The Heart of Juju

by Gustavo Niemeyer, Canonical

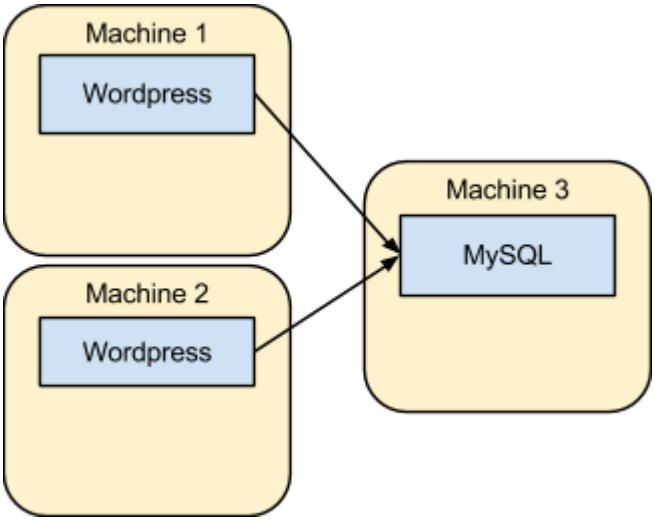
Contents

This post is long enough to deserve an index, but these sections do build up concepts incrementally, so for a full understanding sequential reading is best:

Classical deployments

In a simplistic sense, deploying an application means configuring and running a set of processes in one or more machines to compose an integrated system. This procedure includes not only configuring the processes for particular needs, but also appropriately interconnecting the processes that compose the system.

The following figure depicts a simple example of such a scenario, with two frontend machines that had the Wordpress software configured on them to serve the same content out of a single backend machine running the MySQL database.



Deploying even that simple environment already requires the administrator to deal with a

variety of tasks, such as setting up physical or virtual machines, provisioning the operating system, installing the applications and the necessary dependencies, configuring web servers, configuring the database, configuring the communication across the processes including addresses and credentials, firewall rules, and so on. Then, once the system is up, the deployed system must be managed throughout its whole lifecycle, with upgrades, configuration changes, new services integrated, and more.

The lack of a good mechanism to turn all of these tasks into high-level operations that are convenient, repeatable, and extensible, is what motivated the development of juju. The next sections provide an overview of how these problems are solved.

Preparing a blank slate

Before diving into the way in which juju environments are organized, a few words must be said about what a juju environment is in the first place.

All resources managed by juju are said to be within a juju environment, and such an environment may be prepared by juju itself as long as the administrator has access to one of the supported infrastructure providers (AWS, OpenStack, MAAS, etc).

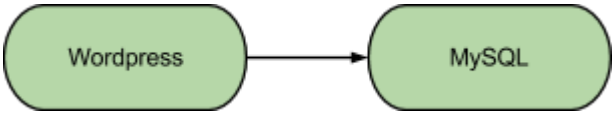
In practice, creating an environment is done by running juju's bootstrap command:

```
$ juju bootstrap
```

This will start a machine in the configured infrastructure provider and prepare the machine for running the juju state server to control the whole environment. Once the machine and the state server are up, they'll wait for future instructions that are provided via follow up commands or alternative user interfaces.

Service topologies

The high-level perspective that juju takes about an environment and its lifecycle is similar to the perspective that a person has about them. For instance, although the classical deployment example provided above is simple, the mental model that describes it is even simpler, and consists of just a couple of communicating services:

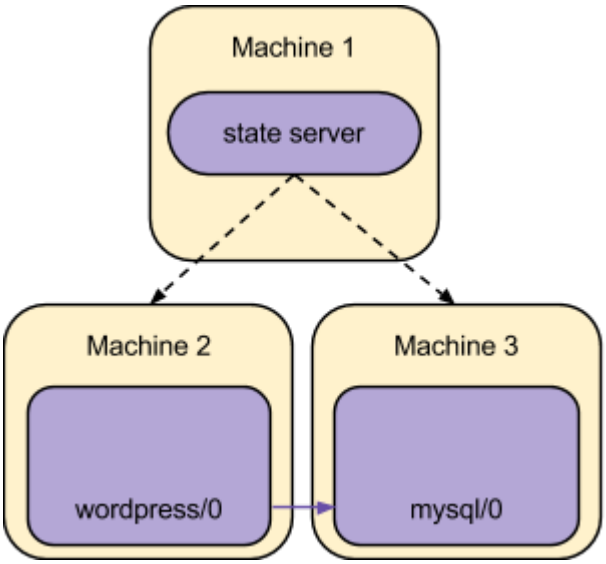


That's pretty much the model that an administrator using juju has to input into the system for that deployment to be realized. This may be achieved with the following commands:

```
$ juju deploy cs:precise/wordpress
$ juju deploy cs:precise/mysql
$ juju add-relation wordpress mysql
```

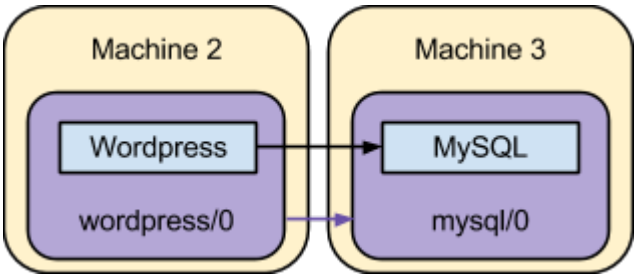
These commands will communicate with the previously bootstrapped environment, and will input into the system the desired model. The commands themselves don't actually change the current state of the deployed software, but rather inform the juju infrastructure of the state that the environment should be in. After the commands take place, the juju state server will act to transform the current state of the deployment into the desired one.

In the example described, for instance, juju starts by deploying two new machines that are able to run the service units responsible for Wordpress and MySQL, and configures the machines to run agents that manipulate the system as needed to realize the requested model. An intermediate stage of that process might conceptually be represented as:



The service units are then provided with the information necessary to configure and start the real software that is responsible for the requested workload (Wordpress and MySQL themselves, in this example), and are also provided with a mechanism that enables service units that were related together to easily exchange data such as addresses, credentials, and so on.

At this point, the service units are able to realize the requested model:



This is close to the original scenario described, except that there's a single frontend machine running Wordpress. The next section details how to add that second frontend machine.

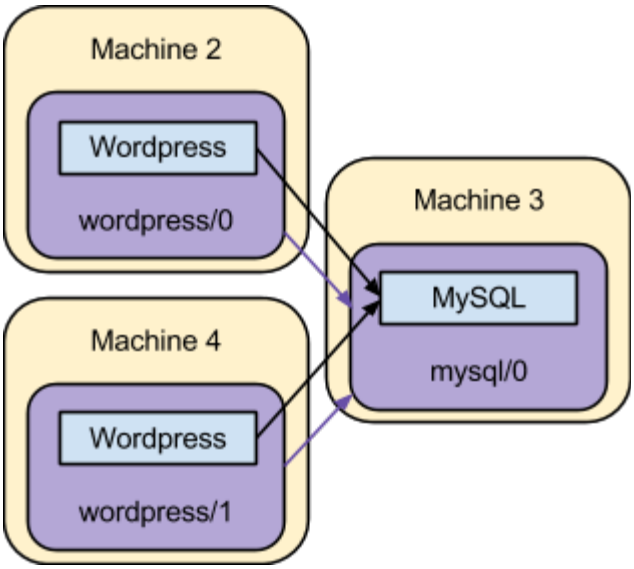
Scaling services horizontally

The next step to match the original scenario described is to add a second service unit that can run Wordpress, and that can be achieved by the single command:

```
$ juju add-unit wordpress
```

No further commands or information are necessary, because the juju state server understands what the model of the deployment is. That model includes both the configuration of the involved services and the fact that units of the wordpress service should talk to units of the mysql service.

This final step makes the deployed system look equivalent to the original scenario depicted:



Although that is equivalent to the classic deployment first described, as hinted by these examples an environment managed by juju isn’t static. Services may be added, removed, reconfigured, upgraded, expanded, contracted, and related together, and these actions may take place at any time during the lifetime of an environment.

The way that the service reacts to such changes isn’t enforced by the juju infrastructure. Instead, juju delegates service-specific decisions to the charm that implements the service behavior, as described in the following section.

Charms

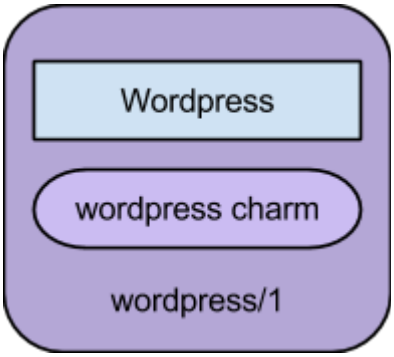
A juju-managed environment wouldn't be nearly as interesting if all it could do was constrained by preconceived ideas that the juju developers had about what services should be supported and how they should interact among themselves and with the world.

Instead, the activities within a service deployed by juju are all orchestrated by a juju charm, which is generally named after the main software it exposes. A charm is defined by its metadata, one or more executable hooks that are called after certain events take place, and optionally some custom content.

The charm metadata contains basic declarative information, such as the name and description of the charm, relationships the charm may participate in, and configuration options that the charm is able to handle.

The charm hooks are executable files with well-defined names that may be written in any language. These hooks are run non-concurrently to inform the charm that something happened, and they give a chance for the charm to react to such events in arbitrary ways. There are hooks to inform that the service is supposed to be first installed, or started, or configured, or for when a relation was joined, departed, and so on.

This means that in the previous example the service units depicted are in fact reporting relevant events to the hooks that live within the wordpress charm, and those hooks are the ones responsible for bringing the Wordpress software and any other dependencies up.



The interface offered by juju to the charm implementation is the same, independently from which infrastructure provider is being used. As long as the charm author takes some care, one can create entire service stacks that can be moved around among different infrastructure providers.

Relations

In the examples above, the concept of service relationships was introduced naturally, because it's indeed a common and critical aspect of any system that depends on more than a single process. Interestingly, despite it being such a foundational idea, most management systems in fact pay little attention to how the interconnections are modeled.

With juju, it's fair to say that service relations were part of the system since inception, and have driven the whole mindset around it.

Relations in juju have three main properties: an *interface*, a *kind*, and a *name*.

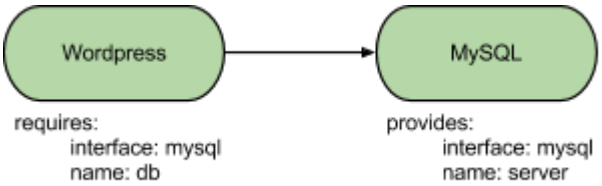
The relation *interface* is simply a unique name that represents the protocol that is conventionally followed by the service units to exchange information via their respective hooks. As long as the name is the same, the charms are assumed to have been written in a compatible way, and thus the relation is allowed to be established via the user interface. Relations with different interfaces cannot be established.

The relation *kind* informs whether a service unit that deploys the given charm will act as a provider, a requirer, or a peer in the relation. Providers and requirers are complementary, in the sense that a service that provides an interface can only have that specific relation established with a service that requires the same interface, and vice-versa. Peer relations are automatically established internally across the units of the service that declares the relation, and enable easily clustering together these units to setup masters and slaves, rings, or any other structural

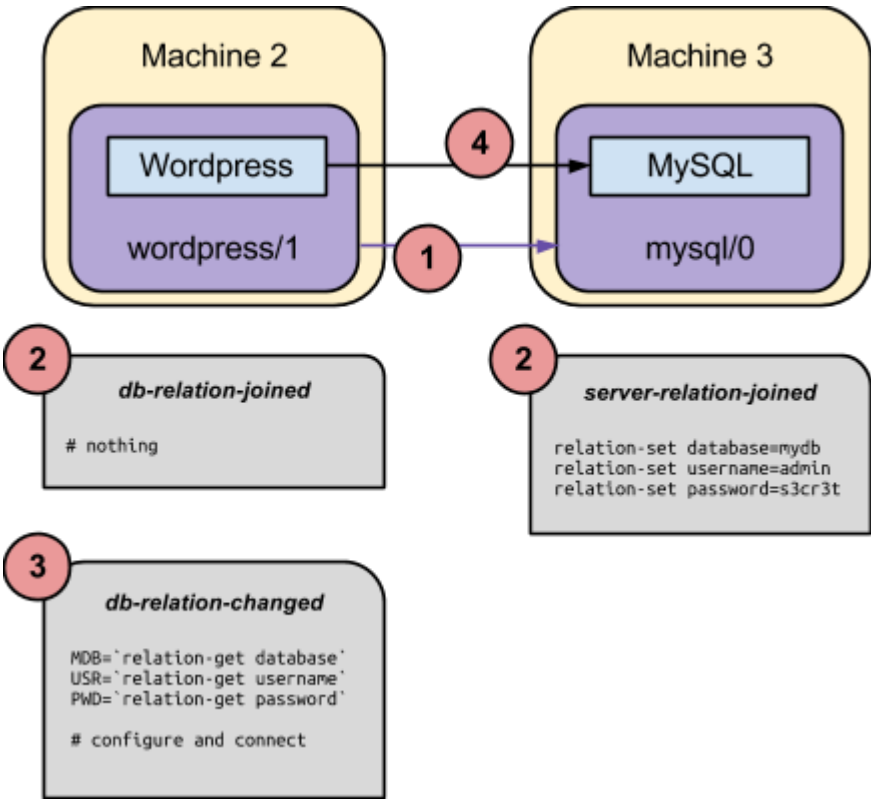
organization that the underlying software supports.

The relation *name* uniquely identifies the given relation within the charm, and allows a single charm (and service and service units that use it) to have multiple relations with the same interface but different purposes. That identifier is then used in hook names relative to the given relation, user interfaces, and so on.

For example, the two communicating services described in examples might hold relations defined as:



When that service model is realized, juju will eventually inform all service units of the wordpress service that a relation was established with the respective service units of the mysql service. That event is communicated via hooks being called on both units, in a way resembling the following representation:



As depicted above, such an exchange might take the following form:

1. The administrator establishes a relation between the wordpress service and the mysql service, which causes the service units of these services (wordpress/1 and mysql/0 in the example) to relate.
2. Both service units concurrently call the relation-joined hook for the respective relation. Note that the hook is named after the local relation name for each unit. Given the conventions established for the mysql interface, the requirer side of the relation does nothing, and the provider informs the credentials and database name that should be used.

3. The requirer side of the relation is informed that relation settings have changed via the relation-changed hook. This hook implementation may pick up the provided settings and configure the software to talk to the remote side.
4. The Wordpress software itself is run, and establishes the required TCP connection to the configured database.

In that workflow, neither side knows for sure what service is being related to. It would be feasible (and probably welcome) to have the mysql service replaced by a mariadb service that provided a compatible mysql interface, and the wordpress charm wouldn't have to be changed to communicate with it.

Also, although this example and many real world scenarios will have relations reflecting TCP connections, this may not always be the case. It's reasonable to have relations conveying any kind of metadata across the related services.

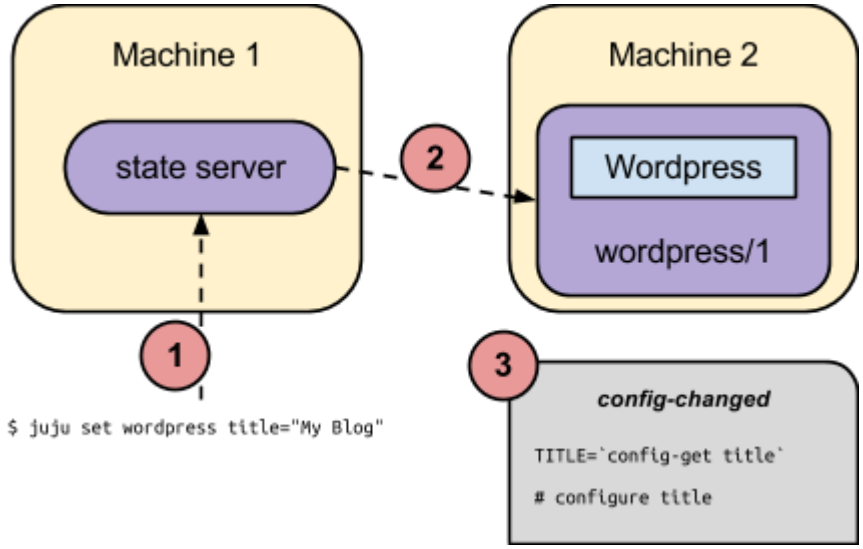
Configuration

Service configuration follows the same model of metadata plus executable hooks that was described above for relations. A charm can declare what configuration settings it expects in its metadata, and how to react to setting changes in an executable hook named *config-changed*. Then, once a valid setting is changed for a service, all of the respective service units will have that hook called to reflect the new configuration.

Changing a service setting via the command line may be as simple as:

```
$ juju set wordpress title="My Blog"
```

This will communicate with the juju state server, record the new configuration, and consequently incite the service units to realize the new configuration as described. For clarity, this process may be represented as:



Taking from here

This conceptual overview hopefully provides some insight into the original thinking that went into designing the juju project. For more in-depth information on any of the topics covered here, the following resources are good starting points:

- The project page
- Getting started guide
- In-depth technical documentation
- Overview presentation at Google I/O 2012