Automating architecture diagrams

Elena Grahovac

GopherCon Russia 2018

Elena Grahovac

webdeva

rumyantseva

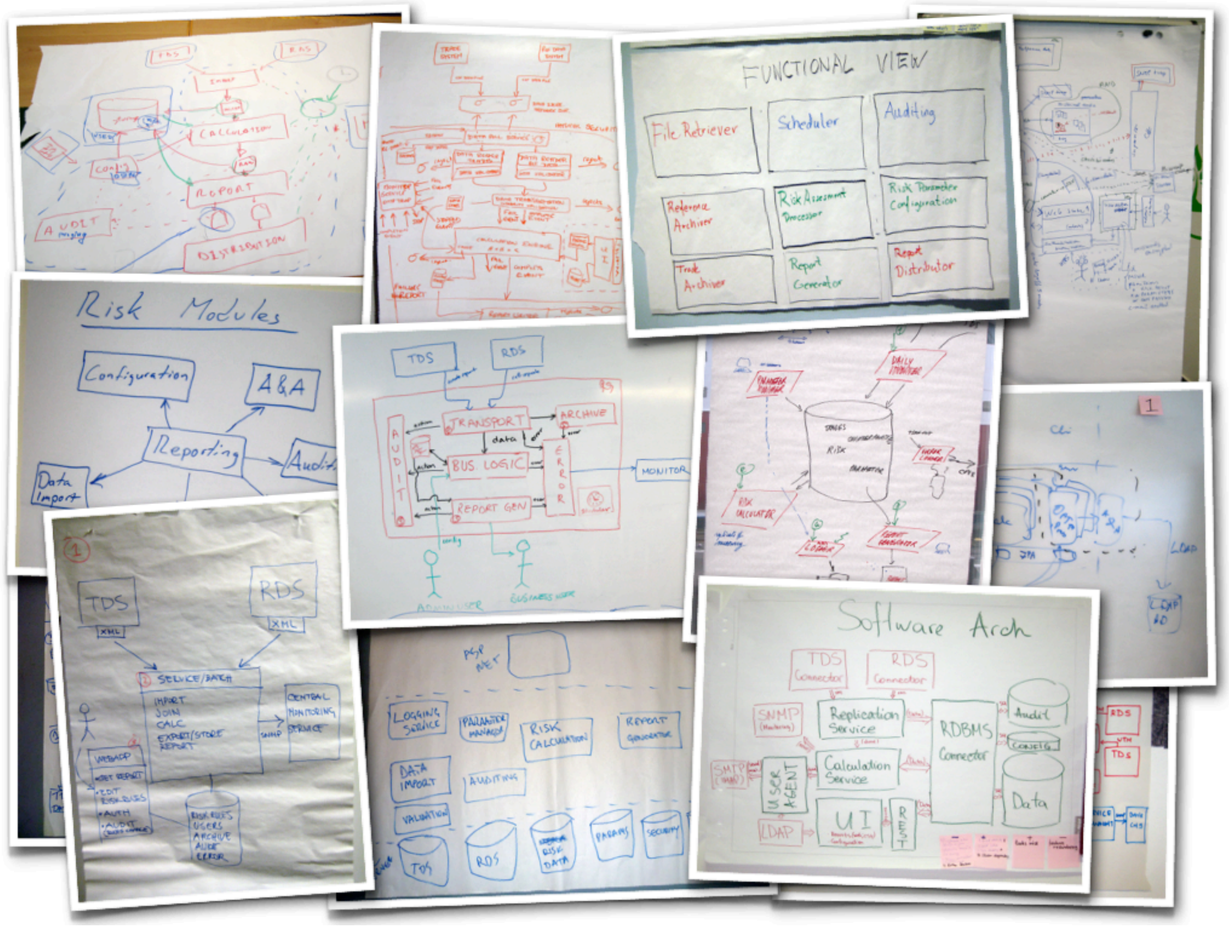
Software Engineer @ DCMN

Co-host @ GolangShow

First day with a new team

- visit a workspace
- meet the colleagues
- participate in the on-boarding sessions
- have a look at the source code
- **explore the architecture**

Explore the architecture...



Source: https://leanpub.com/visualising-software-architecture/read

How to visualise architecture?

- paint a whiteboard sketch
- write knowledge base page
- prepare a map built by some metadata
- receive data from a traffic manager
- collect and store data from the endpoints

Whiteboard sketch

Whiteboard sketch

Pros:

- doesn’t require extra tools
- easy to reproduced
- can show the whole picture step-by-step

Cons:

- require some time to be reproduced
- hard to provide enough details
- may contain some critical errors
- almost always outdated

Metadata

Todo-list

- define all services, cronjobs and other resources via metafiles
- interpret additional resources as services (#12FA)
- get information from the metafiles and visualise the graph

What to define?

- resource type
- unique name
- group (optional)
- dependencies with other resources
- additional information

Resource type

- service
- cronjob
- manual tool
- storage or data bus
- 3rd party service

Unique name

- simple and understandable
- reusable for links
- “semantic identifier”

Example:

- a good one: *event_tracker*
- a bad one: *df*

Group

- combine different modules in one service
- combine different resources by attributes
- reusable for links
- “semantic identifier”

Dependencies

- resource name (id)
- dependency type (e.g.: read-only, read/write)
- additional attributes (protocols, how critical link is, etc)

Additional information

- source code, specification, documentation, monitoring links
- resource owner
- comments
- instance id
- other attributes

Metadata

Pros:

- makes a description of resources and dependencies structured
- metafiles can be defined where you really need them
- helps to identify as many sublevels and “subservices” as you need
- you can build the diagram at every moment of time
- sometimes you can reuse existing metadata (e.g. k8s manifests)

Metadata

Cons:

- metafiles should be created and updated manually as an additional action
- need to write a tool to go through the existing source code and look for metafiles
- need to find a way to visualise the results

When it might be interesting?

- if there are quite a lot of code and you want to understand how it really works
- “software archaeology”
- if you want to document all kind of resources including cronjobs and manual jobs
- if you want to make an architecture diagram without running services

How it might look like: yaml

```yaml
name: user-authenticator
type: service
dependencies:
  - name: user-db
    critical: true
    protocol: postgresql
    type:
      - r
      - w
  - name: microsoft-ad-oauth
    critical: false
    protocol: oauth
    type: r
```

```yaml
name: user-db
type: database
title: PostgreSQL database to
store user authentication data
```

```yaml
name: microsoft-ad-oauth
type: thirdparty
title: Users may be authenticate
with Microsoft Azure AD OAuth
```

How it might look like: Go

```go
type ResourceType string

const (
    ResourceTypeService ResourceType = "service"
    ResourceTypeCronJob ResourceType = "cronjob"
    ResourceTypeManualJob ResourceType = "manual"
    ResourceTypeDB ResourceType = "database"
    ResourceTypeQueue ResourceType = "queue"
)

type DependencyType string

const (
    DependencyTypeRead DependencyType = "r"
    DependencyTypeWrite DependencyType = "w"
)
```

```go
type Resource struct {
    Name string `yaml:"name"`
    Type []ResourceType `yaml:"type"`
    Title string `yaml:"title"`
    Dependencies []Dependency `yaml:"dependencies"`
}

type Dependency struct {
    Name string `yaml:"name"`
    Type DependencyType `yaml:"type"`
    Critical bool `yaml:"critical"`
}
```

How it might look like: k8s

```yaml
apiVersion: v1
kind: Service
metadata:
  name: user-authenticator
  labels:
    app: user-authenticator
...
```

Traffic manager

Overview

- the highest level of automation: let’s not define any additional metadata but reuse existing software
- a traffic management system or a container management system…
- …or even a monitoring system
- works well for services but probably not for cronjobs

Endpoints

Typical configuration: a service

```yaml
hostname: localhost
port: “8885"

db_url: postgres://user:pass@dbsrv/users

oauth:
  auth_url: https://myurl/authorize
  token_url: https://myurl/token
```

```
type C struct {
    Hostname    string         `yaml:"hostname"`
    Port        string         `yaml:"port"`
    PostgresURL string         `yaml:”db_url"`
    OAuthEP     OAuthEndpoint `yaml:"oauth"`
}

type OAuthEndpoint struct {
    AuthURL   string `yaml:”auth_url”`
    TokenURL  string `yaml:”token_url"`
}
```

Typical configuration

```
type C struct {
    Hostname    string          `yaml:"hostname"`
    Port        string          `yaml:"port"`
    PostgresURL string          `yaml:”db_url"`
    OAuthEP     OAuthEndpoint `yaml:"oauth"`
}

type OAuthEndpoint struct {
    AuthURL   string `yaml:”auth_url”`
    TokenURL  string `yaml:”token_url"`
}
```

```
type Resource struct {
    Name string `yaml:"name"`
    Type []ResourceType `yaml:"type"`
    Title string `yaml:"title"`
    Dependencies []Dependency
`yaml:"dependencies"`
}

type Dependency struct {
    Name string `yaml:"name"`
    Type DependencyType `yaml:"type"`
    Critical bool `yaml:"critical"`
}
```

Push vs Pull

Push:

- receives configuration from resources during their release or runtime phase
- works for everything: services, cronjobs, other things
- requires some monitoring in a case if a resource was removed

Pull:

- asks resources in real-time
- requires a list of resources (e.g. service discovery)
- may also collect health check results and other additional information
- requires some monitoring in a case if a resource isn't available
- doesn't work for cron-/manual jobs

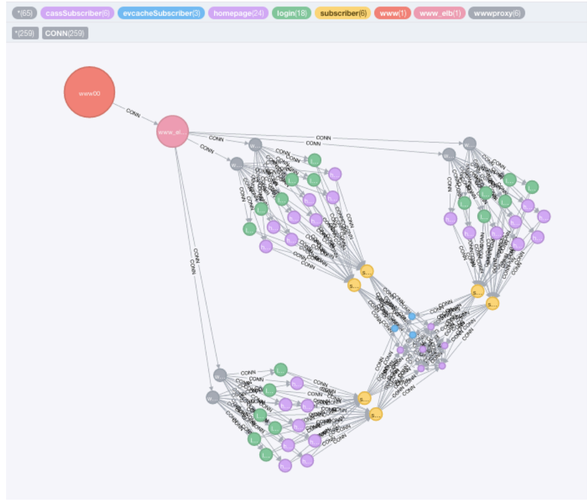
Combination of approaches

- we can combine metadata, traffic management and push-/pull-approaches…
- ...but the fewer, the better!

Example: automating architecture diagram service may go through the source code to only find services and use push-/pull- approach for the rest

Visualisation

- we have a structure, and we can paint it!
- graph databases and tools (neo4j, gephy, graphviz, spigo, ...)
- pros and cons



Source: https://raw.githubusercontent.com/adrianco/spigo/master/tooling/graphneo4j/neo4jnetflix.png

Conclusion

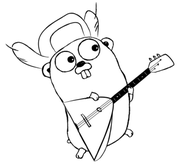
- Automating architecture diagrams might help to solve at least two problems: the visualisation itself and structuring of the project
- Working on the dependency graph, you might find some issues: e.g. hardcoded dependencies, unused dependencies and cycled dependencies, unstructured configurations
- Let's automate it!

Elena Grahovac

webdeva

rumyantseva

golangshow.com



slack.golang-ru.com

These slides and examples:

https://github.com/rumyantseva/gophercon-ru-2018