Common Server Setups For Your Web Application - Part II

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.

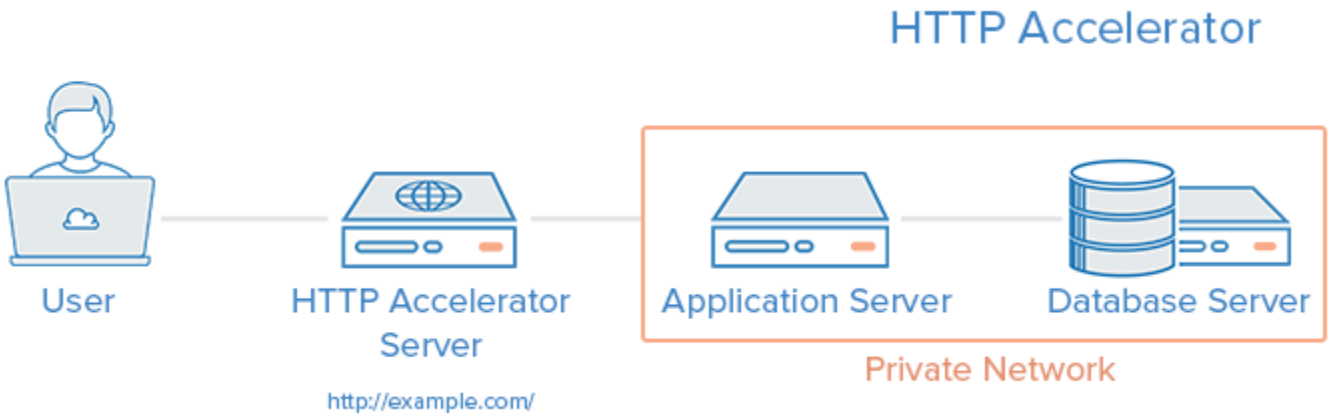
Here is a list of commonly used server setups, with a short description of each, including pros and cons. Keep in mind that all of the concepts covered here can be used in various combinations with one another, and that every environment has different requirements, so there is no single, correct configuration.

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:

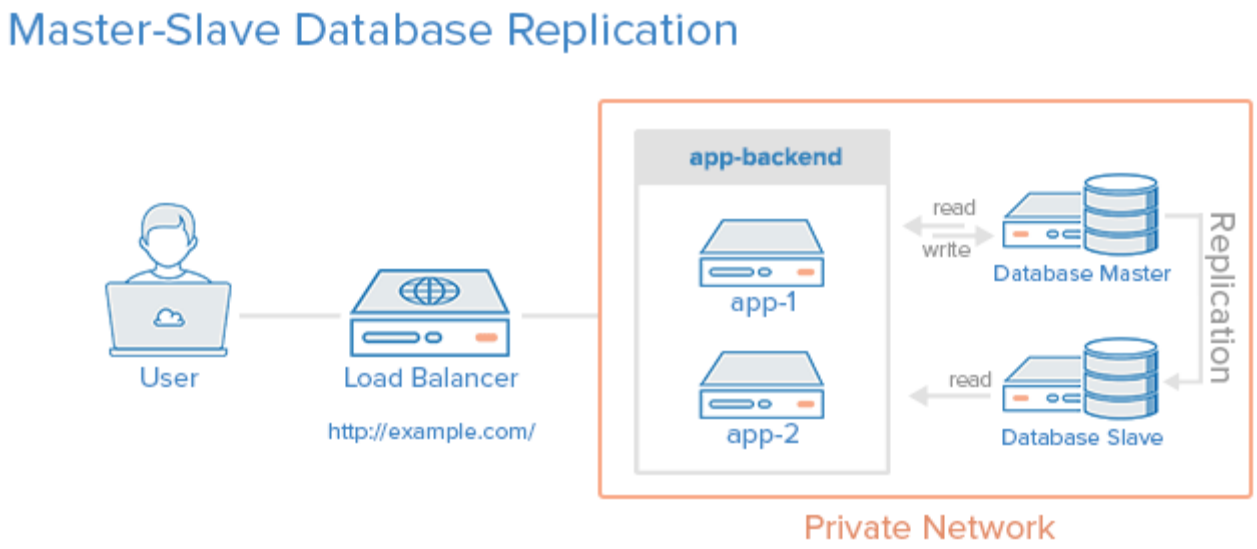
- Requires tuning to get best performance out of it
- If the cache-hit rate is low, it could reduce performance

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)

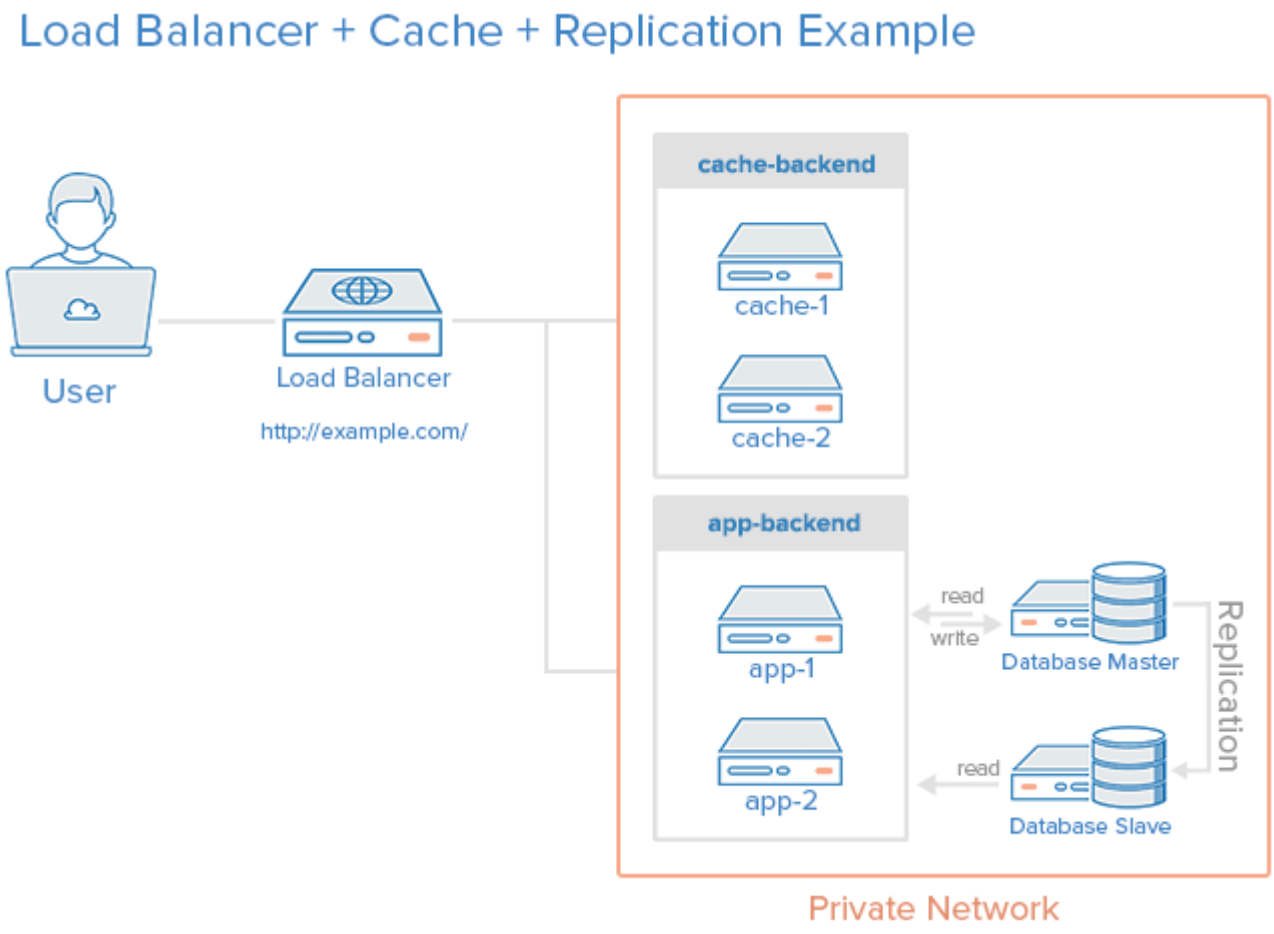
Cons:

- 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

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.

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