Nginx Essentials

Excel in Nginx quickly by learning to use its most essential features in real-life applications

Valery Kholodkov
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Proxying and Caching'
- A synopsis of the book’s content
- More information on Nginx Essentials
About the Author

Valery Kholodkov is a seasoned IT professional with a decade of experience in creating, building, scaling, and maintaining industrial-grade web services, web applications, and mobile application backends. Throughout his career, he has worked for well-known brands, such as Yandex, Booking.com, and AVG. He currently works for his own consultancy firm. Valery has a deep understanding of technology and is able to express its essence, advantages, and risks to a layman, which makes him an accomplished author of technology books.
Preface

2006 was an exciting year. The disappointment that surrounded the dot-com crash had pretty much been superseded by a renewed and more confident growth of Web 2.0 and inspired a search for technologies of a new age.

At that time, I was looking for a web server to power my projects that would do many things in a different way. After getting some experience in large-scale online projects, I knew that the popular LAMP stack was suboptimal and sometimes did not solve certain challenges, such as efficient uploads, geo-dependent rate limiting, and so on.

After trying and rejecting a number of options, I came to know about Nginx and immediately felt that my search was over. It is small yet powerful, with a clean code base, good extensibility, relevant set of features, and a number of architectural challenges solved. Nginx definitely stood out from the crowd!

I immediately got inspired and felt some affinity to this project. I tried participating in the Nginx community, learned, shared my knowledge, and contributed as much as I could.

With time, my knowledge of Nginx grew. I started to get consultancy requests and have been capable of addressing quite sophisticated cases. After some time, I realized that some of my knowledge might be worth sharing with everyone. That's how I started a blog at www.nginxguts.com.

A blog turned out to be an author-driven medium. A more reader-focused and more thorough medium was in demand, so I set aside some time to assemble my knowledge in the more solid form of a book. That's how the book you're holding in your hands right now came into existence.
What this book covers

Chapter 1, Getting Started with Nginx, gives you the most basic knowledge about Nginx, including how to carry out the very basic installation and get Nginx up and running quickly. A detailed explanation of the structure of the configuration file is given so that you know where exactly code snippets from the rest of the book apply.

Chapter 2, Managing Nginx, explains how to manage an operating Nginx instance(s).

Chapter 3, Proxying and Caching, explains how to turn Nginx into a powerful web proxy and cache.

Chapter 4, Rewrite Engine and Access Control, explains how to use the rewrite engine to manipulate URLs and secure your web resources.

Chapter 5, Managing Inbound and Outbound Traffic, describes how to apply various restrictions to inbound traffic, and how to use and manage upstream.

Chapter 6, Performance Tuning, explains how to squeeze the most out of your Nginx server.
Proxying and Caching

Designed as a web accelerator and a frontend server, Nginx has powerful tools to delegate complex tasks to upstream servers while focusing on heavy lifting. Reverse proxy is one such tool that turns Nginx into an essential component of any high-performance web service.

By abstracting away complexities of HTTP and handling them in a scalable and efficient manner, Nginx allows web applications to focus on solving the problem they are designed to solve without stumbling upon low-level details.

In this chapter, you will learn:

• How to set up Nginx as a reverse proxy
• How to make proxying transparent for the upstream server and the end user
• How to handle upstream errors
• How to use Nginx cache

You will find out how to use all features of Nginx reverse proxy and turn it into a powerful tool for accelerating and scaling your web service.

Nginx as a reverse proxy

HTTP is a complex protocol that deals with data of different modality and has numerous optimizations that—if implemented properly—can lead to a significant increase in web service performance.

At the same time, web application developers have less time to deal with low-level issues and optimizations. The mere idea of decoupling a web application server from a frontend server shifts the focus on managing incoming traffic to the frontend, while shifting the focus on functionality, application logic, and features to the web application server. This is where Nginx comes into play as a decoupling point.
An example of a decoupling point is SSL termination: Nginx receives and processes inbound SSL connections, it forwards the request over plain HTTP to an application server, and wraps the received response back into SSL. The application server no longer needs to take care of storing certificates, SSL sessions, handling encrypted and unencrypted transmission, and so on.

Other examples of decoupling are as follows:

- Efficient handling of static files and delegating the dynamic part to the upstream
- Rate, request, and connection limiting
- Compressing responses from the upstream
- Caching responses from the upstream
- Accelerating uploads and downloads

By shifting these functions to a Nginx-powered frontend, you are essentially investing in the reliability of your website.

**Setting up Nginx as a reverse proxy**

Nginx can be easily configured to work as a reverse proxy:

```nginx
location /example {
    proxy_pass http://upstream_server_name;
}
```

In the preceding code, `upstream_server_name` is the host name of the upstream server. When a request for location is received, it will be passed to the upstream server with a specified host name.

If the upstream server does not have a host name, an IP address can be used instead:

```nginx
location /example {
    proxy_pass http://192.168.0.1;
}
```

If the upstream server is listening on a nonstandard port, the port can be added to the destination URL:

```nginx
location /example {
    proxy_pass http://192.168.0.1:8080;
}
```

The destination URL in the preceding examples does not have a path. This makes Nginx pass the request as is, without rewriting the path in the original request.
If a path is specified in the destination URL, it will replace a part of the path from the original request that corresponds to the matching part of the location. For example, consider the following configuration:

```nginx
location /download {
    proxy_pass http://192.168.0.1/media;
}
```

If a request for `/download/BigFile.zip` is received, the path in the destination URL is `/media` and it corresponds to the matching `/download` part of the original request URI. This part will be replaced with `/media` before passing to the upstream server, so the passed request path will look like `/media/BigFile.zip`.

If `proxy_pass` directive is used inside a regex location, the matching part cannot be computed. In this case, a destination URI without a path must be used:

```nginx
location ~* (script1|script2|script3).php$ {
    proxy_pass http://192.168.0.1;
}
```

The same applies to cases where the request path was changed with the `rewrite` directive and is used by a `proxy_pass` directive.

Variables can be a part of the destination URL as well:

```nginx
location ~* ^/(index|content|sitemap).html$ {
    proxy_pass http://192.168.0.1/html/$1;
}
```

In fact, any part or even the whole destination URL can be specified by a variable:

```nginx
location /example {
    proxy_pass $destination;
}
```

This gives enough flexibility in specifying the destination URL for the upstream server. In Chapter 5, Managing Inbound and Outbound Traffic, we will find out how to specify multiple servers as an upstream and distribute connections among them.

**Setting the backend the right way**

The right way to configure a backend is to avoid passing everything to it. Nginx has powerful configuration directives that help you ensure that only specific requests are delegated to the backend.
Proxying and Caching

Consider the following configuration:

```nginx
location ~* \.php$ {
    proxy_pass http://backend;
    [...]  
}
```

This passes every request with a URI that ends with .php to the PHP interpreter. This is not only inefficient due to the intensive use of regular expressions, but also a serious security issue on most PHP setups because it may allow arbitrary code execution by an attacker.

Nginx has an elegant solution for this problem in the form of the `try_files` directive. The `try_files` directive takes a list of files and a location as the last argument. Nginx tries specified files in consecutive order and if none of them exists, it makes an internal redirect to the specified location. Consider the following example:

```nginx
location / {
    try_files $uri $uri/ @proxy;
}

location @proxy {
    proxy_pass http://backend;
}
```

The preceding configuration first looks up a file corresponding to the request URI, looks for a directory corresponding to the request URI in the hope of returning an index of that directory, and finally makes an internal redirect to the named location @proxy if none of these files or directories exist.

This configuration makes sure that whenever a request URI points to an object in the filesystem it is handled by Nginx itself using efficient file operations, and only if there is no match in the filesystem for the given request URI is it delegated to the backend.

Adding transparency

Once forwarded to an upstream server, a request loses certain properties of the original request. For example, the virtual host in a forwarded request is replaced by the host/port combination of the destination URL. The forwarded request is received from an IP address of the Nginx proxy, and the upstream server's functionality based on the client's IP address might not function properly.
The forwarded request needs to be adjusted so that the upstream server can obtain the missing information of the original request. This can be easily done with the `proxy_set_header` directive:

```
proxy_set_header <header> <value>;
```

The `proxy_set_header` directive takes two arguments, the first of which is the name of the header that you want to set in the proxied request, and the second is the value for this header. Again, both arguments can contain variables.

Here is how you can pass the virtual host name from the original request:

```
location @proxy {
  proxy_pass http://192.168.0.1;
  proxy_set_header Host $host;
}
```

The variable `$host` has a smart functionality. It does not simply pass the virtual host name from the original request, but uses the name of the server the request is processed by if the host header of the original request is empty or missing. If you insist on using the bare virtual host name from the original request, you can use the `$http_host` variable instead of `$host`.

Now that you know how to manipulate the proxied request, we can let the upstream server know the IP address of the original client. This can be done by setting `X-Real-IP` and/or the `X-Forwarded-For` headers:

```
location @proxy {
  proxy_pass http://192.168.0.1;
  proxy_set_header Host $host;
  proxy_set_header X-Real-IP $remote_addr;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
```

This will make the upstream server aware of the original client's IP address via `X-Real-IP` or the `X-Forwarded-For` header. Most application servers support this header and take appropriate actions to properly reflect the original IP address in their API.

**Handling redirects**

The next challenge is rewriting redirects. When the upstream server issues a temporary or permanent redirect (HTTP status codes 301 or 302), the absolute URI in the location or refresh headers needs to be rewritten so that it contains a proper host name (the host name of the server the original request came to).
This can be done using the **proxy_redirect** directive:

```nginx
location @proxy {
    proxy_pass http://localhost:8080;
    proxy_redirect http://localhost:8080/app http://www.example.com;
}
```

Consider a web application that is running at `http://localhost:8080/app`, while the original server has the address `http://www.example.com`. Assume the web application issues a temporary redirect (HTTP 302) to `http://localhost:8080/app/login`. With the preceding configuration, Nginx will rewrite the URI in the location header to `http://www.example.com/login`.

If the redirect URI was not rewritten, the client would be redirected to `http://localhost:8080/app/login`, which is valid only within a local domain, so the web application would not be able to work properly. With the `proxy_redirect` directive, the redirect URI will be properly rewritten by Nginx, and the web application will be able to perform the redirect properly.

The host name in the second argument of the **proxy_redirect** directive can be omitted:

```nginx
location @proxy {
    proxy_pass http://localhost:8080;
    proxy_redirect http://localhost:8080/app /;
}
```

The preceding code can be further reduced to the following configuration using variables:

```nginx
location @proxy {
    proxy_pass http://localhost:8080;
    proxy_redirect http://$proxy_host/app /;
}
```

The same transparency option can be applied to cookies. In the preceding example, consider cookies are set to the domain `localhost:8080`, since the application server replies at `http://localhost:8080`. The cookies will not be returned by the browser, because the cookie domain does not match the request domain.
Handling cookies

To make cookies work properly, the domain name in cookies needs to be rewritten by the Nginx proxy. To do this, you can use the `proxy_cookie_domain` directive as shown here:

```nginx
location @proxy {
    proxy_pass http://localhost:8080;
    proxy_cookie_domain localhost:8080 www.example.com;
}
```

In the preceding example, Nginx replaces the cookie domain `localhost:8080` in the upstream response with `www.example.com`. The cookies set by the upstream server will refer to the domain `www.example.com` and the browser will return cookies in subsequent requests.

If cookie path needs to be rewritten as well due to application server being rooted at a different path, you can use the `proxy_cookie_path` directive as shown in the following code:

```nginx
location @proxy {
    proxy_pass http://localhost:8080;
    proxy_cookie_path /my_webapp/ /;
}
```

In this example, whenever Nginx detects a cookie with a prefix specified in the first argument of the `proxy_cookie_path` directive (`/my_webapp/`), it replaces this prefix with the value in the second argument of the `proxy_cookie_path` directive (`/`).

Putting everything together for the `www.example.com` domain and the web application running at `localhost:8080`, we get the following configuration:

```nginx
location @proxy {
    proxy_pass http://localhost:8080;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_redirect http://$proxy_host/app /;
    proxy_cookie_domain $proxy_host www.example.com;
    proxy_cookie_path /my_webapp/ /;
}
```

The preceding configuration ensures transparency for a web application server so that it doesn't even need to know which virtual host it is running on.
Using SSL

If the upstream server supports SSL, connections to the upstream server can be secured by simply changing the destination URL scheme to https:

```plaintext
location @proxy {
    proxy_pass https://192.168.0.1;
}
```

If the authenticity of the upstream server needs to be verified, this can be enabled using the `proxy_ssl_verify` directive:

```plaintext
location @proxy {
    proxy_pass https://192.168.0.1;
    proxy_ssl_verify on;
}
```

The certificate of the upstream server will be verified against certificates of well-known certification authorities. In Unix-like operating systems, they are usually stored in `/etc/ssl/certs`.

If an upstream uses a trusted certificate that cannot be verified by well-known certification authorities or a self-signed certificate, it can be specified and declared as trusted using the `proxy_ssl_trusted_certificate` directive. This directive specifies the path to the certificate of the upstream server or a certificate chain required to authenticate the upstream server in PEM format. Consider the following example:

```plaintext
location @proxy {
    proxy_pass https://192.168.0.1;
    proxy_ssl_verify on;
    proxy_ssl_trusted_certificate /etc/nginx/upstream.pem;
}
```

If Nginx needs to authenticate itself to the upstream server, the client certificate and the key can be specified using the `proxy_ssl_certificate` and `proxy_ssl_certificate_key` directives. The directive `proxy_ssl_certificate` specifies the path to the client certificate in PEM format, while `proxy_ssl_certificate_key` specifies the path to the private key from the client certificate in PEM format. Consider the following example:

```plaintext
location @proxy {
    proxy_pass https://192.168.0.1;
    proxy_ssl_certificate /etc/nginx/client.pem;
    proxy_ssl_certificate_key /etc/nginx/client.key;
}
```
The specified certificate will be presented while setting up the secure connection to the upstream server, and its authenticity will be verified by specified private key.

Handling errors

If Nginx experiences a problem contacting the upstream server or the upstream server returns an error, there is an option to take certain actions.

The upstream server connectivity errors can be handled using the `error_page` directive:

```
location ~* (script1|script2|script3)\.php$ {
  proxy_pass http://192.168.0.1;
  error_page 500 502 503 504 /50x.html;
}
```

This will make Nginx return the document from the file `50x.html` once an upstream connectivity error has occurred.

This will not change the HTTP status code in the response. To change the HTTP status code to successful, you can use the following syntax:

```
location ~* (script1|script2|script3)\.php$ {
  proxy_pass http://192.168.0.1;
  error_page 500 502 503 504 =200 /50x.html;
}
```

A more sophisticated action can be taken upon failure of an upstream server using an `error_page` directive that points to a named location:

```
location ~* (script1|script2|script3)\.php$ {
  proxy_pass http://upstreamA;
  error_page 500 502 503 504 @retry;
}

location @retry {
  proxy_pass http://upstreamB;
  error_page 500 502 503 504 =200 /50x.html;
}
```

In the preceding configuration, Nginx first tries to fulfill the request by forwarding it to the `upstreamA` server. If this results in an error, Nginx switches to a named location `@retry` in an attempt to try with the `upstreamB` server. Request an URI while switching so that the `upstreamB` server will receive an identical request. If this doesn't help either, Nginx returns a static file `50x.html` pretending no error occurred.
If an upstream has replied but returned an error, it can be intercepted rather than passed to the client using the `proxy_intercept_errors` directive:

```nginx
location ~* (script1|script2|script3)\.php$ {
    proxy_pass http://upstreamA;
    proxy_intercept_errors on;
    error_page 500 502 503 504 403 404 @retry;
}

location @retry {
    proxy_pass http://upstreamB;
    error_page 500 502 503 504 =200 /50x.html;
}
```

In the preceding configuration, the `upstreamB` server will be called even when the `upstreamA` server replies but returns erroneous HTTP status code, such as 403 or 404. This gives `upstreamB` an opportunity to fix the soft errors of `upstreamA`, if necessary.

However, this configuration pattern must not proliferate too much. In *Chapter 5, Managing Inbound and Outbound Traffic*, we will find out how to handle such situations in a more elegant way, without sophisticated configuration structures.

### Choosing an outbound IP address

Sometimes, when your proxy server has multiple network interfaces, it becomes necessary to choose which IP address should be used as outbound address for upstream connections. By default, the system will choose the address of the interface that adjoins the network containing the host used as destination in the default route.

To choose a particular IP address for outbound connections, you can use the `proxy_bind` directive:

```nginx
location @proxy {
    proxy_pass https://192.168.0.1;
    proxy_bind 192.168.0.2;
}
```

This will make Nginx bind outbound sockets to the IP address 192.168.0.2 before making a connection. The upstream server will then see connections coming from IP address 192.168.0.2.
Accelerating downloads

Nginx is very efficient at heavy operations, such as handling large uploads and downloads. These operations can be delegated to Nginx using built-in functionality and third-party modules.

To accelerate download, the upstream server must be able to issue the `X-Accel-Redirect` header that points to the location of a resource which needs to be returned, instead of the response obtained from the upstream. Consider the following configuration:

```nginx
location ~* (script1|script2|script3)\.php$ {
    proxy_pass https://192.168.0.1;
}

location /internal-media/ {
    internal;
    alias /var/www/media/;
}
```

With the preceding configuration, once Nginx detects the `X-Accel-Redirect` header in the upstream response, it performs an internal redirect to the location specified in this header. Assume the upstream server instructs Nginx to perform an internal redirect to `/internal-media/BigFile.zip`. This path will be matched against the location `/internal-media`. This location specifies the document root at `/var/www/media`. So if a file `/var/www/media/BigFile.zip` exists, it will be returned to the client using efficient file operations.

For many web application servers, this feature provides an enormous speed up—both because they might not handle large downloads efficiently and because proxying reduces efficiency of large downloads.

Caching

Once Nginx is set up as a reverse proxy, it's logical to turn it into a caching proxy. Fortunately, this can be achieved very easily with Nginx.

Configuring caches

Before you can enable caching for a certain location, you need to configure a cache. A cache is a filesystem directory containing files with cached items and a shared memory segment where information about cached items is stored.
A cache can be declared using the `proxy_cache_path` directive:

```
proxy_cache_path <path> keys_zone=<name>:<size> [other parameters...];
```

The preceding command declares a cache rooted at the path `<path>` with a shared memory segment named `<name>` of the size `<size>`.

This directive has to be specified in the `http` section of the configuration. Each instance of the directive declares a new cache and must specify a unique name for a shared memory segment. Consider the following example:

```
http {
    proxy_cache_path /var/www/cache keys_zone=my_cache:8m;
    [...]
}
```

The preceding configuration declares a cache rooted at `/var/www/cache` with a shared memory segment named `my_cache`, which is 8 MB in size. Each cache item takes around 128 bytes in memory, thus the preceding configuration allocates space for around 64,000 items.

The following table lists other parameters of `proxy_cache_path` and their meaning:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>levels</td>
<td>Specifies hierarchy levels of the cache directory</td>
</tr>
<tr>
<td>inactive</td>
<td>Specifies the time after which a cache item will be removed from the cache if it was not used, regardless of freshness</td>
</tr>
<tr>
<td>max_size</td>
<td>Specifies maximum size (total size) of all cache items</td>
</tr>
<tr>
<td>loader_files</td>
<td>Specifies the number of files a cache loader process loads in each iteration</td>
</tr>
<tr>
<td>loader_sleep</td>
<td>Specifies the time interval a cache loader process sleeps between each iteration</td>
</tr>
<tr>
<td>loader_threshold</td>
<td>Specifies the time limit for each iteration of a cache loader process</td>
</tr>
</tbody>
</table>

Once Nginx starts, it processes all configured caches and allocates shared memory segments for each of the caches.

After that, a special process called cache loader takes care of loading cached items into memory. Cache loader loads items in iterations. The parameters `loader_files`, `loader_sleep`, and `loader_threshold` define the behavior of the cache loader process.
When running, a special process called cache manager monitors the total disk space taken by all cache items and evicts less requested items if the total consumed space is larger than specified in the max_size parameter.

**Enabling caching**

To enable caching for a location, you need to specify the cache using the proxy_cache directive:

```conf
location @proxy {
  proxy_pass http://192.168.0.1:8080;
  proxy_cache my_cache;
}
```

The argument of the proxy_cache directive is the name of a shared memory segment that points to one of the caches configured using the proxy_cache_path directive. The same cache can be used in multiple locations. The upstream response will be cached if it is possible to determine the expiration interval for it. The primary source for the expiration interval for Nginx is the upstream itself. The following table explains which upstream response header influences caching and how:

<table>
<thead>
<tr>
<th>Upstream response header</th>
<th>How it influences caching</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-Accel-Expires</td>
<td>This specifies the cache item expiration interval in seconds. If the value starts from @, then the number following it is UNIX timestamp when the item is due to expire. This header has the higher priority.</td>
</tr>
<tr>
<td>Expires</td>
<td>This specifies the cache item expiration time stamp.</td>
</tr>
<tr>
<td>Cache-Control</td>
<td>This enables or disables caching</td>
</tr>
<tr>
<td>Set-Cookie</td>
<td>This disables caching</td>
</tr>
<tr>
<td>Vary</td>
<td>The special value * disables caching.</td>
</tr>
</tbody>
</table>

It is also possible to explicitly specify an expiration interval for various response codes using the proxy_cache_valid directive:

```conf
location @proxy {
  proxy_pass http://192.168.0.1:8080;
  proxy_cache my_cache;
  proxy_cache_valid 200 301 302 1h;
}
```
This sets the expiration interval for responses with codes 200, 301, 302 to 1h (1 hour). Note that the default status code list for the `proxy_cache_valid` directive is 200, 301, and 302, so the preceding configuration can be simplified as follows:

```nginx
location @proxy {
    proxy_pass http://192.168.0.1:8080;
    proxy_cache my_cache;
    proxy_cache_valid 10m;
}
```

To enable caching for negative responses, such as 404, you can extend the status code list in the `proxy_cache_valid` directive:

```nginx
location @proxy {
    proxy_pass http://192.168.0.1:8080;
    proxy_cache my_cache;
    proxy_cache_valid 200 301 302 1h;
    proxy_cache_valid 404 1m;
}
```

The preceding configuration will cache 404 responses for 1m (1 minute). The expiration interval for negative responses is deliberately set to much lower values than that of the positive responses. Such an optimistic approach ensures higher availability by expecting negative responses to improve, considering them as transient and assuming a shorter expected lifetime.

### Choosing a cache key

Choosing the right cache key is important for the best operation of the cache. The cache key must be selected such that it maximizes the expected efficiency of the cache, provided that each cached item has valid content for all subsequent requests that evaluate to the same key. This requires some explanation.

First, let's consider efficiency. When Nginx refers to the upstream server in order to revalidate a cache item, it obviously stresses the upstream server. With each subsequent cache hit, Nginx reduces the stress on the upstream server in comparison to the situation when requests were forwarded to the upstream without caching. Thus, the efficiency of the cache can be represented as \( \text{Efficiency} = \frac{\text{Number hits} + \text{Number misses}}{\text{Number misses}} \).

Thus, when nothing can be cached, each request leads to a cache miss and the efficiency is 1. But when we get 99 subsequent cache hits for each cache miss, the efficiency evaluates to \( \frac{99 + 1}{1} = 100 \), which is 100 times larger!
Second, if a document is cached but it is not valid for all requests that evaluate to the same key, clients might see content that is not valid for their requests.

For example, the upstream analyses the `Accept-Language` header and returns the version of the document in the most suitable language. If the cache key does not include the language, the first user to request the document will obtain it in their language and trigger the caching in that language. All users that subsequently request this document will see the cached version of the document, and thus they might see it in the wrong language.

If the cache key includes the language of the document, the cache will contain multiple separate items for the same document in each requested language, and all users will see it in the proper language.

The default cache key is `$scheme$proxy_host$request_uri$`.

This might not be optimal because of the following reasons:

- The web application server at `$proxy_host` can be responsible for multiple domains
- The HTTP and HTTPS versions of the website can be identical (`$scheme` variable is redundant, thus duplicating items in the cache)
- Content can vary depending on query arguments

Thus, considering everything described previously and given that HTTP and HTTPS versions of the website are identical and content varies depending on query arguments, we can set the cache key to a more optimal value `$host$request_uri$is_args$args$. To change the default cache item key, you can use the `proxy_cache_key` directive:

```nginx
location @proxy {
    proxy_pass http://192.168.0.1:8080;
    proxy_cache my_cache;
    proxy_cache_key "$host$request_uri$is_args$args";
}
```

This directive takes a script as its argument which is evaluated into a value of a cache key at runtime.
Improving cache efficiency and availability

The efficiency and availability of the cache can be improved. You can prevent an item from being cached until it gets a certain minimum number of requests. This could be achieved using the `proxy_cache_min_uses` directive:

```nginx
location @proxy {
    proxy_pass http://192.168.0.1:8080;
    proxy_cache my_cache;
    proxy_cache_min_uses 5;
}
```

In the preceding example, the response will be cached once the item gets no less than five requests. This prevents the cache from being populated by infrequently used items, thus reducing the disk space used for caching.

Once the item has expired, it can be revalidated without being evicted. To enable revalidation, use the `proxy_cache_revalidate` directive:

```nginx
location @proxy {
    proxy_pass http://192.168.0.1:8080;
    proxy_cache my_cache;
    proxy_cache_revalidate on;
}
```

In the preceding example, once a cache item expires, Nginx will revalidate it by making a conditional request to the upstream server. This request will include the `If-Modified-Since` and/or `If-None-Match` headers as a reference to the cached version. If the upstream server responds with a `304 Not Modified` response, the cache item remains in the cache and the expiration time stamp is reset.

Multiple simultaneous requests can be prohibited from filling the cache at the same time. Depending on the upstream reaction time, this might speed up cache population while reducing the load on the upstream server at the same time. To enable this behavior, you can use the `proxy_cache_lock` directive:

```nginx
location @proxy {
    proxy_pass http://backend;
    proxy_cache my_cache;
    proxy_cache_lock on;
}
```
Once the behavior is enabled, only one request will be allowed to populate a cache item it is related to. The other requests related to this cache item will wait until either the cache item is populated or the lock timeout expires. The lock timeout can be specified using the `proxy_cache_lock_directive` directive.

If higher availability of the cache is required, you can configure Nginx to reply with stale data when a request refers to a cached item. This is very useful when Nginx acts as an edge server in a distribution network. The users and search engine crawlers will see your web site available, even though the main site experiences connectivity problems. To enable replying with stale data, use the `proxy_cache_use_stale` directive:

```bash
location @proxy {
  proxy_pass http://backend;
  proxy_cache my_cache;
  proxy_cache_use_stale error timeout http_500 http_502 http_503 http_504;
}
```

The preceding configuration enables replying with stale data in case of connectivity error, upstream error (502, 503, or 504), and connection timeout. The following table lists all possible values for arguments of the `proxy_cache_use_stale` directive:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>A connection error has occurred or an error during sending a request or receiving a reply has occurred</td>
</tr>
<tr>
<td>timeout</td>
<td>A connection timed out during setup, sending a request or receiving a reply</td>
</tr>
<tr>
<td>invalid_header</td>
<td>The upstream server has returned an empty or invalid reply</td>
</tr>
<tr>
<td>updating</td>
<td>Enables stale replies while the cache item is being updated</td>
</tr>
<tr>
<td>http_500</td>
<td>The upstream server returned a reply with HTTP status code 500 (Internal Server Error)</td>
</tr>
<tr>
<td>http_502</td>
<td>The upstream server returned a reply with HTTP status code 502 (Bad Gateway)</td>
</tr>
<tr>
<td>http_503</td>
<td>The upstream server returned a reply with HTTP status code 503 (Service Unavailable)</td>
</tr>
<tr>
<td>http_504</td>
<td>The upstream server returned a reply with HTTP status code 504 (Gateway Timeout)</td>
</tr>
<tr>
<td>http_403</td>
<td>The upstream server returned a reply with HTTP status code 403 (Forbidden)</td>
</tr>
<tr>
<td>http_404</td>
<td>The upstream server returned a reply with HTTP status code 404 (Not Found)</td>
</tr>
<tr>
<td>off</td>
<td>Disables use of stale replies</td>
</tr>
</tbody>
</table>
Handling exceptions and borderline cases

When caching is not desirable or not efficient, it can be bypassed or disabled. This can happen in the following instances:

- A resource is dynamic and varies depending on external factors
- A resource is user-specific and varies depending on cookies
- Caching does not add much value
- A resource is not static, for example a video stream

When bypass is forced, Nginx forwards the request to the backend without looking up an item in the cache. The bypass can be configured using the `proxy_cache_bypass` directive:

```nginx
location @proxy {
    proxy_pass http://backend;
    proxy_cache my_cache;
    proxy_cache_bypass $do_not_cache $arg_nocache;
}
```

This directive can take one or more arguments. When any of them evaluate to true (nonempty value and not 0), Nginx does not look up an item in the cache for a given request. Instead, it directly forwards the request to the upstream server. The item can still be stored in the cache.

To prevent an item from being stored in the cache, you can use the `proxy_no_cache` directive:

```nginx
location @proxy {
    proxy_pass http://backend;
    proxy_cache my_cache;
    proxy_no_cache $do_not_cache $arg_nocache;
}
```

This directive works exactly like the `proxy_cache_bypass` directive, but prevents items from being stored in the cache. When only the `proxy_no_cache` directive is specified, the items can still be returned from the cache. The combination of both `proxy_cache_bypass` and `proxy_no_cache` disables caching completely.
Now, let's consider a real-world example when caching needs to be disabled for all user-specific pages. Assume that you have a website powered by WordPress and you want to enable caching for all pages but disable caching for all customized or user-specific pages. To implement this, you can use a configuration similar to the following:

```nginx
location ~* wp-.*\.php|wp-admin {
    proxy_pass http://backend;
    proxy_set_header Host $http_host;
    proxy_set_header X-Real-IP $remote_addr;
}

location / {
    if ($http_cookie ~* "comment_author_|wordpress_|wp-postpass_") {
        set $do_not_cache 1;
    }
    proxy_pass http://backend;
    proxy_set_header Host $http_host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_cache my_cache;
    proxy_cache_bypass $do_not_cache;
    proxy_no_cache $do_not_cache;
}
```

In the preceding configuration, we first delegate all requests pertaining to the WordPress administrative area to the upstream server. We then use the `if` directive to look up WordPress login cookies and set the `$do_not_cache` variable to 1 if they are present. Then, we enable caching for all other locations but disable caching whenever the `$do_not_cache` variable is set to 1 using the `proxy_cache_bypass` and `proxy_no_cache` directives. This disables caching for all requests with WordPress login cookies.

The preceding configuration can be extended to extract no-cache flags from arguments or HTTP headers, to further tune your caching.
Summary
In this chapter, you learned how to work with proxying and caching—some of the most important Nginx features. These features practically define Nginx as a web accelerator and being proficient in them is essential to get the most out of Nginx.

In the next chapter, we'll look into how to rewrite engine works in Nginx and the basics of access control.
Where to buy this book

You can buy Nginx Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.