Caddy

Part 1: Detailed Explanation of Caddy

What is Caddy?

Caddy is a modern, open-source web server and reverse proxy written in Go. It's designed with security, ease of use, and performance in mind. Its most famous feature is its ability to **automatically obtain and renew TLS certificates** from Let's Encrypt, making HTTPS the default.

Key Features & Why Caddy Stands Out

1. **Automatic HTTPS:** This is the killer feature. Caddy automatically provisions and renews TLS certificates for your domains. You don't have to manually install `certbot` or configure challenge responses. Just define your domain, and Caddy does the rest.
2. **Simple Configuration with Caddyfile:** The primary configuration method is a human-readable file called the `Caddyfile`. Its syntax is clean and intuitive, drastically reducing the complexity compared to Nginx or Apache configs.
3. **Powerful Reverse Proxying:** Caddy excels at being a reverse proxy. It can seamlessly route incoming requests to various backend services (like Node.js apps, Python APIs, Go services, other containers, etc.), handle load balancing, and rewrite headers.
4. **Built-in Logging and Metrics:** Caddy has structured logging out of the box and can export metrics in Prometheus format.
5. **Extensible with Modules (Caddy 2):** Caddy 2 is built on a modular architecture. You can extend its functionality with a wide array of plugins from the Caddy Download page.
6. **Memory-Safe:** Written in Go, it benefits from memory safety, reducing the risk of common vulnerabilities like buffer overflows.
7. **No Dependencies:** It's a single static binary. You can run it anywhere without installing a runtime or other libraries.

Core Concepts

- **Caddyfile:** The simple, declarative configuration file. Great for most use cases.
- **JSON Config:** Caddy's *native* configuration is JSON. The Caddyfile is actually converted to JSON under the hood. For very advanced, dynamic configurations, you can use JSON directly via the **Admin API**.
- **Sites:** A "site" in Caddy is a configuration block defined by a hostname (e.g., `example.com`). Each site can have its own set of rules.
- **Handlers:** These are the modules that perform tasks. For example, the `file_server` handler serves static files, the `reverse_proxy` handler routes requests to a backend, and the `rewrite` handler changes requests internally.

Part 2: User Guide - Getting Started & Common Configurations

1. Installation

Official Methods (Recommended):

- **Download the Binary:** Visit the Caddy Download Page, select your platform and any plugins you need, and download the binary.
- **Package Managers:**
  - **macOS:** `brew install caddy`
  - **Ubuntu/Debian:** Follow the instructions on the official website to add the repository and `apt install caddy`.
  - **Docker:** `docker pull caddy`

2. Basic Structure of a Caddyfile

The `Caddyfile` typically resides in the same directory from where you run Caddy.

text

```
# Caddyfile

# The site address - this is the key that starts a site block.

# Caddy will automatically manage HTTPS for this domain.

myapp.example.com {
    # Set a root directory for static files (optional for reverse proxy)
    root * /var/www/html
    # Use the file_server to serve static files
    file_server
    # Or, use reverse_proxy to route requests to a backend
    reverse_proxy localhost:3000
}
```

Important: The first line of a site block must be the address. `:80` will work for all domains on port 80, `localhost` will work for localhost.

3. Running Caddy

1. Save your configuration in a file named `Caddyfile`.
2. Open a terminal in that directory.
3. Run:

```
# bash

# If running in the foreground
caddy run

# Or, run as a service/demon (recommended for production)
caddy start
```

If your `Caddyfile` uses a public domain (e.g., `myapp.example.com`), ensure your DNS is pointing to your server's IP. Caddy will fail to start if it can't validate the domain.

4. Common Reverse Proxy Configurations

a) Basic API Proxy

Proxies all requests for `api.example.com` to a backend service running on port 8080.

text

```
api.example.com {
    reverse_proxy localhost:8080
}
```

b) Path-Based Routing

Routes different paths to different backend services. This is very common with microservices.

text

```
apps.example.com {
    # Route /api/* to the API service
    handle_path /api/* {
        reverse_proxy localhost:8001
    }
    # Route /admin/* to the admin dashboard
    handle_path /admin/* {
        reverse_proxy localhost:8002
    }
    # Route everything else to the main frontend app
    handle {
        reverse_proxy localhost:3000
    }
}
```

c) Load Balancing

Distributes traffic across multiple backend instances.

text

```
myapp.example.com {
    reverse_proxy node-server-1:3000 node-server-2:3000 node-server-3:3000 {
        # Choose a load balancing policy (optional)
        lb_policy first
    }
}
```

d) WebSocket Support

WebSocket support is built-in and usually works automatically. For tricky setups, you can be explicit.

text

```
chat.example.com {
    reverse_proxy localhost:5001 {
        # Force header rewrites for WS upgrade
        header_up X-Forwarded-Proto https
        header_up Connection {>Connection}
        header_up Upgrade {>Upgrade}
    }
}
```

e) Header Manipulation

Add, remove, or rewrite headers when proxying.

text

```
api.example.com {
    reverse_proxy localhost:8080 {
        # Add a header to the request sent to the backend
        header_up X-Real-IP {remote_host}
        # Remove a header from the request sent to the backend
        header_down -Server
    }
}
```

5. Serving Static Files

Caddy is also an excellent static file server.

text

```
static.example.com {
    # Set the root directory for this site
    root * /srv/static-files

    # Enable the file server
    file_server

    # Optional: Enable browsing (shows directory listings if no index.html)
    file_server browse
}
```

6. Advanced Configurations with `@` Matchers

You can define conditional blocks using matchers.

text

```
example.com {
    # Matcher named "api" for paths starting with /api or /v1
    @api {
        path /api/* /v1/*
    }

    # Apply a different proxy for API routes
    handle @api {
        reverse_proxy localhost:8080
    }

    # For everything else, serve static files
    handle {
        root * /var/www/html
        file_server
    }
}
```

Part 3: Production & Best Practices

1. The `Caddyfile` vs. JSON API

- **Use the `Caddyfile`** for manual, file-based configuration. It's perfect for most deployments, including Docker.
- **Use the JSON API** for dynamic, automated configurations, or when you need features not expressible in the Caddyfile. The API is used by tools like `docker-gen` or custom scripts.

2. Running as a Service

After installing via a package manager, Caddy typically registers itself as a systemd service.

- **Start:** `sudo systemctl start caddy`
- **Enable (start on boot):** `sudo systemctl enable caddy`

- **Check Status:** `sudo systemctl status caddy`
- **View Logs:** `sudo journalctl -u caddy -f`

3. File Structure for Production

A common Linux production structure:

text

```
/etc/caddy/
├── Caddyfile          # Main configuration file
└── sites/             # (Optional) You can split configs
    └── app1.caddy
/var/lib/caddy/        # Where Caddy stores TLS certificates, etc.
```

4. Using Docker

A simple `docker-compose.yml`:

yaml

```yaml
version: "3.7"

services:
  caddy:
    image: caddy:2-alpine
    restart: unless-stopped
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./Caddyfile:/etc/caddy/Caddyfile
      - ./site:/srv
      - caddy_data:/data
      - caddy_config:/config

volumes:
  caddy_data:
  caddy_config:
```

Summary: Caddyfile Cheat Sheet

Directive	Purpose	Example
`reverse_proxy`	Proxy requests to a backend	`reverse_proxy localhost:3000`
`file_server`	Serve static files from a root	`file_server`
`root`	Set the root directory for files	`root * /var/www`
`handle`	Define a route handling block	`handle /api/* { ... }`
`handle_path`	Like handle, but strips the path prefix	`handle_path /admin/* { ... }`
`@name`	Define a named matcher	`@api { path /api/* }`
`header_up`	Set a header for the upstream request	`header_up X-Real-IP {remote}`
`header_down`	Set a header for the client response	`header_down -Server`
`encode gzip`	Enable gzip compression	`encode gzip`
`log`	Configure logging	`log { output file /var/log/caddy/access.log }`
`tls internal`	Use Caddy's internal CA for HTTPS (for local dev)	`tls internal`
`redir`	Redirect requests	`redir /old-url /new-url 301`

Caddy's philosophy is to simplify web server configuration without sacrificing power. Start with the `Caddyfile` for its incredible readability, and dive into the JSON API only when your needs become more complex. Its automatic HTTPS alone makes it a compelling choice for modern web infrastructure.

Configuring API Services using Revese Proxy Option

Let's say you have 3 services running on port 5001, 5002 and 5003. Here's how to configure Caddy as a reverse proxy for your three services with Caddy Serving on Port 55555

Option 1: Path-Based Routing (Recommended)

This routes different URL paths to different services:

text

```
:55555 {
    # Service 1 - accessible at http://your-domain:55555/service1
    handle_path /service1/* {
        reverse_proxy localhost:5001
    }
    # Service 2 - accessible at http://your-domain:55555/service2
    handle_path /service2/* {
        reverse_proxy localhost:5002
    }
    # Service 3 - accessible at http://your-domain:55555/service3
    handle_path /service3/* {
        reverse_proxy localhost:5003
    }
    # Optional: Default route if no path matches
    handle {
        respond "Available services: /service1, /service2, /service3"
    }
}
```

Option 2: Host-Based Routing (if you have domains)

If you have different domain names pointing to the same server:

text

```
service1.example.com:55555 {
    reverse_proxy localhost:5001
}
service2.example.com:55555 {
    reverse_proxy localhost:5002
}
service3.example.com:55555 {
    reverse_proxy localhost:5003
}
```

Option 3: Subdomain Routing on Same Port

text

```
:55555 {
    # Service 1 - accessible at http://service1.your-domain:55555
    handle host service1.* {
        reverse_proxy localhost:5001
    }
    # Service 2 - accessible at http://service2.your-domain:55555
    handle host service2.* {
        reverse_proxy localhost:5002
    }
    # Service 3 - accessible at http://service3.your-domain:55555
    handle host service3.* {
        reverse_proxy localhost:5003
    }
}
```

Option 4: Load Balancing (All traffic distributed)

If you want to load balance between all three services:

text

```
:55555 {
    reverse_proxy localhost:5001 localhost:5002 localhost:5003 {
        # Optional: choose load balancing policy
        lb_policy round_robin
    }
}
```

Option 5: Advanced Path Rewriting

If your services expect different base paths:

text

```
:55555 {
    # Service 1 - removes /api/v1 prefix when proxying
    handle_path /api/v1/* {
        rewrite * /api/v1{path}
        reverse_proxy localhost:5001
    }
    # Service 2 - maps /app to root of service 2
    handle_path /app/* {
        rewrite * /{path}
        reverse_proxy localhost:5002
    }
    # Service 3 - direct mapping
    handle_path /admin/* {
        reverse_proxy localhost:5003
    }
}
```

How to Set This Up:

1. **Create the Caddyfile:**

   bash

```
mkdir -p ~/caddy-config
cd ~/caddy-config
nano Caddyfile
```

2. **Paste your chosen configuration**

3. **Run Caddy:**

bash

```
# Test configuration
caddy validate

# Run in foreground
caddy run

# Or run as daemon in background
caddy start
```

4. **Test your setup:**

bash

```
curl http://localhost:55555/service1/some-endpoint
curl http://localhost:55555/service2/some-endpoint
curl http://localhost:55555/service3/some-endpoint
```

Complete Working Example (Option 1):

Here's a production-ready configuration with additional features:

text

```
:55555 {
    # Enable logging
    log {
        output file /var/log/caddy/access.log
    }
    # Service 1 - API service
    handle_path /api/* {
        reverse_proxy localhost:5001 {
            # Add headers for the backend
            header_up X-Forwarded-Host {host}
            header_up X-Forwarded-Port {port}
            header_up X-Real-IP {remote}
        }
    }
    # Service 2 - Web app
    handle_path /app/* {
        reverse_proxy localhost:5002 {
            # Support WebSockets
            header_up Connection {>Connection}
            header_up Upgrade {>Upgrade}
        }
    }
    # Service 3 - Admin panel
    handle_path /admin/* {
        reverse_proxy localhost:5003 {
            # Set a custom header
            header_up X-Service-Name "admin-panel"
        }
    }
    # Health check endpoint
    handle /health {
        respond "Caddy proxy is running on port 55555"
    }
    # Default route - show available services
    handle {
        respond "Caddy Reverse Proxy Running on Port 55555\n\nAvailable Services:\n-
/api/* -> :5001\n- /app/* -> :5002\n- /admin/* -> :5003"
    }
}
```

Docker Compose Example:

If you're running everything in Docker:

yaml

```yaml
version: '3.8'

services:
  caddy:
    image: caddy:2-alpine
    ports:
      - "55555:55555"
    volumes:
      - ./Caddyfile:/etc/caddy/Caddyfile
      - caddy_data:/data
    networks:
      - app-network

  service1:
    image: your-service1-image
    networks:
      - app-network

  service2:
    image: your-service2-image
    networks:
      - app-network

  service3:
    image: your-service3-image
    networks:
      - app-network

networks:
  app-network:

volumes:
  caddy_data:
```

And update the Caddyfile to use service names instead of `localhost`:

text

```
:55555 {
    handle_path /service1/* {
        reverse_proxy service1:5001
    }
    handle_path /service2/* {
        reverse_proxy service2:5002
    }
    handle_path /service3/* {
        reverse_proxy service3:5003
    }
}
```

The **path-based routing (Option 1)** is usually the most straightforward approach. Your services will be accessible at:

- `http://your-server:55555/service1`
- `http://your-server:55555/service2`
- `http://your-server:55555/service3`

Caddy Authentication of incoming requests

Caddy has powerful authentication capabilities. Here are several ways to add authentication to your reverse proxy setup:

Option 1: Basic Authentication (Simplest)

This adds username/password protection to all routes:

```
:55555 {
    # Enable basic auth for all routes
    basicauth {
        # Format: username hashed_password
        # Generate with: caddy hash-password
        admin
JDJhJDE0JEV4WmhVM3BubWo1bHZzZFB1R1V1SGVDc1V0N2pYWm1hZHNpZGRVWUxVbkt0c1hLdWllUzU2
        user
JDJhJDE0JG1ZcHA4U2VHNHpmZ0h0dWY1bXAuNE9XQW5LM1BNQ3ZQN3l2LjZvWGFJSHpXTFguT3VxQkZT
    }
    handle_path /service1/* {
        reverse_proxy localhost:5001
    }
    handle_path /service2/* {
        reverse_proxy localhost:5002
    }
    handle_path /service3/* {
        reverse_proxy localhost:5003
    }
}
```

To generate passwords:

bash

```
caddy hash-password --plaintext "your_password"
```

Option 2: JWT Token Authentication

For API tokens and modern auth:

text

```
:55555 {
    # JWT validation for all requests
    jwt {
        primary yes
        set auth url https://your-auth-server/.well-known/jwks.json
        # or use a static secret:
        # secret "your-jwt-secret-here"
    }
    handle_path /service1/* {
        reverse_proxy localhost:5001 {
            # Pass user info to backend
            header_up X-User-Id {jwt.sub}
            header_up X-User-Roles {jwt.roles}
        }
    }
    handle_path /service2/* {
        reverse_proxy localhost:5002 {
            header_up X-User-Id {jwt.sub}
        }
    }
    handle_path /service3/* {
        reverse_proxy localhost:5003 {
            header_up X-User-Id {jwt.sub}
        }
    }
    # Public health endpoint (no auth)
    handle /health {
        respond "OK"
    }
}
```

Option 3: Custom Auth Server (Reverse Proxy to Auth Service)

Route all requests through an authentication service first:

text

```
:55555 {
    # Forward to auth service for validation
    @preAuth path /service1/* /service2/* /service3/*
    handle @preAuth {
        reverse_proxy localhost:8000 {
            # Auth service validates and returns user info
            header_down Auth-Status OK
        }

        # If auth fails, the auth service should return 401/403
        # If auth succeeds, route to the actual service
        @authSuccess header Auth-Status OK
        handle @authSuccess {
            rewrite * {path}
            reverse_proxy localhost:5001 localhost:5002 localhost:5003 {
                @s1 path /service1/*
                handle @s1 {
                    reverse_proxy localhost:5001
                }
                @s2 path /service2/*
                handle @s2 {
                    reverse_proxy localhost:5002
                }
                @s3 path /service3/*
                handle @s3 {
                    reverse_proxy localhost:5003
                }
            }
        }
    }
}
```

Option 4: Caddy with External Auth Header

Validate against an external auth service:

text

```
:55555 {
    # Import auth plugin (you might need to build Caddy with this)
    # import auth
    handle_path /service1/* {
        # Forward to auth service for token validation
        reverse_proxy localhost:8000 {
            # Copy original request headers
            header_up X-Original-URL {uri}
            header_up X-Original-Method {method}

            # Auth service responds with 200 OK or 401/403
        }

        # Only proxy to actual service if auth succeeds
        @authorized header Auth-User *
        handle @authorized {
            reverse_proxy localhost:5001 {
                header_up X-User {http.auth.user}
            }
        }
    }

# Similar for other services...
}
```

Option 5: API Key Authentication

Simple API key validation:

text

```
:55555 {
     # Validate API key from header
    @apiKey header X-API-Key your-secret-api-key-123
    @missingApiKey not header X-API-Key your-secret-api-key-123


    handle @missingApiKey {
        respond "Missing or invalid API key" 401
    }

    handle_path /service1/* {
        reverse_proxy localhost:5001 {
            header_up X-API-Key-Valid "true"
        }
    }

    handle_path /service2/* {
        reverse_proxy localhost:5002 {
            header_up X-API-Key-Valid "true"
        }
    }

    handle_path /service3/* {
        reverse_proxy localhost:5003 {
            header_up X-API-Key-Valid "true"
        }
    }
}
```

Option 6: OAuth2/OpenID Connect

For more complex authentication (requires Caddy with auth plugin):

text

```
:55555 {
    # OAuth2 configuration
    oauth {
        provider github {
            client_id "your-client-id"
            client_secret "your-client-secret"
            scopes user:email
        }
        redirect_url "https://your-domain:55555/oauth2/callback"
        cookie_domain "your-domain"
    }

    handle /oauth2/* {
        respond "OAuth endpoints"
    }

    # Protect all services
    handle_path /service1/* {
        # This will automatically redirect to OAuth if not authenticated
        reverse_proxy localhost:5001 {
            header_up X-User {http.user}
        }
    }

    # ... similar for other services
}
```

Complete Working Example (Basic Auth + Public Health Check)

Here's a production-ready configuration with mixed authentication:

text

```
:55555 {
    # Generate passwords: caddy hash-password --plaintext "password"
    basicauth {
        admin
JDJhJDE0JEV4WmhVM3BubWo1bHZzZFB1R1V1SGVDc1V0N2pYWm1hZHNpZGRVWUxVbkt0c1hLdWllUzU2
        api_user
JDJhJDE0JG1ZcHA4U2VHNHpmZ0h0dWY1bXAuNE9XQW5LM1BNQ3ZQN3l2LjZvWGFJSHpXTFguT3VxQkZT
    }
    # Public endpoints (no auth required)
    handle /health {
        respond "OK" 200
    }
    handle /public/* {
        respond "Public endpoint" 200
    }
    # Protected endpoints
    handle_path /service1/* {
        reverse_proxy localhost:5001 {
            # Pass auth info to backend
            header_up X-Authenticated-User {http.auth.user.id}
        }
    }
    handle_path /service2/* {
        reverse_proxy localhost:5002 {
            header_up X-Authenticated-User {http.auth.user.id}
        }
    }
    handle_path /service3/* {
        reverse_proxy localhost:5003 {
            header_up X-Authenticated-User {http.auth.user.id}
        }
    }
    # Log authentication events
    log {
        output file /var/log/caddy/auth.log
        format filter {
            wrap console
            fields {
                auth_user {http.auth.user.id}
            }
        }
    }
}
```

Testing Your Authentication

For Basic Auth:

bash

```
# Without auth (should get 401)

curl http://localhost:55555/service1/test

#With auth (replace with your credentials)

curl -u admin:password http://localhost:55555/service1/test

# Or with header

curl -H "Authorization: Basic $(echo -n 'admin:password' | base64)"
http://localhost:55555/service1/test
```

For API Key:

bash

```
curl -H "X-API-Key: your-secret-api-key-123" \With auth (replace with your credentials)
```

Recommendations

1. **Start with Basic Auth** if you need simple username/password protection
2. **Use JWT** for API-based applications
3. **Custom Auth Server** gives you the most flexibility for complex requirements
4. **Always pass auth information** to your backend services via headers

The Basic Authentication option is the easiest to implement and comes built-in with Caddy, making it a great starting point for most use cases.

Custom Cookie Authentication

For authenticating a custom cookie, **Option 2 (JWT Token Authentication) is your best choice**, but I'll show you a more specific configuration tailored for custom cookie authentication.

Option 1: Custom Cookie Validation with Caddy (Recommended)

This validates a custom cookie on every request:

text

```
:55555 {
    # Validate custom cookie for all protected routes
    @hasValidCookie header Cookie *my-auth-cookie=valid-token-*
    @missingCookie not header Cookie *my-auth-cookie=*
    # Block requests without the cookie
    handle @missingCookie {
        respond "Missing authentication cookie" 401
    }
    # Block requests with invalid cookie
    handle @hasValidCookie {
        # Your actual service routing
        handle_path /service1/* {
            reverse_proxy localhost:5001 {
                # Pass cookie info to backend
                header_up X-User-ID {http.auth.user.id}
                header_up X-Auth-Valid true
            }
        }
        handle_path /service2/* {
            reverse_proxy localhost:5002 {
                header_up X-User-ID {http.auth.user.id}
                header_up X-Auth-Valid true
            }
        }
        handle_path /service3/* {
            reverse_proxy localhost:5003 {
                header_up X-User-ID {http.auth.user.id}
                header_up X-Auth-Valid true
            }
        }
    }
    # Public endpoints (no auth required)
    handle /health {
        respond "OK" 200
    }
    handle /login {
        respond "Login endpoint" 200
    }
}
```

Option 2: Advanced Cookie Validation with External Service

If you need to validate the cookie against an auth service:

text

```
:55555 {
    # Forward to auth service for cookie validation
    @protected path /service1/* /service2/* /service3/*
    handle @protected {
        # Call auth service to validate the cookie
        reverse_proxy localhost:8000 {
            # Pass the original cookie and request details
            header_up X-Original-Path {path}
            header_up X-Original-Method {method}

            # Auth service responds with:
            # - 200 OK + X-User-ID header (if valid)
            # - 401/403 (if invalid)
        }

        # If auth service returns success, route to actual service
        @authSuccess header X-Auth-Status valid
        handle @authSuccess {
            rewrite * {path}

            # Route to appropriate service
            handle_path /service1/* {
                reverse_proxy localhost:5001 {
                    header_up X-User-ID {http.header.X-User-ID}
                }
            }
            handle_path /service2/* {
                reverse_proxy localhost:5002 {
                    header_up X-User-ID {http.header.X-User-ID}
                }
            }
            handle_path /service3/* {
                reverse_proxy localhost:5003 {
                    header_up X-User-ID {http.header.X-User-ID}
                }
            }
        }

        # If auth fails, return error
        @authFailed header X-Auth-Status invalid
        handle @authFailed {
            respond "Invalid authentication cookie" 401
        }
    }

    # Public routes
```

```
    handle /public/* {
        respond "Public content" 200
    }
}
```

Option 3: JWT in Cookie (Most Secure)

If your custom cookie contains a JWT token:

text

```
:55555 {
    # Extract JWT from cookie and validate
    jwt {
        primary yes
        secret "your-jwt-signing-secret"
        # Extract token from cookie instead of header
        token_source cookie my-auth-cookie
    }
    handle_path /service1/* {
        reverse_proxy localhost:5001 {
            # Pass JWT claims to backend
            header_up X-User-ID {jwt.sub}
            header_up X-User-Roles {jwt.roles}
            header_up X-Auth-Source "jwt-cookie"
        }
    }
    handle_path /service2/* {
        reverse_proxy localhost:5002 {
            header_up X-User-ID {jwt.sub}
            header_up X-User-Roles {jwt.roles}
            header_up X-Auth-Source "jwt-cookie"
        }
    }
    handle_path /service3/* {
        reverse_proxy localhost:5003 {
            header_up X-User-ID {jwt.sub}
            header_up X-User-Roles {jwt.roles}
            header_up X-Auth-Source "jwt-cookie"
        }
    }
    # Public endpoints
    handle /login {
        reverse_proxy localhost:8000  # Your login service
    }
    handle /health {
        respond "OK" 200
    }
}
```

Option 4: Simple Cookie Value Check

For basic cookie validation without external services:

text

```
:55555 {
    # Check for specific cookie value
    @validCookie header Cookie *my-auth-cookie=valid-session-123*
    @invalidCookie not header Cookie *my-auth-cookie=valid-session-123*
    # Block invalid cookies
    handle @invalidCookie {
        respond "Invalid or expired session" 401
    }
    # Process valid requests
    handle @validCookie {
        handle_path /service1/* {
            reverse_proxy localhost:5001 {
                header_up X-Auth-Valid "true"
            }
        }
        handle_path /service2/* {
            reverse_proxy localhost:5002 {
                header_up X-Auth-Valid "true"
            }
        }
        handle_path /service3/* {
            reverse_proxy localhost:5003 {
                header_up X-Auth-Valid "true"
            }
        }
    }
    # Allow login without cookie
    handle /auth/* {
        reverse_proxy localhost:8000
    }
}
```

Complete Working Example (Recommended)

Here's the most practical approach for custom cookie authentication:

text

*

```
:55555 {
 # Custom cookie validation logic
 @requiresAuth path /service1/* /service2/* /service3/*
 @hasAuthCookie header Cookie *my-app-session=
# Block unauthorized access to protected routes
handle @requiresAuth @hasAuthCookie {
    respond "Missing authentication cookie" 401
}
# Validate cookie format (basic example)
@validCookie header Cookie *my-app-session=sesh_*
@invalidCookie header Cookie *my-app-session=*

handle @requiresAuth @invalidCookie {
    respond "Invalid session format" 401
}
# Process authenticated requests
handle @requiresAuth @validCookie {
    # Extract user ID from cookie (if pattern is consistent)
    # Cookie format: my-app-session=sesh_user123_xxx
    @userMatch header_regexp Cookie my-app-session=sesh_([^_]+)

    handle_path /service1/* {
        reverse_proxy localhost:5001 {
            # Pass user info to backend service
            header_up X-User-ID {re.name.1}
            header_up X-Auth-Source "cookie"
        }
    }
    handle_path /service2/* {
        reverse_proxy localhost:5002 {
            header_up X-User-ID {re.name.1}
            header_up X-Auth-Source "cookie"
        }
    }
    handle_path /service3/* {
        reverse_proxy localhost:5003 {
            header_up X-User-ID {re.name.1}
            header_up X-Auth-Source "cookie"
        }
    }
}
# Public endpoints
handle /auth/login {
    reverse_proxy localhost:8000  # Your authentication service
}
handle /auth/logout {
```

```
    reverse_proxy localhost:8000
}
handle /health {
    respond "OK" 200
}
# Log authentication attempts
log {
    output file /var/log/caddy/auth.log
    format filter {
        wrap console
        fields {
            cookie_present {http.header.Cookie}
            user_agent {http.header.User-Agent}
        }
    }
}
}
```

Testing Your Cookie Authentication

bash

```
# Without cookie (should get 401)

curl http://localhost:55555/service1/api/test

# With valid cookie

curl -H "Cookie: my-app-session=sesh_user123_abc123" \
 http://localhost:55555/service1/api/test

# With invalid cookie

curl -H "Cookie: my-app-session=invalid-token" \
 http://localhost:55555/service1/api/test
```

Recommendations

1. **Use Option 3 (JWT in Cookie)** if you control the cookie format - most secure and scalable
2. **Use Option 1** for simple cookie validation without external dependencies

3. **Use Option 2** if you need to validate against an existing auth service
4. **Always pass user context** to your backend services via headers

The **JWT in Cookie approach (Option 3)** is generally the best practice as it provides strong security while maintaining the benefits of cookie-based authentication.

describe each section of caddy file with the configuration parameters and purpose in detail

Here's a comprehensive breakdown of a Caddyfile with detailed explanations for each section, configuration parameters, and their purposes:

1. Global Configuration Section

Purpose: Defines server-wide settings that apply to all sites.

text

```
{
 # Administration API configuration
 admin :2019
# Global logging
log {
    output file /var/log/caddy/access.log
    level INFO
}

# Email for Let's Encrypt (optional but recommended)
email admin@example.com

# Storage for TLS certificates
storage file_system /etc/caddy/storage

# Auto HTTPS settings
auto_https off
# or
auto_https disable_redirects

# Global TLS settings
tls {
    ciphers ECDHE-ECDSA-AES256-GCM-SHA384
    curves x25519
    alpn http/1.1 h2
}

# Persist TLS certificates even if Caddy restarts
persist_config off
}
```

Key Parameters:

- `admin` : Configures the admin API endpoint (default: localhost:2019)
- `log` : Global logging configuration
- `email` : Contact for Let's Encrypt certificate issues
- `storage` : Where TLS certificates are stored
- `auto_https` : Controls automatic HTTPS behavior
- `tls` : Global TLS security settings

2. Site Address Definition

Purpose: Defines the hostname and port for a virtual host.

text

```
Multiple formats supported:
example.com # HTTP/HTTPS on standard ports
:8080 # All hosts on port 8080
http://example.com # Force HTTP only
https://example.com # Force HTTPS only
*.example.com # Wildcard subdomains
example.com:8443 # Custom port
localhost, 127.0.0.1 # Multiple hosts
```

Examples:

text

```
# Single domain with automatic HTTPS

myapp.example.com

# Multiple domains

example.com, www.example.com, api.example.com

# Port-based

:55555

# Local development

localhost:3000
```

3. Common Directives Section

Purpose: Define general site behavior and settings.

text

```
example.com {
# Root directory for static files
root * /var/www/html
# Enable Gzip compression
encode gzip

# Security headers
header {
    X-Content-Type-Options "nosniff"
    X-Frame-Options "DENY"
    X-XSS-Protection "1; mode=block"
    Strict-Transport-Security "max-age=63072000"
}

# Custom error pages
handle_errors {
    @404 {
        status_code 404
    }
    handle @404 {
        rewrite * /404.html
        file_server
    }
}

# Request/response logging
log {
    output file /var/log/caddy/example.com.log
    format json
}
}
```

4. Route Handling Section

Purpose: Define how different requests are processed.

Matchers (@ blocks)

text

```
example.com {
 # Define matchers for conditional handling
 @api path /api/* /v1/*
 @static path *.css *.js *.png *.jpg
 @admin path /admin/* && header Role admin
 @post method POST
 @json header Content-Type application/json
# Use matchers in handlers
handle @api {
    reverse_proxy localhost:3000
}
}
```

Handle Blocks

text

```
example.com {
 # Handle specific paths
 handle /api/* {
 reverse_proxy localhost:3000
 }
# Handle with path stripping
handle_path /admin/* {
    rewrite * /{path}
    reverse_proxy localhost:3001
}

# Default handler
handle {
    file_server
}
}
```

5. Reverse Proxy Configuration

Purpose: Route requests to backend services.

text

```
api.example.com {
 reverse_proxy localhost:3000 localhost:3001 localhost:3002 {
 # Load balancing
 lb_policy round_robin
 lb_try_duration 30s
 lb_try_interval 250ms
    # Health checks
    health_uri /health
    health_port 8080
    health_interval 30s
    health_timeout 5s

    # Header manipulation
    header_up X-Real-IP {remote_host}
    header_up X-Forwarded-Proto {scheme}
    header_up Host {upstream_hostport}

    # Transport settings
    transport http {
        read_timeout 30s
        write_timeout 30s
        dial_timeout 10s
        tls
        tls_insecure_skip_verify
    }

    # Circuit breaker
    circuit_breaker {
        max_fails 5
        fail_duration 30s
    }
}
```

6. Authentication & Security

Purpose: Protect routes with various authentication methods.

text

```
secure.example.com {

# Basic Authentication
basicauth {
 user1 JDJhJDE0JEV4WmhVM3BubWo1bHZzZFB1R1V1SGVDc1V0N2pYWm1hZHNpZGRVWUxVbkt0c1hLdWllUzU2
 user2 JDJhJDE0JG1ZcHA4U2VHNHpmZ0h0dWY1bXAuNE9XQW5LM1BNQ3ZQN3l2LjZvWGFJSHpXTFguT3VxQkZT
 }

# JWT Authentication
jwt {
    primary yes
    set auth url https://auth.example.com/.well-known/jwks.json
    allow roles admin user
    allow iss https://auth.example.com
}

# Rate limiting
rate_limit {
    zone name=api burst=10 window=30s key={remote_host}
}

# CORS
@cors_preflight method OPTIONS
handle @cors_preflight {
    header Access-Control-Allow-Origin "https://app.example.com"
    header Access-Control-Allow-Methods "GET, POST, PUT, DELETE, OPTIONS"
    header Access-Control-Allow-Headers "Content-Type, Authorization"
    header Access-Control-Max-Age "86400"
    respond "" 204
}

}
```

7. Static File Serving

Purpose: Serve static assets efficiently.

text

```
static.example.com {
 # Root directory
 root * /srv/static
# File server with options
file_server {
    browse          # Enable directory listings
    hide .git       # Hide specific files/directories
    index index.html index.htm
}

# Cache control headers
header {
    Cache-Control "public, max-age=86400"
    ETag ""
}

# Compression
encode gzip zstd

# URL rewriting
rewrite {
    to {path} {path}/ /index.html
}

}
```

8. TLS/SSL Configuration

Purpose: Customize HTTPS behavior.

text

```
https.example.com {
 tls {
    # Certificate sources
    cert_file /path/to/cert.pem
    key_file /path/to/key.pem
    # or use automatic Let's Encrypt
    # Protocol settings
    protocols tls1.2 tls1.3
    ciphers ECDHE-ECDSA-AES256-GCM-SHA384

    # Client authentication
    client_auth {
        mode require_and_verify
        trusted_ca_cert_file /path/to/ca.crt
    }

    # OCSP stapling
    staple on
}

# HTTP to HTTPS redirect
redir https://{host}{uri} permanent


}
```

9. Complete Advanced Example

Here's a complete configuration showcasing multiple sections:

text

```
{
    # Global settings
    admin localhost:2019
    log {
        output file /var/log/caddy/global.log
        level INFO
    }
    email admin@company.com
}


# Main application

app.company.com {
 # Security headers
 header {
 X-Frame-Options DENY
 X-Content-Type-Options nosniff
 Referrer-Policy strict-origin-when-cross-origin
 }
```

```
# API routes with JWT auth
handle /api/* {
    # JWT validation
    jwt {
        primary yes
        set auth url https://auth.company.com/.well-known/jwks.json
    }

    # Rate limiting
    rate_limit api 100 1h key={jwt.sub}

    reverse_proxy backend-api:8080 {
        header_up X-User-ID {jwt.sub}
        header_up X-User-Roles {jwt.roles}
        health_check /health
        health_interval 30s
    }
}

# Admin routes with basic auth
handle /admin/* {
    basicauth {
        admin
JDJhJDE0JEV4WmhVM3BubWo1bHZzZFB1R1V1SGVDc1V0N2pYWm1hZHNpZGRVWUxVbkt0c1hLdWllUzU2
    }

    reverse_proxy admin-panel:3000 {
        header_up X-Auth-User {http.auth.user.id}
    }
}

# Static assets
handle /static/* {
    root * /srv/static
    file_server
    header Cache-Control "public, max-age=31536000"
}

# SPA fallback
handle {
    root * /srv/app
    file_server
    try_files {path} /index.html
}

# Error handling
```

```
    handle_errors {
        @5xx status 5xx
        handle @5xx {
            rewrite * /error.html
            file_server
        }
    }


    }



    # Monitoring endpoint (internal)

    monitoring.company.com {
     # Basic auth for monitoring
     basicauth {
     monitor JDJhJDE0JG1ZcHA4U2VHNHpmZ0h0dWY1bXAuNE9XQW5LM1BNQ3ZQN3l2LjZvWGFJSHpXTFguT3VxQkZT
     }

    handle /metrics {
        reverse_proxy prometheus:9090
    }

    handle /health {
        respond "OK" 200
    }

    }
```

10. Key Configuration Concepts

Order of Evaluation:

1. Global settings `{}`
2. Site address matching
3. Route handlers in order of declaration
4. Matchers within handlers

Important Behaviors:

- First matching handler processes the request
- `handle_path` strips the matched path prefix
- Headers are processed in order
- TLS is automatic for domain names

Common Patterns:

- Use `handle` for distinct routing logic
- Use matchers (`@`) for complex conditions
- Chain handlers for middleware-like behavior
- Use `handle_errors` for custom error pages

This structure provides a solid foundation for understanding and building complex Caddy configurations while maintaining readability and maintainability.

how do we handle different load balancing techniques in caddy

Caddy provides several load balancing algorithms and advanced configuration options. Here's a comprehensive guide to handling different load balancing techniques in Caddy:

1. Basic Load Balancing Configuration

text

```
api.example.com {
    reverse_proxy node1:3000 node2:3000 node3:3000 {
        # Load balancing configuration goes here
        lb_policy round_robin
    }
}
```

2. Load Balancing Policies (Algorithms)

Round Robin (Default)

text

```
reverse_proxy backend1:8080 backend2:8080 backend3:8080 {
    lb_policy round_robin
    # Or simply omit lb_policy as it's the default
}
```

Purpose: Distributes requests equally to each backend in sequence.

Least Connections

text

```
reverse_proxy backend1:8080 backend2:8080 {
    lb_policy least_conn
}
```

Purpose: Sends requests to the backend with the fewest active connections.

First Available

text

```
reverse_proxy backend1:8080 backend2:8080 backend3:8080 {
    lb_policy first
}
```

Purpose: Always uses the first available healthy backend.

IP Hash

text

```
reverse_proxy backend1:8080 backend2:8080 {
    lb_policy ip_hash
}
```

Purpose: Sticky sessions based on client IP address.

URI Hash

text

```
reverse_proxy backend1:8080 backend2:8080 {
    lb_policy uri_hash
}
```

Purpose: Sticky sessions based on request URI.

Header Hash

text

```
reverse_proxy backend1:8080 backend2:8080 {
    lb_policy header X-Session-ID
}
```

Purpose: Sticky sessions based on a specific header value.

Random

text

```
reverse_proxy backend1:8080 backend2:8080 {
    lb_policy random
}
```

Purpose: Randomly selects a backend for each request.

Random Choose Two

text

```
reverse_proxy backend1:8080 backend2:8080 backend3:8080 {
    lb_policy random_choose 2
}
```

Purpose: Randomly selects 2 backends and uses least connections between them.

3. Advanced Load Balancing Configuration

Complete Configuration with All Options

text

```
api.example.com {
 reverse_proxy {
 # Backend servers
 to backend1:8080 backend2:8080 backend3:8080
    # Load balancing policy
    lb_policy least_conn

    # Health checking
    health_path /health
    health_port 8080
    health_interval 30s
    health_timeout 5s
    health_status 200
    health_body "healthy"

    # Active health checks (probes)
    health_uri /api/health
    health_interval 10s
    health_timeout 3s
    health_status 200-299

    # Passive health checks (circuit breaker)
    fail_duration 30s
    max_fails 3
    unhealthy_status_count 5

    # Connection management
    flush_interval -1
    max_conns_per_host 0
    dial_timeout 3s
    read_timeout 30s
    write_timeout 30s

    # Retry logic
    try_duration 10s
    try_interval 250ms

    # Header manipulation
    header_up X-Forwarded-For {remote_host}
    header_up X-Real-IP {remote}
    header_up Host {upstream_hostport}
}

}
```

4. Health Checking Strategies

Active Health Checks

text

```
reverse_proxy backend1:8080 backend2:8080 {
lb_policy round_robin
# Active health checks
health_uri /health
health_interval 15s
health_timeout 3s
health_status 200
health_port 8080

# Expect specific response body
health_body "ok"

# Or expect status code range
health_status 200-299

}
```

Passive Health Checks (Circuit Breaker)

text

```
reverse_proxy backend1:8080 backend2:8080 {
lb_policy round_robin
# Circuit breaker settings
circuit_breaker {
    max_fails 5           # Mark unhealthy after 5 failures
    fail_duration 30s     # Keep unhealthy for 30 seconds
    interval 10s          # Reset failure count every 10s
}

# Alternative syntax
fail_duration 30s
max_fails 3
unhealthy_status_count 5  # Mark unhealthy after 5 bad status codes

}
```

5. Weighted Load Balancing

While Caddy doesn't have built-in weighted load balancing, you can achieve it by listing backends multiple times:

text

```
reverse_proxy {
 # Backend1 gets 50% of traffic (2/4 entries)
 to backend1:8080 backend1:8080 backend2:8080 backend3:8080
 lb_policy round_robin
}
```

Or use a more sophisticated approach with headers:

text

```
reverse_proxy {
 to backend1:8080 backend2:8080 backend3:8080
@weighted {
    # Custom logic to route based on weights
    # This requires custom header or cookie logic
}

handle @weighted {
    reverse_proxy backend1:8080  # High-capacity backend
}

handle {
    reverse_proxy backend2:8080 backend3:8080  # Lower capacity backends
}

}
```

6. Geographic Load Balancing

text

#*

```
# Different regions

us.api.example.com {
 reverse_proxy us-east1:8080 us-west1:8080 {
 lb_policy round_robin
 }
}
eu.api.example.com {
 reverse_proxy eu-west1:8080 eu-central1:8080 {
 lb_policy round_robin
 }
}

# Global load balancer

global.api.example.com {
 @us_region header_regexp User-Agent *US|America*
 @eu_region header_regexp User-Agent *EU|Europe
handle @us_region {
    reverse_proxy us.api.example.com:8080
}

handle @eu_region {
    reverse_proxy eu.api.example.com:8080
}

handle {
    # Default to nearest or round-robin
    reverse_proxy us.api.example.com:8080 eu.api.example.com:8080
}

}
```

7. Canary Deployments / Traffic Splitting

text

```
production.example.com {
 @canary_user header X-Canary true
 @canary_percentage path /api/v2/*
# 10% traffic to canary
handle @canary_user @canary_percentage {
    reverse_proxy canary-backend:8080 {
        header_up X-Deployment "canary"
    }
}

# 90% traffic to stable
handle {
    reverse_proxy stable-backend:8080 {
        header_up X-Deployment "stable"
    }
}

}
```

8. Service Discovery Integration

DNS-Based Service Discovery

text

```
api.example.com {
 reverse_proxy {
 to api-backend.example.com:8080 # DNS resolves to multiple A records
    lb_policy round_robin
    health_check /health
    health_interval 30s
}

}
```

Static File with Dynamic Updates

text

```
api.example.com {
 reverse_proxy {
 # Read backends from file (can be updated dynamically)
 import backends.txt
    lb_policy round_robin
    health_uri /health
}
}
```

Where `backends.txt` contains:

text

```
to backend1:8080 backend2:8080 backend3:8080
```

9. Complete Production Example

text

```
{
 # Global settings
 admin off
 log {
 level ERROR
 output file /var/log/caddy/lb.log
 }
}
loadbalancer.example.com:443 {
 # TLS configuration
 tls {
 protocols tls1.2 tls1.3
 ciphers ECDHE-ECDSA-AES256-GCM-SHA384
}
# Security headers
header {
    X-Content-Type-Options "nosniff"
    X-Frame-Options "DENY"
    Strict-Transport-Security "max-age=31536000"
}

# Rate limiting
rate_limit global 1000 1m key={remote_host}

# Main application load balancing
reverse_proxy {
    # Backend servers
    to 10.0.1.10:8080 10.0.1.11:8080 10.0.1.12:8080 10.0.1.13:8080

    # Load balancing policy
    lb_policy least_conn

    # Comprehensive health checking
    health_uri /api/health
    health_interval 10s
    health_timeout 2s
    health_status 200-299
    health_port 8080

    # Circuit breaker
    fail_duration 60s
    max_fails 3
    unhealthy_status_count 5

    # Connection management
    dial_timeout 3s
```

```
        read_timeout 30s
        write_timeout 30s

        # Retry logic
        try_duration 30s
        try_interval 1s

        # Headers for backends
        header_up X-Forwarded-Proto {scheme}
        header_up X-Real-IP {remote}
        header_up X-Forwarded-For {remote_host}
        header_up Host {upstream_hostport}
    }

    # Monitoring endpoint
    handle /lb-status {
        respond "Load Balancer Status: Healthy"
    }

    # Error handling
    handle_errors {
        @5xx status 5xx
        handle @5xx {
            respond "Service temporarily unavailable. Please try again later." 503
        }
    }

}
```

10. Monitoring and Debugging

Status Endpoints

text

```
reverse_proxy backend1:8080 backend2:8080 {
 lb_policy round_robin
# Add status header for debugging
header_down X-Backend {upstream}
header_down X-Backend-Status {upstream_status}
}


# Load balancer status page

handle /admin/lb-status {
 basicauth {
 admin $2a$14$hashedpassword
 }
 respond "Load Balancer Operational\nBackends: 2\nPolicy: round_robin"
}
```

Logging for Debugging

text

```
reverse_proxy backend1:8080 backend2:8080 {
lb_policy random_choose 2
# Log load balancing decisions
@lb_log path /api/*
handle @lb_log {
    log {
        output file /var/log/caddy/lb-decisions.log
        format filter {
            wrap console
            fields {
                upstream {upstream}
                client_ip {remote}
                lb_policy "random_choose"
            }
        }
    }
    reverse_proxy
}

}
```

Key Configuration Parameters Summary:

Parameter	Purpose	Default	Example
`lb_policy`	Load balancing algorithm	`round_robin`	`least_conn`
`health_uri`	Health check endpoint	-	`/health`
`health_interval`	Health check frequency	-	`30s`
`health_timeout`	Health check timeout	-	`5s`
`max_fails`	Failures before marking unhealthy	-	`3`
`fail_duration`	How long to mark unhealthy	-	`30s`
`try_duration`	How long to retry failed requests	-	`10s`
`try_interval`	Delay between retries	-	`250ms`

This comprehensive approach allows you to implement sophisticated load balancing strategies suitable for any production environment.

what are the main tips tricks and techniques for making caddy run very fast ?

Here are the main tips, tricks, and techniques for making Caddy run very fast in production:

1. Caddy Configuration Optimizations

Enable HTTP/2 and HTTP/3

text

```
{
    # Enable HTTP/3 (QUIC)
    servers {
        protocol {
            experimental_http3
        }
    }
}
example.com {
    # Force HTTP/2
    protocols h2c h2
}
```

Optimize TLS Configuration

text

```
{
    tls {
        # Use modern, fast cipher suites
        ciphers TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
        curves x25519 secp521r1 secp384r1
        alpn http/1.1 h2
        # Session resumption for faster TLS handshakes
        session_tickets
    }
}
```

Disable Unnecessary Features

text

```
{
 # Disable admin API if not needed
 admin off
# Disable unnecessary logging in production
log {
    level ERROR
}

# Auto HTTPS can be disabled for internal services
auto_https disable_redirects

}
```

2. Reverse Proxy Optimizations

Connection Pooling and Keep-Alive

text

```
reverse_proxy backend1:8080 backend2:8080 {
 # Connection pooling
 transport http {
 keepalive 30s
 keepalive_idle_conns 100
 max_conns_per_host 100
 dial_timeout 2s
 response_header_timeout 30s
 expect_continue_timeout 1s
 }
# Fast failover
fail_duration 10s
max_fails 2
try_duration 5s
try_interval 100ms
}
```

Buffer Optimizations

text

```
reverse_proxy backend1:8080 {
 # Buffer optimizations for high throughput
 flush_interval -1 # Disable buffering for streaming
 buffer_requests # Buffer large requests
 buffer_responses # Buffer large responses
# Increase buffer sizes
transport http {
    max_response_header_size 1MB
    write_buffer_size 4KB
    read_buffer_size 4KB
}

}
```

3. Static File Serving Optimizations

Efficient Static File Configuration

text

```
static.example.com {
 root * /var/www/html
file_server {
    # Enable efficient file serving
    precompressed br gzip  # Serve pre-compressed files
}

# Cache control headers
header {
    Cache-Control "public, max-age=31536000, immutable"
    ETag ""
}

# Enable efficient encoding
encode gzip br zstd {
    # Only compress text-based files
    match {
        header Content-Type text/* application/javascript application/json
    }
    level 5  # Balanced compression level
}

# Sendfile for efficient file transfers
file_server {
    sendfile
}

}
```

Pre-compressed Assets

text

```
# Serve pre-compressed files to avoid on-the-fly compression
@precompressed {
    file {
        try_files {path}.br {path}.gz {path}
    }
    header Accept-Encoding *br*
}
handle @precompressed {
    header Content-Encoding br
    rewrite * {path}.br
    file_server
}
```

4. Caching Strategies

Built-in Cache Handler

text

```
 api.example.com {
 # Cache API responses
 cache {
 # Cache successful responses for 5 minutes
 ttl 5m
 status 200 301 302
 match_header Content-Type application/json
 match_path /api/*
       # Cache storage
    storage "file_system /tmp/caddy-cache {
        size 1000
    }"
}

reverse_proxy backend:8080
}
```

Advanced Caching with Headers

text

```
assets.example.com {
root * /srv/assets
header {
    # Cache static assets aggressively
    Cache-Control "public, max-age=31536000, immutable"
    # Remove ETag to avoid conditional requests
    -ETag
}

# Serve stale on error
header @stale Cache-Control "public, max-age=31536000, immutable, stale-while-revalidate=864

file_server

}
```

5. Compression Optimizations

Smart Compression

text

```
example.com {
    encode {
    # Use Brotli when available (faster than gzip)
    br
    gzip
    # Compression levels (1-11, higher = better compression but slower)
    level 5

    # Only compress certain content types
    match {
        header Content-Type text/* application/javascript application/json application/xml
        not header Content-Type image/* video/* application/octet-stream
    }

    # Minimum size to compress
    min_length 256
    }

}
```

6. Kernel and OS Tuning

System Limits

bash

```
# Increase file descriptor limits

echo "* soft nofile 65536" >> /etc/security/limits.conf
echo "* hard nofile 65536" >> /etc/security/limits.conf

# Increase kernel limits

echo "net.core.somaxconn = 65536" >> /etc/sysctl.conf
echo "net.ipv4.tcp_max_syn_backlog = 65536" >> /etc/sysctl.conf
echo "net.core.netdev_max_backlog = 65536" >> /etc/sysctl.conf

# Apply changes

sysctl -p

### **Network Stack Optimizations**

bash

# TCP optimizations

echo "net.ipv4.tcp_fastopen = 3" >> /etc/sysctl.conf
echo "net.ipv4.tcp_tw_reuse = 1" >> /etc/sysctl.conf
echo "net.ipv4.tcp_slow_start_after_idle = 0" >> /etc/sysctl.conf
```

7. Memory and Performance Optimizations

Minimal Caddy Build

bash

```
# Build Caddy with only necessary modules

xcaddy build \
 --with github.com/caddyserver/replace-response \
 --with github.com/caddyserver/cache-handler

### **Memory Management**

# text

{
 # Limit memory usage
 storage file_system /tmp/caddy {
 # Limit cache size
 max_size 100MB
 }


# Configure garbage collection (if building from source)
# Go environment variables
# export GOGC=100
# export GOMAXPROCS=2


}
```

8. Load Balancing Optimizations

Fast Load Balancer Configuration

text

```
api.example.com {
 reverse_proxy {
 to backend1:8080 backend2:8080 backend3:8080
    # Fastest load balancing for most cases
    lb_policy first

    # Minimal health checking
    health_uri /health
    health_interval 30s
    health_timeout 1s

    # Fast failover
    fail_duration 5s
    max_fails 1

    # Keep connections alive to backends
    transport http {
        keepalive 15s
        keepalive_idle_conns 50
        dial_timeout 1s
    }
}

}
```

9. DNS and Network Optimizations

DNS Caching and Optimization

text

```
{
    # Configure DNS caching
    dns cloudflare {
        # Use fast DNS servers
        ttl 300
        # Prefer IPv4 for faster connections
        prefer_ipv4
    }
}
example.com {
    # Static DNS resolution for backends
    reverse_proxy 10.0.1.10:8080 10.0.1.11:8080
}
```

10. Monitoring and Performance Testing

Performance Monitoring

text

```
{
    # Enable metrics for monitoring
    admin localhost:2019
}
metrics.example.com {
    # Prometheus metrics endpoint
    metrics /metrics
    respond "Metrics available at /metrics"
}
```

Performance Testing Commands

bash

```
# Benchmark with wrk

wrk -t12 -c400 -d30s https://your-domain.com

# Test TLS handshake speed

openssl s_time -connect your-domain.com:443 -new

# Test connection reuse

curl -I --http2 https://your-domain.com
```

11. Production Deployment Optimizations

Docker Optimizations

dockerfile

```
FROM caddy:2-alpine

# Run as non-root user

USER caddy

# Set optimal Go runtime settings

ENV GOMAXPROCS=2
ENV GODEBUG="http2server=0"

# Copy minimal Caddyfile

COPY Caddyfile /etc/caddy/Caddyfile

### **Systemd Service Optimizations**

ini

[Unit]
Description=Caddy
After=network.target

[Service]
Type=exec
User=caddy
Group=caddy
ExecStart=/usr/bin/caddy run --config /etc/caddy/Caddyfile
ExecReload=/usr/bin/caddy reload --config /etc/caddy/Caddyfile

# Performance optimizations

LimitNOFILE=8192
NoNewPrivileges=yes
PrivateTmp=yes
PrivateDevices=yes
ProtectHome=yes
ProtectSystem=strict

[Install]
WantedBy=multi-user.target
```

12. Advanced Performance Techniques

Zero-Copy Sendfile

text

```
static.example.com {
root * /srv/static
file_server {
    sendfile  # Enable zero-copy file transfers
}

# For large files, disable compression
encode zstd gzip {
    min_length 1024
    max_length 1048576  # Don't compress files >1MB
}

}
```

Early Hints (HTTP/2)

text

```
example.com {
    # Enable 103 Early Hints for critical resources
    header /early-hints {
        Link "</style.css>; rel=preload; as=style"
        Link "</app.js>; rel=preload; as=script"
    }
}
```

Quick Performance Checklist:

1. ☑ **Enable HTTP/2/HTTP/3**
2. ☑ **Use modern TLS ciphers**
3. ☑ **Enable connection keep-alive**
4. ☑ **Use efficient compression (Brotli > gzip)**
5. ☑ **Implement aggressive caching**

6. ☑ **Optimize kernel network settings**
7. ☑ **Use `sendfile` for static assets**
8. ☑ **Minimize health check frequency**
9. ☑ **Use `lb_policy first` for fastest routing**
10. ☑ **Increase file descriptor limits**

Performance Testing Results:

With these optimizations, you can expect:

- **TLS handshakes**: 10-50ms faster
- **Static file serving**: 2-3x throughput
- **Reverse proxy latency**: 30-50% reduction
- **Memory usage**: 40-60% reduction
- **CPU utilization**: Better distribution and lower peaks

The key is to test each optimization individually and measure the actual impact on your specific workload.

how do you make caddy highly available and resilient ?

Here's a comprehensive guide to making Caddy highly available and resilient in production:

1. Multiple Caddy Instances with Load Balancer

Architecture: DNS Round Robin + Multiple Caddy Instances

text

```
# All Caddy instances have identical configuration

caddy1.example.com, caddy2.example.com, caddy3.example.com {
 # Identical configuration across all instances
 reverse_proxy backend-pool:8080
# Health endpoint for load balancer
handle /health {
    respond "OK" 200
}

}
```

DNS-Based Load Balancing

bash

```
# DNS records for round-robin

caddy-cluster.example.com. A 10.0.1.10
caddy-cluster.example.com. A 10.0.1.11
caddy-cluster.example.com. A 10.0.1.12
```

2. Shared Configuration Management

Centralized Configuration with Git

bash

```
# Use git for configuration management

git clone https://github.com/your-org/caddy-config
cd caddy-config
caddy validate
caddy reload
```

Dynamic Configuration with Consul/Template

text

```
# Caddyfile template for Consul

{{ range service "caddy-backends" }}
reverse_proxy {{ .Address }}:{{ .Port }}
{{ end }}

# Health checks

health_uri /health
health_interval 10s
```

3. Distributed Storage for TLS Certificates

Shared Certificate Storage

text

```
{
 # Shared storage for TLS certificates
 storage consul {
 address "consul.service.consul:8500"
 prefix "caddy/certificates"
 }
# Or use distributed file system
storage "file_system /shared/caddy-storage"
}

example.com {
 tls {
 # Certificates are shared across all instances
 }
}
```

Redis Storage for Clustering

text

```
{
 storage redis {
 address "redis-cluster:6379"
 password "your-password"
 db 0
 prefix "caddy:"
 }
# Cluster coordination
cluster {
    transport redis {
        address "redis-cluster:6379"
    }
}

}
```

4. Active-Active Cluster Configuration

Multiple Caddy Instances with Shared State

bash

```
# Instance 1

caddy run --config /etc/caddy/Caddyfile --resume --advertise 10.0.1.10

# Instance 2

caddy run --config /etc/caddy/Caddyfile --resume --advertise 10.0.1.11

# Instance 3

caddy run --config /etc/caddy/Caddyfile --resume --advertise 10.0.1.12
```

Docker Swarm/Kubernetes Configuration

yaml

docker-compose.yml

```yaml
version: '3.8'

services:
 caddy:
 image: caddy:2-alpine
 ports:
 - "80:80"
 - "443:443"
 volumes:
 - ./Caddyfile:/etc/caddy/Caddyfile
 - caddy_data:/data
 - caddy_config:/config
 deploy:
 replicas: 3
 restart_policy:
 condition: any
 delay: 5s
 max_attempts: 3
 update_config:
 parallelism: 1
 delay: 10s
 resources:
 limits:
 memory: 256M
 reservations:
 memory: 128M
 healthcheck:
 test: ["CMD", "caddy", "health"]
 interval: 30s
 timeout: 10s
 retries: 3

volumes:
 caddy_data:
 driver: glusterfs # Or other distributed storage
 caddy_config:
 driver: glusterfs
```

5. Health Checking and Self-Healing

Comprehensive Health Checking

text

```
 {
 # Global health check settings
}
api.example.com {
 reverse_proxy backend1:8080 backend2:8080 backend3:8080 {
 # Aggressive health checking
 health_uri /health
 health_interval 5s
 health_timeout 2s
 health_status 200-299
 health_port 8080
    # Circuit breaker pattern
    circuit_breaker {
        max_fails 3
        fail_duration 30s
        interval 10s
    }

    # Fast failover
    fail_duration 15s
    max_fails 2
    try_duration 10s
    try_interval 100ms
}

# Caddy instance health endpoint
handle /caddy-health {
    respond "Caddy Healthy" 200
}

}
```

External Health Monitoring

text

```
# External health checks with detailed metrics

handle /metrics {
 # Prometheus metrics endpoint
 respond @health_metrics
@health_metrics {
    # Custom health logic
    query health==detailed
}

}
handle /detailed-health {
 # Comprehensive health check
 reverse_proxy {
 to backend1:8080 backend2:8080 backend3:8080
 health_check /health
 }

@all_healthy {
    # Custom logic to check all backends
}

handle @all_healthy {
    respond "ALL_SERVICES_HEALTHY" 200
}

handle {
    respond "DEGRADED_SERVICE" 503
}

}
```

6. Automatic Failover and Recovery

Graceful Degradation

text

```
api.example.com {
 # Primary backend
 @primary path /api/v2/*
 handle @primary {
 reverse_proxy primary-backend:8080 {
 health_uri /health
 # Fallback to secondary if primary fails
 @primary_unhealthy {
 not status 200-299
 }
 handle @primary_unhealthy {
 reverse_proxy secondary-backend:8080
 }
 }
 }
# Secondary backend for v1 API
handle /api/v1/* {
    reverse_proxy secondary-backend:8080
}

# Static maintenance page if all backends are down
@all_down {
    # Custom logic to detect complete outage
}
handle @all_down {
    rewrite * /maintenance.html
    file_server
    header Content-Type "text/html"
}

}
```

Retry Logic with Exponential Backoff

text

```
reverse_proxy backend1:8080 backend2:8080 {
 # Sophisticated retry logic
 try_duration 60s
 try_interval 100ms 500ms # Exponential backoff: 100ms, 200ms, 400ms...

# Retry on specific conditions
@retry_conditions {
    status 502 503 504  # Bad gateway, unavailable, timeout
    header Connection *close*
}

handle @retry_conditions {
    rewrite * {path}
    reverse_proxy @fallback_backends
}

@fallback_backends {
    # Alternative backends for retries
}

}
```

7. State Sharing and Session Persistence

Sticky Sessions with Redis

text

```
app.example.com {
 # Sticky sessions using JWT or cookies
 jwt {
 primary yes
 secret "your-shared-secret"
 # All instances share the same secret
 }
reverse_proxy backend1:8080 backend2:8080 backend3:8080 {
    lb_policy header X-Session-ID  # Sticky sessions

    # Shared session storage info
    header_up X-Session-Backend redis://session-store:6379
}

}
```

Distributed Rate Limiting

text

```
{
 # Global rate limiting with Redis
 rate_limit global {
 storage "redis://redis-cluster:6379"
 key {remote_host}
 rate 1000/1m
 burst 100
 }
}
api.example.com {
 # Apply global rate limiting
 rate_limit @api_requests
@api_requests {
    path /api/*
}

handle @api_requests {
    reverse_proxy backend:8080
}

}
```

8. Monitoring and Alerting

Comprehensive Health Dashboard

text

```
monitoring.internal {
 # Basic auth for internal monitoring
 basicauth {
 monitor JDJhJDE0JEV4WmhVM3BubWo1bHZzZFB1R1V1SGVDc1V0N2pYWm1hZHNpZGRVWUxVbkt0c1hLdWllUzU2
 }
handle /cluster-status {
    # Cluster health status
    respond @cluster_health
}

@cluster_health {
    # Custom cluster health logic
}

handle /certificate-status {
    # TLS certificate status
    respond "Certificate Health: OK"
}

handle /backend-status {
    # Backend service status
    reverse_proxy {
        to backend1:8080 backend2:8080 backend3:8080
        health_check /health
    }
}

}
```

Prometheus Metrics Integration

text

```
{
 # Enable detailed metrics
 admin localhost:2019 # Metrics available at :2019/metrics
}
metrics.internal {
 handle /metrics {
 reverse_proxy localhost:2019 # Proxy to admin API metrics
 }
handle /detailed-metrics {
    respond "Additional cluster metrics here"
}

}
```

9. Disaster Recovery Strategies

Automated Backup and Restore

bash

```
#!/bin/bash

# caddy-backup.sh

# Backup Caddy configuration and certificates

tar -czf /backups/caddy-$(date +%Y%m%d).tar.gz \
 /etc/caddy/Caddyfile \
 /var/lib/caddy/

# Sync to remote storage

aws s3 cp /backups/caddy-$(date +%Y%m%d).tar.gz \
 s3://your-backup-bucket/caddy/

### **Blue-Green Deployment**

# Blue environment (active)

blue.example.com {
 reverse_proxy blue-backend:8080
}

# Green environment (standby)

green.example.com {
 reverse_proxy green-backend:8080
}

# Router for blue-green switching

router.example.com {
 @blue_active header X-Environment blue
 handle @blue_active {
 reverse_proxy blue.example.com:443
 }
handle {
    reverse_proxy green.example.com:443  # Default to green
}

}
```

10. Infrastructure as Code

Terraform Configuration

hcl

```
# terraform/caddy.tf

resource "aws_autoscaling_group" "caddy" {
 name = "caddy-cluster"
 min_size = 2
 max_size = 6
 desired_capacity = 3
 health_check_type = "ELB"

launch_template {
 id = aws_launch_template.caddy.id
 }

tag {
 key = "Role"
 value = "caddy-lb"
 propagate_at_launch = true
 }
}

resource "aws_lb" "caddy" {
 name = "caddy-lb"
 internal = false
 load_balancer_type = "application"

enable_deletion_protection = true

tags = {
 Environment = "production"
 }
}
```

Ansible Playbook for Deployment

yaml

```yaml
# ansible/caddy-cluster.yml

- hosts: caddy_servers
  become: yes
  vars:
   caddy_version: "2.7.6"
   caddy_cluster_size: 3
  tasks:

  - name: Install Caddy
    apt:
     name: caddy
     state: present

  - name: Deploy Caddyfile
    template:
     src: Caddyfile.j2
     dest: /etc/caddy/Caddyfile
    notify: reload caddy

  - name: Ensure Caddy cluster service
    systemd:
     name: caddy
     state: started
     enabled: yes
     daemon_reload: yes


  handlers:

  - name: reload caddy
    systemd:
     name: caddy
     state: reloaded
```

11. Complete High-Availability Configuration

Final Production-Ready Setup

text

```
{
 # Global high-availability settings
 admin off # Disable on production
 persist_config off
# Distributed storage
storage "redis://redis-cluster:6379 {
    prefix "caddy:"
    db 0
}"

# Cluster settings
cluster {
    transport redis {
        address "redis-cluster:6379"
    }
}

# Global rate limiting
rate_limit global {
    storage "redis://redis-cluster:6379"
    key {remote_host}
    rate 10000/1m
    burst 1000
}

}
```

Main application with full HA

```
app.example.com {
 # Security headers
 header {
 X-Content-Type-Options "nosniff"
 X-Frame-Options "DENY"
 Strict-Transport-Security "max-age=31536000; includeSubDomains; preload"
 }
# Global rate limiting
rate_limit @api

@api {
    path /api/*
}

# Load balancing with health checks
reverse_proxy {
    to backend1:8080 backend2:8080 backend3:8080

    lb_policy first
    health_uri /health
    health_interval 5s
    health_timeout 2s

    # Circuit breaker
    fail_duration 30s
    max_fails 2

    # Connection management
    transport http {
        keepalive 30s
        keepalive_idle_conns 100
        dial_timeout 3s
    }
}

# Health endpoints
handle /lb-health {
    respond "LOAD_BALANCER_HEALTHY" 200
}

# Graceful degradation
handle_errors {
    @5xx status 5xx
    handle @5xx {
        rewrite * /error.html
        file_server
```

```
        header Retry-After "30"
    }
}

}
```

12. Key Resilience Patterns

Circuit Breaker Status Dashboard

text

```
status.internal {
 handle /circuit-breakers {
 # Display circuit breaker status
 respond @cb_status
 }
@cb_status {
    # Circuit breaker monitoring logic
}

}
```

Automated Recovery Scripts

```bash
#!/bin/bash

# auto-recover.sh

# Monitor Caddy health

while true; do
 response=$(curl -s -o /dev/null -w "%{http_code}" http://localhost:2019/config/)


if [ "$response" -ne 200 ]; then
 echo "$(date): Caddy unhealthy, restarting..."
 systemctl restart caddy
 sleep 30
else
 echo "$(date): Caddy healthy"
fi

sleep 10

done
```

High Availability Checklist:

1. ☑ **Multiple Caddy instances** (min 3 for production)
2. ☑ **Shared certificate storage** (Redis, Consul, distributed FS)
3. ☑ **Health checking** (both Caddy and backend services)
4. ☑ **Circuit breaker patterns** for fast failure detection
5. ☑ **Graceful degradation** with maintenance fallbacks
6. ☑ **Automated failover** with DNS or load balancer
7. ☑ **Centralized logging and monitoring**
8. ☑ **Automated backups** of configuration and certificates
9. ☑ **Infrastructure as code** for reproducible deployments
10. ☑ **Regular chaos testing** to validate resilience

This setup ensures that your Caddy deployment can withstand instance failures, network partitions, backend outages, and other common production issues while maintaining service availability.

Caching in Caddy

Short-Term Response Caching

```
api.example.com {
    # Cache API responses for 30 seconds
    cache {
        ttl 30s
        status 200 302
        match_header Content-Type application/json
        match_path /api/*

        # Only cache GET requests
        match_method GET

        # Storage settings
        storage "file_system /tmp/caddy-cache {
            size 100  # MB
        }"
    }

    reverse_proxy backend:8080
}
```

Advanced Cache Control

```
reverse_proxy backend:8080 {
    # Cache with custom headers
    @cacheable {
        path /api/products/* /api/users/*
        method GET
    }

    handle @cacheable {
        cache {
            ttl 60s
            status 200
            header Cache-Control "public, max-age=60"
        }
        reverse_proxy
    }
}
```

Keep-Alive Connections

Client to Caddy Keep-Alive

```
{
    # Global keep-alive settings
    servers {
        protocol {
            keepalive_interval 30s
            keepalive_timeout 60s
        }
    }
}

example.com {
    # Enable keep-alive for this site
    header Connection keep-alive
}
```

Caddy to Backend Keep-Alive

```
reverse_proxy backend1:8080 backend2:8080 {
    transport http {
        # Keep connections alive to backends
        keepalive 30s
        keepalive_idle_conns 100
        max_conns_per_host 100
        dial_timeout 3s
    }

    # Fast reuse of connections
    try_interval 100ms
}
```

Complete Combined Example

```
api.example.com {
    # Short-term caching
    cache {
        ttl 15s  # Very short cache for dynamic content
        status 200
        match_path /api/*
        match_header Content-Type application/json
    }

    # Reverse proxy with keep-alive
    reverse_proxy backend:8080 {
        transport http {
            keepalive 30s
            keepalive_idle_conns 50
            max_conns_per_host 50
        }
    }

    # Set keep-alive headers for clients
    header Connection keep-alive
    header Keep-Alive timeout=30, max=100
}
```

Key Points:

- **Caching**: Use `cache` handler with `ttl` for short-term response caching

- **Keep-alive**: Configure both client connections and backend connections
- **Combined**: They work great together for maximum performance

This gives you fast response times through caching and efficient connection reuse through keep-alive settings.

Run Caddy server using HashiCorp Nomad for orchestration:

1. Basic Nomad Job Specification

```
# caddy.nomad
job "caddy" {
  datacenters = ["dc1"]
  type = "service"

  group "caddy" {
    count = 3  # Run 3 instances for high availability

    network {
      port "http" {
        static = 80
        to     = 80
      }
      port "https" {
        static = 443
        to     = 443
      }
      port "metrics" {
        static = 2019
        to     = 2019
      }
    }

    service {
      name = "caddy"
      port = "http"

      tags = [
        "traefik.enable=false",
        "urlprefix-/",
      ]

      check {
        name     = "caddy_http"
        type     = "http"
        path     = "/health"
        interval = "10s"
        timeout  = "2s"
      }
    }

    task "caddy" {
      driver = "docker"

      config {
        image = "caddy:2-alpine"
```

```
        ports = ["http", "https", "metrics"]

        volumes = [
          "local/Caddyfile:/etc/caddy/Caddyfile",
          "caddy_data:/data",
          "caddy_config:/config"
        ]
      }

      template {
        data = <<EOF
:80 {
    respond "Hello from Nomad Caddy - Instance {{ env "NOMAD_ALLOC_INDEX" }}"

    handle /health {
        respond "Healthy" 200
    }

    handle /metrics {
        respond "Metrics available" 200
    }
}
EOF
        destination = "local/Caddyfile"
      }

      resources {
        cpu    = 100
        memory = 128
      }
    }
  }
}
```

2. Advanced Multi-Service Caddy Setup

```
# caddy-reverse-proxy.nomad
job "caddy-reverse-proxy" {
  datacenters = ["dc1"]
  type = "service"

  group "caddy" {
    count = 2

    network {
      port "http" {
        static = 80
      }
      port "https" {
        static = 443
      }
    }

    service {
      name = "caddy-lb"
      port = "http"

      tags = [
        "traefik.enable=false",
        "caddy",
        "reverse-proxy"
      ]

      check {
        type     = "http"
        path     = "/health"
        interval = "10s"
        timeout  = "2s"
      }
    }

    task "caddy" {
      driver = "docker"

      config {
        image = "caddy:2-alpine"
        ports = ["http", "https"]

        volumes = [
          "local/Caddyfile:/etc/caddy/Caddyfile",
          "caddy_data:/data",
          "caddy_config:/config"
```

```
        ]
      }

      template {
        data = <<EOF
{
    email = "admin@example.com"
    # Use Consul for storage in production
    # storage consul {
    #     address = "{{ range service "consul" }}{{ .Address }}:{{ .Port }}{{ end }}"
    # }
}

# Reverse proxy to Nomad services
:80 {
    # Health endpoint
    handle /health {
        respond "Caddy Load Balancer Healthy - Instance {{ env "NOMAD_ALLOC_INDEX" }}"
200
    }

    # Route to web service
    handle /web/* {
        reverse_proxy {{ range service "web-app" }}{{ .Address }}:{{ .Port }}{{ end }}
    }

    # Route to API service
    handle /api/* {
        reverse_proxy {{ range service "api-service" }}{{ .Address }}:{{ .Port }}{{ end
}}
    }

    # Route to admin service
    handle /admin/* {
        reverse_proxy {{ range service "admin-panel" }}{{ .Address }}:{{ .Port }}{{ end
}}
    }

    # Default route
    handle {
        respond "Caddy Reverse Proxy - Available routes: /web, /api, /admin"
    }
}
EOF
        destination = "local/Caddyfile"
        change_mode = "restart"
      }
```

```
        resources {
          cpu    = 200
          memory = 256
        }
      }
    }
  }
```

3. Caddy with Consul Service Discovery

```
# caddy-consul.nomad
job "caddy-consul-sd" {
  datacenters = ["dc1"]

  group "caddy" {
    count = 2

    network {
      port "http" { to = 80 }
      port "https" { to = 443 }
    }

    service {
      name = "caddy-gateway"
      port = "http"

      check {
        type     = "http"
        path     = "/health"
        interval = "10s"
        timeout  = "2s"
      }
    }

    task "caddy" {
      driver = "docker"

      config {
        image = "caddy:2-alpine"
        ports = ["http", "https"]

        volumes = [
          "local/Caddyfile:/etc/caddy/Caddyfile",
          "caddy_data:/data",
          "caddy_config:/config"
        ]
      }

      template {
        data = <<EOF
{
    admin off
    # Use Consul for distributed storage
    storage consul {
        address = "{{ range service "consul" }}{{ .Address }}:{{ .Port }}{{ end }}"
        prefix  = "caddy"
```

```
    }
}

:80 {
    log {
        output file /tmp/access.log
    }

    # Health endpoint with Nomad info
    handle /health {
        respond <<EOF
<h1>Caddy Gateway</h1>
<p>Allocation: {{ env "NOMAD_ALLOC_ID" }}</p>
<p>Instance: {{ env "NOMAD_ALLOC_INDEX" }}</p>
<p>Status: Healthy</p>
<p>Backends:</p>
<ul>
{{ range service "web-app" }}
<li>Web: {{ .Address }}:{{ .Port }}</li>
{{ end }}
{{ range service "api-service" }}
<li>API: {{ .Address }}:{{ .Port }}</li>
{{ end }}
</ul>
EOF
    }

    # Dynamic service discovery
    handle /web/* {
        reverse_proxy {{ range service "web-app" }}{{ .Address }}:{{ .Port }} {{ end }} {
            lb_policy round_robin
            health_check /health
            health_interval 10s
        }
    }

    handle /api/* {
        reverse_proxy {{ range service "api-service" }}{{ .Address }}:{{ .Port }} {{ end
}} {
            lb_policy least_conn
            health_check /health
            health_interval 10s
        }
    }

    # Static file serving
    handle /static/* {
```

```
        root * /usr/share/caddy
        file_server
    }

    # Default route
    handle / {
        respond "Caddy Gateway Running on Nomad"
    }
}
EOF
        destination = "local/Caddyfile"
      }

      resources {
        cpu    = 300
        memory = 512
      }
    }
  }
}
```

4. Caddy with TLS and Automated Certificates

```
# caddy-tls.nomad
job "caddy-tls" {
  datacenters = ["dc1"]

  group "caddy" {
    count = 2

    network {
      port "http" { to = 80 }
      port "https" { to = 443 }
    }

    service {
      name = "caddy-tls"
      port = "https"

      tags = ["https", "tls", "caddy"]

      check {
        type     = "http"
        path     = "/health"
        port     = "https"
        interval = "15s"
        timeout  = "3s"
        tls_skip_verify = true
      }
    }

    task "caddy" {
      driver = "docker"

      config {
        image = "caddy:2-alpine"
        ports = ["http", "https"]

        volumes = [
          "local/Caddyfile:/etc/caddy/Caddyfile",
          "caddy_data:/data",    # Persistent TLS certificates
          "caddy_config:/config" # Persistent configuration
        ]
      }

      template {
        data = <<EOF
{
    email = "admin@your-domain.com"
```

```
    # Persistent storage for TLS certs
    storage file_system /data
}

your-domain.com {
    tls {
        alpn http/1.1 h2
    }

    # Redirect HTTP to HTTPS
    redir https://{host}{uri} permanent

    handle /health {
        respond "TLS Caddy Healthy" 200
    }

    # API routes
    handle /api/* {
        reverse_proxy {{ range service "api" }}{{ .Address }}:{{ .Port }} {{ end }} {
            header_up X-Forwarded-Proto https
        }
    }

    # Static assets
    handle {
        root * /usr/share/caddy
        file_server
    }
}
EOF
        destination = "local/Caddyfile"
      }

      resources {
        cpu    = 200
        memory = 256
      }
    }
  }
}
```

5. System Caddy for Internal Services

```
# system-caddy.nomad
job "system-caddy" {
  datacenters = ["dc1"]
  type = "system"  # Run on every node

  group "caddy" {
    count = 1  # One per node

    network {
      port "http" {
        static = 80
        to     = 80
      }
    }

    service {
      name = "node-caddy"
      port = "http"

      tags = ["system", "node-proxy"]

      check {
        type     = "http"
        path     = "/health"
        interval = "15s"
        timeout  = "3s"
      }
    }

    task "caddy" {
      driver = "docker"

      config {
        image = "caddy:2-alpine"
        ports = ["http"]

        volumes = [
          "local/Caddyfile:/etc/caddy/Caddyfile",
          "/var/run/nomad:/var/run/nomad:ro"  # Access to Nomad socket
        ]
      }

      template {
        data = <<EOF
{
    # Node-local configuration
```

```
    admin off
}

:80 {
    # Node health information
    handle /health {
        respond <<EOF
<h1>Node Caddy - {{ env "attr.unique.hostname" }}</h1>
<p>Node: {{ env "attr.unique.hostname" }}</p>
<p>IP: {{ env "attr.unique.network.ip-address" }}</p>
<p>Allocation: {{ env "NOMAD_ALLOC_ID" }}</p>
<p>Status: Healthy</p>
EOF
    }

    # Proxy to Nomad API (local node)
    handle /nomad/* {
        reverse_proxy 127.0.0.1:4646 {
            rewrite * /v1/{path}
        }
    }

    # Proxy to Consul (local node)
    handle /consul/* {
        reverse_proxy 127.0.0.1:8500 {
            rewrite * /v1/{path}
        }
    }

    # Services on this node
    handle /services/* {
        reverse_proxy {{ range service "local-services" }}{{ .Address }}:{{ .Port }} {{
end }}
    }

    # Default
    handle {
        respond "Node Caddy Gateway - {{ env "attr.unique.hostname" }}"
    }
}
EOF
        destination = "local/Caddyfile"
      }

      resources {
        cpu    = 100
        memory = 128
```

```
      }
    }
  }
}
```

6. Deployment Commands

```
# Deploy Caddy jobs
nomad job run caddy.nomad
nomad job run caddy-consul.nomad
nomad job run caddy-tls.nomad

# Check status
nomad status caddy
nomad logs -f caddy

# Scale Caddy
nomad job scale caddy 5

# Plan updates
nomad job plan caddy.nomad

# Stop Caddy
nomad job stop caddy
```

7. Key Nomad Features Used

- **Service Discovery**: Automatic discovery of backend services
- **Template Rendering**: Dynamic configuration with Consul service data
- **Resource Management**: CPU and memory allocation
- **Health Checking**: Automatic restarts on failure
- **Scaling**: Easy horizontal scaling
- **Port Management**: Dynamic and static port mapping
- **Storage**: Persistent volumes for TLS certificates

- **Multi-Region**: Can deploy across multiple datacenters

8. Production Considerations

```
# Add to your job specifications for production:

update {
  stagger      = "30s"
  max_parallel = 2
  health_check = "checks"
  min_healthy_time = "10s"
  healthy_deadline = "5m"
  auto_revert = true
}

migrate {
  max_parallel = 1
  health_check = "checks"
  min_healthy_time = "10s"
  healthy_deadline = "8m"
}

reschedule {
  attempts       = 15
  interval       = "1h"
  delay          = "30s"
  delay_function = "exponential"
  max_delay      = "1h"
  unlimited      = false
}
```

This setup gives you a highly available, scalable Caddy deployment that integrates seamlessly with the HashiCorp ecosystem (Nomad + Consul) for service discovery and orchestration.