**Load balancing** refers to efficiently distributing incoming network traffic across a group of backend servers, also known as a *server farm* or *server pool*.

Modern high-traffic websites must serve hundreds of thousands, if not millions, of concurrent requests from users or clients and return the correct text, images, video, or application data, all in a fast and reliable manner. To cost-effectively scale to meet these high volumes, modern computing best practice generally requires adding more servers.

A [load balancer](https://www.nginx.com/solutions/load-balancing/) acts as the “traffic cop” sitting in front of your servers and routing client requests across all servers capable of fulfilling those requests in a manner that maximizes speed and capacity utilization and ensures that no one server is overworked, which could degrade performance. If a single server goes down, the load balancer redirects traffic to the remaining online servers. When a new server is added to the server group, the load balancer automatically starts to send requests to it.

In this manner, a load balancer performs the following functions:

* Distributes client requests or network load efficiently across multiple servers
* Ensures high availability and reliability by sending requests only to servers that are online
* Provides the flexibility to add or subtract servers as demand dictates

Load Balancing Algorithms

Different load balancing algorithms provide different benefits; the choice of load balancing method depends on your needs:

* Round Robin – Requests are distributed across the group of servers sequentially.
* Least Connections – A new request is sent to the server with the fewest current connections to clients. The relative computing capacity of each server is factored into determining which one has the least connections.
* IP Hash – The IP address of the client is used to determine which server receives the request.

Session Persistence

Information about a user’s session is often stored locally in the browser. For example, in a shopping cart application the items in a user’s cart might be stored at the browser level until the user is ready to purchase them. Changing which server receives requests from that client in the middle of the shopping session can cause performance issues or outright transaction failure. In such cases, it is essential that all requests from a client are sent to the same server for the duration of the session. This is known as *session persistence*.

The best [load balancers](https://www.nginx.com/solutions/load-balancing/) can handle session persistence as needed. Another use case for session persistence is when an upstream server stores information requested by a user in its cache to boost performance. Switching servers would cause that information to be fetched for the second time, creating performance inefficiencies.

Dynamic Configuration of Server Groups

Many fast-changing applications require new servers to be added or taken down on a constant basis. This is common in environments such as the Amazon [Elastic Compute Cloud](https://www.nginx.com/products/nginx-plus-aws/) (EC2), which enables users to pay only for the computing capacity they actually use, while at the same time ensuring that capacity scales up in response traffic spikes. In such environments it greatly helps if the load balancer can dynamically add or remove servers from the group without interrupting existing connections.

Hardware vs. Software Load Balancing

Load balancers typically come in two flavors: hardware-based and software-based. Vendors of hardware-based solutions load proprietary software onto the machine they provide, which often uses specialized processors. To cope with increasing traffic at your website, you have to buy more or bigger machines from the vendor. Software solutions generally run on commodity hardware, making them less expensive and more flexible. You can install the software on the hardware of your choice or in cloud environments like AWS EC2.

For more information about load balancing, see [NGINX Load Balancing](https://www.nginx.com/resources/admin-guide/load-balancer) in the NGINX Plus Admin Guide.

How Can NGINX Plus Help?

[NGINX Plus](https://www.nginx.com/products/) and [NGINX](http://nginx.org/en) are the best-in-class load balancing solutions used by high-traffic websites such as Dropbox, Netflix, and Zynga. More than [185 million websites](https://news.netcraft.com/archives/2016/09/19/september-2016-web-server-survey.html) worldwide, including [half of the 10,000 busiest websites](http://w3techs.com/technologies/cross/web_server/ranking), rely on NGINX Plus and NGINX to deliver their content quickly, reliably, and securely.

As a software-based load balancer, NGINX Plus is much less expensive than hardware-based solutions with similar capabilities. The comprehensive load balancing capabilities in NGINX Plus enable you to build a highly optimized application delivery network.

Clustering versus Sticky Sessions

For web applications, **clustering** is a load-balancing technique in which multiple application servers are set up to behave as one big server. Generally this requires replicating HttpSession data across the servers, to ensure that a user's web interactions will continue without interruption regardless of which server handles the next request. Session replication achieves very high reliability, but it incurs an extra performance cost (due to the serializing and deserializing of session data and the extra network traffic required).

In contrast, **Sticky Sessions** (also called *session persistence* or *sticky persistence*) is a load balancing technique in which each session is assigned to a particular server for the duration of the session. This approach doesn't require copying HTTPSession data between servers, so it's very scalable. But if a server goes down, all of its sessions are lost.

In general, the sticky sessions approach is the way to go when possible (that is, when performance is more important than session survival). It represents a much more efficient use of resources ... you are scaling *out* not *up*, which is always cheaper. It also means that you don't have to be as careful about what goes into the HTTPSession.

*For details on setting up clustering and sticky sessions, see the documentation of whatever load balancer you are using.*

Clustering

Tapestry is designed to be "a good citizen" of an application server that supports clustering. It is careful about what it writes into the HttpSession. The framework understands that the server that receives a request may not be the same one that rendered the page initially; this knowledge affects many code paths, and it guides the approach Tapestry takes to caching page and component properties.

Your part is to properly manage the objects put into the HttpSession (via @SessionAttribute, @SessionState or @Persist; see [Session Storage](http://tapestry.apache.org/session-storage.html)):

* Don't store anything in the session that you don't have to. Principally this means minimizing the use of @Persist (see [Page Activation](http://tapestry.apache.org/page-navigation.html) and [Using Select With a List](http://tapestry.apache.org/using-select-with-a-list.html)), storing only IDs in the session rather than whole entities.
* Where possible, persist only objects that are immutable (i.e., String, or a primitive or wrapper type).
* Only put *serializable* objects into the session.
* Make use of the @ImmutableSessionPersistedObject annotation and OptimizedSessionPersistedObject interface (both described below).

Again, Tapestry is a good citizen, but from the application server's point of view, it's just another servlet application. The heavy lifting here is application server specific.

Clustering Issues

The Servlet API was designed with the intention that there would be only a modest amount of server-side state, and that the stored values would be individual numbers and strings, and thus, immutable.

However, many web applications do not use the HttpSession this way, instead storing large, mutable objects in the session. This is not a problem for single servers, but in a cluster, anything stored in the session must be serialized to a bytestream and distributed to other servers within the cluster, and restored there.

Most application servers perform that serialization and distribution whenever HttpSession.setAttribute() is called. This creates a data consistency problem for mutable objects, because if you read a mutable session object, change its state, but *don't* invoke setAttribute(), the changes will be isolated to just a single server in the cluster.

Tapestry attempts to solve this: any session-persisted object that is read during a request will be re-stored back into the HttpSession at the end of the request. This ensures that changed internal state of those mutable objects is properly replicated around the cluster.

But while this solution solves the data consistency problem, it does so at the expense of performance, since all of those calls to setAttribute() result in extra session data being replicated needlessly if the internal state of the mutable object hasn't changed.

Tapestry has solutions to this, too:

@ImmutableSessionPersistedObject Annotation

Tapestry knows that Java's String, Number and Boolean classes are immutable. Immutable objects do not require a re-store into the session.

You can mark your own session objects as immutable (and thus not requiring session replication) using the [ImmutableSessionPersistedObject](http://tapestry.apache.org/current/apidocs/org/apache/tapestry5/annotations/ImmutableSessionPersistedObject.html)annotation.

OptimizedSessionPersistedObject Interface

The [OptimizedSessionPersistedObject](http://tapestry.apache.org/current/apidocs/org/apache/tapestry5/OptimizedSessionPersistedObject) interface allows an object to control this behavior. An object with this interface can track when its mutable state changes. Typically, you should extend from the [BaseOptimizedSessionPersistedObject](http://tapestry.apache.org/current/apidocs/org/apache/tapestry5/BaseOptimizedSessionPersistedObject.html) base class.

SessionPersistedObjectAnalyzer Service

The [SessionPersistedObjectAnalyzer](http://tapestry.apache.org/current/apidocs/org/apache/tapestry5/services/SessionPersistedObjectAnalyzer.html) service is ultimately responsible for determining whether a session persisted object is dirty or not (dirty meaning in need of a restore into the session). This is an extensible service where new strategies, for new classes, can be introduced.