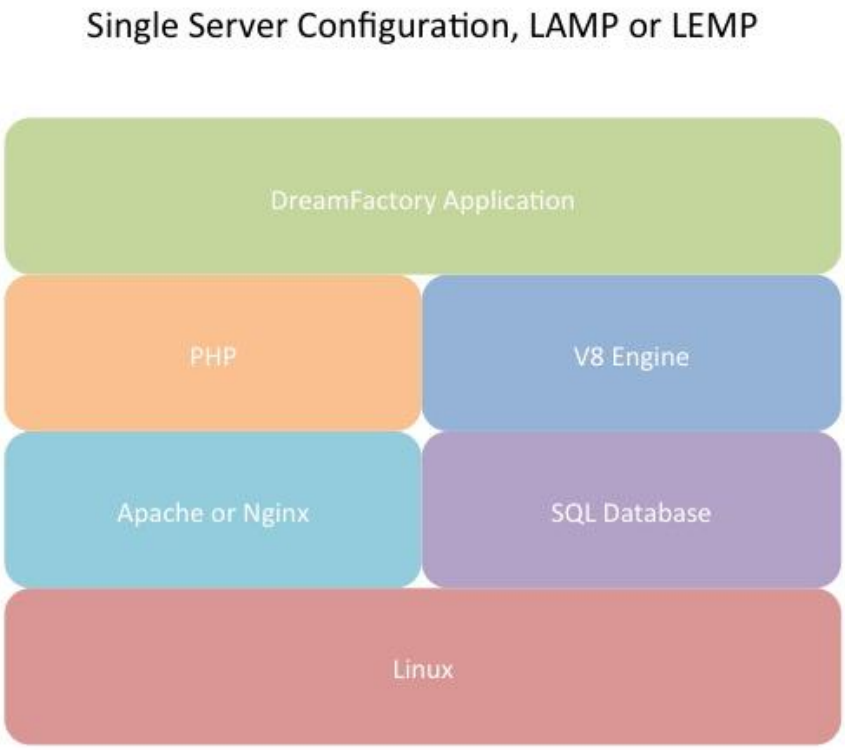
Scaling DreamFactory

This white paper is designed to provide information to enterprise customers about how to scale a DreamFactory Instance. The sections below talk about horizontal, vertical, and cloud scaling capabilities.

Before we dive into the details, the most important thing to know is that DreamFactory is a configurable LAMP stack running PHP. As far as the server is concerned, DreamFactory looks like a website written in WordPress or Drupal. Instead of delivering HTML pages, DreamFactory delivers JSON documents, but otherwise the scaling requirements are similar.

Instead of using traditional session management, where the server maintains the state of the application, DreamFactory handles session management in a stateless manner, not requiring the server to maintain any application state. This makes horizontal scaling a breeze, as you'll see below. For demanding deployments, we suggest using NGINX, more on that later.



This is important because you can apply all the standard things you already know about scaling simple websites directly to scaling DreamFactory. This is not an accident. It makes DreamFactory easy to install on any server and easy to scale for massive deployments of mobile applications and Internet of Things (IoT) devices.

Vertical Scaling

You can vertically scale DreamFactory on a single server through the addition of extra processing power, extra memory, better network connectivity, and more hard disk space. This section discusses how vertical scaling and server configuration can impact performance.

By increasing server processor speed, number of processors, and RAM, you can improve the performance of the DreamFactory engine. Processor speed will especially improve round-trip response times. In our testing, DreamFactory can usually respond to a single service request in 100 to 200 milliseconds.

The other important characteristic is the number of simultaneous requests that DreamFactory can handle. On a single server with vertical scaling, this will depend on processor speed and available RAM to support multiple processes running at the same time. Network throughput is important for both round-trip time and handling a large number of simultaneous transactions.

Default SQL Database

Each DreamFactory instance has a default SQL database that is used to store information about users, roles, services, and other objects required to run the platform. The default Bitnami installation package includes a default SQL database, but you can also hook up any other database for this purpose. When this database is brought online, DreamFactory will create the additional system tables that are needed to manage the platform.

DreamFactory also stores server-side scripts in this default database. These scripts can be written in JavaScript or PHP to customize either the request or response of the API calls running through the engine. DreamFactory uses the V8 engine to execute JavaScript. This allows developers to use JavaScript both on the client and on the server and call the API from either side of the stack. The V8 engine is included in the Bitnami installers and must exist for server-side scripting to work.

Developers can also create tables on the default database for their own projects. Based on application requirements, mobile projects can query this database in various ways, and this activity can impact performance. The DreamFactory user records are also stored in the default database. Anything that you do to boost the performance of this database will increase the speed of the admin console and developer applications.

Local File Storage

Each DreamFactory instance also needs some file storage for HTML5 web application hosting. Each application is given a folder where the developer might store HTML files, graphic images, CSS style sheets, JavaScript files, etc. Native applications might store other documents or resources in local storage for simple download. The size and access speed of the local file system will impact application performance just like a normal web site.

Persistent Storage

By default, DreamFactory uses persistent local storage for two things: system-wide cache data and hosted application files and resources. Many of the Platform as a Service (PaaS) systems such as Pivotal, Bluemix, Heroku, and OpenShift do not support persistent local storage.

For these systems, you need to configure DreamFactory to use memory-based cache storage such as Memcached or Redis. You also need to create a remote cloud storage service such as S3, Azure Blob, Rackspace etc. to store your application files. You can easily configure DreamFactory to use a memory-based cache via the config files.

The database for PaaS needs to be a remote SQL database like ClearDB or whatever the vendor recommends. If you use the local file system to create files at runtime these will disappear when the PaaS image is restarted or when multiple instances are scaled. Working with PaaS is discussed in greater detail under the cloud scaling section, below.

External Data Sources

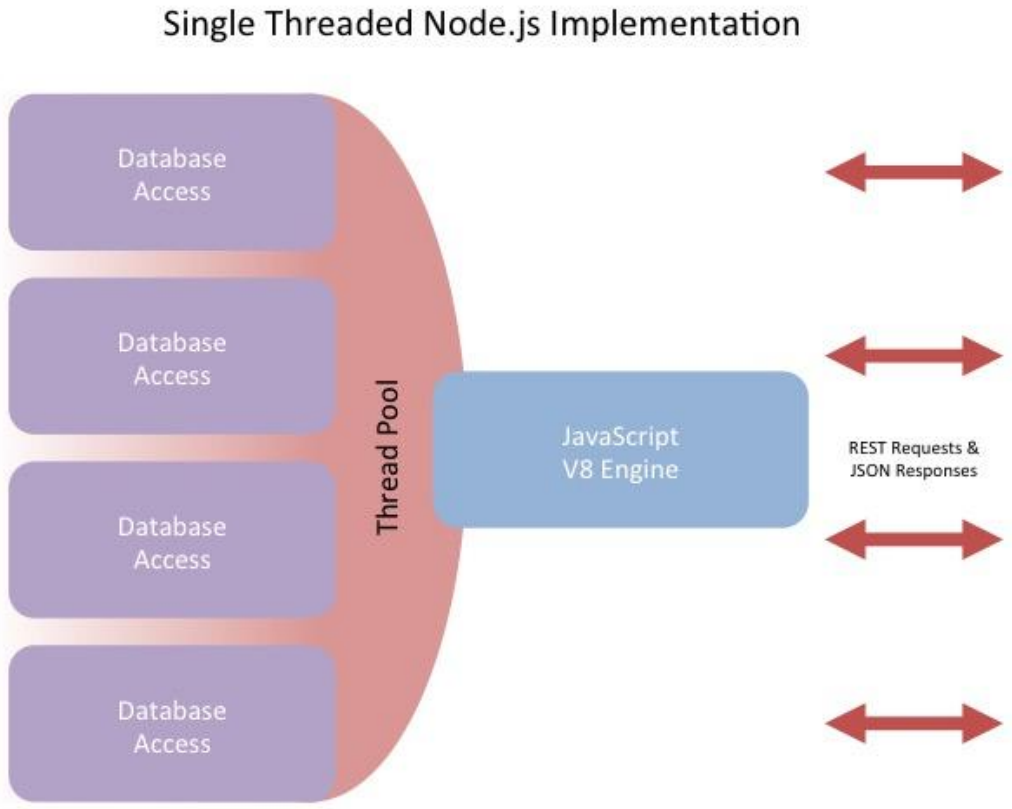
You can hook up any number of external data sources to DreamFactory. DreamFactory currently supports MySQL, PostgreSQL, Oracle, SQL Server, DB2, S3, MongoDB, DynamoDB, CouchDB, Cloudant, and more. Some of the NoSQL databases are specifically designed for massive scalability on clustered hardware. You can hook up any SQL database running in your data center in order to mobilize legacy data. You can also hook up cloud databases like DynamoDB and Azure Tables.

DreamFactory acts as a proxy for these external systems. DreamFactory will inherit the performance characteristics of the database, with some additional overhead for each REST API call. DreamFactory adds a security layer, a customization layer, normalizes the web services, and implements role-based access control for each service. The scalability of these external data sources will depend on service level agreements with your cloud vendor, the hardware behind database clustering, and other factors.

DreamFactory vs. Node.js

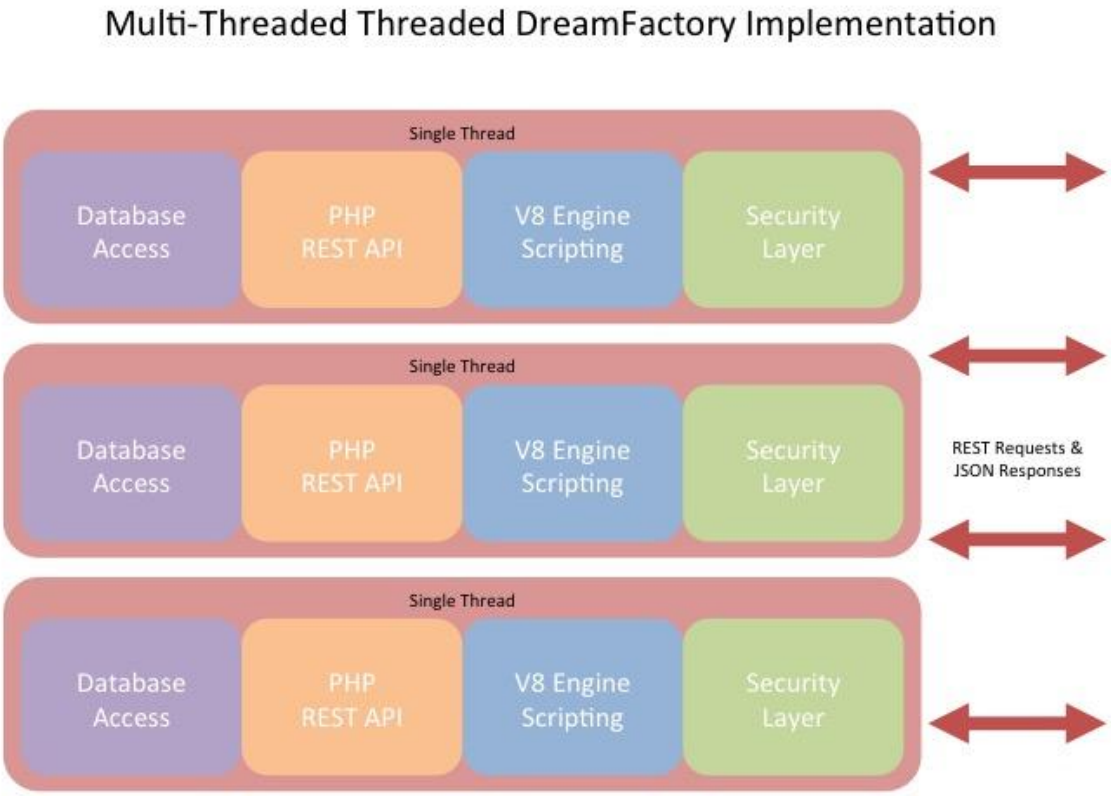
I'm going to take a small detour here and discuss some of the differences between DreamFactory and Node.js. This is helpful background information in order to understand how DreamFactory can be scaled horizontally with multiple servers, load balancers, and clustered databases.

We considered using Node.js for the DreamFactory engine, but were concerned that a single thread would be insufficient to support a massively scalable mobile deployment. The workload in a sophisticated REST API platform is quite comparable to an HTML website written in Drupal or WordPress where multiple threads are required to process all the data.



Another issue was the need for mature interfaces to a wide variety of SQL and NoSQL databases. This was a challenge with Node.js. Instead, we chose PHP because this language is in widespread use and has great frameworks such as Laravel. The main thing we liked about Node.js was the V8 engine. This allows developers to write JavaScript on the client and on the server. DreamFactory harnesses the power of the V8 engine by using the V8Js extension for PHP, except that DreamFactory runs it in parallel for scalability. The V8 engine is also sandboxed for security.

On an Apache server running DreamFactory, we use Prefork MPM to create a new child process with one thread for each connection. You need to be sure that the MaxClients configuration directive is big enough to handle as many simultaneous requests as you expect to receive, but small enough to ensure enough physical RAM for all processes.

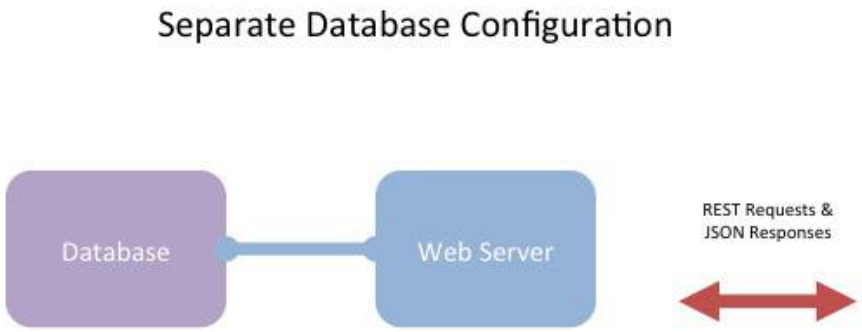


There is a danger that you will have more incoming requests than the server can handle. In this case, DreamFactory will issue an exponential backoff message telling the client to try again later. DreamFactory Enterprise offers additional methods of limiting calls per second. But still, the total number of transactions will be limited. Node.js can potentially handle a very large number of simultaneous requests with event-based callbacks, but in that situation you are stuck with a single thread for all of the data processing. In this situation, Node.js becomes a processing bottleneck for every REST API call.

If you expect a massive number of incoming requests, then consider running DreamFactory on an NGINX server with PHP-FPM instead of Apache. NGINX can maximize the requests per second that the hardware can handle, and reduce the memory required for each connection. This is a “best of both worlds” scenario that allows a conventional web server to handle massive transaction volumes without the processing bottleneck of Node.js.

Horizontal Scaling

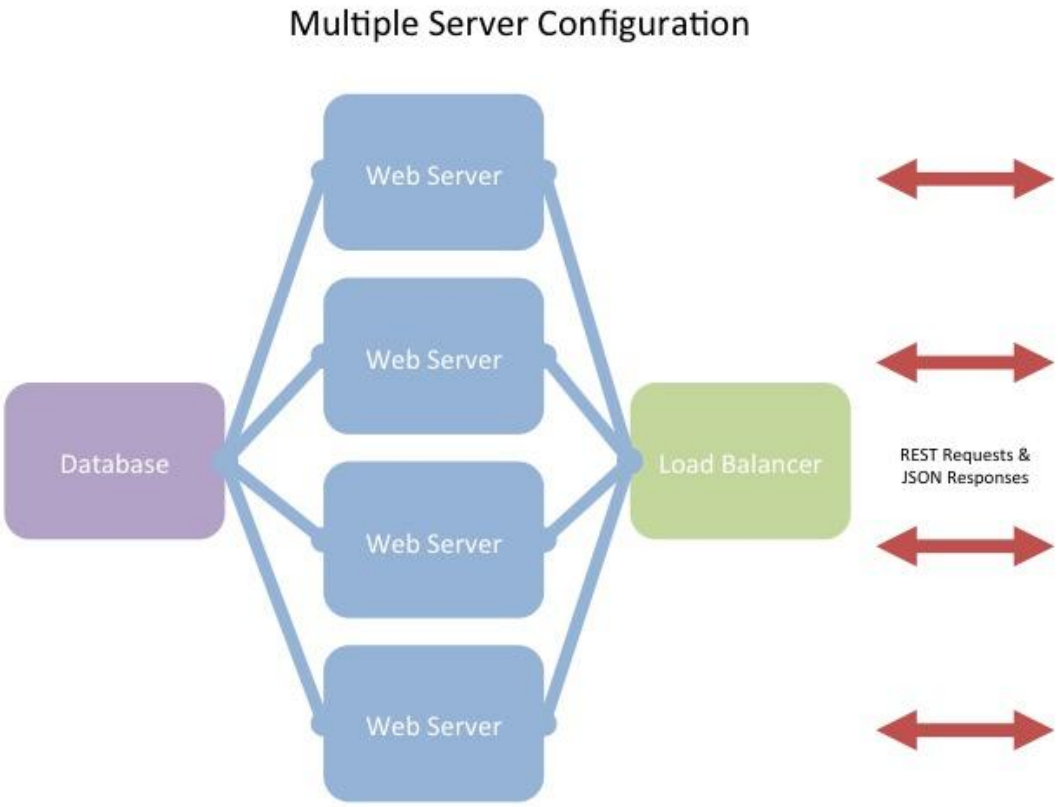
This section discusses ways to use multiple servers to increase performance. The simplest model is just to run DreamFactory on a single server. When you do a Bitnami install, DreamFactory runs in a LAMP stack with the default SQL database and some local file storage. The next step up is to configure a separate server for the default SQL database. There are also SQL databases that are available as a hosted cloud service.



Multiple Servers

You can use a load balancer to distribute REST API requests among multiple servers. A load balancer can also perform health checks and remove an unhealthy server from the pool automatically. Most large server architectures include load balancers at several points throughout the infrastructure. You can cluster load balancers to avoid a single point of failure. DreamFactory is specifically designed to work with load balancers and all of the various scheduling algorithms.

DreamFactory uses JWT (JSON Web Token) to handle user authentication and session in a completely stateless manner. Therefore, a REST API request can be sent to any one of the web servers at any time without the need to maintain user session/state across multiple servers. Each REST API call to a DreamFactory Instance can pass JWT in the request header, the URL query string, or in the request payload. The token makes the request completely aware of its own state, eliminating the need to maintain state on the server.



Shared Local Storage

All of the web servers need to share access to the same local file storage system. In DreamFactory Version 1.9 and below, you will need a shared "storage" drive mounted with NFS or something similar. DreamFactory Version 2.0 and higher supports a more configurable local file system. The Laravel PHP Config File specifies a driver for retrieving files and this can be on a local drive, NFS, SSHFS, Dropbox, S3, etc. This simplifies multiple server setup and also PaaS delivery options.

Multiple Databases

The default SQL database can be enhanced in various ways. You can mirror the database, create database clusters for enhanced performance, and utilize failover clusters for high-availability installations. A full discussion of this topic is beyond the scope of this paper. Some additional resource links are provided below.

Performance Benchmarks

Below are some results that show the vertical scalability of a single DreamFactory Instance calculated with Apache Benchmark. Five different Amazon Web Services EC2 instances were tested. The servers were t2.small, t2.medium, m4.xlarge, m4.2xlarge, and finally m4.4xlarge.

Vertical Scaling Benchmarks

For this test, we conducted 1000 GET operations from the DreamFactory REST API. There were 100 concurrent users making the requests. Each operation searched, sorted, and retrieved 1000 records from a SQL database. This test was designed to exercise the server side processing required for a complex REST API call.

Complex GET: 1000 records searched, sorted, and retrieved 1000 times					
Server	Small	Medium	Large	X Large	XX Large
CPUs	1	2	4	8	16
Memory (GB)	2	4	16	32	64
Total Time (Sec)	31.94	19.29	12.88	6.93	3.56
Avg Requests / Second	31.31	51.84	77.63	144.40	280.76
Avg Time / Request (MS)	3194.20	1928.87	1288.14	692.52	356.17

Looking at the three m4 servers, we see a nice doubling of capacity that matches the extra processors and memory. This really shows the vertical scalability of a single DreamFactory instance. The complex GET scenario highlights the advantages of the additional processor power.

Next, we tried a similar test with a simple GET command that basically just returned a single database record 5000 times. There were 100 concurrent users making the requests. In this situation, the fixed costs of Internet bandwidth, network switching, and file storage start to take over, and the additional processors contribute less.

Regular GET: 1 record retrieved 5000 times					
Server	Small	Medium	Large	X Large	XX Large
CPUs	1	2	4	8	16
Memory (GB)	2	4	16	32	64
Total Time (Sec)	105.94	62.72	41.52	22.03	15.78
Avg Requests / Second	47.20	79.72	120.42	226.93	316.79
Avg Time / Request (MS)	2118.79	1254.33	830.40	440.66	315.66

Look at these results for 5000 simple GETs from the API. As you can see, performance does not fully double with additional processors. This demonstrates the diminishing returns of adding processors without scaling up other fixed assets.

By the way, we also looked at POST and DELETE transactions. The results were pretty much what you would expect and in line with the GET requests tested above.

Horizontal Scaling Benchmarks

Below are some results that show the horizontal scalability of a single DreamFactory Instance calculated with Apache Benchmark. Four m4.xlarge Amazon Web Services EC2 instances were configured behind a load balancer. The servers were configured with a common default database and EBS storage.

Complex GET: 1000 calls	
Server	4 Large
CPUs	16
Memory (GB)	64
Total Time (Sec)	3.26
Avg Requests / Second	292.42
Avg Time / Request (MS)	326.61

Regular GET: 5000 calls	
Server	4 Large
CPUs	16
Memory (GB)	64
Total Time (Sec)	10.69
Avg Requests / Second	433.01
Avg Time / Request (MS)	213.87

First we tested the complex GET scenario. The load balanced m4.xlarge servers ran at about the same speed as the m4.4xlarge server tested earlier. This makes sense because each setup had similar CPU and memory installed. Since this example was bound by processing requirements, there was not much advantage to horizontal scaling.

Next we tested the simple GET scenario. In this case there appears to be some advantage to horizontal scaling. This is probably due to better network IO and the relaxation of other fixed constraints compared to the vertical scalability test.

Concurrent User Benchmarks

We also evaluated the effects of concurrent users simultaneously calling REST API services on the platform. This test used the complex GET scenario where 1000 records were searched, sorted, and retrieved. The test was conducted with three different Amazon Web Services EC2 instances. The servers were m4.xlarge, m4.2xlarge, and m4.4xlarge. We started with 20 concurrent users and scaled up to 240 simultaneous requests.

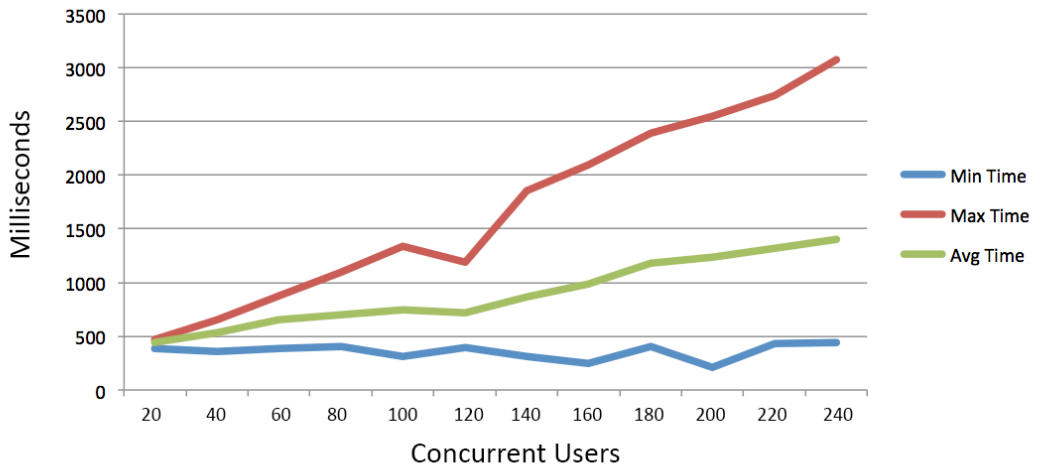
The minimum time for the first requests to finish was always around 300 milliseconds. This is because some requests are executed immediately and finish first while others must wait to be executed.

The maximum time for the last request to finish will usually increase with the total number of concurrent users. Based on the processor size, the maximum time for the last request can increase sharply past some critical threshold. This is illustrated by the 8 processor example, where maximum request times spike past 160 concurrent users.

The 16 processor server never experienced any degradation of performance all the way to 240 concurrent users. This is the maximum number of concurrent users supported by the Apache Bench test program. Even then, the worst round trip delay was less than ½ second. Imagine a real world scenario with 10,000 people logged into a mobile application. If 10% of them made a service request at the same time, you would expect a round trip delay of ½ second on average and a full second in the worst case.

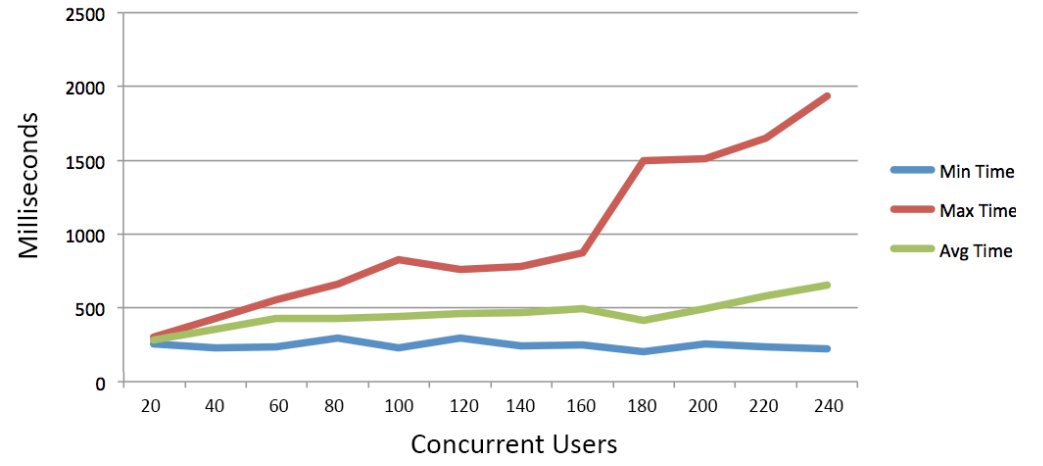
Benchmarks For Concurrent Users, 4 Processors

Complex GET: 1000 records searched, sorted, and retrieved												
Concurrent	20	40	60	80	100	120	140	160	180	200	220	240
Min Time / Request (MS)	385	354	387	403	309	394	310	245	407	211	435	441
Max Time / Request (MS)	470	658	877	1102	1338	1193	1854	2096	2390	2550	2742	3078
Avg Time / Request (MS)	443	536	656	702	749	716	867	988	1178	1232	1322	1405



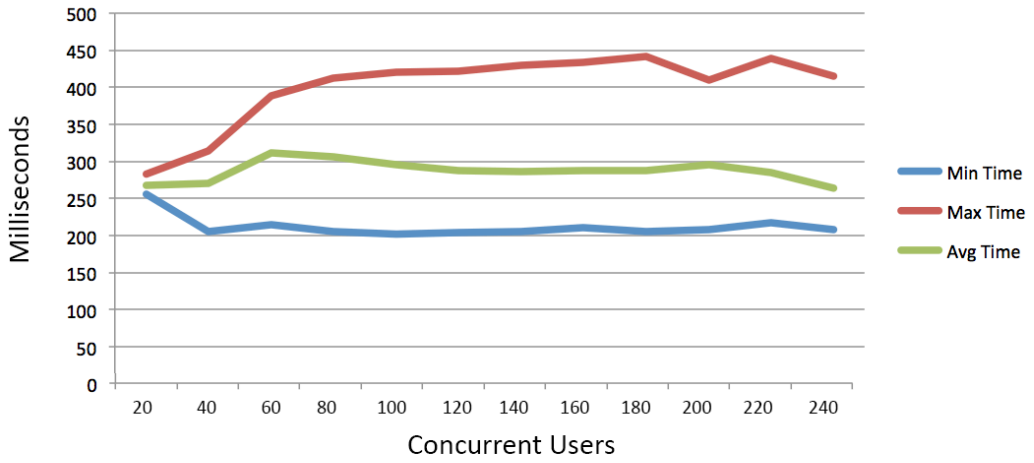
Benchmarks For Concurrent Users, 8 Processors

Complex GET: 1000 records searched, sorted, and retrieved												
Concurrent	20	40	60	80	100	120	140	160	180	200	220	240
Min Time / Request (MS)	253	225	233	296	228	294	243	251	202	255	237	223
Max Time / Request (MS)	300	430	554	657	829	758	781	871	1495	1507	1647	1933
Avg Time / Request (MS)	281	355	427	426	439	463	469	496	417	492	578	656



Benchmarks For Concurrent Users, 16 Processors

Complex GET: 1000 records searched, sorted, and retrieved												
Concurrent	20	40	60	80	100	120	140	160	180	200	220	240
Min Time / Request (MS)	256	206	215	205	202	204	205	211	206	208	217	208
Max Time / Request (MS)	283	314	388	412	421	422	430	433	441	410	439	415
Avg Time / Request (MS)	268	270	311	306	296	288	287	288	288	295	285	264



Your Mileage May Vary

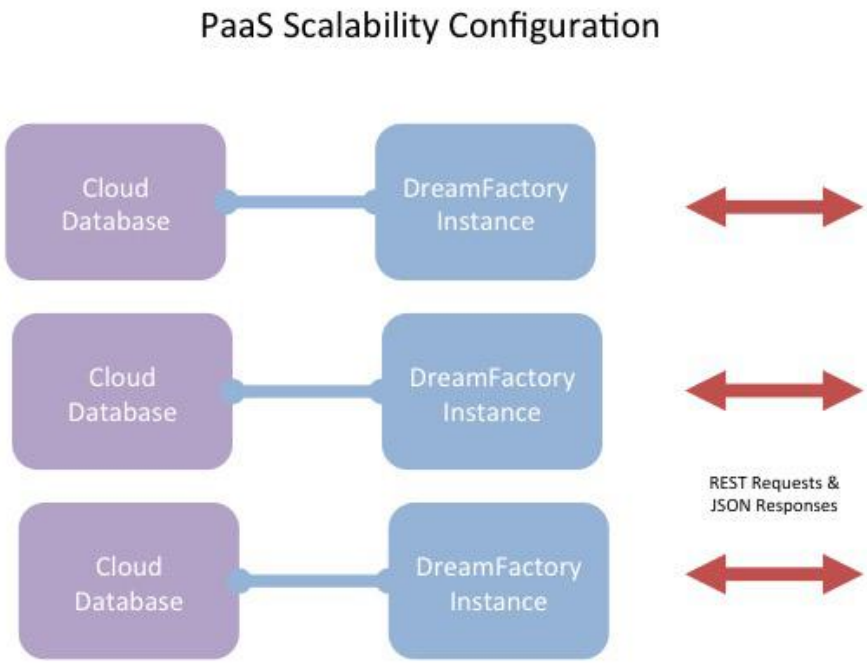
For your implementation, we recommend getting a handle on the time required to complete an average service call. This could depend on database speed, server side scripting, network bandwidth, and other factors. Next, experiment with a few server configurations to see where the limits are. Then scale the implementation to the desired performance characteristics for your application.

In all of my benchmarking tests, there were never any unexplained delays or other performance characteristics that did not respond in a scalable manner. The addition of horizontal or vertical hardware will scale DreamFactory 2.0 in a linear fashion for any requirements that you may have.

Cloud Scaling

Most of the Infrastructure as a Service (IaaS) vendors have systems that can scale web servers automatically. For example, Amazon Web Services can scale EC2 instances with Auto Scaling Groups and Elastic Load Balancers. Auto scaling is built into Microsoft Azure and Rackspace as well. If you want to deploy in the cloud, then check with your vendor for the options they support.

We discussed Platform as a Service (PaaS) deployment options earlier. These systems do not support persistent local file storage, but the trade-off is that your application instance is highly scalable. You can simply specify the maximum number of instances that you would like to run. As traffic increases, additional instances are brought online. If a server stops responding, then the instance is simply restarted.



In Conclusion

DreamFactory is designed to be scaled like a simple website. DreamFactory supports the standard practices for scaling up with additional server capabilities and out with additional servers. DreamFactory has installers or installation instructions for all major IaaS and PaaS clouds, and some of these vendors automatically handle scaling for you.

Refer to the resources below to help you get the most out of your DreamFactory implementation:

Scaling High Traffic Drupal Websites

Comparing Apache with NGINX

Automatic Scaling on Microsoft Azure

Automatic Scaling on Amazon Web Services

Automatic Scaling on Rackspace

Installing DreamFactory on NGINX

Laravel Documentation for Shared File Storage

Database Mirroring and Failover Cluster Instances