Advanced Backend Development: Server Architecture & Networking

Table of Contents

---

1. Servers and Server-Side Communication

What is a Server?

A **server** is a computer system or application that provides services, resources, or data to other computers (clients) over a network. In backend development context:

- **Physical Server**: Hardware machine dedicated to running server applications
- **Virtual Server**: Software-based server running on virtualized hardware (like AWS EC2)
- **Application Server**: Software that handles application logic and serves dynamic content
- **Web Server**: Handles HTTP requests and serves static/dynamic web content

Server-Side Communication Types

1. Synchronous Communication

```
Client Request → Server Processing → Response → Client Receives
```

- **HTTP/HTTPS**: Request-response model
- **gRPC**: High-performance RPC framework
- **GraphQL**: Query language for APIs

2. Asynchronous Communication

```
Client → Message Queue → Server processes when available
```

- **Message Queues**: RabbitMQ, Apache Kafka
- **WebSockets**: Real-time bidirectional communication
- **Server-Sent Events (SSE)**: Server pushes data to client

3. Inter-Service Communication

- **REST APIs**: Stateless communication between microservices
- **Message Brokers**: Pub/Sub patterns for decoupled services
- **Service Mesh**: Infrastructure layer for service-to-service communication

---

2. Client-Server Architecture

Traditional Client-Server Model

```
[Client] ←→ [Network] ←→ [Server]
```

Client Responsibilities:

- User interface presentation
- Input validation
- Request formatting
- Response handling

Server Responsibilities:

- Business logic processing
- Data storage and retrieval
- Authentication and authorization
- Resource management

Modern Architectures

1. Three-Tier Architecture

```
[Presentation Tier] ←→ [Logic Tier] ←→ [Data Tier]
    (Web Browser)      (App Server)     (Database)
```

2. Microservices Architecture

```
[Client] ←→ [API Gateway] ←→ [Service A] ←→ [Database A]
                    ←→ [Service B] ←→ [Database B]
                    ←→ [Service C] ←→ [Cache]
```

3. Serverless Architecture

```
[Client] ←→ [CDN] ←→ [Lambda Functions] ←→ [Managed Services]
```

Communication Patterns

1. **Request-Response**: Traditional HTTP model
2. **Publish-Subscribe**: Event-driven architecture
3. **Message Queues**: Asynchronous processing
4. **Event Streaming**: Real-time data processing

---

3. HTTP Protocol and TCP/IP Stack

HTTP Protocol Overview

HTTP (HyperText Transfer Protocol) is an application-layer protocol for distributed, collaborative, hypermedia information systems.

HTTP Request Structure

```
POST /api/users HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: Bearer token123
Content-Length: 45

{"name": "John", "email": "john@example.com"}
```

HTTP Response Structure

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/users/123
Content-Length: 67

{"id": 123, "name": "John", "email": "john@example.com"}
```

HTTP Methods (Verbs)

- **GET**: Retrieve resource (idempotent)
- **POST**: Create new resource
- **PUT**: Update/replace entire resource (idempotent)
- **PATCH**: Partial update
- **DELETE**: Remove resource (idempotent)
- **HEAD**: Get headers only
- **OPTIONS**: Get allowed methods

HTTP Status Codes

- **1xx**: Informational responses
- **2xx**: Success (200 OK, 201 Created, 204 No Content)
- **3xx**: Redirection (301 Moved, 304 Not Modified)
- **4xx**: Client errors (400 Bad Request, 401 Unauthorized, 404 Not Found)
- **5xx**: Server errors (500 Internal Server Error, 502 Bad Gateway)

How HTTP Works on TCP

TCP/IP Stack Interaction

```
Application Layer    [HTTP Request: GET /api/data]
Transport Layer      [TCP Segment: Port 80, Sequence Numbers]
Network Layer        [IP Packet: Source/Destination IP]
Data Link Layer      [Ethernet Frame: MAC Addresses]
Physical Layer       [Electrical/Optical Signals]
```

TCP Connection Process

1. **Three-Way Handshake**:
   ```
   Client → SYN → Server
   Client ← SYN-ACK ← Server
   Client → ACK → Server
   ```
2. **HTTP Request/Response**:
   ```
   Client → HTTP Request → Server
   Client ← HTTP Response ← Server
   ```
3. **Connection Termination**:

```
Client → FIN → Server
Client ← FIN-ACK ← Server
Client → ACK → Server
```

HTTP/2 and HTTP/3 Improvements

HTTP/2 Features

- **Multiplexing**: Multiple requests over single connection
- **Server Push**: Server initiates resource transfer
- **Header Compression**: HPACK algorithm
- **Binary Protocol**: More efficient than text-based HTTP/1.1

HTTP/3 Features

- **QUIC Protocol**: UDP-based transport
- **Reduced Latency**: Eliminates head-of-line blocking
- **Built-in Encryption**: TLS 1.3 integrated

---

4. OSI Layer Model

Seven Layers of OSI Model

Layer 7: Application Layer

Purpose: Provides network services directly to applications **Protocols**: HTTP, HTTPS, FTP, SMTP, DNS, SSH **Example**: Web browser making HTTP request

```python
# Application Layer - HTTP Request
import requests
response = requests.get('https://api.example.com/data')
```

Layer 6: Presentation Layer

Purpose: Data translation, encryption, compression **Functions**: SSL/TLS encryption, data compression, character encoding **Example**: HTTPS encryption, JPEG compression

Layer 5: Session Layer

Purpose: Manages sessions between applications **Functions**: Session establishment, maintenance, termination **Example**: Database connections, RPC sessions

Layer 4: Transport Layer

Purpose: Reliable data transfer between endpoints **Protocols**: TCP (reliable), UDP (fast) **Functions**: Segmentation, flow control, error detection

```
TCP Header:
Source Port | Dest Port | Sequence # | Ack # | Flags | Window | Checksum
```

Layer 3: Network Layer

Purpose: Routing packets across networks **Protocols**: IP, ICMP, OSPF, BGP **Functions**: Logical addressing, path determination

```
IP Header:
Version | Header Length | Type of Service | Total Length
Identification | Flags | Fragment Offset | TTL | Protocol
```

Layer 2: Data Link Layer

Purpose: Node-to-node data transfer **Protocols**: Ethernet, WiFi, PPP **Functions**: Framing, error detection, MAC addressing

Layer 1: Physical Layer

Purpose: Transmission of raw bits **Components**: Cables, switches, wireless signals **Functions**: Electrical/optical signal transmission

TCP/IP Model vs OSI Model

```
OSI Model        TCP/IP Model
Application  ↗
Presentation ↗    Application
Session      ↗
Transport        Transport
Network          Internet
Data Link    ↘    Network Access
Physical     ↘
```

---

5. Server Fundamentals

What Makes a Computer a Server?

A server is defined by its **role and configuration**, not just hardware:

1. **Server Software**: Web servers (Apache, Nginx), application servers (Node.js, Tomcat)

2. **Network Configuration**: Static IP, open ports, proper firewall rules
3. **Resource Allocation**: CPU, RAM, storage optimized for concurrent requests
4. **Operating System**: Server-grade OS (Linux distributions, Windows Server)

Server Types by Function

1. Web Server

```nginx
# Nginx configuration
server {
    listen 80;
    server_name example.com;

    location / {
        proxy_pass http://localhost:3000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

2. Application Server

```javascript
// Node.js Express server
const express = require('express');
const app = express();

app.get('/api/health', (req, res) => {
    res.json({ status: 'healthy', timestamp: new Date() });
});

app.listen(3000, () => {
    console.log('Server running on port 3000');
});
```

3. Database Server

```sql
```

```sql
-- PostgreSQL server configuration
CREATE DATABASE ecommerce;
CREATE USER api_user WITH ENCRYPTED PASSWORD 'secure_password';
GRANT ALL PRIVILEGES ON DATABASE ecommerce TO api_user;
```

Server Performance Considerations

1. Concurrency Models

- **Thread-per-request**: Traditional Java servlets
- **Event-driven**: Node.js, Nginx
- **Actor model**: Erlang/Elixir
- **Async/await**: Python asyncio, .NET async

2. Caching Strategies

- **In-memory**: Redis, Memcached
- **CDN**: CloudFlare, AWS CloudFront
- **Application-level**: Local caches
- **Database**: Query result caching

3. Load Balancing

```
          [Load Balancer]
         /      |      \
    [Server 1] [Server 2] [Server 3]
         \      |      /
          [Shared Database]
```

---

6. IP Addressing: Public, Static, and Dynamic IPs

IP Address Fundamentals

IPv4 Address Structure

```
192.168.1.100
|  |  | |
|  |  | Host (1-254)
|  |  Subnet (0-255)
|  Network (0-255)
Network Class (A, B, C)
```

Private IP Ranges (RFC 1918)

- **Class A**: 10.0.0.0 - 10.255.255.255 (16.7M addresses)
- **Class B**: 172.16.0.0 - 172.31.255.255 (1M addresses)
- **Class C**: 192.168.0.0 - 192.168.255.255 (65K addresses)

Public IP vs Private IP

Public IP Address

- **Globally unique** address assigned by ISP
- **Routable** on the internet
- **Scarce resource** (IPv4 exhaustion)
- **Required** for direct internet communication

Private IP Address

- **Locally unique** within private network
- **Not routable** on public internet
- **Requires NAT** for internet access
- **Abundant** within private networks

Static vs Dynamic IP

Static IP Address

Characteristics:

- Permanently assigned to device
- Doesn't change over time
- Must be manually configured
- More expensive from ISPs

Use Cases:

- Web servers requiring consistent access
- Email servers needing reliable MX records
- VPN endpoints
- Remote access solutions

Configuration Example:

```bash
```

```
# Linux static IP configuration
sudo nano /etc/netplan/01-network-manager-all.yaml

network:
 version: 2
 ethernets:
  eth0:
   dhcp4: false
   addresses:
    - 192.168.1.100/24
   gateway4: 192.168.1.1
   nameservers:
    addresses: [8.8.8.8, 8.8.4.4]
```

Dynamic IP Address

Characteristics:

- Automatically assigned by DHCP server
- Can change when lease expires
- Automatically configured
- Standard for most consumer connections

DHCP Process:

1. **DHCP Discover**: Client broadcasts request
2. **DHCP Offer**: Server offers IP address
3. **DHCP Request**: Client requests specific IP
4. **DHCP ACK**: Server confirms assignment

Network Address Translation (NAT)

NAT allows multiple devices with private IPs to share a single public IP:

```
Private Network       NAT Router        Internet
192.168.1.10:3000 → 203.0.113.1:8080 → Server
192.168.1.11:3001 → 203.0.113.1:8081 → Server
192.168.1.12:3002 → 203.0.113.1:8082 → Server
```

---

7. AWS EC2 IP Configuration

EC2 Instance IP Types

1. Private IP Address

- **Always assigned** to every EC2 instance
- **Persistent** throughout instance lifecycle
- **VPC subnet** determines IP range
- **Internal communication** within VPC

```bash
# View private IP from inside EC2 instance
curl http://169.254.169.254/latest/meta-data/local-ipv4
# Output: 172.31.32.45
```

2. Public IP Address

- **Dynamically assigned** by default
- **Changes** when instance stops/starts
- **Released** when instance terminates
- **Free** for running instances

```bash
# View public IP from inside EC2 instance
curl http://169.254.169.254/latest/meta-data/public-ipv4
# Output: 54.123.45.67
```

3. Elastic IP Address (Static Public IP)

- **Static public IP** that doesn't change
- **Persistent** across instance lifecycle
- **Can be reassigned** to different instances
- **Charged** when not associated with running instance

EC2 IP Configuration Example

Creating EC2 with Static Configuration

```bash
```

```
# AWS CLI - Launch EC2 instance
aws ec2 run-instances \
  --image-id ami-0abcdef1234567890 \
  --instance-type t3.micro \
  --key-name my-key-pair \
  --subnet-id subnet-12345678 \
  --associate-public-ip-address

# Allocate Elastic IP
aws ec2 allocate-address --domain vpc

# Associate Elastic IP with instance
aws ec2 associate-address \
  --instance-id i-1234567890abcdef0 \
  --allocation-id eipalloc-12345678
```

VPC and Subnet Configuration

```json
{
  "VPC": {
    "CidrBlock": "10.0.0.0/16",
    "Subnets": [
      {
        "Public": "10.0.1.0/24",
        "InternetGateway": "attached"
      },
      {
        "Private": "10.0.2.0/24",
        "NATGateway": "10.0.1.100"
      }
    ]
  }
}
```

EC2 Network Security

Security Groups (Stateful Firewall)

```bash
```

```
# Allow HTTP traffic
aws ec2 authorize-security-group-ingress \
  --group-id sg-12345678 \
  --protocol tcp \
  --port 80 \
  --cidr 0.0.0.0/0

# Allow HTTPS traffic
aws ec2 authorize-security-group-ingress \
  --group-id sg-12345678 \
  --protocol tcp \
  --port 443 \
  --cidr 0.0.0.0/0

# Allow SSH from specific IP
aws ec2 authorize-security-group-ingress \
  --group-id sg-12345678 \
  --protocol tcp \
  --port 22 \
  --cidr 203.0.113.0/32
```

Network ACLs (Stateless Firewall)

```json
{
  "NetworkAcl": {
    "Rules": [
      {
        "RuleNumber": 100,
        "Protocol": "tcp",
        "RuleAction": "allow",
        "PortRange": {"From": 80, "To": 80},
        "CidrBlock": "0.0.0.0/0"
      }
    ]
  }
}
```

---

8. Port Forwarding

Port Forwarding Fundamentals

Port forwarding redirects communication requests from one address and port number combination to another while packets traverse a network gateway (router).

Types of Port Forwarding

1. Static Port Forwarding

Permanent mapping of external port to internal IP and port:

```
External Request: 203.0.113.1:8080
    ↓
Router NAT Table: 8080 → 192.168.1.100:3000
    ↓
Internal Server: 192.168.1.100:3000
```

2. Dynamic Port Forwarding

Temporary mappings created by outbound connections:

```
Internal Request: 192.168.1.100:random_port → Internet
Router creates temporary mapping for response
```

Router Configuration Examples

Consumer Router (Web Interface)

```
Port Forward Settings:
Service Name: Web Server
External Port: 80
Internal IP: 192.168.1.100
Internal Port: 3000
Protocol: TCP
Enable: ✓
```

Professional Router (CLI)

```bash
# Cisco Router Configuration
configure terminal
ip nat inside source static tcp 192.168.1.100 3000 interface FastEthernet0/0 80
ip nat inside source static tcp 192.168.1.100 443 interface FastEthernet0/0 443
exit
```

Linux iptables Configuration

```bash
```

```
# Enable IP forwarding
echo 1 > /proc/sys/net/ipv4/ip_forward

# Port forwarding rule
iptables -t nat -A PREROUTING -p tcp --dport 8080 -j DNAT --to-destination 192.168.1.100:3000
iptables -A FORWARD -p tcp -d 192.168.1.100 --dport 3000 -j ACCEPT
iptables -t nat -A POSTROUTING -j MASQUERADE
```

Port Forwarding Security Considerations

1. Firewall Rules

bash

```bash
# Only allow specific source IPs
iptables -A INPUT -p tcp --dport 3000 -s 203.0.113.0/24 -j ACCEPT
iptables -A INPUT -p tcp --dport 3000 -j DROP
```

2. Application-Level Security

javascript

```javascript
// Express.js with IP whitelisting
const express = require('express');
const app = express();

const allowedIPs = ['203.0.113.10', '203.0.113.20'];

app.use((req, res, next) => {
  const clientIP = req.ip || req.connection.remoteAddress;
  if (!allowedIPs.includes(clientIP)) {
    return res.status(403).json({ error: 'Access denied' });
  }
  next();
});
```

3. Fail2Ban Configuration

ini

```
# /etc/fail2ban/jail.local
[custom-app]
enabled = true
port = 3000
filter = custom-app
logpath = /var/log/myapp.log
maxretry = 5
bantime = 3600
```

---

9. Hosting APIs: Local to Public Access

Scenario: Making Local API Publicly Accessible

You have an API running on your PC (192.168.1.50:3000) and want internet users to access it.

Method 1: Router Port Forwarding

Step 1: Configure Static IP for Your PC

```bash
# Windows - PowerShell
New-NetIPAddress -InterfaceIndex 12 -IPAddress 192.168.1.50 -PrefixLength 24 -DefaultGateway 192.168.1.1

# Linux
sudo nano /etc/netplan/01-network-manager-all.yaml
network:
  version: 2
  ethernets:
    enp0s3:
      dhcp4: false
      addresses: [192.168.1.50/24]
      gateway4: 192.168.1.1
      nameservers:
        addresses: [8.8.8.8]
```

Step 2: Router Configuration

```
Router Admin Panel → Port Forwarding
External Port: 8080
Internal IP: 192.168.1.50
Internal Port: 3000
Protocol: TCP
```

Step 3: Firewall Configuration

```bash
# Windows Firewall
netsh advfirewall firewall add rule name="API Server" dir=in action=allow protocol=TCP localport=3000

# Linux UFW
sudo ufw allow 3000/tcp
sudo ufw enable
```

Method 2: Dynamic DNS for Changing Public IP

Setup Dynamic DNS Service

```bash
# Install ddclient for automatic IP updates
sudo apt install ddclient

# Configuration
sudo nano /etc/ddclient.conf
protocol=dyndns2
use=web, web=checkip.dyndns.com/, web-skip='IP Address'
server=members.dyndns.org
login=your-username
password=your-password
your-domain.dyndns.org
```

Method 3: Cloud Reverse Proxy

Using Cloudflare Tunnels

```bash
```

```
# Install cloudflared
wget https://github.com/cloudflare/cloudflared/releases/latest/download/cloudflared-linux-amd64.deb
sudo dpkg -i cloudflared-linux-amd64.deb

# Authenticate
cloudflared tunnel login

# Create tunnel
cloudflared tunnel create my-api-tunnel

# Configure tunnel
nano ~/.cloudflared/config.yml
tunnel: your-tunnel-id
credentials-file: /home/user/.cloudflared/tunnel-credentials.json

ingress:
  - hostname: myapi.example.com
    service: http://localhost:3000
  - service: http_status:404

# Run tunnel
cloudflared tunnel run my-api-tunnel
```

API Security for Public Access

1. Authentication & Authorization

```javascript
```

```javascript
// JWT-based authentication
const jwt = require('jsonwebtoken');

const authenticate = (req, res, next) => {
  const token = req.header('Authorization')?.replace('Bearer ', '');

  if (!token) {
    return res.status(401).json({ error: 'No token provided' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    res.status(401).json({ error: 'Invalid token' });
  }
};

app.use('/api/protected', authenticate);
```

2. Rate Limiting

```javascript
const rateLimit = require('express-rate-limit');

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests per windowMs
  message: 'Too many requests, please try again later'
});

app.use('/api/', limiter);
```

3. CORS Configuration

```javascript
```

```
const cors = require('cors');

const corsOptions = {
  origin: ['https://myapp.com', 'https://admin.myapp.com'],
  methods: ['GET', 'POST', 'PUT', 'DELETE'],
  allowedHeaders: ['Content-Type', 'Authorization'],
  credentials: true
};

app.use(cors(corsOptions));
```

---

10. Tunneling Solutions

What is Tunneling?

Tunneling creates a secure connection between networks by encapsulating one network protocol within another. It allows you to expose local services to the internet without port forwarding.

Types of Tunneling

1. SSH Tunneling

Create secure tunnel using SSH protocol:

Local Port Forwarding

```bash
# Forward local port 8080 to remote server's port 80
ssh -L 8080:localhost:80 user@remote-server.com

# Access via http://localhost:8080
```

Remote Port Forwarding

```bash
# Make local service accessible from remote server
ssh -R 8080:localhost:3000 user@remote-server.com

# Remote server can access your local API via localhost:8080
```

Dynamic Port Forwarding (SOCKS Proxy)

```bash
```

```bash
# Create SOCKS proxy on local port 1080
ssh -D 1080 user@remote-server.com
```

2. Reverse SSH Tunnel for Public Access

```bash
# On your local machine
ssh -R 8080:localhost:3000 user@your-vps.com

# Public access via: http://your-vps.com:8080
```

Modern Tunneling Solutions

1. ngrok (Popular Choice)

```bash
# Install ngrok
npm install -g ngrok

# Expose local port 3000
ngrok http 3000

# Output:
# Session Status: online
# Forwarding: https://abc123.ngrok.io -> http://localhost:3000
```

2. Cloudflare Tunnel (Enterprise Grade)

```yaml
# ~/.cloudflared/config.yml
tunnel: my-tunnel-id
credentials-file: ~/.cloudflared/credentials.json

ingress:
  - hostname: api.mydomain.com
    service: http://localhost:3000
  - hostname: admin.mydomain.com
    service: http://localhost:3001
  - service: http_status:404
```

3. Serveo (Simple SSH-based)

```bash
# No installation required
ssh -R 80:localhost:3000 serveo.net

# Output: https://abc123.serveo.net -> http://localhost:3000
```

4. LocalTunnel

```bash
# Install and use
npm install -g localtunnel
lt --port 3000

# Output: your url is: https://abc-123.loca.lt
```

Advanced Tunneling with Custom VPS

Setup WireGuard VPN Tunnel

```bash
# Server configuration
[Interface]
PrivateKey = server-private-key
Address = 10.0.0.1/24
ListenPort = 51820

[Peer]
PublicKey = client-public-key
AllowedIPs = 10.0.0.2/32

# Client configuration
[Interface]
PrivateKey = client-private-key
Address = 10.0.0.2/24

[Peer]
PublicKey = server-public-key
Endpoint = your-vps-ip:51820
AllowedIPs = 0.0.0.0/0
```

Production-Ready Tunneling Architecture

Using Nginx Reverse Proxy

```nginx
# /etc/nginx/sites-available/api-tunnel
server {
    listen 80;
    server_name api.yourdomain.com;

    location / {
        proxy_pass http://localhost:8080;  # Tunnel endpoint
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

SSL Certificate with Let's Encrypt

```bash
# Install certbot
sudo apt install certbot python3-certbot-nginx

# Get certificate
sudo certbot --nginx -d api.yourdomain.com

# Auto-renewal
sudo crontab -e
0 12 * * * /usr/bin/certbot renew --quiet
```

Security Considerations for Tunneling

1. Authentication

```bash
# ngrok with auth token
ngrok authtoken your-auth-token
ngrok http 3000 --auth="username:password"
```

2. IP Whitelisting

```bash
```

```
# Cloudflare tunnel with access control
cloudflared access application create \
  --domain api.yourdomain.com \
  --name "API Access" \
  --allowed-email user@company.com
```

3. Network Monitoring

```bash
# Monitor tunnel connections
netstat -an | grep :3000
ss -tuln | grep :3000

# Log tunnel access
tail -f /var/log/nginx/access.log
```

Conclusion

This advanced guide covers the fundamental concepts of backend development, from basic client-server communication to complex tunneling solutions. Key takeaways:

1. **Server Architecture**: Understanding different communication patterns and architectural models
2. **Network Protocols**: HTTP over TCP/IP and the OSI model layers
3. **IP Addressing**: Public vs private, static vs dynamic, and cloud configurations
4. **Port Forwarding**: Making local services accessible from the internet
5. **Tunneling**: Secure alternatives to port forwarding for development and production

For production systems, always prioritize security through proper authentication, encryption, monitoring, and access controls. Consider using cloud services like AWS, Azure, or Google Cloud for scalable and secure deployments.

Next Steps for Learning

- Practice setting up different server configurations
- Implement security measures for production APIs
- Explore container technologies (Docker, Kubernetes)
- Study load balancing and high availability patterns
- Learn about monitoring and observability tools