Web application deployment is the process of making a web application accessible to users by transferring it from a development environment to a hosting environment.

8.1 Introduction to Deployment:

- **Definition:**

Deployment is the act of making a web application operational and accessible to users. It's the final step in the software development lifecycle, moving the application from development to production.

- **Purpose:**

To deliver the application to users, allowing them to interact with it and utilize its functionality.

- **Key aspects:**
- **Transferring code and resources:** Moving the application's files (code, assets, etc.) to the hosting environment.
- **Configuring the environment:** Setting up the server and other necessary components to run the application.
- **Making the application accessible:** Ensuring users can connect to the application through a web browser.
- **Maintaining the application:** Ongoing tasks like updates, bug fixes, and performance monitoring.

8.2 Deployment Platforms:

- **Definition:**

Deployment platforms provide the infrastructure and tools needed to host and manage web applications.

- **Types of platforms:**
- **Cloud platforms:** Services like Google Cloud Run, AWS Elastic Beanstalk, Azure App Service, and Heroku offer managed hosting, scaling, and other features.
- **Virtual Private Servers (VPS):** Virtual machines that allow for more control over the environment, but require more management.
- **Shared Hosting:** A cost-effective option where multiple websites share resources on a single server.

- **On-premise servers:** Servers located within an organization's own infrastructure.

Basic differences between development and production environments.

Here are the **basic differences between development and production environments**:

Feature	Development Environment	Production Environment
Purpose	Used for building, testing, and debugging code	Used to serve the final application to end-users
Stability	Less stable, frequently changing	Highly stable, minimal changes
Performance	Performance is not a priority	Optimized for speed, scalability, and reliability
Security	Lower security controls	High-level security measures in place
Data Used	Test or dummy data	Real user and application data
Access	Accessible only to developers or internal teams	Publicly accessible or accessed by real users
Logging & Debugging	Verbose logging and debugging tools enabled	Minimal logging; debugging disabled or restricted
Error Handling	Detailed error messages for troubleshooting	Generic or masked error messages for security
Updates	Frequent updates and changes	Infrequent updates; follows a deployment schedule

Programming language or framework (e.g., **Django**, **Node.js**, **React**, **Laravel**)

Platform or environment (e.g., **AWS**, **Heroku**, **Docker**)

Type of application (e.g., **web app**, **mobile app**, **API**, etc.)

1. What Are Environment Variables?

Environment variables store configuration values that differ between environments (development, testing, production), such as:

- Database connection strings
- API keys
- Secrets
- Debug settings

These are kept **outside your codebase** to keep things secure and configurable.

2. Development vs. Production Variables

Variable	Development	Production
NODE_ENV	development	production
DEBUG	true	false or not set
DB_URI	Local DB (e.g., localhost)	Remote, secure DB (e.g., AWS RDS)
PORT	3000	80 or 443 (with SSL)
API_KEY	Test key	Live API key

3. How to Set Up Environment Variables

Step 1: Create .env File (For Development)

```
NODE_ENV=development
PORT=3000
DB_URI=mongodb://localhost/dev-db
API_KEY=test123
DEBUG=true
```

Never commit .env to version control – add it to .gitignore.

Step 2: Load Environment Variables

In **Node.js** using the dotenv package:

```
npm install dotenv
```

In your main file (e.g., index.js):

```
require('dotenv').config();

const express = require('express');
const app = express();

const port = process.env.PORT || 3000;
console.log(`Running in ${process.env.NODE_ENV} mode`);

app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```

✅ Step 3: Set Environment Variables in Production

On platforms like **Linux, Heroku, or AWS**, set environment variables directly in the shell or deployment config:

Linux:

```
export NODE_ENV=production
export DB_URI=mongodb+srv://prod-db
```

Heroku:

```
heroku config:set NODE_ENV=production DB_URI=...
```

🔐 4. Basic Production Settings

In production, consider:

- **Disable debug logging** (DEBUG=false)
- **Use secure HTTP (HTTPS)** – with SSL/TLS
- **Enable performance optimizations**
- **Use a process manager** like **PM2** for Node.js
- **Set proper CORS and security headers**
- **Use a .env.production or secrets manager** (e.g., AWS Secrets Manager)

What is Gunicorn?

Gunicorn (Green Unicorn) is a **Python WSGI HTTP server** commonly used to **serve Flask (or Django) apps in production**. It's:

- **Lightweight**
- **Highly performant**
- **Works behind a reverse proxy** (like **Nginx**)
- **Supports multiple worker processes**

✅ 1. Install Gunicorn

Make sure you have Flask and Gunicorn installed (ideally inside a virtual environment):

```
pip install flask gunicorn
```

📝 2. Basic Flask App (app.py)

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello from Flask + Gunicorn!"
```

Make sure this file exposes the app variable – Gunicorn looks for it.

🚀 3. Run with Gunicorn

In the terminal, run:

```
gunicorn app:app
```

Explanation:

- app:app → module_name:app_instance
  - First app: name of your Python file (app.py)
  - Second app: Flask instance (app = Flask(__name__))

Default: runs on http://127.0.0.1:8000

⚙ 4. Common Options

```
gunicorn app:app \
  --bind 0.0.0.0:8000 \
  --workers 4 \
  --log-level info
```

Option	Description
--bind	IP and port to listen on
--workers	Number of worker processes
--log-level	Logging level (debug, info, etc)

5. Production Setup (Gunicorn + Nginx)

In production, Gunicorn is typically run **behind Nginx** for:

- SSL termination (HTTPS)
- Static file handling
- Load balancing

Architecture:

Browser ⇨ Nginx ⇨ Gunicorn ⇨ Flask app

👩‍💻 6. Running Gunicorn with a Process Manager

Use a process manager like **systemd**, **supervisor**, or **honcho** to keep Gunicorn running in the background and restart it on crashes or reboots.

📄 Example systemd Service File

```
 [Unit]
Description=Gunicorn instance to serve Flask app
After=network.target

[Service]
User=ubuntu
Group=www-data
WorkingDirectory=/home/ubuntu/myapp
ExecStart=/home/ubuntu/myenv/bin/gunicorn app:app -b 127.0.0.1:8000 -w 4

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

Why HTTPS?

- Encrypts communication between the user and server
- Protects sensitive data (passwords, tokens)
- Required for most APIs, browsers, and SEO

▯ Prerequisites

✅ You need:

- A **domain name** (e.g., example.com)
- A **public server** (e.g., Ubuntu/Debian on AWS, DigitalOcean, etc.)

- **Nginx** installed (acts as reverse proxy)
- **Gunicorn** serving your Flask app

⚙ Step 1: Install Certbot

For Ubuntu/Debian:

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

🌐 Step 2: Configure Nginx for Your Flask App

Example Nginx config (/etc/nginx/sites-available/flaskapp):

```
server {
   listen 80;
   server_name yourdomain.com www.yourdomain.com;

   location / {
      proxy_pass http://127.0.0.1:8000;  # Gunicorn address
      proxy_set_header Host $host;
      proxy_set_header X-Real-IP $remote_addr;
   }
}
```

Enable the site:

```
sudo ln -s /etc/nginx/sites-available/flaskapp /etc/nginx/sites-enabled
sudo nginx -t  # Check for syntax errors
sudo systemctl reload nginx
```

Ensure your app is live at http://yourdomain.com.

✅ Step 3: Get a Free SSL Certificate with Certbot

```
sudo certbot --nginx
```

Certbot will:

- Automatically update your Nginx config to support HTTPS

- Redirect HTTP → HTTPS
- Install the certificate

You’ll be prompted to:

- Enter your email (for renewal notices)
- Agree to the terms
- Choose whether to redirect HTTP to HTTPS (recommended)

🔄 Step 4: Auto-Renewal of Certificates

Let’s Encrypt certs expire every 90 days, but Certbot installs a **cron job** for automatic renewal.

To test renewal:

```
sudo certbot renew --dry-run
```

🔒 Final HTTPS Nginx Block (Auto-configured)

After Certbot runs, your Nginx block might look like:

```
server {
  listen 80;
  server_name yourdomain.com www.yourdomain.com;
  return 301 https://$host$request_uri;
}

server {
  listen 443 ssl;
  server_name yourdomain.com www.yourdomain.com;

  ssl_certificate /etc/letsencrypt/live/yourdomain.com/fullchain.pem;
  ssl_certificate_key /etc/letsencrypt/live/yourdomain.com/privkey.pem;

  location / {
    proxy_pass http://127.0.0.1:8000;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
  }
}
```

What is Monitoring and Logging?

Term	Purpose
Logging	Capturing and storing information about the application's behavior
Monitoring	Observing the system's health, performance, and availability over time

Together, they help you:

- Debug issues
- Track system health
- Identify performance bottlenecks
- Detect security threats or anomalies

📒 1. Logging Basics

Logging means recording events or messages, typically to a file or service.

🔧 Example in Flask:

```
import logging

logging.basicConfig(level=logging.INFO)

@app.route('/')
def home():
   app.logger.info("Home page accessed")
   return "Welcome"
```

🔥 Common Log Levels:

- DEBUG: Detailed info, used in development
- INFO: General events (e.g. server started)
- WARNING: Something unexpected, not fatal
- ERROR: A problem that caused failure
- CRITICAL: Serious error, program may crash

📂 Log Destinations:

- Console (stdout)
- Log files (/var/log/app.log)
- External log management tools

📡 2. Monitoring Basics

Monitoring tools track:

Metric Type	Examples
System Metrics	CPU, memory, disk, network
Application Metrics	Requests/sec, response time, error rates
Uptime Monitoring	Is the server or endpoint live?

🛠 3. Popular Tools

🔍 Logging Tools:

- **Fluentd**, **Logstash** (log collectors)
- **Elasticsearch + Kibana (ELK Stack)**
- **Graylog**, **Datadog Logs**, **Papertrail**

📊 Monitoring Tools:

- **Prometheus + Grafana** (metrics & visualization)
- **New Relic**, **Datadog**, **Sentry**
- **Uptime Robot**, **Pingdom** (for uptime checks)

✅ 4. Production Best Practices

- Use **structured logs** (JSON format) if integrating with tools
- Separate **access logs** from **application logs**
- Rotate and archive logs (e.g., using logrotate)
- Alert on key events (e.g., repeated 500 errors, downtime)

🚨 Example: Simple Log File in Gunicorn + Flask

bash

CopyEdit

```
gunicorn app:app \
  --access-logfile /var/log/gunicorn/access.log \
  --error-logfile /var/log/gunicorn/error.log
```

Importance of Security in Deployment

Security is **crucial** during deployment because it protects your application, users, and infrastructure from **attacks, data breaches, and service disruptions**.

Here’s why it matters and what’s at stake:

🚨 Why Security Matters in Deployment

Risk	Description
Data Breach	User data (e.g. emails, passwords, payment info) can be exposed or stolen.
Service Downtime	Attacks like DDoS or code injections can crash your app, causing lost revenue and trust.
Unauthorized Access	Weak server or API security can let attackers take control of your systems.
Reputation Damage	A single security failure can destroy user trust and brand credibility.
Legal Penalties	Violating data protection laws (like GDPR or HIPAA) can result in heavy fines.

🛡 Key Security Practices During Deployment

1. Use HTTPS

- Encrypts data in transit.
- Use Let's Encrypt for free SSL/TLS certificates.

2. Environment Variables

- Keep secrets (API keys, DB passwords) **out of code** using .env files or secret managers.
- Never expose sensitive data in logs or error messages.

3. Limit Permissions

- Run apps as **non-root users**.
- Apply **least privilege** principles for file access, database roles, and APIs.

4. Keep Software Up to Date

- Regularly patch vulnerabilities in:
  - Operating systems
  - Dependencies (pip, npm, etc.)
  - Frameworks and libraries

5. Input Validation & Sanitization

- Prevent SQL injection, XSS, and other attacks by validating and escaping user input.

6. Use Firewalls and Rate Limiting

- Block unauthorized access to ports.
- Prevent abuse with **rate limiting** or **API throttling**.

7. Logging & Monitoring

- Track login attempts, errors, and suspicious behavior.
- Use alerting for anomalies.

8. Backup & Recovery

- Have automated backups for databases and important files.
- Test recovery procedures regularly.

9. Authentication & Authorization

- Use strong password policies and secure authentication methods (OAuth, JWT, etc.).
- Enforce role-based access control (RBAC).