSON string.
  - Only public fields (begin with a capital letter) are encoded.

Configuring with JSON Files (cont.)

- Use `json.Unmarshal()` to decode JSON string into a struct.

```
c := Config{}
bytes := []byte(‘{"Host":"127.0.0.1","Port":1234,"Tags":{"foo":"bar"}}‘)
err := json.Unmarshal(bytes, &c)
```

  - Missing fields will have a default value of zero or empty.
  - Extra fields will be ignored.

- Use `interface{}` to decode JSON string as it is.

```
var f interface{}
bytes := []byte(‘{"Foo":"Bar", "Number":1313, "Tags":{"A":"B"}}‘)
err := json.Unmarshal(bytes, &f)
fmt.Println(f)
// map[Number:1313 Foo:Bar Tags:map[A:B]]
```

  - `f` has a type of `map[string]interface{}`, enabling a recursive tree-like data structure for arbitrary JSON data.

- Mapping between struct and JSON string can be customized via struct field tags (like annotations in Java).
- YAML strings are handled similarly.

Outline

Manageability and Application Configuration

Manageability in Cloud Systems

Layers of Configurations

- Default values
  - User shouldn't need to configure everything.
  - Default behavior should be reasonable and unsurprising, e.g. typical behavior out-of-the-box, safe security setting, limited CPU and memory usage.
  - Allow user to gradually learn the options to config.
- Overrides
  - Make the preferences clear when multiple sources of configurations are available, e.g. command-line arguments override environment variables, which override default values.
  - Make the preferences consistent across development and production environments.
  - Provide useful feedback on where a value comes from, especially when misconfiguration happens.

Central Configuration Store

- Provide source of truth for configurations across nodes in a distributed system.
- For example, Kubernetes use etcd
  - Distributed key-value store where configuration values can be obtained.
  - Use consensus algorithm to guarantee consistency when configurations are updated on the fly.
  - Not partition tolerant since it depends on a majority quorum.

Reloading Configurations

- Should we reload configurations when they change?
- No for simplicity: kill and restart
- Yes for to minimize downtime
  - Applications and services may watch for updates automatically combining OS filesystem notifications, polling, and hashing.
  - Or be notified via the SIGHUP signal as a convention: this signal was used to notify the terminal of a process is closed – but for service processes without a terminal, it never happens so we reuse the signal for configuration updates.

Feature Management

- Allow control of program features and flows.
  - Enable experimental features conditionally for testing.
  - Adjust features like algorithms according to use cases.
  - Reduce need to deploy multiple versions when rollout new features.
- Feature flags: enable/disable features via configurations
  - Manage different code versions in one code base, encouraging smaller and faster iterations.
  - Integrate with resilience patterns like circuit breaker to automatically turn on and off.
  - Control feature rollouts to specific users.
- Scripting: complete control of features and flows.
  - For very complicated applications, e.g. mods for games and Tcl scripts for EDA tools.
  - Separate execution flow and features from program binary.
  - Very flexible – nevertheless, it blurs the boundary between manageability and maintainability.

Secrets

- Secrets: passwords, private keys, API keys, tokens
- Must not appear in
  - Source files and source code repositories
  - Container images
  - Logs
- Prefer to config secrets via environment variables
  - Incorrect permissions may make files readable for everyone.
  - Command-line arguments are visible when inspecting processes and containers.

Summary

- Make your application configurable via command-line flags and environment variable, as well as configuration files.