# Shiny Server Administrator's Guide

*Shiny Server Professional v1.5.9*

# Contents

# Chapter 1

# Getting Started

## 1.1 Introduction

Shiny Server enables users to host and manage Shiny applications on the Internet. Shiny is an R package that uses a reactive programming model to simplify the development of R-powered web applications. Shiny Server can manage R processes running various Shiny applications over different URLs and ports. Using Shiny Server offers a variety of benefits over simply running Shiny in R directly. These features allow the administrator to:

- Host multiple applications simultaneously, each at its own URL.
- Support web browsers that don't support WebSocket, including Internet Explorer 8 & 9.
- Enable system users to develop and manage their own Shiny applications.
- Ensure that R processes that crash or are terminated automatically restart for the next user requesting the application.

This manual describes Shiny Server Professional, which offers, among other things, the following additional features:

- Ensure your applications are protected and can only be accessed by specific, authenticated users.
- Scale a Shiny application to support many users by empowering a Shiny application to be backed by multiple R Shiny processes simultaneously.
- Gain insight into the performance and usage of your Shiny applications by monitoring them using a web dashboard.
- Securely encrypt data being sent to and from your applications using SSL.
- Understand and manage current and historical application resource utilization to better configure and optimize your applications.
- Fine-tune the resources devoted to each user of an application by configuring multi-process Shiny applications based on the number of concurrent sessions.
- Monitor the health of your Shiny Server using the health check endpoint.

## 1.2 System Requirements

Shiny Server is currently only supported on the Linux operating system with the following distributions:

- RedHat Enterprise Linux (and CentOS) 6 and higher
- Ubuntu 14.04 and higher
- SUSE Linux Enterprise 12 and higher

We currently only support the x86-64 architecture. As described in the Installation section, you will install R and the Shiny package prior to installing Shiny Server. Root privileges will be required both in the installation process and also at runtime.

Shiny Server Professional allows you to host multiple R processes concurrently to balance the load of incoming requests across multiple Shiny instances. The Utilization Scheduler section explains how to configure this in more detail. We recommend carefully evaluating the computational properties and memory profile of a Shiny application using the tools discussed in the Admin section. This dashboard will provide much more insight into the profile of a Shiny application than was previously possible.

## 1.3 Installation

The Shiny Server installer does not include R or the Shiny R package. Below are the steps for installing each of these separately.

If you had previously installed Shiny Server 0.3.x or 0.4.x, please see the relevant section before proceeding to ensure a seamless upgrade:

- Upgrading from Shiny Server 0.3.x
- Upgrading from Shiny Server 0.4.x

### 1.3.1 Ubuntu (14.04+)

#### Installing R

Shiny Server recommends an installation of R version 3.0 or higher. To install the latest version of R, you should first add the CRAN repository to your system as described here:

- Debian: https://cran.rstudio.com/bin/linux/debian/
- Ubuntu: http://cran.rstudio.com/bin/linux/ubuntu/README.html

You can then install R using the following command:

```
$ sudo apt-get install r-base
```

NOTE: if you do not add the CRAN Debian or Ubuntu repository as described above, this command will install the version of R corresponding to your current system version. Since this version of R may be a year or more old, it is strongly recommended that you add the CRAN repositories so you can run the most recent version of R.

Once R is installed, follow the instructions in Install Shiny to setup the necessary packages in R.

Once the Shiny package is installed, you can begin the installation of Shiny Server. You should have been provided with a `.deb` installer for Shiny Server. If you only have a link to this file, you can use `wget` to download the file to the current directory. First, you will need to install the `gdebi-core` package to install Shiny Server and its dependencies. Once the `.deb` file is available locally, run the following commands to complete the installation of Shiny Server.

```
sudo apt-get install gdebi-core
sudo gdebi shiny-server-<version>.deb
```

This will install Shiny Server into `/opt/shiny-server/`, with the main executable in `/opt/shiny-server/bin/shiny-server` and also create a new `shiny` user. The rest of this guide will document the intricacies of configuring and managing Shiny Server. If you are just looking to get up and running quickly, we recommend that you read the Quick Start section in the appendix, which will walk you through the process of installing and configuring a Shiny application.

### 1.3.2 RedHat/CentOS (6+)

#### Prerequisites

Shiny Server recommends an installation of R version 3.0 or higher. Shiny Server has several dependencies on packages (including R itself) found in the Extra Packages for Enterprise Linux (EPEL) repository. If you do not already have this repository available, you should add it to your system using the instructions found here: https://fedoraproject.org/wiki/EPEL. On some distributions of RedHat/CentOS, the R package references dependencies that are not available by default. In this case, you may need to edit the `/etc/yum.repos.d/redhat.repo` file to enable the `rhel-6-server-optional-rpms` (by setting `enabled = 1`) before you can install the R package.

After enabling EPEL, you should then ensure that you have installed the version of R available from EPEL. You can do this using the following command:

```
$ sudo yum install R
```

Once R is installed, follow the instructions in Install Shiny to setup the necessary packages in R.

Once the Shiny package has been installed, you can begin the installation of Shiny Server. You should have been provided with an RPM file which contains Shiny Server and all of its dependencies (other than R and Shiny). You can install this rpm file using `yum`. If you have only a link to the RPM file, you can use `wget` to download the file to the current directory. You can install this RPM file using `yum`.

```
$ sudo yum install --nogpgcheck shiny-server-<version>.rpm
```

This will install Shiny Server into `/opt/shiny-server/`, with the main executable in `/opt/shiny-server/bin/shiny-server`, and also create a new `shiny` user. The rest of this guide will document the intricacies of configuring and managing Shiny Server. If you are just looking to get up and running quickly, we recommend that you read the Quick Start section in the appendix, which will walk you through the process of installing and configuring a Shiny application.

### 1.3.3 SUSE Linux Enterprise Server 12

#### Prerequisites

Shiny Server recommends an installation of R version 3.0 or higher.

You can install R for SUSE Linux Enterprise Server using the following commands:

```
$ sudo zypper addrepo -f http://download.opensuse.org/repositories/devel\:/languages\:/R\:/pat
$ sudo zypper install R-base
$ sudo zypper install gcc-c++  # Needed to install R packages from source
```

Once R is installed, follow the instructions in Install Shiny to setup the necessary packages in R.

Once the Shiny package has been installed, you can begin the installation of Shiny Server.

#### Installation

You should have been provided with an RPM file which contains Shiny Server and all of its dependencies (other than R and Shiny). You can install this RPM file using `zypper`. If you have only a link to the RPM file, you can use `wget` to download the file to the current directory. You can install this RPM file using `zypper`.

```
$ sudo zypper --no-gpg-checks install shiny-server-<version>.rpm
```

During the installation, `zypper` will prompt to ensure that you still want to install even without the `libffi` dependency. The prompt will look like this:

```
Problem: nothing provides libffi needed by shiny-server
Solution 1: do not install shiny-server
Solution 2: break shiny-server by ignoring some of its dependencies
Choose from above solutions by number or cancel [1/2/c] (c): 2
```

The `libffi` dependency is not required on SLES, so it is safe to install without it. You should enter "2" (ignore the dependencies) at this prompt, and continue with the installation.

This will install Shiny Server into `/opt/shiny-server/`, with the main executable in `/opt/shiny-server/bin/shiny-server`, and also create a new `shiny` user. The rest of this guide will document the intricacies of configuring and managing Shiny Server. If you are just looking to get up and running quickly, we recommend that you read the Quick Start section in the appendix, which will walk you through the process of installing and configuring a Shiny application.

### 1.3.4 Install Shiny

Before Shiny Server can be installed, the Shiny package must be installed in the system library; you typically need `sudo` privileges to install to this library.

#### Setup the Server for Secure Package Installation

R packages can be installed securely using an HTTPS CRAN mirror, or insecurely using an HTTP mirror. Beginning with R version 3.2.2, HTTPS is the default preference when installing packages, but older versions of R default to insecure HTTP. You can change this behavior in older versions of R by setting the `download.file.method` option in your `.Rprofile`. See R Startup Files for details on where these files are located.

You should add the following line to `Rprofile.site`, to configure how all users install packages, or the `~/.Rprofile` file for individual users who will be running Shiny applications in Shiny Server.

*For R 3.2*

```
options(download.file.method = "libcurl")
```

*For R 3.1 and earlier*

```
options(download.file.method = "wget")
```

You could alternatively use `curl` instead of `wget`, if `wget` is not available on your server.

You should also specify a secure default CRAN mirror in this same file. You can do that using the following code:

```
local({
  r <- getOption("repos")
  r["CRAN"] <- "https://cran.rstudio.com/"
  options(repos=r)
})
```

#### Install The Shiny Package

Shiny Server currently requires Shiny version 0.7.0 or later. The following command will download and install the `shiny` package from CRAN in the system library.

```
$ sudo su - -c "R -e \"install.packages('shiny')\""
```

If you have not set a default CRAN repository (as described in the section above), you may need to specify the `repos` parameter from which to download `shiny` and its dependencies.

Once this command completes, you can continue with the installation of Shiny Server.

### 1.3.5 R Installation Location

Shiny Server expects that R is available as an executable named `R` and is in the `PATH` of the user whom you run `shiny-server` as. Note that on some CentOS systems, the `PATH` will be overridden by the startup script to `/sbin:/usr/sbin:/bin:/usr/bin`. On such systems, if `R` is not available in one of these locations (regardless of the user's `PATH`), you'll need to adjust the startup script.

To allow Shiny Server to search for `R` in additional locations, you must alter the file in `/etc/init.d/shiny-server` or `/etc/init/shiny-server.conf`, depending on which startup system you're using (as discussed in Stopping and Starting. You can either adjust the `PATH` variable to include the directory where `R` will be found, or you can set an environment variable named `R` to tell Shiny Server exactly where it should look for the executable.

If you choose to adjust the `PATH`, you can add the directory in which the executable named `R` is found to the line that defines the `PATH` environment variable (`PATH=/sbin:/usr/sbin:/bin:/usr/bin`).

If you choose to tell Shiny Server the exact executable to run (which is necessary if the executable is not named `R` on your system), you must define a new environment variable named `R`. You can do this by adding a line that looks something like `env R=/usr/local/bin/R-3-0-1` for Upstart, or `export R=/usr/local/bin/R-3-0-1` for init.d.

## 1.4 Stopping and Starting

The installer will automatically deploy the necessary scripts to ensure that Shiny Server is started automatically on boot. When possible, we use systemd or Upstart to manage the `shiny-server` service. If neither is available, we will deploy an `init.d` script to start and stop the service automatically.

### 1.4.1 systemd (RedHat 7, Ubuntu 15.04+, SLES 12+)

systemd is a management and configuration platform for Linux. The newest versions of most major Linux distributions have adopted systemd as their default init system.

The Shiny Server installer will automatically install a systemd service called shiny-server, which will cause the `shiny-server` program to be started and stopped automatically when the machine boots up and shuts down. The shiny-server service will also be launched automatically when the installer has successfully installed the program.

To start or stop the server manually, you can use the following commands.

```
$ sudo systemctl start shiny-server
```

```
$ sudo systemctl stop shiny-server
```

You can restart the server with:

```
$ sudo systemctl restart shiny-server
```

This command will shutdown all running Shiny processes, disconnect all open connections, and re-initialize the server.

If you wish to reload the configuration but keep the server and all Shiny processes running without interruption, you can use the systemctl command to send a `SIGHUP` signal:

```
$ sudo systemctl kill -s HUP --kill-who=main shiny-server
```

This will cause the server to re-initialize, but will not interrupt the current processes or any of the open connections to the server.

You can check the status of the shiny-server service using:

```
$ sudo systemctl status shiny-server
```

And finally, you can use the `enable`/`disable` commands to control whether Shiny Server should be run automatically at boot time:

```
$ sudo systemctl enable shiny-server
```

```
$ sudo systemctl disable shiny-server
```

### 1.4.2 Upstart (Ubuntu 14.04, RedHat 6)

Upstart is a system used to automatically start, stop and manage services. The installer will automatically deploy an Upstart script to `/etc/init/shiny-server.conf`. This script will initialize `shiny-server` as soon as the network is activated on the machine, and stop when the machine is being shut down.

The Upstart script will also ensure that `shiny-server` is respawned if the process is terminated unexpectedly. However, in the event that there is an issue that consistently prevents Shiny Server from being able to start (such as a bad configuration file), Upstart will give up on restarting the service after approximately 5 failed attempts within a few seconds. For this reason, you may see multiple repetitions of a bad Shiny Server startup attempt before it transitions to the `stopped` state.

To start or stop the server, run the following commands, respectively.

```
$ sudo start shiny-server
```

```
$ sudo stop shiny-server
```

To restart the server you can run:

```
$ sudo stop shiny-server
$ sudo start shiny-server
```

Note that we recommend `stop` and `start` rather than just `restart`, as `restart` caches some data that may cause the server not to detect some recently made changes.

This command will shutdown all running Shiny processes, disconnect all open connections, and re-initialize the server. Note that `restart` *will not* reread the Upstart definition at `/etc/init/shiny-server.conf`. So if you have changed, for instance, any environment variables in that file, you will need to `stop` and `start` to have those changes take effect.

If you wish to reload the configuration but keep the server and all Shiny processes running without interruption, you can use the `reload` command as in:

```
$ sudo reload shiny-server
```

This will cause the server to re-initialize, but will not interrupt the current processes or any of the open connections to the server.

**Known Issue:** Due to a bug in the version of Upstart that comes with Ubuntu 13.04, `reload` will not behave as expected on that platform and should not be used.

To check the status or retrieve the Process ID associated with `shiny-server`, run the following:

```
$ status shiny-server
```

# Chapter 2

# Server Management

## 2.1 Default Configuration

Initially, Shiny Server uses the following configuration file. Some users will find that this configuration meets their needs; others may find it useful to create a custom configuration. Details about each available setting and parameter are available in the Appendix. As a brief introduction, however, this section discusses the default configuration file is installed at `/etc/shiny-server/shiny-server.conf` if it doesn't already exist:

```
# Define the user we should use when spawning R Shiny processes
run_as shiny;

# Define a top-level server which will listen on a port
server {
  # Instruct this server to listen on port 3838
  listen 3838;

  # Define the location available at the base URL
  location / {
    #### PRO ONLY ####
    # Only up tp 20 connections per Shiny process and at most 3 Shiny processes
    # per application. Proactively spawn a new process when our processes reach
    # 90% capacity.
    utilization_scheduler 20 .9 3;
    #### END PRO ONLY ####

    # Run this location in 'site_dir' mode, which hosts the entire directory
    # tree at '/srv/shiny-server'
    site_dir /srv/shiny-server;

    # Define where we should put the log files for this location
    log_dir /var/log/shiny-server;

    # Should we list the contents of a (non-Shiny-App) directory when the user
```

```
    # visits the corresponding URL?
    directory_index on;
  }
}


# Setup a flat-file authentication system.
auth_passwd_file /etc/shiny-server/passwd;

# Define a default admin interface to be run on port 4151.
admin 4151 {
  # Only permit the user named `admin` to access the admin interface.
  required_user admin;
}
```

Lines beginning with a `#` are treated as comments and not parsed when configuring the server. Shiny Server can be configured to host multiple `server`s on different ports or hostnames. Each `server` can have `location`s that are capable of serving Shiny Applications and static assets, as well. Individual applications can also override the settings applied to their parent `location`. These concepts are explained in further detail in the Server Hierarchy section. The default configuration above will create a single server listening on port 3838, serving any application contained within `/srv/shiny-server/` at the root URL (`/`). Each possible setting in the configuration file is explained in the Appendix.

Most users will want to customize the configuration to meet their needs. The server will load its configuration from a file stored at `/etc/shiny-server/shiny-server.conf`; it is in this file that you should customize your Shiny Server configuration.

This configuration will also create an administrative dashboard running on port 4151. The admin interface requires that a user authenticate themselves, which is discussed further in the chapter on Authentication & Security. The configuration will attempt to use a flat-file authentication system stored at `/etc/shiny-server/passwd`; an empty database is created for you here during installation. To create a new user named `admin` in this file to allow you to login to the dashboard, execute the following command

```
$ sudo /opt/shiny-server/bin/sspasswd /etc/shiny-server/passwd admin
```

and then enter and verify the password you wish to use for this user.

## 2.2 Server Hierarchy

Detailed descriptions of all available parameters are available in the appendix, but it is important to understand the overall hierarchy of the Shiny Server configuration file when editing the file.

### 2.2.1 Server

The server setting defines an HTTP server which will listen on a port/IP address combination. For example, the following lines:

```
server {
  listen 80;
}
```

define a server that would listen on port 80. A server can also define a `server_name` to limit the virtual hostnames on which it listens, as described in the Virtual Hosts section.

Note that, while using port 80 will allow you to access the server without an explicit port in the URL, it would conflict with the default port of any other web server on the same machine.

### 2.2.2 Location

The `location` setting is defined within a `server` and defines how a particular URL path should be served. For instance, the following settings:

```
server {
  ...
  # Define the location '/specialApp'
  location /specialApp {
    # Run this location in 'app_dir' mode, which will host a single Shiny
    # Application available at '/srv/shiny-server/myApp'
    app_dir /srv/shiny-server/myApp
  }

  # Define the location '/otherApps'
  location /otherApps {
    # Run this location in 'site_dir' mode, which hosts the entire directory
    # tree at '/srv/shiny-server/apps'
    site_dir /srv/shiny-server/apps;
  }
...
}
```

would define two locations, one that serves (potentially) a multitude of applications at the URL `/otherApps/`, and another that serves a single application at the URL `/specialApp/`. The various hosting models that can be applied to a location are described in the section on Hosting Models.

`location` directives can also be nested to provide more granular settings for a particular sub-`location`. For instance, if you used Google Analytics but the Finance department wanted to use its own Google Analytics ID for their `finance` directory, you could accomplish that with a configuration like:

```
server {
  ...
  # Define the '/depts' location
  location /depts {
    # Host a directory of applications
```

```
    site_dir /srv/departmentApps;

    # Provide a default/global GAID
    google_analytics_id "UA-12345-1";

    # Define the '/finance' location.
    # Corresponds to the URL '/depts/finance', and the filesystem path
    # '/srv/departmentApps/finance' as defined by the parent location.
    location /finance {
      # Provide a custom GAID for only this sub-location
      google_analytics_id "UA-54321-9";
    }
  }
...
```

Note that the nested `location`'s path is relative to the parent `location`'s, e.g., /depts/finance in this example. Any directive that can be used for a `location` can be used in a nested `location`, and will override the value specified in the parent, if any. If not overridden, the settings will be inherited by sub-`location`s from their parent `location`s.

### 2.2.3 Application

> **Deprecated!** The `application` setting has been deprecated and removed from Shiny Server as of version 0.4.2.

See How Do I Translate My 'application' Settings to Nested 'location's? for help migrating.

## 2.3 `run_as`

Understanding which user will execute the R Shiny processes is important for a variety of reasons. For one, the paths in which R will look for packages (`.libPaths()`) are often user-dependent. In order to ensure that the libraries required to run a Shiny application are installed, you must first understand which user will be running the application. Additionally, directories will likely have restrictions regarding which users are able to read or write in them. The server should be configured in such a way that the user running the Shiny applications has the minimal privileges to do the work required by those applications.

`location`s configured with `user_apps` will be executed as the user in whose home directory the application was found. For locations configured with `site_dir`, `user_dirs`, and `app_dir`, the `run_as` setting will be used to determine which user should spawn the R Shiny processes. This setting can be configured globally, or for a particular `server` or `location`. For example, the following configuration:

```
location / {
  run_as tim;
}
```

would execute all applications contained within this scope as the user `tim` with all of `tim`'s relevant `.libPaths`.

Additionally, R processes are spawned using the Bash login shell. This means that prior to the execution of the R session, the Bash shell will read and execute commands from this file if it exists:

```
/etc/profile
```

After reading that file, it looks for the following files, and reads and executes commands from the first one that exists and is readable (it is important to note that only one of these files will be read and executed):

```
~/.bash_profile
~/.bash_login
~/.profile
```

So in the above configuration example, any environment variables defined in `/etc/profile` or `/home/tim/.bash_profile` (assuming `tim`'s home directory was in the standard location) would be available to the R process, since Shiny Server is configured to run applications as the user named `tim`.

### 2.3.1 :HOME_USER:

`:HOME_USER:` is a special instance of a `run_as` user. When combined with the `user_dirs` hosting model (described in the section entitled Host Per-User Application Directories), this setting will instruct Shiny Server to run the process as the user in whose home directory the application exists. For instance, an application stored in `/home/jim/ShinyApps/app1` would run as `jim`, whereas an app stored in `/home/kelly/ShinyApps/app1` would run as `kelly`.

Because this username is "special", it may make sense to provide an additional username as a "fallback" user. For instance, in the following configuration:

```
run_as :HOME_USER: shiny;

...

location /users {
  user_dirs;
}

location /apps {
  site_dir /srv/shiny-server;
  log_dir /var/log/shiny-server;
  directory_index on;
}
```

all of the `user_dirs` applications hosted in `/users` would run as `:HOME_USER:` (the user in whose home directory the application exists), since that is the highest-priority `run_as` user. Because `:HOME_USER:` is only meaningful to the `user_dirs` hosting model, and not to `site_dir`, the applications hosted in `/apps` will not run as `:HOME_USER:` but as whichever user comes next in the list – in this case, as `shiny`.

You could also accomplish this same effect with the following arrangement:

```
location /users {
  run_as :HOME_USER:;
  user_dirs;
}

location /apps {
  run_as shiny;
  site_dir /srv/shiny-server;
  log_dir /var/log/shiny-server;
  directory_index on;
}
```

As described in Host Per-User Application Directories, `user_apps` and `user_dirs` differ only in how they behave with respect to the `run_as` setting. `user_apps` will *always* host the apps as if `run_as` were set to `:HOME_USER:`, regardless of how `run_as` is actually set. `user_dirs`, on the other hand, will respect the `run_as` configuration.

### 2.3.2 Running Shiny Server with Root Privileges

Aside from spawning R Shiny processes as particular users, the `shiny-server` process itself can be configured to run as different users to control its privileges. There are many scenarios in which Shiny Server would need to be run as root:

1. If `user_apps` or `user_dirs` is enabled for any location. In order to host applications as various users, Shiny Server must have root privileges.
2. If your configuration uses `run_as` to spawn applications as multiple different users.
3. If you're running any server on a privileged port (a port in the range of 1-1024).
4. If you're using `auth_pam` to leverage PAM authentication. (Pro only)
5. If you're using `auth_google` and store your client secret in a file. (Pro only)
6. If you specify a value for `metrics_user` (Pro only)

By default, the `shiny-server` process will be started as the `root` user, then will spawn R processes according to the corresponding `run_as` setting. You can, however, run the `shiny-server` process as a non-privileged user such as the `shiny` user if none of the limitations above are violated.

Shiny Server Professional leverages PAM to spawn sessions for users. See the section on PAM Sessions to find more details about how you can use PAM to tailor constraints on the Shiny processes that Shiny Server spawns.

For Shiny Server Pro, if it detects that it is running as root but has no need to do so (i.e., none of the criteria above were met), then it will drop its privileges and run as an unprivileged user. Note that, for security reasons, that change is irreversible. This means that once Shiny Server drops its privileges, it will not be able to regain them should a new configuration be loaded that requires such privileges.

By default, if `shiny-server` detects that it does not need to run with `root` privileges, it will attempt to run as an unprivileged user to be more secure. It will determine which user to run as by inspecting the `run_as` directives in your configuration file – if there is only one user defined in a `run_as` statement, and the aforementioned constraints are satisfied, the entire Shiny Server process will just run as that user. The default configuration sets `run_as` to `shiny`, so the process will run as the `shiny` user. Note that, while the process will run as the defined user, it may not behave in exactly the same way as interactively logging in as that user. In particular, the supplemental groups will not be set for the user.

Be aware that upon installing Shiny Server, permissions on the file system are set to enable the `shiny` user to read and write from various directories as is necessary to allow Shiny Server to run as the `shiny` user. Running Shiny Server as another user will require that you adjust the permissions to grant this other user the necessary privileges to run Shiny Server. In particular, ensure that the user has write privileges on these paths (recursively):

- `/var/lib/shiny-server/` (or whatever custom `SHINY_DATA_DIR` setting you are using)
- `/var/log/shiny-server/` (and/or whatever other directories you use for logging)

and read privileges on these paths (recursively):

- `/srv/shiny-server/` (and/or whatever other directories you are using to host Shiny applications)
- `/opt/shiny-server/`
- `/etc/shiny-server/` (Note that you should enable **only** read access on this directory, as you likely don't want to allow your Shiny applications (which also run as `shiny`) to be able to write to your configuration or password file.)

Finally, if the directory `/tmp/shiny-server/` exists, it (and all files within it) should be owned by the user you specify.

### 2.3.3 `:AUTH_USER:`

`:AUTH_USER:` is an additional special case of `run_as` available in Shiny Server Pro. `:AUTH_USER:` instructs Shiny Server to run applications as whatever user the visitor to the application is logged in as. For this reason, it only works when using PAM Authentication – as that is the only authentication

scheme that can ensure that the username the visitor is logged in as is a real user on the server which would be capable of running Shiny applications.

If the user is not logged in, this setting will have no effect and the next user in the `run_as` chain will be used. If there are no additional users, then the user will be prompted to login.

Much like `:HOME_USER:`, `:AUTH_USER:` can be part of a list of users provided to `run_as`. Again, the first username provided will be used if it is available; if it is not, the second will be used, and so on.

So a configuration such as the following:

```
run_as :HOME_USER: :AUTH_USER: shiny;
```

would indicate that the following priorities should be used:

1. If in the `user_dirs` hosting mode (the only mode which respects `:HOME_USER:`), then `:HOME_USER:` should be used to determine which user to run the applications as.
2. Otherwise, `:AUTH_USER:` should be used to determine who to run the applications as. Thus, if the visitor is logged in as `tim`, then the application will be started as the user `tim`. If the user is not logged in, then the process will proceed to the next option.
3. Finally, if neither of the other two could be used (i.e. we are not in the `user_dirs` hosting mode and the visitor is not logged in), then the application will be run as the `shiny` user.

Alternatively, in a configuration like the following:

```
run_as :AUTH_USER:;
```

We provide no "fallback" user in this case, so if the visitor is not logged in, no process will be started and the visitor will be prompted to login. Once logged in, Shiny Server Pro will start a process based on whom the visitor logged in as.

Note that only one "fallback" user should be provided in the `run_as` configuration. It would be incorrect to use a configuration such as:

```
run_as user1 user2;
```

as both of these users are regular, non-special users. In this configuration, `user2` would never be used, as the program would emit an error if `user1` was not a valid user on the system.

There is a known issue at the time of writing in which combining `:AUTH_USER:` with a Utilization Scheduler will create a new scheduler for each user. This means that with the following configuration:

```
run_as :AUTH_USER:
utilization_scheduler 3 .5 2;
```

each user will be permitted to spawn two processes.

## 2.4 PAM Sessions

Shiny Server Professional uses PAM (Pluggable Authentication Modules) for user authentication as well to establish the resources available for R sessions. Binding resources (and limits on their use) to Shiny sessions is accomplished by calling the PAM session API. This section explains how to configure and customize PAM sessions with Shiny Server.

### 2.4.1 Session Profile

Shiny Server supports PAM Authentication, in addition to PAM sessions. PAM Authentication is described in the linked section and is used to determine the constraints around when users should be allowed to log in. PAM sessions, however, are used to set the constraints and limitations on the Shiny process that a user may spawn.

The PAM directive that enables authentication with `root` privilege only (`auth sufficient pam_rootok.so`) needs to be present in the PAM profile. The behavior that Shiny Server requires is essentially the same as that of the `su` command (impersonation of a user without a password). Therefore, by default Shiny Server uses the `/etc/pam.d/su` profile for running Shiny sessions.

#### 2.4.1.1 Creating a Custom Profile

The `/etc/pam.d/su` profile has different default behavior depending upon your version of Linux and local configuration. Depending upon what type of behavior you want associated with Shiny sessions (e.g. mounting of disks, setting of environment variables, enforcing of resource limits, etc.), you will likely want to create a custom profile for Shiny sessions. For example, if you wanted to use a profile named `shiny-session`, you would add this to the configuration file:

```
pam_sessions_profile shiny-session;
```

Here is what the custom profile might contain in order to enable a few common features of PAM sessions (this is based on a modified version of the default `su` profile on Ubuntu):

*/etc/pam.d/shiny-session*

```
# This allows root to su without passwords (this is required)
auth       sufficient pam_rootok.so

# This module parses environment configuration file(s)
# and also allows you to use an extended config
# file /etc/security/pam_env.conf.
# parsing /etc/environment needs "readenv=1"
session    required   pam_env.so readenv=1

# Locale variables are also kept in /etc/default/locale in etch
# reading this file *in addition to /etc/environment* does not hurt
session    required   pam_env.so readenv=1 envfile=/etc/default/locale
```

```
# Enforces user limits defined in /etc/security/limits.conf
session    required   pam_limits.so

# The standard Unix authentication modules
@include common-auth
@include common-account
@include common-session
```

The above serves as a good default session profile for Shiny Server. If you want to learn more about PAM profile configuration, the following are good resources:

- http://www.linux-pam.org/Linux-PAM-html/Linux-PAM_SAG.html
- http://linux.die.net/man/8/pam.d
- http://www.linuxjournal.com/article/2120
- http://www.informit.com/articles/article.aspx?p=20968

## 2.5 r_path

The `r_path` setting can be used to specify which version of R should be used in a particular context. Shiny applications may be written with dependencies that require a particular version of R. In such cases, multiple versions of R could be installed on a server and `r_path` could be used to define which applications should use which version of R.

The following example demonstrates a potential use of this setting.

```
run_as shiny;

# In general, use the R 3.0 binary provided below.
r_path "/opt/R-3.0/bin/R";

server {
  listen 3838;

  location / {
    site_dir /srv/shiny-server;
    log_dir /var/log/shiny-server;

    location /oldApp {
      # For this particular location, use an older version of R.
      r_path "/opt/R-2.14/bin/R";
    }

  }
}
```

The value of `r_path` should specify the absolute path to the R executable.

## 2.6 Local App Configurations

In the global configuration file, the `allow_app_override` setting can be specified to enable local app configurations. If present, this setting enables owners of Shiny applications to customize the properties of their own applications using a file named `.shiny_app.conf`. This file can be placed within an application directory (alongside the `server.R` and `ui.R` files) and can contain settings that configure how Shiny Server should manage the application.

This behavior is controlled by the `allow_app_override` setting, which is disabled by default. In order to enable local application configurations, you can add to your configuration file `allow_app_override true;` (or just `allow_app_override;` for short).

Server administrators should be mindful that allowing application owners to specify their own scheduler parmeters has potential performance implications, as an application owner could enable his or her application to consume more resources than is desired. It is thus recommended that write privileges on the `.shiny_app.conf` files be granted judiciously in the underlying filesystem.

## 2.7 Hosting Model

There are currently four different methods of configuring a location, three of which serve Shiny applications. These three are described below; see the following section on Redirecting to learn about the fourth mode.

### 2.7.1 Host a Directory of Applications

A location that uses `site_dir` will host an entire directory tree – both Shiny Apps and static assets. This is the location used to serve an asset or application in `/srv/shiny-server/` in the default configuration file for Shiny Server:

```
# Define the location '/'
location / {
  site_dir /srv/shiny-server/
}
```

The above configuration instructs Shiny Server to make the `/srv/shiny-server/` directory available at the base URL (/). Any Shiny applications stored in this directory (or its subdirectories), along with any static assets (including images, data, JavaScript/CSS files, etc.), will be made available at the corresponding URL. For example, see the following directory tree:

```
+---/srv/shiny-server
|   +---shinyApp1
|       +---server.R
|       +---ui.R
|   +---shinyApp2
|       +---server.R
|       +---ui.R
```

```
|   +---assets
|       +---style.css
|       +---script.js
```

If this server were available at `http://server.com`, the `location` settings above would make the following publicly available to the user, along with any other file in the tree:

| URL | Definition |
|---|---|
| `http://server.com/shinyApp1` | Serve the Shiny App defined in 'shinyApp1' |
| `http://server.com/shinyApp2` | Serve the Shiny App defined in 'shinyApp2' |
| `http://server.com/assets/style.css` | Serve this static CSS file |
| `http://server.com/assets/script.js` | Serve this static JS file |

### 2.7.2 Host a Single Application

A location configured to use `app_dir` will instruct Shiny Server to attempt to serve a single application hosted at the given directory. For instance,

```
# Define the location '/specialApp'
location /specialApp {
  app_dir /srv/shiny-server/myApp;
}
```

configures the `location` responsible for the path `/specialApp` to use this `app_dir` router, which serves a single application stored in the directory `/srv/shiny-server/myApp`. This configuration assumes a `server.R` file is available at `/srv/shiny-server/myApp/server.R`, along with a corresponding `ui.R` file. If so, the application saved there would be available at the URL `http://server.com/specialApp`.

### 2.7.3 Host Per-User Application Directories

A location configured to use `user_dirs` will allow users on the system to create and manage their own Shiny applications and make them available in their home directories. This directive will host any application stored in an eligible user's `~/ShinyApps` directory publicly at a URL prefaced by their username.

This privilege can be restricted only to users of particular groups using the `members_of` restriction. For instance, the following configuration:

```
run_as :HOME_USER:;

# Define the root location
location / {
  user_dirs;
```

```
  # Only allow members of the 'shinyUsers' group to host personal applications.
  members_of shinyUsers;
}
```

will, for any user who is a member of the `shinyUsers` group, publicly host any Shiny application available in the user's `~/ShinyApps` directory. For instance, if a user named `tina` who is a member of the `shinyUsers` group, has `/home/tina` as a home directory, and has an application called `shinyApp1` in `/home/tina/ShinyApps/`, that application would be available on this server at the URL `http://server.com/tina/shinyApp1`. Any other application in `/home/tina/ShinyApps/` would also be publically available at a similar URL.

The `user_dirs` setting will ultimately replace the `user_apps` setting. The two models are very similar, but differ in how they handle `run_as` settings. `user_apps` ignores the `run_as` setting and always runs applications as the user in whose home directory they exist. `user_dirs`, on the other hand, will respect the `run_as` setting. You can use the special `run_as` user `:HOME_USER:` to make `user_dirs` operate the same way as `user_apps` does; thus, the two following configurations would have an identical effect:

```
location / {
  user_apps;
}
```

```
location / {
  run_as :HOME_USER:;
  user_dirs;
}
```

Note that, while Shiny Server supports the ability to run in `user_dirs` as a single user, this may require some tweaking to make it function correctly on your server. For instance, the following configuration:

```
location / {
  run_as shiny;
  user_dirs;
}
```

would attempt to start the applications housed in your users' home directories as the `shiny` user. However, many distributions restrict access to users' private `/home` directories by default, meaning that the `shiny` user would not be able to read or write to the directories it needs to in order to run these applications properly.

## 2.8 Redirecting

The final mode in which a location can operate will redirect to another URL. Such locations will immediately send a response to the client informing them of the URL to which they should redirect,

and the status code that should be used when informing the client of the redirection. Typically, a redirect will use the `301` status code for a permanent redirect, or a `302` status code for a temporary redirect. The final option when configuring a location for redirection is whether or not it should use exact matching. If a redirecting location is configured to use exact matching, only requests for that exact URL will be redirected. If not, any requests for that URL path or any subpath of that URL will be redirected. For example,

```
# Define a location at the base URL of this 'server'
location / {
  # Redirect traffic from '/shinyApp1/' to 'http://server.com' temporarily.
  location /shinyApp1 {
    redirect "http://server.com" 302 true;
  }
}
```

will redirect any requests for the exact path `/shinyApp1` to `http://server.com` temporarily.

## 2.9 Virtual Hosts

The `server_name` setting allows Shiny Server to route incoming requests to a particular `server` element based on the hostname of the request. This will serve as an additional level of filtering on incoming traffic. Bear in mind that traffic must be destined for this server (i.e., traffic that was bound for the IP address and port on which this server was listening) in order to be evaluated for a matching hostname.

For example:

```
server {
  # Instruct this server to listen on port 80
  listen 80;

  # Only accept requests for the hostname 'server1.com'
  server_name server1.com;

  # Define the location for this server.
  location / {
    site_dir /srv/shiny-server1;
    log_dir /var/log/shiny-server1;
  }
}

server {
  # Instruct this server to listen on port 80
  listen 80;

  # Only accept requests for the hostname 'server2.com'
```

```
  server_name server2.com;

  # Define the location for this server.
  location / {
    site_dir /srv/shiny-server2;
    log_dir /var/log/shiny-server2;
  }
}
```

This example presupposes that, on this network, the domains `server1.com` and `server2.com` both resolve to the same IP address. In the configuration above, we first create a server that listens on port `80` and will only accept traffic whose hostname matches `server1.com`. We then configure a second server that also listens on port `80`, and require that it only accept traffic whose hostname equals `server2.com`.

This configuration would allow an application stored in `/srv/shiny-server1/shinyApp` to be accessible at `http://server1.com/shinyApp` and an application stored in `/srv/shiny-server2/shinyApp` to be accessible at `http://server2.com/shinyApp`.

## 2.10 Custom Templates

Shiny Server can utilize custom templates when generating static pages such as directory indexes and error pages. This feature is controlled using the `template_dir` directive and can be applied globally, or to a particular `server` or `location`.

To utilize this feature, you will need a dedicated directory to store your templates; in this example, we assume you are using `/etc/shiny-server/templates/`. Inside this directory, you will place the handlebars-enabled HTML templates for your pages.

When a static page is requested, Shiny Server will attempt to respond with the appropriate template. For instance, if it encountered a 404 error, it will first look to see if you provided a specific template for handling 404 errors in the specified directory (`error-404.html`). If it does not find this file, it will look for a generic error template (`error.html`). If it cannot find a suitable template in your template directory, it will fall back on the default templates that are provided with Shiny Server (which are stored in `/opt/shiny-server/templates`).

When naming your templates, you must conform to the specific hyphen-delimited pattern that Shiny Server uses for file names. All error pages start with `error`, but the file name for a 404 error page would add an additional element (`404`) following a hyphen: `error-404`. All files are expected to use the `.html` extension.

The tables below describe the page templates that can be defined in Shiny Server:

| Template Name | Description |
|---|---|
| error-403 | Client is forbidden from accessing this page. Or, in Shiny Server Pro, that the user is signed in |
| error-404 | Page not found |
| error-500 | General server error |
| error-503-users | The application has exceeded its maximum # of users |
| error-503-license | (Pro Only) You have exceeded the number of concurrent users allotted for your license. |

| Template Name | Description |
|---|---|
| directoryIndex | Shown when `directory_index` is enabled for a location and the user visits the base URL. |
| login | (Pro Only) Used to render the login page. |

Again, any of the template names that have hyphens in them can be generalized by trimming off the hyphen and subsequent word(s). For instance, a file named `error-503.html` could be used as the template for *both* `error-503-users` pages, as well as `error-503-license` pages. More generally, a single template can be used for all errors by naming it `error.html`. If a more specific template does not exist, this page would be used when generating a page for any type of error.

We recommend starting with the existing templates and modifying them to create your own branding. To do this, copy the relevant HTML files out of `/opt/shiny-server/templates/` into your template directory (for example, `/etc/shiny-server/templates/`). For example, to create a custom error page, execute:

```
$ mkdir /etc/shiny-server/templates/
$ cp /opt/shiny-server/templates/error.html /etc/shiny-server/templates/
```

Now open the new template in your preferred text editor and modify it. Initially, leave the overall structure intact, but add some language at the top of the page (immediately under the line containing `<h1>{{title}}</h1>`), or add some new CSS rules, and save the file.

Now instruct Shiny Server to use this new template in your configuration by adding the line:

```
template_dir /etc/shiny-server/templates;
```

at the top/bottom of the file so it applies globally. Of course, you can later specify another `template_dir` for particular `location`s if you wish.

When you restart the server and access an error page, you should see your new text or CSS styling on the page that is loaded. (Try accessing a hidden file like `http://server.com:3838/.hidden`, where `server.com` is the domain name or IP address of your server, which should use the `error-403` or `error` template.) Likewise, you can copy the `directoryIndex.html` template or, for Shiny Server Pro, the `login.html` template to your custom directory and begin customizing.

For performance reasons, templates are cached in memory once they are loaded. Thus, you will find that changes to your template may not be immediately visible when you refresh the page. To empty the cache and force Shiny Server to load in the new changes to your template, you can either `restart` or `reload` the server (see the section on Stopping and Starting).

Finally, note that these templates currently only apply for HTML pages that are generated by Shiny Server. An HTML page generated by the Shiny package will *not* have custom templates applied at this time. Currently, this is most visible with 404 errors. If you access a URL which Shiny Server cannot direct to a Shiny application (for instance, `http://server.com:3838/flargdarg`, where `server.com` is the domain name or IP address of your server), you will see a Shiny-Server-generated 404 error page which has custom templates applied (assuming this `server` is configured to have a custom `error-404.html` or `error.html` template). However, if you were to access a file that does not exist in a valid application, you would see an un-styled 404 page that did not leverage your custom

template. For instance, if you have an application deployed at `http://server.com:3838/myApp` and you attempt to access `http://server.com:3838/myApp/flargdarg`, the request would be sent to Shiny which would then look for this file and, if it is not found, would return a 404 page. Shiny Server would pass this response back directly to the client, since it was not generated by Shiny Server. We will be reviewing this architecture in future versions – please let us know if you find that the current organization inhibits your workflow.

## 2.11 Set Custom Headers

Shiny Server Pro offers the `set_header` configuration directive to allow you to configure additional HTTP headers that will be included in all HTTP responses that the server generates. You can use this option to specify things like caching behaviors, HTTP Strict Transport Security (HSTS), or even a limited version of CORS support.

To use this option, include a line like the following in your configuration file:

```
set_header "Strict-Transport-Security" "max-age=36000";
```

which will add an HTTP header called `Strict-Transport-Security` with a value of `max-age=36000` to every HTTP response Shiny Server Pro sends.

Headers may be overwritten by Shiny Server Pro if they conflict with a header that would otherwise have been set. For instance, an entry of `set_header "Content-Type" "somevalue"` would generally have no effect as it would always be overwritten by Shiny Server Pro.

## 2.12 Server Log

All information related to Shiny Server itself, rather than a particular Shiny application, is logged in the global system log stored in `/var/log/shiny-server.log`. This log should be checked often to ensure Shiny Server is performing as expected. Any errors and warnings that Shiny Server needs to communicate will be written here.

If `logrotate` is available when Shiny Server is installed, a `logrotate` configuration will be installed. The default configuration is to rotate the logfile when it exceeds 1MB in size. The old log file will be compressed and stored alongside the original log file with a `.1.gz` extension (then `.2.gz`, etc.). Up to twelve archived log files will be maintained; upon the thirteenth log rotation, the oldest log file will be deleted.

For logs related to individual Shiny Applications, see the section on Logging and Analytics.

### 2.12.1 Access Logs

An access log can be configured globally using the `access_log` parameter. This log is not enabled by default. This setting controls the location of the access log as well as the format used. The access log can be useful to audit the security and activity taking place on your Shiny Server installation. These logs can be audited manually or automatically to inspect how often various resources are

being accessed, or by whom they are being accessed (using the originating IP address). Currently, one access log is created for the entire Shiny Server process; all Shiny applications share this access log.

The access logs will be written using the morgan logging library; additional details can be found in the "Formats" section of their documentation. In brief, there are five pre-defined logging formats for access logs specified in the documentation referenced above:

- `combined` ':remote-addr - - [:date] ":method :url HTTP/:http-version" :status :res[content-length] ":referrer" ":user-agent" '
- `common` ':remote-addr - - [:date] ":method :url HTTP/:http-version" :status :res[content-length]'
- `short` ':remote-addr - :method :url HTTP/:http-version :status :res[content-length] - :response-time ms'
- `tiny` ':method :url :status :res[content-length] - :response-time ms'
- `dev` concise output colored by response status for development use

For example, the following would configure an access log that uses the `tiny` format:

```
access_log /var/log/shiny-server/access.log tiny;

server {
  ...
```

## 2.13 Environment Variable Settings

We strive to expose most settings through the configuration file described previously. However, there are some settings that are not included in the configuration file that some users may still wish to change. Where possible, we allow users to configure these settings using environment variables. Some of these environment variables will later be promoted to configuration directives, but will be documented here until that time.

Typically, it is best to define these environment variables in the startup script used to run Shiny Server. This varies depending on which Linux distribution and version you are running.

#### 2.13.0.1 systemd (RedHat 7, Ubuntu 15.04+, SLES 12+)

File to change:

```
/etc/systemd/system/shiny-server.service
```

How to define the environment variable:

```
[Service]
Environment="SHINY_LOG_LEVEL=TRACE"
```

Commands to run for the changes to take effect:

```
$ sudo systemctl stop shiny-server
$ sudo systemctl daemon-reload
$ sudo systemctl start shiny-server
```

#### 2.13.0.2 Upstart (Ubuntu 12.04 through 14.10 and RedHat 6)

File to change:

```
/etc/init/shiny-server.conf
```

How to define the environment variable:

```
env SHINY_LOG_LEVEL=TRACE
```

Commands to run for the changes to take effect:

```
$ sudo stop shiny-server
$ sudo start shiny-server
```

#### 2.13.0.3 init.d (RedHat 5, SLES 11)

File to change:

```
/etc/init.d/shiny-server
```

How to define the environment variable:

```
export SHINY_LOG_LEVEL=TRACE
```

Commands to run for the changes to take effect:

```
$ sudo /sbin/service shiny-server restart
```

### 2.13.1 SHINY_LOG_LEVEL

Defines the verbosity of logging which will be used by Shiny Server. Valid options – in order of decreasing verbosity – are `TRACE`, `DEBUG`, `INFO`, `WARN`, and `ERROR`. More verbose levels of logging (such as `TRACE`) may be helpful when debugging an issue or trying to understand exactly what Shiny Server is doing, but will likely be far too much information for a system with even moderate load.

The default if this environment variable is not defined explicitly is `INFO`.

### 2.13.2 R

Defines the path to the R executable that should be used when running Shiny. Systems that have multiple versions of R installed, or do not have R on the path before starting Shiny Server, can use this setting to point to a particular version of R.

Note that, for any `location` with a configured `r_path` in the configuration file, this environment variable will not take effect. If no environment variable is found, Shiny Server will expect an executable named `R` to be on the path.

### 2.13.3 SHINY_DATA_DIR

The historical databases of Shiny applications used in the Admin dashboard can consume multiple gigabytes of disk space on a system hosting many applications. This environment variable allows you to define the directory in which Shiny Server should persist data to disk. To estimate the amount of disk space required, we recommend reserving 10MB in this directory for every Shiny application you plan to host on a server. Additionally, ensure that this directory is writable by the `shiny` user, or whichever user you are running Shiny Server as (as is discussed in detail in the `run_as` section).

Note that Shiny Server expects a directory named `monitor/rrd/` to already exist inside `SHINY_DATA_DIR` when it starts.

By default, Shiny Server will create and store data in `/var/lib/shiny-server/`.

# Chapter 3

# Deploying Applications

## 3.1 Schedulers & Application Restarts

A scheduler is responsible for fulfilling incoming requests to a particular application. Each version of an application (see below) will have its own associated scheduler. Each scheduler can have different properties to control things like how many concurrent connections it should accept.

A scheduler can be specified at many locations in the configuration file and will be inherited by inner blocks. For instance, a scheduler definition found in a `server` block will be applied to every `location` in that `server`, unless overridden.

### 3.1.1 Restarting an Application

Some changes that you make to the code, assets, or environment of an application will require the application's R processes to be restarted for the changes to take effect. These include upgrades to packages that are used by the application, changes to `.Renviron`/`.Rprofile` or other R source files, or modifications to data files that are read-only at startup time.

Fortunately, Shiny applications generally do not need to be restarted when changes to `ui.R` or `server.R` are made, as Shiny will check for changes to these files on page load.

An application can be restarted by altering the "modified time" on a file named `restart.txt` in the application's directory. This can most easily be done using the `touch` utility, as in `touch restart.txt`, which will update the modified timestamp on this file to the current time. Upon the next new connection to the application, Shiny Server will spawn a new R process to run the "new" (restarted) Shiny Application for this and future users. When this occurs, the old processes will remain unaltered, and open connections will remain active and valid until the last connection closes itself.

This could have unintuitive consequences regarding the number of processes that may be running for a given application. Even if using a single-process Simple Scheduler for an application, it is possible to have multiple R processes associated with this application, each corresponding to a separate restart timestamp. This behavior is likely to change in future versions of Shiny Server.

### 3.1.2 Simple Scheduler

The Simple Scheduler is the only scheduler available in the Open Source edition of Shiny Server. It associates a single R process with a single Shiny application. This scheduler accepts a single parameter which specifies the maximum number of concurrent sessions. Once this number is reached, users attempting to create a new session on this application will receive a 503 error page.

```
# Define a new location at the base URL
location / {
  # Define the scheduler to use for this location
  simple_scheduler 15;

  ...
}
```

In this example, the `location` defined at the base URL in this `server` will be configured to use the Simple Scheduler and to limit each application to a maximum of 15 simultaneous connections – meaning that if there are already 15 active connections on the application when a new user attempts to visit this application, that user will receive a 503 error page. When one of the 15 active connections is disconnected, one new "seat" will be available for another user to connect to the application.

### 3.1.3 Utilization Scheduler

The Utilization Scheduler is a more sophisticated scheduler available in Shiny Server Professional that allows a single Shiny application to be powered by multiple R Shiny processes. Incoming traffic will be routed to the R process associated with this Shiny application that has the fewest current connections.

The Utilization Scheduler is configured by three parameters.

- `maxRequestsPerProc` – The number of concurrent connections that a single R process can support before it will begin to return 503 errors for new sessions. The default is 20.
- `loadFactor` – The "trigger percentage" (on a scale from 0 to 1) at which a new R process should be spawned. Once session capacity reaches this percentage, a new process will be spawned (unless the `maxProc` limit has been reached). The default is 0.9.
- `maxProc` – The maximum number of processes associated with this Shiny application that should exist concurrently. Note that this does not include restarted versions of the same application in its count. The default is 3.

As an example:

```
location /shinyApp1 {
  # Define the scheduler to use for this application
  utilization_scheduler 5 0.5 3;
```

```
  ...
}
```

The above configuration would create a Utilization Scheduler for this application that supports a maximum of 3 processes, a maximum of 5 connections per process, and a load factor of 0.5. Initially, only one process would be created to facilitate the first and second concurrent connection to this application. The third connection, however, would surpass the load factor ($3/5 > 0.5$), so a new R process would be spawned. The next request would then be routed to this new process. This pattern would continue until the load factor was again surpassed, at which point the third process would be spawned. With a maximum of 3 processes, no more would be created after the third. So the 16th concurrent connection would be turned away with a 503 error.

Be aware that, upon the restart of an application using a Utilization Scheduler, the resources allotted to this application will effectively double for a time. For instance, if an application is configured to use 4 `maxProc` and has sufficient load to create and maintain that many R processes, a restart of the application will instruct Shiny Server to create an R process for a "new" Shiny application that is also eligible to run up to 4 Shiny processes in parallel. Thus, with one restart and sufficient traffic to this application, it is possible that it could be running 8 Shiny processes. This behavior is likely to change in future versions of Shiny Server.

## 3.2 R Markdown

In addition to serving traditional Shiny applications out of a directory (a `server.R` file and an associated UI), Shiny Server now supports interactive R Markdown documents. To take advantage of this capability, you will need to make sure that R Markdown is installed and available for all users. You can do so by running the following command:

```
sudo su - -c "R -e \"install.packages('rmarkdown')\""
```

If a hosted directory does not include a `server.R` file, Shiny Server will look to see if it contains any `.Rmd` files. If so, Shiny Server will host that directory in "R Markdown" mode using `rmarkdown::run`.

Particular `Rmd` files can be accessed by referencing their full path including the filename, e.g., `http://myserver.org/mydocs/hello.Rmd`. If a request is made to a directory rather than to a particular `Rmd` file, Shiny Server will attempt to serve the file `index.Rmd`. If that file does not exist, the user will get an error alerting them that the file is not available.

## 3.3 Application Timeouts

Each Shiny Application has two timeouts associated with it:

(1) **`app_init_timeout`** – Describes the amount of time (in seconds) to wait for an application to start. After the specified number of seconds has elapsed, if the R process still has not become responsive, it will be deemed an unsuccessful startup and the connection will be closed. The default value for `app_init_timeout` is 60 seconds.

(2) `app_idle_timeout` – Defines the amount of time (in seconds) an R process with no active connections should remain open. After the last connection disconnects from an R process, this timer will start and, after the specified number of seconds, if no new connections have been created, the R process will be killed. The default value for `app_idle_timeout` is 5 seconds.

Typically, these two parameters will be correlated. Shiny Applications that involve little processing to start (therefore have a small `app_init_timeout`) can often be closed with minimal concern (and thus would have a small `app_idle_timeout`). Conversely, applications that require a substantial amount of data to be loaded on startup may merit a longer `app_init_timeout` to give the data time to load, and a longer `app_idle_timeout` as the task of spawning a new process is more expensive and should be minimized.

## 3.4 Session Timeouts

The `app_session_timeout` setting can be used to disconnect idle Shiny connections automatically. The setting accepts a single value that dictates the number of seconds after which an idle session will be disconnected. Here "idleness" is measured by a connection's interaction with the server. Incoming or outgoing data will reset the countdown time for each user's session. If the number of seconds specified elapses without any data being sent to or from the client, the user's session will be disconnected. The default value for `app_session_timeout` is 0, which means that sessions will never be automatically disconnected.

This parameter can be configured globally, or for a particular `server` or `location`. It can be used to ensure that stale R processes are eventually closed by terminating sessions after they become idle for some period of time.

Session timeouts are configurable on a per-server or per-application basis. Controlling the session timeout can be important in helping the administrator manage the resource allocation for a given application, and in freeing up resources from idle connections that will then be available to newly created sessions.

## 3.5 Logging and Analytics

### 3.5.1 Application Error Logs

Error logs are created for each R Shiny process separately. For `location`s configured to use `user_apps` or `user_dirs`, these logs are created in each user's `~/ShinyApps/log/` directory. For `location`s configured with `app_dir` or `site_dir`, the directory in which these logs are created is managed by the `log_dir` setting, which can be specified globally, for a particular `server`, or for a particular `location`. By default, these logs will be located at `/var/log/shiny-server/`. To change the directory at the `location` level, the `log_dir` can be specified in the following way:

```
location / {
  log_dir /var/log/shiny-server/;
}
```

The log files will be created in the following format:

```
<application directory name>-YYYYMMDD-HHmmss-<port number or socket ID>.log
```

A log file will be created for each R process when it is started. However, if a process closes successfully, the error log associated with that process will be automatically deleted. The only error log files that will remain on disk are those associated with R processes that did not exit as expected.

You can override this behavior using the `preserve_logs` configuration option. If you set `preserve_logs true;` in your configuration file, Shiny Server will *never* delete the logs from your R processes, regardless of their exit code. Be aware that this will cause log files to accumulate very quickly on a busy server. This setting is only recommended for debugging purposes; if it were to be enabled on a production server, you would need to pay close attention to the rotation and archiving of logs to prevent your file system becoming overwhelmed with log files.

If you are looking for log messages related to Shiny Server itself, rather than individual Shiny applications, see the section on the Server Log.

### 3.5.2 Log File Permissions

By default, application log files are created with mode `0640`, which prohibits users other than `shiny` and `root` from accessing them.

The default log file mode can be customized with the `log_file_mode` directive, which – like `log_dir` – can be specified globally, for a particular `server`, or for a particular `location`. To modify log visibility at the `location` level, making logs accessible to all users on the system, the `log_file_mode` can be specified in the following way:

```
location / {
  log_file_mode 0644;
}
```

Note that the `log_file_mode` value must begin with a zero because it is an octal number.

### 3.5.3 Google Analytics

Shiny Server is capable of automatically inserting the necessary JavaScript code to enable Google Analytics tracking globally, or for a particular `server` or `location`. This is managed by the `google_analytics_id` setting, which can be used as follows:

```
location / {
  google_analytics_id UA-12345-1;
}
```

## 3.6 Program Supervisors

You may also wish to run R sessions under a program supervisor that modifies their environment or available resources. You can specify a supervisor (and the arguments that control its behavior) using the `exec_supervisor` setting. For example:

```
...

location /low {

  ...

  exec_supervisor "nice -n 10";
}
```

This example uses the `nice` command to run the Shiny processes in the affected location with a lower scheduling priority. See http://linux.die.net/man/1/nice for more details on `nice`. Note that for `nice`, in particular, it is possible to accomplish the same thing using the `pam_limits` module (and even specify a custom priority level per user or group).

It is important to note that the `exec_supervisor` will be executed as the same user the Shiny app will be running as. So if Shiny Server is running as `root` but runs applications as `shiny`, the `exec_supervisor` will be executed as `shiny`. Because most systems only allow the `root` user to assign a negative (higher priority) `nice` privilege, this means that exec_supervisor would not be able to assign a negative nice value without reconfiguring the server to allow certain users to assign negative `nice` values to a process.

## 3.7 Reactivity Log

The reactivity log is a browser-based tool for analyzing and debugging reactive dependencies in a Shiny Application. We strongly recommend that you read the Overview of Reactivity in the Shiny tutorial to get the most out of the reactivity log. The symbols and representations introduced in that article are reused in the log. This tool should not be used in a production environment, as it exposes details that should not be publicly available. Many application authors find it to be a useful tool to use locally, however.

Currently, reactivity logs are maintained for an entire R process. Therefore, this feature should be used in a fresh R session in which only one Shiny application is active. To enable the reactivity log, run the command `options(shiny.reactlog=TRUE)` before running your application. Once you've started your application and are viewing it in a browser, you can use `Ctrl+F3` (or `Command+F3` for Mac users) to view the reactivity log. This visualization creates a snapshot of all reactive activity that had occurred up to the moment it was created. If you want to view the reactive activity that has occurred since opening the reactivity log, you must refresh the browser.

The `showReactLog()` function exposes this feature from within R, and can be used outside of strictly Shiny contexts to generate a static HTML file visualizing reactivity. You can see the full documentation using `?showReactLog` in R.

Since the reactivity log exposes the architecture and some of the code behind your application, it is inadvisable to enable this feature in a production environment. This functionality is not enabled by default in Shiny Server. We recomend using the reactivity log in a local R process not managed by Shiny Server.

## 3.8 Specifying Protocols

Shiny Server provides a wide variety of techniques to keep the data in the web browser synchronized. The preferred technique, and the one most widely used, is the use of WebSockets. However, if WebSockets are not supported – either by some intermediate network between Shiny Server and your client, or by your client's web browser – then a fallback protocol will be used. In total, Shiny Server provides nine different methods of connecting to the server in real-time. In order of preference, they are:

1. WebSocket
2. XDR Streaming
3. XHR Streaming
4. iframe Eventsource
5. iframe HTML File
6. XDR Pollilng
7. XHR Polling
8. iframe XHR Polling
9. JSONP Polling

Each of these methods will be tried by the client in the above order until a successful connection to Shiny Server is established. If you would like to omit one or more of these protocols from the list, you can currently do so only from the client (we hope to make a server-side configuration available in a future release of Shiny Server).

To change the available protocols from the client, open a Shiny Application and press the keyboard shortcut: `Ctrl+Alt+Shift+A` (or, from a Mac: `control+option+shift+A`). This will open a window that will allow you to select or deselect any of the above protocols. After you confirm the changes, these settings will be saved in your browser for future visits to this server. These settings will take effect upon loading any Shiny application hosted on this domain, and will last until you explicitly change them again; they will only have an effect on the browser in which this action was performed.

### 3.8.1 Disabling WebSockets on the Server

Shiny Server uses SockJS to facilitate communication between the browser and the server. On most networks, SockJS will be able to resolve a functioning protocol that will allow it to communicate reliably between the browser and Shiny Server. Some environments, however, will perform some

level of filtering or proxying that hinders one or more of the available protocols. Older networking equipment will likely not function well with WebSockets, for instance.

Most users will not need to worry about what protocol their network will support, as SockJS will gracefully determine the best protocol when trying to establish the connection. However, some environments have found consistent problems with having particular protocols enabled, and find that a particular protocol or set of protocols are most reliable in their environment. In this case, you can use the `disable_protocols` setting to disable any of the available protocols.

For instance, if you find that WebSockets and XDR polling are problematic on your networking equipment, you could add the following configuration to disable them:

```
disable_protocols websocket xdr-polling;
```

The protocol names to be used when disabling are as follows:

```
websocket
xdr-streaming
xhr-streaming
iframe-eventsource
iframe-htmlfile
xdr-polling
xhr-polling
iframe-xhr-polling
jsonp-polling
```

# Chapter 4

# Authentication & Security

## 4.1 Authentication Overview

Shiny Server Professional offers the ability to authenticate individual users. By specifying authentication requirements on particular `server`s or `location`s, the administrator can control the set of applications particular users are allowed to access.

Authentication can be implemented by integrating into an existing LDAP or Active Directory database, relying on Google accounts, or by using a "flat-file" authentication system that is contained in a local file. To ensure that your users' passwords are being protected, it is **strongly encouraged** that any Shiny application requiring authentication use SSL to encrypt the usernames and passwords being transmitted.

The current user's username and groups (where applicable) will be available in Shiny Server Pro (version 0.8 and later) in the `session` parameter of the `shinyServer` function. For instance:

```
shinyServer(function(input, output, session) {
  output$username <- reactive({
    session$user
  })

  output$groups <- reactive({
    session$groups
  })
})
```

It is important to note that when a user logs in to Shiny Server Pro, that authentication will be applied server-wide. This means that a visitor who authenticates as a particular user on one application will be identified as that user to the other applications on this server to which they connect. There is currently no notion in Shiny Server Pro of logging in exclusively to only one particular application.

## 4.2 Auth Duration

Regardless of which authentication mechanism is used, the duration of authentication can be configured using the `auth_duration` setting. This setting controls the amount of time (in minutes) for which a user should remain logged in after they stop using their last application. This setting should be set to 10 minutes or greater.

## 4.3 Flat-File Authentication

Shiny Server Pro offers flat-file authentication as a simple and easy-to-configure authentication mechanism. This method is self-contained and not integrated into either the system's user/password database, nor any Enterprise authentication mechanism. Thus, usernames and passwords must be created explicitly for each user that should exist in Shiny Server.

The storage of usernames and passwords is handled by a single file that can be specified using the `auth_passwd_file` setting as follows:

```
run_as shiny;

auth_passwd_file /etc/shiny-server/passwd;

server {
  location / {
    ...
```

This will instruct Shiny Server to look up all usernames and passwords in the file stored at `/etc/shiny-server/passwd`. This file should have zero or more lines in the format `username:{scrypt-hashed-password}`. The scrypt encryption algorithm is used to protect users' passwords from theft; the hashed passwords in this file are expected to be in base64 format. We provide the sspasswd utility with Shiny Server Professional; this utility vastly simplifies the process of managing these "sspasswd" files.

You should think carefully before adjusting the permissions on this file. For instance, if you were to enable the `shiny` user to write to your password database, any Shiny application running as `shiny` (the default) would now be able to modify your password database. Because the passwords are securely hashed, granting `shiny` *read* access to this file is not problematic and, in fact, is enabled by default.

### 4.3.1 sspasswd

The `sspasswd` utility comes with Shiny Server Professional, and can be used to manage the username/password file. By default, it is not made available on the `PATH`, but you can find it in `opt/shiny-server/bin/`. The general pattern for the utility is to provide the file to use for storage followed by a username, as in:

```
$ sudo /opt/shiny-server/bin/sspasswd /etc/shiny-server/passwd tina
```

To create a (or overwrite an existing) password file, use the `-c` switch. The default behavior will be to add the username specified (`tina`, in the example above) to the file after prompting the user for a password (or reading it from `stdin`). To delete a user from the file, use the `-D` switch. User/password combinations can be verified using the `-v` switch. Finally, the `-C` switch will set the maximum amount of time (in seconds) for encryption. The larger this value is, the more secure the password hash will be.

Currently, the '`:`', '`$`' and newline ('`\r`' or '`\n`') characters are prohibited in usernames.

## 4.4 Google Authentication

Shiny Server Pro supports the ability to use Google for the management of your users via the `auth_google` setting. In this environment, users attempting to log in to your Shiny Server would be presented with a button asking them to log in via Google. If they have been granted access to your server, they would be returned to your Shiny Server presenting the email address associated with their Google account. At that point, this email address can be used to grant or restrict access to individual `application`s and `location`s. This section will show you how to setup this type of Google Authentication for Shiny Server.

### 4.4.1 Create a Google Application

Before you can use Google Authentication on your server, you must register an "application" with Google. This is the application that your users will be asked to "connect" to when logging in to your Shiny Server. To do this, you can visit Google's Developer's Console, which, at the time of writing, is available at https://console.developers.google.com/.

In the Developer's Console, you must create a project, then give it a name and ID. Under "APIs & auth", you'll find the "Credentials" section. There you have the ability to "Create new Client ID"; do that now. The type of application is a "Web application". For the "Authorized JavaScript Origins", you must explicitly list any name a user might use to access your server including the domain name, port, and protocol. For instance, if you offer both HTTPS and HTTP access to your Shiny Server, you'll need to include both; if you have multiple domain names pointing to your server, you should include all of those domains; if you have servers running on multiple ports, you'll need to include them all explicitly here.

In the "Authorized redirect URI" field, you again must provide all possible ways a user may be accessing your server, but append the string `__login__/callback` (note the **double** underscores on either side of "login") to each URI.

For instance, if you have a server available at both `example.org` and `example.net`, supporting both HTTP and HTTPS, offering web services with or without a `www.` subdomain prefix, and using the standard ports (80 and 443, respectively), you would use the following origins:

- `http://example.org`
- `http://www.example.org`
- `https://example.org`

- `https://www.example.org`
- `http://example.net`
- `http://www.example.net`
- `https://example.net`
- `https://www.example.net`

And the Authorized redirect URIs would include:

- `http://example.org/__login__/callback`
- `http://www.example.org/__login__/callback`
- `https://example.org/__login__/callback`
- `https://www.example.org/__login__/callback`
- `http://example.net/__login__/callback`
- `http://www.example.net/__login__/callback`
- `https://example.net/__login__/callback`
- `https://www.example.net/__login__/callback`

You can then "Create" your Client ID.

You should see your new "Client ID for web application" displayed after you have created your new Client ID. You can inspect it to confirm that the Redirect URIs and Javascript Origins look correct (if not, you can click "Edit settings"). If they look correct, you should note the "Client ID" and "Client secret"; you'll enter these settings into your Shiny Server configuration file later.

At this point, you have created a Google application that you will be able to use to allow your users to login via Google on your Shiny Server. You can always add more details to your Google application here if you like, including an icon and more textual details.

### 4.4.2 Configure Shiny Server

Now that you have a Google application created, you can customize your Shiny Server configuration to instruct it to a) Use Google for its authentication, and b) provide your Google application's settings.

You can edit the configuration file stored at `/etc/shiny-server/shiny-server.conf` to replace any existing authentication system (if you have one) with the `auth_google` directive. This setting first takes the Client ID, then the Client Secret you created in Google earlier like so:

```
auth_google 12345.google.com ABCDEF;
```

At this point, you can restart your Shiny Server (see Stopping and Starting if you're unsure how). Now when you try to access a protected resource such as the Admin page or a location secured with a `required_user` setting, you should be prompted with a login screen asking you to log in via Google. Click that link, then decide whether or not you would like to grant access to your new application. If you do grant access and your email is included in the `required_user` field associated with the resource you are trying to access, you will be let in. Otherwise, you will still be logged in as that user, but you will not be granted access to that resource. Note that Google Authentication does not support the `required_group` setting.

We recommend that you protect your OAuth application secret from unprivileged users. If you are using `auth_google`, it is strongly encouraged that you read the following section on Securing the Client Secret.

It may be necessary to educate your users on the behavior of Single Sign On (SSO) systems like Google. Specifically, some users may not understand that logging out from a Shiny app will *not* log them out of Google. Currently, logging out of Google will also not log you out of Shiny Server. Rather, there is a one-time connection between the two systems at the moment the user logs in. From then on, their Google session and their Shiny Server session will be completely independent.

### 4.4.3 Securing the Client Secret

In addition to providing the client secret as text in the config file, you could alternatively specify the absolute path to a file containing the client secret as in the following example:

```
auth_google 12345.google.com /etc/shiny-server/ga-secret.txt;
```

In this case, Shiny Server would read the client secret from the file stored at `/etc/shiny-server/ga-secret.txt`. Additionally, if the client secret is a path to a file rather than the secret itself, Shiny Server will retain `root` privileges, allowing you to tighten the security around the file containing your client secret to allow only the `root` user to read it. For example, you could secure the above file using:

```
# Change the file to be owned by the root user.
chown root:root /etc/shiny-server/ga-secret.txt

# Set the permissions so that only root can read the file.
chmod 600 /etc/shiny-server/ga-secret.txt
```

In this way, you can secure your client secret from Shiny applications or other users who are allowed to log in to the server. If you expect that many users or any untrusted users may be logging into or deploying apps on the server on which Shiny Server is installed, we strongly recommend that you use this configuration to ensure the privacy of your secret key.

### 4.4.4 Email Address Restrictions

By default, the Google auth strategy will allow any user with a Google account to log in to your system. Though you can always use `required_user` to restrict access to a resource, some organizations wish to only allow particular users to login at all. For instance, if you were not

concerned about which individual users could access which applications, but wanted to ensure that only users within your organization can access *any* application, you may want a more broad restriction on the users you authorize. As discussed in the section on User Authentication, you can use `required_user *;` to indicate that any logged in user should be able to access a resource.

For such needs, we provide the ability to globally filter which email addresses will be authorized into your Shiny Server at all – regardless of what the individual application's `required_user` settings are. To do this, add filters onto the `auth_google` setting in your configuration file. In the simplest form, you could add one filter to require that the email address the user is using with Google be of a particular form, such as `"*@myorganization.org"`. This would allow any user who logged into Google with the email address that ended in `@myorganization.org` to log on to the server (see example below). Once logged on, the `required_user` filters would be applied as usual to control access to individual applications.

Additionally, you can provide as many filters as you like on the `auth_google` directive to indicate whether email addresses matching that filter should be admitted (`+`) or rejected (`-`). The filters will be applied one at a time in order, until the given email address matches one. The filters can be prefixed with a `+` or `-` sign to indicate whether the filter is "positive" (in which case, any user matching the filter should be admitted) or "negative" (in which case, any user matching the filter should be rejected). If no sign is provided, it is assumed that the filter is "positive". Filters will be evaluated *in the order in which they are provided*, and the first filter an email matches will be used to determine whether or not that email address should be allowed to log in to Shiny Server. If an email address matches none of the provided filters, it will be rejected.

A few examples may be helpful:

```
auth_google 12345.google.com ABCDEF "*";
```

The filter `"*"` will be interpreted as a positive filter, meaning any email address matching that filter (every possible email address) will be admitted. This is the default behavior if you do not provide any filters in the configuration file.

```
auth_google 12345.google.com ABCDEF "*@myorganization.org";
```

The filter `"*@myorganization.org"` will admit any user whose email address ends in `@myorganization.org`, and reject any email address that does not.

```
auth_google 12345.google.com ABCDEF "-jim@example.org" "*@example.org";
```

In these filters, if the user `kelly@example.org` attempted to the system, the first filter would be checked and it would be found that `kelly@example.org` does not match the filter `"jim@example.org"`, so it would continue on to the next filter. There it would be found that `kelly@example.org` *does* match the filter `*@example.org`, and because this filter is a positive filter (you could have prefixed it with a `+`, if you wished), this user would be granted access to the system. However, a user `john@gmail.com` would not match the first filter or the second filter, so he would be denied access to the system, as he did not match any positive filter. Finally, the user `jim@example.org` would match the first filter, and because that filter is a negative filter, this user would be denied access to the system.

We support a lightweight pattern matching syntax. Specifically, you can use the `*` character to match any string of length 0 or more, the `?` character will match any *one* character, and square brackets can be used to match against a set of characters such as `[a-e]` to match any character between `a` and `e`. All patterns are *not* case-sensitive.

## 4.5 PAM Authentication

Shiny Server Professional can authenticate users via the Linux standard PAM (Pluggable Authentication Module) API. PAM is typically configured by default to authenticate against the system user database (`/etc/passwd`); however, it can also be configured to authenticate against a wide variety of other systems including Active Directory and LDAP.

This section describes the PAM configuration used for authentication. Note that PAM can be used for both authentication and to tailor the environment for user sessions (PAM sessions). This section describes only authentication; see the PAM Sessions section for details on how Shiny Server can be configured to use PAM sessions.

### 4.5.1 PAM Basics

PAM profiles are located in the `/etc/pam.d` directory. Each application can have its own profile, with a default profile used for applications without one (the default profile is handled differently depending on which version of Linux you are running).

To learn more about PAM and the many options and modules available for it, see the following:

- http://en.wikipedia.org/wiki/Pluggable_authentication_module
- http://www.centos.org/docs/5/html/Deployment_Guide-en-US/ch-pam.html
- http://tldp.org/HOWTO/User-Authentication-HOWTO/x115.html
- http://linux.die.net/man/8/pam

### 4.5.2 Default PAM Configuration

#### Debian/Ubuntu

On Debian and Ubuntu systems, Shiny Server does not provide a Shiny-Server-specific PAM configuration file. Instead, Shiny Server uses the `/etc/pam.d/other` profile, which by default inherits from a set of common configuration files:

*/etc/pam.d/other*

```
@include common-auth
@include common-account
@include common-password
@include common-session
```

If the `/etc/pam.d/other` profile reflects the authentication system and policies that you would like Shiny Server to use, then no further configuration is required. If you want to create a custom PAM profile for Shiny Server, you must create a file named `/etc/pam.d/shiny-server` to specify whatever settings are appropriate.

**RedHat/CentOS/SLES**

On RedHat and CentOS systems, applications without their own PAM profiles are denied access by default. Therefore, to ensure that Shiny Server is running and available after installation, a default PAM profile is installed at `/etc/pam.d/shiny-server`. This profile is configured to require a user-id greater than 500, and to authenticate users against local system accounts:

*/etc/pam.d/shiny-server*

```
auth       requisite      pam_succeed_if.so uid >= 500 quiet
auth       required       pam_unix.so nodelay
account    required       pam_unix.so
```

This default PAM profile may not reflect the authentication behavior that you want for Shiny Server. In that case, some customization may be required. If you have already set up another PAM profile (e.g. `/etc/pam.d/login`) with the desired behavior, then it may be enough to copy that profile over the Shiny Server one. For example:

```
$ sudo cp /etc/pam.d/login /etc/pam.d/shiny-server
```

### 4.5.3 PAM and Kerberos

Shiny Server Professional supports integration with Kerberos for seamless authentication to other applications via Kerberos tickets. To enable this feature, you need to have the following defined in your `/etc/shiny-server/shiny-server.conf` file:

```
run_as :AUTH_USER:;
auth_pam true;
pam_sessions_profile shiny-session;
```

This assumes you are using a file named `shiny-session` for the pam.d session profile, but the name could be anything as long as it matches your actual filename.

You also need to customize your Shiny Server pam.d files: `shiny-server` and `shiny-session`. We provide a very simple example of these files here that uses `pam_krb5.so` as a guide.

*/etc/pam.d/shiny-server*

```
auth       sufficient     pam_krb5.so
account    required       pam_krb5.so
session    requisite      pam_krb5.so
```

*/etc/pam.d/shiny-session*

```
auth        required      pam_krb5.so
account     [default=bad success=ok user_unknown=ignore] pam_krb5.so
password    sufficient    pam_krb5.so use_authtok
session     requisite     pam_krb5.so
```

## 4.6 LDAP and Active Directory

Lightweight Directory Access Protocol (LDAP) is a popular protocol for storing and validating user information in an enterprise. LDAP can store information about users and their group memberships, which Shiny Server Pro is able to query with a user's username and password. Fundamentally, the LDAP protocol describes a framework for storing hierarchical data, and can be configured in a multitude of ways. Active Directory is one popular directory service which implements LDAP and encourages a certain model for storing data, while other vendors of LDAP systems often have their own distinct default configurations. A holistic overview of LDAP is outside of the scope of this document, so if you lack a solid background in LDAP, you might benefit from consulting with an LDAP administrator in your organization to configure these settings.

An LDAP configuration in Shiny Server Pro might look like this:

```
auth_ldap ldap://ldap.example.org/dc=example,dc=org {
  group_filter "memberUid={username}";
  group_search_base ou=Groups;
}
```

It is fully possible to use the `auth_ldap` configuration to integrate with an Active Directory system. However, Active Directory encourages a certain pattern, which we capture in the `auth_active_dir` directive that extends `auth_ldap`. This configuration can be used to make setup simpler for groups using Active Directory. If you are using an Active Directory server, you may be able to specify the `auth_active_dir` setting and not worry about providing any of the others.

An Active Directory configuration setting may look something like the following:

```
auth_active_dir ldaps://dc01.example.org/dc=example,dc=org example.org{
  trusted_ca /etc/ssl/certs/example-org.cert;
}
```

User accounts with empty usernames or passwords are not supported.

The parent directive for all LDAP-related settings is `auth_ldap` or `auth_active_dir`. Both accept an LDAP URL as their first argument. `auth_active_dir` accepts a second argument that is the suffix (typically a domain name) to be added to all usernames when attempting to bind. All other child settings within this directive are not required, but may be needed depending on your LDAP configuration.

The LDAP URL should look like the following:

```
ldaps://ldap.example.org:1234/dc=example,dc=org
```

It begins with the protocol to be used when contacting the LDAP server – either `ldap://` (for unencrypted LDAP) or `ldaps://` (for LDAP over an encrypted SSL tunnel). We do not currently support StartTLS. If your LDAP server supports `ldaps://`, this option is more secure, and ensures the username and passwords are not being transmitted between Shiny Server and your LDAP server in plain text. Next, the hostname or IP of the LDAP server follows the protocol. Bear in mind that the given hostname or IP *must* match the hostname associated with the SSL certificate if using `ldaps://`. If using a non-standard port (anything other than 389 for `ldap://` or 636 for `ldaps://`), you can follow the hostname with a colon and the port number that should be used. Following the hostname/port should be a forward slash then the root DIT of the directory to use.

There are a variety of settings that will be used to guide Shiny Server Pro's interaction with your LDAP server. These will all be defined inside of an `auth_ldap` or `auth_active_dir` setting, and are described below. Be aware that the primary difference between `auth_ldap` and `auth_active_dir` is the default values assigned to these settings. If any of these settings are not specified, the default values will be used.

Some of the settings may include dynamic values that are listed under the "Variables Available" section of each description below. These variables are enclosed in curly braces ("{" and "}") and will be replaced with their value before being used.

### 4.6.1 `base_bind`

By default, the `auth_ldap` and `auth_active_dir` directives instruct Shiny Server Pro to use single-bind LDAP authentication for username and password validation. Double-bind LDAP authentication is used when the `base_bind` directive is defined. This directive specifies a user DN and password for the initial LDAP bind operation. The authenticated connection allows Shiny Server Pro to search for a DN associated with the user attempting to log in. The discovered DN is subsequently provided to a second bind operation.

```
base_bind <user-dn> <password-or-filepath>;
```

The `user_filter` directive is used during the user DN search, and must be defined whenever using `base_bind` with `auth_ldap`. The `auth_active_dir` directive provides a default value to `user_filter` but `auth_ldap` does not.

**Variables Available**

- `{root}` – The root DIT provided to the parent `auth_ldap` or `auth_active_dir` setting.

Here is an example `auth_ldap` specification using a `searcher` account during double-bind LDAP authentication with the required `user_filter` directive.

```
auth_ldap ldap://ldap.example.org/dc=example,dc=org {
  base_bind "uid=searcher,ou=People,{root}" "password-for-searcher";
  user_filter "uid={username}";
}
```

### 4.6.2 Securing the LDAP password

In our earlier example, we specified the password as plain text in the Shiny Server Pro configuration file. Alternately, we could use a fully-qualified path to a file that contains only the LDAP password, as in the following example:

```
auth_ldap ldap://ldap.example.org/dc=example,dc=org {
  base_bind "uid=searcher,ou=People,{root}" "/etc/shiny-server/ldap-base-bind-password.txt";
  user_filter "uid={username}"
}
```

With this configuration, Shiny Server Pro reads the LDAP password from the file `/etc/shiny-server/ldap-base-bi` Shiny Server Pro will retain `root` privileges when using a file to contain the LDAP password. This allows you to tighten the permissions around the file containing the LDAP password to only allow the `root` user to read it.

For example, we could secure the above file using:

```
# Change the file to be owned by the root user.
chown root:root /etc/shiny-server/ldap-base-bind-password.txt

# Set the permissions so that only root can read the file.
chmod 600 /etc/shiny-server/ldap-base-bind-password.txt
```

This protects the LDAP password from Shiny applications or other users who are allowed to login to the server. If you expect that many users or any untrusted users may be logging into or deploying apps on the server on which Shiny Server Pro is installed, we strongly recommend that you use this configuration to ensure the privacy of your LDAP password.

### 4.6.3 `user_bind_template`

Shiny Server Pro will transform the username entered at the login screen before attempting to "bind" (authenticate) to the LDAP server using this template. In general (notable exception being Active Directory), this pattern should map the given username to the user's DN in the LDAP database. In systems where that may not be possible (such as Active Directory), you can use the `user_filter` setting to lookup the user's DN given their username. In either case, this setting should manipulate the given username into the username used to perform the LDAP `bind` operation.

**Variables Available**

- `{username}` – The username entered by the user on the login screen.
- `{root}` – The root DIT provided to the parent `auth_ldap` or `auth_active_dir` setting.

**Default Value**

- For `auth_ldap` – `uid={username},ou=People,{root}`
- For `auth_active_dir` – `{username}@example.org` (where "example.org" is the domain name you provided as the second argument to `auth_active_dir`).

### 4.6.4 group_search_base

Defines the subtree in which groups are stored. This will be used as the root of all LDAP queries that attempt to find the groups of which a user is a member. The given value will be followed by a comma and the root DIT given in the auth_ldap or auth_active_dir parent setting. If configured to use an empty string as the base, then the unmodified root DIT will be used as the group search base.

**Default Value**

- For auth_ldap – ou=Groups
- For auth_active_dir – cn=Users

### 4.6.5 group_filter

The LDAP query to use when determining a user's group membership. The query should return all groups of which the given user is a member.

By default, auth_ldap checks the uniqueMember field – a setting from the groupOfUniqueMembers LDAP class. If you are using the posixGroup LDAP class to store your group memberships, you will likely need to set this value to memberUid={username}.

**Variables Available**

- {username} – The username entered by the user at the login screen.
- {userDN} – The computed DN for the entered user. If user_filter is unspecified, this will be the value calculated using the pattern from user_bind_template. If user_filter is provided, this will be the DN of the first object that matched the query specified in user_filter.
- {root} – The root DIT provided to the parent auth_ldap or auth_active_dir setting.

**Default Value**

- For auth_ldap – uniqueMember={username}
- For auth_active_dir – member:1.2.840.113556.1.4.1941:={userDN}

### 4.6.6 group_name_attribute

The attribute of the LDAP group object in which the group name is stored. This field will be compared to the entries in the required_group setting to determine whether or not a user should be granted access to a Shiny application.

**Default Value**

cn

### 4.6.7 `trusted_ca`

By default, Shiny Server Pro trusts many standard SSL Certificate Authorities (CAs, such as Verisign). If your organization uses a non-trusted Certificate Authority to sign its SSL certificates, you will need to tell Shiny Server Pro explicitly to trust this CA's certificate. You can do this by placing the CA's certificate (in PEM format) in a file on your machine, and pointing this setting to that file. You can add multiple trusted CAs (space-delimited) if you desire.

If this value is provided, the standard list of trusted CAs will be overridden with the provided certificate.

**Default Value**

None.

### 4.6.8 `check_ssl_ca`

When using LDAP over SSL (`ldaps://`), Shiny Server Pro will check that the SSL certificate was signed by a recognized Certificate Authority (CA) (though this can be modified using the `trusted_ca` setting). `check_ssl_ca` can be used to disable the checking of CAs entirely. By default, this option will be 'true' to enable SSL CA validation.

Disabling these checks is necessary if your LDAP server uses a self-signed certificate, for instance. Otherwise, this setting should typically be left enabled, as disabling it will largely defeat the security gained by using SSL.

**Default Value**

`true`

### 4.6.9 `user_filter`

Some systems (notably many Active Directory implementations) do not use the username as a part of the user's DN. In such systems, it may be necessary to perform an extra LDAP query after binding to determine the user's DN based on their username, before group membership can be determined. This setting stores the LDAP filter used to find the user object that matches the entered username.

Using the default provided for `auth_active_dir` (`userPrincipalName={userBind}`), as an example. Shiny Server Pro will attempt to bind to the LDAP server using the given username (after being manipulated as defined in `user_bind_template`) and password. If successful, it will then search for an object whose attribute `userPrincipalName` matches the username manipulated by `user_bind_template`. If found, the returned object's DN will be made available to the `group_filter` as the `{userDN}` variable.

**Default Value**

- For `auth_ldap` – N/A
- For `auth_active_dir` – `userPrincipalName={userBind}`

### 4.6.10 `user_search_base`

Defines the subtree in which users are stored. This will only be used if `user_filter` is not empty, as that is the only time that Shiny Server Pro would need to search for a user. If provided, this will be used as the root of the LDAP queries used to lookup a user by his or her username. The given value will be followed by a comma and the root DIT given in the `auth_ldap` or `auth_active_dir` parent setting. If configured to use an empty string as the base, the unmodified root DIT will be used as the user search base.

**Default Value**

- For `auth_ldap` – `ou=People`
- For `auth_active_dir` – `cn=Users`

## 4.7 Proxied Authentication

Proxied authentication can be enabled by using the `auth_proxy` configuration option. Organizations that employ a centralized authentication system not supported by Shiny Server may use this feature to leverage a proxied authentication system. In such a model, traffic destined for Shiny Server Professional would first be sent through an authenticating proxy, which would handle user authentication and header setting as appropriate, to designate the user and (optionally) groups. Shiny Server Pro would then leverage this information to make the decision about whether or not to allow the current user to access certain resources.

Precautions should be taken to ensure that the Shiny Server machine is not accessible directly by users, but is instead *only* accessible through the authenticating proxy. Otherwise, an attacker would merely need to provide fraudulent HTTP headers when accessing Shiny Server to imitate another user or claim membership in a particular group.

The `auth_proxy` option allows you to configure a header name for usernames and (optionally) another header for groups. Both are case-insensitive.

```
auth_proxy X-USER X-GROUPS;
```

The above configuration would expect a header named `X-USER` to provide the username for each incoming request, and would look for a header named `X-GROUPS` to provide the comma-delimited group names of which this user is a member. By default when using `auth_proxy`, the username header is `X-Auth-Username` and the groups header is not enabled.

The authenticating proxy should not provide headers if the user is not authenticated. If the user is authenticated, the username header should always be specified. The groups header is optional, even if configured.

## 4.8 Required Users & Groups

### 4.8.1 User Authentication

The `required_user` setting controls which users should be able to access a particular `server`, `location`, or `admin`. For example:

```
location /app1 {
  required_user kim tom;
}
```

This directive enables authentication on the app1 `location`, which would present any new, unauthenticated visitor to the application with a login page. If the visitor entered valid credentials against your configured authentication mechanism, then he or she would be checked against the list of allowed users for this application. If found (i.e., if the user name was either `kim` or `tom`), the visitor would be granted access to the application. Otherwise, he or she would be unable to access the application.

There are two special cases for `required_user`. Including the user named `*` will admit any user who is able to login. Additionally, including the user `**` will admit any visitor, whether or not they are logged in; this is the default behavior if no `required_user` setting is specified.

It is encouraged that any `server` accepting user credentials be secured via SSL.

### 4.8.2 Group Authentication

The `required_group` setting allows the administrator to control access based on the groups of which users are a member. In this way, entire sets of users can be granted access to a particular location via one line in the configuration file.

Neither flat-file authentication nor Google Authentication supports the notion of user groups, but this feature can be used with PAM, Proxied Authentication, LDAP or Active Directory by adding a restriction like the following:

```
location /app1 {
  required_group shinyUsers admins;
}
```

This configuration would grant access to any user who is a member of either the `shinyUsers` or `admins` group.

#### 4.8.2.1 `group_list`

Shiny Server Pro uses an encrypted and signed browser cookie to store the names of all the groups a user belongs to. If a user's list of groups has a total number of characters in the thousands, then login can fail.

You can work around this limitation using the `group_list` directive (introduced in Shiny Server Pro 1.5.2) to filter down the list of groups returned from a user login. You can specify groups to whitelist or blacklist using exact name matching, wildcard expressions, or regular expressions.

Some examples:

Include only the "development" and "marketing" groups:

```
group_list include development marketing;
```

Exclude all groups that contain the word `private`:

```
group_list exclude_glob *private*;
```

You can use the `group_list` directive multiple times, and their effects will be combined in order, with intersection (as opposed to union) semantics. In other words, each additional `group_list` directive can only further limit, not expand, the list of groups for a user. (For example, `group_list include foo;` followed by `group_list include bar;` means only groups that exactly match `"foo"` and exactly match `"bar"`; since this is impossible, the resulting group list would always be empty.)

The `group_list` settings are applied at login time, so they cannot be customized on a location-by-location basis; instead, they must be specified globally.

## 4.9 SSL

In Shiny Server Professional, SSL encryption is available for any configured `server`, and in the `admin` interface. SSL is a means of encrypting traffic between web clients and servers, and should be used for any application that will accept or transmit sensitive information such as passwords. Untrusted SSL certificates can be generated locally by anyone for free; signed SSL certificates can be obtained from authorized Certificate Authorities fairly inexpensively. Untrusted SSL certificates will still encrypt the traffic between a client and server, but (in most browsers) the client will need to proceed through a warning about the untrusted nature of the SSL certificate in use. SSL can be enabled on a `server` by configuring the two required parameters: the path to the SSL key, and the path to the SSL certificate.

```
server {
  # Instruct this server to listen on port 443, the default port for HTTPS
  # traffic
  listen 443;
  ssl /etc/shiny-server/ssl-key.pem /etc/shiny-server/ssl.cert;

  ...
}
```

If a valid SSL key and certificate were provided, a user would now need to preface their request with "https://" to be able to visit any application hosted within this `server` or on the `admin`. It is strongly recommended that any Shiny Server Pro configuration that expects to accept its users' LDAP or Active Directory credentials use SSL to secure the usernames and passwords on their applications and admin interface.

## 4.10 Proxied Headers

Typically, HTTP headers sent to Shiny Server will not be forwarded to the underlying Shiny application. However, Shiny Server Professional is able to forward specified headers into the Shiny application using the `whitelist_headers` configuration directive, which can be set globally or for a particular `server` or `location`.

```
server {
  listen 3838;
  whitelist_headers myorg_userrole myorg_privileges;

  ...

}
```

The above configuration would allow the two listed `myorg_` headers (case-insensitive) to be forwarded from the originating HTTP traffic into the Shiny session. In Shiny, these headers would be converted to capital letters and prefaced with `HTTP_` to designate that the headers originated in an HTTP request. They would then be available as objects in the `session$request` environment inside an application's `server.R` file. For example, the `myorg_userrole` object could be accessed via the following `server.R` file.

```
library(shiny)
shinyServer(function(input, output, session){
  output$userrole <- renderText({
    session$request$HTTP_MYORG_USERROLE
  })
})
```

If you would like to use this feature to handle user authentication, see the Proxied Authentication section.

## 4.11 Clickjacking Protection

Clickjacking is the term for a variety of techniques that malicious websites use to trick users into clicking on links or buttons on third-party pages.

One particular kind of clickjacking involves iframes. A malicious website can embed another, non-malicious website in an obscured or transparent iframe. Then, the malicious site can overlay the invisible iframe over a link that looks clickable. When the user clicks on the link, the click actually registers inside the iframe, on the non-malicious website. The click on the non-malicious website might do something like reset the user's password, or delete something of theirs.

Shiny Server offers a degree of clickjacking protection with the `frame_options` directive, which gives the Shiny Server administrator control over the `X-Frame-Options` header. Since 2009, it's been possible for servers to set the `X-Frame-Options` HTTP header in responses. Most modern

browsers interpret this header, which can prevent applications from appearing in iframes and thus being vulnerable to clickjacking.

The following configuration sets the `X-Frame-Options` header to `deny` on every Shiny Server response, preventing any application from appearing in an iframe in most modern browsers:

```
frame_options deny;
server {
  listen 3838;

  ...

}
```

# Chapter 5

# Monitoring the Server

## 5.1 Admin

### 5.1.1 Configuration

The Admin interface provided with Shiny Server Professional is disabled by default. To enable it, use the `admin` setting to specify the port number on which the admin interface should run.

```
admin 3957 {
  required_user tom sarah;
}
```

The above would enable the admin interface on port 3957. This interface would then be available from any modern web browser on that port. First-time visitors would be presented with a login page. If the visitor were able to authenticate themselves to the Shiny Server application as either `tom` or `sarah`, they would be granted access to the admin dashboard.

The required user must be one connected to Shiny Server as described in Authentication & Security.

You can also leverage SSL to secure the usernames and passwords accepted on the admin login screen.

Note that the Admin interface is currently not supported on any version of Internet Explorer.

### 5.1.2 Organization

The admin interface offers a real-time view into the current state of your Shiny Server. There is no need to refresh the page periodically, as the information is automatically updated as changes occur. The interface is divided into four main sections.

**Dashboard**

Initially when logging in, the user will be presented with the Dashboard view. This view shows a basic overview of the state of Shiny Server, including the current RAM and CPU utilization, the number of connections, and historical RAM and CPU saturation.

### Applications

The highest-level grouping in the admin interface is the notion of an "Application." An application represents a Shiny App stored in a particular directory on your server. There may be multiple R processes behind one application due to application restarts, but all of these R processes are considered part of the same application. Initially, you will be presented with a table displaying an overview of the various directories on your server in which applications are stored, the current number of processes associated with that application, and the number of open connections to that application.

Shiny Server does not crawl the disk to discover all the potential Shiny applications that could be spawned. Instead, it only becomes aware of the existence of a Shiny application after it has been visited for the first time.

Clicking on any "App Directory" in the table will take you to a detailed view of that application. On this page, you can track current and historical RAM usage, along with CPU utilization, active processes, and open connections.

You will also see the current and historical "latency" for this application. Keep in mind that, because R is single-threaded, it is only able to compute one request at at time. Thus, while one user's request is being computed, any other requests that are assigned to this R process will be inserted into a queue until the R process becomes free. Only then will computation begin on the second request. The latency of an application represents the delay between when a request initially arrived at Shiny Server and the moment the assigned R process began computation to fulfill that request. A Shiny application that is not computationally intensive and does not have many active users may see a latency of just a few milliseconds. Conversely, an application which is computationally intensive and/or overwhelmed with more users than the associated process(es) can handle will see a much higher latency.

### Processes

The "Processes" tab provides access to information about the underlying R Shiny processes that support the various applications being hosted on this server. The overview table reveals with which application each process is associated, the latency of this application, and its current memory consumption. Clicking on the ID of the process will take you to a detailed view of this process. Here you can see the current CPU utilization, open connections, and various other information helpful for understanding this process.

### Connections

This tab displays all of the connections currently open between Shiny Server and its visitors. You can see which application the connection is accessing, when it was opened, its current status (whether it is idle or awaiting the results of some analysis), the protocol being used, whether or not the connection is encrypted, the IP address associated with the connection, and – if relevant – the username to whom the connection belongs. You can also filter the list to view only those connections associated with a particular app directory or status.

The protocol each connection is using will be displayed in a blue oval at the far left edge of the table, using some abbreviation. If you hover the cursor over the oval, you will see the full protocol name. You can find more details on the various protocols supported by Shiny Server in the section on Specifying Protocols.

### 5.1.3 Killing Processes and Connections

The Admin interface also provides the ability to kill individual processes or connections. Alongside any table in which a process is listed, if you point your mouse at an application or connection, you will see a button appear on the right side which allows you to kill that app or connection. Clicking this button will present you with a confirmation dialogue box that asks you to confirm that you wish to disconnect this connection or kill this process. If you confirm, that process or connection will be terminated.

After terminating a connection, any users will immediately be disconnected from the server and the application will appear "greyed out." Keep in mind that, in some Shiny applications, this could forfeit some of the work being done by this user on that application. After terminating a process, any users with connections open to that process will be immediately disconnected. Again, this could potentially cause the user to lose some data or work if they were using a complex Shiny application.

## 5.2 Graphite

The Admin interface is often the preferred tool to monitor the performance of a single Shiny Server Pro server. However, for organizations that manage multiple servers or Shiny Server Pro instances, it may be convenient to send server monitoring data to a centralized database, rather than local RRD files. For such organizations, Shiny Server Pro can output its metrics to Graphite (or any other engine compatible with the Carbon protocol). This can be enabled in addition to, or in place of, local RRD file monitoring. Note that if RRD monitoring is disabled, no data will be recorded for later inspection in the Admin interface.

To enable Graphite metrics, use the `graphite_enabled` configuration directive. This setting accepts two optional arguments: the address of the Graphite server (`host`) and the `port` on which Carbon – the component of Graphite responsible for the intake of new data – is listening on that server. By default, these will be set to `127.0.0.1` and `2003`, respectively.

```
# enable Graphite metrics on server graphite.example.org at port 2003
graphite_enabled graphite.example.org 2003;
```

RRD metrics are enabled by default and used by the Admin interface. If you choose to disable RRD metrics, you can use the `rrd_disabled` setting. If this setting is present in your configuration file, RRD metrics will not be recorded; in such a configuration, the admin interface cannot be enabled.

Changes to either of these settings will not take effect until the server is restarted.

## 5.3 Health Check Endpoint

Some advanced deployment scenarios are able to leverage a "health check endpoint", which provides an automated way to ensure that Shiny Server is online and responsive.

> **Deprecated!** The `/ping` endpoint has been removed from Shiny Server as of version 1.1.0 in favor of the `/__health-check__` endpoint.

The URL `/__health-check__` (note the double underscores on either side) can be used on any port Shiny Server is instructed to listen on, to test the health of Shiny Server. If the server is online and responsive, it will respond with an HTTP 200 code and information about the server, including the number of current connections and the current CPU load. By default, the page will look something like this:

```
server-version: 1.2.3.4
active-connections: 8
active-apps: 2
active-processes: 3
cpu-percent: 13
memory-percent: 49
swap-percent: 39.1
load-average: 1.01953125
```

### 5.3.1 Customizing Responses

Shiny Server Pro uses a templating system to determine how to respond to visits to the health-check URL. By default, it uses the following template:

```
server-version: #server-version#
active-connections: #active-connections#
active-apps: #active-apps#
active-processes: #active-processes#
cpu-percent: #cpu-percent#
memory-percent: #memory-percent#
swap-percent: #swap-percent#
load-average: #load-average#
```

The values enclosed in `#` symbols are treated as variables that will be substituted for their true values before being returned. The template above shows all of the variables that are currently available to the endpoint.

You can specify whatever format you like for these variables by creating a custom template in `/etc/shiny-server/health-check`. This file will be used to structure the response to the health check after substituting the available variables. For instance, you could configure the endpoint to return this data in XML by placing the following content in the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<health-check>
  <server-version>#server-version#</server-version>
  <active-connections>#active-connections#</active-connections>
  <active-apps>#active-apps#</active-apps>
  <active-processes>#active-processes#</active-processes>
  <cpu-percent>#cpu-percent#</cpu-percent>
  <memory-percent>#memory-percent#</memory-percent>
  <swap-percent>#swap-percent#</swap-percent>
  <load-average>#load-average#</load-average>
</health-check>
```

### 5.3.2 Providing Multiple Health-Check Endpoints

The file at `/etc/shiny-server/health-check` specifies the default template to use for the health check. Shiny Server Pro supports multiple health check endpoints that can be accessed using different URLs. For instance, you can setup a new endpoint that responds with JSON using the following template:

```
{
  'server-version' : '#server-version#',
  'active-connections': #active-connections#,
  'active-apps': #active-apps#,
  'active-processes': #active-processes#,
  'cpu-percent': #cpu-percent#,
  'memory-percent': #memory-percent#,
  'swap-percent': #swap-percent#,
  'load-average': #load-average#
}
```

Store this text in a new file at `/etc/shiny-server/health-check-json` (the file is suffixed by a *hyphen* then some custom string). This endpoint would then be available at `http://<server-address-and-port>/__health-check__/json` (the URL is suffixed by a *forward slash* then the same custom string used in the file name).

# Chapter 6

# Licensing & Activation

## 6.1 Product Activation

### 6.1.1 Activation Basics

When Shiny Server Professional is first installed on a system, it operates in evaluation mode for a period of time and then subsequently requires activation for continued use. To determine the current license status of your system, you can run the following command:

```
$ sudo /opt/shiny-server/bin/license-manager status
```

After purchasing a license to Shiny Server Professional, you will receive a product key that can be used to activate the license on a given system. You can perform the activation with the following commands:

```
$ sudo /opt/shiny-server/bin/license-manager activate <product-key>
$ sudo restart shiny-server  # or "sudo /sbin/service shiny-server restart" or "sudo systemctl
```

Note that you need to reload or restart the server for licensing changes to take effect.

If you want to move your license of Shiny Server Professional to another system, you should first deactivate it on the system you are moving from. For example:

```
$ sudo /opt/shiny-server/bin/license-manager deactivate
```

You can then perform the activation commands on the new server location.

## 6.2 Connectivity Requirements

To activate or deactivate a Shiny Server Professional license, internet connectivity is required for communication with the licensing server. If your server is behind an internet proxy or not connected to the internet at all, you will need to activate the license differently, as described below.

### 6.2.1 Proxy Servers

If your server is behind an internet proxy, you may need to add an additional command line flag to indicate the address and credentials required to communicate through the proxy. Note, however, that this may not be necessary if either the `http_proxy` or `all_proxy` environment variable is defined (these are read and used by Shiny Server Professional when available).

If you do need to specify a proxy server explicitly, you can do so using the `--proxy` command line parameter. For example:

```
$ sudo /opt/shiny-server/bin/license-manager --proxy=http://127.0.0.1/ activate <product-key>
```

Proxy settings can include a host-name, port, and username/password if necessary. The following are all valid proxy configurations:

```
http://127.0.0.1/
http://127.0.0.1:8080/
http://user:pass@127.0.0.1:8080/
```

If the port is not specified, the license manager will default to using port 1080.

### 6.2.2 Offline Activation

If your system has no connection to the internet, it is also possible to perform an offline activation. To do this, we recommend using our offline activation app which will walk you through the process: RStudio Offline Activation

You first generate an offline activation request with the following command:

```
$ sudo /opt/shiny-server/bin/license-manager activate-offline-request <product-key>
```

Executing this command will print an offline activation request to the terminal, which you should copy and paste into our offline activation application or send to RStudio customer support (support@rstudio.com). You will receive a reply with a file attachment that can be used to activate offline as follows:

```
$ sudo /opt/shiny-server/bin/license-manager activate-offline <activation-file>
$ sudo reload shiny-server  # or "sudo /sbin/service shiny-server reload"
```

Note that you need to reload or restart the server for licensing changes to take effect.

If you want to renew your Shiny Server Professional license or move it to another system, you can also perform license deactivation offline. You can do this as follows:

```
$ sudo /opt/shiny-server/bin/license-manager deactivate-offline
```

Executing this command will print an offline deactivation request to the terminal, which you should copy and paste into our offline activation application or send to RStudio customer support (support@rstudio.com).

You can also perform an offline check of your current license status using the following command:

```
$ sudo /opt/shiny-server/bin/license-manager status-offline
```

## 6.3 Floating Licenses

If you stop and start Shiny Server Pro instances frequently, for instance because you're running them inside virtual machines or containers, you may wish to use floating licensing instead of traditional licensing.

To use floating licensing, you run a small, lightweight server, which holds a license that grants you the right to run a certain number of concurrent Shiny Server Pro instances.

When Shiny Server Pro starts, it will connect to the license server and obtain a temporary lease, releasing it when Shiny Server Pro is stopped. Using this method, you can have any number of Shiny Server Pro instances, so long as you do not run more instances at once than specified in your license.

### 6.3.1 The Shiny Server Pro License Server

The Shiny Server Pro License Server site contains license server downloads for all RStudio products. Download the license server for Shiny Server Pro and install it with a command like the following for your distribution:

```
$ sudo dpkg -i ssp-license-server-1.0.6-x86_64.deb
```

Then, you then activate your license key with the command:

```
$ sudo ssp-license-server activate <product-key>
$ sudo ssp-license-server start
```

If you wish to use your license on a different server at a later time, you may deactivate it with this command:

```
$ sudo ssp-license-server deactivate
```

A license key which distributes floating license leases is not the same as a traditional license key, and the two cannot be used interchangeably. If you have purchased traditional license keys and wish to exchange them for a floating license key, or vice versa, please get in touch with RStudio customer support (support@rstudio.com).

The file `/etc/ssp-license-server.conf` contains configuration settings for the Shiny Server Pro License server, including the network port to listen on and any proxy settings required for connecting to the Internet.

### 6.3.2 License Server Offline Activation

The `ssp-license-server activate` command requires an Internet connection. If your license server has no connection to the Internet it's also possible to perform an offline activation. The process for doing this on the license server is identical to the process used to activate Shiny Server Pro offline. Generate an offline activation request as follows:

```
$ sudo ssp-license-server activate-offline-request <product-key>
```

Executing this command will print an offline activation request to the terminal which you should copy and paste and then send to RStudio customer support (support@rstudio.com). You will receive a reply with a file attachment that can be used to activate offline as follows:

```
$ sudo ssp-license-server activate-offline <activation-file>
$ sudo restart ssp-license-server
```

### 6.3.3 Using Floating Licensing

Once your license server is up and running, you need to tell Shiny Server Pro to use floating licensing instead of traditional licensing with the `license_type` configuration directive.

Ensure */etc/shiny-server/shiny-server.conf* contains the following:

```
license_type floating;
```

The value `floating` indicates that Shiny Server Pro should connect to a remote licensing server to obtain a license; the value `traditional` can be used to explicitly specify traditional (local) activation.

If `license_type` is unspecified, it defaults to `traditional`.

Then, tell Shiny Server Pro which licensing server to connect to and specify the port if not using the default.

```
$ /opt/shiny-server/bin/license-manager license-server <server-hostname-or-ip:port>
$ sudo restart shiny-server
```

You only need to run the license-server command once; Shiny Server Pro saves the server name and will use it on each subsequent startup.

By default, the Shiny Server Pro License Server listens on port 8989. If you wish to use a different port, you will need to specify the port in `/etc/ssp-license-server.conf`, and specify `license-server` as `<server-hostname-or-ip:port>`.

Depending on your system configuration, it is possible that the Server Server Pro service will be started before the service which allows hostname resolution (this is known to be the case for example on some Amazon EC2 systems). If this is the case, you'll want to specify the license server using a private IP address rather than a hostname, so that Shiny Server Pro can acquire a license immediately when starting up.

### 6.3.4 Configuring License Leases

When using floating licenses, you can optionally determine how long the license leases last by setting the lease length value on the licensing server. This value is in seconds, so for instance to make license leases last 30 minutes you would use the following syntax:

*/etc/ssp-license-server.conf*

```
<lease length="1800"/>
```

The lease length controls how frequently the Shiny Server Pro instances need to contact the licensing server to renew their license leases in order for the lease to remain valid.

A shorter lease length will increase tolerance to failures on Shiny Server Pro instances by making leases available for reuse more quickly. Shiny Server Pro will release its lease immediately if shut down normally, but if abnormally terminated, the lease will not be released until it expires.

A longer lease length will increase tolerance to transient failures on the network and the Shiny Server Pro License Server. Any such issues that can be resolved before the lease is due for renewal won't interrupt use of Shiny Server Pro.

We generally recommend using a longer lease length. Use a short lease length only if your environment routinely encounters abnormal terminations of the server or the container/instance on which it runs.

### 6.3.5 Troubleshooting Floating Licensing

To validate that the license server has been successfully activated, run the `activation-status` command. This will report the version of the server as well as the license key and the number of available slots.

```
$ sudo ssp-license-server activation-status
```

If your server is activated but you're still having trouble with floating licensing, you can tell the Shiny Server Pro License Server to emit more detailed logs. Change the log level to notification:

*/etc/ssp-license-server.conf*

```
<log file="/var/log/rstudio-licensing.log" level="notification"/>
```

Then, restart the license server, tail the licensing log, and start your Shiny Server Pro instances.

```
$ sudo ssp-license-server restart
$ tail -f /var/log/rstudio-licensing.log
```

At the notification level, the licensing log will tell you the total number of licenses associated with your key, and how many are currently in use. It will also notify you when Shiny Server Pro instances acquire leases, and when those leases are released, renewed, or expired. No rotation is done for this log, so it's recommended to use the warning level in production.

# Chapter 7

# Appendix

## 7.1 Quick Start

We recommend reading through the relevant sections of this guide to gain a complete understanding of how to operate Shiny Server. However, if you are just looking to get running quickly, you can follow one of the guides below.

All of these guides will help you customize your Shiny Server configuration file, stored at `/etc/shiny-server/shiny-server.conf`. By altering this file, you can control exactly how Shiny Server runs.

The following guides are available:

- Host a directory of applications - This is the default configuration that Shiny Server will use until you provide a custom configuration in `/etc/shiny-server/shiny-server.conf`. This guide will show you how to serve multiple applications from a single directory on disk – `/srv/shiny-server/`. If you are not sure where to start, we recommend this guide.
- Let users manage their own applications - This guide will serve Shiny applications created by your users and stored in their home directories.
- Run Shiny Server on multiple ports - This guide will introduce a multi-server configuration, which demonstrates how to run two distinct services on different ports on your server. It also configures one server with multiple locations.

And for Pro features:

- Require user authentication on an application - This example will help you set up a local username/password database, and use it to secure your applications.
- Host a secure Shiny Server - If you would like to set up a secure instance of Shiny Server that is encrypted with HTTPS (SSL/TLS), this guide will show you how.
- Host an Application Supported by Multiple R Processes - This guide will demonstrate how to scale a Shiny application to multiple R processes.

### 7.1.1 Host a directory of applications

Shiny Server is configured by a file called `/etc/shiny-server/shiny-server.conf`. As a complement to this Quick Start guide, we provide a convenient executable in `/opt/shiny-server/bin/deploy-example` that can copy the example configuration file associated with this Quick Start to that location, and install a directory of example applications. **Be sure to back up your current configuration file first**, then run the following command:

```
sudo /opt/shiny-server/bin/deploy-example default
```

This configuration expects your Shiny applications to be hosted in `/srv/shiny-server/`. The installer will also place one sample application in `/srv/shiny-server/sample-apps/hello/`.

By default, Shiny Server listens on port 3838, so the default application will be available at `http://<server-address>:3838/sample-apps/hello/`.

If you wanted to change the port or the directory Shiny Server uses, you must adjust the configuration file. To begin customizing your server, open `/etc/shiny-server/shiny-server.conf` in your preferred text editor, and modify the line that says `listen 3838`. This is the line that instructs your server to listen on port 3838 – change it to another port such as `12345` and save the file. You will need to restart Shiny Server to make it aware of the new configuration file. The Stopping and Starting section explains this in more detail. For now, you will run this command if you are on a distribution that supports Upstart (most recent distributions, including Ubuntu 12 and later and CentOS 6):

```
sudo restart shiny-server
```

If you are on an older system that relies on init.d (such as CentOS 5), you will run the following command, instead:

```
sudo /sbin/service shiny-server restart
```

You should now find that Shiny Server is no longer running on port 3838, but is on whichever new port you selected.

### 7.1.2 Let users manage their own applications

Some administrators of Shiny Server would prefer to give users – or some trusted subgroup of users – the ability to manage and host their own Shiny applications. This allows users to work directly with their applications rather than requiring an administrator to deploy and update applications on their behalf. This guide will document how to create such an environment, using the `user_dirs` directive documented in the section named Host Per-User Application Directories.

As a complement to this Quick Start guide, we provide a convenient executable in `/opt/shiny-server/bin/deploy-example` that can configure Shiny Server to use the example configuration file associated with this guide (by placing it in `/etc/shiny-server/shiny-server.conf`). It will also install a directory of example applications. **Be sure to back up your current configuration file first**, then run the following command:

```
sudo /opt/shiny-server/bin/deploy-example user-dirs
```

This configuration enables all users on the system to host their own applications by creating a `ShinyApps` directory in their home directory; you can always alter the configuration file later to tailor it to meet your needs. You can deploy an application under your username by copying an example that comes with Shiny Server into your own `ShinyApps` directory. To do this, copy the entire `/opt/shiny-server/samples/sample-apps/hello/` directory into `~/ShinyApps` using the following command:

```
mkdir ~/ShinyApps
sudo cp -R /opt/shiny-server/samples/sample-apps/hello ~/ShinyApps/
```

If that command succeeds, you have successfully installed a `user_dirs` Shiny application!

By default, Shiny Server listens on port 3838, so your new application will be available at `http://<server-address>:3838/<your_username>/hello` where `<your_username>` is your Linux username.

If you wanted to restrict the privilege of running Shiny applications to a particular set of users on your system, you can uncomment the `members_of` line in the configuration file. You can then specify the Linux group name of which a user must be a member before he or she would be allowed to host Shiny applications.

### 7.1.3 Run Shiny Server on multiple ports

This guide will describe how to run Shiny Server on multiple ports simultaneously, and also how to run a single server with multiple locations.

As a complement to this Quick Start guide, we provide a convenient executable in `/opt/shiny-server/bin/deploy-example` that can configure Shiny Server to use the example configuration file associated with this guide (by placing it in `/etc/shiny-server/shiny-server.conf`). It will also install a directory of example applications. **Be sure to back up your current configuration file first**, then run the following command:

```
sudo /opt/shiny-server/bin/deploy-example multi-server
```

This configuration will run two servers: one on port 3838, and one on port 4949. As you can see in the configuration file, a variety of different `location`s are defined with different hosting models.

The server running on port 3838 defines two locations. The first is at the URL `/users`, which makes user applications available using the `user_dirs` hosting model, as documented in the section entitled Host Per-User Application Directories. The second location is at the URL `/example1`, and hosts a single application that should be stored in `/srv/shiny-server/sample-apps/hello/`. An application is provided here during installation.

You should copy that application to your home directory, as well, so you have something to view in the `/users` location. You can copy that application to your home directory with the following commands:

```
# Copy to your home directory's 'ShinyApps' folder
mkdir ~/ShinyApps
cp -R /srv/shiny-server/sample-apps/hello ~/ShinyApps/
```

You should now be able to access applications in a variety of ways:

- On the server running on port 3838, you have a `user_dirs` location at `/users` hosting your personal applications. You can access the one deployed earlier at a URL like `http://<server-address>:3838/users/<your_username>/hello` where `<your_username>` is your Linux username.
- Also on the server running on port 3838, you have a single application available at the `/example1` location. The URL `http://<server-address>:3838/example1` should point you to that example.
- Finally, the server on port 4949 will host any application defined in `/srv/shiny-server`. You can access the one that comes with Shiny Server at a URL like `http://<server-address>:4949/sample-apps` which should point you to the same application you just loaded a moment ago.

### 7.1.4 Require user authentication on an application

Shiny Server Pro is capable of multiple different methods of authenticating users, as described in the chapter on Authentication & Security. Out of the box, it is configured to use a Flat-File Authentication database stored in `/etc/shiny-server/passwd`. This database is initially empty; to create a user named `admin`, execute the following command:

```
$ sudo /opt/shiny-server/bin/sspasswd /etc/shiny-server/passwd admin
```

then enter the new password for the `admin` user when prompted. Once completed, you will have a single user defined in your database named `admin`.

You should now be able to log in to your administrative dashboard at a URL like `http://myserver.com:4151/`, after replacing `myserver.com` with your server's IP or hostname, with the username `admin` and the password you just created. See the chapter entitled Admin for more details on how to take advantage of this interface.

After defining usernames and passwords in this password database, you can also restrict access to `server`s and `location`s using the `required_user` directive.

As a complement to this Quick Start guide, we provide a convenient executable in `/opt/shiny-server/bin/deploy-example` that can configure Shiny Server to use the example configuration file associated with this guide (by placing it in `/etc/shiny-server/shiny-server.conf`). It will also install a directory of example applications. **Be sure to back up your current configuration file first**, then run the following command:

```
sudo /opt/shiny-server/bin/deploy-example auth
```

The specifications laid out in the example configuration file (in `/etc/shiny-server/shiny-server.conf`) dictate that the sample Shiny application requires authentication – you must be logged in as the user named "admin" to access it. If you have visited the Administrative Dashboard on port 4151, then you may already be logged in. You can test your configuration by either logging out, or using another browser/computer to confirm that you are unable to access the application you just copied without logging in (this application will be hosted at a URL like `http://<server-address>:3838/samples-apps/hello`). However, once you log in as the user named "admin" using the password you defined above, you will be granted access to this application.

Note that, because the configuration in use is structured like so:

```
server {
  listen 3838;

  location / {

    ...

    location /sample-apps/hello {
      required_user admin;
    }
  }
}
```

*only* the `hello` location requires authentication; all other applications are freely available. If you were to move the `required_user` directive to the parent location, however, all applications defined in that location would require authentication as the user named "admin".

### 7.1.5 Host a secure Shiny Server

Shiny Server Professional is able to serve Shiny applications using SSL/TLS, an encrypted channel between your server and your clients.

The configuration in use here expects your SSL key and certificate to be available in `/etc/shiny-server/server.key` and `/etc/shiny-server/server.cert`, respectively. If you have an existing SSL certificate stored elsewhere, you can update the configuration file at `/etc/shiny-server/shiny-server.conf` to point to the appropriate files.

If you do not have an SSL certificate available, you can setup a "self-signed" certificate. This certificate will allow you to encrypt traffic between your clients and server, but will not be "trusted" by most browsers. Thus, when a user visits a page secured via a self-signed certificate, the user will get a warning advising them not to proceed. We recommend that you to obtain a certificate from an established Certificate Authority if you implement this in a production environment; however, for illustrative purposes, we will describe how to create a self-signed certificate and integrate it into Shiny Server. The details of creating and signing SSL certificates is outside the scope of this guide, but the commands are given below for your convenience.

To create a certificate, you must have the `openssl` library installed on your server. You can then create an SSL key using the following command:

```
sudo openssl genrsa -out /etc/shiny-server/server.key 1024
```

You will then generate a certificate signing request (CSR) using the following command:

```
sudo openssl req -new -key /etc/shiny-server/server.key \
  -out /etc/shiny-server/server.csr
```

(Where a command includes a \ you can either enter that literally as it appears in this document, or you can omit it and enter the whole command on a single line.) This command will ask you for information about your organization, which you can omit if you plan to sign this certificate yourself.

You can then sign that request yourself using this command:

```
sudo openssl x509 -req -days 3650 \
  -in /etc/shiny-server/server.csr \
  -signkey /etc/shiny-server/server.key \
  -out /etc/shiny-server/server.cert
```

As a complement to this Quick Start guide, we provide a convenient executable in `/opt/shiny-server/bin/deploy-example` that can configure Shiny Server to use the example configuration file associated with this guide (by placing it in `/etc/shiny-server/shiny-server.conf`). It will also install a directory of example applications. **Be sure to back up your current configuration file first**, then run the following command:

```
sudo /opt/shiny-server/bin/deploy-example ssl
```

It is a good idea to inspect the log file at this point to ensure that there were no problems using your SSL key and certificate. The server log file is stored in `/var/log/shiny-server.log`. You can preview the last 20 lines of this file using the following command.

```
sudo tail -n 20 /var/log/shiny-server.log
```

If there were no errors in the log related to using your SSL configuration, you should now be able to connect to your server from a browser by visiting a URL like `https://<server-address>:3939/sample-apps/hell`
Note that, because of the configuration settings in this example, you *must* specify the `https://` protocol and port 3939 when visiting the page. If you signed the certificate yourself, your browser will likely prompt you about the untrusted certificate. If you instruct your browser to accept the certificate, you will be taken to your application, which now is secured via SSL/TLS encryption.

### 7.1.6 Host an Application Supported by Multiple R Processes

You can configure Shiny Server Professional to run multiple Shiny processes for a single application. You could choose to distribute incoming traffic evenly across up to three processes (the default), or even create a separate Shiny process for each incoming user. The configuration details for this setup are discussed in the section on the Utilization Scheduler.

As a complement to this Quick Start guide, we provide a convenient executable in `/opt/shiny-server/bin/deploy-example` that can configure Shiny Server to use the example configuration file associated with this guide (by placing it in `/etc/shiny-server/shiny-server.conf`). It will also install a directory of example applications. **Be sure to back up your current configuration file first**, then run the following command:

```
sudo /opt/shiny-server/bin/deploy-example multi-proc
```

This configuration includes the following line to specify that a Utilization Scheduler should be used when serving Shiny applications:

```
utilization_scheduler 1 .7 5;
```

This configuration dictates that a Shiny process should only have one active Shiny session, and that there should be at most 5 Shiny processes per application. This configuration would be appropriate for an application that is computationally intensive. In such an application, one session may tie up the Shiny process for minutes at a time with computation, which would create an unpleasant experience for other users trying to connect to the same process during that window. With this configuration, we can ensure that each user gets his or her own designated Shiny process.

You should be able to connect to the Shiny Server sample application from a browser by visiting a URL like `http://<server-address>:3838/sample-apps/hello`.

To inspect the processes that are being created, and understand how they are performing, you can open Shiny Server's Admin Dashboard. Before you can login to the dashboard, you must create a user who is allowed to access the Dashboard. Shiny Server Pro comes with multiple different methods of authenticating users (as described in the chapter on Authentication & Security). Out of the box, it is configured to use a Flat-File Authentication database stored in `/etc/shiny-server/passwd`. This database is initially empty; to create a user named `admin`, execute the following command:

```
$ sudo /opt/shiny-server/bin/sspasswd /etc/shiny-server/passwd admin
```

then enter the new password for the `admin` user when prompted. Once completed, you will have a single user defined in your database named `admin`.

You should now be able to log in to your administrative dashboard at a URL like `http://<server-address>:4151/` with the username `admin` and the password you just created. See the chapter entitled Admin for more details on how to take advantage of this interface.

You will be able to monitor the processes that are being created in real-time using the Dashboard. You can also drill down into each process to verify that the connections are being assigned to their own private Shiny processes as expected.

## 7.2 Configuration Settings

Listed below are all the directives that are supported in Shiny Server config files.

**Applies to** indicates the kind of parent scope that this directive normally appears inside.

**Inheritable** means that you can put this directive at a higher level in the hierarchy and it will be inherited by any children to which it might apply. Inherited directives can be overridden by using the directive again in a child scope.

### 7.2.1 run_as

The user the app should be run as. This user should have the minimal amount of privileges necessary to successfully run the application (i.e. read-only access to the Shiny application directory). Note that this directive cannot be used with user_apps, as `user_apps` always run as the user who owns the application.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `users` | `String` | `multiple` | The username that should be used to run the app. If using 'user_dirs', this can also be a special keyword such as ':HOME_USER:' which will instruct 'user_dirs' to run as the user in whose home directory the application exists. Alternatively it could be ':AUTH_USER:' which will run the application as the user who the visitor is logged in as. If using a special keyword like ':HOME_USER:' or ':AUTH_USER:', you can specify an additional username afterwards which will be used when the special cases are not available, such as if the user is not logged in and the 'location' is not served by 'user_dirs'. | |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.2 license_type

License type to use. Available types are `traditional` and `floating`; default is `traditional`.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `type` | `String` | `required` | The type of license, traditional or floating. | |

Applies to: Top-level

Inheritable: Yes

### 7.2.3 access_log

The file path of the HTTP access log.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `path` | `String` | `required` | The file path where the access log should be written | |
| `format` | `String` | `optional` | The log file format; see the Connect documentation (http://www.senchalabs.org/connect/logger.html) under "Formats" | `default` |

Applies to: Top-level

Inheritable: Yes

### 7.2.4 server

Declares an HTTP server. You need one of these for each port/IP address combination this Shiny Server instance should listen on.

This directive has no parameters.

Applies to: Top-level

Inheritable: Yes

Child directives: listen, [server_name]

### 7.2.5 listen

Directs the enclosing server scope to listen on the specified port/IP address combination.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `port` | `Integer` | `required` | Port to listen on | |
| `host` | `String` | `optional` | IP address to listen on ('*' or '::' for all, or '0.0.0.0' for all IPv4); hostnames are not allowed, please use raw IPv4 or IPv6 address only. IPv6 zone IDs are not supported. | `*` |

Applies to: server

Inheritable: Yes

### 7.2.6 server_name

Directs the enclosing server scope to only honor requests that have the given host headers (i.e. virtual hosts).

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `names` | `String` | `multiple` | The virtual hostname(s) to bind this server to | |

Applies to: server

Inheritable: Yes

### 7.2.7 location

Creates a scope that configures the given URL as a website (site_dir), specific application (app_dir), autouser root (user_apps), autouser root with run_as support (user_dirs), or redirect to a different URL (redirect).

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `path` | `String` | `required` | The request path that this location should match | |

Applies to: server, location

Inheritable: Yes

Child directives: [run_as], location, [site_dir], [directory_index], [user_apps], [user_dirs], [app_dir], redirect, [log_dir], [log_file_mode], [members_of], application, [pam_sessions_profile], [exec_supervisor], [r_path], [bookmark_state_dir], [disable_websockets], [disable_protocols], [log_as_user], reconnect, [sanitize_errors], [frame_options]

### 7.2.8 site_dir

Configures the enclosing location scope to be a website that can contain both Shiny applications and unrelated static assets in a single directory tree.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `rootPath` | `String` | `required` | The path to the root directory of the website | |

Applies to: application, location

Inheritable: Yes

### 7.2.9 directory_index

When enabled, if a directory is requested by the client and an `index.html` file is not present, a list of the directory contents is created automatically and returned to the client. If this directive is not present in a custom config file, the default behavior is to disable directory indexes. However, it is enabled if no config file is present at all (in other words, the default config file has it enabled).

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `enabled` | `Boolean` | `required` | Whether directory contents should automatically displayed | |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.10 user_apps

DEPRECATED! This directive has been deprecated in favor of user_dirs, which offers more flexibility with regards to the run_as configuration. Configures the enclosing location scope to be an autouser root, meaning that applications will be served up from users' ~/ShinyApps directories and all Shiny processes will run as the user in whose directory the application is found.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `enabled` | `Boolean` | `optional` | Whether this location should serve up all users' /ShinyApps directories | `true` |

Applies to: location

Inheritable: Yes

### 7.2.11 user_dirs

Configures the enclosing location scope to be an autouser root, meaning that applications will be served up from users' ~/ShinyApps directories. This directive does respect an affiliated run_as setting, meaning that the applications will be executed as whichever user is configured in the applicable run_as setting. Note that many distributions, by default, will prohibit users from being able to access each other's home directories.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `enabled` | `Boolean` | `optional` | Whether this location should serve up all users' /ShinyApps directories | `true` |

Applies to: location

Inheritable: Yes

### 7.2.12 app_dir

Configures the enclosing location scope to serve up the specified Shiny application.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `path` | `String` | `required` | The path to the Shiny application directory | |

Applies to: location

Inheritable: Yes

### 7.2.13 redirect

Configures the enclosing location to redirect all requests to the specified URL.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `url` | `String` | `required` | The URL (or base URL) to redirect to | |
| `statusCode` | `Integer` | `optional` | The status code to send with the response (usually 301 for permanent redirects or 302 for temporary redirects) | `302` |
| `exact` | `Boolean` | `optional` | Whether to match on the URL exactly; if false, any subpaths will match as well | `true` |

Applies to: location

Inheritable: Yes

### 7.2.14 log_dir

Directs the application to write error logs to the specified directory. Only applies to location scopes that are configured with app_dir or site_dir, as user_apps (autouser) always writes error logs to `~/ShinyApps/log`.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `path` | `String` | `required` | The path to which application log files should be written | |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.15 log_file_mode

Specifies the file permissions to use for newly created application log files. Since Shiny Server v1.5.8, umask will be ignored; the mode will be applied via chmod.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `mode` | `String` | `optional` | The file mode to use, interpreted as an octal number. Set this to '0644' to allow all users on the system to read log files. | `0640` |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.16 members_of

Restricts a user_apps or user_dirs (autouser) scope to require membership in one or more groups (or, if no arguments are passed, lifts group restrictions from a members_of directive in a parent scope).

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `groups` | `String` | `multiple` | Users must be a member of at least one of these groups in order to deploy applications; if no groups are provided, then all users are allowed | |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.17 required_group

Restricts a location to require membership in one or more groups in order to access the application.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `groups` | `String` | `multiple` | Users must be a member of at least one of these groups in order to access the application | |

Applies to: Top-level, server, location, admin, application

Inheritable: Yes

### 7.2.18 required_user

Restricts a location to only be available to the specified users.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `users` | `String` | `multiple` | Users who are allowed to access the application | |

Applies to: Top-level, server, location, admin, application

Inheritable: Yes

### 7.2.19 auth_ignore_case

Enables case-insensitive matching on required_group and required_user; that is, `required_user John;` would match a user logged in as `john` if [auth_ignore_case] is present. This directive MUST NOT be used with auth systems that permit distinct users to have usernames that vary only by case (or for distinct groups to have groupnames that vary only by case), as this would make the required user/group security trivial to defeat for anyone who can create new users.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `enabled` | `Boolean` | `optional` | Whether case-insensitive matching should be used. Defaults to true, but only if 'auth_ignore_case' is present. | `true` |

Applies to: Top-level, server, location, admin, application

Inheritable: Yes

### 7.2.20 auth_passwd_file

Configures the server to use the specified file as the username/password store. The file should contain zero or more lines in the format `username + ":{scrypt}" + hashed-password`, where hashed-password is the scrypted password in base64 format. Use the `sspasswd` utility to create and modify passwd files.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `path` | `String` | `required` | The file path where the username/password file can be found | |

Applies to: Top-level

Inheritable: Yes

### 7.2.21 auth_proxy

Configures the server to use proxied authentication. In this mode, authentication is performed by an intermediary proxy between the user and Shiny Server Pro. The proxy is the one to handle authentication and will provide the username and groups of the current user in separate HTTP headers.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `userHeader` | `String` | `optional` | The name of the HTTP header containing the username of the current user. | `X-Auth-Usern` |
| `groupsHeader` | `String` | `optional` | The name of the HTTP header containing the groups for the current user. Groups should be comma-delimited. Leave this value empty if your proxy will not provide group information. | |

Applies to: Top-level

Inheritable: Yes

### 7.2.22 auth_duration

Approximate time in minutes that a user should be kept logged in, from the time that they finish using their last application. The default value is 120.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `minutes` | `Float` | `required` | The number of minutes; must be greater than or equal to 10 or applications may behave unpredictably | |

Applies to: Top-level

Inheritable: Yes

### 7.2.23 google_analytics_id

Configure Google Analytics tracking code to be inserted in Shiny application pages.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `gaid` | `String` | `required` | The Google tracking ID, for example, UA-18988-1 | |

Applies to: Top-level, server, location, application

Inheritable: Yes

### 7.2.24 app_init_timeout

Defines the amount of time Shiny Server will wait for an R process to start before giving up. Defaults to 60 seconds.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `timeout` | `Integer` | `required` | The number of seconds to wait for the application to start. | |

Applies to: Top-level, server, location, application

Inheritable: Yes

### 7.2.25 app_idle_timeout

Defines the amount of time an R process will persist with no connections before being terminated. Defaults to 5 seconds.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `timeout` | `Integer` | `required` | The number of seconds to keep an empty R process alive before killing it. | |

Applies to: Top-level, server, location, application

Inheritable: Yes

### 7.2.26 app_session_timeout

Defines how long an idle user's session should remain connected to the server. Defaults to 0.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `timeout` | `Integer` | `required` | The number of seconds after which an idle session will be disconnected. If 0, sessions will never be automatically disconnected. | |

Applies to: Top-level, server, location, application

Inheritable: Yes

### 7.2.27 http_keepalive_timeout

Defines how long a keepalive connection will sit between HTTP requests/responses before it is closed. Defaults to 45 seconds.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `timeout` | `Float` | `required` | The number of seconds to keep a connection alive between requests/responses. | |

Applies to: Top-level

Inheritable: Yes

### 7.2.28 http_allow_compression

Whether gzip/deflate compression is supported for HTTP responses. If this directive is not included, the default behavior is to support gzip/deflate compression.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `enabled` | `Boolean` | `optional` | Whether or not this is enabled. Default is true. | `true` |

Applies to: Top-level

Inheritable: Yes

### 7.2.29 sockjs_heartbeat_delay

How often the SockJS server should send heartbeat packets to the server. These are used to prevent proxies and load balancers from closing active SockJS connections. Defaults to 25 seconds.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `delay` | `Float` | `required` | The number of seconds to wait between heartbeat packets. | |

Applies to: Top-level

Inheritable: Yes

### 7.2.30 sockjs_disconnect_delay

How long the SockJS server should wait between HTTP requests before considering the client to be disconnected. Defaults to 5 seconds. If this value needs to be adjusted above 10 seconds, it's a good idea to disable websockets using the disable_websockets directive, as that transport protocol has an effective 10 second limit built in.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `delay` | `Float` | `required` | The number of seconds to wait before giving up. | |

Applies to: Top-level

Inheritable: Yes

### 7.2.31 simple_scheduler

A basic scheduler which will spawn one single-threaded R worker for each application. If no scheduler is specified, this is the default scheduler.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `maxRequests` | `Integer` | `optional` | The maximum number of requests to assign to this scheduler before it should start returning rejecting incoming traffic using a '503 - Service Unavailable' message. Once this threshold is hit, users attempting to initialize a new session will receive 503 errors. | `100` |

Applies to: Top-level, server, location, application

Inheritable: Yes

### 7.2.32 utilization_scheduler

A multi-process scheduler which will load-balance incoming traffic to multiple R processes. Each incoming request will be routed to the running worker with the fewest open connections, unless the request is explicitly targetting a particular worker.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `maxRequestsPerProc` | `Integer` | `required` | The maximum number of simultaneous requests (web socket or HTTP) per R process. | |
| `loadFactor` | `Float` | `required` | A decimal value in (0, 1] defining the capacity at which a new R process should be pre-emptively spawned. | |
| `maxProc` | `Integer` | `required` | The maximum number of concurrent R processes to run for this application. | |

Applies to: Top-level, server, location, application

Inheritable: Yes

### 7.2.33 ssl

The absolute paths to the files to use as the SSL key and certificate for this server.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `key_path` | `String` | `required` | The absolute path to the SSL key for this server. | |
| `cert_path` | `String` | `required` | The absolute path to the SSL certificate for this server. | |

Applies to: server, admin

Inheritable: Yes

Child directives: [ssl_min_version]

### 7.2.34 ssl_min_version

Specifies the minimum supported protocol version(s) of TLS for this server.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `version` | `String` | `required` | Valid values are 'tlsv1' (TLS v1.0), 'tlsv11' (TLS v1.1), and 'tlsv12' (TLS v1.2). SSL v2 and v3 are not supported. If the 'ssl_min_version' directive is not present, then a reasonable default will be chosen (currently 'tlsv11'). It is highly recommended that 'tlsv1' be avoided, as it is vulnerable to CBC attacks. | |

Applies to: ssl

Inheritable: Yes

### 7.2.35 allow_app_override

If present, will allow users to override the global defaults for a scheduler by customizing the parameters associated with a scheduler or even the type of scheduler used.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `enabled` | `Boolean` | `optional` | Whether or not this is enabled. Default is true. | `true` |

Applies to: Top-level

Inheritable: Yes

### 7.2.36 admin

If present, will provide an administrative interface on the specified port allowing specified users to monitor deployed Shiny applications.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `port` | `Integer` | `required` | The port on which the admin interface will be made available. | |

Applies to: Top-level

Inheritable: Yes

Child directives: ssl, [template_dir]

### 7.2.37 whitelist_headers

Enumerates the HTTP headers that should be trusted and passed from the original HTTP request to the Shiny application.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `headers` | `String` | `multiple` | The list of headers which should be trusted. | |

Applies to: Top-level, server, location, application

Inheritable: Yes

### 7.2.38 auth_google

Authenticate to a Google Application using a configured application ID and secret.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `id` | `String` | `required` | The Client ID available in your Google Apps Console. | |
| `secret` | `String` | `required` | The Client Secret available in your Google Apps Console, or the absolute path to the file storing the secret. If using a file path, Shiny Server will retain root privileges under the assumption that the file is only readable by the root user. | |
| `filters` | `String` | `multiple` | Used to limit the Google users who will be granted access to the system by filtering based on their email address. Filters may begin with a '+' or a '-' to indicate whether this is a 'positive' match (in which case these users should be granted access), or a 'negative' match (in which case any matching email should be rejected). These filters will iteratively be applied to the incoming email address until a match is found, at which point the user will either be admitted or rejected based on whether the filter was a 'positive' or 'negative' filter. A filter not beginning with a '+' or '-' is assumed to be a 'positive' filter. The asterisk can be used as a wildcard. | |

Applies to: Top-level

Inheritable: Yes

### 7.2.39 auth_pam

Authenticate using the local server's PAM configuration.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `forwardPassword` | `Boolean` | `optional` | If true, will retain the user's password when they login so that it can be provided to PAM when spawning processes on this user's behalf. This can be used where PAM features such as Kerberos or 'pam_mount' which require the user's password are in use. | `false` |

Applies to: Top-level

Inheritable: Yes

### 7.2.40 auth_ldap

Define how the LDAP (or Active Directory) server will be accessed.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `url` | `String` | `required` | The URL for the LDAP server and the root DIT of the directory. The URL can either use the unsecured 'ldap://' protocol, or the SSL-secured 'ldaps://' protocol, followed by the hostname or IP address of the LDAP server. Optionally, a port number can be provided after the LDAP server's address prefixed by a colon. If no port number is provided, the default port numbers of 389 and 636 for 'ldap://' and 'ldaps://' (respectively) will be used. This should be followed by a forward slash and the root DIT of the directory to use for authentication. For example, one configuration could be 'ldaps://ldap.exapmle.org:1636/dc=example,dc=org'. | |

Applies to: Top-level

Inheritable: Yes

### 7.2.41 auth_active_dir

Define how the Active Directory server will be accessed. This provides the same functionality as auth_ldap, but tailors the defaults to be appropriate for Active Directory.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `url` | `String` | `required` | The URL for the Active Directory (AD) server and the root DIT of the directory. The URL can either use the unsecured 'ldap://' protocol, or the SSL-secured 'ldaps://' protocol, followed by the hostname or IP address of the AD server. Optionally, a port number can be provided after the LDAP server's address prefixed by a colon. If no port number is provided, the default port numbers of 389 and 636 for 'ldap://' and 'ldaps://' (respectively) will be used. This should be followed by a forward slash and the root DIT of the directory to use for authentication. For example, one configuration could be 'ldaps://ad.example.org:1636/dc=example,dc=org'. | |
| `userSuffix` | `String` | `required` | The suffix (typically the domain name) to be appended to usernames when attempting to bind to the AD server. The default 'user_bind_template' will be set to '{username}@' followed by the userSuffix. This setting will be ignored if 'user_bind_template' is explicitly set. | |

Applies to: Top-level

Inheritable: Yes

Child directives: [ldap_timeout], [base_bind], [user_bind_template], [trusted_ca], [check_ssl_ca], [user_filter], [user_search_base], [group_search_base], [group_filter], [group_name_attribute]

### 7.2.42 ldap_timeout

Define the timeout for the parent LDAP connection. This timeout will be used when trying to connect to the LDAP server, and also when trying to query the server once connected. If not provided, a default timeout of 10 seconds will be used.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `timeout` | `Integer` | `required` | The number of seconds to wait for an LDAP query or connections before labeling it a failure. If 0, will never timeout. | |

Applies to: [auth_ldap], [auth_active_dir]

Inheritable: Yes

### 7.2.43 base_bind

Define credentials used to perform the initial LDAP bind request for double-bind authentication. If not provided, single-bind authentication is performed.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `user` | `String` | `required` | Defines a DN used during the initial LDAP bind request. The {root} variable will be automatically substituted by the root DIT (as provided in the parent 'auth_ldap' or 'auth_active_dir' directive) before attempting to bind to the LDAP server. | |
| `password` | `String` | `required` | The password used during the initial LDAP bind request. The password may be specified as plaintext or as an absolute path to the file storing the secret. If using a file path, Shiny Server will retain root privileges under the assumption that the file is only readable by the root user. | |

Applies to: [auth_ldap], [auth_active_dir]

Inheritable: Yes

### 7.2.44 user_bind_template

Define the template by which the username provided by the user should be converted to the username used to bind to the LDAP server. We currently only support storing all users in a single node of the directory.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `template` | `String` | `required` | The template used in defining the DN of users in the LDAP database when trying to bind. The following variables will be automatically substituted before attempting to bind to the LDAP server: {username} will be replaced by the username provided on the login page; {root} will be replaced by the root DIT (as provided in the parent 'auth_ldap' directive). If this setting is not provided, defaults will be used. For 'auth_active_dir' systems, the the default is '{username}@example.org' (where 'example.org' is the domain inferred/provided in 'auth_active_dir'). For other 'auth_ldap' systems, the default will be 'uid={username},ou=People,{root}'. | |

Applies to: [auth_ldap], [auth_active_dir]

Inheritable: Yes

### 7.2.45 trusted_ca

By default, many commonly accepted Certificate Authorities (CAs) will be trusted when establishing secure ldap (ldaps://) connections. This parameter instructs Shiny Server to instead trust this CA certificate. Organizations that are using an internal CA to sign their SSL certificates or are using a self-signed SSL certificate on their LDAP server will need to provide the associated CA certificate in order for Shiny Server to trust the ldaps:// connection.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `certificate` | `String` | `multiple` | The path to the CA certificate to be trusted in PEM format. | |

Applies to: [auth_ldap], [auth_active_dir]

Inheritable: Yes

### 7.2.46 check_ssl_ca

Define whether or not to check the Certificat Authority (CA) of the SSL certificate returned from the LDAP server. This must be true if using a self-signed SSL certificate. If not provided, this is set to 'true'. This should only be set to false if you know that your LDAP server uses an untrusted SSL certificate.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `value` | `Boolean` | `required` | If true, will check the certificate against the default list of trusted CAs (or the list provided in the 'trusted_ca' setting). If false, the returned SSL certificate will not be checked. | |

Applies to: [auth_ldap], [auth_active_dir]

Inheritable: Yes

### 7.2.47 user_filter

The filter to be used when querying the LDAP database to find the LDAP user object which contains the username. In Active Directory, the CN of the users are often not the username. In such instances, it is necessary to map the entered username to the user's DN. This will be done by querying in [user_search_base] for objects using this filter. If this field is left empty, it will be assumed that the name the user enters is the same as the associated LDAP user's CN, and can be used in the user's DN as instructed by [user_bind_template]. The following variables are available: {username} will be replaced by the username provided on the login page; {root} will be replaced by the root DIT (as provided in the parent auth_ldap directive); {userBind} will be replaced by the username after applying the `user_bind_template`. The default for this setting in `auth_ldap` is to leave this field empty; the default in [auth_active_dir] is 'userPrincipalName={userBind}'

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `filter` | `String` | `required` | The filter to use to find the LDAP user object referenced by the given username (as entered by the user on the login page). | |

Applies to: [auth_ldap], [auth_active_dir]

Inheritable: Yes

### 7.2.48 user_search_base

Set the subtree which will be the root of user searches. If this setting is omitted, the defaults of 'cn=Users' or 'ou=People' will be used for systems using [auth_active_dir] and auth_ldap, respectively. The given string will be followed by a comma and the root DIT to form the base of the search for users. If empty, the unaltered root DIT will be used. This setting will only be used if user_filter is set; otherwise, no queries for users will be executed.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `base` | `String` | `required` | The part of the tree which stores users. | |

Applies to: [auth_ldap], [auth_active_dir]

Inheritable: Yes

### 7.2.49 group_search_base

Set the subtree which will be the root of group searches. If this setting is omitted, the defaults of 'cn=Users' or 'ou=Groups' will be used for systems using [auth_active_dir] and auth_ldap, respectively. The given string will be followed by a comma and the root DIT to form the base of the search for groups. If empty, the unaltered root DIT will be used.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `base` | `String` | `required` | The part of the tree which stores groups. | |

Applies to: [auth_ldap], [auth_active_dir]

Inheritable: Yes

### 7.2.50 group_filter

The LDAP query to use in determining group membership. The query should return all groups of which the given user is a member. Three variables are available for this query: '{username}''will be replaced with the username entered at the login screen;'{userDN}'will be replaced with the user's full DN (using the [user_bind_template] to compute this); {root} will be replaced by the root DIT (as provided in the parent auth_ldap directive). For [auth_active_dir], the default is 'member:1.2.840.113556.1.4.1941:={userDN}'. For `auth_ldap`, the default is 'uniqueMember={username}'.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `filter` | `String` | `required` | The filter to use to determine group membership. | |

Applies to: [auth_ldap], [auth_active_dir]

Inheritable: Yes

### 7.2.51 group_name_attribute

The attribute of a group which contains the group name.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `attribute` | `String` | `optional` | The name of the attribute. Default is 'cn' for both 'auth_active_dir' and 'auth_ldap' | `cn` |

Applies to: [auth_ldap], [auth_active_dir]

Inheritable: Yes

### 7.2.52 application

DEPRECATED. This setting is deprecated and no longer enforced in Shiny Server. It is not permitted in configuration files.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `empty` | `String` | `required` | | |

Applies to: location

Inheritable: Yes

Child directives: [required_group], [required_user], [auth_ignore_case], [google_analytics_id], [app_init_timeout], [app_idle_timeout], [app_session_timeout], [simple_scheduler], [utilization_scheduler], [whitelist_headers]

### 7.2.53 template_dir

A directory containing custom templates to be used when generating pages in Shiny Server.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `dir` | `String` | `required` | The directory containing HTML templates. | |

Applies to: Top-level, server, location, admin

Inheritable: Yes

### 7.2.54 pam_sessions_profile

When spawning a Shiny process for a user, the PAM profile specified here will be used. The default is to use the 'su' profile. You can customize this behavior by creating a new profile in '/etc/pam.d' and referencing it here.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `profile` | `String` | `required` | The name of the profile in '/etc/pam.d' you want to use. | |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.55 exec_supervisor

The program supervisor under which Shiny processes should be run. This can be a command that modifies Shiny's environment or resources. The default is to have no supervisor (i.e. the empty string '').

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `command` | `String` | `required` | The command which prefixes the call to start the R process which will run Shiny. | |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.56 r_path

The path to the R executable.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `path` | `String` | `required` | The absolute path on disk to the R executable you wish to use. | |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.57 bookmark_state_dir

A directory for storing persisted application state. If Shiny Server is running without root privileges, then the run_as account must have read/write access to this directory. If no [bookmark_state_dir] directive is provided, `/var/lib/shiny-server/bookmarks` will be used.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `dir` | `String` | `required` | The directory where bookmark data should be stored. | |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.58 disable_websockets

Disable WebSockets on connections to the server. Some networks will not reliably support WebSockets, so this setting can be used to force Shiny Server to fall back to another protocol to communicate with the server. This is equivalent to adding 'websocket' to disable_protocols

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `val` | `Boolean` | `optional` | Whether or not WebSockets should be disabled. | `true` |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.59 rrd_disabled

If present, RRD logging will be disabled. Note that having RRD is a prerequisite for enabling the admin interface. While this option is present, no data will be logged to be viewed in the Admin interface in the future, either.

This directive has no parameters.

Applies to: Top-level

Inheritable: Yes

### 7.2.60 disable_protocols

Disable some of the SockJS protocols used to establish a connection between your users and your server. Some network configurations cause problems with particular protocols; this option allows you to disable those.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `names` | `String` | `multiple` | The protocol(s) to disable. Available protocols are: 'websocket', 'xdr-streaming', 'xhr-streaming', 'iframe-eventsource', 'iframe-htmlfile', 'xdr-polling', 'xhr-polling', 'iframe-xhr-polling', 'jsonp-polling' | |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.61 graphite_enabled

Enable logging of metrics to a Graphite server.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `host` | `String` | `optional` | The address of the Graphite server. Defaults to the local server. | `127.0.0.1` |
| `port` | `Integer` | `optional` | The port on the Graphite server to which we should send metrics. | `2003` |

Applies to: Top-level

Inheritable: Yes

### 7.2.62 preserve_logs

By default, log files from Shiny processes that exited successfully (exit status 0) will be deleted. This behavior can be overridden by setting this property to `true` in which case Shiny Server will not delete the log files from any Shiny process that it spawns. WARNING: This feature should only be enabled when combined with proper log rotation. Otherwise, thousands of log files could quickly accrue and cause problems for the file system on which they are stored.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `enabled` | `Boolean` | `optional` | Whether or not logs should be preserved. | `false` |

Applies to: Top-level

Inheritable: Yes

### 7.2.63 set_header

Add a custom header to all HTTP responses coming from the server.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `key` | `String` | `required` | The name of the header field. | |
| `value` | `String` | `required` | The value for the header. | |

Applies to: Top-level

Inheritable: Yes

### 7.2.64 log_as_user

By default, the log files for R processes are created and managed by the user running the server centrally (often root). In the typical scenario in which the logs are stored in a server-wide directory, this is desirable as only root user may have write access to such a directory. In other cases, such as using user_dirs on a system in which the users' home directories are on an NFS mount which uses `root_squash`, creating log files as root may be a problem. In those scenarios, this option can be set to true to have the log files created by the users running the associated processes.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `enabled` | `Boolean` | `optional` | Whether or not the log files should be managed by the owner of the process to which they belong. | `false` |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.65 reconnect

When a user's connection to the server is interrupted, Shiny Server will offer them a dialog that allows them to reconnect to their existing Shiny session for 15 seconds. This implies that the server will keep the Shiny session active on the server for an extra 15 seconds after a user disconnects in

case they reconnect. After the 15 seconds, the user's session will be reaped and they will be notified and offered an opportunity to refresh the page. If this setting is true, the server will immediately reap the session of any user who is disconnected.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `enabled` | `Boolean` | `optional` | Whether or not to offer to automatically reconnect disconnected users. | `true` |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.66 sanitize_errors

If this setting is true (the default), only generic error messages will be shown to the client (unless these were wrapped in `safeError()`).

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `enabled` | `Boolean` | `optional` | Whether or not to sanitize error messages on the client. | `true` |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.67 metrics_user

Shiny Server Pro will spawn a process to track and collect historical metrics data. If we're able to drop root privileges, then we'll spawn this process as the user that we're running the server daemon as. If we need to retain root privileges, then we'd use the `shiny` user to spawn this process. This setting will override that behavior, forcing Shiny Server to retain root privileges and to spawn the metrics process as the given user. You must ensure that the specified user has write access to all files in `/var/lib/shiny-server/monitor/` (or the custom value of `SHINY_DATA_DIR`) as well as the directory itself.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `user` | `String` | `required` | The username to use when spawning the metrics process. | |

Applies to: Top-level

Inheritable: Yes

### 7.2.68 disable_login_autocomplete

Adds the HTML attribute 'autocomplete=

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `enabled` | `Boolean` | `optional` | Whether or not to add the 'autocomplete="off"' attribute. | `true` |

Applies to: Top-level

Inheritable: Yes

### 7.2.69 frame_options

Sets the X-Frame-Options header on URLs served from Shiny applications, to prevent the app from being embedded in a browser frame or iframe. This can be used as a mitigation for clickjacking attacks. If no option is provided, the default behavior is `allow`.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `value` | `String` | `required` | One of 'allow', 'deny', 'sameorigin', or 'allow-from' (case insensitive). ('allow' removes the X-Frame-Options header.) | |
| `url` | `String` | `optional` | If 'value' is 'allow-from', the URL from which framing should be allowed. | |

Applies to: Top-level, server, location

Inheritable: Yes

### 7.2.70 auth_frame_options

Sets the X-Frame-Options header on login-related URLs served from Shiny Server Pro, to prevent login pages from being embedded in a browser frame or iframe. This protects from clickjacking attacks. If no option is provided, the default behavior is `deny`.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `value` | `String` | `required` | One of 'allow', 'deny', 'sameorigin', or 'allow-from' (case insensitive). ('allow' removes the X-Frame-Options header.) | |
| `url` | `String` | `optional` | If 'value' is 'allow-from', the URL from which framing should be allowed. | |

Applies to: Top-level

Inheritable: Yes

### 7.2.71 secure_cookies

Adds the `secure` flag to HTTP cookies set by Shiny Server Pro; by default, this flag is never sent. This directive should only be used if all authenticated apps are to be served via HTTPS only.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `policy` | `String` | `required` | Currently, only the value 'always' is accepted for this parameter. | |

Applies to: Top-level

Inheritable: Yes

### 7.2.72 group_list

Restricts the set of groups that will be applied to a logged in user. Shiny Server Pro currently has a bug where a user being a member of too many groups will cause login to fail. This command can be used to narrow down large group lists to sizes Shiny Server Pro can handle.

| Parameter | Data type | Type | Description | Default |
|---|---|---|---|---|
| `command` | `String` | `required` | One of 'include', 'exclude', 'include_glob', 'exclude_glob', 'include_regex', 'exclude_regex'. | |
| `pattern` | `String` | `multiple` | The group name, glob pattern, or regex indicating the group(s) to include or exclude. | |

Applies to: Top-level

Inheritable: Yes

## 7.3 Migrating & Updating

### 7.3.1 Upgrading from Shiny Server Pro 1.4.5.x and older

#### 7.3.1.1 Error messages now sanitized by default

In previous versions of Shiny Server Pro, if an error occurred in a Shiny app while processing an output, the error message would be shown in place of the output. Due to concerns that such error messages might include sensitive data (such as file paths that reveal the directory structure of the server), Shiny Server Pro 1.4.6+ in conjunction with Shiny 0.14+ will now "sanitize" such errors by default, replacing the specific error message with a generic one like "An error has occurred. Check your logs or contact the app author for clarification."

If you prefer the old behavior, you can add a `sanitize_errors off;` directive to your `/etc/shiny-server/shiny-server.conf` config file. Adding this directive at the top level will restore the old behavior for the entire server. You can also use the directive at the server or location level if only certain applications should show specific error messages.

Alternatively, the app code itself can opt out of sanitized errors by adding the line `options(shiny.sanitize.errors = FALSE)` in `global.R` or at the top of `app.R`.

See http://shiny.rstudio.com/articles/sanitize-errors.html for more information.

#### 7.3.1.2 Custom login templates now must include a CSRF token field

Shiny Server Pro 1.4.6 includes mitigations against CSRF attacks. For the most part, this should not affect your installation of Shiny Server Pro. The exception is if you are using a custom template for your login page. In that case, you'll need to add a new field to your login `<form>` tag:

```
<input id="csrfToken" type="hidden" name="_csrf" value="{{csrf_token}}"/>
```

### 7.3.2 Upgrading from Shiny Server 0.4.x

There are problems on some systems with updating from 0.4.x versions to a more recent version. Thus, any user updating from the 0.4.x line should uninstall the package before installing the new one. In all versions 0.5.x and later, updating can simply be done by installing the newer version over the old one without explicitly removing the old package.

### 7.3.3 Upgrading from Shiny Server 0.3.x

Shiny Server 0.4 and later include many major changes to the architecture of Shiny Server. There are a few things to be aware of when upgrading from Shiny Server 0.3.x.

First, Shiny Server is now distributed via `deb` and `rpm` installers, rather than using `npm`. Therefore, you must first uninstall the old version of Shiny Server that was installed with `npm`.

```
$ sudo npm uninstall -g shiny-server
```

Shiny Server no longer requires node.js to be installed on the system, either. If you have no other need for node.js, you can uninstall it at this time, as well.

Running the latest Shiny Server installer will replace the scripts at `/etc/init/shiny-server.conf` or `/etc/init.d/shiny-server`. If you have made any modifications to these startup scripts that you wish to keep, save a copy of these files before running the installer.

The default directories for logging and Shiny application hosting have also moved in this release. The default logging directory used to be `/var/shiny-server/log`; it is now `/var/log/shiny-server`. The default hosting directory was previously `/var/shiny-server/www` and is now `/srv/shiny-server/`. If you have not explicitly overridden these directories in the `/etc/shiny-server/shiny-server.conf` file, the new location will be created and used in the latest Shiny Server.

Finally, Shiny Server 0.4 and later require the Shiny package to be at least version 0.7.0. You can check the system-wide version of Shiny you have installed using the following command:

```
$ sudo su - -c "R -e \"packageVersion('shiny')\""
```

(Running this command using the `sudo su - -c` preface will allow you to see the system-wide version of Shiny. Individual users may have installed more recent versions of Shiny in their own R libraries.) If the installed version of Shiny predates 0.7.0, you should follow the instructions in Install Shiny to update to the most recent version.

At this point, you can proceed with the Installation instructions associated with your Operating System.

## 7.4 Frequently Asked Questions

### 7.4.1 What are the hardware requirements for running Shiny Server?

The performance footprint of a Shiny application is almost entirely dependent upon the Shiny application code. There are two factors to consider when selecting the hardware platform for Shiny Server.

**RAM**

The memory requirements of a Shiny application depend heavily on the amount of data loaded when running the application. Users often find that a Shiny R process requires a minimum of 50MB

of RAM – beyond this, the amount of memory consumed by an application is determined by the data loaded or generated by that application. The Scoping Section of the Shiny Tutorial describes in detail how to take advantage of Shiny's scoping rules to share data across multiple Shiny sessions. This enables application developers to load only one copy of data into memory but still share this data with multiple shiny sessions.

**CPU**

R is fundamentally a single-threaded application. Unless parallel packages and tools are specifically selected when designing a Shiny application, the R process (and the associated Shiny application) will be run serially on a single processing core. Therefore, the typical Shiny application may saturate the processing core to which it is assigned, but will be unable to leverage other cores on the server that may be idle at that time.

### 7.4.2 Does Shiny Server work on Amazon Elastic Compute Cloud (EC2)?

Absolutely. Many customers deploy Shiny Server and Shiny Server Professional on the Amazon Web Services (AWS) framework.

### 7.4.3 Can I Deploy Shiny Server on an Isolated Network?

Yes. Shiny Server does not require a connection to the Internet to work properly, so you are free to deploy it in whatever network configuration you prefer. Offline activation is also available for Shiny Server Professional customers.

### 7.4.4 How Do I Translate My 'application' Settings to Nested 'location's?

Shiny Server 0.4.2 introduced a new model for managing applications at a more granular level by supporting nested `location` directives. Any existing `application` directives in your configuration file will need to be migrated to this nested location format in order to be supported by any version of Shiny Server after 0.4.2.

`application` directives were addressed by their full path on disk. Nested `locations`, on the other hand, are indexed by their relative path within their parent location. For instance, the following exemplifies how you could redirect a particular application in the old model:

```
location / {
  site_dir /srv/shiny-server;

  # Define rules to apply to one application, stored at '/srv/shiny-server/app1'
  application /srv/shiny-server/app1 {
    redirect "http://rstudio.com" 302 true;
  }
}
```

In the new "nested locations" model, the equivalent would be:

```
location / {
  site_dir /srv/shiny-server;

  # Define rules to apply to one application, stored at '/srv/shiny-server/app1'
  location /app1 {
    redirect "http://rstudio.com" 302 true;
  }
}
```

This structure is much more powerful than simple `application` settings. The following configuration defines a parent location (`depts`) with some settings, then overrides those settings for a particular directory (`finance`) within that location.

```
server {
  ...
  # Define the '/depts' location
  location /depts {
    # Host a directory of applications
    site_dir /srv/departmentApps;

    # Provide a default/global GAID
    google_analytics_id "UA-12345-1";

    # Define the '/finance' location.
    # Corresponds to the URL '/depts/finance', and the filesystem path
    # '/srv/departmentApps/finance' as defined by the parent location.
    location /finance {
      # Provide a custom GAID for only this sub-location
      google_analytics_id "UA-54321-9";

      location /fall2014/app1 {
        redirect "http://rstudio.com" 302 true;
      }
    }
  }
...
```

In this case, all applications stored in the `depts/finance/` directory would have a different Google Analytics ID than the rest of the applications on this server. Additionally, the application at `depts/finance/fall2014/app1` would be redirected to `http://rstudio.com`.

### 7.4.5 When I open a lot of Shiny Docs at once in Firefox, why do some not open?

Firefox limits the persistent connections that the browser can have open to a single server. For most use cases, this will not be a problem. But a user who opens multiple Shiny Docs with many embedded Shiny components all at once may hit this limit.

To resolve this problem, increase the network.http.max-persistent-connections-per-server setting (we tested by setting it to 50). Once that limit is increased, you should be able to open many complex Shiny Docs simultaneously from Firefox without issue.

### 7.4.6 Can I Bulk-Import in Flat-File Authentication?

Some users may have an existing table of usernames and passwords they wish to import into a flat-file authentication system using the `sspasswd` tool. The shell script below can be used as a model for scripting such a solution, but be aware of the security concern mentioned below.

```
#!/bin/bash

PWFILE="passwdfile"
USERFILE="users.txt"

if [ ! -f "$PWFILE" ]; then
  touch $PWFILE
fi

while read p; do
  parts=( $p )
  echo "Adding ${parts[0]}..."
  echo "${parts[1]}" | /opt/shiny-server/bin/sspasswd \
    $PWFILE "${parts[0]}"
done < $USERFILE
echo "Done."
```

The script above will extract the usernames and passwords from the file specified in the `USERFILE` variable, and will use the file name provided in the `PWFILE` variable as the file in which the usernames and passwords should be stored. This script expects a user file in which the values are stored in a tab-delimited format, with the username in the first column and the password in the second.

Most secure password-generating tools recommend that the password be typed in an interactive console in which the text can be hidden, rather than passing it in via a command-line argument. There are two main reasons for this:

1. Such commands (including the password, if it were provided as an argument to the command) are stored in the history file for your shell. Thus, a user with access to your shell history may be able to retrieve the password.

2. In a model like the one used above, the passwords will not be stored in your shell's history, but they will be available in the system's process table. The `echo` command above (and the password associated with it) will likely be accessible by any other user on the system while that command is executing.

Be sure you fully understand the security implications of the above script before using it, especially if using it on an unsecured or multi-user server.

### 7.4.7 How Can I Set The trusted_ca For My LDAP Server Over SSL?

The easiest way to confirm an SSL connection is to use the `openssl` tool to connect to your LDAP server. If you do not already have the SSL certificates for your server, you can download them using this tool. If you run

```
openssl s_client -connect <LDAP server address>:<port> -showcerts
```

you should get significant output. (The default LDAPS port is 636.)

If you review this output, in particular the last few lines, you should see a "result". If there is a problem, it may say something like `Verify return code: 19 (self signed certificate in certificate chain)`, which indicates that there is an issue with trusting the SSL connection between you and your LDAPS server. If you see an error like the one above, you need to instruct your client to trust a particular Certificate Authority (CA) that the `openssl` tool does not trust by default. Once you retrieve the CA certificate for your organization (which should also be the last certificate returned by the command above if you are actually connected to the right server), you can tell `openssl` to trust that CA by using a command in the format of

```
openssl s_client -connect <server-address>:<port> -CAfile <file.pem>
```

Assuming that the certificate matches the CA you provide, and that everything is in the right format, you should get a line of output from `openssl` that says, `Verify return code: 0 (ok)`. Once you see that, you know you have your CA certificate in the right format.

There is one important check that the `openssl` tools does not perform that you should do before trying to use the certificate in Shiny Server Pro. You will need to confirm that the hostname you are using matches the SSL certificate. You can do that manually, or use `curl` by running `curl --cacert <file.pem> ldaps://<server-address>:<port>/`. If you see some LDAP output,perhaps starting with `DN:`, and no errors, then things are working properly and you have the right hostname.

Once you have the CA certificate working in the above tests, then you are ready to apply it to Shiny Server Pro. The CA certificate should be in PEM format and *only include one certificate per file*. You can add these file references using the `trusted_ca` setting in your Shiny Server Pro configuration as follows:

```
auth_ldap ... {
  trusted_ca /etc/ssl/ca1.pem /etc/ssl/ca2.pem;
}
```

This example setting includes two CA certificates that Shiny Server Pro should trust. Shiny Server Pro should now be able to connect to your LDAPS server successfully when you attempt to authenticate users.