Малачевский А.А.

"🚀 PowerShell 7 для Linux-админов: Учебный курс"

Краткое содержание:
Учебный курс предназначен для системных администраторов Linux, DevOps-инженеров и всех, кто хочет использовать PowerShell 7 как современный кроссплатформенный инструмент для автоматизации. Программа охватывает установку, основы работы с файлами, системой, сетью, а также продвинутые темы: безопасность, взаимодействие с Docker/Kubernetes, интеграцию с bash и Ansible.

Структура курса:

- Модуль 0. Введение в PowerShell на Linux
- Модуль 1. Основы команд и работы с объектами
- Модуль 2. Работа с файлами, CSV, JSON, XML
- Модуль 3. Системные команды и мониторинг
- Модуль 4. Сетевые задачи и работа с API
- Модуль 5. Условные конструкции, циклы, обработка ошибок
- Модуль 6. Автоматизация через cron и логирование
- Модуль 7. Создание собственных функций и модулей
- Модуль 8. Безопасность скриптов и защита данных
- Модуль 9. Расширенные темы: SSH, Ansible, Docker, Kubernetes
- 📎 Приложение: Руководство по настройке SSH-сервера на Linux Mint 22

Формат материала:

- Теоретические блоки
- Примеры кода
- Практические задания
- Ссылки на официальную документацию и полезные ресурсы

Технические требования:

- Ubuntu 20.04+ / Debian 11+ / Linux Mint 20+
- PowerShell 7.2+
- Желательно: знание bash/cron/systemd

Цель проекта:
Дать Linux-администам практический инструмент автоматизации, повысить их квалификацию в использовании PowerShell на Linux-серверах.

🚀 PowerShell 7 для Linux-администраторов: Учебный курс

🧰 Целевая аудитория:

- Linux-администраторы, DevOps-инженеры и системные администраторы, которым нужен современный кроссплатформенный скриптовый инструмент
- Пользователи Ubuntu, Debian, Linux Mint (и совместимых дистрибутивов)

📦 Требования:

- Ubuntu 20.04+, Debian 11+, Linux Mint 20+
- PowerShell 7.2+
- VS Code (рекомендуется)
- Желательно знание bash/cron/systemd

📚 Структура курса

Модуль 0. Введение

- Что такое PowerShell 7 и зачем он на Linux
- Отличия от bash
- Установка PowerShell 7 на Ubuntu/Debian/Mint
- Запуск PowerShell в терминале (`pwsh`)
- Установка и настройка VS Code с расширением PowerShell

Модуль 1. Основы PowerShell на Linux

- PowerShell как оболочка и язык
- Сравнение с Bash: переменные, команды, пайплайн
- Cmdlet'ы и структуры объектов
- `Get-Help`, `Get-Command`, `Get-Member`

Модуль 2. Работа с файлами и данными

- Работа с текстами, CSV, JSON, XML
- Чтение/запись файлов (`Get-Content`, `Out-File`)

- Работа с `ConvertTo-Json`, `ConvertFrom-Json`
- Пример: обработка логов из `/var/log`

Модуль 3. Работа с системой

- Системные команды в PowerShell (`ps`, `top`, `uptime`, `df`)
- Аналоги `systemctl`, `journalctl` через PowerShell
- Получение и фильтрация процессов
- Пример: скрипт мониторинга служб и логов

Модуль 4. Сетевые задачи и API

- Получение HTTP-запросов (`Invoke-WebRequest`, `Invoke-RestMethod`)
- Работа с REST API
- Проверка доступности серверов и сервисов
- Пример: мониторинг статуса сайтов

Модуль 5. Условные конструкции и циклы

- `if`, `switch`, `foreach`, `while`
- Обработка ошибок (`try/catch`)
- Примеры: резервное копирование, парсинг файлов, автоочистка

Модуль 6. Автоматизация и планирование

- Автозапуск скриптов
- PowerShell + cron: организация расписаний
- Создание логов, отчётов, email-уведомлений
- Пример: ежедневная проверка системы с отчетом

Модуль 7. Модули, функции и структуры

- Написание собственных функций
- Переиспользуемые модули
- Структура модулей PowerShell

- Пример: модуль для работы с PostgreSQL или Nginx

Модуль 8. Безопасность скриптов

- Защита паролей и токенов
- Ограничения исполнения (ExecutionPolicy)
- Пример: безопасная авторизация к API

Модуль 9. Расширенные темы

- Удалённое управление через SSH
- PowerShell + Ansible: возможно ли?
- PowerShell + Docker, Kubernetes
- Интеграция с Bash-скриптами

📦 Дополнительно:

- Готовые шаблоны скриптов
- Мини-библиотека команд для Linux-админов
- Тесты и задания после каждого модуля

Модуль 0. Введение в PowerShell 7 на Linux (Debian, Ubuntu, Mint)

🎯 Цель модуля:

Познакомиться с PowerShell 7 как кроссплатформенным инструментом автоматизации, научиться устанавливать и запускать его на системах Debian/Ubuntu/Mint, а также подготовить среду разработки с VS Code.

🔍 0.1 Что такое PowerShell и зачем он на Linux?

PowerShell — это командная оболочка и язык сценариев от Microsoft. Начиная с версии 7, это полностью **открытый проект** с поддержкой Linux, macOS и Windows.

Зачем использовать PowerShell на Linux?

- Автоматизация повторяющихся задач
- Обработка данных (CSV, JSON, XML)
- Работа с API и сетевыми сервисами
- Удалённое администрирование
- Современная альтернатива Bash в ряде задач

> PowerShell — это "Bash с объектами". Всё, что возвращает команда, — это объекты, а не строки.

💻 0.2 Установка PowerShell 7 на Debian/Ubuntu/Mint

🩺 Проверка системы

```
lsb_release -a
uname -m
```

> Требуется: Debian 10+, Ubuntu 20.04+, архитектура x64 или ARM64

📦 Установка через Microsoft APT-репозиторий

1. Установить зависимости:

```
sudo apt update && sudo apt install -y wget apt-transport-https software-properties-common
```

2. Добавить ключ и репозиторий:

```
wget -q https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
```

> Для Debian или Mint нужно заменить URL и конфигурацию — уточняется на https://learn.microsoft.com/powershell

3. Установка PowerShell:

```
sudo apt update
sudo apt install -y powershell
```

☑ Проверка установки

```
pwsh
```

Должен открыться интерактивный PowerShell (pwsh). Введите:

```
Write-Host "Hello, Linux PowerShell!"
```

🛠 0.3 Установка VS Code и расширения PowerShell

Установка VS Code

```
sudo snap install code --classic
```

Установка расширения PowerShell:

1. Открыть VS Code
2. Перейти в Extensions (Ctrl+Shift+X)
3. Найти "PowerShell" от Microsoft и установить

После установки VS Code будет распознавать `.ps1` скрипты и предлагать автозаполнение, подсветку синтаксиса и запуск прямо из редактора.

🧪 0.4 Первые команды

```
Get-Date
Get-Process
Get-ChildItem
```

Все команды возвращают **объекты**, не строки. Их можно фильтровать, сортировать, передавать в конвейере.

```
Get-Process | Where-Object {$_.CPU -gt 1} | Sort-Object CPU -Descending
```

☑ Что дальше?

Теперь вы готовы изучать PowerShell на практике. Следующий модуль — основы команд и синтаксиса.

Дополнительные ресурсы:

- https://learn.microsoft.com/powershell
- https://github.com/PowerShell/PowerShell
- https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell
- https://github.com/jidibinlin/Free-DevOps-Books-1/blob/master/book/PowerShell%20in%20Depth.pdfcode
-
- https://github.com/jidibinlin/Free-DevOps-Books-1/blob/master/book/PowerShell%20in%20Depth.pdf

Модуль 1. Основы PowerShell на Linux

🎯 Цель модуля:

Познакомиться с PowerShell как полноценной оболочкой и языком сценариев на Linux, понять отличия от Bash, научиться использовать cmdlet'ы и работать с объектами.

🐚 1.1 PowerShell как оболочка и язык

PowerShell — это не только командная строка, но и полноценный объектно-ориентированный язык. Он объединяет в себе возможности оболочки, языка сценариев и управления системой.

Пример запуска:

```
pwsh
```

Теперь вы внутри PowerShell. Все команды называются **cmdlet'ами** (читается как "камдлет"). Их формат:

```
Глагол-Существительное
```

Например: `Get-Process`, `Set-Item`, `Remove-Item`

⚖ 1.2 Сравнение с Bash

Задача	Bash	PowerShell
Переменная	`MYVAR=hello`	`$myVar = "hello"`
Вывод переменной	`echo $MYVAR`	`$myVar`
Цикл `for`	`for i in ...; do ...`	`foreach ($i in ...) { ... }`
Получение списка	`ls`	`Get-ChildItem`
Поиск по тексту	`grep`	`Select-String`
Конвейер	строки (`	`)

Главное отличие: в PowerShell команды передают **объекты**, а не строки.

💠 1.3 Cmdlet'ы и структуры объектов

Cmdlet — это команда PowerShell. Каждая из них возвращает **объекты**, у которых есть свойства и методы.

Пример:

```
Get-Process
```

Возвращает список процессов. Каждый — это объект типа `System.Diagnostics.Process`

Посмотрим свойства:
```
Get-Process | Get-Member
```

Фильтрация:
```
Get-Process | Where-Object { $_.CPU -gt 1 }
```

Сортировка:
```
Get-Process | Sort-Object CPU -Descending
```

📄 1.4 Базовые команды справки

🔍 Получить помощь по команде:
```
Get-Help Get-Process
```

🔎 Найти команду по ключевому слову:
```
Get-Command *service*
```

🔬 Получить структуру объекта:
```
Get-Service | Get-Member
```

🧪 Примеры

📌 Пример 1: Просмотр процессов с использованием фильтрации
```
Get-Process | Where-Object { $_.WorkingSet -gt 100MB }
```

Выводит процессы, использующие более 100 МБ оперативной памяти.

📌 Пример 2: Список служб, отсортированный по статусу
```
Get-Service | Sort-Object Status
```

📌 Пример 3: Поиск строк в файле журнала
```
Get-Content /var/log/syslog | Select-String "error"
```

(Работает на Linux, если есть доступ к системным логам)

🧠 Задания для практики

1. Создайте переменную `$name` со значением вашего имени и выведите её на экран.
2. Используйте `Get-Command` для поиска всех команд, содержащих слово "item".
3. Выведите список всех служб (`Get-Service`), отфильтровав только те, которые в статусе "Running".
4. Используйте `Select-String` для поиска слова "sudo" в файле `/var/log/auth.log`.
5. Узнайте, сколько памяти использует процесс `pwsh`, используя `Get-Process` и фильтрацию по `ProcessName`.

✅ Что дальше?

Теперь вы понимаете основы работы с PowerShell: синтаксис, объекты, справка, основные команды. Следующий модуль — работа с файлами и форматами данных: CSV, JSON, XML и логами Linux.

📚 Полезно почитать:

- `Get-Help` — лучший друг новичка
- https://learn.microsoft.com/powershell
- `Get-Command`, `Get-Member`, `Where-Object`, `Sort-Object` — ядро повседневной работы

📂 Модуль 2. Работа с файлами и данными

🎯 **Цель модуля:**
Научиться работать с различными форматами данных (текст, CSV, JSON, XML), освоить команды чтения и записи файлов, а также научиться обрабатывать системные логи в PowerShell на Linux.

📄 2.1 Работа с текстами, CSV, JSON и XML

PowerShell позволяет легко обрабатывать файлы разных форматов. Все данные представлены в виде объектов, что делает возможной гибкую фильтрацию и трансформацию.

📌 Примеры форматов:

- **Текст:** обычные .txt и .log файлы — читаются как строки, но могут быть разбиты по строкам, словам, шаблонам.
- **CSV:** табличные данные, которые превращаются в объекты с именованными полями.
- **JSON:** структура ключ-значение — отлично работает с ConvertFrom-Json и ConvertTo-Json.
- **XML:** поддерживается через [xml]каст и доступ к узлам как к свойствам объекта.

📂 2.2 Чтение и запись файлов

Основные команды:

◇ **Чтение файлов:**

```
Get-Content /var/log/syslog
```

Читает содержимое файла построчно.

◇ **Запись в файл:**

```
"Привет, Linux PowerShell!" | Out-File ~/example.txt
```

Записывает строку в файл.

🧪 Пример: запись результатов команды в файл:

```
Get-Process | Where-Object {$_.CPU -gt 1} | Out-File ~/high-cpu.txt
```

🧠 2.3 Работа с JSON

JSON-формат особенно полезен при взаимодействии с API и хранения конфигураций.

📌 Пример преобразования объектов в JSON:

```
Get-Process | Select-Object Name, CPU | ConvertTo-Json
```

📌 Пример чтения JSON:

```
$json = Get-Content ~/config.json | ConvertFrom-Json
$json.Database.Host
```

🧪 2.4 Пример: Обработка логов из /var/log

Рассмотрим анализ системных логов на примере файла `/var/log/auth.log`.

📌 Поиск упоминаний входа по SSH:

```
Get-Content /var/log/auth.log | Select-String "sshd"
```

📌 Подсчёт числа попыток входа:

```
(Get-Content /var/log/auth.log | Select-String "sshd").Count
```

📌 Фильтрация по дате:

```
Get-Content /var/log/auth.log | Where-Object { $_ -match "Apr 24" }
```

🧠 Это может быть полезно для:

- анализа попыток входа,
- выявления ошибок безопасности,
- создания собственных систем мониторинга.

🧠 Задания для практики

- Создайте файл `~/test.txt` и запишите в него список процессов с использованием `Get-Process` и `Out-File`.
- Преобразуйте результат `Get-Service` в формат JSON и сохраните в файл.
- Считайте данные из любого JSON-файла и выведите на экран одно из вложенных свойств.
- Используйте `Select-String`, чтобы найти ошибки в логах `/var/log/syslog`.
- Подсчитайте количество строк в файле `/var/log/auth.log`, содержащих дату сегодняшнего дня.

✅ **Что дальше?**
Теперь вы умеете читать и записывать файлы, обрабатывать структурированные данные и анализировать логи. Следующий модуль познакомит вас с командами управления системой в PowerShell.

📎 **Полезно почитать:**

- `Get-Help Get-Content`
- `Get-Help ConvertTo-Json`
- Документация по JSON в PowerShell
- Документация PowerShell по CSV и XML

🖳 Модуль 3. Работа с системой

🎯 **Цель модуля:**
Научиться использовать PowerShell для получения информации о состоянии системы, управления службами и мониторинга процессов, а также создания скриптов, аналогичных systemctl и journalctl.

⚙ 3.1 Системные команды в PowerShell

PowerShell предоставляет команды, аналогичные популярным утилитам Linux: `ps`, `top`, `uptime`, `df` — но с объектным выводом, что делает их удобными для анализа.

📌 Примеры:

- **Процессы:**

```
Get-Process
```

- **Аптайм системы (через .NET):**

```
(New-TimeSpan -Start (Get-CimInstance
Win32_OperatingSystem).LastBootUpTime).ToString()
```

- **Информация о дисках:**

```
Get-PSDrive -PSProvider FileSystem
```

- **Нагрузка системы (аналог top):**

```
Get-Process | Sort-Object CPU -Descending | Select-Object -First 10
```

🛠 3.2 Аналоги systemctl и journalctl через PowerShell

PowerShell не заменяет systemd, но может вызывать внешние команды или оборачивать их в сценарии.

📌 Примеры:

- **Проверка статуса службы:**

```
systemctl status ssh | Out-String
```

- **Запуск службы:**

```
Start-Process "systemctl" -ArgumentList "start ssh"
```

- **Чтение логов через journalctl:**

```
journalctl -u ssh -n 20 | Out-String
```

Для автоматизации удобно обернуть такие вызовы в функции.

🔍 3.3 Получение и фильтрация процессов

PowerShell позволяет детально фильтровать и сортировать процессы.

📌 Примеры:

- **Список всех процессов по имени:**

```
Get-Process | Where-Object {$_.Name -like "*ssh*"}
```

- **Сортировка по памяти:**

```
Get-Process | Sort-Object WorkingSet -Descending | Select-Object -First 5
```

- **Процессы текущего пользователя:**

```
Get-Process | Where-Object { $_.StartInfo.UserName -like "$env:USER*" }
```

🧪 3.4 Пример: скрипт мониторинга служб и логов

📌 Сценарий: следим за статусом службы `ssh` и пишем результаты в лог.

```
$service = "ssh"
$status = (systemctl is-active $service)
$log = "$(Get-Date): Служба $service — $status"
$log | Out-File -FilePath ~/ssh-status.log -Append
```

📌 Расширенный вариант с логами:

```
$entries = journalctl -u ssh -n 5 | Out-String
$timestamp = Get-Date
@"
[$timestamp] Состояние службы: $status
Последние записи журнала:
$entries
"@ | Out-File ~/ssh-monitor.log -Append
```

🧠 Задания для практики

- Выведите список всех процессов, занимающих более 100 МБ памяти.
- Используйте `systemctl` через PowerShell, чтобы перезапустить службу `cron` и зафиксировать это в логе.
- Создайте PowerShell-функцию, которая проверяет статус службы и возвращает "OK"/"Ошибка".
- Считайте последние 10 записей `journalctl` для любой службы и выведите их на экран.

- Напишите мини-скрипт, который проверяет доступность нескольких системных служб.

✅ **Что дальше?exit**

📝 **Полезно почитать:**

- `Get-Process`, `Sort-Object`, `Where-Object`
- `Start-Process`, `Out-File`
- Документация `systemctl` и `journalctl` на Linux
- PowerShell + Linux: управление системой

🌐 Модуль 4. Сетевые задачи и API

🎯 **Цель модуля:**
Освоить работу с HTTP-запросами, научиться взаимодействовать с REST API, проверять доступность удалённых ресурсов и автоматизировать сетевой мониторинг с помощью PowerShell на Linux.

🌍 4.1 Получение HTTP-запросов

PowerShell предоставляет два основных инструмента для работы с веб-запросами:

📌 **Invoke-WebRequest** — для загрузки HTML-страниц, файлов и проверки доступности сайтов:

```
Invoke-WebRequest https://www.wikipedia.org
```

📌 **Invoke-RestMethod** — для работы с REST API (JSON, XML и др.):

```
Invoke-RestMethod https://api.github.com/repos/PowerShell/PowerShell
```

Результатом будет объект, содержащий все поля JSON как свойства.

🔗 4.2 Работа с REST API

REST API активно используется в современных сервисах. PowerShell упрощает обращение к ним и разбор полученных данных.

📌 Пример: Получение информации о репозитории PowerShell на GitHub:

```
$response = Invoke-RestMethod https://api.github.com/repos/PowerShell/PowerShell
$response.stargazers_count
```

📌 Пример POST-запроса (отправка данных):

```
Invoke-RestMethod -Uri "https://reqres.in/api/login" -Method Post -Body
@{email="eve.holt@reqres.in"; password="cityslicka"}
```

💡 Сервис https://reqres.in предоставляет открытое API для тренировки.

🛡️ 4.3 Проверка доступности серверов и сервисов

Проверка сетевой доступности — важный элемент мониторинга.

📌 Пинг:

```
Test-Connection -ComputerName www.google.com -Count 2
```

📌 Проверка HTTPS-порта:

```
Test-NetConnection -ComputerName www.microsoft.com -Port 443
```

📌 Обёртка в собственную функцию:

```
function Test-Site {
    param([string]$url)
    $res = Invoke-WebRequest -Uri $url -UseBasicParsing -ErrorAction
SilentlyContinue
    if ($res.StatusCode -eq 200) {"$url доступен"} else {"Ошибка на $url"}
}
```

🧪 4.4 Пример: мониторинг статуса сайтов

Сценарий: автоматическая проверка нескольких сайтов и логирование результата.

```
Goft.com",
    "https://api.github.com",
    "https://www.openstreetmap.org"
)

foreach ($site in $sites) {
    try {
        $r = Invoke-WebRequest -Uri $site -UseBasicParsing -TimeoutSec 5
        "$site OK: $($r.StatusCode)" | Out-File ~/site-monitor.log -Append
    } catch {
        "$site FAIL: $($_.Exception.Message)" | Out-File ~/site-monitor.log -
Append
    }
}
```

🧠 Задания для практики

- Используйте `Invoke-WebRequest`, чтобы загрузить HTML содержимое главной страницы Википедии.
- Получите с помощью `Invoke-RestMethod` список открытых задач из API GitHub.
- Проверьте доступность сайтов Google, Microsoft и OpenStreetMap.
- Напишите скрипт, который по расписанию (cron) проверяет статус указанных ресурсов.
- Преобразуйте JSON из `https://jsonplaceholder.typicode.com/users` в таблицу и отсортируйте по имени.

✅ **Что дальше?**
Теперь вы умеете взаимодействовать с API, проверять доступность сайтов и создавать сетевые сценарии на PowerShell. Далее — циклы, условия и обработка ошибок для устойчивых автоматизированных решений.

📝 **Полезно почитать:su**

- `Get-Help Invoke-WebRequest`
- `Get-Help Invoke-RestMethod`
- Документация PowerShell по сетевым командам
- Справочные API:
  https://api.github.com,
  https://reqres.in,
  https://jsonplaceholder.typicode.com

🌀 Модуль 5. Условные конструкции и циклы

🎯 **Цель модуля:**
Научиться строить логические условия, организовывать циклы и грамотно обрабатывать ошибки в скриптах PowerShell, чтобы создавать устойчивые и надёжные автоматизации на Linux.

💠 5.1 Условные конструкции: if, switch

PowerShell поддерживает мощные конструкции для проверки условий и принятия решений.

📌 Примеры:

- **If / Else:**

```
$cpu = (Get-Process | Measure-Object CPU -Sum).Sum
if ($cpu -gt 1000) {
    Write-Host "Высокая загрузка CPU"
} else {
    Write-Host "CPU в норме"
}
```

- **Switch:**

```
$service = "ssh"
switch ($service) {
    "ssh" { Write-Host "Работаем с SSH"; break }
    "nginx" { Write-Host "Работаем с Nginx"; break }
    default { Write-Host "Неизвестная служба" }
}
```

🔁 5.2 Циклы: foreach, while

Циклы позволяют многократно выполнять действия над коллекциями данных.

📌 Примеры:

- **foreach:**

```
$sites = @("https://google.com", "https://github.com")
foreach ($site in $sites) {
    Invoke-WebRequest -Uri $site -UseBasicParsing
}
```

- **while:**

```
$count = 0
while ($count -lt 5) {
    Write-Host "Итерация $count"
    $count++
}
```

🚨 5.3 Обработка ошибок: try/catch

PowerShell позволяет перехватывать ошибки для корректного реагирования на сбои.

📌 Пример:

```
try {
    Invoke-WebRequest -Uri "https://nonexistent.website" -ErrorAction Stop
} catch {
    Write-Host "Ошибка при подключении: $($_.Exception.Message)"
}
```

💡 Использование `-ErrorAction Stop` важно для принудительного выброса ошибки внутри `try`.

🧪 5.4 Примеры из практики

📌 **Резервное копирование важных файлов:**

```
$source = "/etc/ssh/sshd_config"
$destination = "/home/user/backup/"
if (Test-Path $source) {
    Copy-Item $source -Destination $destination
    Write-Host "Бэкап успешно создан."
} else {
    Write-Host "Файл не найден."
}
```

📌 **Парсинг логов с условной обработкой ошибок:**

```
try {
    Get-Content /var/log/auth.log | Select-String "Failed password"
} catch {
    Write-Host "Ошибка чтения логов!"
}
```

🧠 Задания для практики

- Напишите скрипт, который проверяет статус 3-х служб (`ssh`, `cron`, `nginx`) через `systemctl` и выводит их состояние с помощью `switch`.
- Используйте `foreach`, чтобы пройтись по списку IP-адресов и проверить их доступность.
- Напишите скрипт, который копирует файлы только если они существуют (`if`) и логирует все ошибки (`try/catch`).
- Реализуйте цикл `while`, который повторяет проверку сайта до тех пор, пока он недоступен.
- Используйте `try/catch`, чтобы корректно обрабатывать неудачные HTTP-запросы.

✅ **Что дальше?**
Теперь вы умеете строить условия, писать циклы и обрабатывать ошибки в PowerShell! Следующий модуль — **автоматизация и планирование задач через cron и скрипты**.

📝 **Полезно почитать:**

- `Get-Help if`, `Get-Help switch`, `Get-Help foreach`, `Get-Help while`
- Документация по обработке ошибок в PowerShell
- Базовые паттерны написания скриптов на PowerShell

🛠️ Модуль 6. Автоматизация и планирование

🎯 **Цель модуля:**
Научиться запускать PowerShell-скрипты по расписанию через `cron`, создавать автоматические отчёты и уведомления. Построим полноценную базу для ежедневных задач без ручного вмешательства!

📅 6.1 Автозапуск скриптов

Запуск PowerShell-скриптов вручную — не всегда удобно.
Автоматизация начинается с правильного автозапуска.

📌 Пример запуска скрипта вручную:

```
pwsh /home/user/scripts/check-services.ps1
```

📌 Делаем скрипт исполняемым:

```
chmod +x /home/user/scripts/check-services.ps1
```

И добавляем шебанг в начало скрипта:

```
#!/usr/bin/env pwsh
```

Теперь скрипт можно вызывать напрямую.

⏰ 6.2 PowerShell + cron: организация расписаний

На Linux для планирования заданий традиционно используется `cron`.

📌 Открыть редактор заданий:

```
crontab -e
```

📌 Добавить задачу:

```
0 6 * * * /home/user/scripts/daily-report.ps1
```

◇ Объяснение:

- `0 6 * * *` — каждый день в 6:00 утра.
- Запускается скрипт `daily-report.ps1`.

💡 Не забудьте использовать полные пути и указывать оболочку pwsh.

📃 6.3 Создание логов, отчётов и email-уведомлений

📌 Пример записи результата в лог:

```
$date = Get-Date -Format "yyyy-MM-dd HH:mm:ss"
"[$date] Служба SSH активна." | Out-File ~/logs/ssh-check.log -Append
```

📌 Пример отправки email (если настроен `sendmail` или аналог):

```
Send-MailMessage -From "monitor@example.com" -To "admin@example.com" -Subject
"Отчёт о проверке" -Body "Все системы работают нормально." -SmtpServer
"smtp.example.com"
```

🧠 В связке с cron можно сделать полноценный мониторинг с ежедневными отчётами уведомлениями.

🧪 6.4 Пример: ежедневная проверка системы с отчётом

Мини-скрипт для ежедневного контроля служб:

```
$services = @("ssh", "cron", "nginx")
foreach ($service in $services) {
    $status = (systemctl is-active $service)
    "$service: $status" | Out-File ~/logs/service-status.log -Append
}
Send-MailMessage -From "server@example.com" -To "admin@example.com" -Subject
"Ежедневный отчёт о службах" -Body (Get-Content ~/logs/service-status.log | Out-
String) -SmtpServer "smtp.example.com"
```

💡 Такой скрипт легко привязать к `cron` для автоматической отправки ежедневных отчётов.

🧠 Задания для практики

- Напишите скрипт для проверки загрузки диска (`Get-PSDrive`) и логируйте превышение 80% использования.
- Настройте `cron`, чтобы запускать скрипт проверки сайтов каждые 6 часов.
- Реализуйте отправку email-уведомлений о статусе служб через PowerShell.
- Создайте лог-файл, в который будет добавляться результат проверки служб каждый день.
- Сделайте скрипт самодостаточным: если папка логов не существует — он её создаёт.

✅ **Что дальше?**
Теперь вы умеете планировать скрипты, создавать автоматические отчёты и уведомления. Следующий модуль — **модули, функции и структуры в PowerShell** — сделает ваши скрипты ещё мощнее и удобнее для повторного использования!

📝 **Полезно почитать:**

- `Get-Help cron`, `Get-Help Send-MailMessage`
- Документация по планированию задач в Linux
- Примеры PowerShell-автоматизаций на GitHub

🛠️ Модуль 7. Модули, функции и структуры

🎯 **Цель модуля:**
Научиться создавать собственные функции и модули в PowerShell, упорядочивать код в логичные структуры и использовать переиспользуемые сценарии для реальной автоматизации в Linux-среде.

🧩 7.1 Написание собственных функций

Функции позволяют переиспользовать код и делают скрипты более чистыми и удобными для сопровождения.

📌 Простейшая функция:

```
function Get-Hello {
    param([string]$Name)
    Write-Host "Привет, $Name!"
}
```

📌 Вызов функции:

```
Get-Hello -Name "Linux-админ"
```

💡 Функции можно выносить в отдельные файлы и подключать при необходимости.

📦 7.2 Переиспользуемые модули

Модули — это собрания функций, переменных и логики, объединённые в один файл для удобного подключения.

📌 Создание простого модуля:

1. Файл `MyTools.psm1`:

```
function Test-Site {
    param([string]$url)
    try {
        $res = Invoke-WebRequest -Uri $url -UseBasicParsing -TimeoutSec 5
        if ($res.StatusCode -eq 200) { return "$url доступен" }
        else { return "$url ошибка" }
    } catch {
        return "$url недоступен"
    }
}
```

2. Импорт модуля:

```
Import-Module /home/user/modules/MyTools.psm1
```

3. Вызов функции из модуля:

```
Test-Site -url "https://github.com"
```

🔩 7.3 Структура модулей PowerShell

Чтобы модуль работал корректно, желательно соблюдать структуру:

📂 Каталог модуля:

```
MyModule/
├── MyModule.psd1 (манифест, необязательно для простых модулей)
├── MyModule.psm1 (код модуля)
├── README.md (описание функций)
```

💡 Для простых задач достаточно одного файла `.psm1`. Для более сложных проектов лучше оформить полноценный `.psd1`-манифест.

💉 7.4 Пример: модуль для работы с PostgreSQL или Nginx

📌 Простейший модуль для мониторинга Nginx:

```
function Get-NginxStatus {
    $status = (systemctl is-active nginx)
    if ($status -eq "active") {
        return "Nginx работает"
    } else {
        return "Nginx не работает"
    }
}
```

Импортируем, вызываем, получаем быстрый ответ о состоянии сервера.

🧠 Задания для практики

- Напишите функцию для проверки статуса указанной службы (`systemctl is-active`).
- Создайте модуль с 3 функциями: проверка сайта, проверка службы, запись в лог.
- Импортируйте свой модуль в скрипт ежедневного мониторинга.
- Оформите структуру собственного модуля с README.
- Реализуйте функцию, которая подключается к API и парсит JSON-ответ.

✅ Что дальше?

Теперь вы умеете создавать модули и функции в PowerShell — ваш код становится масштабируемым и удобным для развития. В следующем модуле — **безопасность скриптов** и защита данных!

📝 **Полезно почитать:**

- `Get-Help function`, `Get-Help Import-Module`
- Документация по созданию модулей PowerShell
- Лучшие практики оформления скриптов для Linux и DevOps

🛡️ Модуль 8. Безопасность скриптов

🎯 **Цель модуля:**
Научиться защищать скрипты PowerShell: шифровать и скрывать чувствительные данные (пароли, токены), ограничивать выполнение скриптов с помощью Execution Policy, а также строить безопасные сценарии подключения к внешним сервисам и API.

🔐 8.1 Защита паролей и токенов

Никогда не храни пароли или токены открытым текстом в скриптах!

📌 Простейшее безопасное хранение:

1. Сохранить пароль в зашифрованном виде:

```
Read-Host "Введите пароль" -AsSecureString | ConvertFrom-SecureString | Out-File ~/secure-password.txt
```

2. Загрузить зашифрованный пароль в скрипте:

```
$securePassword = Get-Content ~/secure-password.txt | ConvertTo-SecureString
```

3. Расшифровать при необходимости:

Для использования, например, в аутентификации API или SSH.

🛡️ 8.2 Ограничение исполнения скриптов (Execution Policy)

Execution Policy защищает систему от запуска непроверенных скриптов.

📌 Проверить текущую политику:

```
Get-ExecutionPolicy
```

📌 Установить политику только для подписанных скриптов:

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Варианты политик:

- `Restricted` — скрипты запрещены
- `RemoteSigned` — локальные можно, скачанные должны быть подписаны
- `AllSigned` — любые скрипты должны быть подписаны
- `Unrestricted` — все разрешены (небезопасно)

🔑 8.3 Пример: безопасная авторизация к API

📌 Как подключиться к API с использованием защищённого токена:

```
$token = Get-Content ~/secure-token.txt | ConvertTo-SecureString
$plainToken = [System.Net.NetworkCredential]::new("", $token).Password

Invoke-RestMethod -Uri "https://api.example.com/data" -Headers @{ Authorization
= "Bearer $plainToken" }
```

✅ Таким образом токен не хранится открытым текстом в скрипте!

🧠 Задания для практики

- Создайте защищённый файл с зашифрованным паролем и напишите скрипт для его чтения.
- Проверьте текущую Execution Policy на своей машине.
- Попробуйте изменить Execution Policy только для своего пользователя на `RemoteSigned`.
- Exit
-

✅ **Что дальше?**
Теперь вы знаете, как создавать более защищённые и профессиональные скрипты в PowerShell! В следующем модуле — **расширенные темы: SSH, Ansible, Docker, Kubernetes и интеграция PowerShell с DevOps-миром.**

📚 **Полезно почитать:**

- `Get-Help Get-ExecutionPolicy`
- `Get-Help Set-ExecutionPolicy`
- Документация: Secure Credentials in PowerShell

🎯 ИТОГО

Модуль 8 полностью готов, кибербезопасность уровня DevOps включена! 🚀

🌐 Модуль 9. Расширенные темы

🎯 **Цель модуля:**
Научиться использовать PowerShell 7 для удалённого управления по SSH, интеграции с Ansible, работы с контейнерами через Docker и Kubernetes, а также комбинировать PowerShell с bash-скриптами в реальных DevOps-проектах.

📡 9.1 Удалённое управление через SSH

PowerShell 7 поддерживает **нативное подключение к удалённым Linux/Windows-серверам через SSH**.

📌 Подключение к удалённой машине:

```
Enter-PSSession -HostName server.example.com -UserName user -SSHTransport
```

📌 Запуск команд удалённо:

```
Invoke-Command -HostName server.example.com -UserName user -SSHTransport -
ScriptBlock { hostname; uptime }
```

✅ Очень удобно для управления серверами без поднятия WinRM.

🤝 9.2 PowerShell + Ansible: возможно ли?

Ansible традиционно ориентирован на YAML и bash, **но**:

- PowerShell скрипты можно вызывать через **Ansible tasks**.
- В Ansible можно подключать PowerShell как внешний скрипт:

📌 Пример playbook-фрагмента:

```
- name: Run PowerShell script
  ansible.builtin.shell: |
    pwsh /home/user/scripts/check-services.ps1
```

✅ Таким образом можно объединить автоматизацию Ansible и скрипты PowerShell на Linux!

🐳 9.3 PowerShell + Docker / Kubernetes

PowerShell позволяет управлять контейнерами через стандартные CLI-инструменты `docker` и `kubectl`.

📌 Пример запуска контейнера через PowerShell:

```
docker run -d --name nginx-server -p 80:80 nginx
```

📌 Пример команды для Kubernetes:

```
kubectl get pods
```

✅ PowerShell — это просто удобная оболочка для вызова любых CLI-команд.

🔀 9.4 Интеграция PowerShell и bash-скриптов

На Linux очень важно, что **PowerShell и bash могут работать вместе**:

📌 Вызов bash-команды из PowerShell:

```
bash -c "ls -la /etc"
```

📌 Вызов PowerShell-скрипта из bash:

```
pwsh /home/user/scripts/monitor.ps1
```

✅ Это позволяет легко объединять лучшее от обоих миров!

🧠 Задания для практики

- Подключитесь к своему серверу через `Enter-PSSession` по SSH.
- Напишите Ansible playbook, вызывающий ваш PowerShell-скрипт.
- Поднимите тестовый контейнер nginx через PowerShell.
- Выполните bash-команду из PowerShell-скрипта.
- Настройте mini-кластер через `kind` (Kubernetes in Docker) и управляйте через PowerShell.

📚 **Полезно почитать:**

- `Get-Help Enter-PSSession`
- `Get-Help Invoke-Command`
- Документация Ansible для PowerShell-скриптов
- Docker CLI и Kubernetes документация
- Официальное руководство по PowerShell Remoting over SSH

ПРИЛОЖЕНИЕ

🔌 Что такое SSH и зачем он?

SSH (Secure Shell) — это способ подключаться к другому компьютеру по сети, как будто ты сидишь прямо у него в терминале.
Полезно, если:

- ты хочешь управлять мини-ПК без монитора 🧒💻
- хочешь передавать файлы через `scp` или `sftp`
- настраиваешь сервак или просто хочешь выглядеть как хакер 😎

🛠️ Как включить SSH-сервер в Linux Mint 22

◇ Шаг 1: Установить OpenSSH Server

Mint по умолчанию ставит только **клиент**, но не сервер.

Открой терминал и введи:

```
sudo apt update
sudo apt install openssh-server
```

📦 Это установит пакет `openssh-server` и автоматически запустит службу `sshd`.

◇ Шаг 2: Проверить, работает ли SSH

```
systemctl status ssh
```

Увидишь что-то типа:

```
Active: active (running)
```

✅ Всё хорошо.
❌ Нет? Введи:

```
sudo systemctl enable --now ssh
```

◇ Шаг 3: Узнать IP-адрес

Введи: `ip a`

Или: `hostname -I`

Запомни IP, например: `192.168.1.42`

◇ Шаг 4: Подключаемся с другого компьютера

С Windows, Linux, macOS — не важно. Главное, чтобы был SSH-клиент.
В терминале другого ПК пиши:

```
ssh username@192.168.1.42
```

(где `username` — имя пользователя на Mint)

Пример:

```
ssh konrad@192.168.1.42
```

Введи пароль — и ты внутри! 🎉

🔒 Дополнительно: Фаервол и безопасность

▪ Mint использует `ufw` (Uncomplicated Firewall). Если включен, открой порт:

```
sudo ufw allow ssh
```

Проверь статус:

```
sudo ufw status
```

▪ Защити от брутфорса:

- Установи `fail2ban`
- Или отключи вход по паролю и оставь только вход по ключу (если интересно — расскажу как 🔐)

✅ Кратко по шагам:

1. `sudo apt install openssh-server`
2. `sudo systemctl enable --now ssh`
3. Узнать IP: `hostname -I`
4. С другого ПК: `ssh user@IP`
5. (опц.) `sudo ufw allow ssh`

🧠 **SSH = суперспособность**
С ним ты можешь управлять компьютером с кухни, с телефона— если там есть Wi-Fi.