Automation Anywhere Enterprise V11.1 Architecture & Implementation Guide

Table of Contents

Audience

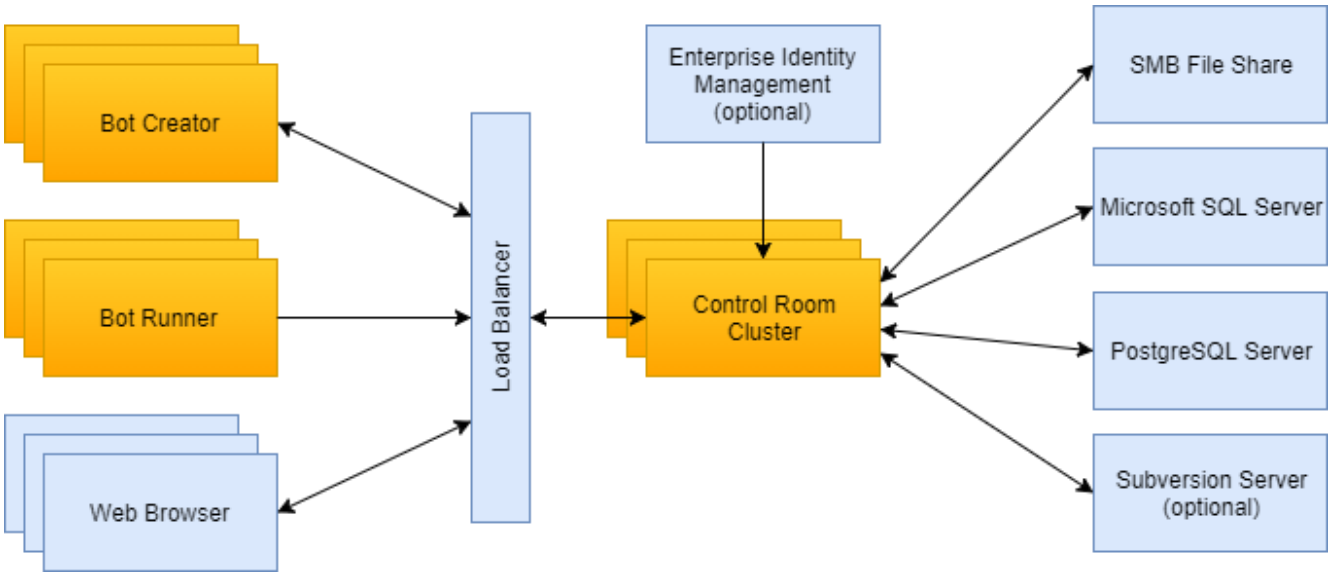
This document is primarily intended for IT Managers, Enterprise Architects and Technical Decision Makers to assist in implementing RPA using Automation Anywhere Enterprise Suite.

1 Glossary of Terms & Acronyms

NAME / ACRONYM	DEFINITION
Bot	Contains business process logic
Bot Runner (BR)	Software used to run a bot
Bot Creator (BC)	Software used to author bots
Control Room (CR)	Software used to manage, monitor and deploy bots
Subversion Version Control (SVN)	Software optionally used to manage and track bot versions
Database (DB)	Software used to store metadata and other system data
High Availability (HA)	Deployment architecture that maintains system availability within a single datacenter when a subset of the system fails.
Disaster Recovery (DR)	Deployment architecture that allows system availability to be restored via a backup datacenter due to unexpected events or natural disasters. Some downtime and loss of recently modified data is acceptable.
Active Directory (AD)	Directory service that provides user authentication, authorization in a typical enterprise
PostgreSQL	Database server software
REST API (REST)	Protocol used to expose business services to clients
Service Tier	Group of machines tasked with hosting services that implement business logic and serve static assets such as images and JavaScript.
Data Tier	Group of machines which are responsible for providing data storage services.

2 Architecture

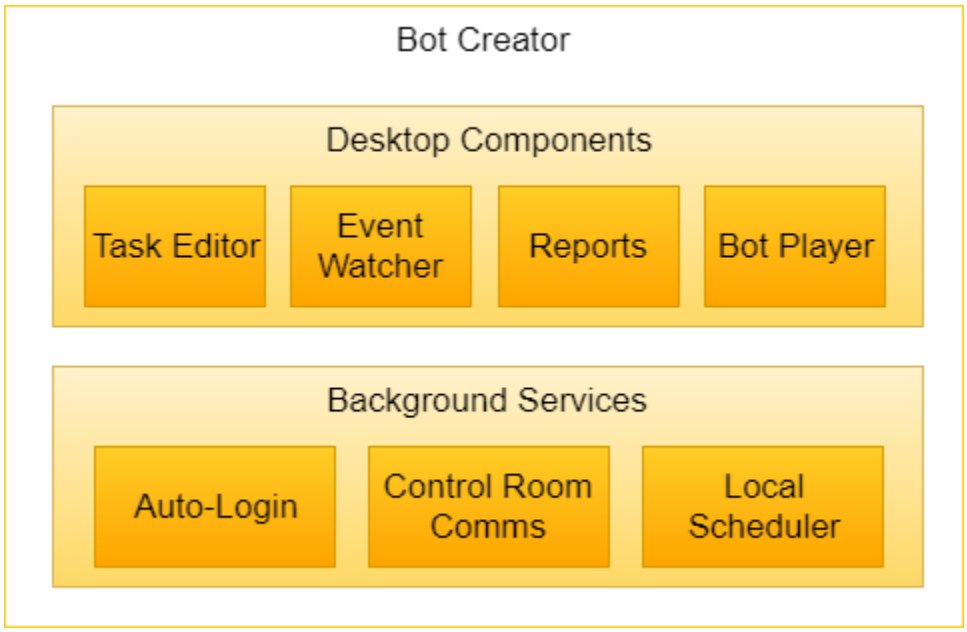
2.1 Overview



COMPONENT	FUNCTION
Control Room Cluster	A group of servers that are used to manage, monitor and deploy bots.
Bot Creator	Software used to author and upload bots.
Bot Runner	Software used to run bots that have been authored via Bot Creator. Bot Runners download bots from the Control Room.
Web Browser	Used by users and administrators to access the Control Room.
Enterprise Identity Management	The Control Room can optionally use an enterprise identity management system such as Active Directory, or a SAML identity provider, to authenticate users.
SMB File Share	An SMB file share is used by all members of the Control Room cluster to store or cache bots and configuration information. It must be accessible by every Control Room server.
Microsoft SQL Server	Primary database used to store all critical data related to Control Room operation including: bot, access control and analytics information.
PostgreSQL Server	Database used to store metadata related to analytics dashboards.
Subversion Server	When the optional bot versioning feature is enabled, bots are stored and versioned in an external Subversion server.

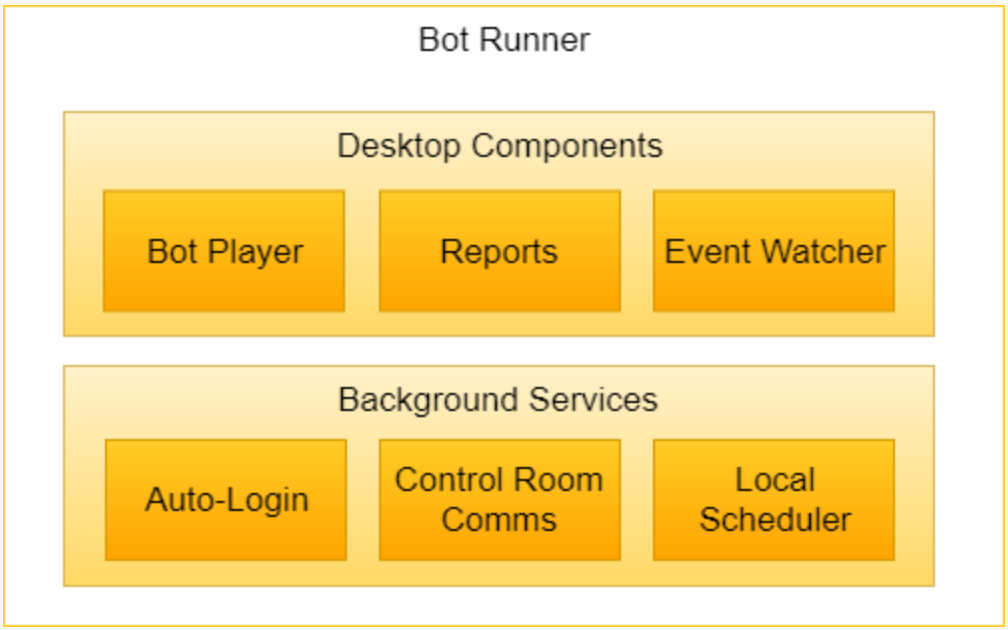
Load Balancer	Network Load Balancer that distributes HTTP(S) requests between Control Room cluster members

2.2 Bot Creator



A “Bot” is a self-contained task designed to be run with little to no human intervention. Bot Creator is our proprietary development client that can be used to author bots.

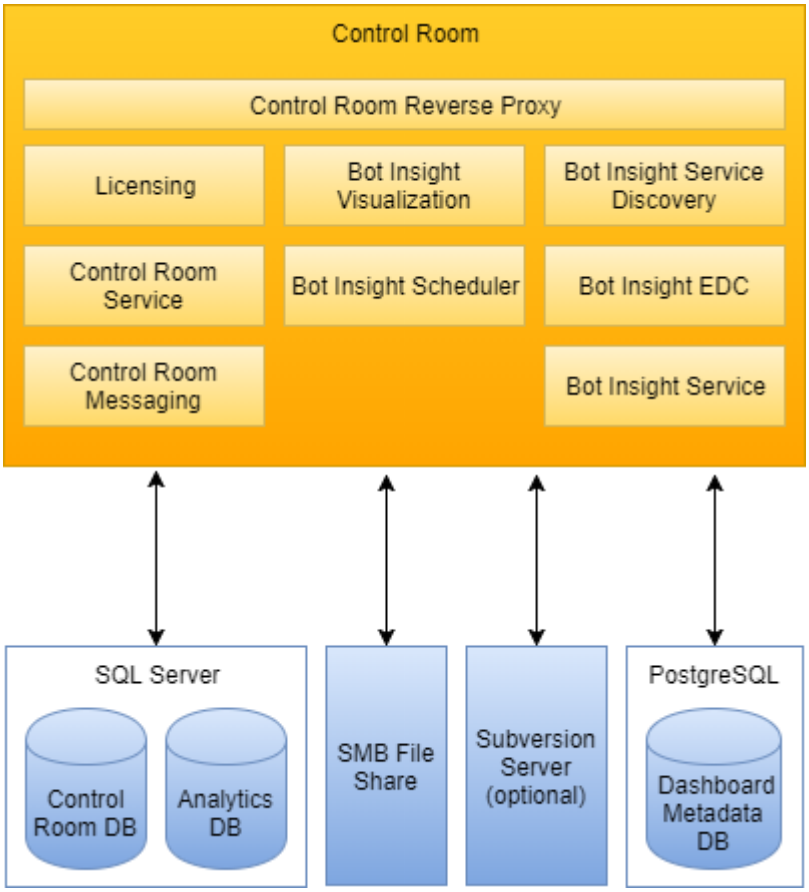
2.3 Bot Runner



Bot Runner is the machine that runs the bot. Typically, you use a Bot Creator to create a bot that many Bot Runners will run at scale.

2.4 Control Room

Control Room is a centralized management point for all bots being used throughout your environment. Control Room manages, schedules, executes, configures various aspects of bots and Bot Runners using a collection of specialized web services. A reverse proxy is responsible for listening for remote connection requests and forwarding those requests to the correct specialized service.



3 Deployment

3.1 Planning

AAE suite offers a variety of deployment options to meet various levels of enterprise cost/price performance needs. It is important to identify key requirements before you pick a deployment model.

3.2 Deployment Models

At a high-level, there are 3 ways to install Automation Anywhere Enterprise depending on your business continuity requirements.

3.2.1 Single-Node Deployment

A single-node deployment is used for some proof-of-concept deployments. It is not recommended for production workloads.

A single Control Room installation is deployed without the need of a load balancer.

Pros:

- Easy to setup and configure
- Single server required

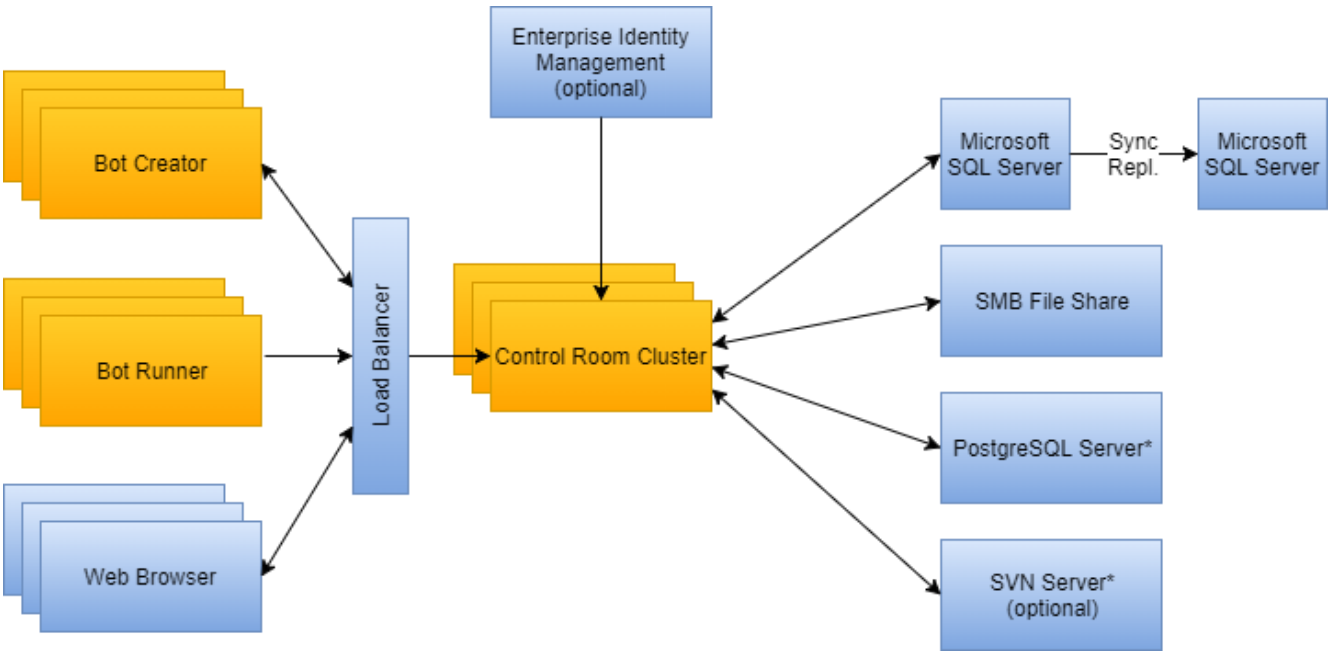
Cons:

- No disaster recovery
- No high availability
- Susceptible to hardware failures

Use Case:

- Proof of concept
- Single-user use scenarios

3.2.2 High Availability Deployment Model



**See Graceful Degradation*

The High Availability Deployment Model (HA) provides tolerance for failures of individual Control Room cluster servers and database servers. A load balancer must be configured to front all HTTP(S) requests for the Control Room. Configuring your load balancer to use a round-robin algorithm is recommended. Synchronous

replication should be configured between the master SQL Server and slave(s) so that consistency is maintained after a database node failure.

The synchronous replication required for HA can be achieved by utilizing either:

- Setting the backup replica to Synchronous-Commit mode of SQL Server Always On availability groups
- SQL Server Database Mirroring

Pros:

- Maintains availability when failures occur within a single datacenter

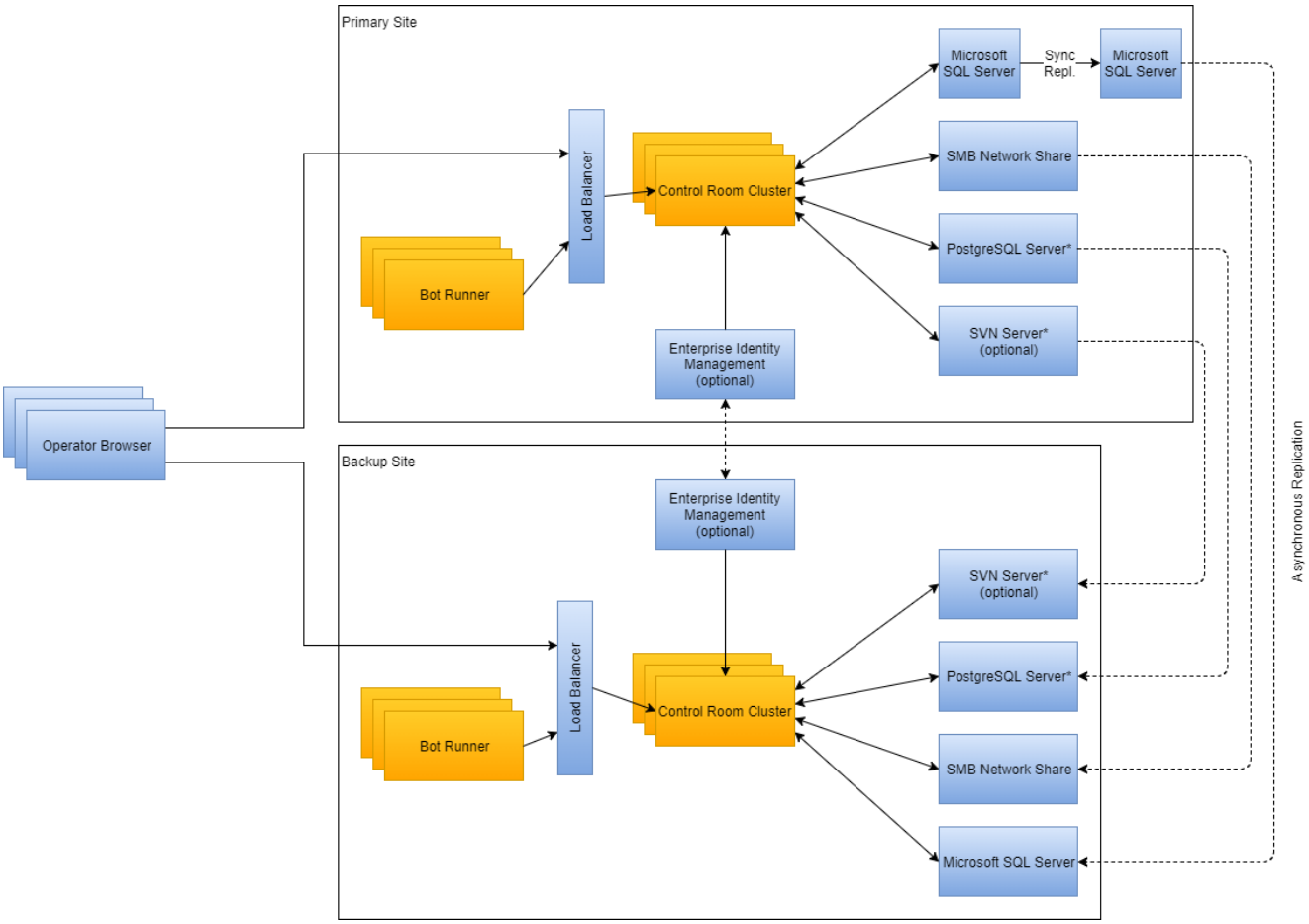
Cons:

- No protection against datacenter outage

Use Case:

- Small to medium-size businesses that do not require multi-site disaster recovery

3.2.3 High Availability with Disaster Recovery Deployment Model



**See Graceful Degradation*

To support disaster recovery (DR) a second Control Room cluster should be deployed to an additional datacenter. Asynchronous, rather than synchronous replication should be configured between sites for all supporting services so that off-site replication does not impact performance of the primary site. When a failover to a backup site occurs very recent changes made on the primary site may be lost.

Failing over to the DR site should normally be performed as a manual operation.

Pro:

- Provides business continuity when faced with datacenter outage or loss

Con:

- Increased operational burden

3.2.3.1 Replication Details

The database replication configuration for disaster recovery is an extension of the high availability configuration. This configuration requires the use of Always On availability groups.

The replica that is on-site with the primary replica should, like an HA-only deployment, be configured in Synchronous-Commit mode.

The replica at the DR location should be configured in Asynchronous-Commit mode so that the latency and reliability of the inter-datacenter does not impact the performance and availability of the primary site.

The DR replica should be configured to not offer any database services until failover is triggered. The Control Room nodes at the DR location will automatically connect and start servicing connections once the database is available.

3.2.3.2 Failure Mode

With asynchronous replication there is the possibility that a transaction that occurs on the primary site may not reach the DR replica before the failure occurs. Note that this is true not just for the Automation Anywhere solution but also for the applications that are being automated.

If Workload Management is processing items when a failure occurs some work items that were awaiting delivery to a device will be placed into an error state. This is so that there is no danger that work items will be processed twice. Work items can be manually reviewed and marked as ready to be processed or complete as appropriate.

3.2.4 Graceful Degradation

Certain dependencies of the Automation Anywhere system do not require full high availability (HA) to continue to successfully deploy and run bots.

3.2.4.1 PostgreSQL

PostgreSQL is used to store dashboard metadata. If PostgreSQL fails, no dashboard graphs will be available until PostgreSQL service is restored.

3.2.4.2 Subversion (SVN) Server

Subversion is optionally used to store previous versions of a bot. If Subversion is unavailable it will not be possible to change what the current production version of a bot is. However, the current production version of all bots will still be available for deployment.

3.2.5 Minimum Hardware Specifications

Up-to 1000 simultaneous bot deployment and executions.

	BOT RUNNER	BOT CREATOR	CONTROL ROOM	SQL SERVER	POSTGRESQL SERVER
Operating System	Microsoft Windows 7 SP1	Microsoft Windows 7 SP1	Windows Server 2012 R2 or later	Windows Server 2008 SP2 or later	Windows Server 2008 R2 or later / Red Hat Enterprise Linux / Ubuntu LTS
Processor	Intel Core i5 2.6 GHz	Intel Core i5 2.6 GHz	8 core Intel Xeon Processor	4 core Intel Xeon Processor	2 core Intel Xeon Processor
RAM	8 GB	8 GB	16GB	8GB	4GB
Storage	32 GB	32 GB	500GB	500GB	10GB
Other	.NET Framework 4.6 (Windows 8.1 and Window Server 2012 R2: 4.6.1)	.NET Framework 4.6 (Windows 8.1 and Window Server 2012 R2: 4.6.1)	.NET Framework 4.6 (Windows 8.1 and Window Server 2012 R2: 4.6.1)	SQL Server 2012 or later	PostgreSQL 9.5

3.3 Install

Please request the installer and appropriate license keys from your account manager. See product documentation on how to install.

3.4 Configure

3.4.1 Firewall Rules

3.4.1.1 Bot Creator and Runner

PROTOCOL	PORT(S)	RULE
TCP	80/443	Out – Control Room Load Balancer

3.4.1.2 Control Room

PROTOCOL	INCOMING PORT(S)	USAGE	CLIENT(S)
TCP	80	HTTP	Web browsers, Bot Runners, Bot Creators
TCP	443	HTTPS and Web Socket	Web browsers, Bot Runners, Bot Creators

TCP	5672	Cluster Communication	Control Room servers
TCP	47500 – 47600	Cluster Communication	Control Room servers
TCP	47100 – 47200	Cluster Communication	Control Room servers

WARNING: It is critical that communication between the Control Room servers is sufficiently protected as it contains security sensitive information that is not encrypted. All network hosts, apart from Control Room servers, should be blocked from accessing the cluster communication ports listed above.

3.4.2 Bot Runner/Creator

1. **Schedules (Bot Creator Only)**
    a. Config File Name : Settings.xml
    b. Config File Location : C:\Users\Public\Documents\Automation Schedules
    c. Log File Name: SchedulerService.LogFile.txt
    d. Log File Location : C:\Users\Public\Documents\Automation Schedules\LogFiles

2. **AutoLogin**
    a. Config File Name : Automation.Autologin.Settings.xml
    b. Config File Location : C:\Users\Public\Documents\Automation Autologin

3. **Client**
    a. Config File Name : AA.Settings.xml
    b. Config File Location : C:\Users**aaadmin**\Documents\Automation Anywhere Files

3.4.3 Database

The Database connection string is configured in <Install Location>\config\boot.db.properties

3.4.4 Version Control

Version control (SVN) location can be configured via the Control Room UI.

4 Operations

4.1 Data Retention

At some point the performance of searching and sorting historical data on the Control Room will be impacted by the amount of historical data stored. The table below shows the SQL tables that may have data deleted from them without impacting the availability of the Control Room. *Note that modifying or removing any other records than those listed here may leave your Control Room cluster in an unrecoverable state.*

DATABASE	TABLE	TIMESTAMP COLUMN
Control Room	ACTIVITY	created_on
Control Room	JOBEXECUTIONS	updated_on

To purge data use the DELETE query, for example:

```
delete from [ACTIVITY] where created_on < DATEADD(DAY, -30, GETDATE());
```

To delete entries from the activity table that are older than 30 days.

4.2 Logging

Logging data is generated throughout our product. For logging to be more useful, we recommend that you consolidate you logs into one central machine / area.

4.2.1 Windows Built-In Logging Strategy

Logging data needs to be gathered in one central location for better and more efficient consumption.

Windows has a built-in “centralized logging”. To setup a single machine that pulls all logging data from other machines into a single using subscriptions.

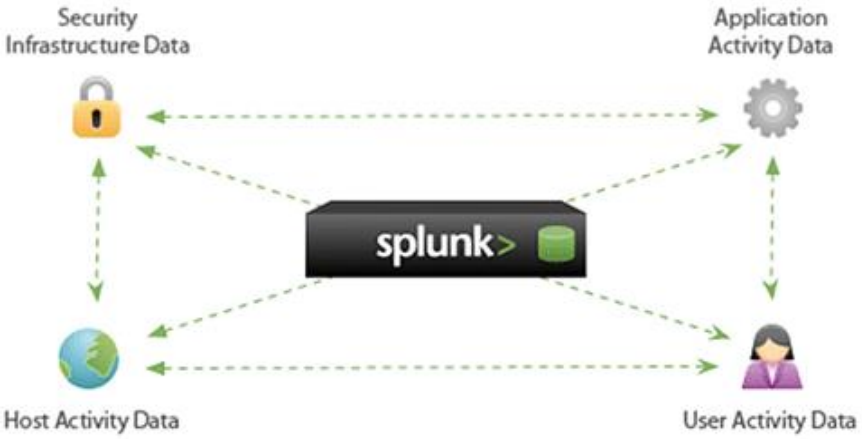
1. Open Event Viewer on the Central Logging machine
2. Subscribe to Logging Events from other computers

If a more elaborate solution is required, we recommend using a product like Splunk to gather and store log in a central location for collection and analysis.

4.2.2 Splunk Strategy

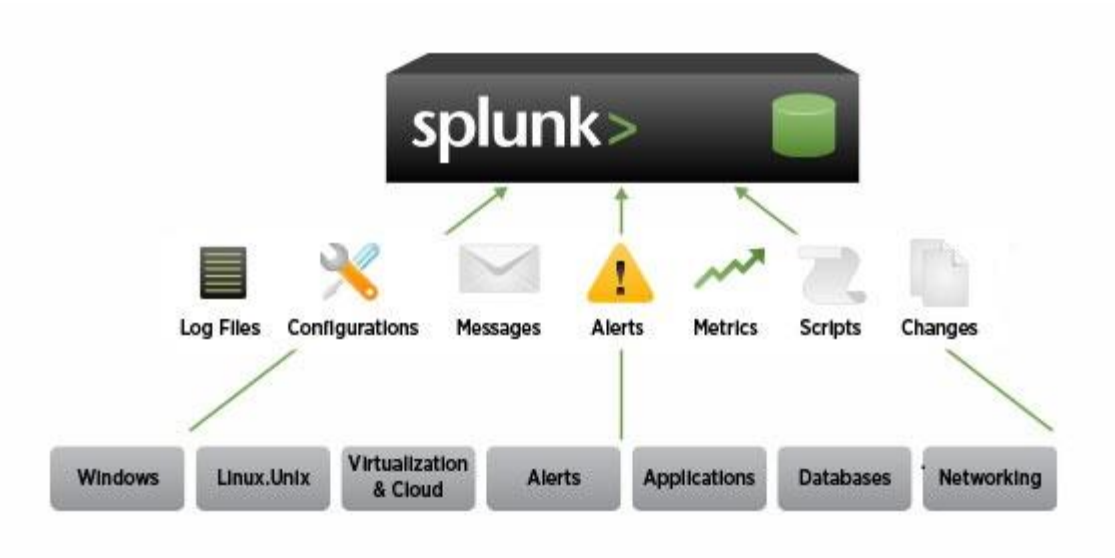
Logging is only incredibly useful when we have the ability to see a “birds-eye-view” of all logging events across systems and apps in a single area / tool as it provides a holistic picture to your entire environment. Hence, we recommend you use a tool like Splunk to aggregate logs from various sources in one central location.

Splunk can collect various types of logging data:



How does Splunk fit into Automation Anywhere Enterprise?

Splunk fits nicely into existing logging infrastructure of AAE as well as your network / system environment giving you a single holistic view of your system. Splunk has an enterprise-grade "Universal Forwarders" that can be installed on most Operating Systems and Networking environment. This light-weight software agent (Universal Forwarder) monitors logs that are already being generated and forwards it to Splunk Indexing Server, all in real-time. This makes deployment of Splunk incredibly easy and scalable. More importantly, it allows you to "see" a birds-eye-view as well as drill-down options of all of your logging data.



4.2.3 Log Event Types

- Infrastructure Logging
    - Network, Router, Switch, Firewall, Gateways, etc.
- Systems Logging
    - Windows Event Viewer, Web Server Logs, Machine Logs

- Application Logging
    - Control Room
    - Bot Runner
    - Bot Creator (Dev Client)
    - Bot Farm
    - Credential Vault
    - Analytics

4.2.4 Log Retention

Log Retention is usually determined by a company's existing log retention policies. In general, we recommend that log files be archived to "cold storage" after 5 years since the file was generated. Until then, we recommend that they are archived in a "warm storage". And finally, it is recommended that you keep logs files ***in*** your servers for at least a month (i.e. "hot storage")

Cold Storage: Long term storage such as the use of magnetic tape that survives the test of time.
Warm Storage: Corporate-wide backups that are generally available for 1 year.
Hot Storage: The log files are on the server that generated it.

4.2.5 Log Rotation

Log rotation is highly recommended to ensure logs are kept a manageable file size in the file system. Therefore, we recommend that you rotate the log every 24 hours (i.e. log file per 24 hours). Depending on the amount of log data being generated, we recommend that the log file be rotated accordingly. There two options:

Log Rotation by Time: Create a single log file per 24 hours
Log Rotation by Size: Rotate the log file (create new log file) depending on size the file
Log Rotation by Both: Some combination of both to limit size & time per file

4.3 Monitoring & Alerting

For monitoring and alerting, you can use built-in Windows functionality.

4.3.1 Windows Built-in Performance Monitor Solution:

You can use Windows' performance counters to capture vital information such as CPU load, memory, etc. onto We recommend deploying Nagios in your environment.

1. Open Performance Monitor
2. Create new data collector set, choose data collector such as CPU, memory, Disk IO, etc.
3. In the Alert Box, specify the email address and SMS so that you can be notified up on a failure.

4.3.2 Nagios

Nagios is a powerful enterprise-grade that provides you with an instant awareness of your Automation Anywhere IT infrastructure. It allows you to detect and repair problems and mitigate future issues before they affect your end-users.

There are 2 parts to consider when it comes to monitoring as a strategy:

4.3.3 Monitoring

- **Machine Vitals** (machine, VM, devices, etc.) - CPU Load, Memory Pressure, Disk Space & IO, Processes, and other system metrics. Machines include control room, bot farm, bot runner, etc.
- **Network** – protocol, uptime, overload, throughput, ping, latency, DNS, etc.
- **Application** – bot running, log peek, scheduling service, database service, web server, LB, log truncation, etc.

4.3.4 Alerting

Nagios can send alerts when critical infrastructure components fail. It can also be configured to notify recovery. There are 3 alerting methods:

- **Email** - Sends an email to administrator or IT team upon critical / important events
- **SMS** – Sends a text message upon critical events
- **Custom Script** – Alerting logic can highly be customized based on several properties. For example, an escalation can be setup based on severity, time, etc.

4.4 System maintenance

4.4.1 Database Maintenance Plan

We strongly recommend the following database backup strategy (at a minimum):

- Weekly – Full Database Backup
- Every 3 Days – Differential Backups
- Daily – Incremental Backups Every 24 Hours
- Hourly - Transaction Log Backup Every Hour

5 Development Design, Guidelines & Coding Standards

5.1 Purpose

The purpose of this document is to be an advanced guide to developing automations and providing guidelines and standards for automation development. Why use this document? Because not every person developing an automation:

- is aware that a task (or any code set) is generally read 10 times more than it is changed
- is aware of some of the pitfalls of certain constructions in automation development
- is aware of the impact of using (or neglecting to use) particular solutions on aspects like maintainability and performance
- knows that not every person developing an automation is capable of understanding an automation as well as the original developer

Following standards creates clean, easier to read, easier to maintain, stable automations.

5.2 The Don't Repeat Yourself Principle (DRY)

The Don't Repeat Yourself Principle is a principle commonly found in software engineering. The aim of this principle is to reduce the repetition of information. In the context of automation, it refers to creating a set of commands or bits of logic that are repeated throughout a single task.

For example, suppose an automation has the need print a notepad document as a PDF file. The task might look like the following:

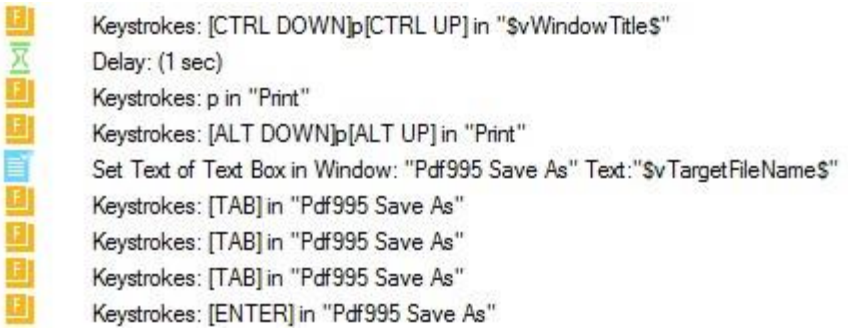


FIGURE 1

In this particular automation, there is a need to print a file as a PDF document three times. The temptation would be to simply copy and paste these nine lines into the three places where they are needed. It's fast and simple, and helps resolve the immediate goal – getting this automation finished quickly. However, cutting and pasting these lines 3 times is short-sighted.

Consider for example on the development machine the PDF print driver is called "Pdf995". When the automation is moved to a production machine it is discovered that the PDF print driver on that

machine is not called “Pdf995”, but rather “CutePDF”. Now the entire automation task must be opened and edited. In addition, there are three places to find this code, and because this is copied and pasted the automation is now 27 lines longer than it would have been. Long tasks are more difficult to edit and they take longer to edit. Not only that, a task that was tested and verified to be bug-free and production ready has now been edited. Since it has been edited, it is no longer production ready because it has been changed, and should now be re-tested.

By saving an extra minute of time cutting and pasting these nine lines in three different places, 30 minutes to an hour of maintenance time have now been added because the task must be edited and it’s longer than it normally would have been. If a new developer has been tasked with making the changes and they are unfamiliar with the automation, they must now do more analysis to determine all of the places that need the change. If the task is short that’s not so bad, but if it’s long it’s much more time consuming.

Of course in this example, using variables could alleviate some of these issues. But now yet another variable must be added and maintained, making the automation more cluttered. And there’s always the possibility that the issue can’t be resolved by simply using a variable.

So what’s the solution? A sub-task, that is called by the task needing this service performed. By placing these commands into a task that is called by a parent task, we realize a number of benefits:

1) the main task is now 27 lines shorter
2) when it moves to production, only 9 lines have to be located, analyzed and edited and they live in a single task, making the automation much easier to maintain
3) these lines can be called by any number of tasks, even in other automations

Sub-tasks can be referred to as a “helper task” or “utility task”, since their only purpose is to help the calling task. Figure 2 is an example of what the helper task for the above example might look like:

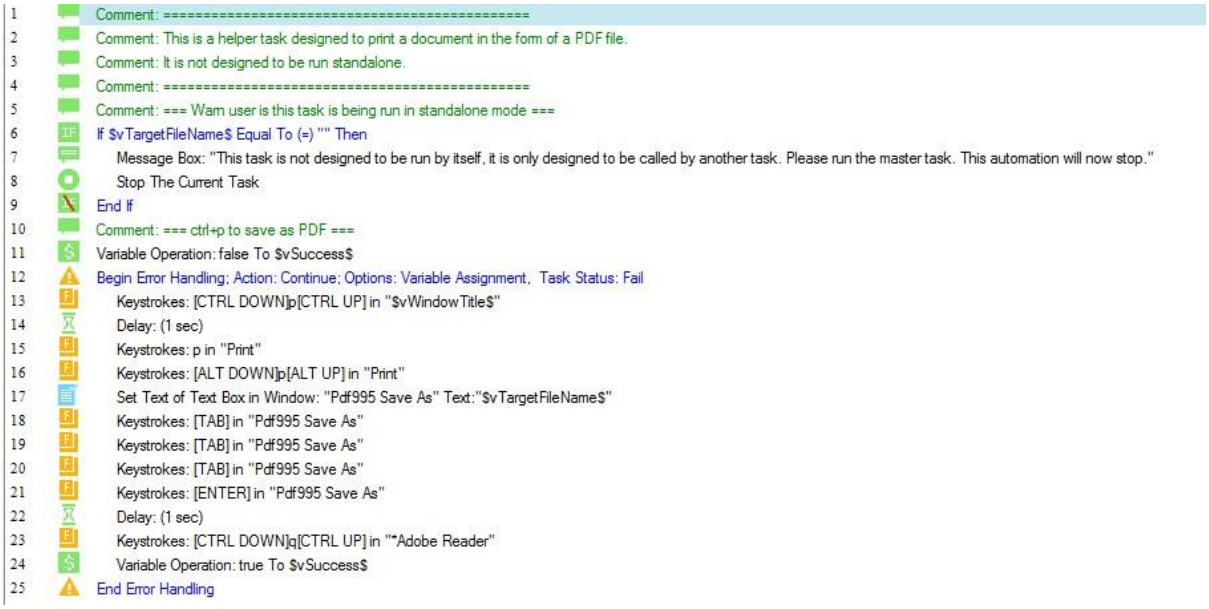


FIGURE 2

If any changes are required to this specific set of commands, only this helper task will need to be edited, and only this helper task will need to be retested.

Tip: Remember that Excel sessions, CSV/text file sessions and browser sessions (web recorder) cannot be shared across tasks. So sub-tasks must be included in such a way as they do not break these sessions.

5.3 The Rule of Three

Keeping in mind the Don't Repeat Yourself Principle, The Rule of Three is a great way to prevent introducing repeating code or commands. Duplication of a set of commands in automation is a bad practice because it makes the automation harder to maintain. An AA task should never be more than 500 lines in general, and should preferably be only a few hundred lines. When a rule is encoded in a replicated set of tasks changes, whoever maintains the code will have to change it in all places correctly. This process is error-prone and often leads to problems. If the set of commands or rules exist in only one place, then they can be easily changed there. Additionally, if factored out properly, sub-tasks can be made to be reusable, making them even more productive.

This rule is typically only applied to a small number of commands, or even single commands. For example, if a task calls a sub-task, and then calls it again when it fails, it is acceptable to have two call sites; however, if it is to be called five times before giving up, all of the calls should be replaced with loop containing a single call.

Tip: Sub-tasks should be small and focused (have only a single responsibility or only a few responsibilities).

5.4 The Single Responsibility Principle

In object-oriented programming, the single responsibility principle states that every class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility. This principle directly applies to automation development as well.

Now imagine a single automation task that is 2000 lines long. This task is far too long and should be split into several sub-tasks. The automation developer decides to put several of the repeated tasks into another sub-task. Consider for example he or she picks three repeated sections and puts them in a subtask. This new helper task now handles printing a PDF, but also handles saving a file to a specific folder, and in addition to that it also handles moving a file from one folder to another. The developer manages this which part is called by passing an action variable.

The above example would break the Single Responsibility Principle. The developer has reduced the number of lines in the master task, which is good, but has now a sub-task that is far too big. Additionally, if any one of those responsibilities (printing a PDF, moving a file or saving a file) need to be modified, the entire helper task must be modified. This creates the possibility for introducing a bug in task that would not have otherwise been affected.

The proper approach would be to create three sub-tasks, each having their own responsibility – One for print to a PDF, another for moving the files, and another for saving files.

5.5 Decoupling and Loose Coupling

When possible, sub-tasks should not have a dependency on the calling task. In automation this is often unavoidable. However, developing with this in mind can improve the overall architecture and maintainability of an automation.

Using the login sub-task as an example, if the login task can only be called by one single master task, then it is tightly coupled to that master task. If the login sub-task is designed in such a way that the URL of the page it uses has to be set by the calling task, then it cannot run by itself. It cannot be unit-tested alone, and other tasks cannot call it without knowing the URL of the login page before calling it. If the calling task must provide the login page URL to the login sub-task, then all tasks that use the login sub-task are more tightly coupled to that login sub-task. And if the URL changes, more than one task must be changed.

However, if the login sub-task contains all of the information it needs to login to the web application, including the URL, then it is a truly stand-alone sub-task. It can be unit-tested, and it can be called by any other task without the need to be provided the URL. It is then “decoupled” from other tasks and is much more maintainable.

5.6 Avoid Bi-Directional Dependencies

As previously described, a sub-task that depends on the calling task for information causes problems, a calling task depending on a sub-task for information can also cause problems. The problem here is that if the sub-task is changed, then all calling tasks may also need a change, since they have dependencies on that sub-task. If a sub-task cannot be changed without calling tasked being changed, it is dependent and not truly decoupled. If a calling task cannot be changed without all sub-tasks being changed, they are not truly decoupled, and have bi-directional dependencies. This not only creates a maintenance nightmare, but it makes unit-testing nearly impossible. Sub-tasks should be as standalone and reusable as possible.

5.7 Test Driven Design

If automations are developed using the above principles, the result will be many smaller tasks, many being reusable. These smaller tasks can then be tested alone, in a unit-test fashion.

For example, take an automation that always starts with a login to a web-based application. The login step should be in a separate sub-task. Using this approach, if anything changes with the login UI, only the login task has to be modified. If several other tasks are using the login step, none of those tasks will need to be modified or re-tested if the login UI changes. Only the single login task will need to be modified, and as such, only that login task will need to be tested if something changes, therefore testing that single unit. This is far more efficient than modifying a single large task, or set of tasks if something changes in a smaller unit.

For example, take an automation that always starts with a login to a web-based application. The login step is only performed once, at the beginning of the automation, so it appears to not make sense to split it out into a separate task. In reality, it does actually make sense to split the login step into its own task. The reason is test driven design and greater maintainability.

With the login step in its own sub-task, if there is a change to the login screen, only that task needs to be modified. If the task is created using the Single Responsibility Principle, it only performs on function – logging in to the application. Additionally, if the task is designed to be loosely coupled, it doesn't need to know anything about the master task calling it. The URL, user name and password can be put into the automation for testing purposes, and then that task can but tested independently.

This approach may not be possible in all cases, but when possible this can be make an automation much easier to maintain and deploy.

5.8 Testing

After creating an automation task and beginning preparation for a deployment either to a production environment or delivery to a customer, the automation should be fully tested. If it does not pass testing, either errors should be corrected or error handling should be put into place so that unexpected events do not cause the automation to fail.

5.9 Error Handling

Automating applications, especially browser-based applications, can be a moving target at times. If a web page changes for example, it will often break the automation. Or most commonly, if a page doesn't appear when the automation expects, it can cause an error. The key to a successful automation is predicting and handling unexpected events (e.g. a *Save As* dialog not appearing within a specific time frame or a file not found message).

Care should be taken when automating components that rely on network connectivity. If your automation works with a web browser it should be able to gracefully handle situations where the browser cannot load the target page due to an outage. The Error Handling command can be used to not only log errors, but to also provide a means to recover from an error.

For example, consider an automation that downloads a file from a web site. After clicking on the download link, the automation waits 15 seconds for the download prompt to appear at the bottom of IE.
the automation uses a "wait for window to exist" command to determine when the Save As dialog appears.

5.10 Avoid Too Many Sub-tasks

While all of the above principles are an excellent way to design an automation, it is possible however to "overdo it". If an automation has 30 helper tasks for example, that's going to be a bit difficult to maintain as well, and is an indication of a poorly thought-out architecture. The number of sub-tasks, or helper tasks should be a manageable amount that do not cause confusion.

An automation that has 30 helper tasks, or would be thousands and thousands of lines without the use of helper tasks, probably indicates a business process that is too large for one automation. Such a process should be broken down into pieces, and each of those separate pieces encapsulated in their own automations.

5.11 Giant Business Processes Lead to Giant Automations

The key to the successful automation of a business process is a well-defined, well-thought-out strategy. If a business process is so large for example, that the resulting automation will either be one file that is 10,000 lines long, that is an example of a task that is too long. If the automation is refactored into several sub-tasks, with utility and helper automations, but now consists of 30 helper tasks, it is still an example of an automation that is not well architected.

If a business process is so large that it requires more than 8 or 10 sub-tasks, or any one task contains thousands of lines, then the automation approach to the process should be reconsidered. Some things to take into account:

1) What parts of the business process can be split into their own separate automations?

2) Can the business process itself be reduced in size? Are there any redundant or unnecessary steps?

5.12 Commenting

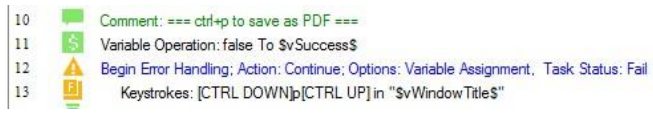
Most automations require changes after they are placed into production. Sometimes those changes can be frequent, depending on the type and scope of the automation. The difference between a change being a relatively straight-forward task and a complete nightmare is determined by two things: how cleanly the automation was architected, and how well it is documented and commented. Good commenting can mean a difference of hours during a maintenance cycle.

All comments should be written in the same language, be grammatically correct, and contain appropriate punctuation.

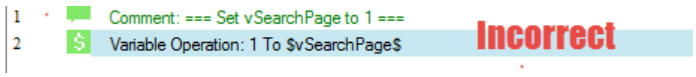
General rules:

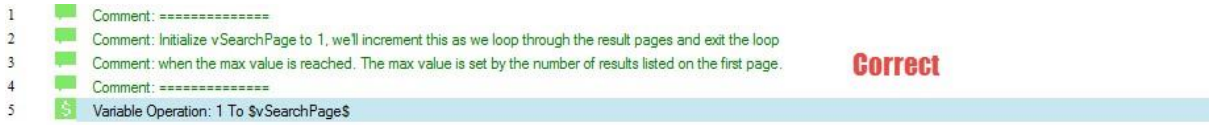
- Box important sections with repeating slashes, asterisks or equal signs:

- Use one-line comments to explain assumptions, known issues and logic insights, or mark automation segments:



- Make comments meaningful:





- Only use comments for bad task lines to say “fix this” – otherwise remove, or rewrite that part of the task!
- Include comments using Task-List keyword flags to allow comment-filtering. Example:
  // TODO: Place database command here
  // UNDONE: Removed keystroke command due to errors here

- Never leave disabled task lines in the final production version. Always delete disabled task lines.
- Try to focus comments on the why and what of a command block and not the how. Try to help the reader understand why you chose a certain solution or approach and what you are trying to achieve. If applicable, also mention that you chose an alternative solution because you ran into a problem with the obvious solution.

5.13 Naming Conventions

Use bumpyCasing for variables and CamelCasing for task names:

- CamelCase is the practice of writing compound words or phrases such that each word or abbreviation begins with a capital letter (e.g. PrintUtility)
- bumpyCase is the same, but always begins with a lower letter (e.g. backgroundColor).

Do not use underscores. Underscores waste space and do not provide any value. Readability can be achieved by using Bumpy Casing and Camel Casing.

Always use lower case Boolean values “true” and “false”. Never deviate, stick to this method of defining a boolean state. The same should also be applied to flags. Always use “true” or “false” for Boolean variables, never a 0 or 1 or anything else (Figure 3).

FIGURE 3

Don’t include numbers in variable names.

Don’t prefix fields. For example, don’t use g_ or s_ or just _. It’s okay to use the letter v in order to make finding the variable simpler.

Definitely avoid single character variable names. Never use “i” or “x” for example. A person should always be able to look at a variable name and gain some clue about what it is for.

Name flags with Is, Has, Can, Allows or Supports, like isAvailable, isNotAvailable, hasBeenUpdated. Name scripts with a noun, noun phrase or adjective like Utility or Helper for example FileSaveHelper.atmx. Also use verb-object pairs when naming scripts like GetMostRecentVersion. Name variables with a descriptive name like employeeFirstName or socialSecurityNumber.

5.14 Logging

One phrase that will be repeated throughout this section will be this: Logs should be easy to read and easy to parse. There are two receivers for log files, humans and machines. When humans are the receiver, they’ll may be a developer looking for information in order to debug, analyse performance, or look for errors. They may also be an analyst, looking for audit information or performance information. In either case, logs should be easy to look at and understand, and they should be easy to parse with a tool or import into Excel. What follows is a set of standards to ensure logging is properly executed.

5.14.1 Types of Logs and Messages

There are about five different types of messages that should be logged. Depending on the type, they may go into separate log files. For example, informational messages should go into the primary process log, where as an error message would into both the process log and an error log. Debugging information should go into a debug log.

Types of Logs

Process/Informational log – The process log is meant to be an informational log. It can be used for monitoring normal operation of a task, but more importantly, it can be used for auditing. Use the

process log for an audit trail can be an excellent method for determining if a business process was completed properly. For example, was an order placed, or a ticket completed without error.

Error log – The error log is for detailed error messages. When an error occurs in a task, notification that an error occurred should go into the process log. Detailed information about the error should go into the error log.

Debug log - Debugging information should go into its own log file, and should be turned off when in production mode. An isProductionMode variable should be used to turn these statements off when the automation is moved to production.

Performance log - Performance logging can either go into the process/informational log or it can go into the performance log. In some cases it may be desirable to have it in its own log file.

Types of Messages

ERROR – something terribly wrong had happened, that must be investigated immediately. The task cannot perform its function properly here. For example: database unavailable, mission critical use case cannot be continued, file is busy and cannot be opened.

WARN – the task might be continued, but take extra caution. For example: “Task is running in development mode”. The task can continue to operate, but the message should always be justified and examined.

INFO – Important business process has finished. In ideal world, and administrator or user should be able to understand INFO messages and quickly find out what the application is doing. For example if an application is all about booking airplane tickets, there should be only one INFO statement per each ticket saying “[Who] booked ticket from [Where] to [Where]”. Other definition of INFO message: each action that changes the state of the application significantly (database update, external system request).

DEBUG – Developers stuff. Any information that is helpful in debugging an automation, and should not go into the process log. An isProductionMode variable should be used to turn these statements off when the automation is moved to production.

PERFORMANCE – Performance logging can either go into the process/informational log or it can go into the performance log, if a separate performance log has been created. Performance should track how long it takes to perform specific steps, but too much granularity should be avoided. In most cases, performance logging should be limited to an overall business process. For example, how long it took to complete an order, or how long it took to process an invoice.

5.14.2 Know What You Are logging

Every time you issue a logging statement, take a moment and have a look at what exactly will land in your log file. Read your logs afterwards and spot malformed sentences.

5.14.3 Know Who is Reading Your Log

Logs will be imported by the customer. Logs should be easy to read and easy to parse. Users, customers and automation developers will be looking at logs to:

- determine if a process completed successfully
- if a process didn't complete, they'll be looking for information as to why
- determine if the automation is performing as expected
- interactively tailing the logs
- parsing the logs with a tool, or importing the logs into Excel in order to gather and analyse metrics
- importing the logs into a database

5.14.4 Be Concise and Descriptive

Each logging statement should contain both data and description. Consider the following examples:

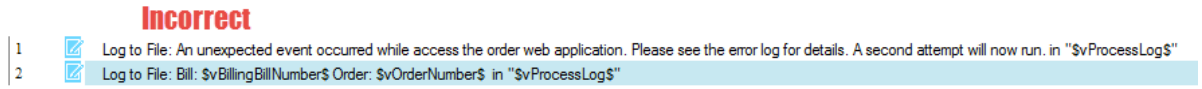


FIGURE 4

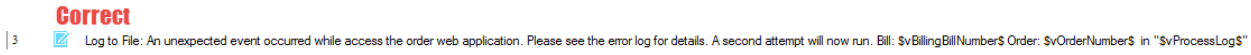


FIGURE 5

In Figure 4, two lines are used to communicate one set of information. All of the information should be on one line, and in one pass to the log command. Was it so hard to include thee actual message type, message id, etc. in the warning string? I know something went wrong, but what? What was the context? Be detailed, don't make the reader wonder what is going on.

5.14.5 Avoid the "Magic Log"

Another anti-pattern is the "magic-log". If a log file is long, or has grown to a large size, it may be tempting to insert characters or strings that make things easier to find certain lines. They're meaningful to the developer, but completely meaningless to anyone else. This is why they're called "magic". An example would be "Message with XYZ id received".

If this approach is used, you'll end up with a process log file that looks like a random sequence of characters. Instead, a log file should be readable, clean and descriptive. Don't use magic numbers, log values, numbers, ids and include their context. Show the data being processed and show its meaning. Show what the automation is actually doing. Good logs can serve as a great documentation of the automation itself.

And of course never log passwords or any personal information!

5.14.6 Formatting

Use the log to file feature built into Automation Anywhere. Use the built-in time stamp in the log to file command, don't create your own method and format for time stamping, even for Excel. This is not the industry standard, and you're trying to solve a problem that doesn't even exist. It's up to the receiver of the log to manage the time stamps. If the customer or receiver needs a different timestamp, then it makes sense to modify it, but not before.

A good format to use is timestamp, logging level, machine name, name of the task, and the message. All of these values should be tab delimited, for easy importing or parsing.

5.14.7 Avoid Side Effects

Performance is the primary concern with logging. Normal logging commands themselves are not costly in terms of performance. However, consideration should be made for excessive logging (inside of a small loop with many iterations for example).

5.14.8 Log Sub-Task Variables

When passing variables back and forth to sub-tasks, adding debugging log statements is recommended. These should be used to see the values of variables when they enter the sub-task, and what they have been set to when being passed back (where relevant). An isProductionMode variable should be used to turn these statements off when the automation is moved to production.

5.14.9 Consider Size, Rotating and Dozing

Always consider how large a log file will become. By nature, log files generally become quite large in a short period of time. Additional code should be considered for checking the current date, and if the day has changed, creating a new log. This way a new log file is created for each day, and older log files can be compressed and archived (dozing).

5.14.10 Log Errors Properly

Logging errors is one of the most important roles of logging. If an error is logged from within an AA error handling block, the line number and description of the error coming from AA should be included in the log message.

5.14.11 Easy to Read, Easy to Parse

There are two groups of receivers particularly interested in your application logs: human beings (you might disagree, but programmers belong to this group as well) and computers (typically shell scripts written by system administrators). Logs should be suitable for both of these groups. Logs should be readable and easy to understand just like the code should.

On the other hand, if your application produces 500 MB of logs each hour, no man and no graphical text editor will ever manage to read them entirely. In this case, other tools will be uised. If it is possible, try to write logging messages in such a way, that they could be understood both by humans and computers,

e.g. avoid formatting of numbers, use patterns that can be easily recognized by regular expressions, etc.

5.14.12 Interfacing with External Systems

If an automation will be interfacing with other systems i.e. Metabots, APIs, REST or SOAP calls, be sure to log those calls, and if appropriate their responses.

5.15 VB Script

Automation Anywhere has the ability to call VB script. It is recommended to limit the use of VB script to situations where there are simply no other choices. The reasons are:

- VB script is difficult to maintain
- VB script is typically not understood by the customer
- VB script files are usually prohibited by most IT departments
- VB script files cannot easily be moved (blocked by email)

Another thing to remember is to **never** use AA to write VB script, or create a VB script file. Doing so is extremely difficult to maintain and is an anti-pattern at best. At worst, it demonstrates the ability to embed and deliver a malicious payload in an automation.

5.16 Configuration Files

Always separate your initial variable values from your task. This means do not set necessary variable values in the task. You will need to change these values when you run the task in different environments like UAT or PROD. Use a configuration file and read those variables into the task at start time. Make use of system path variables to load the configuration file so it can be located no matter where AA is installed on the system.