# 1. Implementation of Web Architecture

Miquel Àngel París i Peñaranda

Web Application Deployment

2nd C-VET Web Application Development

## Index of contents

## 1. Goals.

1. Install and configure web servers on virtual machines and/or the cloud.
2. Perform functional tests on web and application servers.
3. Document the installation and configuration processes performed on web and application servers.

# 2. Web Applications.

## 2.1 Characteristics of Web Applications

1. **User Interface**: Web applications typically have a user interface (UI) accessible through a web browser, allowing users to interact without installing software.
2. **Interactivity**: They offer a high level of interactivity, often using JavaScript, AJAX, or modern frameworks like React, Angular, or Vue.js.
3. **Accessibility**: Web applications are platform-independent and accessible from any device with an internet connection and a browser.
4. **Scalability**: They can be scaled up to handle increasing loads, often using cloud services and microservices architecture.
5. **Security**: Web applications require robust security measures like authentication, encryption, and protection against vulnerabilities like SQL injection or cross-site scripting (XSS).

## 2.2 Types of Web Applications

1. **Monolithic**: All components (frontend, backend, database) are tightly coupled and deployed together as a single unit. This type is simple to develop initially but challenging to scale and maintain.
2. **Layered Architecture (n-tier)**:
   - **Presentation Layer**: Handles UI and user interaction.
   - **Business Logic Layer**: Contains the core application logic.
   - **Data Access Layer**: Manages communication with the database.

   This type promotes separation of concerns, making maintenance easier.
3. **Microservices**: The application is broken into smaller, independently deployable services that communicate over a network. This approach offers high scalability, maintainability, and fault isolation.

4. **Serverless Architecture**: Focuses on deploying functions instead of entire applications, with infrastructure management handled by a cloud provider. It is cost-effective for event-driven use cases.
5. **Single Page Applications (SPA)**: Uses frameworks like React or Angular to create applications that load a single HTML page and dynamically update as the user interacts with the app.

## 2.3. Structure of a Web Application

- **Frontend**: The client-side component where the user interacts, usually involving HTML, CSS, and JavaScript.
- **Backend**: The server-side component handling the business logic, authentication, data processing, and database interaction. It may be built using Java, Python, Node.js, etc.
- **Database**: Stores application data. It could be relational (MySQL, PostgreSQL) or non-relational (MongoDB, Redis).
- **API Layer**: Often used to connect the frontend with the backend or other services, usually through REST or GraphQL APIs.

## 2.4. Resources

1. **Servers**: Web servers (e.g., Apache, Nginx) and application servers (e.g., Tomcat, Node.js) to run the application.
2. **Database Servers**: For managing data, e.g., MySQL, PostgreSQL, or MongoDB.
3. **Load Balancers**: Distribute incoming traffic across multiple servers for better performance and reliability.
4. **Cloud Services**: AWS, Azure, or Google Cloud for hosting, scaling, and managing web applications.
5. **Monitoring Tools**: Tools like Prometheus, Grafana, or New Relic to monitor application performance and server health.

## 2.5. Deployment Descriptor

A deployment descriptor is a configuration file that describes how an application should be deployed on a server. Common examples include:

- **`web.xml` (Java EE)**: Specifies servlets, filters, security settings, and context parameters for Java web applications.
- **`app.yaml` (Google App Engine)**: Defines environment settings, scaling, and other parameters.
- **`docker-compose.yml`**: Used to define and run multi-container Docker applications.
- **`manifest.json`**: Specifies metadata for deploying applications in platforms like Cloud Foundry.

## 2.6. Additional Topics to Consider

- **CI/CD Pipelines**: Continuous Integration and Continuous Deployment processes automate testing and deployment.
- **Security Considerations**: Implementing HTTPS, authentication, authorization, and data protection measures.
- **Performance Optimization**: Caching strategies, CDN usage, and minimizing server response times.
- **Scaling Strategies**: Vertical vs. horizontal scaling, using autoscaling groups, and container orchestration (Kubernetes).
- **Version Control Integration**: Using Git and version control systems to manage code changes and deployments.

## 3. Web Server vs Web Application Server

By understanding both web and application servers, you can optimize performance, scalability, and the security of your deployed applications.

### 3.1. Web Server

A **web server** is responsible for serving static content like HTML, CSS, JavaScript, and images to users. Its primary function is to handle incoming HTTP requests, process them, and deliver the appropriate static resources back to the client (usually a browser).

#### 3.1.1. Fundamentals:

- **Static Content Delivery**: It deals with static files like HTML, images, and CSS, providing fast response times for these resources.
- **Request Handling**: It listens for HTTP/HTTPS requests and responds accordingly by delivering the requested resources.
- **Protocol Support**: It supports protocols like HTTP, HTTPS, and often FTP for file transfer.

#### 3.1.2. Types and Characteristics

1. **Apache HTTP Server**:
   - Open-source and widely used.
   - Highly configurable with modules (e.g., mod_ssl for SSL).
   - Serves static files and proxies requests to application servers.
   - Supports HTTP, HTTPS, and virtual hosting.
2. **Nginx**:
   - Lightweight and high-performance web server, designed for speed and scalability.
   - Can function as a reverse proxy and load balancer.

   - Optimized for serving static files and handling large amounts of concurrent connections.

3. **IIS (Internet Information Services)**:
   - Microsoft's web server, tightly integrated with the Windows ecosystem.
   - Offers support for ASP.NET applications and integrates with Windows authentication mechanisms.

### 3.1.3. Authentication and Security

- **Basic Authentication**: Uses simple username and password combinations encoded in Base64. It is not secure unless combined with HTTPS.
- **Digest Authentication**: A more secure form of basic authentication, it hashes credentials before sending them over the network.
- **SSL/TLS**: Web servers use SSL certificates to provide encrypted connections over HTTPS, ensuring secure communication.

## 3.2. Web Application Server

A **web application server** handles dynamic content generation. Unlike a web server, it can execute complex logic, interact with databases, process business rules, and generate dynamic HTML or JSON responses that are customized for each request. It typically works in tandem with a web server to handle the dynamic aspects of a web application.

### 3.2.1. Fundamentals

- **Dynamic Content Generation**: It handles business logic and dynamic content, often integrating with databases to generate data-driven pages.
- **Application Hosting**: Hosts web applications, often using technologies like Java, Python, PHP, Node.js, etc.
- **Session Management**: Manages user sessions and state, often used in authentication and maintaining user data across multiple requests.

### 3.2.2. Types and Characteristics

1. **Apache Tomcat**:
    - Designed for running Java applications, specifically Java Servlets and JSP (JavaServer Pages).
    - Handles HTTP requests but focuses on dynamic content generation, rather than static content.
2. **Node.js**:
    - A runtime for executing JavaScript server-side, commonly used for building scalable, non-blocking web applications.
    - Acts as both a web server and application server, handling HTTP requests and executing business logic.
3. **Microsoft's ASP.NET Core**:
    - An open-source, cross-platform framework for building web apps using .NET.
    - Often paired with IIS, it is capable of handling both static content and dynamic processing.
4. **JBoss (WildFly)**:
    - A Java EE application server, capable of hosting enterprise-level Java applications.
    - Supports the full suite of Java EE services (transactions, messaging, etc.).

### 3.2.3. Authentication and Security

- **Session-based Authentication**: The server generates a session ID after a user logs in, storing the session data server-side and linking it to the user's requests.
- **Token-based Authentication (JWT, OAuth)**: Instead of session IDs, tokens are generated and sent with every request, often used in stateless applications and APIs.

- **OAuth/OpenID Connect**: Commonly used for allowing third-party applications to authenticate users via another service, such as Google or Facebook, without handling credentials directly.

## 3.3. Protocols

### 3.3.1. HTTP/HTTPS:

- **HyperText Transfer Protocol (HTTP)** is the foundation of communication between a client (usually a browser) and a server.
- **HTTPS** is the secure version of HTTP, ensuring data transmitted between the client and server is encrypted via SSL/TLS.

### 3.3.2 WebSocket:

A communication protocol providing full-duplex communication channels over a single TCP connection, allowing real-time communication between server and client.

#### 3.3.3. FTP (File Transfer Protocol):

Used by some web servers to allow uploading or downloading files from the server.

#### 3.3.4. AJP (Apache JServ Protocol):

A binary protocol used to bridge communication between a web server (like Apache) and a web application server (like Tomcat), enabling efficient load balancing and request handling.

## 3.4. Differences Between Web Server and Web Application Server

1. **Content Handling**:
   - Web Server: Handles static content like HTML, CSS, and images.
   - Web Application Server: Processes dynamic content, interacting with business logic and databases.

2. **Technologies**:
   - Web Server: Typically supports HTTP and HTTPS.
   - Web Application Server: In addition to HTTP/HTTPS, it may support WebSocket, AJP, and others.
3. **Request Handling**:
   - Web Server: Simply forwards the request, potentially proxying it to an application server.
   - Web Application Server: Executes application logic, processes data, and generates dynamic responses.

## 3.5. Additional Topics to Consider

1. **Reverse Proxy**: Web servers like Nginx often act as reverse proxies, forwarding requests to web application servers.
2. **Load Balancing**: Web and application servers may be clustered for scalability, with load balancers distributing traffic among multiple instances.
3. **Caching Mechanisms**: Servers implement caching strategies (e.g., Nginx caching, content delivery networks) to reduce load and improve performance.

## 4. Installation Options for Web Applications

When deploying a web application, you can choose different environments to install and run your application. Each option comes with specific benefits and trade-offs in terms of scalability, cost, and management.

### 4.1. Physical Machines

A **physical machine** refers to a dedicated server where the hardware resources are directly allocated to your application. You own or rent the machine, and the entire system is at your disposal.

#### 4.1.1. Advantages:

- Full control over hardware and software configurations.
- No resource sharing, so it can offer better performance for resource-intensive applications.
- More predictable performance since no other users can affect resource availability.

#### 4.1.2. Disadvantages:

- Expensive to maintain and scale (adding more physical machines is costly and time-consuming).
- Hardware failures could lead to downtime unless mitigated with backups or redundancy.
- Requires dedicated IT staff for maintenance, updates, and repairs.

#### 4.1.3. Use Cases:

High-performance applications or industries with strict data security requirements, like finance or healthcare.

## 4.2. Virtual Machines (VMs)

A **virtual machine** simulates a complete hardware environment, allowing multiple VMs to run on a single physical server using a hypervisor like VMware, Hyper-V, or KVM. Each VM behaves as a separate machine with its own operating system.

### 4.2.1. Advantages:

- **Resource efficiency**: Multiple virtual machines can run on a single physical server, utilizing hardware resources more efficiently.
- **Isolation**: Each VM is independent, making it more secure and reliable.
- **Scalability**: Easier to scale by adding more VMs instead of physical machines.
- **Portability**: VMs can be moved or duplicated between physical machines for easier scaling or backups.

### 4.2.2. Disadvantages:

- **Performance overhead**: VMs introduce some resource overhead because of the hypervisor layer.
- **Complexity**: More complex than physical machines to configure and manage.

### 4.2.3. Use Cases:

Environments needing scalability, flexibility, and isolated environments like development, testing, and production environments.

## 4.3. Containers (e.g., Docker, Kubernetes)

**Containers** are lightweight, isolated environments for running applications, providing OS-level virtualization. Unlike VMs, containers share the host machine’s kernel but have their own libraries and dependencies.

### 4.3.1. Advantages:

- **Lightweight**: Containers have very low overhead compared to VMs since they don't need their own OS.
- **Fast Deployment**: Containers can be started or stopped in seconds.
- **Consistency**: Containers allow you to create a consistent environment across development, testing, and production.
- **Microservices Support**: Containers are ideal for running microservices, where each component of your application can be deployed independently.
- **Orchestration**: Tools like Kubernetes allow automated scaling, deployment, and management of containerized applications.

#### 4.3.2. Disadvantages:

- **Security**: Since containers share the same kernel, a vulnerability in the host OS could affect all containers.
- **Complexity**: Container orchestration (e.g., with Kubernetes) can add complexity to managing applications at scale.

#### 4.3.3. Use Cases:

Microservices-based applications, CI/CD environments, environments where fast scaling and resource efficiency are critical.

### 4.4. Cloud (e.g., AWS, Azure, Google Cloud)

**Cloud environments** provide virtualized infrastructure and services that allow applications to be hosted remotely. Cloud providers like AWS, Azure, and Google Cloud offer various services, including virtual machines, containers, databases, load balancers, and serverless functions.

#### 4.4.1. Advantages:

- **Scalability**: Elastic scaling allows applications to grow or shrink based on demand, saving costs during low usage times.

- **Cost-Effective**: Pay only for the resources you use (compute, storage, network), which can be more economical for businesses.
- **Global Availability**: Applications can be deployed across multiple regions for low-latency global access and high availability.
- **Managed Services**: Many cloud services handle infrastructure management tasks such as backups, monitoring, and security patches.

### 4.4.2. Disadvantages:

- **Dependence on Vendor**: You're tied to the cloud provider's infrastructure (vendor lock-in).
- **Security and Compliance**: Storing sensitive data in the cloud can introduce regulatory and security concerns.
- **Costs**: Over time, cloud costs can grow if not managed properly, especially with unexpected scaling.

### 4.4.3. Use Cases:

Highly scalable applications, SaaS products, startups, or companies looking for a pay-as-you-go model with global reach.

## 5. Production Environment and Test Environment

When deploying web applications, it's crucial to separate the production and testing environments to ensure stability, quality, and security.

### 5.1. Test Environment (Development/Staging Environment)

A **test environment** (or staging environment) is used for development, testing, and validation before deploying code to the production environment. It’s an isolated environment where developers can experiment with new features, bug fixes, and upgrades.

#### 5.1.1. Key Characteristics:

- **Similar to Production**: It should closely resemble the production environment to ensure accurate testing (e.g., same server configurations, databases, etc.).
- **Testing Ground**: Used for integration testing, performance testing, security testing, and user acceptance testing (UAT).
- **Rolling Back**: If an issue is detected in the test environment, changes can be reverted without affecting live users.

#### 5.1.2. Best Practices:

- Maintain an isolated database for testing to prevent any corruption of real data.
- Simulate load and stress conditions to test performance in a safe environment.
- Implement Continuous Integration/Continuous Deployment (CI/CD) pipelines to automatically deploy new changes to the test environment.

### 5.2. Production Environment

The **production environment** is the live environment where the application is accessible to real users. It hosts the final, stable version of the application that is intended for everyday use.

### 5.2.1. Key Characteristics:

- **Stability and Performance**: The production environment needs to be optimized for reliability, security, and high performance.
- **Security**: Production environments handle real user data, so they must comply with security measures like encryption, firewalls, intrusion detection, and logging.
- **Monitoring**: Applications running in production should be continuously monitored for performance, security, and availability.
- **Scalability**: Should be designed to scale based on user demand, either vertically (adding more resources to a single machine) or horizontally (adding more machines or instances).

### 5.2.2. Best Practices:

- Avoid making direct changes to the production environment without prior testing in staging or test environments.
- Use automated monitoring and alerting tools like Prometheus, Datadog, or Grafana to detect any performance or security issues in real-time.
- Implement rollback strategies in case a deployment in production causes unexpected issues.

## 5.3. Key Differences Between Test and Production Environments

### 5.3.1. Purpose:

- Test Environment: Used for testing code, new features, or performance improvements before going live.
- Production Environment: The live environment used by end-users.

### 5.3.2. Data:

- Test Environment: Uses dummy or sanitized data to avoid corruption of real data.

- Production Environment: Handles real user data, requiring more stringent security measures.

### 5.3.3. Risk:

- Test Environment: Low risk, as any errors or crashes do not affect end-users.
- Production Environment: High risk, as downtime or bugs can impact real users and potentially damage the company's reputation.

### 5.3.4. Access:

- Test Environment: Usually only accessible to developers and testers.
- Production Environment: Accessible to real users, and often includes security measures to restrict admin-level access.

## 5.4. Additional Considerations

### 5.4.1. Version Control:

Using tools like Git to manage versions between test and production environments is critical for tracking changes and ensuring that only stable, tested versions reach production.

### 5.4.2. Blue-Green Deployments:

A deployment strategy where two environments (blue and green) are used in parallel. One environment serves the live production, while the other is updated and tested, reducing downtime during updates.

By choosing the right installation method and carefully managing both test and production environments, you can ensure smoother deployments and minimize risk.

## Web Links

In this section, you will find the relevant links of interest necessary to expand and explore the contents of the unit.

- MDN Web Docs - Mozilla provides a comprehensive guide on server-side web development, including types of web applications and their characteristics.
- Martin Fowler - Articles from Martin Fowler's blog, particularly on monolithic vs. microservices architecture.
- Codecademy - Offers an overview of frontend and backend development, explaining how these components fit together.
- Geeks for Geeks - Describes different layers in a web application with examples.
- AWS Web Application Hosting - Amazon Web Services documentation on hosting web applications.
- Google Cloud Web App Deployment - A guide from Google Cloud on deploying scalable web applications.
- Oracle's Java EE Documentation - Detailed information about `web.xml` and deployment descriptors in Java.
- Docker Documentation - Official Docker documentation on `docker-compose.yml` and container orchestration.
- CI/CD Pipelines - A beginner's guide to Continuous Integration and Continuous Deployment by Atlassian.
- OWASP - A great resource for understanding web application security best practices.
- MDN Web Docs - Web Servers - A beginner-friendly explanation of web servers.
- Difference Between Web Server and Application Server - A detailed comparison of the two types of servers.

- Apache HTTP Server Documentation - Official documentation for the Apache web server.
- Nginx Documentation - Comprehensive information about Nginx as a web server and reverse proxy.
- HTTP/1.1 Documentation - W3C - The official documentation of the HTTP/1.1 protocol.
- Introduction to WebSockets - MDN - A good resource to learn about the WebSocket protocol for real-time communication.
- What is HTTPS? - A clear explanation of HTTPS and SSL certificates from Let's Encrypt.
- OWASP Authentication Cheat Sheet - A comprehensive guide on secure authentication practices.
- Token-Based Authentication: An Introduction - A guide to JWT (JSON Web Tokens), commonly used for token-based authentication.
- OAuth 2.0 and OpenID Connect - A great resource from Auth0 on OAuth 2.0 and OpenID Connect for third-party authentication.
- Apache Tomcat Documentation - The official documentation for Apache Tomcat, a popular Java-based web application server.
- Node.js Documentation - Documentation for Node.js, used for building scalable web applications.
- ASP.NET Core Documentation - Microsoft's official documentation for ASP.NET Core.
- Nginx as a Reverse Proxy - An overview of Nginx as a reverse proxy and load balancer.
- Load Balancing 101 - AWS documentation on load balancing, explaining how it distributes traffic for better performance.

- Web Caching Basics - A beginner-friendly explanation of caching in web applications.
- Server Basics: What is a Dedicated Server? - Overview of dedicated physical servers and their use cases.
- What is a Virtual Machine? - VMware - A comprehensive explanation of VMs and how they work.
- Virtualization Guide - Red Hat - Red Hat's guide to virtualization, including the role of virtual machines.
- What is Docker? - Docker Documentation - A beginner's guide to Docker containers.
- Kubernetes Documentation - The official Kubernetes documentation for container orchestration.
- AWS EC2 Documentation - Learn about Amazon EC2, virtual servers in the AWS Cloud.
- Google Cloud Deployment Guides - Comprehensive guides on deploying applications using Google Cloud.
- Microsoft Azure Virtual Machines - Azure's guide on using virtual machines in the cloud.
- DevOps Best Practices: Staging and Production Environments - A detailed article on the differences and best practices for staging and production environments.
- Best Practices for Testing in Staging Environments - An article by Martin Fowler discussing strategies and best practices for testing in pre-production environments.
- Blue-Green Deployment Strategy - AWS - AWS documentation on blue-green deployment strategy.
- What is CI/CD? - Atlassian - Learn about how Continuous Integration and Continuous Deployment environments work.

- Kubernetes Patterns and Best Practices - An in-depth look into deploying scalable applications with Kubernetes.
- The Twelve-Factor App - A methodology for building modern web applications that can be deployed efficiently on various platforms, including cloud environments.