# Chapter 12. Backups

If one relies solely on marketing materials provided by cloud hosts and resellers, one might be forgiven for thinking that the cloud is a magical place where nothing breaks and everything Just Works. Unfortunately, this is not the case. Cloud-based infrastructure requires just as much backup planning as a traditional self-hosted architecture—sometimes more, because of the dynamic nature of the cloud. Fortunately there is a silver lining: along with new challenges, the cloud provides new features to make backups simpler and reduce implementation time.

Although business types can think of the cloud as a single logical entity, we must look beyond that high-level presentation to view our cloud as a series of datacenters spread across multiple regions, and plan our backups accordingly. To run a highly available service, you would not put all of your servers in a single datacenter. You should plan backups with the same consideration in mind.

Furthermore, it is also important to think about off-site backups. When working with AWS, this can mean one of two things: storing backups outside your primary region, or going a step further and storing them entirely outside AWS.

You are trying to protect yourself against two separate risks. If a single AWS region goes down, it would be relatively simple to deploy your operations to another region. For example, if you host your application in us-east-1 and store your backups in eu-west-1, you can redeploy your application to us-east-2, and restore servers from the backups ineu-west-1 to get your application up and running again.

However, if the AWS API is unavailable, it can become impossible to retrieve these backups, no matter which region they are hosted in, rendering them useless in protecting against this particular failure mode.

Backups are a means to an end, not an end in themselves. What we are trying to achieve is a way to restore operations in the event of failure, no matter the cause or severity of this failure. Unless your backups put you in a position where you can restore after failures, they are of no use whatsoever.

An untested backup procedure is useless. In fact, an untested backup procedure can be worse than no backups at all, as it provides a false sense of security. Perform regular restore tests to make sure that your documented backup procedure works as planned.

This chapter presents some of the ways traditional tools and AWS-specific features can be used to implement reliable backup procedures.

##### RDS DATABASE SNAPSHOT

If you are using Amazon’s RDS service to host your database, you can use the RDS Database Snapshot feature. This process can be automated so that Amazon automatically backs up your database according to your specified schedule, or you can manually create backup snapshots before performing potentially destructive operations.

When you use RDS, automated snapshots are automatically enabled and will be carried out during the maintenance window chosen when creating the database instance. The process of enabling and disabling automated backups is described in Amazon’s [Working With Automated Backups](http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_WorkingWithAutomatedBackups.html) documentation. You can find a more general explanation of how automated backups work in [DB Instance Backups](http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Overview.BackingUpAndRestoringAmazonRDSInstances.html). The Related Topics section of the latter page provides more detail on working with automated backups.

If you rely on RDS snapshots, it is important to keep track of the most recent snapshot ID when backing up the other files required for your application to ensure that the database schema referenced in your code matches the data contained in the snapshot. This can be done by regularly querying the ID of the most recent snapshot and storing it in a text file alongside the other application files and making sure it is included in the backup. When restoring from backups, this file will let you know which corresponding DB snapshot should be restored.

Finally, even if you are using RDS, you might wish to follow the other steps in this chapter to regularly take a full dump of the database. This will ensure that you can restore the backup to a non-RDS database. RDS snapshots can be used only to restore to RDS databases, and do not provide any facility to make off-site backups.

At the time of writing, RDS supports MySQL, Oracle, Microsoft’s SQL Server, and PostgreSQL, all of which can be used with automated backups. Because PostgreSQL is a relatively recent addition to the list of supported database engines, a lot of people are still running their own PostgreSQL database clusters on EC2 instances, outside RDS. Later in this chapter, we will look at ways to manually back up databases that are not part of RDS.

# Backing Up Static Files from EC2 Instances to S3

One common use of EC2 is to host web applications such as WordPress blogs. This section describes how an EC2-hosted WordPress blog can be backed up, with the backup archive being stored on S3. Although the steps taken will differ if you are using a different application—or indeed, your own custom application—the general steps will be the same.

Dynamic websites usually consist of two major components that need to be backed up: the database in which content and configuration options are stored, and static files such as images and HTML files. Furthermore, these websites might allow users to upload their own files, such as profile images for blog authors. All of these components need to be backed up to ensure that the entire site can be restored in the event of an EC2 instance failing.

If you are backing up a custom application, I’ll assume that the code that powers your application is stored in an external source control system such as GitHub. Therefore, backing up these files is outside the scope of this section. If you are not using version control for your application, you could back up the code files as part of the file-based backup archive.

The first step in the backup process is to create a snapshot of your database contents.

For the sake of simplicity, we will assume the WordPress blog is configured to use a MySQL instance running on the same host as WordPress. If your database is hosted on a separate instance, you will need to modify some of the example commands to connect to the correct host in order to dump the contents of the database.

We will begin by creating a database dump file containing the SQL statements required to re-create your database. MySQL helps you do this with the conveniently named mysqldumpcommand. For example, if your WordPress database is named my\_blog, the following command can be used to dump its contents to a file located at /var/backups/my\_blog/database.sql:

mysqldump my\_blog > /var/backups/my\_blog/database.sql

After the database dump has completed, you can move on to backing up the application files. Let’s assume they are stored in /srv/vhosts/my\_blog. First, copy all the files into the backups directory with the following command:

rsync -av /srv/vhosts/my\_blog/ /var/backups/my\_blog/

This command will copy all files. The -a option indicates that rsync should run in archive mode, which, among other things, ensures that file permissions and ownerships are copied.

To reduce the amount of data stored in S3, you can create a compressed tar archive before transferring it to S3. This is done with the following:

DATE='date -u +"%Y%m%d%H%M"'

BACKUP\_FILE="/var/backups/my\_blog\_${DATE}.tgz"

tar zcvf ${BACKUP\_FILE} /var/backups/my\_blog/

Finally, you can transfer the resulting file to S3 with the s3cmd command:

s3cmd put "${BACKUP\_FILE}" s3://my-blog-backups/

To keep the WordPress instance clean, delete the temporary files used during the backup:

rm -rf /var/backups/my\_blog/\*

This last step is optional. Leaving these files in place would speed up subsequent backups, because rsync would have a smaller amount of data to transfer, needing to transfer only files that had changed since the last backup.

Put these commands together to make a script that can be easily used to automatically back up the database and files required to recover your blog in the event of a disaster.

Restoring the resulting archive is simply a case of extracting the files to the correct location and importing the SQL statements file into the database. For example:

cd /srv/vhosts/myblog

tar xcvf my\_blog\_201206011200.tar.gz

mysql my\_blog < myblog/database.sql

rm myblog/database.sql

Later in this section, we will look at ways to move this data out of S3 and into off-site storage for additional reliability.

# Rolling Backups with S3 and Glacier

When keeping backups on S3, it is important to keep track of how much data you are storing. Each additional byte will gradually increase your monthly Amazon bill, so there is no point in storing data that you will never need to use. Your backup strategy should reflect this. In many ways, the traditional approaches used with tape backups are still applicable: one tape per day of the week, one tape for each of the last four weeks, and finally one tape for each month. A total of 23 tapes would allow you to restore from any day of the previous week, any week of the previous month, and any month of the previous year.

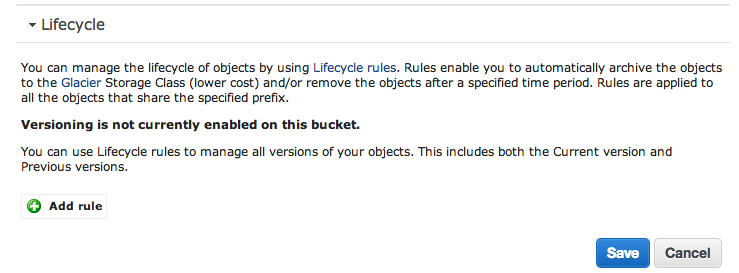
A similar approach can also be used in S3 to keep your data storage requirements to a minimum. In this section, we will look at how this can be implemented using S3’s Object LifeCycle Management and Object Archival features. This method relies on S3 for the last month’s worth of backups, and monthly backups are automatically transitioned to the Glacier archival service on the assumption that they will be required less frequently than daily or monthly backups.

Let’s assume that you are using one of the methods described in this chapter to store your backups in S3. We will configure S3 to automatically transition backup objects to Glacier after they reach a certain age and delete them automatically from S3. This will keep your S3 backups bucket clean and ensure that you do not gradually build up a huge Amazon bill for storing unnecessary files.

For this example, we will assume that your daily backup archives are prefixed with /backups/daily/ in your S3 bucket—for example, /backups/daily/20131003.tar.gz.

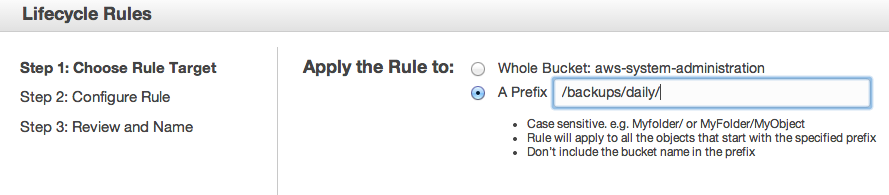
The rules that govern when S3 objects are transitioned to Glacier are stored as a life cyclesubresource on your S3 bucket. A life cycle configuration can contain up to 1,000 rules controlling when your files—or subsets thereof—are transitioned to Glacier. Life cycle configurations can be created and modified using the AWS API, or via the AWS Management Console. In this example, we will use the Management Console to create our lifecycle configuration.

Begin by finding your S3 bucket in the AWS Management Console. Right-click the bucket object and select Properties; then navigate to the Lifecycle tab shown in [Figure 12-1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch12.html#lifecycle_properties).



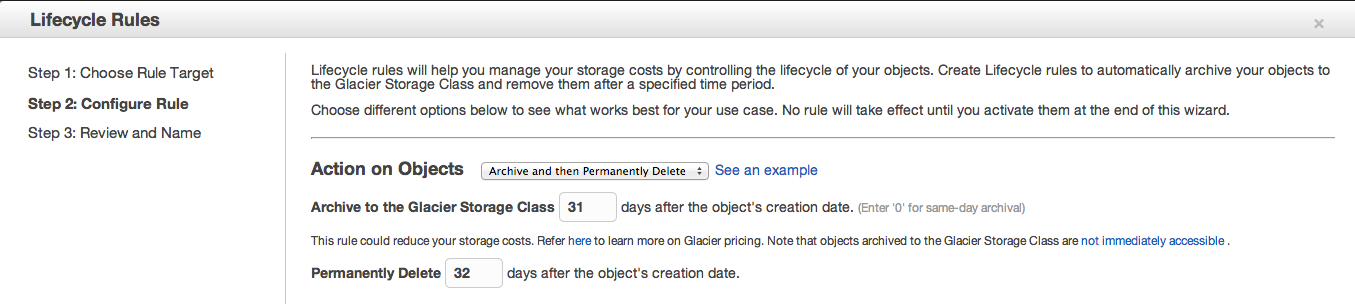
###### *Figure 12-1. Lifecycle Properties*

Click Add Rule to open the Lifecycle Rules Wizard shown in [Figure 12-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch12.html#lifecycle_add_rule_1).



###### *Figure 12-2. Choose Rule Target*

The first screen lets us choose which objects the chosen action will be applied to. In the Prefix box, enter /backups/daily/, as we want this particular rule to apply only to the daily backup archives. Click Configure Rule to proceed to the next screen shown in [Figure 12-3](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch12.html#lifecycle_add_rule_2). This screen lets us choose what action will be performed on the objects. Select Archive and then Permanently Delete to see the options relevant to the rule we are creating.



###### *Figure 12-3. Configure Rule*

Our desired rule consists of two actions: transitioning the object to Glacier and removing the original S3 object after it has been transitioned to Glacier. Because this is such a common pattern, Amazon allows us to configure both of these actions from a single screen.

Enter 31 in the Archive to the Glacier Storage Class input box, and 32 in the Permanently Delete input box as shown in [Figure 12-3](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch12.html#lifecycle_add_rule_2).

In this example, both of these time values should be set to 31 days. Life cycle rules can be set only in daily increments, so we set this to 31 to ensure that we do not transition objects too early in months with fewer than 31 days. The object cannot be deleted before it is archived, so we must wait a day between archiving the objects to Glacier and deleting the original objects from the S3 bucket.

Click Review to confirm the options you have selected. On this screen, you can provide the rule with an optional name. Finally, click Create and Activate Rule.

You can now store your backups in S3, and all objects will be moved to Glacier after they are older than 31 days.

At the time of writing, there is no equivalent for deleting or moving objects out of Glacier, which means you will still need to manually remove objects from your Glacier account after you are sure they are no longer required. Manually in this case might mean deleting the objects via the Glacier Management Console or writing a script that searches for archives older than (for example) six months and deleting them.

# PostgreSQL and Other Databases

Amazon offers hosted and managed PostgreSQL databases via its RDS service. For a long time, RDS supported only MySQL databases, so it was common for PostgreSQL users to run their own PostgreSQL instances on EC2 instances. This also meant that users were responsible for providing their own backup procedures instead of relying on the backup features provided by RDS. For this reason, this section goes into some detail about the manual process of backing up databases that are hosted on EC2 instances, as opposed to running on RDS. While this section uses PostgreSQL-specific commands, the general principles are applicable to most database engines.

Backing up dynamic databases is not as simple as copying the database files to a remote location. First, PostgreSQL must be informed that a database backup is about to executed. If you forget this step, the database files could be copied while they are effectively in an unusable state, making restoration procedures either very time-consuming or in some cases impossible. PostgreSQL must be informed that a database backup is about to be executed so that it can flush any in-memory data to disk and ensure that the files on disk are in a state that will allow them to be used to restore data.

There are two main methods of backing up a PostgreSQL database running on EC2: pg\_dump and snapshots. The latter is more complicated, but better for large databases.

## pg\_dump

pg\_dump and its companion commands are distributed with PostgreSQL. pg\_dump dumps all the data from the specified database, without interfering with reads and writes. Any changes made to the database after you start pg\_dump will not appear in the dump, but the dump will be consistent. Data can be dumped in a variety of formats, which can be restored with the pg\_restore command. The default option is to create a file containing SQL statements, but it is also possible to dump data in a PostgreSQL-specific format or create a tar archive of the data.

This method is especially suited for smaller databases (less than 5 GB) because the resulting dump file can be transferred directly to S3. For example, to dump the my\_db database and store it in an S3 bucket named my-db-backups, you could use the following script:

#!/bin/bash -e

DB\_NAME="my\_db"

BUCKET\_NAME="my-db-backups"

DATE=`date -u +"%Y%m%d%H%M"`

BACKUP\_DIR="/var/backups"

BACKUP\_FILE="${BACKUP\_DIR}/${DB\_NAME}\_${DATE}.tar"

mkdir -p $BACKUP\_DIR

# Dump the database as a tar archive

pg\_dump ${DB\_NAME} --file="${BACKUP\_FILE}" --format=tar

# Copy the tar file to S3

s3cmd put "${BACKUP\_FILE}" s3://${BUCKET\_NAME}

# Delete the local tar file

rm ${BACKUP\_FILE}

This script first dumps the data from PostgreSQL using the pg\_dump command. Once the backup file has been created, it is copied to an S3 bucket. Finally, the local copy of the tar file is deleted to ensure that you do not gradually use up all the space available on the device on which /var is mounted. Bash’s -e option is used to ensure that the script fails immediately if any of the commands in the script fails.

To restore a database backed up using this script, simply copy the backup file from the S3 bucket onto the new PostgreSQL instance and use the pg\_restore command to load the data into the new database. For example:

pg\_restore --dbname=my\_db /path/to/backup/file.tar

As your database grows, the time taken to run the pg\_dump and pg\_restore commands will increase, making this option less attractive. If you want to make backups once per hour, this process will become useless as soon as it takes longer than 60 minutes to complete, because you will never be able to complete a backup before a new one is started.

## Snapshots and Continuous Archiving

Another option for backing up EC2-hosted PostgreSQL databases is to use the snapshotting capabilities of EBS volumes. This is slightly more complex, but provides a much quicker way of backing up larger databases. This method does not produce a single file that can be used with the pg\_restore command. Instead, it uses PostgreSQL’s [base backup](http://www.postgresql.org/docs/9.3/static/continuous-archiving.html#BACKUP-BASE-BACKUP) feature to produce one or more EBS snapshots that can be used to provision new EBS volumes containing your backed-up data.

This method relies on PostgreSQL’s continuous archiving features. Configuring this is beyond the scope of this book. Refer to the PostgreSQL documentation on [Continuous Archiving](http://www.postgresql.org/docs/9.3/static/continuous-archiving.html) for information on how to configure and test this feature.

In a nutshell, continuous archiving will periodically store your data in an external location so that it can be used for a restore later. This is done by archiving the write-ahead log (WAL) files, which essentially play back all operations performed on your database. This allows you to restore the database to a time of your choosing (point-in-time recovery). However, playing back all the WAL files can take some time. To reduce restoration time, create a base backupthat is used as a starting point for recovery. This allows you to restore a smaller number of WAL files in external storage and play back only WAL files that were created after a particular base backup was taken.

###### TIP

WAL-E is a program designed to help create and manage PostgreSQL WAL files and create base backups. Although it is still worth learning and understanding how the underlying concepts of WAL and base backups work, WAL-E can make the day-to-day usage of continuous archiving a lot simpler and more reliable. It can be found at [GitHub](https://github.com/wal-e/wal-e).

For performance reasons, it is recommended that you put PostgreSQL’s data directory on its own EBS volume. The data directory location will vary depending on the version of PostgreSQL and the operating system in use. For example, a PostgreSQL 9.4 instance on Ubuntu will be stored in /var/lib/postgresql/9.3/main. Attaching an additional EBS volume to your instance and mounting the data directory on this volume will improve performance, because PostgreSQL will not be contending with the operating system in order to read and write files.

###### TIP

PostgreSQL’s tablespace feature allows you to store each table in a separate on-disk location. This feature makes it possible to store each table on a different EBS volume, further improving performance. In conjunction with EBS Provisioned IOPS, which provide a higher level of performance than vanilla EBS volumes, this can dramatically improve the performance of disk-bound workloads.

### BACKING UP

This section, and the following, assume that you have at least two EBS volumes attached to your PostgreSQL EC2 instance. The first volume is used to mount the root of the filesystem (the / directory). The second volume is used solely for PostgreSQL’s data directory and is mounted at /var/lib/postgresql/9.3/main. For example purposes, the data EBS volume will be created with the device name /dev/sda2, a typical name for a Linux device responding to the popular SCSI protocol. It is further assumed that you have installed PostgreSQL and created a database for testing purposes according to PostgreSQL’s documentation.

Begin by connecting to the PostgreSQL instance as the superuser by running the psql command. Depending on how you configured your database, you might need to provide the superuser password at this point. Once connected to the database, issue the pg\_start\_backup command:

SELECT pg\_start\_backup('test\_backup');

This command will create a backup label file in the data directory containing information about the backup, such as the start time and label string. It is important to retain this file with the backed-up files, because it is used during restoration. It will also inform PostgreSQL that a backup is about to be performed so that a new checkpoint can be created and any pending data can be written to disk. This can take some time, depending on the configuration of the database.

Once the command completes, make a snapshot of the EBS volume on which the PostgreSQL database is stored.

###### NOTE

The filesystem type you are using might require a slightly different approach at this stage. This example assumes you are using XFS, which requires that the filesystem be frozen before a snapshot is made. This ensures that any attempts to modify files on this filesystem will be blocked until the snapshot is completed, ensuring the integrity of the files.

First, freeze the filesystem with the xfs\_freeze command:

xfs\_freeze -f /var/lib/postgresql/9.3/main

Once the filesystem has been frozen, it is safe to take a snapshot using the EBS snapshotting tools. This can be done from the AWS Management Console or from the command-line tools. Because the final goal is to use this process as part of an automated backup script, we will use the command-line tools:

# Find out the ID of the instance on which this command is executed

EC2\_INSTANCE\_ID=$(ec2metadata --instance-id)

# The device name is used to filter the list of volumes attached to the instance

DEVICE="/dev/sda2"

# Find the ID of the EBS volume on which PostgreSQL data is stored

VOLUME\_ID=(`ec2-describe-volumes --filter="attachment.instance-id=${EC2\_INSTANCE\_ID}" --filter="attachment.device="${DEVICE} | awk '{ print $2 }'`)

ec2-create-snapshot ${VOLUME\_ID}

These commands will query the EC2 API to find out the volume ID of the EBS volume containing the PostgreSQL data before using the ec2-create-snapshot command to create a snapshot of this volume. The ID of the new snapshot will be printed as part of the output from ec2-create-snapshot.

Although the snapshot has not yet been fully created, it is safe to begin using the volume again. AWS will create a snapshot of the volume in the state it was in when the ec2-create-snapshot command was executed, even if data is subsequently changed.

As soon as the snapshotting command finishes, the XFS volume can be unfrozen:

xfs\_freeze -u /var/lib/postgresql/9.3/main

Finally, you can inform PostgreSQL that the backup has been completed and that it should resume normal operations:

SELECT pg\_stop\_backup();

Note that the backup is not entirely complete until the WAL archives identified by the pg\_stop\_backup command have been archived (by way of the archive\_commandconfiguration directive in postgresql.conf).

### RESTORING

Restoring from a backup created with this method is much simpler, but more time-consuming. Whether your original PostgreSQL instance has suffered from some kind of catastrophic failure or you are restoring the data to a new machine (perhaps for development or testing purposes), the procedure is essentially the same.

First, you will need to launch a new instance using the same EBS configuration. That is, the new instance should use the same EBS volume layout as the original instance. There is one difference, however: the data volume (/dev/sda2) should use the most recent snapshot created using the method described in the previous section. When the instance is launched, the snapshot will be used to create the sda2 device, ensuring that the PostgreSQL data is already in place when the instance is launched.

Once the instance is ready, you can log in and start PostgreSQL with this command:

/etc/init.d/postgresql-9.3 start

PostgreSQL must be configured using the same configuration files used for the original instance. The continuous archiving configuration directives will let PostgreSQL know how to restore the archived WAL files. This is controlled through the restore\_command configuration directive.

During the startup process, PostgreSQL will recognize that the files in the data directory were created as part of a base backup and will automatically restore the required WAL files by executing the restore command. This can take some time depending on how much data was changed since the base backup was taken, and how quickly the archived WAL files can be restored from remote storage.

PostgreSQL will begin accepting connections as soon as it has finished restoring the WAL files and the database is up-to-date. At this point, the state of the data should match the state it was in just before the original database was terminated. If the original database was uncleanly terminated (for example, the actual EC2 instance was terminated), some data might be missing if PostgreSQL did not have time to archive the final WAL file(s).

As mentioned, this approach is a tad more complex than simply dumping a file full of SQL statements representing your data. However, it does provide a lot more flexibility in restoring your data, and is the only valid option after your database grows to the point where dumping SQL statements is no longer viable.

# Off-Site Backups

As I explained earlier, you may want to move some backups outside the AWS ecosystem (EC2 and S3) altogether for those rare but not impossible times that AWS goes down, and the AWS API is unavailable. You can use another cloud provider to host your off-site backups, or provision a new server (entirely outside AWS) to act as a backup host.

Assuming that your data has been backed up to S3 using one of the other methods described in this chapter, you can use tools such as s3cmd or s3funnel to regularly pull this data from S3 and store it on another server.

###### WARNING

Remember, backups without a reliable and regularly tested restore procedure can be worse than useless, providing nothing but a false sense of security.

Test your restore procedures regularly!