Capítulo 2

Cloud computing Fundamentos

Javier Esparza Peidro - jesparza@dsic.upv.es

Contenido

2.1. Introducción

Cloud computing es un modelo de computación que proporciona recursos computacionales de manera remota. Los recursos computacionales internamente se implementan por medio de diversas técnicas de *virtualización* y son consumidos a través de *servicios*.

En las siguientes secciones primero se analiza el concepto de servicio y las arquitecturas orientadas a servicios (SOA). Después se revisa el concepto de virtualización, los distintos tipos de virtualización y se pone énfasis en la virtualización hardware, que es una tecnología clave para habilitar cloud computing.

2.2. SOA

Las arquitecturas orientadas a servicios o SOA (*Service Oriented Architecture*) surgen para dar solución a los problemas que se originan al integrar distintas aplicaciones para compartir datos. La idea principal consiste en romper el sistema en subsistemas independientes, denominados servicios, cada uno de los cuales suministra un subconjunto de capacidades bien definidas y las expone a través de una interfaz estable, basada en tecnologías estándar que favorecen la interoperabilidad.

Existen diversas tecnologías que permiten implantar esta estrategia. La que posee mayor recorrido son los servicios web, basados en los estándares WSDL/-SOAP. Esta aproximación incluye una gran cantidad de extensiones que permiten construir sistemas muy complejos, basados en transacciones distribuidas.

Como consecuencia de la complejidad de estas especificaciones y de las soluciones obtenidas, en los últimos tiempos ha proliferado una nueva generación de servicios RESTful, que procuran aplicar los principios de arquitectura que rigen la WWW a la construcción de sistemas.

Además de servicios web y servicios RESTful, existen otras alternativas que permiten implementar arquitecturas basadas en servicios. Algunos ejemplos serían

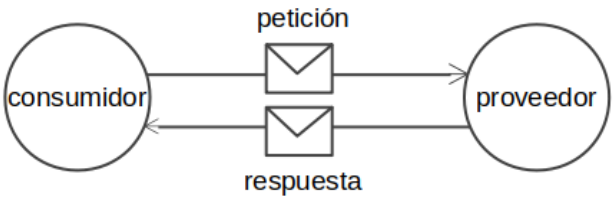


Figura 2.1: Comunicación entre servicios

gRPC[1], que es una tecnología muy eficiente basada en llamada a procedimiento remoto, o GraphQL[2], que promete mejorar la aproximación REST.

2.3. Servicios

Un servicio es un fragmento de software que ofrece una colección de *capacidades*, que son accesibles a través de un *contrato de servicio*. El contrato de servicio describe, entre otras cosas, la interfaz del servicio (su API).

Las capacidades suministradas por el servicio suelen estar relacionadas por un contexto común (contexto funcional), que puede ser un proceso de negocio, una entidad de negocio, etc. El servicio encapsula la lógica necesaria para implementar dicha colección de capacidades.

En función del alcance de la funcionalidad suministrada, los servicios presentan distinta *granularidad*. Un servicio con granularidad fina suministra funciones muy concretas y acotadas, mientras que un servicio con granularidad gruesa proporciona funciones con gran alcance. En general, se recomienda diseñar servicios de grano grueso para reducir la complejidad de la arquitectura (se requieren menos servicios para implementar un determinado proceso de negocio) y optimizar el uso de la red.

El servicio es ofrecido por un *proveedor* y utilizado por un *consumidor*. El rol de consumidor puede ser adoptado por cualquier fragmento de software, incluyendo otro servicio. Para favorecer el desacoplamiento, la comunicación entre servicios suele efectuarse por medio de paso de *mensajes* síncronos y/o asíncronos. El consumidor envía un mensaje al proveedor, solicitando una capacidad publicada en el contrato de servicio. El proveedor procesa la petición y podrá responder con un mensaje o no. La figura 2.1 ilustra los intercambios que tienen lugar usando este mecanismo de comunicación.

La comunicación entre consumidor y proveedor puede efectuarse de manera directa (i.e. el consumidor accede directamente al proveedor) o de manera in-

[1] https://grpc.io
[2] https://graphql.org

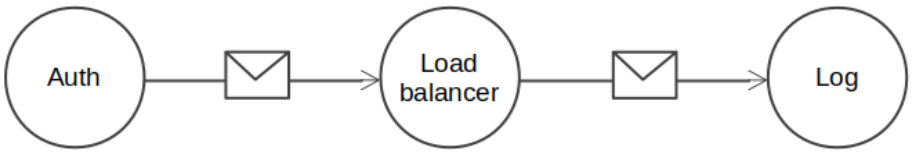


Figura 2.2: Intermediario entre servicios

directa, por medio de *intermediarios*. Los intermediarios enrutan los mensajes a sus destinatarios, y pueden ser pasivos, si no modifican el contenido del mensaje (e.g. registran en un log los mensajes, efectúan un balanceo de carga, ...), o activos, si pueden modificar el contenido del mensaje (e.g. encriptación, compresión, ...). Estos intermediarios suelen recibir la denominación de *agentes*. La figura 2.2 describe dos servicios interconectados por medio de un intermediario, que actúa a modo de balanceador de carga.

La orientación a servicios y las arquitecturas SOA no están ligadas a ninguna plataforma tecnológica. En realidad, cualquier tecnología capaz de construir componentes distribuidos podría utilizarse para implementar servicios.

2.4. Principios de diseño

Un principio de diseño es una práctica generalizada y aceptada con el objetivo de obtener características deseables en el diseño del software. El paradigma orientado a servicios comprende 8 *principios de diseño* fundamentales [Erl, 2007], que están interrelacionados, es decir, aplicar un principio de diseño posee un efecto directo sobre el resto:

- *Contrato de servicio estándar.*

  Los servicios expresan su propósito y sus capacidades por medio de un contrato. El contrato debe ser autodescriptivo, y debería incluir la siguiente información: las capacidades del servicio, sus limitaciones, su interfaz (o su API), las garantías de calidad (QoS) que ofrece, etc.

  La interfaz del servicio recoge las operaciones publicadas por el servicio, así como los datos de entrada y salida. El esquema que siguen los datos intercambiados también debe ser definido. Las garantías de calidad de servicio (QoS - Quality of Service) se describen en SLAs (Service Level Agreement).

  Todos los servicios incluidos en un mismo inventario deben adherirse a los mismos estándares de diseño, favoreciendo la interoperabilidad entre ellos.

- *Bajo acoplamiento.*

El acoplamiento determina el nivel de dependencia entre dos cosas. El contrato de servicio desacopla la interfaz del servicio de su implementación. Además, el contrato no debe imponer dependencias sobre sus consumidores, ni sobre su implementación. De este modo, el servicio puede evolucionar de manera independiente, sin afectar a sus consumidores, o a su implementación.

- *Abstracción.*

  El contrato de servicio actúa de interfaz entre los consumidores del servicio y su implementación. El contrato debe contener únicamente la información imprescindible sobre las capacidades implementadas por el servicio, y cómo es posible acceder a ellas. Cualquier otro detalle sobre la lógica que encapsula debería quedar oculto. De este modo, se favorece su reutilización y evolución independiente de su implementación.

- *Reusabilidad.*

  Las capacidades que exponen los servicios es lo suficientemente genérica para ser reutilizada en cualquier contexto de negocio, o con cualquier tecnología. Los servicios de entidad y de utilidad están especialmente concebidos para encapsular lógica reutilizable, ya que pueden ser utilizados en diferentes procesos de negocio. En cambio, los servicios de tarea soportan la lógica específica de los procesos de negocio, y hacen uso de los anteriores.

- *Autonomía.*

  La autonomía de un servicio mide su capacidad de auto-gobierno, es decir, de tomar decisiones sin requerir intervención externa. Un servicio autónomo posee un alto grado de control sobre el entorno en el que se ejecuta, así como los recursos que consume. Se reduce por tanto la dependencia sobre recursos ajenos sobre los que no se posee ningún control y que pueden suponer un origen de problemas.

  Con esta medida es posible obtener servicios más predecibles y fiables, y que evolucionan de manera independiente sin afectar a sus potenciales consumidores. Este principio está íntimamente relacionado con los principios de abstracción y bajo acoplamiento.

- *Sin estado.*

  Los servicios guardan el mínimo estado posible, delegando su gestión a una entidad externa especializada en el mantenimiento de datos (e.g. una base de datos externa). De este modo, se minimiza el consumo de recursos del servicio y se favorece su escalabilidad, pudiendo atender demandas

crecientes de trabajo. Sin embargo, delegar la gestión de datos a un entidad externa puede influir en el tiempo de respuesta del servicio y genera una dependencia adicional del servicio, incrementando su acoplamiento y reduciendo su autonomía.

- *Descubrimiento.*

  Los servicios deben ser descubribles. Para ello, los contratos de servicio deben ser complementados con metadatos que describan el servicio de manera consistente (incluyendo el propósito del servicio, sus capacidades y limitaciones), dichos metadatos deben ser registrados en repositorios indexables y se deben habilitar mecanismos de búsqueda sobre los repositorios para obtener aquellos servicios que se correspondan con los criterios de búsqueda.

  Es importante diferenciar el descubrimiento en tiempo de diseño y en tiempo de ejecución. En tiempo de diseño se buscan servicios compatibles para ser incluidos en un proceso de negocio en repositorios de servicios. En tiempo de ejecución se buscan los servicios que cumplen una determinada interfaz en registros de servicios.

- *Composición.*

  Los servicios pueden ser fácilmente agregados en soluciones más complejas, dando lugar a servicios compuestos. La efectividad de este principio depende en gran medida de la correcta aplicación del resto de principios de diseño. La existencia de un contrato estándar garantiza la interoperabilidad con otros servicios, servicios con bajo acoplamiento pueden combinarse entre sí sin esperar la aparición de dependencias no deseables entre ellos, si el servicio es autónomo y sin estado puede reutilizarse de manera fiable en múltiples composiciones. Por último, si los servicios son descubribles, un servicio compuesto puede modificar su composición de manera dinámica en función de diversos criterios.

2.5. Servicios RESTful

Los servicios RESTful surgen como alternativa a los servicios web SOAP, que son muy potentes pero también muy complejos, y poco eficientes. Los servicios RESTful permiten manipular un conjunto de recursos de manera remota utilizando tecnologías estándar. Además, verifican los principios de arquitectura REST, extraídos de las lecciones aprendidas con el sistema distribuido con más éxito de la historia, la WWW.

2.5.1. Principios de arquitectura

REST es el acrónimo de Representational State Transfer y aparece por primera vez en la tesis doctoral de Roy Fielding [Fielding, 2000], en la que se analizaban distintos estilos de arquitectura de aplicaciones distribuidas. En dicho trabajo, se utiliza el término REST para acuñar un estilo de arquitectura, cuyo principal representante es la WWW, y en la que se verifican un conjunto de principios bien definidos:

- Cliente-servidor: existen dos roles bien diferenciados, el cliente y el servidor. Ambos componentes evolucionan de manera independiente.
- Sin estado: es una restricción que se impone a las comunicaciones entre cliente y servidor. Cada petición procedente del cliente debe contener toda la información necesaria para ser totalmente comprendida por el servidor. No es posible confiar en que el servidor almacenará información de contexto entre peticiones consecutivas.
- Cache: para incrementar la eficiencia en las comunicaciones, las respuestas pueden ser cacheadas por elementos intermedios.
- Interfaz uniforme: todos los componentes ofrecen una interfaz homogénea y estándar, que permite actuar sobre una colección de recursos bien identificados, utilizando para ello un conjunto de operaciones autodescriptivas que permiten manipular representaciones de los recursos. Gracias a esta interfaz uniforme, resulta muy sencillo utilizar nuevos servicios. La contrapartida es la pérdida de eficiencia, pues no es posible optimizar los protocolos y formatos de manera específica.
- Sistema a capas: el sistema se organiza en capas jerárquicas, donde una capa únicamente puede acceder a su inmediata inferior. Esta estrategia simplifica la integración con sistemas legados, aunque puede añadir cierta sobrecarga a las comunicaciones.
- Código bajo demanda: se trata de una característica opcional, que permite extender las capacidades del cliente descargando código desde el servidor. Aunque facilita la actualización y mejora continua del cliente, reduce la visibilidad del sistema.

En la actualidad, si un servicio verifica los principios de arquitectura REST se dice que es un servicio RESTful. Un servicio RESTful posee una arquitectura fuertemente orientada a recursos, y los principios REST se aplican de acuerdo a las siguientes reglas:

- El concepto fundamental en un servicio RESTful es el recurso. Un servicio envuelve una colección de recursos, donde cada recurso posee un identificador (o URI - Uniform Resource Identifier) único.
- Los recursos se manipulan a través de representaciones. Los datos que se transmiten entre el consumidor y el proveedor del servicio son documentos que representan a dichos recursos.
- Un mismo recurso puede disponer de múltiples representaciones (e.g. XML, JSON, JPEG, etc.).
- Los recursos son accedidos usando operaciones simples CRUD (Create - Read - Update - Delete). Para ello, es habitual utilizar el protocolo de comunicación HTTP, que permite la transferencia y manipulación de recursos remotos de manera nativa. Por tanto, el mapeo entre mensajes HTTP y las operaciones sobre los recursos es directo.
- El proveedor del servicio no mantiene ningún tipo de estado de las interacciones efectuadas por el cliente.

Los principios REST persiguen obtener arquitecturas software con una serie de propiedades muy deseables [Erl et al., 2012]:

- Rendimiento: en cuanto a la latencia y el consumo de ancho de banda de la red. En REST, por un lado, los mensajes suelen resultar más compactos que con otras tecnologías. Además, se habilitan caches para decrementar el número de intercambios. Por último, la complejidad de las interacciones es mínima, ya que los mensajes son autocontenidos.
- Escalabilidad: en cuanto a la carga de trabajo que es capaz de atender. En REST se habilita por medio de replicación de instancias y balanceo de la carga entre éstas. Como los contratos son homogéneos, los mensajes son autocontenidos y las instancias no poseen estado, las comunicaciones se pueden dirigir a cualquier instancia de manera indistinta.
- Simplicidad: en cuanto a la complejidad de la arquitectura software resultante. Es un objetivo primordial en REST. Por un lado, los servicios evolucionan de manera independiente. Por otro lado, la interfaz homogénea simplifica la interacción entre componentes. Por último, al no disponer de estado, los servicios RESTful son más sencillos de comprender.
- Modificabilidad: en cuanto a la sencillez de actualización de los servicios. La distinción en roles cliente-servidor de las arquitecturas REST junto al contrato uniforme, facilita la evolución independiente de cada pieza de la arquitectura.

- Visibilidad: en cuanto a la facilidad de comprensión (monitorización) de la comunicación entre servicios. En REST el contrato uniforme contribuye de manera definitiva a comprender la naturaleza de los intercambios, sin necesidad de conocer los detalles de los servicios involucrados.

- Portabilidad: en cuanto a la reutilización de los servicios en nuevos entornos. El contrato uniforme, generalmente basado en tecnologías estándar, y el código bajo demanda, que permite distribuir código interpretable a distintas plataformas, constituyen la base necesaria para facilitar la portabilidad de servicios.

- Fiabilidad: en cuanto a la probabilidad de fallo de los servicios. Las arquitecturas REST incrementan la fiabilidad del sistema. Por un lado, los servicios pueden ser fácilmente monitorizados y es posible detectar sus fallos. Al no disponer de estado, se replican con facilidad y es posible balancear la carga a instancias saludables.

2.5.2. El protocolo HTTP

A pesar de que los principios REST no fuerzan la adopción de HTTP como protocolo de transporte, por sus características genuinas suele ser la opción habitual. HTTP es un protocolo de comunicación sin estado que permite la transferencia y manipulación de recursos en Internet. Distintas versiones del protocolo se recogen en diferentes especificaciones[3]. A continuación se revisará el esquema básico del protocolo.

En HTTP los recursos se identifican por medio de URLs (Uniform Resource Locator) con la siguiente sintaxis genérica:

```
http://máquina:puerto/path?query#fragmento
```

Por ejemplo:

```
http://ejemplo.com/data/api/user/username?kind=alumno
```

HTTP soporta distintas operaciones sobre los recursos, que reciben el nombre de métodos HTTP:

- `GET`: recupera un recurso identificado por una URI desde el servidor.

- `POST`: envía un nuevo recurso al servidor.

[3] HTTP/1.1 en RFC 2068, sustituido por RFC 2616, y después por RFC 7230-7237. HTTP/2 en RFC 7540. HTTP/3 en borrador

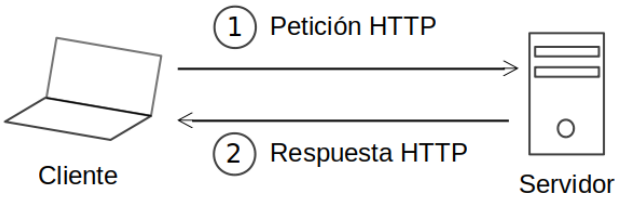


Figura 2.3: Esquema de funcionamiento HTTP

- `PUT`: reemplaza un recurso identificado por una URI por un nuevo contenido en el servidor.
- `DELETE`: elimina un recurso del servidor.
- `OPTIONS`: recupera información acerca de un recurso identificado por una URI.
- Otros como `HEAD`, `CONNECT`, `TRACE`.

El protocolo está basado en una arquitectura cliente-servidor, y sigue un esquema de intercambio de mensajes petición-respuesta sobre una conexión TCP/IP fiable. El cliente es un programa (e.g. un navegador) que establece una conexión TCP/IP contra el servidor y envía uno o más mensajes de petición HTTP. El servidor es un programa (e.g. un servidor web) que acepta conexiones de entrada TCP/IP, recibe mensajes de petición HTTP y envía mensajes de respuesta HTTP. Este esquema de funcionamiento queda reflejado en la figura 2.3.

Tanto los mensajes de petición como de respuesta se codifican en ASCII. A continuación se lista un ejemplo de petición HTTP GET que solicita un recurso:

```
GET /index.html HTTP/1.1

Host: www.ejemplo.org
User-Agent: Mozilla/4.0 (compatible; MSIE 4.0; Windows 95)
Accept: image/gif, image/jpeg, text/*, /*/*
```

Toda petición HTTP comienza con una línea de petición que contiene tres campos: el método HTTP que se solicita, la URI del recurso sobre el que se desea efectuar la operación y la versión del protocolo que se está utilizando para codificar el mensaje. A continuación, tras un `CRLF`[4] se especifica una colección de cabeceras que contienen información adicional sobre la petición o sobre el cliente.

[4] CRLF: Carriage Return - Line Feed, representa un salto de línea

En este caso, la cabecera `Host` identifica al servidor destinatario de la petición, la cabecera `User-Agent` describe el cliente que efectúa la petición y la cabecera `Accept` especifica el tipo de recurso que se espera obtener. El mensaje finaliza con un nuevo `CRLF`. Una petición HTTP también puede incluir contenido, por ejemplo cuando se desea crear o modificar un recurso. A continuación se lista un ejemplo de petición HTTP PUT cuyo propósito es actualizar un recurso:

```
PUT /tareas/1234 HTTP/1.1

Host: www.ejemplo.org
Content-Type: application/json
Content-Length: 50

{"id": "1234", "titulo": "Tarea 1", "fecha": "06/06/15" ...}
```

El mensaje posee la misma estructura que el mensaje de petición anterior, pero además incluye el contenido a actualizar. En esta ocasión es importante utilizar cabeceras que describan el contenido, como `Content-Type` y `Content-Length`, que determinan el tipo de contenido y su longitud respectivamente.

Existen multitud de cabeceras HTTP disponibles, que persiguen distintos objetivos. Algunas de las más usadas son:

- `Host`: es necesaria a partir de la versión 1.1 del protocolo, y determina el host al que se dirige la petición.
- `User-Agent`: proporciona información acerca del cliente que efectúa la petición. De este modo, el servidor podría potencialmente adaptar su respuesta a dicho cliente.
- `Accept`: especifica los tipos de recursos que se espera recibir, usando los tipos MIME.
- `Content-Type`: si la petición contiene datos, esta cabecera especifica el tipo MIME de dichos datos.
- `Content-Length`: si la petición contiene datos, la longitud de dichos datos.

Cuando el servidor recibe la petición HTTP, analiza su contenido y procede a efectuar la operación solicitada. Cuando finaliza la operación, envía un mensaje de respuesta al cliente. A continuación se lista un ejemplo de respuesta HTTP:

```
HTTP/1.1 200 Ok
```

```
Date: Fri, 06 Jul 2001 8:26:55 GMT
Server: Apache/1.3.20 (Unix)
Content-Type: text/html; charset=iso-8859-1
Content-Length: 67

<html>
<head><title></title></head>
<body>Hello World!!</body>
</html>
```

La respuesta comienza por la línea de estado, que posee tres campos: la versión del protocolo HTTP utilizada, el código de estado y una descripción textual de dicho código de estado. El código de estado es un número de tres cifras que determina el resultado de la operación solicitada por el cliente. Existen múltiples códigos de estado:

- 1xx: el mensaje de respuesta es meramente informativo.
- 2xx: la operación solicitada ha tenido éxito. Dependiendo del tipo de operación, es posible identificar distintos códigos de éxito. Por ejemplo, 200/"Ok" determina éxito sin información adicional, 201/"Created" indica que el recurso ha sido creado con éxito (útil para operaciones POST) ó 204/"No content" indica que la operación ha tenido éxito pero no se devuelve ningún contenido (útil para operaciones DELETE).
- 3xx: indica redirección, posiblemente porque el recurso ha sido movido a otra ubicación.
- 4xx: indica que ha sucedido un error causado por el cliente. Por ejemplo, 400/"Bad Request" determina un error en la petición, 401/"Unauthorized" indica que el cliente no posee permisos para acceder al recurso ó 404/"Not Found" indica que el recurso no existe.
- 5xx: indica que ha sucedido un error causado por el servidor. Por ejemplo, 500/"Internal Server Error" indica que algo falló en el procesamiento de la petición HTTP, 501/"Not Implemented" determina que la operación solicitada no ha sido implementada ó 503/"Service Unavailable" indica que el servicio solicitado no está disponible.

Tras la línea de estado, se inserta un `CRLF` y a continuación una lista de cabeceras que contienen información adicional sobre la respuesta o sobre el servidor. En este caso, la cabecera Date determina el instante de tiempo en el que se generó la respuesta, la cabecera `Server` contiene información sobre el servidor y las

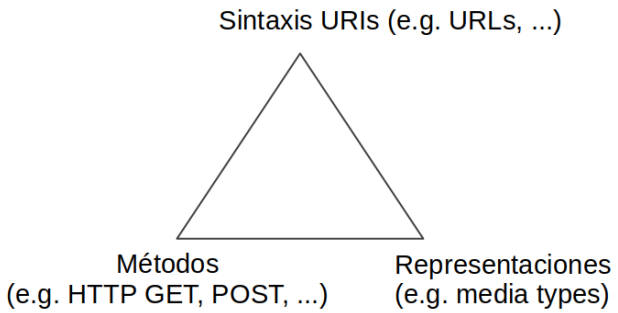


Figura 2.4: Elementos que determinan el contrato de servicio

cabeceras `Content-Type` y `Content-Length` describen el tipo de contenido que incluye la respuesta y su longitud, respectivamente. A continuación, se incluye un `CRLF` y finalmente el contenido de la respuesta. Como ya se ha comentado, no todas las respuestas poseen contenido, de manera que esta última parte es opcional.

Del mismo modo que sucedía con las peticiones HTTP, el protocolo soporta un gran número de cabeceras. Algunas de las más usadas son:

- `Date`: el instante de tiempo en el que se generó el mensaje.
- `Server`: información acerca del servidor.
- `Content-Type`: el tipo MIME del contenido incluido en la respuesta.
- `Content-Length`: la longitud del contenido.
- `Cache-Control`: permite controlar el comportamiento de las caches.
- `Expires`: determina el instante de tiempo en el que la respuesta quedará obsoleta. Útil en el caso de caches.
- `Last-Modified`: la fecha de última modificación del recurso. Útil para el uso con caches.

2.5.3. El contrato de servicio

El contrato de un servicio RESTful (o su API) queda determinado por tres elementos [Erl, 2016] (ver figura 2.4): la sintaxis utilizada para identificar a los distintos recursos, los métodos necesarios para manipular los recursos y las representaciones de los recursos que se transfieren.

Para identificar los recursos se utilizan generalmente URIs, para definir los métodos se aprovechan los métodos del protocolo HTTP, y para definir las representaciones se utilizan los *media types*. Además, es recomendable definir la API del servicio de manera formal. Existen diversas alternativas para ello: OpenAPI, WADL, etc. En los siguientes apartados se exploran todas estas cuestiones.

2.5.3.1. Identificadores de los recursos

Un servicio RESTful encapsula una colección de recursos. Los recursos se suelen identificar por medio de URIs[5], cuya sintaxis verifica el siguiente esquema:

```
{scheme}://{authority}{path}?{query}
```

Las URIs de los recursos deberían de poseer una estructura lógica y deberían resultar lo suficientemente descriptivas y predecibles. Generalmente, todo servicio RESTful publica todos sus recursos bajo una URI base, por ejemplo `http://www.example.org`. A partir de esta URI base se construyen las URIs de todos los recursos que gestiona. A continuación se listan algunas recomendaciones para el diseño de URIs:

- Una colección de recursos debería quedar representada por un nombre (no una acción), generalmente en plural. Por ejemplo, si un servicio RESTful se publica bajo la URL `http://www.example.org` y gestiona una colección de usuarios, es recomendable que la URI `http://www.example.org/users` represente la colección de todos los usuarios y la URI de cada usuario se construya utilizando un patrón similar a `http://www.example.org/users/{userId}`.
- Las URIs deberían construirse con caracteres en minúsculas. Se fomenta el uso del carácter '-' y se debería evitar utilizar el carácter '_'.
- Las URIs que representan subcolecciones (para modelar jerarquía, composición o relaciones) deberían de componerse utilizando '/'. Por ejemplo, la URI `http://www.example.org/users/{userId}/accounts` podría representar todas las cuentas de un usuario y la URI de cada cuenta podría construirse utilizando un patrón similar a `http://www.example.org/users/{userId}/accounts/{accountId}`.
- Para filtrar u ordenar una colección de recursos se suelen utilizar parámetros de query en la URI. Por ejemplo, para recuperar los usuarios con nombre Pepe se podría utilizar una URI similar a `http://www.example.org/users?name=Pepe`.

[5] RFC 3986

- Es habitual que se publiquen distintas versiones de una misma API. Es recomendable que la versión se incluya en la propia URI. Por ejemplo `http://www.example.org/v1/users`.

2.5.3.2. Métodos

Además de las URIs que permiten identificar a los recursos, el contrato de servicio debe considerar las operaciones que se pueden efectuar sobre estos, son los métodos. Todos los servicios publicados en un mismo inventario deben ofrecer una interfaz uniforme y por tanto deben exponer las mismas operaciones. Por este motivo, las operaciones expuestas por un servicio RESTful suelen ser muy genéricas, y representan las operaciones básicas CRUD (Create-Read-Update-Delete).

En la actualidad, la aproximación habitual es que los servicios RESTful se hagan accesibles a través del protocolo HTTP. Además, el protocolo HTTP es parte esencial del contrato, ya que no sólo proporciona el protocolo de transporte y el formato de los mensajes intercambiados, sino también las operaciones disponibles.

De este modo, en un servicio RESTful, un mensaje de petición HTTP especifica una operación sobre un recurso. El recurso queda identificado por la URI y la operación solicitada queda representada por el método HTTP. Es común utilizar los distintos métodos HTTP con el siguiente significado:

- `GET`: recupera un recurso o una colección de recursos.
- `POST`: crea un nuevo recurso en una colección de recursos.
- `PUT`: actualiza un recurso dentro de una colección de recursos.
- `DELETE`: elimina un recurso dentro de una colección de recursos.

2.5.3.3. Representaciones de los recursos

Por último, en el contrato de servicio es necesario describir las representaciones de los recursos (o tipos de datos) con los que puede trabajar cada operación. Cada método puede gestionar distintas representaciones (distintos tipos de datos) de un mismo recurso. Por ejemplo, un método HTTP GET puede transferir un recurso en texto plano, en formato XML, JSON, o incluso en formato de imagen JPG.

Para especificar el tipo de representación se suele utilizar la sintaxis proporcionada por los *media types*[6]. Un *media type* consiste de un tipo y un subtipo,

[6] Anteriormente denominados MIME types: RFC 2045, RFC 6838

estructurado en forma de árbol, y puede soportar distintos parámetros. El esquema global es:

```
type / subtype [; parameter]
```

Algunos de los tipos principales son `application`, `audio`, `image`, `text`, `video`, ... Algunos ejemplos de media types son: `application/json`, `application/xml`, `image/jpeg`, `text/plain`, `text/html`, `text/xml`, etc.

En una petición HTTP GET, el consumidor del servicio RESTful suele especificar el tipo de representación requerido estableciendo la cabecera HTTP `Accept`, por ejemplo:

```
Accept: text/html; charset=UTF-8
```

Cualquier otro mensaje HTTP que incluya contenido, especifica su naturaleza utilizando la cabecera HTTP `Content-Type`, por ejemplo:

```
Content-Type: application/json
```

En la actualidad el tipo de representación más utilizado en los servicios RESTful es JSON. JSON es el acrónimo de JavaScript Object Notation[7]. Se trata de un formato de datos basado en texto, muy eficiente, que recuerda a la sintaxis que se utiliza para definir objetos literales en JavaScript. JSON soporta los tipos de datos `number`, `boolean`, `string`, `null`, `object` y `array`. Se trata de un conjunto de datos básicos que soporta cualquier lenguaje de programación moderno y además permite describir cualquier estructura de datos.

A continuación se muestra un ejemplo de un documento codificado en JSON:

```
{
 "id": 1, "title": "family", "owner": true, "date": null,
 "messages": [
   {"id": 1, "content": "hello"}, {"id": 2, "content":"bye"}
  ]
}
```

Los objetos son diccionarios que se representan con la sintaxis `{entrada,entrada,...}`. Cada entrada de un diccionario posee la sintaxis `clave:valor`, donde la `clave` es siempre un `string` y el `valor` puede ser cualquier tipo de datos soportado por JSON. Los arrays se representan con la sintaxis `[valor,valor,...]`.

[7] https://json.org

2.5.4. Implementación de servicios

Para implementar un servicio RESTful se puede utilizar cualquier tecnología capaz de trabajar con el protocolo HTTP. En esta sección se explorará el framework Flask, que permite construir servicios RESTful de una manera muy eficaz utilizando el lenguaje de programación Python.

2.5.4.1. Flask

Flask[8] es un framework minimalista que proporciona las funcionalidades esenciales para construir aplicaciones web en la parte del servidor. Debido precisamente a su simplicidad, representa una alternativa ideal para construir servicios RESTful. Además, posee múltiples extensiones que permiten ampliar fácilmente sus capacidades básicas.

La manera más conveniente de trabajar con Flask es utilizando los entornos virtuales de Python 3.X. Para ello, primero es necesario crear el entorno virtual:

```
> python3 -m venv venv
```

A continuación se activará dicho entorno virtual y se instalará Flask:

```
> source venv/bin/activate      (en Windows venv/Scripts/activate)
(venv) > pip install flask
```

Para comprobar que Flask está correctamente instalado, es recomendable ejecutar el intérprete interactivo de Python e importar Flask:

```
(venv) > python3
>>> import flask
```

Por último, se creará la primera aplicación Flask en el fichero `hello.py`.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
  return '<h1>Hello World!</h1>'
```

[8] https://flask.palletsprojects.com/en/2.0.x/

El código anterior crea una aplicación Flask y registra una ruta, que responde a peticiones HTTP sobre la URL raíz, devolviendo un fragmento HTML. Para ejecutar la aplicación es necesario crear un servidor web que se comunique con ella. Flask incorpora un servidor web de desarrollo. Antes de ejecutarlo es necesario indicarle por medio de la variable de entorno FLASK_APP el script en el que se crea la aplicación Flask.

```
(venv) > export FLASK_APP=hello.py (en Windows: set FLASK_APP=hello.py)
(venv) > flask run
* Serving Flask app 'hello.py' (lazy loading)
* Environment: production
   WARNING: This is a development server. Do not use it in a
   production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Ahora es posible comprobar que la aplicación funciona accediendo a la URL `http://127.0.0.1:5000/` desde cualquier navegador web.

Cuando se desarrolla en Flask resulta muy conveniente habilitar el modo Debug. Este modo activa los módulos *reloader* y *debugger*. El primero detecta cambios en el código fuente y recarga la aplicación de manera automática. El segundo muestra información muy útil por el navegador web en caso de detectar una excepción no controlada. El modo Debug se activa por medio de la variable de entorno FLASK_DEBUG=1.

```
> export FLASK_DEBUG=1 (en Windows: set FLASK_DEBUG=1)
```

El comando flask dispone de múltiples opciones, que es conveniente conocer:

```
> flask --help
```

2.5.4.2. Rutas

Los clientes de una aplicación/servicio web efectúan peticiones HTTP que son recibidas en un servidor web. El servidor web las redirige a la aplicación Flask. La aplicación Flask mapea URLs a funciones que generan una respuesta. La asociación entre una URL y la función que la maneja recibe el nombre de ruta.

La manera más sencilla de registrar una ruta es utilizando el decorador `app.route(url, opts)` sobre la función que maneja dicha ruta. En el decorador, por un lado se indica la URL que se pretende manejar, y por otro lado distintas opciones de configuración.

```
@app.route('/')
def index():
  return '<h1>Hello World!</h1>'
```

Es posible definir rutas dinámicas (o paramétricas). Una ruta dinámica posee una URL con parámetros. Los parámetros se definen al registrar la ruta con la sintaxis `<param>` y son huecos que se rellenan cuando el cliente efectúa peticiones HTTP. De este modo, una ruta dinámica representa a una colección de rutas estáticas. Para recibir los parámetros, la función manejadora define argumentos adicionales.

```
@app.route('/user/<name>')
def user(name):
  return '<h1>Hello, {}!</h1>'.format(name)
```

Por defecto, los argumentos que recibe la función manejadora en una ruta dinámica son de tipo String. Al registrar la ruta, es posible especificar conversores con la sintaxis `<converter:param>`. Los conversores disponibles son `string`, `int`, `float`, `path`, `uuid`.

```
@app.route('/user/<int:userId>')
def user(userId):
  return f'User {userId}'
```

Por defecto, al registrar una ruta, todas las peticiones HTTP de tipo GET, HEAD y OPTIONS sobre la URL son redirigidas a la función manejadora. Es posible especificar los métodos HTTP sobre los que la ruta se activará utilizando la opción `methods`. De este modo resulta sencillo implementar APIs RESTful.

```
@app.route('/', methods=['GET','POST','PUT','DELETE'])
def index():
  return '<h1>Hello World!</h1>'
```

Una vez registradas las rutas, éstas se mantienen en un diccionario disponible en la propiedad `app.url_map`.

```
>>> from hello import app
>>> app.url_map
Map([<Rule '/' (HEAD, OPTIONS, GET) -> index>,
<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
<Rule '/user/<name>' (HEAD, OPTIONS, GET) -> user>])
```

Al crear una aplicación Flask, se registra de manera automática una ruta capaz de servir contenido estático a partir del subdirectorio `/static`.

Propiedad/método	Descripción
method	El método HTTP
url	La URL completa solicitada por el cliente
host, path, query_string	Distintas secciones de la URL ya analizada
remote_addr	La dirección IP del cliente
headers	Diccionario con todas las cabeceras HTTP incluídas en la petición
args	Diccionario con todos los argumentos pasados en la query_string de la petición
get_data()	Devuelve los datos incluidos en el cuerpo de la petición HTTP
get_json()	Devuelve un diccionario que contiene los datos insertados en el cuerpo de la petición HTTP en formato JSON

Cuadro 2.1: Objeto Request

2.5.4.3. Petición

Cuando Flask recibe una petición HTTP crea un contexto de ejecución únicamente disponible para la función que gestiona dicha petición. En este contexto se registran algunos objetos como el objeto global flask.request, que contiene toda la información sobre la petición HTTP actual. Para acceder a dicho objeto primero es necesario importarlo. Una vez importado es posible acceder a sus propiedades desde cualquier función.

```
from flask import request
@app.route('/')
def index():
  user_agent = request.headers.get('User-Agent')
  return '<p>Your browser is {}</p>'.format(user_agent)
```

El objeto `Request` publica propiedades y métodos muy útiles. En la tabla 2.1 se listan algunos de ellos:

Con toda esta información disponible, la función que maneja la petición HTTP puede conocer con exactitud qué solicita el cliente, y responder adecuadamente.

2.5.4.4. Respuesta

La función que maneja la petición HTTP es responsable de devolver una respuesta. Flask convierte automáticamente el valor devuelto por dicha función

en respuesta HTTP.

Si el valor devuelto por la función es un String, entonces se devuelve una respuesta con código de estado 200, y con cuerpo el String devuelto, cuyo tipo MIME se establece a `text/html`.

```
@app.route('/')
def index():
  return '<h1>Hello World!</h1>'
```

Si la función manejadora devuelve un diccionario, entonces se devuelve una respuesta con código de estado 200, con cuerpo el resultado de invocar `jsonify()` sobre el diccionario devuelto, y como tipo MIME `application/json`. Este comportamiento facilita mucho la implementación de APIs RESTful, que generalmente devuelven objetos al cliente.

```
@app.route('/user/<name>')
def user(name):
  user = findUserByName(name)   // find user externally
  return user
```

La función manejadora puede adquirir un mayor control sobre la respuesta devolviendo una tupla. En este caso, se espera que la tupla se construya del siguiente modo: `(response, status)` o bien `(response, headers)` o bien `(response, status, headers)`. Donde `response` es el contenido devuelto, `status` el código de estado y `headers` un diccionario con las cabeceras que se deben incluir en la respuesta.

```
@app.route('/')
def index():
  return '<h1>Bad Request</h1>', 400
```

Por último, existe una manera de construir la respuesta manualmente. Para ello, es necesario importar la función `flask.make_response()`, que devuelve un objeto que permite construir dicha respuesta. Esta última opción permite un control total sobre la respuesta devuelta al cliente, y habilita funcionalidades avanzadas como pro ejemplo incluir cookies.

```
from flask import make_response
@app.route('/')
def index():
  response = make_response('<h1>This document carries a cookie!</h1>')
  response.set_cookie('answer', '42')
  return response
```

2.5.4.5. Errores

Cuando sucede un error en Flask, se devuelve un código de estado apropiado. Los códigos 4XX se utilizan para indicar errores en los datos incluidos en la petición y los códigos 5XX se utilizan para indicar errores originados en la aplicación Flask.

Es posible personalizar las respuestas que se envían al cliente cuando surge un error. Para ello es necesario registrar manejadores de errores utilizando el decorador `errorhandler()`.

```
@app.errorhandler(404)
def page_not_found(e):
    return '<h1>Not found!</h1>', 404
```

Al procesar una petición, si se detecta algún problema, es posible abortar la petición en curso utilizando la función `abort()`. Esta función acepta detalles sobre el error a generar y lanza una excepción, de manera que el procesamiento de la petición actual finaliza de manera abrupta.

```
@app.route("/")
def index():
  if error:
    abort(404, description="Resource not found")
```

2.5.4.6. Hooks

En ocasiones resulta útil ejecutar cierto código antes y/o después de procesar cualquier petición HTTP. Algunos ejemplos de ello serían: inicializar una conexión a la base de datos, efectuar acciones de autenticación o control de accesos a los recursos, compresión/descompresión de datos, etc. Para ello, Flask permite registrar hooks. Un hook es una función marcada con un decorador especial, que es invocada automáticamente por Flask en ciertos momentos en el ciclo de procesado de una petición HTTP. Flask habilita los hooks listados en la tabla 2.2.

A continuación se muestra un ejemplo en el que se muestra por pantalla detalles de cada petición HTTP recibida por Flask:

```
from flask import Flask, request
app = Flask(__name__)

@app.before_request
def before():
```

Decorador	Descripción
before_request	La función se ejecuta antes de procesar cada petición
before_first_request	La función se ejecuta únicamente una vez, antes de procesar la primera petición HTTP recibida por Flask
after_request	La función se ejecuta tras procesar cada petición, si no se lanza una excepción
rteardown_request	La función se ejecuta tras procesar cada petición, incluso si se lanza una excepción

Cuadro 2.2: Hooks de Flask

```
    msg = f'method={request.method},url={request.url}'
    print(msg)
```

Es habitual que los hooks registren cierta información (e.g. información sobre el usuario actual, la conexión a la base de datos, los datos descifrados, etc.) para que sea accesible a las funciones que gestionarán posteriormente la petición. Para ello, es posible importar el diccionario global `Flask.g`, que está activo en el contexto de la petición actual y puede ser accedido por cualquier manejador que se ejecute en dicho contexto. A continuación se lista un ejemplo:

```
from flask import Flask, g
app = Flask(__name__)

@app.before_request
def before():
    print("New request!!")
    g.msg = "Hello world!!"

@app.route('/')
def index():
    return f'<h1>{g.msg}</h1>'
```

2.5.4.7. Extensiones

Las extensiones son paquetes que añaden funcionalidad extra a la aplicación Flask. Existen múltiples extensiones que permiten incorporar infinidad de utilidades, por ejemplo para enviar mails, acceder a bases de datos, etc.

Es posible obtener un listado de todas estas extensiones haciendo una consulta de los paquetes PyPI con el tag `Framework::Flask:`[9]

[9] https://pypi.org/search/?c=Framework+%3A%3A+Flask

Existe una extensión denominada Flask RESTful que facilita la creación de servicios RESTful[10]. En esta guía no se utilizará esta extensión.

2.5.4.8. Implementación de una API RESTful

Implementar una API RESTful en Flask consiste en registrar un conjunto de rutas capaces de gestionar una colección de recursos. Al poseer un contrato uniforme, cualquier servicio RESTful se implementa de una manera similar. A continuación se lista un esquema del código necesario para gestionar una colección de tareas:

```
from flask import Flask
app = Flask(__name__)

@app.route('/myapp/tasks')
def listTasks(): ...
@app.route('/myapp/tasks/<taskId>')
def getTaskById(taskId): ...
@app.route('/myapp/tasks', methods=['POST'])
def addTask(): ...
@app.route('/myapp/tasks/<taskId>', methods=['PUT'])
def updateTask(taskId): ...
@app.route('/myapp/tasks/<taskId>', methods=['DELETE'])
def deleteTask(taskId): ...
```

La primera ruta, que atiende peticiones HTTP GET sobre la URI `'/myapp/tasks'`, permite obtener todas las tareas, posiblemente filtradas por ciertas condiciones. La segunda ruta contiene un parámetro, y permite recuperar una tarea a partir de su identificador. La tercera ruta atiende peticiones HTTP POST sobre la URI `'/myapp/tasks'` y se utiliza para crear nuevas tareas. La cuarta ruta permite actualizar una tarea y la última ruta permite eliminarla.

Este patrón de rutas es común en cualquier implementación de una API RESTful y puede sistematizarse. El verdadero trabajo reside en implementar el manejador registrado en cada ruta. En general, este código siempre incluirá dos pasos básicos:

1. Procesar la información de entrada

2. Generar la información de salida

[10] https://flask-restful.readthedocs.io/en/latest/

2.5.4.9. Procesar la información de entrada

Si la petición a procesar es HTTP GET, entonces se desea recuperar uno o varios recursos. En este caso el mensaje a procesar no posee contenido. Los únicos datos que puede incorporar son la URI solicitada y opcionalmente una query que permita filtrar los resultados obtenidos, y que se añade a la URI utilizando el patrón `http://uri?q1=yy&q2=zz`. En Flask la URI solicitada se puede recuperar utilizando la propiedad `flask.request.url`, o bien `flask.request.path` si únicamente se desea obtener el path del recurso. La query estaría disponible en `flask.request.query_string`, o bien `flask.request.args` si se desea acceder al diccionario de parámetros ya analizado. Por ejemplo, para recuperar las credenciales en una operación de autenticación se podría usar un código similar a éste:

```
from flask import Flask, request
app = Flask(__name__)

@app.route('/myapp/sessions', methods=['GET'])
def login():
  user = request.args['user']
  password = request.args['password']
```

Si la petición no es HTTP GET, entonces los datos están en el cuerpo del mensaje. Para recuperarlos es posible utilizar las funciones `flask.request.get_data()` o bien `flask.request.get_json()`, si los datos están codificados en formato JSON.

Por ejemplo, para procesar una petición HTTP POST como la siguiente:

```
POST /myapp/tasks HTTP/1.1

Host: www.ejemplo.org
Content-Type: application/json
Content-Length: 50

{"id": "1234", "title": "Tarea 1", "date": "06/06/15" ...}
```

Se podrıía crear una aplicación Flask con un código similar a éste:

```
from flask import Flask, request
app = Flask(__name__)

@app.route('/myapp/tasks', methods=['POST'])
```

```
def addTask():
  task = request.get_json()
  print(f'id:{task.id}')
  print(f'titulo:{task.title}')
  return 'OK', 200
```

En una API RESTful es muy común registrar rutas dinámicas (o paramétricas). Por ejemplo, si se desea registrar una ruta para recuperar recursos por identificador, la aproximación habitual es crear una ruta dinámica (paramétrica) como la siguiente:

```
@app.route('/myapp/tasks/<taskId>')
def getTaskById(taskId):
  print(f'id:{taskId}')
```

Un parámetro es un hueco que debe ser suministrado por el cliente que efectúa la petición HTTP, de manera que constituye un mecanismo adicional de paso de información de entrada al servicio RESTful. Los parámetros son mapeados a los argumentos de la función manejadora.

2.5.4.10. Generar la información de salida

Una vez se ha obtenido toda la información necesaria para procesar la petición HTTP, se efectúa la operación solicitada. Generalmente ello incluye la consulta o modificación de un almacén de datos. Una vez finalizada (o mientras se procesa) es necesario generar una respuesta HTTP, que en ocasiones contiene datos.

La estrategia más sencilla para generar la respuesta HTTP es devolver el contenido solicitado. Si se trata de un String, entonces Flask asume que es de tipo MIME `text/html` y lo inserta en el cuerpo de la respuesta HTTP, añadiendo además un código de estado 200. Si se devuelve un diccionario entonces Flask lo formatea a JSON y asume que el tipo MIME es `application/JSON`.

```
@app.route('/myapp/tasks/<taskId>')
def getTaskById(taskId):
  task = searchById(taskId)
  return task
```

Si se desea devolver una respuesta sin contenido lo habitual es devolver una tupla donde el primer elemento es un String vacío y el segundo un código de estado que determina que la respuesta no posee contenido, por ejemplo 204.

```
@app.route('/myapp/tasks/<taskId>', methods=['DELETE'])
def removeTask(taskId):
  removeTaskInDB(taskId)
  return '', 204
```

Por último, la manera más habitual de notificar errores consiste en utilizar la función `abort()`.

```
@app.route('/myapp/tasks/<taskId>')
def findTaskById(taskId):
  abort(404)
```

2.5.5. Consumo de servicios

Para efectuar operaciones sobre una API RESTful únicamente es necesario disponer de un cliente HTTP. El navegador es una alternativa al alcance de cualquier usuario. Cuando se introduce una URL en la barra de direcciones, el navegador automáticamente efectúa una petición HTTP GET solicitando la URI en cuestión. Es por tanto una primera opción si únicamente se desea efectuar peticiones HTTP GET.

Si también se desea efectuar otro tipo de peticiones (POST, PUT, DELETE, etc.), entonces es necesario trabajar con herramientas más especializadas. Existen múltiples opciones disponibles, como Postman[11], Insomnia[12] o Advanced REST Client[13], por nombrar algunas herramientas. Todas ellas permiten construir peticiones HTTP y explorar las respuestas HTTP obtenidas.

Para acceder a un servicio RESTful utilizando Python existen diversas opciones Una de las más comunes es utilizar la librería *requests*[14]. Se trata de una librería que facilita la construcción de peticiones

El primer paso consiste en instalar la librería:

```
> pip install requests
```

La librería permite efectuar cualquier tipo de petición HTTP. Para ello facilita diversos métodos de conveniencia, cuyos parámetros son autoexplicativos:

- `requests.get(url,params=None)`
- `requests.post(url,data=None,json=None)`

[11] https://www.postman.com/
[12] https://support.insomnia.rest/
[13] https://install.advancedrestclient.com/install
[14] https://docs.python-requests.org/en/master/

- `request.put(url,data=None,json=None)`
- `request.delete(url)`
- etc.

Cuando se efectúa una petición HTTP, se recibe un objeto de la clase `requests.Response`, que contiene toda la información sobre la respuesta, por ejemplo:

- `.status_code`: el código de estado de la respuesta (e.g. 100, 404, etc.)
- `.headers`: diccionario que contiene todas las cabeceras incluidas en la respuesta
- `.content`: contenido de la respuesta en bytes
- `.text`: contenido de la respuesta en texto
- `.json()`: devuelve el contenido en formato JSON
- etc.

En el siguiente fragmento de código se muestran algunos ejemplos de uso:

```
import requests
response = requests.get("http://localhost:5000/")
print(response.text)
response = requests.post("http://localhost:5000/users", json={"name":"Pepe"})
print(response.json())
response = requests.delete("http://localhost:5000/users/pepe")
print(response.status_code)
```

2.6. Virtualización

La *virtualización* es el proceso de crear una versión virtual (no real), abstracta o lógica de algo, incluyendo el hardware de un ordenador, un dispositivo de almacenamiento, una red de comunicaciones, un sistema operativo, etc.

Para ello, en un entorno virtualizado se aplica el principio orientado a capas, y se identifican tres capas fundamentales: el *guest*, el *host* y la *capa de virtualización*. El *guest* representa el recurso expuesto, que interactúa con la *capa de virtualización* en lugar de con el *host*, como sería habitual. El *host* representa el recurso original, que sufre el proceso de virtualización. La *capa de virtualización* es responsable de simular el entorno en el que se ejecutará el guest, y

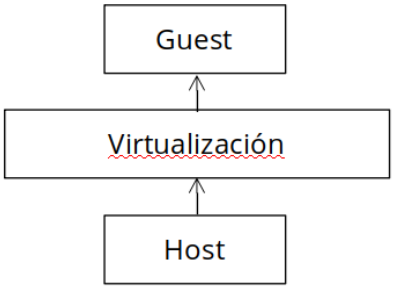
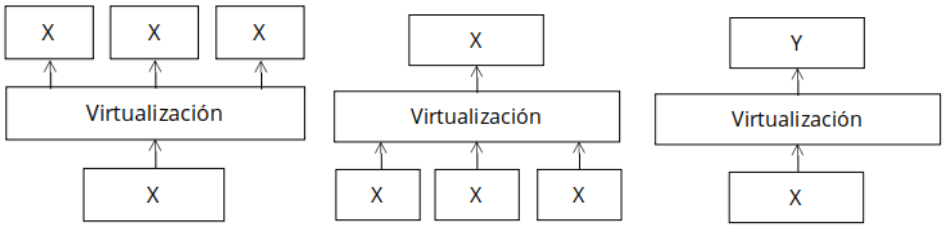


Figura 2.5: Virtualización



(a) Multiplexado (b) Agregación (c) Emulación

Figura 2.6: Técnicas de virtualización

suele ser un programa software (en ocasiones complementado con herramientas hardware). La figura 2.5 ilustra esta idea básica.

Con este procedimiento se produce un desacople importante entre el recurso expuesto y el recurso físico y se pueden habilitar distintas estrategias como *multiplexado*, *agregación* o *emulación* (ver figura 2.6).

El *multiplexado* (o *compartición*) permite crear múltiples recursos virtuales a partir de un único recurso físico. Se produce por tanto un incremento en la tasa de utilización del recurso físico, y puede aplicarse en el espacio o en el tiempo. El multiplexado espacial permite particionar el recurso físico en múltiples recursos virtuales. Por ejemplo, un sistema operativo lo aplica cuando las páginas de memoria física son mapeadas a las páginas de memoria virtual que se asignan a los distintos procesos. El multiplexado temporal permite compartir el mismo recurso físico en el tiempo entre distintas entidades virtuales. Esto sucede por ejemplo cuando el planificador del sistema operativo asigna períodos de ejecución de la CPU a los distintos procesos.

La *agregación* produce el efecto inverso, reúne múltiples recursos físicos y los expone como una única abstracción. Un ejemplo conocido sería una controladora RAID, que permite agregar múltiples discos que son vistos por el sistema operativo como un único volumen.

Por último, la *emulación* permite exponer un recurso virtual que no se co-

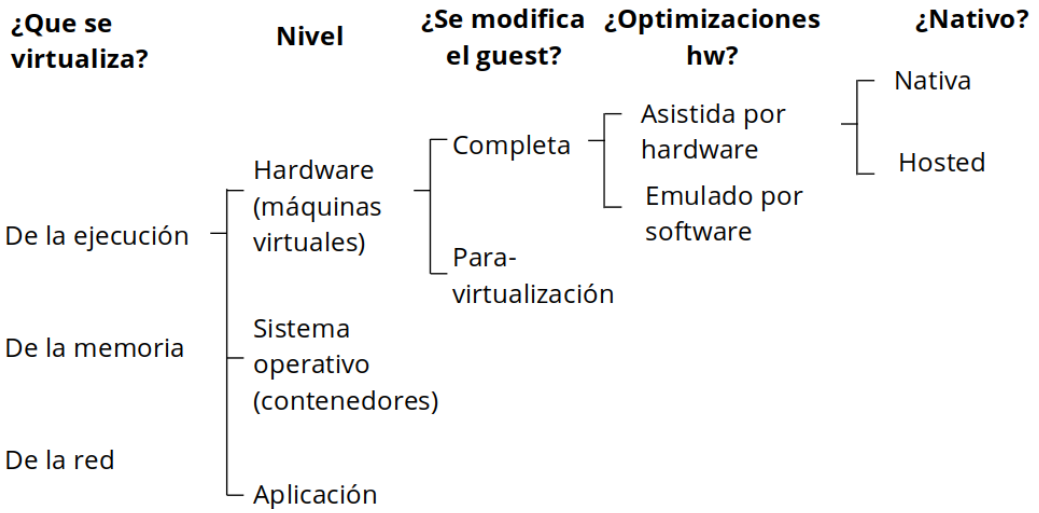


Figura 2.7: Taxonomía de las técnicas de virtualización

rresponde con el recurso físico virtualizado.

Las tecnologías de virtualización actuales suelen combinar todas estas estrategias de virtualización.

Las distintas técnicas de virtualización pueden clasificarse entorno a diversos ejes. A modo de esquema se presenta en la figura 2.7 una taxonomía de técnicas de virtualización que no pretende resultar completa.

Atendiendo a la naturaleza de los recursos emulados, es posible inducir la siguiente clasificación de técnicas de virtualización:

- *Virtualización de la ejecución*: el objetivo consiste en emular un entorno de ejecución. Incluye todas las formas de máquinas virtuales, pudiendo emular un entorno de ejecución de bajo nivel (i.e. recursos hardware), como VirtualBox, o un entorno de ejecución de alto nivel como la JVM.

- *Virtualización del almacenamiento*: permite desacoplar la organización física de los dispositivos de almacenamiento de su organización lógica. En ocasiones también se identifica con el término *Software Defined Storage* (*SDS*). Un ejemplo de esta aproximación sería los *SANs* (*Storage Area Network*).

- *Virtualización de la red*: a partir de una o varias redes físicas se configura una red lógica. Es habitual distinguir entre virtualización externa, que involucra múltiples redes (e.g. VLANs) y virtualización interna, cuyo objetivo es crear una red virtual en el interior de un único servidor para interconectar las máquinas virtuales o contenedores que se ejecutan en su interior.

2.7. Beneficios de la virtualización

A pesar de que implantar un sistema virtualizado con ciertas garantías puede resultar muy costoso, la virtualización conlleva importantes ventajas que es necesario conocer:

- *Mejora en la utilización de recursos*

  Gracias al multiplexado, múltiples servidores virtuales pueden ser ejecutados en un único servidor físico. Esta propiedad recibe la denominación de *consolidación de servidores*. Esto permite una mejor utilización de recursos, e implica una reducción en la adquisición de nuevos dispositivos hardware. Además, es posible efectuar un particionado inteligente de los recursos requeridos por cada máquina virtual, en función de la calidad de servicio exigida por el consumidor.

- *Mejora en el impacto medioambiental*

  Al disponer de menos dispositivos hardware y aprovechar mejor la utilización de los recursos disponibles, se reduce la inversión en energía y refrigeración, y por tanto la huella de carbono.

- *Mejora de la eficiencia y la productividad*

  Al existir menos servidores y menor infraestructura física, se invierte menos tiempo en su configuración y mantenimiento. Además, las operaciones con recursos virtuales son muy rápidas y pueden automatizarse.

- *Reducción de costes*

  Todos los beneficios anteriores, adquisición de un menor número de recursos, reducción de la inversión en energía e incremento de la productividad, contribuyen a una importante reducción de costes.

- *Mejora de la seguridad*

  La capa de virtualización permite controlar cualquier aspecto de la ejecución del guest, proporcionando un entorno seguro para éste, y mostrando u ocultando selectivamente los recursos disponibles en el host. Además, la actividad del guest está permanentemente monitorizada y puede ser filtrada o bloqueada, en caso de efectuar operaciones dañinas.

- *Mejora de la escalabilidad, fiabilidad y resiliencia*

  Los recursos virtuales se gestionan con mucha agilidad. Pueden ser creados, replicados, movidos, reemplazados en pocos minutos. Esta condición permite reaccionar ante posibles fallos de una manera muy rápida, reduciendo drásticamente el tiempo de recuperación.

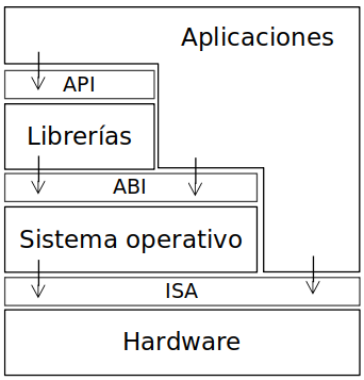


Figura 2.8: Capas de un sistema computacional tradicional

- *Mejora de la portabilidad*

  Resulta habitual empaquetar el guest en algún formato fácilmente distribuible, generalmente denominado imagen. Esta imagen puede ser distribuida y ejecutada en cualquier entorno que disponga de la misma capa de virtualización. Si el formato de empaquetado es estándar, se eliminan las dependencias con proveedores específicos (i.e. *vendor lock-in*).

2.8. Virtualización de la ejecución

La virtualización de la ejecución permite emular un entorno de ejecución completo, y es la forma de virtualización más común. En este contexto, un sistema computacional se puede descomponer en las siguientes capas fundamentales (ver figura 2.8): hardware, sistema operativo, librerías y aplicaciones. La interfaz entre estas capas está muy bien definida. El *ISA* (*Instruction Set Architecture*) representa la interfaz con el hardware y determina el juego de instrucciones del procesador, los registros, la memoria y la gestión de interrupciones. El *ABI* (*Application Binary Interface*) representa la interfaz con el sistema operativo y define las llamadas al sistema, el formato de los ejecutables, la alineación de memoria, las convenciones utilizadas para invocación de funciones, etc. Finalmente, las *APIs* (*Application Programming Interface*) son utilizadas por las aplicaciones para acceder a librerías o al sistema operativo.

Esta separación en capas tan clara favorece la virtualización de cualquiera de ellas, pudiendo ofrecer a su inmediata superior una versión simulada por software. Siguiendo este esquema, es posible identificar distintos niveles de virtualización de la ejecución:

- *Virtualización a nivel de hardware*: se virtualizan los dispositivos hardwa-

re de un computador. Es habitual virtualizar todos los componentes de un computador, dando lugar a máquinas o servidores virtuales. Algunos ejemplos de esta aproximación serían Virtual Box o Xen.

- *Virtualización a nivel de sistema operativo*: se crean entornos de ejecución independientes denominados contenedores. Cada contenedor tiene la "ilusión" de estar ejecutándose sólo sobre el sistema operativo. Algunos ejemplos de esta aproximación serían Docker o Linux Containers.

- *Virtualización a nivel de aplicación*: las aplicaciones se ejecutan sobre un entorno virtualizado, favoreciendo su portabilidad entre múltiples sistemas. La aplicación puede ser consciente de que está siendo virtualizada (e.g. una aplicación Java puede ejecutarse en cualquier entorno que disponga de una JVM) o no (e.g. Wine permite la ejecución de una aplicación Windows en un sistema operativo Linux).

2.9. Virtualización hardware

La virtualización hardware crea un entorno de ejecución que simula el hardware de un computador. De ello se encarga una capa adicional de software (en ocasiones complementada con hardware) denominada *hipervisor* o *Virtual Machine Manager - VMM*. Sobre esta capa se pueden ejecutar múltiples guest que reciben el nombre de *máquinas virtuales* o *servidores virtuales*. Cada guest incluye su propio sistema operativo, así como las librerías y aplicaciones que se desean incluir en dicha máquina virtual.

De acuerdo con [Popek and Goldberg, 1974] un hipervisor debe cumplir con las siguientes características:

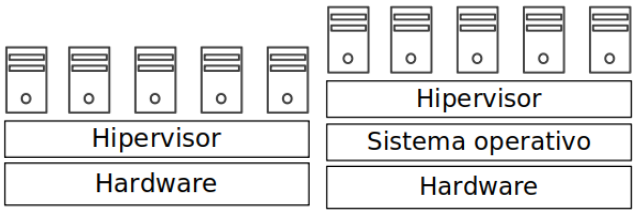
- La ejecución del guest sobre el hipervisor debe ser equivalente a su ejecución directamente sobre el host.

- La ejecución del guest sobre el hipervisor debe experimentar una penalización mínima de rendimiento.

- El hipervisor debe mantener un control absoluto sobre los recursos del sistema.

Existen dos clases principales de hipervisores [Goldberg, 1973]: hipervisores de tipo I e hipervisores de tipo II (ver figura 2.9).

El hipervisor de tipo I o *hipervisor nativo* (ver figura 2.9a) - se ejecuta directamente sobre el hardware, sin ningún intermediario. Es por tanto responsable de la planificación y reserva de los recursos del sistema. Este tipo de hipervisor es



(a) Hipervisor de tipo I (b) Hipervisor tipo II

Figura 2.9: Tipos de hipervisores

considerado el más eficiente y el más seguro, pero resulta más difícil de instalar y configurar, ya que debe soportar directamente el hardware subyacente. Xen, VMWare ESX Server y Microsoft Hyper-V son ejemplos de hipervisores de tipo I.

El hipervisor de tipo II o *hipervisor hosted* (ver figura 2.9a) - es una aplicación que se ejecuta en un sistema operativo, como un proceso más. Este tipo de hipervisor aprovecha la gestión del hardware que ya efectúa el sistema operativo y por lo tanto resulta muy sencillo de instalar y configurar. Sin embargo, no es tan eficiente como el tipo I, ya que añade una capa adicional que retarda todas las operaciones efectuadas sobre la máquina virtual, y además consume un mayor número de recursos. La seguridad de este hipervisor también se ve limitada, ya que cualquier problema de estabilidad del sistema operativo afectaría directamente al hipervisor. VMWare Workstation, Oracle VirtualBox, KVM y Microsoft VirtualPC son ejemplos de hipervisores de tipo II.

Para cumplir su cometido, el hipervisor utiliza diversas técnicas. Para comprender estas técnicas primero es necesario comprender un mecanismo de seguridad que implementan los sistemas informáticos actuales: los *dominios de protección jerárquica* o *anillos de protección*.

2.9.1. Anillos de protección

Los anillos de protección es una estrategia que implementan de manera conjunta el hardware y el sistema operativo y consiste en definir una colección de anillos de protección concéntricos en los que se ejecuta código. Cada anillo de protección representa un nivel de privilegios y permite al código que se ejecuta acceder a determinados recursos. Por ejemplo, en la arquitectura x86 el anillo 0 posee el máximo nivel de privilegios y permite acceso directo a todos los recursos físicos del sistema. El código que se ejecuta en este nivel se dice que se ejecuta en modo *privilegiado*, modo *kernel* o modo *supervisor*. Las aplicaciones de usuario se suelen ejecutar en el anillo 3; se dice que se ejecutan en modo *usuario*. Los anillos intermedios no suelen usarse. La figura 2.10 ilustra esta idea.

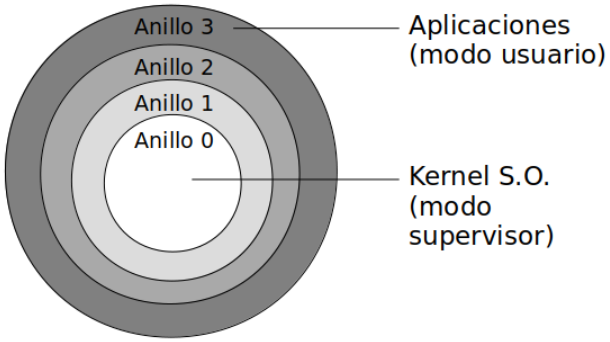


Figura 2.10: Anillos de protección

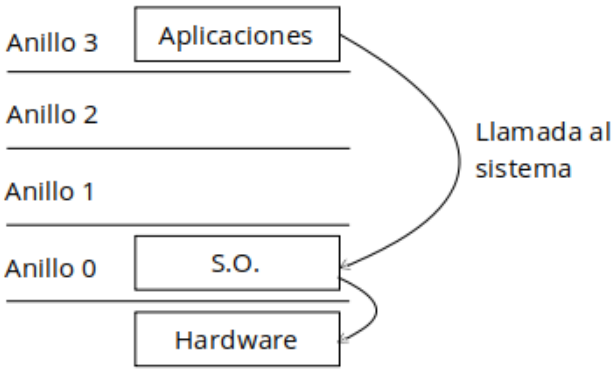
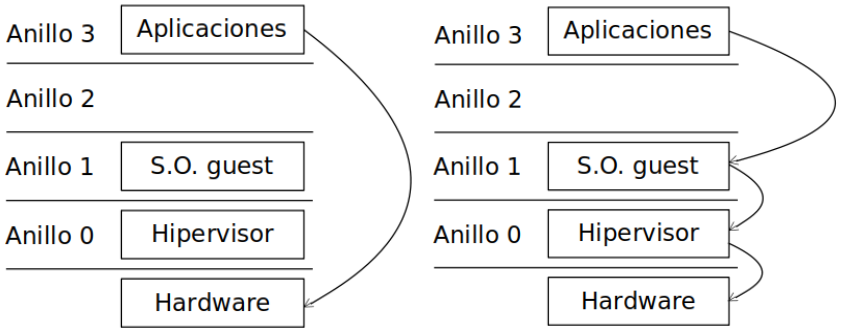


Figura 2.11: Llamada al sistema

Los sistemas operativos como *Linux* o *Windows* utilizan ambos modos. Las aplicaciones se ejecutan en modo usuario, que está muy restringido y no permite efectuar operaciones de entrada/salida. El núcleo del sistema operativo se ejecuta en modo *privilegiado*, en el anillo 0. Cuando una aplicación necesita ejecutar una operación de entrada/salida, efectúa una *llamada al sistema*. Entonces se ejecuta el código del núcleo (confiable) en modo *privilegiado*, y devuelve el control a la aplicación, pasando de nuevo al modo usuario (ver figura 2.11).

El hipervisor necesita acceder a los recursos del sistema sin limitaciones y por tanto debe ser ejecutado en el anillo 0. Por lo tanto, el sistema operativo guest debe ejecutarse en un anillo superior, con un menor nivel de privilegios. Existen distintas alternativas para habilitar esta estrategia:

- *Virtualización completa*
- *Paravirtualización*
- *Virtualización asistida por hardware*



(a) Ejecución de instrucciones no privilegiadas

(b) Ejecución de instrucciones privilegiadas

Figura 2.12: Virtualización completa

En las siguientes secciones se analizan estas estrategias.

2.9.2. Virtualización completa

El sistema operativo guest se ejecuta sin modificación alguna en el anillo 1. Las instrucciones no privilegiadas se ejecutan directamente sobre el hardware (figura 2.12a). Las instrucciones privilegiadas son interceptadas por el hipervisor, y son traducidas y emuladas utilizando técnicas como la *traducción binaria dinámica* (*DBT - dynamic binary translation*), que permite modificar instrucciones sobre la marcha (ver figura 2.12b).

Este modo de funcionamiento es costoso y comporta una importante sobrecarga en la ejecución del guest. La ventaja principal de esta aproximación es que el sistema operativo guest no necesita ser modificado. Ejemplos de esta aproximación son los hipervisores de tipo II QEMU, Oracle VirtualBox y Microsoft VirtualPC.

2.9.3. Paravirtualización

El sistema operativo guest se modifica para poder ejecutarse en el anillo 0, junto al hipervisor. El hipervisor publica una API de *hiperllamadas*, que es accedida por el sistema operativo guest (ver figura 2.13).

En esta configuración el diseño del hipervisor es más sencillo y la ejecución del guest es más eficiente. Sin embargo, resulta imprescindible modificar el núcleo de cada sistema operativo guest (y mantener este código actualizado). Además, bajo cargas muy pesadas, el uso de hiperllamadas puede comportar una sobrecarga significativa. Algunos ejemplos de esta aproximación son los hipervisores de tipo

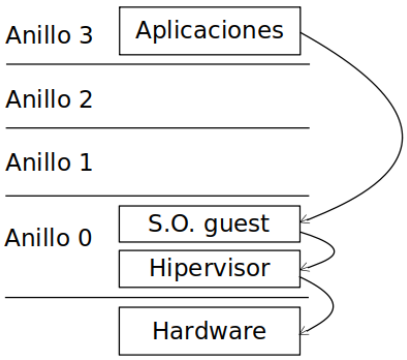


Figura 2.13: Paravirtualización

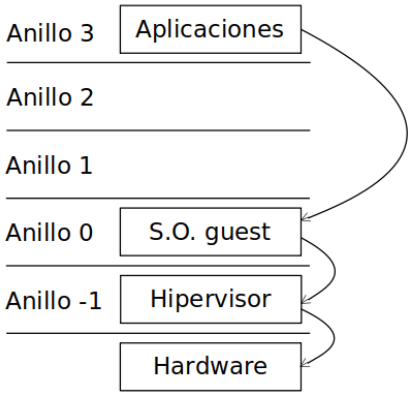


Figura 2.14: Virtualización asistida

I Xen y Microsoft Hyper-V.

2.9.4. Virtualización asistida por hardware

Para ofrecer un mejor soporte a la virtualización completa, los fabricantes de procesadores más populares han incorporado nuevas extensiones (Intel VT-x en 2005 y AMD-V en 2006) que introducen un nuevo anillo de protección -1 en el que se puede ejecutar el hipervisor. Por lo tanto, los sistemas operativos guest pueden ejecutarse en el anillo 0, sin modificación alguna y teniendo acceso a todos los recursos del sistema. Las llamadas privilegiadas son automáticamente redirigidas al hipervisor, evitando utilizar técnicas de traducción más costosas como la traducción binaria (ver figura 2.14).

Esta estrategia permite por tanto aplicar virtualización completa de una manera muy eficiente.

Oracle Virtual Box y VMware Workstation son algunos ejemplos de hiperviso-

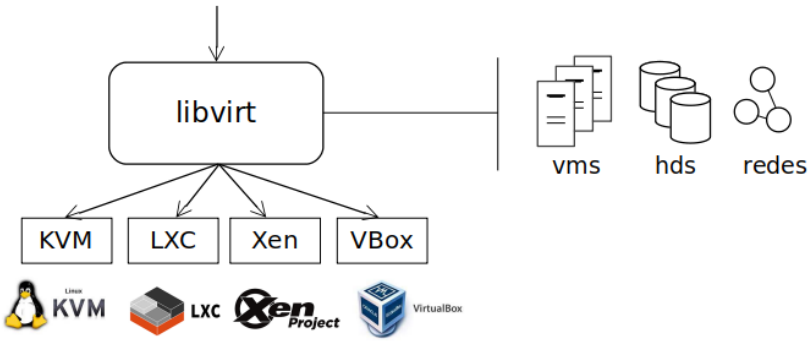


Figura 2.15: Libvirt

res de tipo II que aplican esta técnica, siempre y cuando las extensiones hardware estén disponibles; en otro caso implementan virtualización completa utilizando técnicas más complejas, como la traducción binaria dinámica. Por otro lado, VMWare ESXi y Microsoft Hyper-V son ejemplos de hipervisores de tipo I que aprovechan estos avances hardware.

KVM es un módulo que se inserta en el núcleo del sistema operativo Linux y también aprovecha las extensiones de virtualización hardware, pero representa un caso curioso, pues es considerado al mismo tiempo hipervisor tipo I y tipo II. Hipervisor de tipo II porque requiere un sistema operativo instalado previamente para poder funcionar, pero hipervisor de tipo I porque transforma el sistema operativo existente en hipervisor. KVM se utiliza en combinación con QEMU, para acelerar la ejecución de las máquinas virtuales creadas con este hipervisor de tipo II.

2.10. libvirt

En la actualidad existen multitud de soluciones de virtualización hardware: QEMU, KVM, VirtualBox, Xen, Virtual-PC, Hyper-V, VMWare, etc. Todas estas tecnologías permiten gestionar los mismos tipos de recursos virtuales utilizando distintas técnicas, y por medio de diversas herramientas y APIs. En lugar de estudiar un único hipervisor, en esta sección se presentará una solución denominada *libvirt*, que reúne en una única herramienta los conceptos compartidos por un gran número de hipervisores (ver figura 2.15).

Libvirt[15] es un *middleware* que permite gestionar múltiples hipervisores utilizando una API común. Esta API proporciona acceso a los principales recursos virtuales que la mayoría de hipervisores gestionan (e.g. máquinas virtuales, redes, volúmenes, etc.). Después, una colección de drivers se encarga de efectuar las

[15] https://libvirt.org/

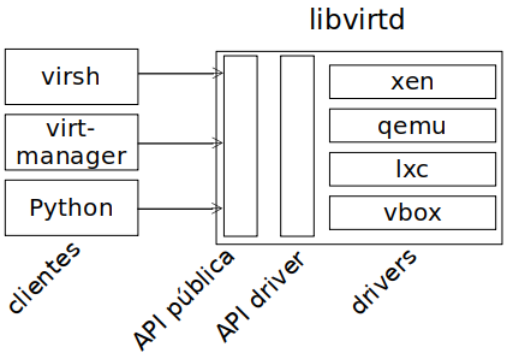


Figura 2.16: Arquitectura básica de libvirt

traducciones pertinentes a los hipervisores, que son quienes realmente gestionan los recursos.

Libvirt se limita a efectuar una operación sobre un único hipervisor en cada interacción. Por lo tanto, no soporta operaciones que trabajan simultáneamente sobre múltiples hipervisores. Es por ello que *libvirt* se utiliza como pieza básica para construir soluciones más complejas, como por ejemplo entornos cloud.

2.10.1. Arquitectura

Libvirt es una solución que permite gestionar los hipervisores instalados en un nodo. Para ello, cuenta con los siguientes elementos (ver figura 2.16):

- Los clientes
- El demonio libvirt
- Los drivers de libvirt

El cliente es un fragmento software que accede a la API publicada por el demonio *libvirt*. Puede ser una aplicación que accede directamente a la API o bien a través de una librería[16] (binding) facilitada en un lenguaje de programación específico, como *Python*, *Java* o *PHP*. Algunas herramientas cliente muy populares son:

- *virsh*: permite gestionar *libvirt* utilizando la línea de comandos (CLI)
- *virt-manager*[17]: es una herramienta que permite gestionar *libvirt* de manera gráfica

[16] https://libvirt.org/bindings.html
[17] https://virt-manager.org/

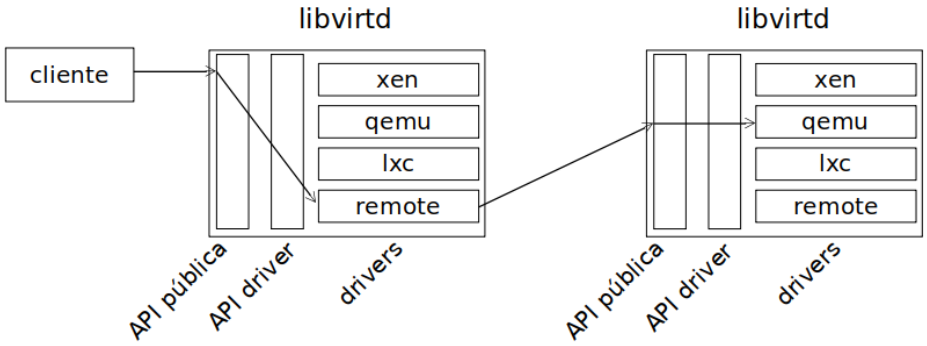


Figura 2.17: Libvirt remoto

- *virt-install*: es una herramienta CLI que facilita la instalación de un sistema operativo en una máquina virtual

- etc.

El demonio *libvirtd* debe estar ejecutándose en el nodo cuyos recursos se desea gestionar. En la actualidad este demonio monolítico está sufriendo un proceso de refactorización, y se prevé que en el futuro se descomponga en múltiples subdemonios, cada uno encargándose de conectar con un hipervisor diferente. Este demonio publica la API de libvirt a través de sockets UNIX (para clientes ejecutándose en el mismo nodo) y de sockets TCP (para clientes ejecutándose en otra máquina). El demonio puede funcionar en modo *system*, ejecutándose como *root* y con acceso a todas las funcionalidades o en modo *session*, ejecutándose como un usuario regular y por tanto proporcionando un subconjunto de todas las capacidades disponibles

El demonio *libvirtd* dispone de una colección de drivers que le permiten traducir las llamadas a la API en llamadas al hipervisor que finalmente ejecutará la operación. Cada driver proporciona acceso a un hipervisor diferente. Cuando el demonio *libvirtd* arranca, todos los drivers disponibles se registran. Cuando posteriormente un cliente solicita una operación sobre un hipervisor, el demonio *libvirtd* localiza el driver correspondiente y delega la operación en éste.

Cuando la operación tiene lugar de manera local, la comunicación entre el cliente y *libvirtd* tiene lugar a través de sockets UNIX. Cuando el cliente desea acceder al demonio *libvirtd* ejecutándose en otra máquina, la comunicación se produce a través de un driver específico denominado *remote*. El cliente se comunica con el demonio *libvirtd* local a través del driver *remote*. El demonio *libvirtd* local se conecta con el demonio *libvirtd* remoto. El demonio *libvirtd* remoto localiza el driver adecuado para la conexión y delega en éste (ver figura 2.17)

2.10.2. Instalación

El demonio *libvirt* únicamente puede ser instalado en máquinas *Linux*. Para ello, será necesario instalar el paquete pertinente según la distribución. Por ejemplo, con el siguiente comando se instala todo el subsistema *libvirt* en *Debian*:

```
> sudo apt-get install libvirt-daemon-system
```

Respecto a los clientes, algunos soportan distintos sistemas operativos. Por ejemplo, el cliente *virt-viewer*[18] puede ser instalado tanto en *Linux* como en *Windows*. Para instalar el cliente básico *virsh* se puede utilizar el siguiente comando en *Debian*:

```
> sudo apt-get install libvirt-clients
```

Para instalar el cliente gráfico *virt-manager* en *Debian* se puede usar el siguiente comando:

```
> sudo apt-get install virt-manager
```

Por último, para acceder a *libvirt* desde Python será necesario instalar el siguiente paquete en *Debian*:

```
> sudo apt-get install python-libvirt
```

2.10.3. El modelo libvirt

En esta sección se presentará la terminología y los conceptos que implementa *libvirt*. Se ilustrará cómo trabajar con *libvirt* utilizando la herramienta cliente *virsh* y el binding disponible para el lenguaje de programación Python.

En *libvirt* cada máquina física recibe la denominación de *nodo*. Un *nodo* puede contener múltiples *hipervisores*. Un *hipervisor* es una capa de software que permite crear múltiples máquinas virtuales. Cada máquina virtual recibe la denominación de *dominio*. La figura 2.18 ilustra este modelo.

[18] https://virt-manager.org/download/

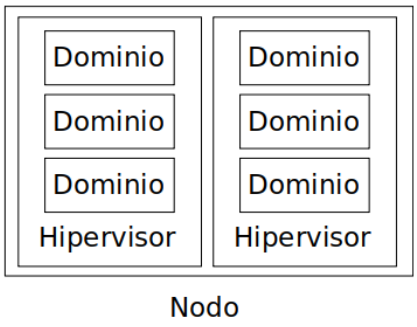


Figura 2.18: Conceptos libvirt

2.10.3.1. Conexiones

El primer paso para empezar a trabajar con *libvirt* consiste en crear una *conexión* contra un hipervisor. El hipervisor puede estar ejecutándose en el nodo local o en un nodo remoto, y es posible acceder a él en *modo sistema* (*system*) o *modo usuario* (*session*).

La conexión se define por medio una *URI*[19] que puede ser local o remota, con el siguiente formato:

```
driver[+unix]:///[system|session]   // local
driver[+transport]://[username@][hostname][:port]/[path][?params] // remota
```

Algunos ejemplos serían:

- `qemu:///system`: se conecta al hipervisor local QEMU en modo sistema.
- `vbox:///session`: se conecta al hipervisor local VirtualBox en modo usuario.
- `qemu://node.example.com/system`: se conecta al hipervisor QEMU ejecutándose en la máquina `node.example.com`.
- `xen+ssh://root@node.example.com/`: se conecta al hipervisor Xen ejecutándose en la máquina `node.example.com`. La conexión se efectúa a través de SSH utilizando el usuario `root`.

La herramienta *virsh* permite ejecutar distintos comandos sobre un hipervisor a través de *libvirt*. Para especificar el hipervisor al que se desea conectar se utiliza el parámetro `--connect=<url>` o `-c <url>`. A continuación se lista un ejemplo en el que se listan los dominios existentes en el hipervisor QEMU ejecutándose en la máquina local:

[19] https://libvirt.org/uri.html

Comando/Método	Descripción
hostname/getHostname()	Devuelve el nombre del host
maxvcpus/getMaxVcpus()	Devuelve el número máximo de CPUs por guest soportado
nodeinfo/getInfo()	Devuelve una lista con información variada del host (info de CPU, memoria, ...)
version/getVersion()	Devuelve la versión del driver
uri/getURI()	Devuelve la URI
...	

Cuadro 2.3: Comandos/métodos para obtener información

```
> virsh --connect=qemu:///system list --all
```

En Python, cualquier operación enviada a *libvirt* se debe realizar a través de una conexión. Por lo tanto, el primer paso siempre consiste en abrir una conexión contra un hipervisor. Para ello, se pueden utilizar las funciones `open()`, `openReadOnly()` u `openAuth()`. Una vez se ha acabado de trabajar con *libvir*, es necesario cerrar la conexión abierta con el método `.close()`. A continuación se lista un ejemplo:

```
import libvirt
import sys

con = None
try:
  con = libvirt.open("qemu:///system")
except libvirt.libvirtError as e:
  print(repr(e), file=sys.stderr)
  exit(1)
con.close()
exit(0)
```

Cada hipervisor (o su driver) dispone de unas capacidades diferentes. Es posible obtener esta información en formato XML a partir de la conexión utilizando el comando `virsh capabilities` o bien el método `.getCapabilities()` de la conexión en Python. Además, existen múltiples comandos/métodos que permiten obtener información adicional del hipervisor. En la tabla 2.3 se listan algunos de ellos.

A continuación se muestra cómo podría obtenerse información utilizando `virsh` primero y Python después:

```
> virsh -c qemu:///session hostname
> virsh -c qemu:///session nodeinfo


host = con.getHostname()
print('Hostname:'+host)

nodeinfo = con.getInfo()
print('Model: '+str(nodeinfo[0]))
print('Memory size: '+str(nodeinfo[1])+'MB')
print('Number of CPUs: '+str(nodeinfo[2]))
print('MHz of CPUs: '+str(nodeinfo[3]))
print('Number of NUMA nodes: '+str(nodeinfo[4]))
print('Number of CPU sockets: '+str(nodeinfo[5]))
print('Number of CPU cores per socket: '+str(nodeinfo[6]))
print('Number of CPU threads per core: '+str(nodeinfo[7]))
```

2.10.3.2. Dominios

Un dominio representa una máquina virtual, y posee varios identificadores:

- ID: identificador único de un dominio en el hipervisor. Un dominio inactivo recibe el valor de 0. Cualquier dominio en ejecución posee un valor entero superior.
- name: un string corto, único en un hipervisor.
- UUID: identificador único, independientemente del hipervisor en el que se ejecute el dominio,

Existen dos tipos de dominios:

- Volátiles (*transient*): sólo existen mientras están en ejecución. Cuando finalizan su ejecución se elimina cualquier rastro en el hipervisor.
- Persistentes (*persistent*): poseen una configuración que se mantiene en un almacén de datos del hipervisor. Cuando el dominio finaliza su ejecución, es posible gestionarlo a través de su configuración. Un dominio volátil puede ser transformado en persistente creando una configuración para él.

Para listar los dominios existentes en un hipervisor se utiliza el comando `virsh list`. Este comando admite múltiples flags, que permiten filtrar los dominios a consultar, por ejemplo: `--all`, `--inactive`, `--persistent`, `--transient`, etc. También es posible obtener la información de un dominio utilizando el comando `virsh dominfo`. A continuación se lista un ejemplo:

Método	Descripción
`.ID()`/`.UUIDString()`	Devuelve los identificadores del dominio
`.OSType()`	Devuelve el sistema operativo instalado
`.info()`	Devuelve una lista con información hardware [estado, maxmem, mem, cpus, cpu_time]
`.state()`	Devuelve el estado del dominio
...	

Cuadro 2.4: Métodos para consultar dominio

```
> virsh -c qemu:///session list --all
> virsh -c qemu:///session dominfo midominio
```

En Python existen diversos métodos para obtener información sobre los dominios:

- `.listDefinedDomains()`, `.listDomainsID()`: devuelve una lista con los IDs de los dominios activos.
- `.lookupByID()`, `.lookupByName()`, `.lookupByUUID()`: devuelve un objeto con información sobre el dominio especificado
- `.listAllDomains()`: devuelve una lista de objetos con información sobre dominios. Es posible filtrar los resultados utilizando las constantes pertenecientes al módulo `libvirt`: `.VIR_CONNECT_LIST_DOMAINS_[ACTIVE|INACTIVE|PERSISTENT|TRANSIENT|RUNNING|PAUSED|SHUTOFF|...]`

Una vez recuperado el objeto que contiene información sobre un dominio, es posible acceder a dicha información utilizando distintos métodos. En la tabla 2.4 se listan algunos de ellos.

Un dominio atraviesa diversas fases a lo largo de su vida (ver figura 2.19):

- *Undefined*: es el estado de partida, el dominio no se ha definido ni creado.
- *Defined*: el dominio se ha definido pero no está en ejecución.
- *Running*: el dominio está ejecutándose.
- *Paused*: el dominio está pausado y podría reactivarse en cualquier momento.
- *Saved*: el dominio está pausado y se ha guardado en forma de imagen en memoria persistente. Podría restaurarse en cualquier momento.

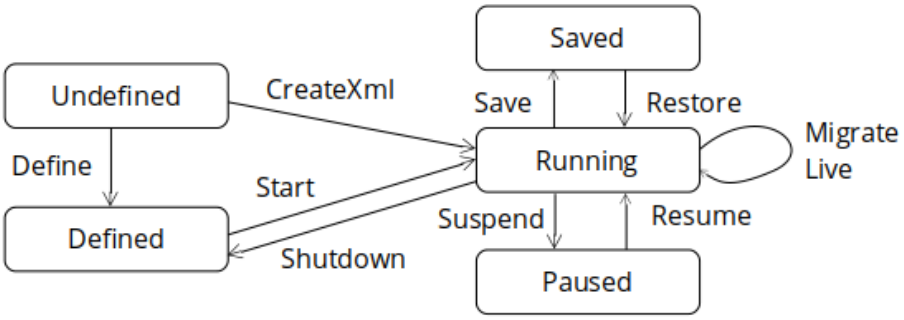


Figura 2.19: Ciclo de vida de un dominio

Para crear un nuevo dominio se requiere un fichero de configuración XML[20], y se pueden utilizar los siguientes comandos/métodos:

- El comando `virsh create <file>` o el método `con.createXML()`: crea un dominio volátil y lo ejecuta a partir de un fichero XML.
- El comando `virsh define <file>` o el método `con.defineXML()`: crea la definición de un dominio persistente a partir de un fichero XML, pero no lo ejecuta. Será necesario utilizar el comando `virsh start <dom>` o el método `dom.create()`.

Una manera sencilla de obtener el fichero XML es utilizar el comando `virsh dumpxml`, o el método `dom.XMLDesc()`, a partir de un dominio preexistente, y editarlo posteriormente. A continuación se muestra un ejemplo:

```
> virsh dumpxml <domain> > domain.xml
> virsh create domain.xml
```

```
dom = con.lookupByID(5)
xml = dom.XMLDesc(0)
dom = con.createXML(xml)
if dom == None:
  print('Failed to create a domain from an XML definition.',
        file=sys.stderr)
  exit(1)
print('Guest '+dom.name()+' has booted', file=sys.stderr)
```

Es muy habitual crear nuevos dominios a través de la herramienta gráfica `virt-manager` o bien la herramienta en línea de comandos `virt-install`:

[20] https://libvirt.org/formatdomain.html

Comando	Método	Descripción
`virsh start`	`dom.create()`	Arranca un dominio
`virsh shudown`	`dom.shutdown()`	Para un dominio de manera ordenada
`virsh destroy`	`dom.destroy()`	Para un dominio de manera abrupta
`virsh suspend`	`dom.suspend()`	Pausa un dominio en memoria
`virsh save`	`dom.save()`	Guarda un dominio en almacenamiento persistente
`virsh resume`	`dom.resume()`	Reactiva el dominio pausado
`virsh restore`	`dom.restore()`	Restaura el dominio
...	...	...

Cuadro 2.5: Comandos/métodos para gestionar el ciclo de vida de un dominio

```
virt-install \
             --connect qemu:///system \
             --virt-type kvm \
             --name MyNewVM \
             --ram 512 \
             --disk path=/var/lib/libvirt/images/MyNewVM.img,size=8 \
             --vnc \
             --cdrom /var/lib/libvirt/images/Fedora-14-x86_64-Live-KDE.iso \
             --network network=default,mac=52:54:00:9c:94:3b \
             --os-variant fedora14
```

Una vez creado el dominio, es posible modificar su estado con distintos comandos/métodos. La tabla 2.5 lista algunos de ellos.

2.10.3.3. Pools de almacenamiento

El almacenamiento persistente se gestiona a través de *pools*. Un *pool* es un espacio de almacenamiento configurado por el administrador del sistema. Puede ser un directorio local, un *share* de *NFS*, un dispositivo *iSCSI*, etc. Un pool contiene múltiples *volúmenes*. Cada *volumen* es utilizado por un dominio como un dispositivo de bloques, es decir, constituye el espacio de almacenamiento del dominio.

Existen múltiples comandos/métodos para gestionar los pools y los volúmenes existentes en un hipervisor. En la tabla 2.6 se listan algunos de ellos.

A continuación se lista un ejemplo:

```
pools = conn.listAllStoragePools(0)
if pools == None:
```

Comando	Método	Descripción
`virsh pool-list`	`con.listAllStoragePools()`	Lista los pools
`virsh pool-info`	`con.storagePoolLookupByName()`	Recupera la información de un pool
`virsh pool-create`	`con.storagePoolCreateXML()`	Crea un nuevo pool
`virsh pool-define`	`con.storagePoolDefineXML()`	Define un nuevo pool
`virsh pool-start`	`pool.start()`	Arranca un pool
`virsh pool-destroy`	`pool.destroy()`	Para un pool
`virsh pool-undefine`	`pool.undefine()`	Elimina un pool
`virsh vol-list`	`pool.listVolumes()`	Lista los volúmenes
`virsh vol-info`	`pool.storageVolLookupByName()`	Recupera la información de un volumen
`virsh vol-create`	`pool.createXML()`	Crea un nuevo volumen
`virsh vol-delete`	`vol.delete()`	Elimina un volumen
...	...	...

Cuadro 2.6: Comandos/métodos para gestionar pools y volúmenes

```
  print('Failed to locate any StoragePool objects.', file=sys.stderr)
  exit(1)
for pool in pools:
  info = pool.info()
  print('Pool: '+pool.name())
  print('  Num volumes: '+str(pool.numOfVolumes()))
  print('  Pool state: '+str(info[0]))
  print('  Capacity: '+str(info[1]))
  print('  Allocation: '+str(info[2]))
  print('  Available: '+str(info[3]))
```

2.10.3.4. Redes

En *libvirt* los dominios se interconectan por medio de sus dispositivos de red a *redes virtuales*. Para implementar una red virtual, *libvirt* utiliza el concepto de *switch virtual* (ver figura 2.20). En un host Linux el switch virtual se suele implementar por medio del *bridge* (*puente*) de Linux, una construcción software que permite reenviar paquetes a todas las interfaces de red conectadas al *bridge*.

Internamente, el bridge de Linux se implementa por medio de una interfaz de red. Al instalar el demonio *libvirtd*, se suele crear automáticamente la red virtual `'default'`, que implica la creación de la interfaz de red `virbr0`. Es posible

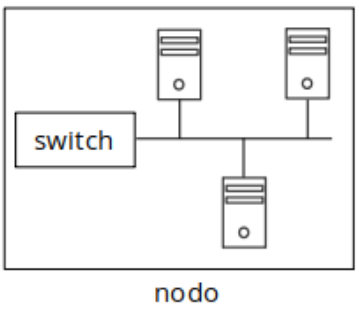


Figura 2.20: Switch virtual

obtener todas las interfaces de red del sistema emitiendo el siguiente comando:

```
> ip addr show
```

En libvirt, un switch virtual puede operar en tres modos diferentes:

- Modo NAT (*Network Address Translation*) (figura 2.21): es el modo por defecto. En este modo, cualquier dominio conectado al switch utiliza la dirección IP del host para comunicarse con el exterior, utilizando para ello un mecanismo de traducción de direcciones implementado por el firewal *iptables*. En este modo los dominios no son visibles desde el exterior. El switch virtual puede actuar de servidor DHCP/DNS. Para ello, *libvirt* utiliza el programa *dnsmasq*.
- Modo *routed* (figura 2.22): en este caso el switch virtual se conecta directamente a la LAN del host, y enruta el tráfico de los dominios directamente, sin traducir direcciones. Los dominios forman parte de una subred enrutada por el switch. Para que los paquetes lleguen desde el exterior es necesario configurar los routers externos, para que encaminen los paquetes adecuadamente.
- Modo *aislado* (figura 2.23): en este caso los dominios conectados al switch únicamente pueden comunicarse entre sí.

Una red virtual posee un nombre y un UUID, y puede ser volátil o persistente. Si es persistente, el hipervisor almacena su configuración en almacenamiento persistente. Para crear una red persistente es necesario especificar su configuración utilizando un fichero XML[21].

Existen múltiples comandos/métodos que permiten gestionar las redes virtuales. En la tabla 2.7 se listan algunos de ellos.

[21] https://libvirt.org/formatnetwork.html

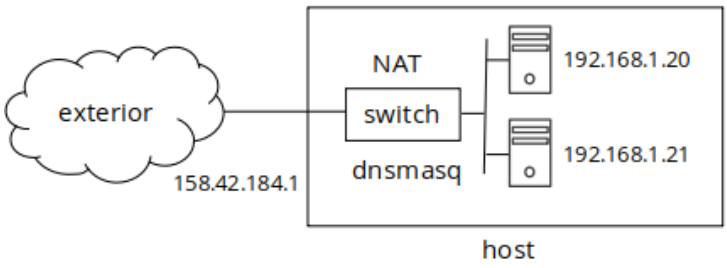


Figura 2.21: Modo de red *NAT*

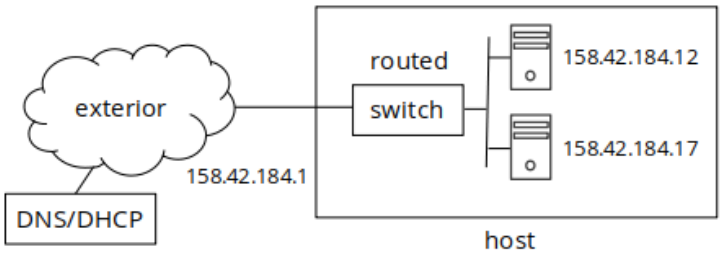


Figura 2.22: Modo de red *routed*

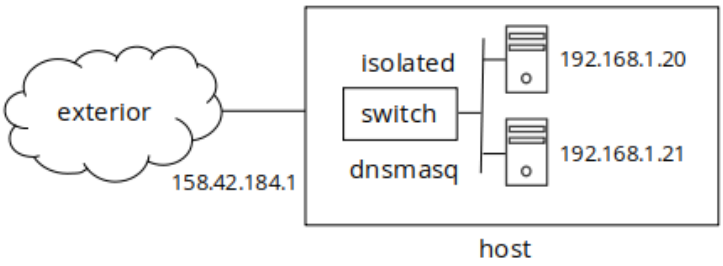


Figura 2.23: Modo de red *isolated*

Comando	**Método**	**Descripción**
`virsh net-list`	`con.listNetworks()`	Lista las redes
`virsh net-info`	`con.networkLookupByName()`	Recupera la información de una red
`virsh net-create`	`con.networkCreateXML()`	Crea una nueva red
`virsh net-define`	`con.networkDefineXML()`	Define una nueva red
`virsh net-start`	`net.start()`	Arranca una red
`virsh net-destroy`	`net.destroy()`	Para una red
`virsh net-undefine`	`net.undefine()`	Elimina una red
...	...	...

Cuadro 2.7: Comandos/métodos para gestionar redes

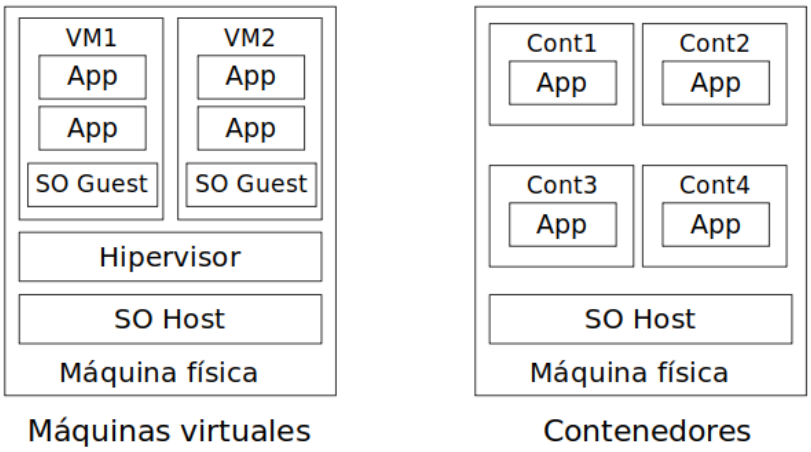


Figura 2.24: Máquinas virtuales vs contenedores

2.11. Contenedores

Un contenedor es un paquete que comprende un entorno de ejecución completo, incluyendo el código de aplicación, sus dependencias y todos los ficheros y herramientas de soporte. Cada contenedor se ejecuta de manera aislada sobre el sistema operativo y es posible monitorizar y controlar todos los recursos que utiliza. No se trata de un concepto nuevo, y es posible encontrar iniciativas similares en el pasado, que evitaban que un programa accediera a recursos protegidos, como las "jaulas" en FreeBSD o las "zonas" en Solaris. Los contenedores expanden esta idea, aislando a los procesos de todos los recursos, excepto de aquellos que explícitamente se especifican.

Por lo tanto, los contenedores son unidades autocontenidas que incluyen todos los recursos necesarios para su ejecución, lo cual simplifica al máximo su distribución y ejecución en múltiples plataformas. Por este motivo, los contenedores constituyen una herramienta extraordinaria para la implementación efectiva de microservicios.

Los contenedores utilizan tecnologías de virtualización ligera a nivel de sistema operativo, que permiten aislar los procesos y sus recursos en compartimentos estancos. Con estas tecnologías, todos los contenedores comparten una misma imagen del sistema operativo, algo que contrasta con la virtualización a nivel de hardware, en la que cada máquina virtual posee una imagen completa del sistema operativo. El resultado es una reducción drástica en el consumo de recursos y la posibilidad de ejecutar cientos de contenedores sobre una misma máquina física. En la figura 2.24 se muestra la estructura de un sistema ejecutando máquinas virtuales pesadas y contenedores.

Esta nueva aproximación de ejecución de software ha revolucionado la in-

dustria, y un gran número de compañías ha procedido a "contenerizar"[22] gran parte de sus sistemas. Los contenedores resultan también de gran utilidad en el contexto de las Devops, que incluyen todos los procesos de desarrollo, testing y operaciones del software. Gracias a esta nueva herramienta, desarrolladores, testers y personal de operaciones trabajan en un entorno homogéneo y exactamente bajo las mismas condiciones.

Debido a la gran importancia que esta tecnología ha adquirido en los últimos tiempos, han surgido iniciativas como OCI[23] (*Open Container Initiative*), con el objeto de crear estándares relacionados con los formatos y la ejecución de contendores.

En la actualidad existen diversas aproximaciones para implementar contenedores: Docker, LXC/LXD, OpenVZ, etc. La mayoría de estas alternativas están basadas en el sistema operativo Linux, que favorece su implementación suministrando las siguientes herramientas base:

- *Linux namespaces*: permite particionar los recursos del sistema operativo (i.e. procesos, ficheros, dispositivos de red, usuarios, etc.). Los procesos de una partición sólo pueden acceder a los recursos que pertenecen a la misma partición.

- *Cgroups* (a.k.a. control groups): permite aislar, monitorizar y limitar el consumo de recursos (i.e. CPU, memoria, I/O disco, red, etc.) de una colección de procesos.

- *Sistemas de ficheros estructurados en capas* (*layered file systems*), en los que cada capa posee un contenido diferente, pero pueden ser combinadas para ofrecer una imagen única del sistema de ficheros.

Gestionar todos estos instrumentos de manera manual es complejo y fuente de múltiples errores. Herramientas como Docker automatizan gran parte de los procesos y simplifican de manera dramática la gestión de contenedores, democratizando su uso para todo tipo de usuario.

2.12. Docker

En la actualidad, Docker[24] representa la tecnología de contenedores con mayor éxito. En realidad, Docker no implementa la tecnología de contenedores, sino que la hace extraordinariamente accesible, suministrando una colección de

[22] "Containerization" en inglés
[23] https://opencontainers.org
[24] https://www.docker.com/

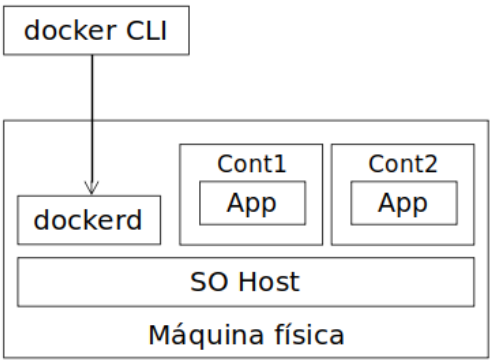


Figura 2.25: Cliente-servidor docker

herramientas e introduciendo nuevos conceptos y buenas prácticas. Por debajo, estas herramientas hacen uso de las facilidades ofrecidas por el núcleo del sistema operativo Linux para aislar procesos y recursos.

Docker suministra la tecnología de contenedores básica a través de *Docker Engine*. Se trata de una solución cliente-servidor, en la que el demonio `dockerd` publica una API REST (Docker Engine API), que es consumida por la herramienta en línea de comandos (CLI) `docker`. Esta última herramienta se limita a enviar los comandos solicitados por el usuario al demonio `dockerd`, que los ejecuta y devuelve los resultados. El demonio `dockerd` es el responsable de crear y gestionar todos los objetos Docker, como imágenes, contenedores, redes y volúmenes. La figura 2.25 ilustra este funcionamiento.

El demonio `dockerd` se ejecuta con privilegios de —root— y por defecto se comunica con sus clientes a través de un socket Unix, y no de un puerto TCP. Por defecto, el socket Unix es propiedad del usuario —root— y por tanto únicamente es posible accederlo con privilegios de superusuario (o utilizando el comando `sudo`). También es posible crear un grupo Unix denominado `docker` y añadir usuarios regulares. En este caso, cuando el demonio `dockerd` arranque, creará un socket Unix accesible a los miembros del grupo `docker`.

Una vez instalado el software, es posible comprobar la versión ejecutando el comando:

```
> docker --version
Docker version 19.03.13, build 4484c46d9d
```

En esencia, un *contenedor* Docker es un proceso en ejecución. Este proceso se ejecuta de manera totalmente aislada del host y del resto de contenedores, utilizando para ello las facilidades de *Linux namespaces* y *control groups*.

Cada contenedor es creado a partir de una *imagen* Docker, que proporciona el sistema de ficheros privado del contenedor. La imagen es un paquete que incluye

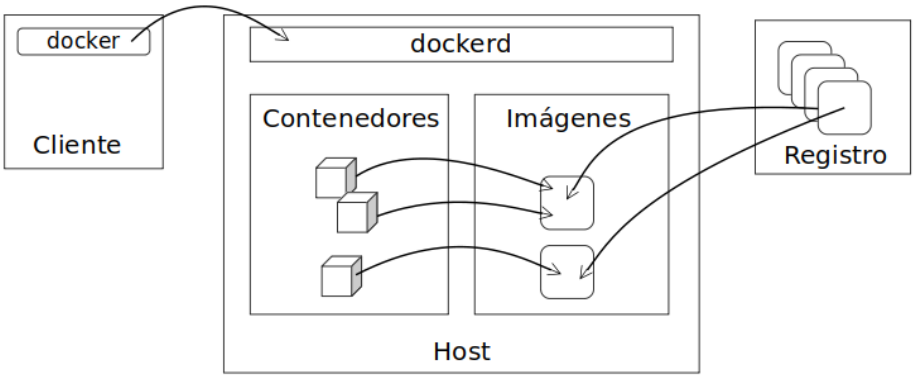


Figura 2.26: Arquitectura docker

todos los ficheros necesarios para ejecutar una aplicación: el código, el entorno de ejecución, las herramientas, librerías y configuraciones. A partir de una misma imagen es posible crear cualquier número de contenedores.

Las imágenes pueden ser locales o proceder de registros centralizados, como el *Docker Hub*, que contiene +100.000 imágenes registradas por proveedores, proyectos open-source o la comunidad. La figura 2.26 ilustra estos conceptos.

A continuación se procederá a crear un primer contenedor utilizando la herramienta CLI docker:

```
> docker run hello-world
```

Este comando contacta con el *Docker Hub*, descarga la imagen `hello-world`, la instala localmente y crea un nuevo contenedor a partir de ella. A continuación el contenedor comienza su ejecución y finaliza, mostrando por pantalla un mensaje. Si se vuelve a ejecutar el mismo comando, se creará un segundo contenedor a partir de la misma imagen. En este último caso el proceso será más rápido, ya que no será necesario volver a descargar la imagen `hello-world`, pues ya fue descargada anteriormente.

Utilizar Docker conlleva importantes beneficios:

- Gestión de dependencias. Una aplicación compuesta por múltiples componentes puede hacer uso de un gran número de librerías, con alta probabilidad de que unas librerías entren en conflicto con otras, por ejemplo en relación con el versionado. Con Docker, cada componente y sus dependencias se ejecutan en un contenedor aislado.

- Portabilidad. Cualquier contenedor puede ejecutarse en cualquier sistema con Docker instalado, garantizando las mismas condiciones de ejecución.

- Seguridad. El código que se ejecuta en el interior de un contenedor se ejecuta de manera aislada, y no tiene acceso a los recursos protegidos del sistema. Esto significa que sus efectos están estrictamente limitados a su contexto de ejecución, y no pueden propagarse fuera de él.

2.12.1. Contenedores

Todas las operaciones que permiten gestionar contenedores están disponibles a través del subcomando `docker container`.

```
> docker container --help
```

Para arrancar un nuevo contenedor se utiliza el comando `docker container run <image>` (el comando `docker run <image>` es un atajo).

```
> docker container run debian
```

Este comando ha efectuado mucho trabajo. Primero ha comprobado si la imagen `debian` estaba instalada localmente. Al no encontrarla, ha contactado con el *Docker Hub*, la ha descargado y la ha instalado localmente. A continuación ha creado un contenedor a partir de dicha imagen y por último lo ha ejecutado. Por defecto, la imagen `debian` ejecuta un comando vacío en el contenedor arrancado y finaliza su ejecución.

Es posible modificar el comando a ejecutar en el interior del contenedor, listándola a continuación:

```
> docker container run debian echo 'Hola mundo!'
Hola mundo!
```

Se puede observar que el comando se ha ejecutado de manera satisfactoria. El siguiente comando mostraría el contenido de la imagen `debian`.

```
> docker container run debian ls -la
```

Para listar los contenedores se utiliza el comando `docker container ls` (atajo con `docker ps`).

```
> docker container ls
CONTAINER ID    IMAGE     COMMAND             STATUS
```

Al ejecutar este comando se observa una lista vacía. Esto sucede porque por defecto sólo se listan los contenedores en ejecución. Todos los contenedores que se han ejecutado anteriormente ya han finalizado. Para listar todos los contenedores es necesario utilizar la opción `-a`.

```
> docker container ls -a
CONTAINER ID   IMAGE   COMMAND                  STATUS
f6702576b9bd  debian   "ls -la"                 Exited (0) 54 seconds ago
30ce4104b651  debian   "echo 'Hola mundo!'"     Exited (0) About a minute ago
343f664dd227  debian   "bash"                   Exited (0) About a minute ago
```

Ahora se listan todos los contenedores que se han ejecutado y que han finalizado (columna `STATUS`). Por defecto, una vez finalizado, el contenedor no se elimina. Para eliminarlo es necesario ejecutar el comando `docker container rm <container>` (`docker rm`).

```
> docker container rm f6702576b9bd
f6702576b9bd
```

Otra opción es arrancar el contenedor con la opción `--rm`. En este caso, cuando el contenedor finalice su ejecución, automáticamente será eliminado.

```
>  docker container run --rm debian ls -la
```

Cada contenedor posee un identificador generado aleatoriamente de tipo SHA-256 (32 bytes), que se representa por medio de un número hexadecimal de 64 dígitos. La probabilidad de generar dos identificadores idénticos es mínima y se puede asumir que los identificadores generados son únicos. Al listar los contenedores, aparecen sus identificadores en formato reducido (columna `CONTAINER ID`), con 12 caracteres hexadecimales en lugar de 32. Se puede obtener el identificador completo utilizando el siguiente comando:

```
> docker container inspect f6702576b9bd
[
  {
    "Id": "f6702576b9bda129659a3b72e8e303760de5be6628706ae7b85ff528a8...",
    "Created": "2020-11-08T15:24:45.9592318Z",
    "Path": "ls",
    "Args": [
      "-la"
    ],
    ...
  }
]
```

Es posible referirse a los contenedores por medio de sus identificadores (completos o en formato reducido), pero también se pueden utilizar nombres más inteligibles. Por defecto, todos los contenedores reciben un nombre generado de manera aleatoria (columna `NAMES`), pero es posible establecerlo de manera manual por medio del flag `--name`:

```
> docker container run --name micontenedor debian
```

Una vez un contenedor dispone de un nombre reconocible, es posible referenciarlo de una manera más amigable.

```
> docker container inspect micontenedor
[
  {
    "Id": "34502576b9bda129659a3b72e8e303760de5be6628706ae7b85ff528a8...",
    "Created": "2020-11-08T15:24:45.9592318Z",
     ...
  }
]
```

Un contenedor se puede ejecutar en modo interactivo (se captura la entrada por teclado y se crea un terminal virtual para poder interactuar con él) o en modo detached (se ejecuta en background y docker no espera a que finalice su ejecución). Para ejecutarlo en modo interactivo se utilizan los flags `--interactive --tty` (atajo con `-it`):

```
> docker container run -it --name c1 debian /bin/bash
root@3851bc24e8f8:/# ls
```

Entonces se podrá interactuar con el contenedor a través del intérprete de comandos `/bin/bash`. Si se finaliza el proceso con el comando `exit`, el contenedor finalizará su ejecución. Otra posibilidad consiste en pasar al contenedor a segundo plano, dejándolo en ejecución. Para ello es posible presionar `Ctrl+P Ctrl+Q`. Si ahora se listan los contenedores en ejecución aparecerá.

```
> docker container ls
CONTAINER ID    IMAGE     COMMAND         CREATED              STATUS
3851bc24e8f8    debian    "/bin/bash"     23 minutes ago       Up 23 minutes
```

Es posible retomar el modo interactivo utilizando el comando `docker container attach` (atajo con `docker attach`).

```
> docker container attach 3851bc24e8f8
root@3851bc24e8f8:/# exit
```

Al referirse al contenedor, es posible hacerlo a través de su identificador (completo o reducido), o de su nombre.

```
> docker container attach c1
root@3851bc24e8f8:/# exit
```

Para ejecutar un proceso en modo detached se utiliza el flag `--detached` (atajo con `-d`). Este modo permite ejecutar servicios en background. Por ejemplo, el siguiente comando crea un contenedor "c2" en modo detached que duerme indefinidamente, y por tanto no finaliza su ejecución:

```
> docker container run -d --name c2 debian sleep infinity
```

Si se listan los contenedores aparecerá este nuevo contenedor en ejecución.

```
> docker container ls
CONTAINER ID   IMAGE    COMMAND            CREATED         STATUS
beae179f9b12   debian   "sleep infinity"   5 minutes ago   Up 5 minutes
```

Si se desea observar la salida estándar de un contenedor ejecutándose en modo detached se puede utilizar el comando `docker container logs` (atajo con `docker logs`). En el siguiente ejemplo se arranca un contenedor "c3" que muestra el tiempo actual cada 5 segundos y posteriormente se observan las últimas 10 líneas de su salida estándar.

```
> docker container run -d --name c3 debian \\
  /bin/bash -c "while true; do echo `date`; sleep 5s; done"
> docker container logs --tail 10 c3
```

Si un contenedor está en ejecución, es posible ejecutar comandos en su interior utilizando el comando `docker container exec` (atajo con `docker exec`), tal y como se muestra en el siguiente ejemplo:

```
> docker container exec c3 ls -la
```

Durante su existencia, un contenedor pasa por múltiples estados, tal y como se muestra en la figura 2.27.

Es posible controlar el ciclo de vida de un contenedor por medio de los siguientes comandos:

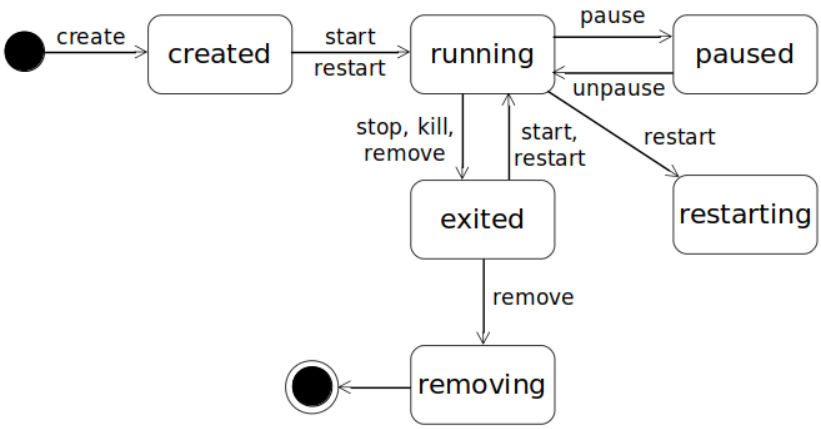


Figura 2.27: Ciclo de vida de un contenedor

- `docker container create <image>`: crea un contenedor, pero no lo arranca
- `docker container start <container>`: arranca un contenedor, ya existente
- `docker container pause <container>`: pausa un contenedor en ejecución (pausa todos los procesos ejecutándose en su interior)
- `docker container unpause <container>`: restaura un contenedor pausado (restaura todos los procesos que estaban pausados)
- `docker container restart <container>`: reinicia un contenedor
- `docker container stop <container>`: para un contenedor
- `docker container remove <container>`: elimina un contenedor

En realidad, el comando `docker container run debian` automatiza varias operaciones, que pueden efectuarse de manera individual:

1. Se comprueba si la imagen debian está instalada localmente.

   ```
   > docker image ls debian
   ```

2. Al no encontrarla, contacta con el *Docker Hub*, la descarga y la instala localmente.

   ```
   > docker image pull debian
   ```

3. Crea un contenedor a partir de la imagen. Al crear el contenedor se le asigna un identificador.

   ```
   > docker container create debian
   f45389d4704cf7c9a82a5e36b76995af0c0e5be801399ec7e75adc4...
   ```

4. Se ejecuta el contenedor.

   ```
   > docker container start f45389d4704cf7c9a82a5e36b76995...
   ```

Al arrancar un contenedor, es posible establecer variables de entorno con el flag `--env` (`-e`). De este modo, el host puede comunicar al contenedor parámetros de configuración de manera sencilla.

```
> docker run -it -e MSG=hello debian
root@02dbf3eac68d:/# echo $MSG
hello
```

2.12.2. Redes

Cada contenedor se ejecuta de manera aislada. Esto incluye su sistema de red. El sistema de red del contenedor queda totalmente desacoplado del sistema de red del host en el que se ejecuta. Cada contenedor posee su propia pila de red, es decir, posee su propia dirección IP, gateway, tabla de routing, servicios DNS, etc.

Sin embargo, la conectividad del contenedor con el exterior y con el resto de contenedores es plenamente configurable a través de una colección de drivers extensible suministrada por Docker. Cada driver es capaz de crear un tipo de red diferente, con unas características específicas.

Por defecto, Docker proporciona los siguientes drivers:

- `bridge`: permite interconectar a distintos contenedores por medio de un puente, que propaga los paquetes entre las distintas interfaces de red conectadas. Por defecto los contenedores no son visibles desde fuera de la red.
- `host`: el contenedor tiene acceso al sistema de red del host, eliminando el aislamiento.
- `macvlan`: permite asignar una dirección MAC al contenedor, y por tanto puede aparecer en la red como un dispositivo físico.
- `none`: deshabilita el subsistema de red.
- otros: existen múltiples drivers desarrollados por otras compañías.

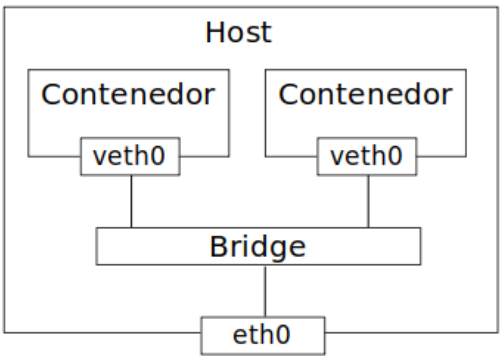


Figura 2.28: Redes bridge

2.12.2.1. Redes bridge

Un puente (bridge) es un dispositivo (puede ser físico o software) que propaga tráfico entre distintos segmentos de red. Utilizando este concepto, Docker permite gestionar redes de tipo `bridge`. Una red de tipo `bridge` utiliza un puente software (el bridge de Linux) que permite la comunicación entre todos los contenedores conectados a la misma red `bridge`. Para habilitar este efecto, Docker genera las reglas de red necesarias. La figura 2.28 ilustra este funcionamiento.

Una red `bridge` únicamente permite la comunicación entre contenedores que se ejecutan en el mismo host. Si se desea comunicar a contenedores que pertenecen a distintos hosts es necesario utilizar redes `overlay`.

Docker siempre crea una red `bridge` por defecto, denominada `bridge`. Es posible listar todas las redes existentes por medio del siguiente comando:

```
> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
87734e3e357f        bridge              bridge              local
08bf0f8a3dcb        host                host                local
b37b884f3a36        none                null                local
```

Cuando se crea un contenedor, por defecto se conecta a la red `bridge`, recibe una dirección IP incluida en dicha red y puede comunicarse con el exterior y con el resto de contenedores que pertenecen a la misma red. Sin embargo, no puede recibir tráfico externo de manera directa, pues la red `bridge` queda oculta fuera del host. En las siguientes líneas se crea un contenedor con nombre "c1" y a continuación se hace ping a una máquina en Internet.

```
> docker container run -it --name c1 debian /bin/bash
root@058883bd8472:/# ping www.google.com
```

```
PING www.google.com (216.58.201.164) 56(84) bytes of data.
64 bytes from mad08s06-in-f4.1e100.net ... ttl=115 time=17.9 ms
64 bytes from mad08s06-in-f4.1e100.net ... ttl=115 time=17.9 ms
...
```

Es posible observar las interfaces de red del contenedor ejecutando el siguiente comando en el interior del contenedor:

```
root@058883bd8472:/# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state ...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
10: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ...
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff ...
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

También se puede obtener toda la información sobre el contenedor (incluyendo sus interfaces de red) desde el host utilizando el siguiente comando:

```
> docker container inspect c1
[
  {
  "Id": "058883bd84729a78a5a8ed9ba6e157485a148b76910e49f9cdb36c...",
  "Created": "2020-11-08T14:47:49.6945016Z",
  ...
  "NetworkSettings": {
    ...
    "Networks": {
      "bridge": {
        "IPAMConfig": null,
        "Links": null,
         "Aliases": null,
         "NetworkID": "a657300130d9423085f6954ccc046e08ad2fde27...",
         "EndpointID": "d4fbfe80a52145a5af10ce35c236ab369d09e3a...",
         "Gateway": "172.17.0.1",
         "IPAddress": "172.17.0.2",
         "IPPrefixLen": 16,
         "IPv6Gateway": "",
         "GlobalIPv6Address": "",
```

```
            "GlobalIPv6PrefixLen": 0,
            "MacAddress": "02:42:ac:11:00:02",
            "DriverOpts": null
        }
      }
    }
]
```

El comando devuelve gran cantidad de información formateada en JSON. Bajo la entrada `NetworkSettings > Networks` es posible encontrar toda la información de red.

Como se puede observar, la dirección IP del contenedor "c1" es 172.17.0.2. A continuación se dejará al contenedor "c1" ejecutándose en segundo plano, con la combinación de teclas `Ctrl+P Ctrl+Q`. Desde una consola en el host es fácil comprobar que el contenedor es alcanzable:

```
> ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.120 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.063 ms
...
```

Sin embargo, si se dispone de otra máquina física conectada a la misma red, se puede comprobar que el contenedor no es visible.

A continuación se creará un segundo contenedor "c2" y se comprobará que el contenedor "c1" también es alcanzable desde éste.

```
> docker container run -it --name c2 debian /bin/bash
root@450ea5cc22d9:/# ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.203 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.079 ms
...
```

Con esta configuración de red es muy sencillo crear distintos contenedores interconectados entre sí. A continuación se muestra un ejemplo que utiliza la herramienta netcat, una sencilla utilidad que permite trabajar con conexiones TCP/UDP. Para ello, a continuación se abrirán dos consolas, cada una conectada a un contenedor. Se instalará el paquete netcat[25] en el interior de ambos contenedores. Se arrancará netcat en modo servidor en el contenedor "c1" y se abrirá una conexión cliente desde "c2" hasta "c1".

[25] http://netcat.sourceforge.net

```
> docker attach c1
root@058883bd8472:/# apt-get update && apt-get install -y netcat
root@058883bd8472:/# nc -l -p 8000

> docker attach c2
root@450ea5cc22d9:/# nc 172.17.0.2 8000
```

Una vez establecida la conexión entre cliente/servidor es posible observar que cada línea escrita en el cliente "c2" es correctamente recibida en el servidor "c1".

Además de la red `bridge` creada por defecto por Docker, es posible crear más redes bridge, es lo que Docker denomina redes `bridge de usuario`. Cada red `bridge de usuario` crea una nueva red local a la que es posible añadir nuevos contenedores. Trabajar con redes `bridge de usuario` conlleva algunas ventajas:

- Estas redes implementan un mecanismo de resolución de nombres de contenedores DNS automático. Es posible referirse a los contenedores por sus nombres en lugar de únicamente por sus IPs.
- Mejora el aislamiento de contenedores, de manera que únicamente podrán comunicarse entre sí los contenedores incluidos en la red.
- Es posible configurar con diferentes parámetros cada red `bridge de usuario` (e.g. MTU, reglas iptables, etc.)

El siguiente comando crea una nueva red `bridge de usuario`:

```
> docker network create mynet
640c6b85729cdf42f4a8ebe6ce0c8f9a590e2e6486a257242814e4d6ad34cb65
> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
87734e3e357f        bridge              bridge              local
08bf0f8a3dcb        host                host                local
640c6b85729c        mynet               bridge              local
b37b884f3a36        none                null                local
```

Como se puede observar, la nueva red aparece en el listado de redes. Una vez creada, es posible arrancar nuevos contenedores en su interior utilizando el flag `--network`.

```
> docker container run -d --name c3 --network mynet debian sleep infinity
> docker container run -it --name c4 --network mynet debian
root@2ee48b387f94:/# ip addr
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN ...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
23: eth0@if24: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc ...
    link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.18.0.3/16 brd 172.18.255.255 scope global eth0
       valid_lft forever preferred_lft forever
root@2ee48b387f94:/# ping www.google.com
PING www.google.com (216.58.201.164) 56(84) bytes of data.
64 bytes from mad08s06-in-f4.1e100.net ... ttl=115 time=17.9 ms
64 bytes from mad08s06-in-f4.1e100.net ... ttl=115 time=18.1 ms
```

Como con cualquier red `bridge` los contenedores "c3" y "c4" reciben una dirección IP privada en la red, pueden acceder al exterior, y son alcanzables desde el host. Además, los contenedores pueden resolver sus nombres de manera automática, sin necesidad de conocer sus direcciones IP.

```
root@2ee48b387f94:/# ping c3
PING c3 (172.18.0.2) 56(84) bytes of data.
64 bytes from c3.mynet (172.18.0.2): ... ttl=64 time=0.158 ms
```

Sin embargo, los contenedores "c3" y "c4" no son visibles desde un contenedor ubicado en cualquier otra red `bridge`.

```
> docker container attach c1
root@058883bd8472:/# ping c3
ping: c3: Name or service not known
root@058883bd8472:/# ping 172.18.0.3
PING 172.18.0.3 (172.18.0.3) 56(84) bytes of data.
```

Es posible conectar un contenedor en ejecución a una nueva red con el siguiente comando:

```
> docker network connect mynet c1
```

Si se ejecuta el siguiente comando, es posible observar bajo la entrada `NetworkSettings/Networks` que ahora el contenedor está conectado a dos redes, y posee dos direcciones IP.

```
> docker inspect c1

[
  {
  "Id": "058883bd84729a78a5a8ed9ba6e157485a148b76910e49f9cdb...",
  "Created": "2020-11-08T14:47:49.6945016Z",
  ...
  "NetworkSettings": {
    ...
    "Networks": {
      "bridge": {
        "IPAMConfig": null,
        "Links": null,
         "Aliases": null,
         "NetworkID": "a657300130d9423085f6954ccc046e08ad2f...",
         "EndpointID": "d4fbfe80a52145a5af10ce35c236ab369d0...",
         "Gateway": "172.17.0.1",
         "IPAddress": "172.17.0.2",
         "IPPrefixLen": 16,
         "IPv6Gateway": "",
         "GlobalIPv6Address": "",
         "GlobalIPv6PrefixLen": 0,
         "MacAddress": "02:42:ac:11:00:02",
         "DriverOpts": null
       },
       "mynet": {
         "IPAMConfig": {},
         "Links": null,
         "Aliases": [ "920face5bd15" ],
         "NetworkID": "640c6b85729cdf42f4a8ebe6ce0c8f9a590e2e6...",

         "EndpointID": "2eb9f73933f55a8c562a14427522582bad8879...",
         "Gateway": "172.18.0.1",
         "IPAddress": "172.18.0.4",
         "IPPrefixLen": 16,
         "IPv6Gateway": "",
         "GlobalIPv6Address": "",
         "GlobalIPv6PrefixLen": 0,
         "MacAddress": "02:42:ac:12:00:04",
         "DriverOpts": {}
       }
```

```
    }
  }
]
```

Desde el interior del contenedor es sencillo comprobar su conectividad, ya que todos los contenedores de ambas redes están accesibles.

```
> docker container attach c1
root@058883bd8472: ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state ...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
25: eth0@if26: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ...
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff ...
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
27: eth1@if28: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ...
    link/ether 02:42:ac:12:00:04 brd ff:ff:ff:ff:ff:ff ...
    inet 172.18.0.4/16 brd 172.18.255.255 scope global eth1
       valid_lft forever preferred_lft forever
```

Es posible desconectar a un contenedor de una red por medio del siguiente comando:

```
> docker network disconnect mynet c1
```

2.12.2.2. Port mapping

Por defecto, los puertos publicados por un contenedor no son accesibles desde el mundo exterior. El contenedor se ejecuta en el interior de un host y sólo es visible desde el host y desde los contenedores ejecutándose en la misma red `bridge`.

Docker permite mapear los puertos de un contenedor al host, dando visibilidad externa al servicio publicado por dicho contenedor. Para ello, docker crea reglas en el firewall del host que habilitan port forwarding. Para ello, al arrancar el contenedor se utiliza el flag `--publish <port_host>:<port_container>` (atajo con `-p`). En el siguiente ejemplo se arranca un servidor `nginx` y se mapea su puerto al puerto 8080 del host, siendo por tanto accesible desde el exterior de la red `bridge` en la que se arranca el contenedor:

```
> docker container run --name nginx -d -p 8080:80 nginx
```

Para obtener los puertos de un contenedor que actualmente están mapeados al host es posible utilizar el siguiente comando:

```
> docker port nginx
80/tcp -> 0.0.0.0:8080
```

2.12.2.3. Red host

Si se añade un contenedor a la red host, entonces el contenedor y el host comparten la misma pila de red. Ello significa que el contenedor no recibe una dirección IP privada, ni ninguna otra configuración de red.

Con esta configuración, cuando el contenedor publica un servicio en un puerto, está directamente accesible desde el mundo exterior, y no es necesario habilitar mecanismos de mapeo de puertos. Si se ejecuta el siguiente comando es posible comprobar que el servidor web nginx está disponible directamente en el puerto 80 del host.

```
> docker run --name nginx --network host -d nginx
```

La ventaja de esta aproximación es que las comunicaciones pueden resultar más eficientes, al no existir redirección de paquetes. Sin embargo, al compartir la pila de red, el contenedor podría potencialmente comprometer al host en el que se ejecuta. No es recomendable utilizar esta estrategia salvo que existe un motivo importante para ello.

2.12.3. Almacenamiento

Por defecto, todos los ficheros creados/modificados en el interior de un contenedor se almacenan en una capa de lectura/escritura ubicada por encima de la imagen a partir de la cual se crea el contenedor. Esta capa es temporal y se elimina automáticamente cuando el contenedor se destruye.

Docker proporciona dos herramientas fundamentales para gestionar la información persistente de los contenedores: *volúmenes* y *mounts* (o puntos de montaje). Independientemente del mecanismo de persistencia escogido, el contenedor podrá acceder a la información a través de un directorio o por medio de un fichero contenido en su sistema de ficheros.

Los *volúmenes* permiten almacenar toda la información en un lugar gestionado íntegramente por Docker, que se encarga de controlar su ciclo de vida, facilitando su compartición, copia, migración, etc. Con este mecanismo, el contenedor y sus volúmenes resultan entidades sumamente portables. Además, Docker habilita un sistema de plugins que permite almacenar la información a través de

distintos medios y por distintas vías (e.g. directorios locales, servidores remotos, etc.). Los volúmenes son el mecanismo de persistencia recomendado por Docker.

Los *mounts* permiten montar en el interior del contenedor ficheros o directorios existentes en el host. Al no existir intermediarios, el mecanismo es muy eficiente. Por ello, es habitual utilizar este mecanismo para compartir ficheros de configuración o repositorios de código fuente entre el host y sus contenedores. Sin embargo, se crea una dependencia adicional entre el contenedor y el host en el que se ejecuta, ya que el contenedor espera que el host disponga de ciertos ficheros o una determinada estructura de directorios, quedando por tanto comprometida la portabilidad de la solución.

2.12.3.1. Volúmenes

Se trata del mecanismo de persistencia recomendado por Docker. Los volúmenes se almacenan en un lugar específico del host (`/var/lib/docker/volumes` para volúmenes locales en Linux) y son íntegramente gestionados por Docker. Los volúmenes deberían ser modificados únicamente desde el interior de los contenedores.

Por defecto, cuando se crea un volumen, Docker crea un directorio en el host. Cuando el volumen se monta en un contenedor, el contenedor tiene acceso a dicho directorio. Un volumen puede montarse en múltiples contenedores al mismo tiempo.

Un volumen puede tener un nombre o ser anónimo. Si es anónimo, Docker genera automáticamente un identificador.

Se pueden crear volúmenes de dos maneras:

- Explícitamente, utilizando el comando `docker volume create`
- Implícitamente, creando un contenedor que haga uso de un volumen nuevo

A continuación se crea un nuevo volumen, y se listan todos los volúmenes gestionados por Docker:

```
> docker volume create vol1
vol1
> docker volume ls
DRIVER          VOLUME NAME
local           vol1
```

También es posible inspeccionar todos los detalles de un volumen utilizando el comando `docker volume inspect`:

```
> docker volume inspect vol1
[
    {
        "CreatedAt": "2020-11-11T13:56:25+01:00",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/vol1/_data",
        "Name": "vol1",
        "Options": {},
        "Scope": "local"
    }
]
```

Para montar un volumen en un contenedor se utiliza el flag `--volume <vol-name>:<mount-path>:<opts>` (atajo con `-v`), especificando el nombre del volumen (si es anónimo este campo no es necesario), el punto de montaje dentro del contenedor (path absoluto) y opcionalmente el modo (`ro`). Si el volumen no existe se crea automáticamente.

```
> docker run -it -v vol2:/vol2 debian /bin/bash
root@4e424bc2ca19:/# touch /vol2/hola.txt
root@4e424bc2ca19:/# ls -la /vol2
total 8
drwxr-xr-x 2 root root 4096 Nov 11 13:12 .
drwxr-xr-x 1 root root 4096 Nov 11 13:12 ..
-rw-r--r-- 1 root root    0 Nov 11 13:12 hola.txt
```

A continuación es posible comprobar que el volumen se ha creado:

```
> docker volume inspect vol2
[
    {
        "CreatedAt": "2020-11-11T14:12:36+01:00",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/vol2/_data",
        "Name": "vol2",
        "Options": null,
        "Scope": "local"
    }
]
```

Y aunque no es recomendable, se puede observar desde el host que el directorio especificado en la entrada `Mountpoint` efectivamente posee el contenido de dicho volumen.

El siguiente comando arranca un contenedor con MongoDB que almacena toda la información en el volumen `vol3`.

```
> docker container run -it --name mongo -v vol3:/data/db -d mongo
```

A continuación es sencillo obtener la IP del nuevo contenedor, inspeccionando la entrada `NetworkSettings/Networks` y conectarse con el servidor a través de la herramienta CLI `mongo`:

```
> docker container inspect mongo
[
  {
  "Id": "1377f08bba8f482954dfd3701f4efc75aca0530998e08ade08ad3...",
  "Created": "2020-11-12T11:47:50+01:00",
  ...
  "NetworkSettings": {
    ...
    "Networks": {
      "bridge": {
        "IPAMConfig": null,
        "Links": null,
         "Aliases": null,
         "NetworkID": "a65730ade87f8eae5bb03f7c2816d0d3132fde27...",
         "EndpointID": "d4100d49d8f95813dee833877895fe369d09e3a...",
         "Gateway": "172.17.0.1",
         "IPAddress": "172.17.0.2",
         "IPPrefixLen": 16,
         "IPv6Gateway": "",
         "GlobalIPv6Address": "",
         "GlobalIPv6PrefixLen": 0,
         "MacAddress": "02:42:ac:11:00:02",
         "DriverOpts": null
       }
     }
   }
]

> mongo 172.17.0.2
```

```
> show dbs
> exit
```

A continuación, se obtendrá el path donde está almacenando la información el contenedor con el siguiente comando:

```
> docker volume inspect vol3
[
    {
        "CreatedAt": "2020-11-12T11:47:50+01:00",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/vol3/_data",
        "Name": "vol3",
        "Options": null,
        "Scope": "local"
    }
]
```

Y se puede comprobar que efectivamente MongoDB está almacenando toda la información en dicho directorio.

Los volúmenes representan un mecanismo muy conveniente para crear backups, restaurarlos o migrar datos entre contenedores. Para efectuar estas operaciones es necesario acceder al volumen desde el interior de contenedores específicamente creados para efectuar estas operaciones. En estas operaciones resulta muy útil el flag `--volumes-from`, que permite montar los volúmenes de un contenedor determinado. Por ejemplo, el siguiente comando crea un contenedor y un volumen:

```
> docker run -v /data --name store debian /bin/bash
```

El siguiente comando arranca un contenedor que monta dicho volumen y crea un backup en el fichero `backup.tar` ubicado en el directorio de trabajo actual:

```
> docker run --volumes-from store -v $(pwd):/backup debian tar cvf /backup/backup.ta
```

El siguiente comando arranca un contenedor que restaura el backup en un segundo volumen:

```
> docker run -v /data2  -v $(pwd):/backup debian bash -c "cd /data2 && tar xvf /back
```

Los volúmenes existen de manera independiente a los contenedores. Pueden eliminarse de manera explícita utilizando el comando `docker volume rm`:

```
> docker volume ls
DRIVER              VOLUME NAME
local               vol1
> docker volume rm vol1
vol1
```

Por último, se pueden eliminar los volúmenes no usados como el siguiente comando:

```
> docker volume prune
```

Para la gestión de volúmenes, Docker utiliza un sistema extensible basado en drivers/plugins (`VolumeDriver`). Esta estrategia simplifica la creación de nuevos tipos de volúmenes, que se almacenan en local o en sistemas remotos (e.g. NFS, Amazon S3, ...). Independientemente del driver escogido para gestionar un determinado volumen, el contenedor siempre accede del mismo modo al contenido del mismo.

Para utilizar un nuevo driver el primer paso consiste en instalarlo. El siguiente ejemplo instala un driver que permite trabajar con volúmenes almacenados en una máquina remota a través de SSH.

```
> docker plugin install --grant-all-permissions vieux/sshfs
```

A continuación, para crear un volumen utilizando dicho driver es necesario especificar el flag `--driver`. Algunos drivers requieren información adicional para poder proceder. Es posible especificar estas opciones con el parámetro `-o`. En el caso del driver `vieux/sshfs` será necesario especificar información de autenticación en la máquina remota, tal y como se muestra a continuación:

```
> docker volume create --driver vieux/sshfs \
  -o sshcmd=test@node2:/home/test \
  -o password=testpassword \
  sshvolume
```

Cada driver posee unas opciones diferentes, de manera que es necesario consultar la documentación para comprender cómo funciona.

2.12.3.2. Mounts

Cuando se utiliza un mount, un fichero o un directorio del host se monta directamente en el contenedor. El fichero/directorio es referenciado por medio de su path completo en el host. Si no existe, se crea automáticamente.

Los mounts poseen capacidades limitadas con respecto a los volúmenes, y no pueden ser gestionados con la herramienta CLI `docker`, pero resultan muy eficientes, al no existir intermediarios entre el contenedor y el fichero/directorio. Sin embargo, comprometen seriamente la portabilidad y la independencia del contenedor, ya que únicamente podrán utilizar los mounts si estos existen en el host en el que se ejecutan.

Para utilizar un mount en un contenedor se utiliza el flag `--volume <host-path>:<mount-path>:<opts>` (atajo con `-v`), especificando el path absoluto del fichero/directorio en el host, el path absoluto donde se desea montar dentro del contenedor y opcionalmente el modo (`ro`).

En el siguiente ejemplo, se crea un contenedor que monta el directorio del host `$(pwd)/data` (`/data` en el directorio de trabajo actual) en el contenedor y genera un fichero `hello.txt` en su interior.

```
> docker run -it -v $(pwd)/data:/data debian
root@ee7a5d9609af:/# touch data/hello.txt
root@ee7a5d9609af:/# ls -la /data/
total 8
drwxr-xr-x 2 root root 4096 Nov 12 11:32 .
drwxr-xr-x 1 root root 4096 Nov 12 11:32 ..
-rw-r--r-- 1 root root    0 Nov 12 11:32 hello.txt
root@ee7a5d9609af:/# exit
```

Desde fuera del contenedor también es fácil comprobar que el fichero se ha creado satisfactoriamente y se puede consultar su contenido.

2.12.4. Imágenes

Cada contenedor se crea a partir de una imagen. Una imagen se compone de una pila de capas. Cada capa aplica modificaciones sobre la inmediata anterior, añadiendo, modificando o eliminando contenido. Las capas únicamente mantienen las diferencias sobre la capa anterior, optimizando al máximo su dimensión.

Todas las capas de una imagen son de sólo lectura. Cuando se crea un contenedor a partir de una imagen, se añade una capa adicional de lectura/escritura por encima del resto. Esta capa suele denominarse la "capa del contenedor". Todos los cambios efectuados por el contenedor se almacenan en esta capa. Esta estructura en capas se ilustra en la figura 2.29.

Por lo tanto, todos los contenedores que se basan en la misma imagen comparten las mismas capas de sólo lectura (ver figura 2.30), aunque cada uno posee su propio estado, encapsulado en su capa de lectura/escritura. Cuando el contenedor se elimina, únicamente se elimina su capa de lectura/escritura.

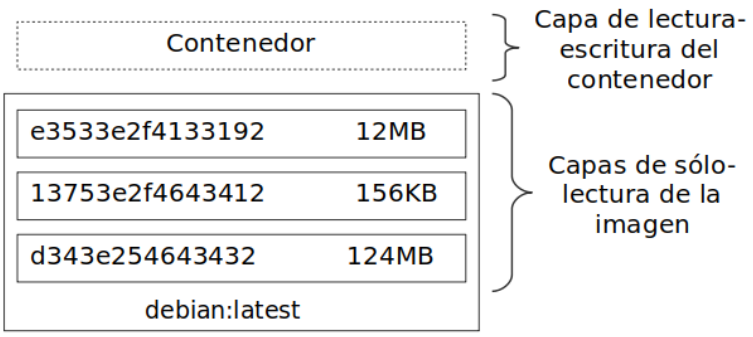


Figura 2.29: Capas de un contenedor

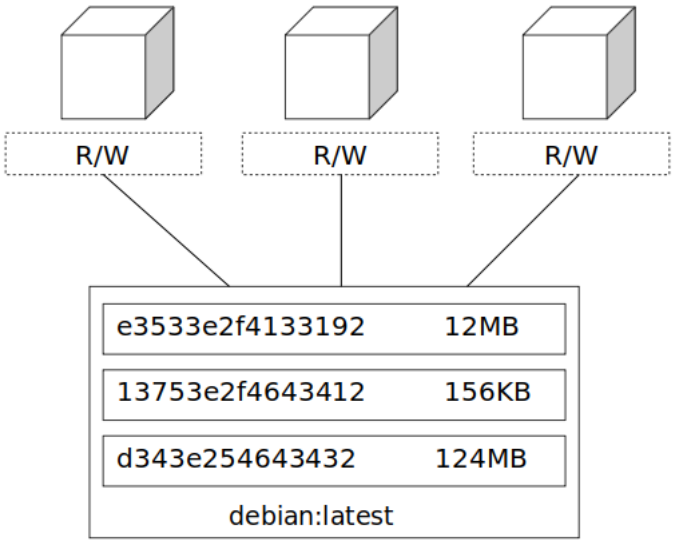


Figura 2.30: Capas compartidas entre varios contenedores

La gestión de las capas es efectuada por un driver especializado (*storage driver*). Docker proporciona distintas alternativas (e.g. `overlay2`, `aufs`, `devicemapper`, `brtfs`, `zfs`, etc.), cada una con sus ventajas e inconvenientes. Cada driver puede utilizar un sistema de ficheros de soporte diferente (e.g. `xfs`, `ext4`, `btrfs`, `zfs`, etc.). Para conocer el driver que se está utilizando, así como el sistema de ficheros de soporte se puede ejecutar el siguiente comando:

```
> docker info
Containers: 9
  Running: 1
  Paused: 0
  Stopped: 8
 Images: 10
 Server Version: 19.03.13
 Storage Driver: overlay2
  Backing Filesystem: extfs
...
```

Cuando se ejecuta un contenedor con el comando `docker container run` (`docker run`) la imagen base del contenedor es automáticamente descargada. Para descargar una imagen de manera explícita se utiliza el comando `docker image pull` (`docker pull`):

```
> docker image pull debian
```

Docker primero comprueba si la imagen está instalada localmente. Si no está instalada, procede a su descarga. Para ello, Docker contacta con el *Docker Hub* y la descarga. Si la imagen se compone de múltiples capas, Docker efectúa este proceso de manera independiente por cada capa. De este modo, si múltiples imágenes comparten una misma capa, ésta se descarga e instala una única vez.

Por ejemplo, la imagen `ubuntu` contiene tres capas que se descargan de manera independiente, tal y como se puede observar en el siguiente ejemplo:

```
> docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
6a5697faee43: Pull complete
ba13d3bc422b: Pull complete
a254829d9e55: Pull complete
Digest: sha256:fff16eea1a8ae92867721d90c59a75652ea66d29c0529...
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

Las capas de sólo lectura que componen una imagen se suelen almacenar en el directorio /var/lib/docker/<storage-driver> (sorprendentemente, los nombres de los subdirectorios no coinciden con los identificadores de las capas).

```
> sudo ls -la /var/lib/docker/overlay2
total 32
drwx------  6 root root 12288 nov 13 10:44 .
drwx--x--x 14 root root  4096 nov 13 06:49 ..
drwx------  4 root root  4096 nov 13 10:44 16d0eb770a9c1960d79112d...
drwx------  3 root root  4096 nov 13 10:44 1bdb24e4ccc600ced2dd1ee...
drwx------  4 root root  4096 nov 13 10:44 b23decd4eed000c5b426880...
drwx------  2 root root  4096 nov 13 10:44 l
```

La capa de lectura/escritura que se crea para cada contenedor se suele almacenar en el directorio /var/lib/docker/containers. En esta ocasión, el nombre del subdirectorio sí coincide con el identificador del contenedor. Si se desea obtener el tamaño de la capa del contenedor (y del conjunto de capas - virtual) también se puede utilizar la opción -s al listar los contenedores con docker container ps -sa (docker ps -sa).

```
> docker run ubuntu
> docker ps -sa
CONTAINER ID   IMAGE    COMMAND    STATUS               SIZE
fe5b83c02e30   ubuntu  "/bin/bash" Exited (0) 54 se...  0B (virtual 72.9MB)
> sudo ls -la /var/lib/docker/containers
total 12
drwx------  3 root root 4096 nov 13 10:59 .
drwx--x--x 14 root root 4096 nov 13 06:49 ..
drwx------  4 root root 4096 nov 13 10:59 fe5b83c02e304a05b81cc54d55aa0e...
```

Para consultar las imágenes instaladas localmente se utiliza el comando docker image ls (docker images). Con este comando, además es posible comprobar su identificador y su dimensión.

```
> docker image ls
REPOSITORY     TAG        IMAGE ID        CREATED       SIZE
ubuntu         latest     d70eaf7277ea    2 weeks ago   72.9MB
```

2.12.4.1. Crear una imagen

Existen dos maneras de crear una nueva imagen en Docker:

- Partir de una imagen padre y modificarla por medio de un contenedor.
- Construir la imagen de manera automática a partir del script `Dockerfile`.

A continuación se procederá a crear una imagen a partir de las modificaciones efectuadas por un contenedor. El siguiente código primero crea un contenedor a partir de la imagen `debian`, creando por tanto una capa de lectura/escritura (la capa del contenedor) sobre dicha imagen. A continuación se modifica el contenido de dicha capa, en este caso añadiendo el fichero `HelloWorld.txt`.

```
> docker run -it debian
Unable to find image 'debian:latest' locally
latest: Pulling from library/debian
e4c3d3e4f7b0: Pull complete
Digest: sha256:8414aa82208bc4c2761dc149df67e25c6b8a9380e5d8c4e7...
Status: Downloaded newer image for debian:latest
root@532ab88e8bf4:/# echo 'Hello world!' > HelloWorld.txt
```

Para obtener un listado de los cambios efectuados en la capa del contenedor se puede utilizar el comando `docker diff <container>`.

```
> docker ps
CONTAINER ID   IMAGE     COMMAND      STATUS
532ab88e8bf4   debian    "bash"       Up 11 minutes
> docker diff 532ab88e8bf4
A /HelloWorld.txt
```

Para crear una nueva imagen a partir de los cambios introducidos en la capa del contenedor se utiliza el comando `docker commit <container> <image-name>`. El flag `-a` especifica el autor de la imagen, y el flag `-m` especifica un mensaje que se quedará registrado junto a la imagen.

```
> docker ps
CONTAINER ID   IMAGE     COMMAND      STATUS
532ab88e8bf4   debian    "bash"       Up 11 minutes
> docker commit 532ab88e8bf4 my-img -a "yo@upv.es" -m "My first image"
sha256:be5672a6eb91e01ac125b1fd8eda205e7e55121a457d7401a9263613...
```

Si se listan las imágenes instaladas en el host se mostrará la nueva imagen creada.

```
> docker image ls
REPOSITORY     TAG            IMAGE ID          SIZE
my-img         latest         be5672a6eb91      114MB
ubuntu         latest         d70eaf7277ea      72.9MB
debian         latest         1510e8501783      114MB
```

A continuación es posible crear contenedores a partir de la nueva imagen, y se puede comprobar que contienen los nuevos cambios introducidos.

```
> docker run -it my-img
root@8c84071cc954:/# ls -la
...
-rw-r--r--   1 root root   13 Nov 14 07:34 HelloWorld.txt
...
```

El comando `docker commit` añade una nueva capa de sólo lectura a una nueva imagen. Además de la fotografía del sistema de ficheros del contenedor, la capa puede contener otra información del contexto de ejecución del contenedor, como sus variables de entorno, el directorio de trabajo, etc. Si esta información no fue definida en el contenedor, entonces se hereda de la imagen original.

Para eliminar imágenes instaladas en el host se utiliza el comando `docker image rm` (`docker rmi`):

```
> docker image rm my-img
```

Esta nueva posibilidad permite construir una imagen de manera interactiva. En el siguiente ejemplo se crea una imagen con Node.js instalado en su interior.

```
> docker run -it debian
root@8c84071cc954:/# apt-get update && apt-get install -y curl
root@8c84071cc954:/# curl -sL https://deb.nodesource.com/setup_14.x|bash -
root@8c84071cc954:/# apt-get install -y nodejs
> docker ps
CONTAINER ID   IMAGE     COMMAND      STATUS
532ab88e8bf5   debian    "bash"       Up 11 minutes
> docker commit 532ab88e8bf5 mynode -a "yo@upv.es" -m "My Node.js image"
```

2.12.4.2. Dockerfile

Docker recomienda construir una imagen de manera automática, leyendo las instrucciones de un fichero de texto, que por defecto se denomina `Dockerfile`. De este modo, la creación de la imagen es determinista, repetible y portable.

Para construir una imagen se utiliza el comando `docker image build <context>` (`docker build`) y se requiere un fichero `Dockerfile` y un contexto. Por defecto, el fichero de comandos `Dockerfile` se encuentra en el directorio actual, aunque es posible modificar este comportamiento con la opción `-f <file>`. El contexto puede ser un directorio o una URL (apuntando a un repositorio git, fichero `.tar`, fichero de texto), y contiene los recursos necesarios para construir la imagen. El siguiente comando construye una imagen a partir del fichero `Dockerfile` disponible en el directorio actual y utilizando el directorio actual como contexto:

```
> docker build .
Sending build context to Docker daemon  6.51 MB
...
```

La construcción de la imagen tiene lugar en el demonio `dockerd`, y no en `docker` CLI. Se trata de un detalle muy importante, porque primero es necesario transferir todo el contexto al demonio `dockerd`. El contexto se transfiere de manera recursiva (directorios, subdirectorios, etc.), de manera que se recomienda incluir únicamente los ficheros estrictamente necesarios. Se pueden omitir ficheros/directorios del contexto incluyendo el fichero `.dockerignore`:

```
# comment
*/temp*
*/*/temp*
temp?
```

Una vez transferido el contexto, el demonio `dockerd` procesa el fichero `Dockerfile`, ejecutando las instrucciones de manera secuencial. La ejecución de cada instrucción tiene lugar en el interior de un contenedor independiente y produce cambios en su capa de lectura/escritura. Tras finalizar la ejecución del contenedor, se confirman los cambios efectuados en una nueva capa de sólo lectura, que formará parte de la imagen. Por lo tanto, la construcción de una nueva imagen puede conllevar la creación de múltiples capas.

Cuando se construye una imagen se le asigna un identificador único. Además, se le pueden asignar nombres o etiquetas (aliases), comúnmente denominados *tags*, que permiten referenciar a la imagen de una manera más cómoda. Para ello, se utiliza la opción `--tag <img-name>:<tag>` (`-t`). La parte `<img-name>` puede tener cualquier formato, pero si se desea almacenar en *Docker Hub* debe seguir la convención `<user>/<name>`. El `<tag>` es opcional; si no se especifica ninguno por defecto se añade `latest`.

```
> docker build -t john/myimg:1.0 -t john/myimg:latest .
```

El fichero `Dockerfile` soporta múltiples comandos, algunos de ellos producen modificaciones en las capas que componen la imagen y otros configuran diversos aspectos de la imagen. Todas las instrucciones siguen la misma sintaxis:

```
# Comment
INSTRUCTION arguments
```

`FROM`

La primera instrucción del fichero Dockerfile debe ser la instrucción `FROM <image>`. Esta instrucción especifica la imagen padre de la imagen que se pretende construir. Se recomienda utilizar las imágenes oficiales de Docker.

```
FROM debian:latest
```

`LABEL`

La instrucción `LABEL` añade metadatos a la imagen. Se trata de pares clave-valor, que registran información en la imagen, y que son heredadas por todas las imágenes hijas. Es la manera habitual de incluir información descriptiva de la imagen, como su versión, su descripción o su autor.

```
FROM debian
LABEL version="1.0"
LABEL description="Ejemplo de imagen"
LABEL maintainer="pepe@upv.es"
```

`RUN`

La instrucción más habitual es `RUN <command>`. Esta instrucción ejecuta cualquier número de comandos en una nueva capa y confirma (`commit`) los resultados. La imagen confirmada será utilizada como base para la siguiente instrucción en `Dockerfile`. Para evitar crear un número elevado de capas, es habitual concatenar múltiples comandos en su interior con `&&`. Es común utilizar esta instrucción para instalar paquetes software (e.g. `apt-get update && apt-get install -y xxx`).

En el siguiente ejemplo se define un fichero `Dockerfile` que construye una imagen con Node.js instalado en su interior.

```
FROM debian:latest
RUN apt-get update && apt-get install -y curl
RUN curl -sL https://deb.nodesource.com/setup_14.x | bash -
RUN apt-get install -y nodejs

> docker build -t mynodejs .
```

```
Sending build context to Docker daemon  3.584kB
Step 1/4 : FROM debian:latest
 ---> 1510e8501783
Step 2/4 : RUN apt-get update && apt-get install -y curl
 ---> Running in f9ccefb53310
...
Removing intermediate container f9ccefb53310
 ---> 76c5cedd98f6
Step 3/4 : RUN curl -sL https://deb.nodesource.com/setup_14.x | bash -
 ---> Running in 0239b2fb8e9f
...
Removing intermediate container 0239b2fb8e9f
 ---> c91773858fc8
Step 4/4 : RUN apt-get install -y nodejs
 ---> Running in 300bf25946b6
...
Removing intermediate container 300bf25946b6
 ---> 0437305f72b7
Successfully built 0437305f72b7
Successfully tagged mynodejs:latest

> docker images
REPOSITORY    TAG            IMAGE ID            CREATED              SIZE
mynodejs      latest         0437305f72b7        About a minute ago   336MB

> docker run -it --rm mynodejs
root@ed750b65a0f7:/# node --version
v14.15.0
```

COPY

La instrucción COPY <src> <dst> copia ficheros del contexto a la imagen. <src> se interpreta relativo a la raíz del contexto mientras que <dst> es una ruta absoluta o relativa al directorio de trabajo actual. Si el fichero/directorio no existe en la imagen se crea automáticamente.

En el siguiente ejemplo se define un fichero Dockerfile que construye una imagen con el runtime PHP, y se copia el contenido del directorio www disponible en el contexto al directorio /www del contenedor. A continuación se construye la imagen y se crea un contenedor a partir de ella que ejecuta un servidor web que sirve el contenido del directorio /www.

```
FROM debian
```

```
COPY www /www
RUN apt-get update && apt-get install -y php-cli

> docker build -t myphp .
...
> docker run -it --rm myphp php -S 0.0.0.0:8080 -t /www
PHP 7.3.19-1~deb10u1 Development Server started at Sun Nov 15 15:56:53 2020
Listening on http://0.0.0.0:8080
Document root is /www
Press Ctrl-C to quit.
```

ADD

La instrucción ADD <src> <dst> copia ficheros o directorios procedentes de <src> a la imagen. <src> puede estar incluido en el contexto, o puede ser una URL que permite añadir contenido desde ubicaciones remotas.

```
FROM debian
ADD http://example.com/last_version.tar /last_version
```

ENTRYPOINT

Existen dos instrucciones que permiten establecer el comando a ejecutar cuando se arranca un contenedor a partir de la imagen: ENTRYPOINT y CMD. La instrucción ENTRYPOINT <command> determina el comando base a ejecutar, y puede ser redefinido con docker CLI con la opción docker run --entrypoint <command> <image>. La instrucción CMD <command> complementa a la anterior, suele utilizarse para definir parámetros por defecto sobre ENTRYPOINT y suele ser redefinida por el usuario por medio del comando docker run <image> <cmd>.

```
FROM debian
COPY www /www
RUN apt-get update && apt-get install -y php-cli
ENTRYPOINT php -S 0.0.0.0:8080 -t /www

> docker build -t myphp .
...
> docker run -d myphp
> docker run --entrypoint php -S 0.0.0.0:8080 -t /www myphp
```

ENV

La instrucción ENV <key>=<value> permite definir variables de entorno que estarán disponibles en el resto de instrucciones en el proceso de construcción de la imagen. Pueden ser accedidas utilizando la sintaxis $VAR_NAME.

Estas variables persistirán en cualquier contenedor creado a partir de la imagen. Son muy útiles para configurar diversos aspectos de los contenedores y pueden ser (re)definidas por el usuario utilizando el comando `docker run --env <key>=<value>`.

`ARG`

La instrucción `ARG <varname>[=<default value>]` permite definir variables disponibles únicamente en el proceso de construcción de la imagen. Pueden ser accedidas utilizando la sintaxis `$VAR_NAME`.

Estas variables no estarán disponibles en los contenedores que se crean a partir de la imagen. Pueden ser (re)definidas por el usuario utilizando la opción `docker build --build-arg <varname>=<value>`.

`EXPOSE`

La instrucción `EXPOSE <port>/<protocol>` informa acerca de los puertos en los que el contenedor escuchará. Esta instrucción no mapea el puerto, simplemente informa acerca de los puertos expuestos. Posteriormente, al arrancar un contenedor, es posible mapear cada puerto de manera individual con la opción `-p <host>:<container>`, o bien todos los puertos expuestos con la opción `-P`.

```
FROM debian
EXPOSE 80/tcp
EXPOSE 80/udp
```

`VOLUME`

La instrucción `VOLUME` permite especificar la creación de un volumen, y en qué lugar del contenedor se montará. Cuando se arranca el contenedor, el volumen se crea de manera automática. En el siguiente ejemplo se define un volumen que se montará en el path `/vol1`, se construye la imagen y se crea un contenedor a partir de ella. Se puede comprobar la creación del volumen.

```
FROM debian
VOLUME /vol1
```

```
> docker build -t myvol .
Sending build context to Docker daemon  3.584kB
Step 1/2 : FROM debian
 ---> 1510e8501783
Step 2/2 : VOLUME /vol1
 ---> Running in 45b65854ea89
Removing intermediate container 45b65854ea89
 ---> dbb21e889706
Successfully built dbb21e889706
```

```
Successfully tagged test:latest
> docker run -it myvol
root@05eb0f690eca:/# ls /
bin  boot  dev etc  ...  sys  tmp  usr  var  vol1
> docker volume ls
DRIVER    VOLUME NAME
local     1f91a2a23185e5b9fea16cd0773f449e6f245643fcae0ebc6d0d1e3d9fedc176
```

Si el contenedor finaliza su ejecución, es posible comprobar que el volumen sigue existiendo. Si el contenedor se ejecuta con la opción `--rm`, cuando finaliza su ejecución, también se elimina el volumen.

2.12.4.3. Repositorio de imágenes

Existen tres maneras de obtener imágenes en Docker:

- A través de Docker Hub.
- A partir de un fichero.
- A través de un Docker Registry.

Por defecto, cuando se solicita una imagen, Docker contacta con el *Docker Hub*. Se trata de un servicio mantenido por Docker que contiene gran cantidad de imágenes. El `Docker Hub` ofrece dos tipos de imágenes:

- Imágenes oficiales, proporcionadas y mantenidas por Docker. Estas imágenes poseen un identificador sencillo (e.g. `debian`, `node`, etc.). En general, se trata de imágenes de sistemas operativos base (e.g. `debian`, `ubuntu`, etc.), que sirven como punto de partida para crear imágenes más complejas, o bien imágenes que contienen entornos de ejecución populares (e.g. `node`, `python`, etc.), u otro tipo de servicios de utilidad (e.g. `mongo`, `redis`, `mysql`, etc.).
- Imágenes de usuario, creadas y gestionadas por usuarios/compañías. Poseen un identificador del tipo `user/image-name`. Estas imágenes pueden contener cualquier tipo de entorno de ejecución. El *Docker Hub* ofrece la posibilidad de registrar organizaciones, equipos, usuarios, etc.

También es posible transferir las imágenes a partir de ficheros. En el host donde está la imagen instalada se puede empaquetar en un fichero con el comando `docker image save -o <file> <image>` (`docker save`).

```
> docker image save -o debian.tar debian
```

Para instalar dicha imagen en otro host se utiliza el comando `docker image load -i <file>` (`docker load`):

```
> docker image load -i debian.tar
```

Por último, Docker proporciona una implementación open-source de un servicio que permite almacenar y distribuir imágenes, denominado el *Docker Registry*. El *Docker Registry* permite a una compañía disponer de su propio repositorio de imágenes.

Este servicio se distribuye como un contenedor más, de manera que para utilizarlo es necesario disponer de una instalación Docker. El siguiente comando arranca el registro, y publica su puerto 5000 en el host.

```
> docker run -d -p 5000:5000 --name registry registry:2
```

Los usuarios pueden interactuar con el registro utilizando los comandos `docker pull/push`. Para comprender cómo trabajar con el registro primero es necesario revisar cómo se nombran las imágenes.

Cada imagen posee un identificador único. También es posible referirse a ella a través de nombres, etiquetas o tags. Existen tres maneras de asignar tags a una imagen:

- Cuando se crea una nueva imagen de manera interactiva utilizando el comando `docker commit <image> <img-name>:<tag>`.
- Cuando se construye una imagen utilizando el comando `docker build -t <img-name>:<tag> <path>`.
- De manera explícita utilizando el comando `docker tag <image> <img-name>:<tag>`.

El formato de `<img-name>` puede ser cualquiera, pero generalmente refleja su origen. Por ejemplo, la imagen `debian` en realidad es una versión abreviada de `docker.io/library/debian`, que identifica el host `docker.io` en el que se encuentra la imagen - el *Docker Hub* -, y su nombre `library/debian`. En el *Docker Hub* el nombre de una imagen debe seguir el formato `<repository>/<name>`, donde `<repository>` suele representar a un usuario. Por lo tanto, con carácter general, se puede decir que el nombre de una imagen sigue el patrón `<registry-host>:<port>/<repository>/<name>`.

Gracias a esta convención, cuando se emiten los comandos `docker pull/push` Docker sabe con qué registro debe contactar. Con el siguiente comando Docker contacta con el *Docker Hub*, y descarga la imagen `debian:latest`.

```
> docker pull debian
```

A continuación se añadirá una nueva etiqueta a la misma imagen usando `docker tag`, indicando que debería almacenarse en otro registro. Después se enviará la imagen a dicho registro utilizando `docker push`. Después se eliminará la imagen con `docker rmi` y se procederá a descargarla desde el nuevo registro con `docker pull`.

```
> docker tag debian:latest localhost:5000/mydebian
> docker images
REPOSITORY                TAG           IMAGE ID
debian                    latest        1510e8501783
localhost:5000/mydebian   latest        1510e8501783  s
...
> docker push localhost:5000/mydebian
The push refers to repository [localhost:5000/mydebian]
9780f6d83e45: Pushed
latest: digest: sha256:60cb30babcd1740309903c37d3d408407... size: 529
> docker rmi localhost:5000/mydebian:latest
Untagged: localhost:5000/mydebian:latest
Untagged: localhost:5000/mydebian@sha256:60cb30babcd1740309903c37d3d408407d19
> docker pull localhost:5000/mydebian
Using default tag: latest
latest: Pulling from mydebian
Digest: sha256:60cb30babcd1740309903c37d3d408407d190cf73015aeddec9086ef3f393a
Status: Downloaded newer image for localhost:5000/mydebian:latest
localhost:5000/mydebian:latest
```

2.13. Orquestación de contenedores

Las arquitecturas basadas en microservicios poseen una complejidad muy elevada. En tiempo de ejecución se componen de un gran número de componentes interconectados ejecutándose de manera concurrente. Gestionar de manera coherente el arranque de todos estos componentes, su interconexión, replicación, monitorización, detección y recuperación ante fallos sólo es posible por medio de herramientas automatizadas.

Si los microservicios se implementan por medio de contenedores, estas herramientas reciben el nombre de orquestadores de contenedores. Existen diversas alternativas que ofrecen diferentes niveles de servicio, desde soluciones capaces de construir, ejecutar e interconectar contenedores de manera automática (e.g.

Docker Compose) hasta potentes frameworks (e.g. *Kubernetes*) que recubren un clúster de máquinas y que además de desplegar configuraciones de manera automática, son capaces de garantizar su calidad de servicio, replicando, arrancando y parando contenedores en diversas máquinas, conforme resulte necesario, y todo ello sin intervención humana.

2.14. Docker Compose

Docker Compose es una herramienta que permite definir y ejecutar aplicaciones con múltiples contenedores en una misma máquina. Con Compose todos los servicios de la aplicación se definen en un fichero `.yaml`. Después se utiliza este fichero para ejecutar la aplicación. Compose permite automatizar la creación e interconexión de múltiples contenedores.

En esencia, Compose permite automatizar todos los comandos que se deberían ejecutar sobre la herramienta CLI `docker` para poner en marcha una aplicación, a saber: construir las imágenes de la aplicación, crear los volúmenes y redes, arrancar los contenedores e interconectarlos.

El primer paso para empezar a trabajar con la herramienta consiste en instalarla. Cada sistema operativo dispone de sus propias instrucciones. Una vez instalado, trabajar con Compose es muy sencillo, y el proceso consta de tres pasos básicos:

1. Definir las imágenes de los distintos componentes de la aplicación, con los correspondientes `Dockerfile`.

2. Definir la aplicación con todos sus servicios en el fichero `docker-compose.yml`.

3. Ejecutar el comando `docker-compose up` para arrancar la aplicación.

2.14.1. docker-compose.yml

El fichero `docker-compose.yml` contiene toda la configuración de la aplicación, e incluye tres secciones fundamentales:

- `services`: incluye los contenedores que formarán parte de la aplicación, y que se crearían de manera manual utilizando el comando `docker run`.

- `volumes`: incluye los volúmenes que se crearán para almacenar la información de los contenedores, y que se crearían de manera manual utilizando el comando `docker volume create`.

- `networks`: incluye las redes que interconectarán los distintos contenedores, y que se crearían de manera manual utilizando el comando `docker network create`.

El fichero `docker-compose.yml` contiene cuatro entradas fundamentales, `version`, `services`, `volumes` y `networks`, y verifica el siguiente esquema:

```
version: "3.8"
services:
  service1: ...
  service2: ...
volumes:
  vol1: ...
  vol2: ...
networks:
  net1: ...
  net2: ...
```

Una vez creado el fichero, se puede arrancar la aplicación de manera automática utilizando el comando:

```
> docker-compose up
```

La sección `services` contiene una entrada por cada contenedor que se desea crear, e incluye toda la configuración que se aplicará a dicho contenedor al arrancarlo. Por defecto, el contenedor recibe el nombre del servicio. Existen opciones para configurar literalmente cualquier aspecto del contenedor. Por defecto, el contenedor recibe el nombre del servicio. A continuación se listan algunas de las más comunes. Para obtener un listado completo se recomienda consultar la referencia oficial.

La opción `image` determina la imagen base del contenedor.

```
services:
  webapp:
    image: debian
```

Con la opción `build` se pueden especificar distintas opciones para construir de manera automática la imagen en la que se basará el contenedor. La opción más sencilla es especificar el contexto a partir del cual se creará la imagen.

```
services:
  webapp:
    build: ./dir
```

La opción `command` se utiliza para definir el comando que se ejecutará cuando el contenedor arranque. Esta opción sobrescribe la configuración `CMD` definida en el fichero `Dockerfile`.

```
services:
  webapp:
    build: ./webapp
    command:
      - 'npm run serve'
```

La opción `environment` se utiliza para definir variables de entorno.

```
services:
  webapp:
    image: debian
    environment:
      PUBLISH_URL: '/webapp'
      SSL: true
```

La opción `ports` mapea los puertos del contenedor al host, utilizando la sintaxis `HOST_PORT:CONTAINER_PORT`.

```
services:
  webapp:
    image: debian
    ports:
      - "8080:80"
      - "10022:22"
```

La opción `volumes` determina los volúmenes o puntos de montaje del host que se montarán en el contenedor. Los volúmenes disponibles para los servicios se especifican en la sección raíz `volumes`.

```
services:
  db:
    image: db
    volumes:
      - /var/lib/log              # crea un nuevo volumen automáticamente
      - data-volume:/var/lib/db   # monta un volumen con nombre
  backup:
    image: backup-service
    volumes:
```

```
      - /opt/data:/var/lib/mysql  # monta un directorio del host

volumes:
  data-volume:
```

Por defecto, Compose crea una única red a la que conecta todos los contenedores. Esta red recibe el mismo nombre que el directorio raíz en el que se encuentra el fichero `docker-compose.yml`. En esta red, los contenedores son alcanzables entre sí por sus nombres. La opción `networks` determina las redes a las que se conectará el servicio. Estas redes se deben definir en la sección raíz `networks` del fichero `docker-compose.yml`.

```
services:
  webapp:
    image: debian
    networks:
      - net1
      - net2

networks:
  net1:
    net2:
```

En ocasiones, unos servicios dependen de otros, y es importante el orden en el que se arrancan. Para establecer estas dependencias se utiliza la opción `depends_on`.

```
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```

2.14.2. CLI

Una vez definida la configuración de la aplicación en el fichero `docker-compose.yml`, se utiliza la herramienta CLI `docker-compose` para gestionarla. Esta herramienta posee numerosos comandos.

```
>  docker-compose
 ...
build            Build or rebuild services
config           Validate and view the Compose file
create           Create services
down             Stop and remove containers, networks, images, and volumes
events           Receive real time events from containers
exec             Execute a command in a running container
help             Get help on a command
images           List images
kill             Kill containers
logs             View output from containers
pause            Pause services
port             Print the public port for a port binding
ps               List containers
pull             Pull service images
push             Push service images
restart          Restart services
rm               Remove stopped containers
run              Run a one-off command
scale            Set number of containers for a service
start            Start services
stop             Stop services
top              Display the running processes
unpause          Unpause services
up               Create and start containers
version          Show version information and quit
```

Sin lugar a dudas, los comandos más relevantes, que permiten arrancar y parar una aplicación, son, respectivamente:

```
> docker-compose up
> docker-compose down
```

El primer comando construye, crea y arranca todos los contenedores definidos en el fichero `docker-compose.yml`. Se pueden crear distintos entornos aislados, definiendo para ello un nombre de proyecto, bien con el parámetro `-p <project_name>`, bien definiendo la variable de entorno `COMPOSE_PROJECT_NAME`. En la práctica, esto significa que el mismo `docker-compose.yml` puede ser usado para arrancar múltiples aplicaciones independientes que se ejecutan en paralelo sin interferencias.

Compose trata de ensamblar la aplicación de manera inteligente, cacheando contenedores y preservando el contenido de los volúmenes creados. Para ello, al arrancar una aplicación, se comprueba si los contenedores/volúmenes se crearon anteriormente, y en tal caso se reutilizan. En caso contrario, se crean nuevos.

Referencias

[Erl, 2007] Erl, T. (2007). *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, USA.

[Erl, 2016] Erl, T. (2016). *Service-Oriented Architecture: Analysis and Design for Services and Microservices*. Prentice Hall Press, USA, 2nd edition.

[Erl et al., 2012] Erl, T., Carlyle, B., Pautasso, C., and Balasubramanian, R. (2012). *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*. Prentice Hall Press, USA, 1st edition.

[Fielding, 2000] Fielding, R. T. (2000). Rest: architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California*.

[Goldberg, 1973] Goldberg, R. P. (1973). Architectural principles for virtual computer systems. Technical report, HARVARD UNIV CAMBRIDGE MA DIV OF ENGINEERING AND APPLIED PHYSICS.

[Popek and Goldberg, 1974] Popek, G. J. and Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421.