On Software Resources under the New Model
Classification and Support Strategy

We propose a model for `gCube` in which software resources are broadly classified with respect to their relationship to the system and their role in networked interactions. This classification defines a terminology for the design and documentation of the system.

We make a first distinction in role between:

- **services**: resources that accept requests made over the network;
- **clients**: resources that make requests to services;

A service may act as a client towards other services, while a client that does not act as a service is a **pure client**.

Further, as we do for other types or resources, we distinguish between resources as follows:

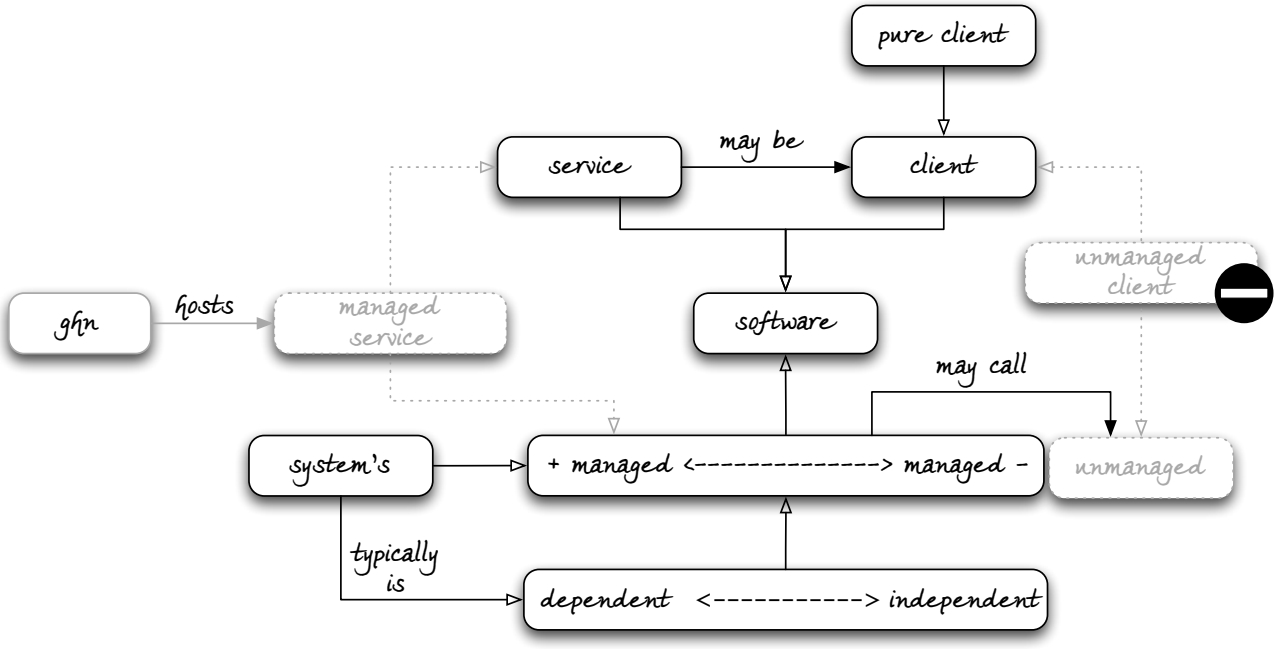
- **managed resources**: resources upon which the system applies some of its management functions. For services, management functions start with publications and controlled sharing. For clients, management functions focus on support for interactions with managed services. The deployment of managed services (static or dynamic) qualifies the target hardware resources as **gCube Hosting Nodes**.
- **unmanaged services**: services that lie outside the purview of the system but can be called, directly or indirectly, by managed services. Intuitively, these are resources made available to infrastructures based on deployments of the system.

Unmanaged clients, i.e. clients of unmanaged services, are of no interest to the system.

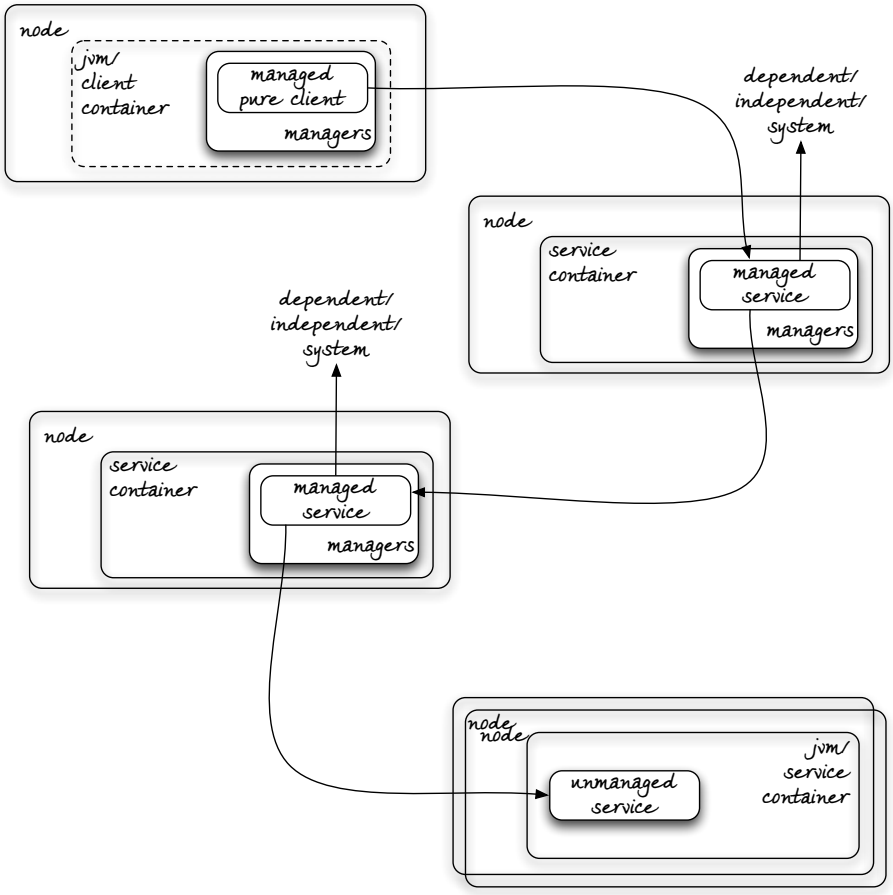
Finally, we distinguish between managed resources as follows:

- **system resources**: resources that are included in system releases. These include (pure) clients and services.
- **dependent resources**: resources that have compile-time dependencies to system resources. System resources normally have inter-dependencies, hence they are typically dependent resources. However they do not *have* to be.
- **independent resources**: resources that have no compile-time dependencies to system resources.

The classification above is informally captured in the following diagram:



A dynamic view of some interactions between resources is provided by the following diagram:



A key goal in the design of the system is to broaden the class of managed resources, in particular the the management functions that can be offered over independent resources. This is essentially a requirement for **transparent resource management**. In its ideal form, the requirement is for **zero-dependency management**, though this in practice may be attained only for a small subset of management functions and, for some resources, it may not be attained at all. When zero-dependency management is unattainable for all or some management functions, we pursue transparency with solutions that minimise required dependencies.

In all cases, transparent management requires lightweight integration mechanisms with existing technologies. We consider requirements for service configuration (hence re-packaging) to fall within our zero-dependency management goal. For clients, configuration requirements *may* not arise at all.

We prioritise integration with `Java` technologies and, within those, with formal and de-facto standards. We fall back to ad-hoc integration whenever standards are not available. Within standards, we prioritise those that, in theory or in practice, assume the `HTTP` protocol, leaving integration based on other protocols as a future goal.

For services, we prioritise integration with resources deployed in `Servlet` containers, from interactive Web Applications to Web Services. In particular, we focus on plain `HTTP` services (`REST` services) and `SOAP` services (`WS` services). For `Rest` services, we prioritise integration with `JAX-RS`, but also consider `Restlet` and `ReastEasy` as de-facto standards. For `WS` services, we prioritise integration with `JAX-WS`. In all cases, we seek uniform integration solutions based on the listeners and filters defined by the `Servlet` specification, taking particularly advantage of the annotation-based configuration introduced in version `3.0`.

For clients, the only `Java` standard at the time of writing is the `API java.net`, which is a low-level client `API`. Future versions of `JAX-RS` are expected to standardise a high-level client `APIs`. Apache's `HTTPClient` is another low-level `APIs` that serves as a de-facto standard. Recently, the high-level

client APIs offered by Jersey (JAX-RS's RI), Restlet, and RestEasy are increasingly popular as configurable abstractions over java.net and/or Apache's HttpClient APIs. Here, our integration strategy relies mainly on an embedded HTTP proxy server bootstrapped by an instrumentation agent of the JVM.