



AVIOX MAGAZINE

DJANGO, REACT
AND PYTHON
FOUNDATIONS,
AND THE SERVERS
THAT RUN THEM

Building the Web

VOL. 1 • NO. 1
JULY 2026

In this issue

DEVELOPMENT

3

Single Page Applications (SPAs): The Future of Modern Web Development	3
Asynchronous Programming in Python	5
How to Use Django Signals in Your Django Application	8
How to Use Custom Middleware in Your Django Application	10
How to Secure React Applications with Auth0 Authentication	12

RUNNING IT IN PRODUCTION

15

Running Jenkins in Docker: A Practical Server Setup	15
Securing Nginx with Let's Encrypt	17

FROM THE EDITOR

Welcome to the first issue of Aviox Magazine. Each month we collect the best of what we have published on Aviox Blogs into one place, so you can read it in order, at your own pace.

This issue goes back to the foundations. We start with why single page applications took over the modern web, then look at asynchronous Python, Django signals and custom middleware, and how to add secure sign-in to a React app with Auth0.

Building is only half the job, so we finish in production: running Jenkins in Docker, and putting Nginx behind free HTTPS certificates from Let's Encrypt. We hope you enjoy it.

— *The Aviox Blogs Editors*

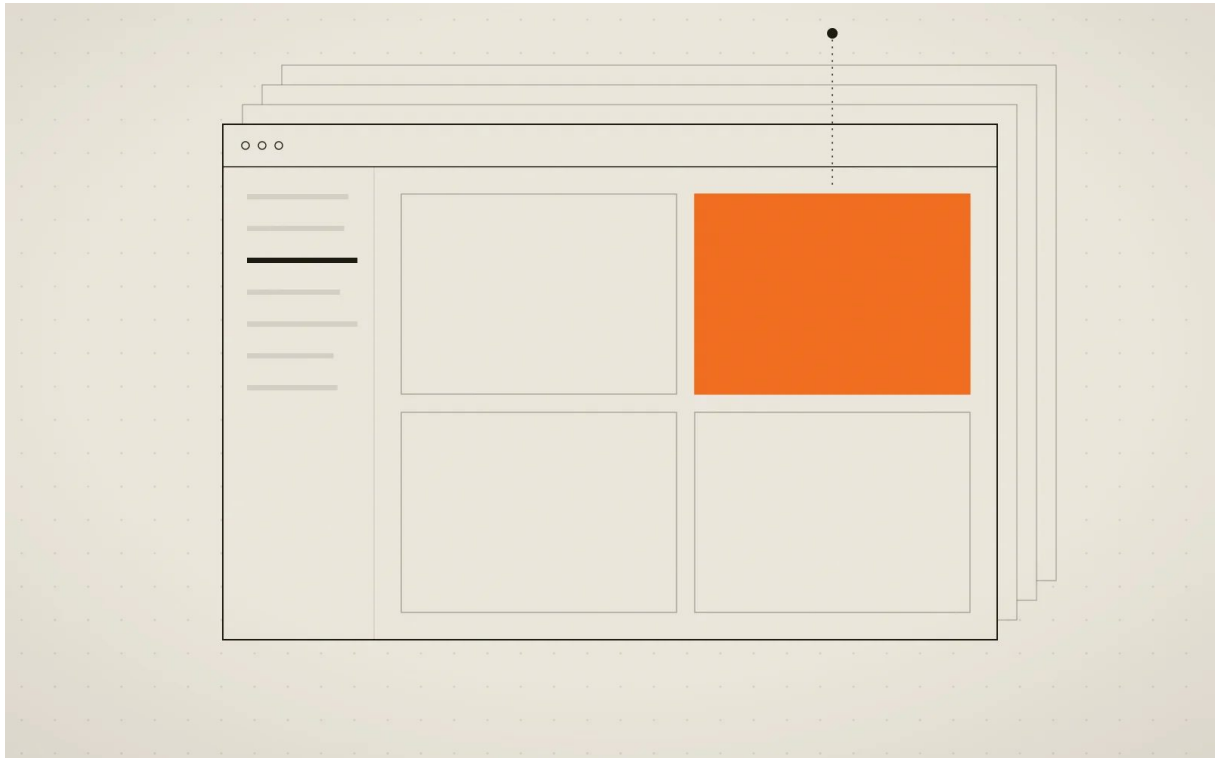
© 2026 Aviox Blogs. All rights reserved, except where noted: stories republished from other publishers remain theirs and are credited with their licence. No other part of this publication may be copied, changed or republished without written permission from Aviox Blogs.
Read this issue online at <https://blogs.avioxtechnologies.com/magazine/building-the-web>

DEVELOPMENT

Single Page Applications (SPAs): The Future of Modern Web Development

Load the application once, then update only what changes. Why the single-page model reshaped how the web feels — and what it asks of the teams who build it.

By Yogesh Joshi · 10 Jun 2026 · 4 min read



Introduction

The way users interact with websites has evolved significantly over the years. Today's users expect fast, seamless, and responsive digital experiences without unnecessary page reloads or delays. To meet these expectations, developers increasingly rely on Single Page Applications (SPAs), a modern approach to web development that delivers smooth and engaging user experiences.

Single Page Applications have transformed how web applications are built by allowing content to update dynamically within a single webpage. This approach reduces loading times and creates a more app-like experience for users.

What is a Single Page Application?

A Single Page Application (SPA) is a web application that loads a single HTML page and dynamically updates content as users interact with it. Instead of requesting an entirely new page from the server every time a user clicks a link or performs an action, SPAs update only the necessary content on the screen.

This results in faster navigation, improved performance, and a more responsive user experience.

How Single Page Applications Work

Unlike traditional websites that reload complete pages for every request, SPAs load the core application once and communicate with servers in the background using APIs. When users perform actions such as navigating between sections, submitting forms, or viewing new content, the application updates only the relevant parts of the interface.

Diagram comparing a traditional multi-page site, where every click requests a full HTML page, with a single-page application, where the shell loads once and later interactions exchange only JSON with an API.

After the first load, a SPA exchanges data — not documents — with the server.

This dynamic behavior makes the application feel similar to a desktop or mobile app while still running inside a web browser.

Key Benefits of SPAs

Faster user experience

Since only specific content updates rather than entire pages, users experience significantly faster interactions and smoother navigation throughout the application.

Reduced server load

SPAs minimize repeated page requests, reducing the workload on servers and improving application efficiency.

Better performance

Once the application is loaded, subsequent interactions are handled quickly, leading to improved responsiveness and overall performance.

Enhanced user engagement

A seamless interface encourages users to stay longer and interact more with the application, leading to higher engagement and satisfaction.

Simplified development

Modern frameworks and libraries make it easier for developers to build reusable components and maintain large-scale applications efficiently.

This story continues for Dispatch subscribers — about 2 more minutes of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

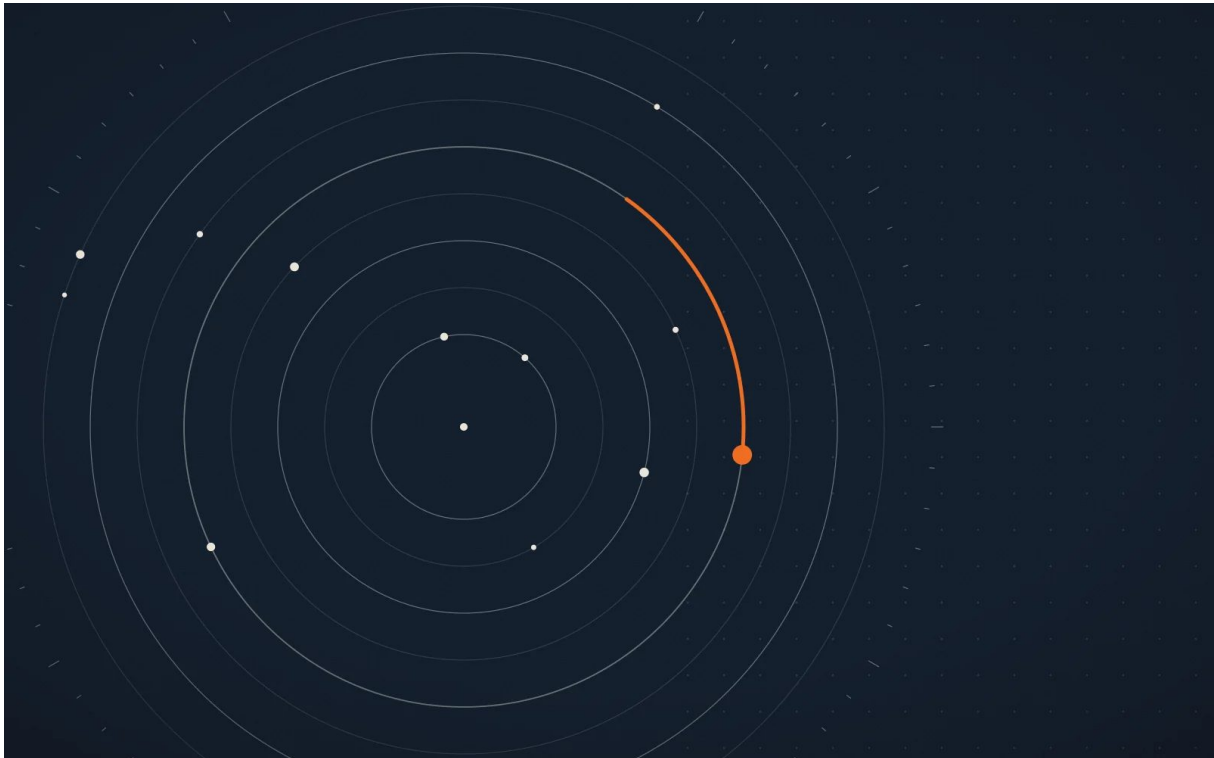
Read online: <https://blogs.avioxtechnologies.com/story/single-page-applications-spas-the-future-of-modern-web-development>

DEVELOPMENT

Asynchronous Programming in Python

Most programs spend their time waiting — on networks, disks and other services. Asynchronous Python turns that waiting into useful work.

By Yogesh Joshi · 10 Jun 2026 · 4 min read



Introduction

In modern software development, performance and responsiveness are essential factors in building successful applications. Traditional programming approaches often execute tasks sequentially, causing delays when applications need to perform multiple operations simultaneously. Asynchronous programming in Python provides an efficient solution by allowing tasks to run concurrently without blocking the execution flow.

This approach helps developers build faster, more scalable, and highly responsive applications, making it an important skill for modern Python development.

What is Asynchronous Programming?

Asynchronous programming is a programming paradigm that enables a program to initiate a task and continue executing other operations before the first task completes. Instead of waiting for one operation to finish, the application can handle multiple activities simultaneously, improving overall efficiency and user experience.

This model is especially useful for operations that involve waiting, such as network requests, database interactions, file processing, and external API calls.

Why Asynchronous Programming Matters

Modern applications often interact with multiple services and resources. When these operations are performed synchronously, the application spends a significant amount of time waiting for responses. Asynchronous programming eliminates unnecessary waiting periods and allows applications to make better use of available resources.

The primary benefits include improved performance, enhanced responsiveness, better scalability, reduced resource consumption, and the ability to handle large numbers of concurrent users efficiently.

Key Concepts of Asynchronous Programming

Understanding asynchronous programming requires familiarity with several core concepts. Tasks are executed independently and can pause when waiting for external resources. During this waiting period, other tasks continue running, ensuring that the application remains productive.

Diagram of the asyncio event loop: three tasks take turns running; whenever one awaits I/O it yields to the loop, which resumes another task that is ready.

When a task awaits, it hands control back to the loop instead of blocking it.

The event loop acts as the central coordinator, managing task execution and scheduling operations whenever resources become available. This architecture enables smooth and efficient handling of concurrent workloads.

Seeing it in code

This short example from the Aviox Blogs engineering docs schedules a coroutine as a task, carries on with other work, and only waits for the result when it is actually needed:

```
import asyncio

async def square(number):
    await asyncio.sleep(1) # stands in for a network or disk call
    return number ** 2

async def main():
    task = asyncio.create_task(square(5))
    await asyncio.sleep(0.5)
    print("Doing some other work...")
    result = await task
    print("Result:", result)

asyncio.run(main())

tasks.py
```

Concurrency still needs limits. A semaphore caps how many coroutines can enter a section at once — useful when an external API enforces rate limits:

```
import asyncio

async def worker(semaphore, name):
    async with semaphore:
        print(f"{name} is working")
        await asyncio.sleep(1)
```

```
print(f"{name} is done")
```

```
async def main():  
    semaphore = asyncio.Semaphore(2)  
    await asyncio.gather(*(worker(semaphore, f"Worker {i}") for i in range(1, 5)))
```

```
asyncio.run(main())
```

workers.py

This story continues for Dispatch subscribers — about 1 more minute of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

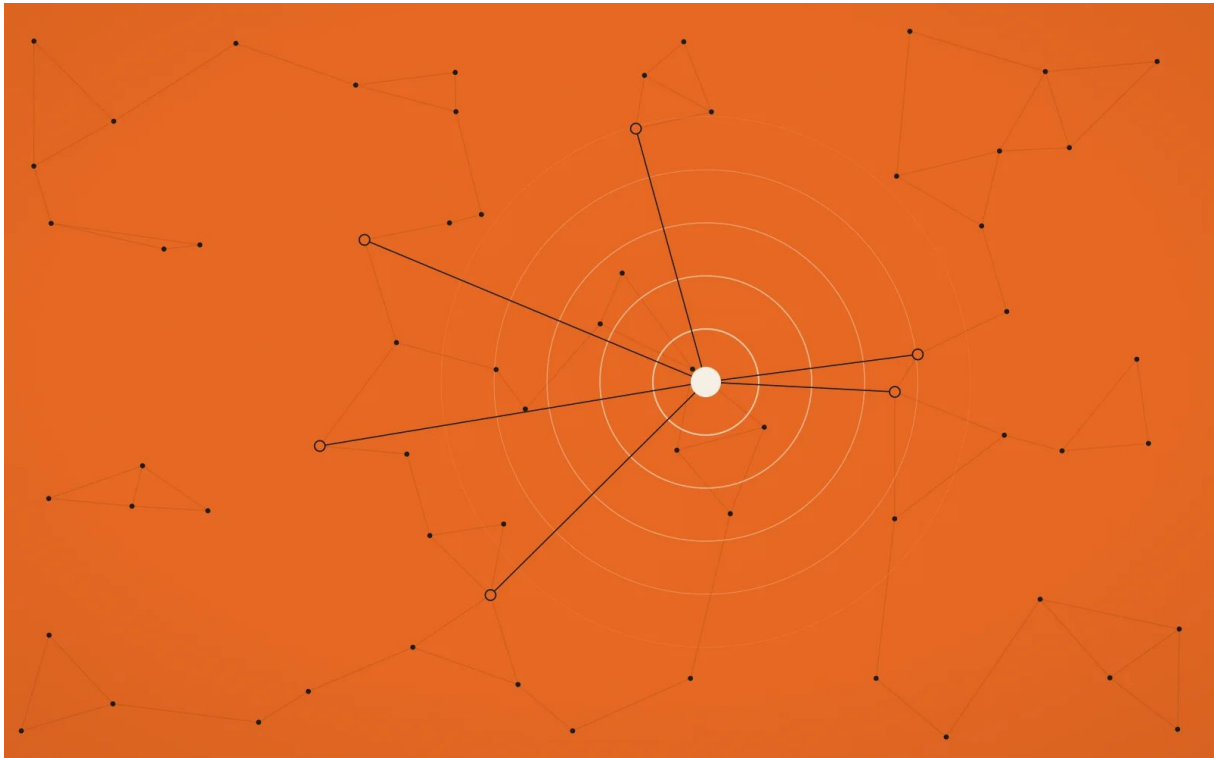
Read online: <https://blogs.avioxtechnologies.com/story/asynchronous-programming-in-python>

DEVELOPMENT

How to Use Django Signals in Your Django Application

Signals let one part of a Django application react to events in another without the two ever depending on each other. Here is when to reach for them — and when not to.

By Yogesh Joshi · 10 Jun 2026 · 5 min read



Introduction

Django is a powerful web framework that provides developers with numerous tools to build scalable and maintainable applications. Among its many features, Django Signals offer an elegant way to enable communication between different parts of an application without creating tight dependencies between components.

Signals allow certain actions to automatically trigger additional functionality when specific events occur. This makes it easier to keep applications organized, reusable, and easier to maintain. Whether you need to send notifications, update records, create logs, or synchronize data across modules, Django Signals can help automate these tasks efficiently.

In this guide, we will explore the purpose of Django Signals, their benefits, common use cases, and best practices for implementing them in modern Django applications.

What are Django Signals?

Django Signals are a mechanism that allows different parts of an application to communicate with one another when predefined events occur. Instead of placing all logic directly inside views or models, developers can use signals to execute actions automatically whenever specific events are triggered.

Signals help create a loosely coupled architecture where application components remain independent while still working together efficiently.

Diagram: a model save acts as the sender and emits a `post_save` signal, which three independent receivers handle — creating a profile, sending a welcome email and writing an audit log.

A single event fans out to receivers that know nothing about each other.

For example, when a new user registers, a signal can automatically create a user profile, send a welcome email, or log the registration activity without modifying the registration process itself.

Why Use Django Signals?

As applications grow, maintaining clean and organized code becomes increasingly important. Django Signals provide a way to separate business logic from core application functionality. Some common reasons for using signals include:

- Automating repetitive tasks
- Reducing code duplication
- Keeping applications modular
- Improving maintainability
- Creating event-driven workflows
- Managing background operations

By separating event-based actions from primary application logic, developers can build more scalable and flexible systems.

How Django Signals Work

Signals operate through a sender and receiver mechanism. When a specific event occurs, the sender emits a signal. One or more receivers listen for that signal and execute predefined actions in response.

This communication happens automatically behind the scenes, allowing developers to react to events without modifying existing application workflows. The result is a cleaner architecture that supports better organization and future scalability.

This story continues for Dispatch subscribers — about 4 more minutes of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

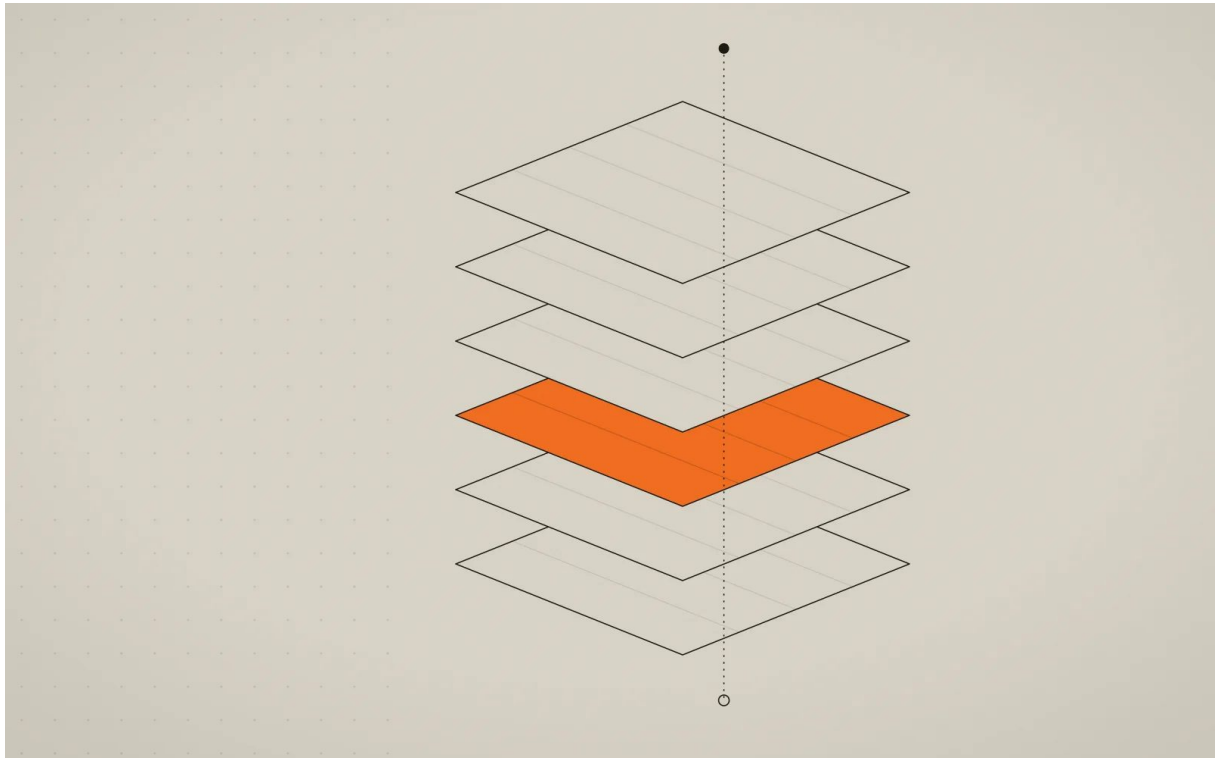
Read online: <https://blogs.avioxtechnologies.com/story/how-to-use-django-signals-in-your-django-application>

DEVELOPMENT

How to Use Custom Middleware in Your Django Application

Every request and response in Django passes through the same pipeline. Custom middleware is how you add logging, security and monitoring to all of it at once.

By Yogesh Joshi · 10 Jun 2026 · 6 min read



Introduction

Django is one of the most popular Python web frameworks, known for its simplicity, scalability, and robust architecture. One of the powerful features provided by Django is Middleware, which allows developers to process requests and responses globally throughout the application.

Custom middleware enables developers to execute specific logic before a request reaches a view or before a response is returned to the user. Whether you want to track user activity, implement custom authentication, log requests, or modify responses, middleware provides a clean and efficient solution.

In this guide, we will explore the concept of custom middleware, its benefits, use cases, and best practices for implementing it in Django applications.

What is Middleware in Django?

Middleware is a framework of hooks that sits between the web server and the Django application. It processes requests before they reach the view layer and handles responses before they are sent back to the client.

Think of middleware as a pipeline through which every request and response passes.

This allows developers to perform operations globally without modifying individual views or business logic. Middleware plays a critical role in managing security, authentication, sessions, caching, logging, and request processing.

Why Use Custom Middleware?

While Django provides several built-in middleware components, there are situations where application-specific functionality is required. Custom middleware helps developers add reusable functionality across the entire application. Some common reasons for using custom middleware include:

- Request and response logging
- User activity tracking
- Performance monitoring
- Custom authentication checks
- API request validation
- Security enhancements
- Request filtering
- Response customization

By centralizing these tasks, developers can keep application code cleaner and easier to maintain.

How Middleware Works

Whenever a user sends a request to a Django application, the request passes through a sequence of middleware components before reaching the target view. After the view generates a response, the response travels back through the middleware stack before being returned to the browser.

Diagram: a request travels down through Security, Session, Authentication and a custom middleware layer to the view; the response travels back up through the same layers in reverse order.

Requests travel down the stack in settings order; responses travel back up in reverse.

This request-response lifecycle allows middleware to inspect, modify, or reject requests and responses at any stage of the process. The middleware execution order is determined by the order in which middleware classes are registered within the application settings.

This story continues for Dispatch subscribers — about 4 more minutes of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

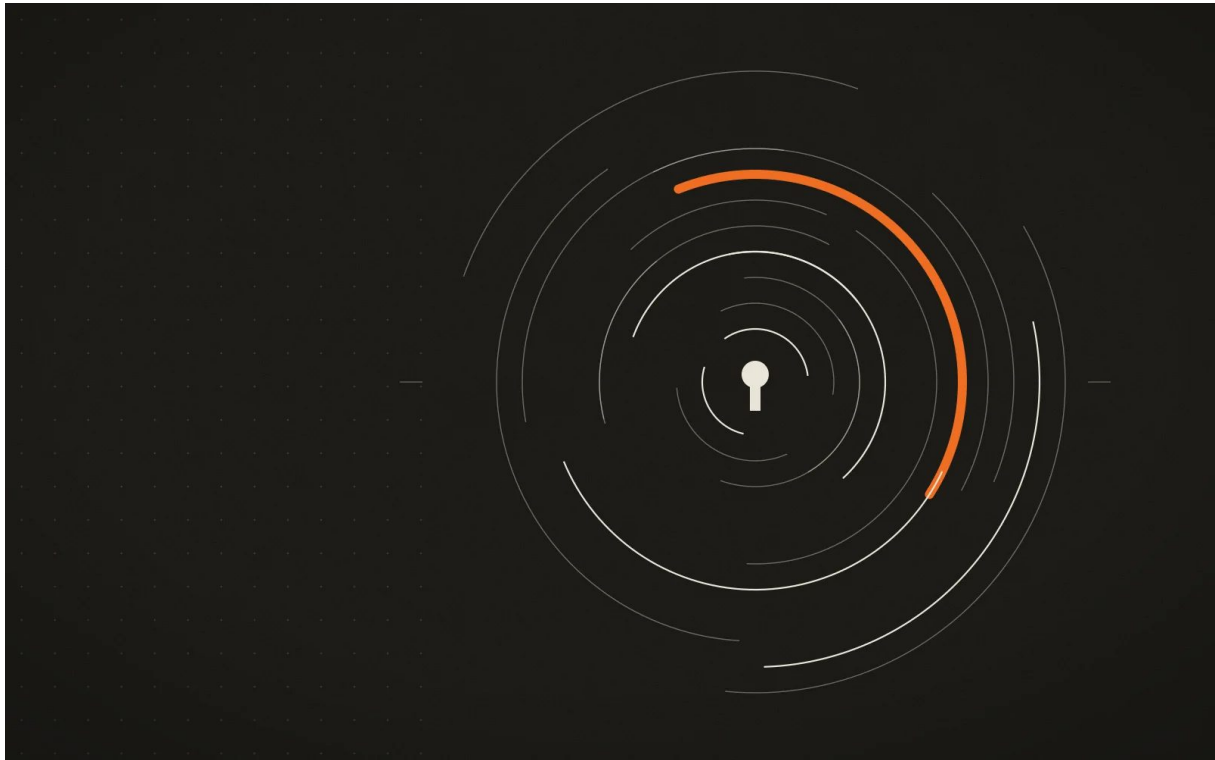
Read online: <https://blogs.avioxtechnologies.com/story/how-to-use-custom-middleware-in-your-django-application>

DEVELOPMENT

How to Secure React Applications with Auth0 Authentication

Authentication is too important to improvise. Auth0 gives React teams secure login, MFA and single sign-on without building identity infrastructure from scratch.

By Yogesh Joshi · 10 Jun 2026 · 5 min read



Introduction

Security is one of the most critical aspects of modern web application development. As React applications continue to grow in complexity and handle sensitive user information, implementing a reliable authentication system becomes essential. Authentication ensures that only authorized users can access protected resources and perform specific actions within an application.

Auth0 is a popular identity and access management platform that simplifies authentication and authorization for developers. It provides secure login, user management, social authentication, multi-factor authentication, and enterprise identity solutions without requiring developers to build authentication systems from scratch.

In this guide, we will explore how Auth0 helps secure React applications and why it has become a preferred authentication solution for modern web development.

Understanding Authentication in React Applications

Authentication is the process of verifying a user's identity before granting access to an application. Without proper authentication mechanisms, applications become vulnerable to unauthorized access, data breaches, and security threats.

React applications often communicate with APIs and backend services, making it important to ensure that only authenticated users can access protected resources. A secure authentication system helps maintain user privacy, protect sensitive information, and build trust among users.

What is Auth0?

Auth0 is a cloud-based identity platform that provides authentication and authorization services for web, mobile, and enterprise applications. It enables developers to implement secure login systems quickly while following modern security standards and best practices. Auth0 supports various authentication methods, including:

- Email and password authentication
- Social login providers
- Single Sign-On (SSO)
- Multi-Factor Authentication (MFA)
- Enterprise identity providers
- Passwordless authentication

By handling complex authentication workflows, Auth0 allows developers to focus on building application features rather than security infrastructure.

Why Use Auth0 with React?

React applications are highly interactive and often require secure user authentication. Auth0 integrates seamlessly with React and offers several advantages that improve both security and user experience.

Enhanced security

Auth0 follows industry-standard security protocols and continuously updates its platform to protect against emerging threats.

Faster development

Developers can implement authentication functionality without building and maintaining custom authentication systems.

Better user experience

Users can log in using multiple authentication options, reducing friction during the onboarding process.

Scalability

Auth0 is designed to support applications ranging from small startups to large enterprise systems with millions of users.

Compliance and reliability

Organizations benefit from enterprise-grade security features, monitoring, and compliance support.

This story continues for Dispatch subscribers — about 4 more minutes of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

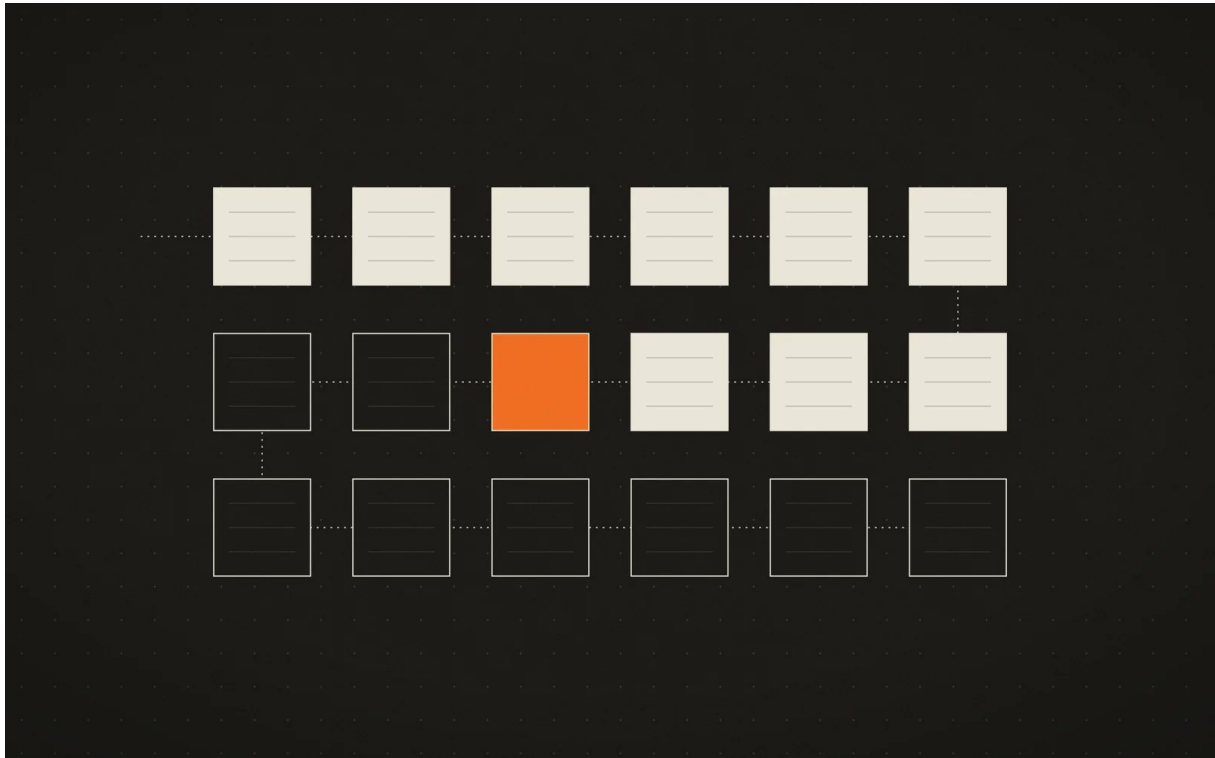
Read online: <https://blogs.avioxtechnologies.com/story/how-to-secure-react-applications-with-auth0-authentication>

TECHNOLOGY

Running Jenkins in Docker: A Practical Server Setup

A CI server you can recreate in one command. Run Jenkins in a container, keep its state in a volume, and give it controlled access to Docker for building images.

By Aviox Blogs Engineering · 22 Jul 2026 · 2 min read



Running Jenkins in a container keeps the server itself disposable: its jobs, plugins and credentials live in a Docker volume, so the container can be upgraded or replaced without losing anything. This guide sets that up on an Ubuntu server and prepares Jenkins to build and deploy Docker-based applications.

Before you start

You'll need an Ubuntu server (20.04 or 22.04 recommended), root or sudo access, and an internet connection.

Install Docker

```
sudo apt update
sudo apt install -y docker.io
sudo systemctl enable --now docker
docker --version
```

Give Jenkins a Docker client

The official `jenkins/jenkins:lts` image does not include the Docker CLI. To let pipelines run `docker build`, extend the image with the client — the daemon itself stays on the host:

```
FROM jenkins/jenkins:lts
```

```
USER root
RUN apt-get update \
  && apt-get install -y --no-install-recommends ca-certificates curl \
  && install -m 0755 -d /etc/apt/keyrings \
  && curl -fsSL https://download.docker.com/linux/debian/gpg -o /etc/apt/keyrings/docker.asc \
asc \
  && echo "deb [signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/debian $(. /etc/os-release && echo "$VERSION_CODENAME") stable" \
  > /etc/apt/sources.list.d/docker.list \
  && apt-get update && apt-get install -y docker-ce-cli \
  && rm -rf /var/lib/apt/lists/*
USER jenkins
```

Dockerfile

```
docker build -t aviox/jenkins-docker .
```

Run the container

```
docker run -d \
  --name jenkins \
  --restart unless-stopped \
  -p 8080:8080 \
  -p 50000:50000 \
  -v jenkins_home:/var/jenkins_home \
  -v /var/run/docker.sock:/var/run/docker.sock \
  --group-add $(stat -c '%g' /var/run/docker.sock) \
  aviox/jenkins-docker
```

Option	Purpose
-p 8080:8080	The Jenkins web interface
-p 50000:50000	Communication with Jenkins agents
-v jenkins_home:...	Persistent storage for jobs, plugins and credentials
-v /var/run/docker.sock:...	Lets Jenkins drive the host's Docker daemon
--group-add ...	Grants the jenkins user permission to use that socket

The Docker socket is root access

Anything that can talk to docker . sock can control the host. Restrict who can create and edit jobs, keep plugins updated, and consider dedicated build agents for untrusted code.

This story continues for Dispatch subscribers — about 1 more minute of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

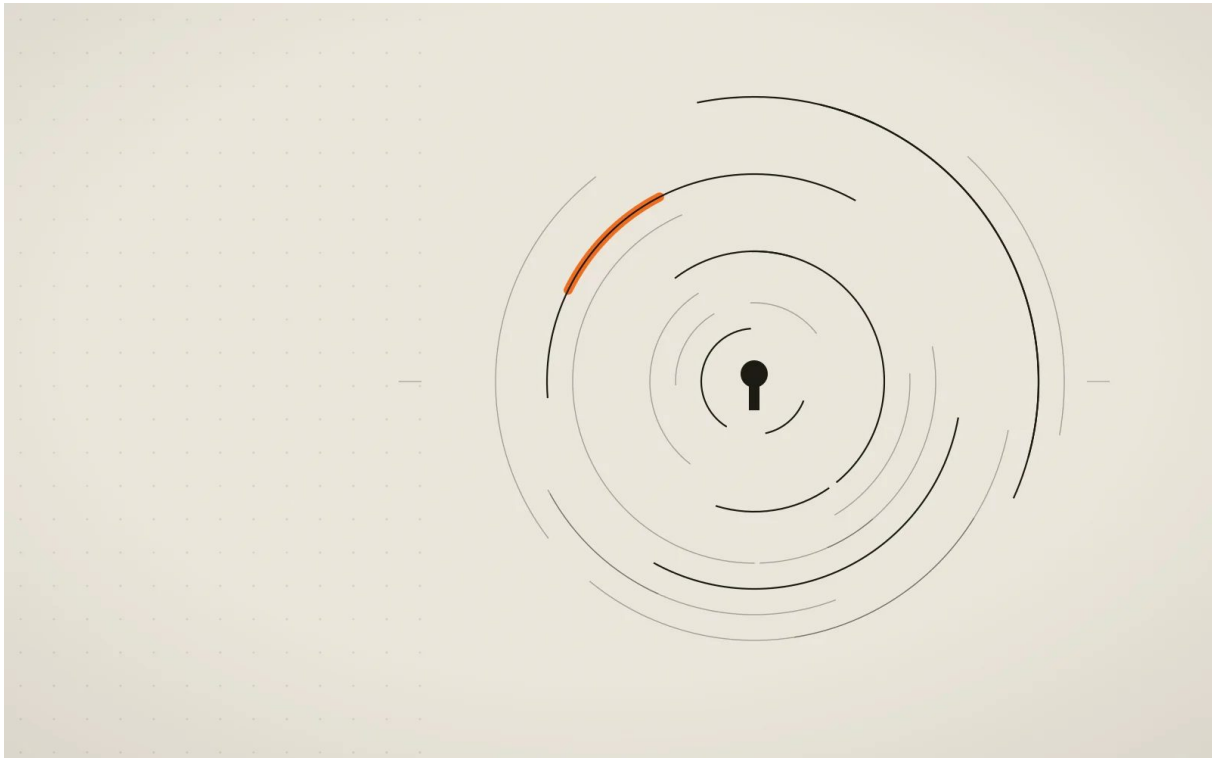
Read online: <https://blogs.avioxtechnologies.com/story/running-jenkins-in-docker>

TECHNOLOGY

Securing Nginx with Let's Encrypt

Free certificates, issued and renewed automatically. A complete walkthrough from an HTTP-only Nginx site to HTTPS with redirects and a tested renewal timer.

By Aviox Blogs Engineering · 29 Jul 2026 · 2 min read



Let's Encrypt is a certificate authority that issues free TLS certificates, making HTTPS available to every site. Its client, Certbot, automates nearly everything — obtaining a certificate, rewriting the Nginx configuration and setting up renewal.

Before you start

You need an Ubuntu server (the steps were written for 20.04 and apply to later releases), a registered domain — example . com below — with an A record pointing at the server, and an application already served by Nginx.

Install Certbot

Install Certbot and its Nginx plugin with apt:

```
sudo apt install certbot python3-certbot-nginx
```

Confirm the Nginx configuration

Certbot finds the right server block by matching its `server_name` directive to the domains you request. Open your site's configuration:

```
sudo nano /etc/nginx/sites-available/example.com
```

Find the existing `server_name` line. It should list your domain:

```
server {  
    listen 80;  
    server_name example.com www.example.com;  
    # ...  
}
```

`/etc/nginx/sites-available/example.com`

If you changed anything, validate the syntax and reload Nginx:

```
sudo nginx -t  
sudo systemctl reload nginx
```

This story continues for Dispatch subscribers — about 1 more minute of reading. Subscribing is free: sign up at <https://blogs.avioxtechnologies.com/subscribe> and download this issue again to get every story in full.

Read online: <https://blogs.avioxtechnologies.com/story/securing-nginx-with-lets-encrypt>