Understanding Infrastructure as Code

By Michael Wittig and Andreas Wittig

In this article, excerpted from Amazon Web Service in Action, we will explain Infrastructure as Code.

Infrastructure as Code describes the idea of using a high level programming language to control IT systems. In software development, tools like automated tests, code repositories and build servers are increasing quality of software engineering. If your infrastructure can be treated as code you can apply the same techniques to infrastructure code than to your application code. In the end, you will improve the quality of your infrastructure by using automated tests, code repositories and build servers.

WARNING	**Infrastructure as Code is not Infrastructure as a Service** **Don't mix the two terms! Infrastructure as a Service (IaaS) is renting servers, storage, network on a pay-per-use pricing model.**

Automation and DevOps movement

DevOps is a method driven by software development to bring development and operations closer together. The goal is to deliver rapidly developed software to the customer without a negative impact on quality. Communication and collaboration between development and operations is therefore necessary.

Multiple deploys per day are only possible if your pipeline from code changes to deployment is fully automated. If you commit into the repository, the source code is automatically built and tested against your automated tests. If the build passed the tests, it is automatically installed in your testing environment. Maybe some integration tests are triggered now. After the integration tests have passed, the change is propagated into production. But this is not the end of the process. Now you need to carefully monitor your system and analyze your logs in real time to ensure that the change was successful.

If your infrastructure is automated, you can spawn a new system for every change introduced to the code repository and run the integration tests isolated from other changes that were pushed to the repository at the same time. So whenever a change is made to the cod[MW1], a new system is created (servers, databases, networks, etc.) to run the change in isolation.

Inventing an infrastructure language: JIML

For the purpose of understanding Infrastructure as Code in detail, you will invent a new language to describe infrastructure. The language is named JSON Infrastructure Markup Language better known as JIML. Figure 1 shows the infrastructure that will be created in the end.

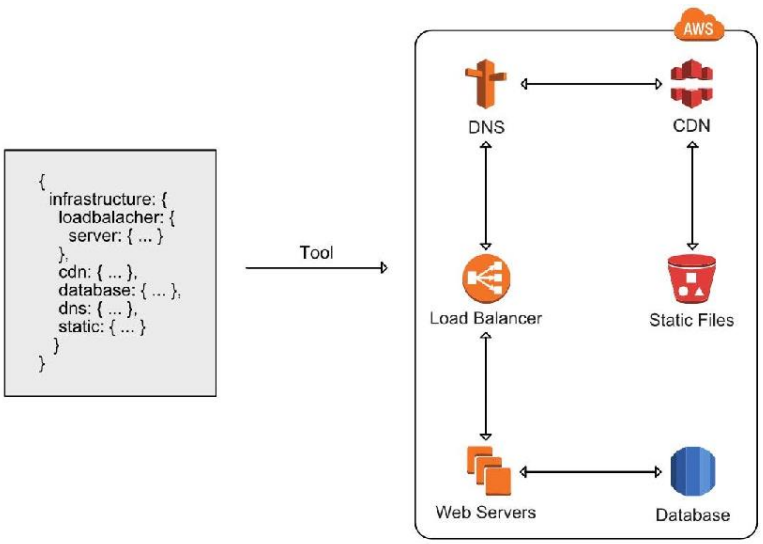

Figure 1 From JIML blueprint to infrastructure: infrastructure automation

The infrastructure consists of:

- Load balancer (LB)
- Virtual servers

- Database (DB)
- DNS entry
- Content Delivery Network (CDN)
- Bucket for static files

To reduce issues with syntax, let's say that we based JIML on JSON. The following JIML program creates the infrastructure shown in figure 1. The $ sign is used to indicate a reference to an id.

Listing 1 Infrastructure description in JSON Infrastructure Markup Language (JIML)

```
{
  "region": "us-east-1",
  "resources": [{
    "type": "loadbalancer",
    "id": "LB",
    "config": {
      "server": {
        "cpu": 2,
        "ram": 4,
        "os": "ubuntu"
      },
      "servers": 2
    },
    "waitFor": "$DB"
  }, {
    "type": "cdn",
    "id": "CDN",
    "config": {
      "defaultSource": "$LB",
      "sources": [{
        "path": "/static/*",
        "source": "$BUCKET"
      }]
    }
  }, {
    "type": "database",
    "id": "DB",
    "config": {
      "password": ***",
      "engine": "MySQL"
    }
  }, {
    "type": "dns",
    "config": {
      "from": "www.mydomain.com",
      "to": "$CDN"
    }
  }, {
```

```
    "type": "bucket",
    "id": "BUCKET"
  }]
}
```

How can this JSON be turned into AWS API calls?

1. The JSON input is parsed.
2. The JIML interpreter creates a dependency graph by connecting the resources with their dependencies.
3. The JIML interpreter derives a linear flow of commands from the dependency graph by traversing the tree from the bottom (leafs) to the top (root). The commands are expressed in a pseudo language.
4. The commands in pseudo language are translated into AWS API calls by the JIML runtime.

Let's have a look at the dependency graph created by the JIML interpreter in figure 2.

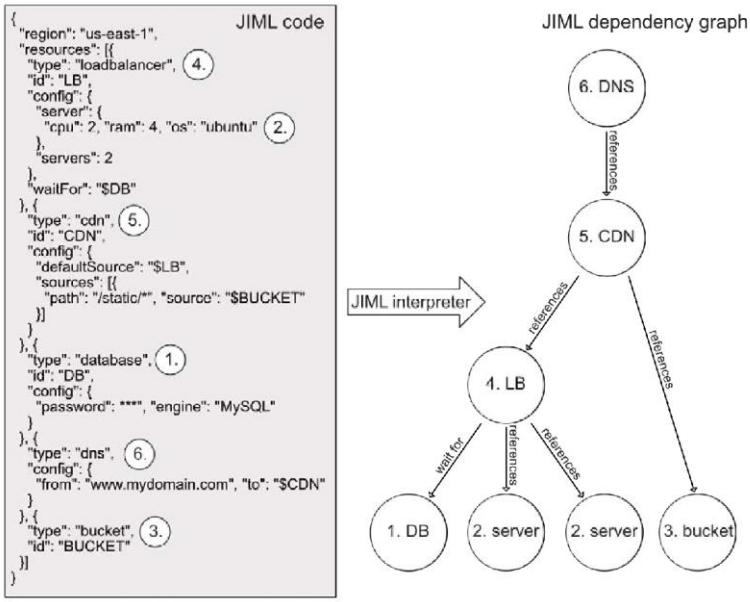

Figure 2 JIML interpreter figures out the order in which resources needs to be created

Traverse the dependency graph from bottom to top, from left to right. The nodes at the bottom have no children like DB (1), servers (2) and bucket (3). Nodes without children have no dependencies. The LB (4) node depends on the DB node and the two servers nodes. The CDN (5) node depends on the LB and bucket node. Finally the DNS (6) node depends on the LB node.

The JIML interpreter now turns the dependency graph into a linear flow of commands using a pseudo language. The pseudo language represents the steps that are needed to create all the resources in the correct order. The nodes at the bottom have no dependencies and are therefore easy to create. That's why they are created first.

Listing 2 Linear flow of commands in a pseudo language is derived from the dependency graph

```
$DB = database create {"password": ***", "engine": "MySQL"} (1)

$SERVER1 = server create {"cpu": 2, "ram": 4, "os": "ubuntu"} (2)$SERVER2 =
server create {"cpu": 2, "ram": 4, "os": "ubuntu"}

$BUCKET = bucket create {} (3)

await [$DB, $SERVER1, $SERVER2] (4)$LB = loadbalancer create {"servers":
[$_SERVER1, $_SERVER2]} (5)

await [$LB, $BUCKET] $CDN = cdn create
{...} (6)

await $CDN $DNS = dns create {...} (7)

await $DNS
```

1. create database
2. create servers
3. create bucket
4. wait for the dependencies
5. create load balancer
6. create CDN
7. create DNS entry

The last step, translating the commands of the pseudo language into AWS API calls, is skipped. You already learned everything you need to know about Infrastructure as Code: It's all about dependencies.