5 Common Server Setups For Your Web Application

By: Mitchell Anicas

https://www.digitalocean.com/community/tutorials/5-common-server-setups-for-your-web-application

Introduction

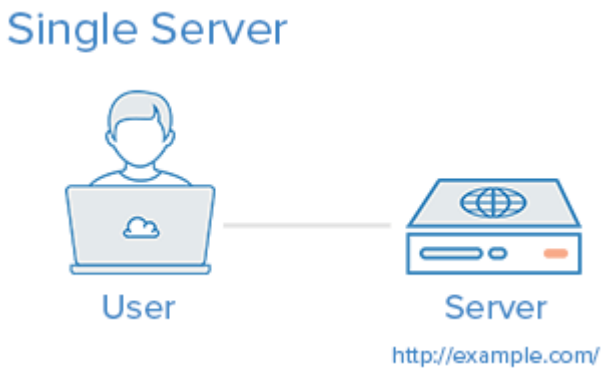
When deciding which server architecture to use for your environment, there are many factors to consider, such as performance, scalability, availability, reliability, cost, and ease of management.

Here is a list of commonly used server setups, with a short description of each, including pros and cons. Keep in mind that all of the concepts covered here can be used in various combinations with one another, and that every environment has different requirements, so there is no single, correct configuration.

1. Everything On One Server

The entire environment resides on a single server. For a typical web application, that would include the web server, application server, and database server. A common variation of this setup is a LAMP stack, which stands for Linux, Apache, MySQL, and PHP, on a single server.

Use Case: Good for setting up an application quickly, as it is the simplest setup possible, but it offers little in the way of scalability and component isolation.



Pros:

- Simple

Cons:

- Application and database contend for the same server resources (CPU, Memory, I/O, etc.) which, aside from possible poor performance, can make it difficult to determine the source (application or database) of poor performance
- Not readily horizontally scalable

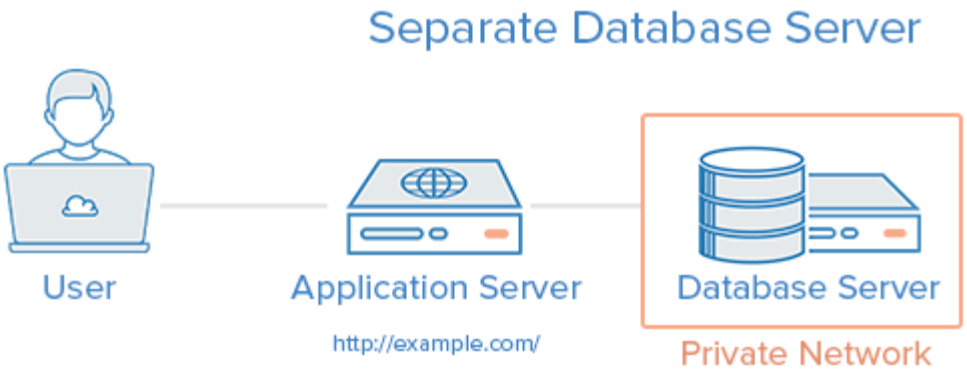
Related Tutorials:

- How To Install LAMP On Ubuntu 14.04

2. Separate Database Server

The database management system (DBMS) can be separated from the rest of the environment to eliminate the resource contention between the application and the database, and to increase security by removing the database from the DMZ, or public internet.

Use Case: Good for setting up an application quickly, but keeps application and database from fighting over the same system resources.



Pros:

- Application and database tiers do not contend for the same server resources (CPU, Memory, I/O, etc.)
- You may vertically scale each tier separately, by adding more resources to whichever server needs increased capacity
- Depending on your setup, it may increase security by removing your database from the DMZ

Cons:

- Slightly more complex setup than single server
- Performance issues can arise if the network connection between the two servers is high-latency (i.e. the servers are geographically distant from each other), or the bandwidth is too low for the amount of data being transferred

Related Tutorials:

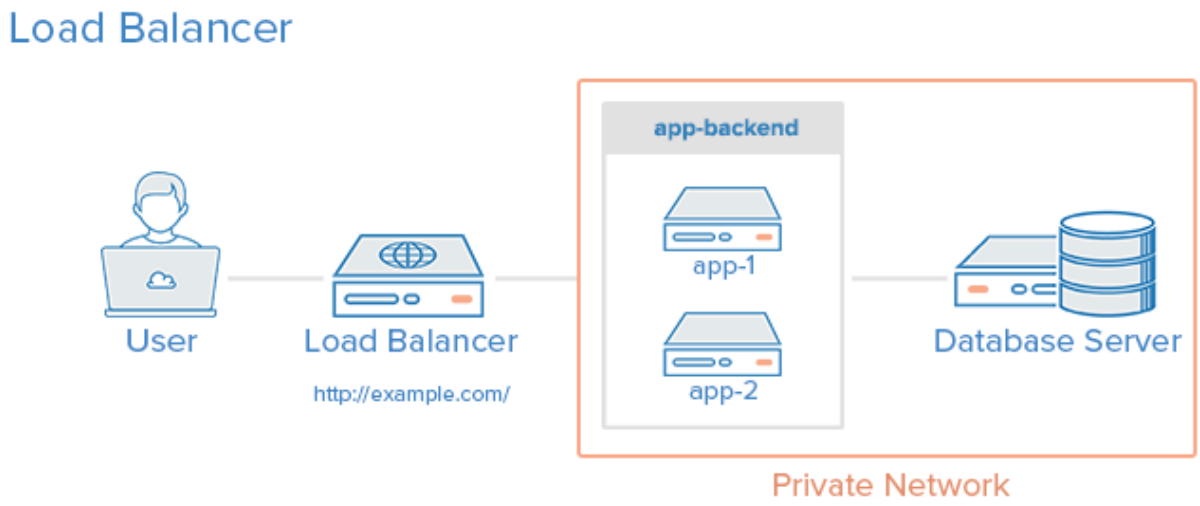
- How To Set Up a Remote Database to Optimize Site Performance with MySQL
- How to Migrate A MySQL Database To A New Server On Ubuntu 14.04

3. Load Balancer (Reverse Proxy)

Load balancers can be added to a server environment to improve performance and reliability by distributing the workload across multiple servers. If one of the servers that is load balanced fails, the other servers will handle the incoming traffic until the failed server becomes healthy again. It can also be used to serve multiple applications through the same domain and port, by using a layer 7 (application layer) reverse proxy.

Examples of software capable of reverse proxy load balancing: HAProxy, Nginx, and Varnish.

Use Case: Useful in an environment that requires scaling by adding more servers, also known as horizontal scaling.



Pros:

- Enables horizontal scaling, i.e. environment capacity can be scaled by adding more servers to it
- Can protect against DDOS attacks by limiting client connections to a sensible amount and frequency

Cons:

- The load balancer can become a performance bottleneck if it does not have enough resources, or if it is configured poorly
- Can introduce complexities that require additional consideration, such as where to perform SSL termination and how to handle applications that require sticky sessions
- The load balancer is a single point of failure; if it goes down, your whole service can go down. A *high availability* (HA) setup is an infrastructure without a single point of failure. To learn how to implement an HA setup, you can read this section of How To Use Floating IPs.

Related Tutorials:

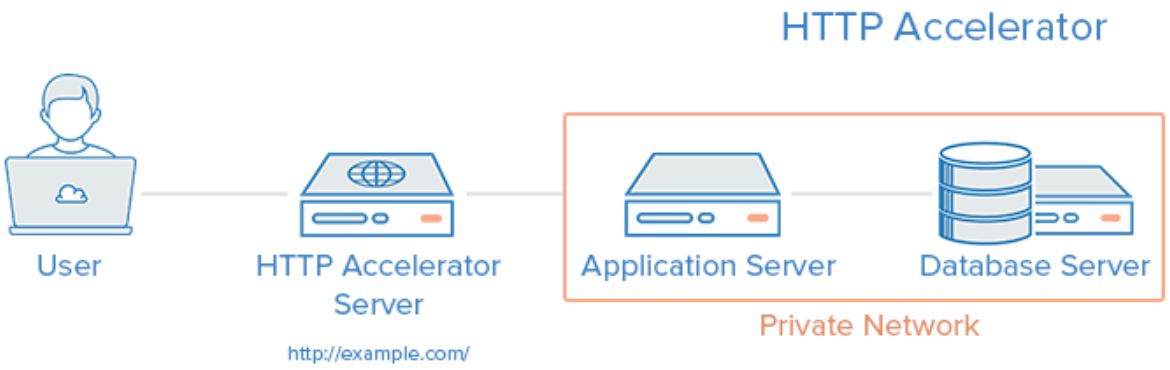
- An Introduction to HAProxy and Load Balancing Concepts
- How To Use HAProxy As A Layer 4 Load Balancer for WordPress Application Servers
- How To Use HAProxy As A Layer 7 Load Balancer For WordPress and Nginx

4. HTTP Accelerator (Caching Reverse Proxy)

An HTTP accelerator, or caching HTTP reverse proxy, can be used to reduce the time it takes to serve content to a user through a variety of techniques. The main technique employed with an HTTP accelerator is caching responses from a web or application server in memory, so future requests for the same content can be served quickly, with less unnecessary interaction with the web or application servers.

Examples of software capable of HTTP acceleration: Varnish, Squid, Nginx.

Use Case: Useful in an environment with content-heavy dynamic web applications, or with many commonly accessed files.



Pros:

- Increase site performance by reducing CPU load on web server, through caching and compression, thereby increasing user capacity
- Can be used as a reverse proxy load balancer
- Some caching software can protect against DDOS attacks

Cons:

- Requires tuning to get best performance out of it
- If the cache-hit rate is low, it could reduce performance

Related Tutorials:

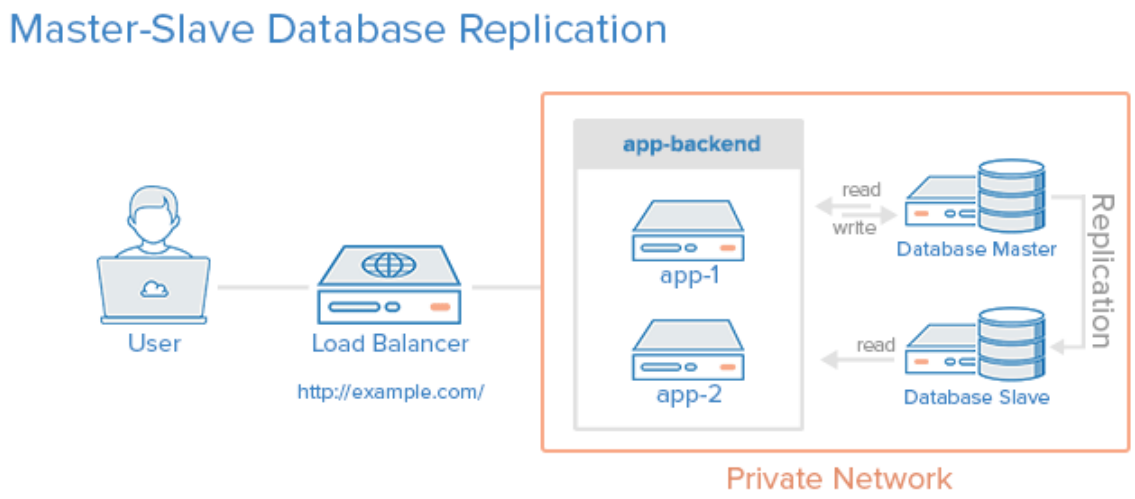
- How To Install Wordpress, Nginx, PHP, and Varnish on Ubuntu 12.04
- How To Configure a Clustered Web Server with Varnish and Nginx
- How To Configure Varnish for Drupal with Apache on Debian and Ubuntu

5. Master-Slave Database Replication

One way to improve performance of a database system that performs many reads compared to writes, such as a CMS, is to use master-slave database replication. Master-slave replication requires a master and one or more slave nodes. In this setup, all updates are sent to the master node and reads can be distributed across all nodes.

Use Case: Good for increasing the read performance for the database tier of an application.

Here is an example of a master-slave replication setup, with a single slave node:



Pros:

- Improves database read performance by spreading reads across slaves
- Can improve write performance by using master exclusively for updates (it spends no time serving read requests)

Cons:

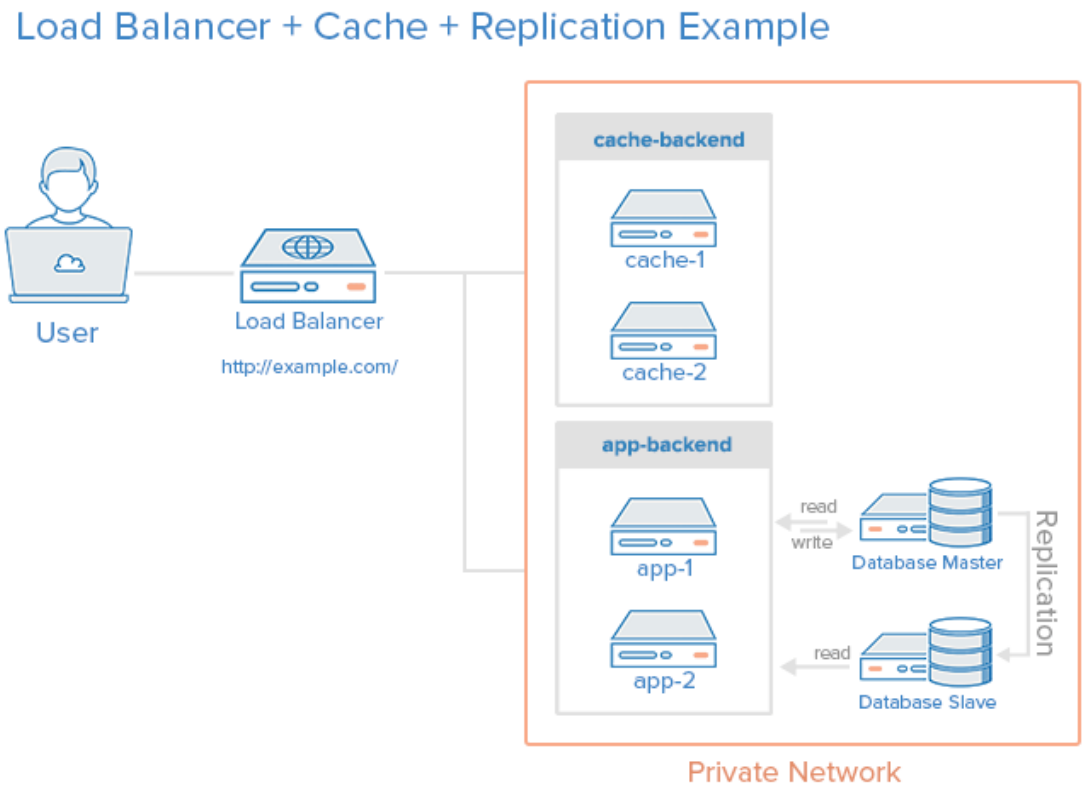
- The application accessing the database must have a mechanism to determine which database nodes it should send update and read requests to
- Updates to slaves are asynchronous, so there is a chance that their contents could be out of date
- If the master fails, no updates can be performed on the database until the issue is corrected
- Does not have built-in failover in case of failure of master node

Related Tutorials:

- How To Optimize WordPress Performance With MySQL Replication On Ubuntu 14.04
- How To Set Up Master Slave Replication in MySQL

Example: Combining the Concepts

It is possible to load balance the caching servers, in addition to the application servers, and use database replication in a single environment. The purpose of combining these techniques is to reap the benefits of each without introducing too many issues or complexity. Here is an example diagram of what a server environment could look like:



Let's assume that the load balancer is configured to recognize static requests (like images, css, javascript, etc.) and send those requests directly to the caching servers, and send other requests to the application servers.

Here is a description of what would happen when a user sends a requests dynamic content:

1. The user requests dynamic content from *http://example.com/* (load balancer)
2. The load balancer sends request to app-backend
3. app-backend reads from the database and returns requested content to load balancer
4. The load balancer returns requested data to the user

If the user requests static content:

1. The load balancer checks cache-backend to see if the requested content is cached (cache-hit) or not (cache-miss)
2. *If cache-hit*: return the requested content to the load balancer and jump to Step 7. *If cache-miss*: the cache server forwards the request to app-backend, through the load balancer
3. The load balancer forwards the request through to app-backend
4. app-backend reads from the database then returns requested content to the load balancer
5. The load balancer forwards the response to cache-backend
6. cache-backend *caches the content* then returns it to the load balancer
7. The load balancer returns requested data to the user

This environment still has two single points of failure (load balancer and master database server), but it provides the all of the other reliability and performance benefits that were described in each section above.

Conclusion

Now that you are familiar with some basic server setups, you should have a good idea of what kind of setup you would use for your own application(s). If you are working on improving your own environment, remember that an iterative process is best to avoid introducing too many complexities too quickly.

Let us know of any setups you recommend or would like to learn more about in the comments below!

By Mitchell Anicas