1. Introduction to Service-Oriented Architecture (SOA)

Service-Oriented Architecture (SOA) is an architectural approach that organizes software into **modular services** where each service:

- Handles a **specific business function**
- Communicates over a network using **standard protocols** (HTTP, SOAP)
- Is **loosely coupled** and self-contained
- Promotes **reuse, scalability, and platform independence**

Ex: Restaurant System

Think of SOA like a restaurant operation:

- **Waiter** = User Interface Service (takes orders, serves customers)
- **Chef** = Business Logic Service (prepares food, manages recipes)
- **Cashier** = Payment Service (handles transactions)
- **Manager** = Orchestration Service (coordinates everything)

Each role performs distinct functions without knowing others' internal workings, yet they collaborate seamlessly.

Why SOA Emerged: Breaking the Monolith

Problems with Monolithic Architecture

- **Tightly coupled code** → Small changes risk breaking entire system
- **Full redeployment** required for any update → Slow release cycles
- **Single point of failure** → One bug can crash everything
- **Difficult to scale** → Must scale entire application, not just bottlenecks
- **Technology lock-in** → Entire system uses same tech stack
- **Team dependencies** → Teams must coordinate all changes

SOA Benefits

- **Easier maintenance** by isolating services
- **Reusability** across different applications
- **Independent scalability** of individual services
- **Technology flexibility** for different teams
- **Fault isolation** improves overall reliability
- **Faster time to market** with parallel development
- **Better team autonomy** and productivity

2. Evolution of Web Architectures(Web Evolution Timeline)

Era	Period	Key Characteristics	Examples
Web 1.0	1990s-early 2000s	Static, read-only pages	Basic HTML sites, company brochures
Web 2.0	Mid-2000s-2015	Dynamic, user-generated content	Facebook, YouTube, blogs, wikis
Web 3.0	2015-present	Semantic, AI-friendly, decentralized	Voice assistants, IoT, blockchain
Web 4.0	Emerging	AI-driven, context-aware, IoT-integrated	Smart cities, predictive systems

Architecture Evolution Path

1. **Monolithic** (1990s)- All components bundled together, Simple but inflexible
2. **Three-Tier** (Late 1990s) -Separates presentation, logic, and data layers,Better organization but still coupled
3. **SOA** (Early 2000s) -Modular services communicating over network,Business-focused, reusable components
4. **Microservices** (2010s) - Fine-grained, independently deployable services,Cloud-native, highly scalable
5. **Serverless** (2015+) - Function-based, event-driven architecture,Pay-per-use, zero server management

3. SOA Core Principles

1. Loose Coupling : Services operate independently with minimal dependencies

Benefits:

- Changes in one service don't break others
- Services can be developed by different teams
- Easier testing and debugging

Example: Payment service doesn't need to know how inventory management works

2. Service Reusability : Design services to be generic and reusable across applications

Benefits:

- Reduces development time and costs
- Consistent business logic across systems
- Easier maintenance

Example: Authentication service used by web app, mobile app, and API

3. Service Contract : Clear specification of service inputs, outputs, and behavior

Components:

- **Interface definition** (API endpoints)
- **Data formats** (JSON, XML schemas)
- **Error handling** specifications
- **Service level agreements** (SLAs)

Benefits:

- Enables parallel development
- Prevents breaking changes
- Clear expectations for consumers

4. Service Abstraction : Hide internal implementation details from service consumers

Benefits:

- Improved security
- Flexibility to change internal logic
- Simplified integration

Example: Weather service API hides whether data comes from satellites, weather stations, or third-party APIs

5. Service Discoverability : Services must be easily found and understood via registries and documentation

Requirements:

- **Service registry** for cataloging services
- **Comprehensive documentation** with examples
- **Versioning strategy** for updates
- **Metadata** describing capabilities

Benefits:

- Prevents duplicate development
- Improves integration efficiency
- Reduces onboarding time

6. Statelessness : Each service request contains all necessary information; no session state stored

Benefits:

- Easy horizontal scaling
- Simple load balancing
- Improved reliability and fault tolerance
- Simplified retry mechanisms

Example: REST API where each request includes authentication token

7. Interoperability : Services communicate across different platforms, languages, and systems

Requirements:

- **Standard protocols** (HTTP, SOAP)
- **Common data formats** (JSON, XML)
- **Platform-neutral interfaces**

Benefits:

- Legacy system integration
- Mixed technology environments
- Vendor independence

Principle Application Exercise

Scenario	Primary SOA Principle	Explanation
Payment service used across web, mobile, and desktop apps	**Reusability**	Same service serves multiple clients
Services built in Java, .NET, Python communicate seamlessly	**Interoperability**	Cross-platform compatibility
API documentation hides internal database schema	**Abstraction**	Implementation details hidden
Load balancer distributes requests to any available node	**Statelessness**	No session affinity required
Swagger documentation helps teams find and use services	**Discoverability**	Easy service location and understanding
Order service works independently of inventory service	**Loose Coupling**	Minimal interdependencies
Service defines exact input/output JSON schemas	**Service Contract**	Clear interface specification

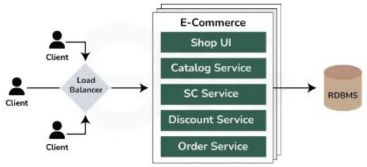
4. Architecture Patterns Comparison

Monolithic Architecture

[Client Browser] → [Web Server (UI + Business Logic + Data Access)] → [Database]

Benefits:

- **Structure**: Single deployable unit containing all functionality
- Simple development and testing initially
- Easy deployment and monitoring
- Good performance (no network calls)
- Straightforward debugging



Drawbacks:

- Difficult to scale specific components
- Technology lock-in for entire application
- Large codebase becomes unwieldy
- Single point of failure
- Long deployment cycles

Best For: Small applications, prototypes, simple business domains

Three-Tier Architecture

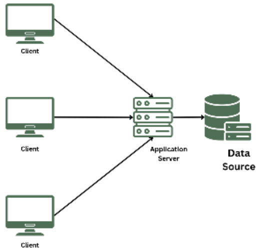
[Presentation Layer] ↔ [Business Logic Layer] ↔ [Data Access Layer]

↓ ↓ ↓

[Web Browser] [App Server] [Database]

Benefits:

- **Structure**: Logical separation into presentation, business, and data layers
- Better separation of concerns
- Easier maintenance than monolith
- Some reusability of business logic
- Improved security through layering



Limitations:

- Still deployed as single unit
- Limited scalability options
- Tight coupling between layers

Best For: Medium-sized applications with clear layer boundaries

Service-Oriented Architecture (SOA)

[User Interface] → [Service Layer: Auth Service | Order Service | User Service] → [Databases]

SERVICES
PLATFORM
BEST PRACTICES
SERVICE ORIENTED ARCHITECTURE (SOA)
USERS
PROCESS

Benefits:

- **Structure**: Business logic organized into reusable services
- Modular, reusable business components
- Better team organization and autonomy
- Easier integration with external systems
- Platform and technology flexibility
- Improved fault isolation

Challenges:

- Services can become "mini-monoliths"
- Requires careful service design and governance
- Network communication overhead
- More complex testing and debugging

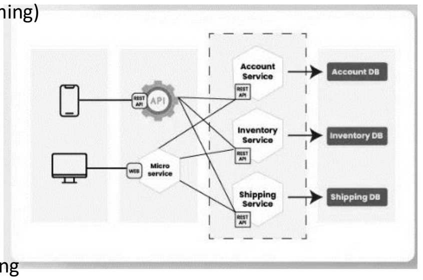
Best For: Enterprise applications, complex business domains, systems requiring integration

Microservices Architecture

```
[UI] → [API Gateway] → [Auth Service | Product Service | Order Service | Email Service]
                              ↓              ↓               ↓              ↓
                          [Auth DB]     [Product DB]    [Order DB]    [Email Queue]
```

Benefits:

- **Structure**: Fine-grained, independently deployable services with dedicated databases
- Rapid, independent development and deployment
- Technology diversity (polyglot programming)
- Improved fault tolerance and resilience
- Fine-grained scalability
- Better alignment with DevOps practices



Challenges:

- High operational complexity
- Network latency and reliability issues
- Data consistency challenges
- Requires extensive monitoring and logging
- Service mesh complexity

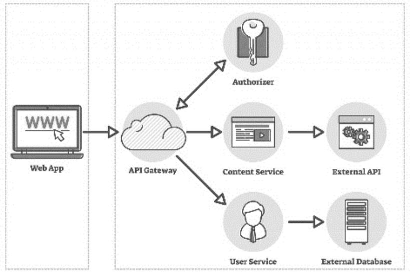
Best For: Large-scale applications, cloud-native systems, organizations with mature DevOps

Serverless Architecture

```
[UI] → [API Gateway] → [Lambda Functions: Login() | GetProduct() | ProcessOrder()]
                                ↓                  ↓                ↓
                   [Cloud Services: DynamoDB |     S3        |     SQS        ]
```

Benefits:

- **Structure**: Event-driven functions with managed cloud services
- Zero server management overhead
- Automatic scaling and high availability
- Pay-per-execution cost model
- Fast development and deployment
- Built-in security and compliance



Limitations:

- Cold start latency issues
- Vendor lock-in concerns
- Limited execution time and resources
- Complex state management
- Difficult local development and testing

Best For: Event-driven applications, APIs with variable load, rapid prototyping

Architecture Comparison Matrix

Aspect	Monolith	Three-Tier	SOA	Microservices	Serverless
Complexity	Low	Medium	Medium-High	High	Medium
Scalability	Limited	Moderate	Good	Excellent	Automatic
Development Speed	Fast (initially)	Moderate	Moderate	Fast (mature teams)	Very Fast
Operational Overhead	Low	Low	Medium	High	Minimal
Technology Flexibility	None	Limited	Good	Excellent	Limited
Team Size	Small	Small-Medium	Medium-Large	Large	Small-Medium
Deployment Frequency	Week/Monthly	Weekly	Daily	Multiple/day	Continuous

5. Web Development Foundations

How Web Communication Works

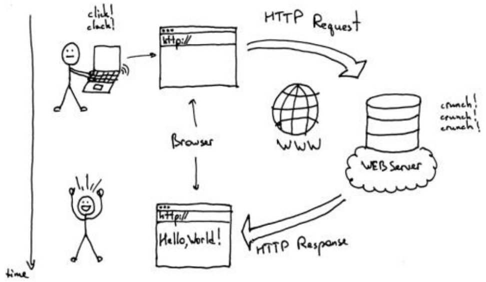
Step-by-Step Process

1. **DNS Lookup**: Browser translates domain name (google.com) to IP address (172.217.14.206)
2. **TCP Connection**: Establishes reliable connection with web server
3. **HTTP Request**: Browser sends request with method, headers, and optional body
4. **Server Processing**: Web server processes request, potentially calling backend services
5. **HTTP Response**: Server sends status code, headers, and content
6. **Rendering**: Browser parses HTML, CSS, and executes JavaScript

Example HTTP Request/Response

```
Request:
GET /api/products HTTP/1.1
Host: api.example.com
Authorization: Bearer token123
Accept: application/json

Response:
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1234
{"products": [...]}
```



Website Building Blocks

01.HTML (HyperText Markup Language): Provides structure and content for web pages

- Uses tags to define elements (<h1>, <p>, <div>)
- Semantic meaning improves accessibility and SEO
- Forms collect user input for backend processing
- **SOA Connection**: HTML forms submit data to service endpoints

02.CSS (Cascading Style Sheets) : Controls visual presentation and layout

- Separates content from presentation
- Responsive design for multiple devices
- Animations and interactive effects
- **Analogy**: Like interior design and decoration for a house structure

JavaScript : Adds interactivity and dynamic behavior

- DOM manipulation (changing page content)
- Event handling (user interactions)
- AJAX calls to backend services
- Client-side validation and processing
- **SOA Bridge**: JavaScript makes HTTP requests to consume services via APIs

6. HTML Fundamentals

Version	Year	Key Features
HTML 1.0	1991	Basic tags, hyperlinks
HTML 2.0	1995	Forms, tables, internationalization
HTML 3.2	1997	Style sheets, scripts, applets
HTML 4.01	1999	Improved accessibility, multimedia
XHTML 1.0	2000	XML-based syntax, stricter rules
HTML5	2012	Semantic elements, multimedia, APIs

🚀 **HTML5 Advantages**

- **Backward compatibility** with all existing web pages
- **Reduced plugin dependency** (native video, audio, graphics)
- **Semantic elements** improve meaning and accessibility
- **Cross-platform consistency** across devices and browsers
- **Enhanced APIs** for local storage, geolocation, offline apps
- **Better error handling** and parsing

HTML Document Structure

```
<!DOCTYPE html>              <!-- Document type declaration -->
<html lang="en">             <!-- Root element with language -->
 <head>                  <!-- Document metadata -->
  <meta charset="UTF-8">      <!-- Character encoding -->
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Page Title</title>    <!-- Browser tab title -->
  <link rel="stylesheet" href="styles.css">  <!-- External CSS -->
  <script src="script.js"></script>        <!-- External JavaScript -->
 </head>
 <body>                  <!-- Visible content -->
  <header>                 <!-- Page header -->
   <nav>Navigation</nav>      <!-- Navigation menu -->
  </header>
  <main>                  <!-- Main content -->
   <article>Content</article> <!-- Self-contained content -->
   <aside>Sidebar</aside>    <!-- Complementary content -->
  </main>
  <footer>                 <!-- Page footer -->
   <p>&copy; 2024 Company</p>
  </footer>
 </body>
</html>
```

Essential HTML Tags

Structural Elements

Tag	Purpose	Example
<header>	Page/section header	<header><h1>Site Title</h1></header>
<nav>	Navigation menu	<nav><ul><li><aref="/">Home</a></li></ul></nav>
<main>	Primary content	<main><article>...</article></main>
<section>	Document section	<section><h2>About Us</h2>...</section>
<article>	Self-contained content	<article><h3>Blog Post</h3>...</article>
<aside>	Sidebar content	<aside><h4>Related Links</h4>...</aside>
<footer>	Page/section footer	<footer><p>Copyright info</p></footer>

Content Elements

Tag	Purpose	Example
<h1> to <h6>	Headings (largest to smallest)	<h1>Main Title</h1>
<p>	Paragraph	<p>This is a paragraph of text.</p>
<a>	Hyperlink	<a href="https://example.com">Link text</a>
<img>	Image	<img src="photo.jpg" alt="Description">
<ul>, <ol>, <li>	Lists	<ul><li>Item 1</li><li>Item 2</li></ul>
<table>, <tr>, <td>	Tables	<table><tr><td>Cell</td></tr></table>

Form Elements

Tag	Purpose	Example
<form>	Form container	<form action="/submit" method="post">
<input>	Input field	<input type="text" name="username">
<textarea>	Multi-line text	<textarea name="message"></textarea>
<select>, <option>	Dropdown	<select><option>Choice 1</option></select>
<button>	Button	<button type="submit">Submit</button>
<label>	Field label	<label for="email">Email:</label>

Formatting Elements

Tag	Purpose	Visual Effect
<strong>	Important text	**Bold** (semantic)
<em>	Emphasized text	*Italic* (semantic)
<b>	Bold text	**Bold** (presentation)
<i>	Italic text	*Italic* (presentation)
<u>	Underlined text	<u>Underlined</u>
<code>	Computer code	Monospace font
<pre>	Preformatted text	Preserves spaces and line breaks
<br>	Line break	Forces new line
<hr>	Horizontal rule	Horizontal line separator

7. CSS Styling : CSS Implementation Methods

1. Inline Styles

```
<p style="color: red; font-size: 16px;">Red text</p>
```

- **Pros**: Quick for single elements, highest specificity
- **Cons**: Not reusable, mixes content with presentation

2. Embedded Styles

```
<head>
  <style>
    p { color: blue; }
    .highlight { background-color: yellow; }
  </style>
</head>
```

- **Pros**: Page-specific styles, faster than external files
- **Cons**: Not reusable across pages, increases HTML size

3. External Stylesheets

```
<link rel="stylesheet" href="styles.css">
/* styles.css */
p { color: green; }
.highlight { background-color: yellow; }
```

- **Pros**: Reusable, cacheable, separates concerns
- **Cons**: Additional HTTP request, external dependency

CSS Selectors

Basic Selectors

Selector	Syntax	Example	Description
Element	element	p { color: red; }	Selects all <p> elements
Class	.classname	.highlight { background: yellow; }	Selects elements with class="highlight"
ID	#idname	#header { font-size: 24px; }	Selects element with id="header"
Universal	*	* { margin: 0; }	Selects all elements

Combination Selectors

Selector	Syntax	Example	Description
Descendant	ancestor descendant	div p { margin: 10px; }	<p> inside <div>
Child	parent > child	ul > li { list-style: none; }	Direct <li> children of <ul>
Adjacent	element + next	h1 + p { margin-top: 0; }	<p> immediately after <h1>
Attribute	[attribute]	[required] { border: 2px solid red; }	Elements with required attribute

Pseudo-classes and Pseudo-elements

Selector	Example	Description
:hover	a:hover { color: blue; }	Element when hovered
:focus	input:focus { outline: 2px solid blue; }	Element when focused
:nth-child()	tr:nth-child(even) { background: #f0f0f0; }	Even table rows
::before	h1::before { content: "★ "; }	Insert content before element
::after	p::after { content: " →"; }	Insert content after element

Color Specification Methods

1. Named Colors

```
color: red;
background-color: lightblue;
border-color: darkgreen;
```

Advantages: Easy to remember and read **Limitations**: Limited color options (~140 names)

2. Hexadecimal Values

```
color: #FF0000;       /* Red */
color: #00FF00;       /* Green */
color: #0000FF;       /* Blue */
color: #FFF;          /* White (shorthand for #FFFFFF) */
color: #333;          /* Dark gray (shorthand for #333333) */
```

Format: #RRGGBB where RR, GG, BB are hexadecimal values (00-FF)

3. RGB Functional Notation

```
color: rgb(255, 0, 0);       /* Red */
color: rgb(0, 255, 0);       /* Green */
color: rgb(100%, 0%, 0%);    /* Red with percentages */
```

Values: 0-255 for integers, 0%-100% for percentages

4. RGBA (with Alpha Transparency)

```
background-color: rgba(255, 0, 0, 0.5);    /* Semi-transparent red */
background-color: rgba(0, 0, 0, 0.8);      /* Semi-transparent black */
```

Alpha: 0 (fully transparent) to 1 (fully opaque)

5. HSL and HSLA

```
color: hsl(0, 100%, 50%);       /* Red */
color: hsl(120, 100%, 50%);     /* Green */
color: hsla(240, 100%, 50%, 0.7); /* Semi-transparent blue */
```

HSL: Hue (0-360°), Saturation (0-100%), Lightness (0-100%)

CSS Box Model

```
.box {
  width: 300px;          /* Content width */
  height: 200px;         /* Content height */
  padding: 20px;         /* Inner spacing */
  border: 2px solid black; /* Border */
  margin: 10px;          /* Outer spacing */
}
```

Total width: width + padding-left + padding-right + border-left + border-right + margin-left + margin-right

Common CSS Properties

Typography

```
font-family: 'Arial', sans-serif;  /* Font family */
font-size: 16px;                 /* Text size */
font-weight: bold;               /* Text weight */
font-style: italic;              /* Text style */
text-align: center;              /* Text alignment */
text-decoration: underline;      /* Text decoration */
line-height: 1.5;                /* Line spacing */
letter-spacing: 2px;             /* Character spacing */
```

Layout and Positioning

```
display: flex;                   /* Flexbox layout */
position: relative;              /* Positioning context */
top: 10px;                       /* Position from top */
left: 20px;                      /* Position from left */
width: 100%;                     /* Element width */
height: 50vh;                    /* Element height (viewport) */
max-width: 1200px;               /* Maximum width */
min-height: 300px;               /* Minimum height */
```

Spacing and Borders

```
margin: 10px 20px;               /* Vertical | Horizontal */
padding: 10px 15px 20px 25px;    /* Top | Right | Bottom | Left */
border: 2px solid #333;          /* Width | Style | Color */
border-radius: 5px;              /* Rounded corners */
box-shadow: 0 2px 4px rgba(0,0,0,0.1); /* Shadow */
```

8. Data Formats and Communication

HTTP Protocol

HTTP Methods and Their Usage

Method	Purpose	Idempotent	Safe	Example Usage
GET	Retrieve data	yes	yes	GET /api/users/123
POST	Create resource	no	no	POST /api/users (create user)
PUT	Update/replace resource	yes	no	PUT /api/users/123 (update user)
PATCH	Partial update	no	no	PATCH /api/users/123 (update email)
DELETE	Remove resource	yes	no	DELETE /api/users/123
HEAD	Get headers only	yes	yes	HEAD /api/users/123 (check existence)
OPTIONS	Get allowed methods	yes	yes	OPTIONS /api/users (CORS preflight)

HTTP Status Codes

Range	Category	Common Codes	Meaning
1xx	Informational	100 Continue	Request received, continue
2xx	Success	200 OK, 201 Created, 204 No Content	Request successful
3xx	Redirection	301 Moved, 304 Not Modified	Further action needed
4xx	Client Error	400 Bad Request, 401 Unauthorized, 404 Not Found	Client error
5xx	Server Error	500 Internal Error, 502 Bad Gateway, 503 Unavailable	Server error

HTTP Headers Examples

Request Headers:

```
GET /api/products HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
Accept: application/json
Content-Type: application/json
User-Agent: Mozilla/5.0...
```

Response Headers:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1234
Cache-Control: max-age=3600
Access-Control-Allow-Origin: *
```

Data Exchange Formats

XML (eXtensible Markup Language)

```
<?xml version="1.0" encoding="UTF-8"?>
<customer>
  <id>12345</id>
  <name>Jane Doe</name>
  <email>jane@example.com</email>
  <address>
    <street>123 Main St</street>
    <city>Colombo</city>
    <country>Sri Lanka</country>
  </address>
  <orders>
    <order id="001" date="2024-01-15">
      <total>150.00</total>
    </order>
  </orders>
</customer>
```

Characteristics:

- **Self-documenting** with descriptive tags
- **Schema validation** ensures data integrity
- **Namespace support** prevents naming conflicts
- **Rich metadata** with attributes
- **Verbose syntax** increases payload size ← **Disadvantage**
- **Complex parsing** compared to JSON ← **Disadvantages**

Use Cases: SOAP services, configuration files, document storage, enterprise integration

JSON (JavaScript Object Notation)

```
{
  "customer": {
    "id": 12345,
    "name": "Jane Doe",
    "email": "jane@example.com",
    "address": {
      "street": "123 Main St",
      "city": "Colombo",
      "country": "Sri Lanka"
    },
    "orders": [
      {
        "id": "001",
        "date": "2024-01-15",
        "total": 150.00
      }
    ],
    "active": true,
    "preferences": null
  }
}
```

Characteristics:

- **Lightweight** and compact syntax
- **Human-readable** and easy to understand
- **Native JavaScript support** for web applications
- **Fast parsing** and generation
- **Wide language support** across platforms
- **Limited data types** (string, number, boolean, null, object, array) ← **Disadvantage**
- **No comments** or schema validation built-in ← **Disadvantage**

Use Cases: REST APIs, web applications, NoSQL databases, configuration files

JSON vs XML Comparison

Aspect	JSON	XML
Syntax	Lightweight, minimal	Verbose with tags
Parsing Speed	Fast	Slower
File Size	Smaller	Larger
Data Types	Limited built-in types	Flexible with schemas
Schema Validation	External tools needed	Built-in DTD/XSD support
Comments	Not supported	Supported
Human Readability	Very readable	Readable but verbose
Web Browser Support	Native	Requires parsing

API Communication Patterns

Request-Response Pattern

```
// Synchronous communication
const response = await fetch('/api/users/123');
const user = await response.json();
console.log(user.name);
```

Event-Driven Pattern

javascript

```
// Asynchronous communication
websocket.on('user-updated', (userData) => {
  updateUserInterface(userData);
});

// Publish-Subscribe pattern
eventBus.publish('order-created', orderData);
eventBus.subscribe('order-created', sendConfirmationEmail);
```

Batch Processing Pattern

javascript

```
// Process multiple requests together
const batchRequest = {
  requests: [
    { method: 'GET', url: '/api/users/1' },
    { method: 'GET', url: '/api/users/2' },
    { method: 'POST', url: '/api/orders', data: orderData }
  ]
};
const batchResponse = await fetch('/api/batch', {
  method: 'POST',
  body: JSON.stringify(batchRequest)
});
```

9. SOA Tools & Standards

Core SOA Technologies

SOAP (Simple Object Access Protocol)

- **Purpose**: Protocol for exchanging structured information in web services
- XML-based messaging protocol
- Built-in error handling and security
- Transport protocol independent (HTTP, SMTP, TCP)
- Supports complex data types and transactions

Example SOAP Message:

Xml :

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
    <authentication>
      <username>user123</username>
      <password>pass456</password>
    </authentication>
  </soap:Header>
  <soap:Body>
    <getCustomer xmlns="http://example.com/customer">
      <customerId>12345</customerId>
    </getCustomer>
  </soap:Body>
</soap:Envelope>
```

Benefits	Drawbacks
Standardized protocol with strict specifications	Complex setup and configuration
Built-in security with WS-Security	Verbose XML messages increase bandwidth
Transaction support with WS-Transaction	Performance overhead compared to REST
Enterprise-grade reliability and error handling	-

REST (Representational State Transfer)

REST Principles:

1. **Stateless**: Each request contains all necessary information
2. **Client-Server**: Clear separation of concerns
3. **Cacheable**: Responses can be cached for performance
4. **Uniform Interface**: Consistent API design patterns
5. **Layered System**: Intermediary layers for scalability
6. **Code on Demand** (optional): Server can send executable code

Example REST API:

http

```
GET /api/customers/12345 HTTP/1.1
Host: api.example.com
Accept: application/json
Authorization: Bearer token123

HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 12345,
  "name": "Jane Doe",
  "email": "jane@example.com",
  "created": "2024-01-15T10:30:00Z"
}
```

Benefits	Limitations
Simple and lightweight	Limited security compared to SOAP
HTTP-native leveraging existing infrastructure	No built-in reliability mechanisms
High performance with minimal overhead	Stateless constraint can be limiting
Wide adoption and tool support	

WSDL (Web Services Description Language)

- **Purpose**: XML document describing web service interfaces
- **Types**: Data type definitions
- **Messages**: Input/output message formats
- **Port Types**: Available operations
- **Bindings**: Protocol and data format specifications
- **Services**: Service endpoints and locations

Example WSDL Structure:

xml

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <!-- Data type definitions -->
  </types>
  <message name="GetCustomerRequest">
    <part name="customerId" type="xsd:int"/>
  </message>
  <message name="GetCustomerResponse">
    <part name="customer" type="tns:Customer"/>
  </message>
  <portType name="CustomerService">
    <operation name="GetCustomer">
      <input message="tns:GetCustomerRequest"/>
      <output message="tns:GetCustomerResponse"/>
    </operation>
  </portType>
</definitions>
```

UDDI (Universal Description, Discovery, and Integration)

- **Purpose**: Registry for discovering and publishing web services
- **Business Registry**: Company information and services
- **Service Registry**: Technical service descriptions
- **Binding Registry**: Service access points and protocols

UDDI Data Structure:

xml

```
<businessEntity businessKey="uuid:12345">
  <name>Example Corporation</name>
  <description>Leading provider of web services</description>
  <businessServices>
    <businessService serviceKey="uuid:67890">
```

```
      <name>Customer Management Service</name>
      <bindingTemplates>
        <bindingTemplate bindingKey="uuid:abcde">
          <accessPoint>http://api.example.com/customer</accessPoint>
        </bindingTemplate>
      </bindingTemplates>
    </businessService>
  </businessServices>
</businessEntity>
```

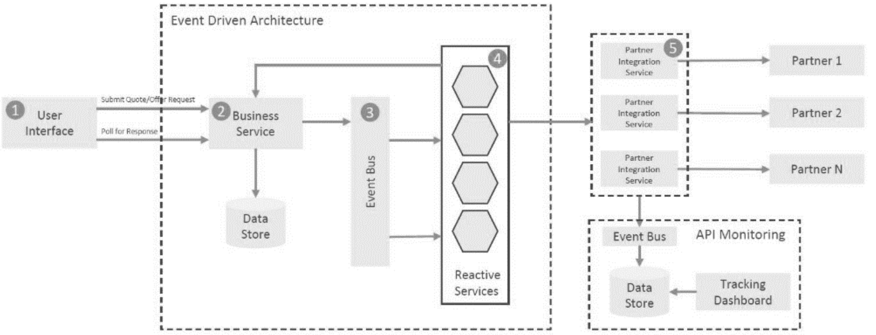
SOA Infrastructure Components

01.Enterprise Service Bus (ESB) : Middleware platform for service integration and communication

- **Message routing** between services
- **Protocol transformation** (SOAP ↔ REST)
- **Data format conversion** (XML ↔ JSON)
- **Service orchestration** and workflow management
- **Security** and authentication
- **Monitoring** and logging

ESB Architecture:

```
[Service A] ←→ [ESB: Router + Transformer + Monitor] ←→ [Service B]
                        ↕
                  [Service Registry]
```

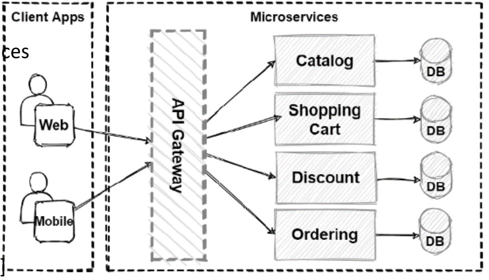


Benefits	Challenges
Centralized service management	Single point of failure risk
Protocol agnostic communication	Performance bottleneck potential
Loose coupling between services	Complex configuration and management
Built-in monitoring and governance	-

02. API Gateway : Single entry point for client-service communication

Responsibilities:

- **Request routing** to appropriate services
- **Authentication** and authorization
- **Rate limiting** and throttling
- **Request/response transformation**
- **Caching** for performance
- **Analytics** and monitoring



API Gateway Pattern:

```
[Mobile App] → [API Gateway] → [User Service]
[Web App]   →          ↓         → [Order Service]
[Partner API] →     [Security]   → [Payment Service]
                    [Caching]
                    [Analytics]
```

API Documentation Standards

OpenAPI/Swagger Specification : Standard for describing REST APIs

Example OpenAPI Document:

yaml

```yaml
openapi: 3.0.0
info:
  title: Customer API
  version: 1.0.0
  description: API for managing customer data
servers:
  - url: https://api.example.com/v1
paths:
  /customers/{id}:
    get:
      summary: Get customer by ID
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: Customer found
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Customer'
        '404':
          description: Customer not found
components:
  schemas:
    Customer:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
        email:
          type: string
          format: email
```

Benefits:

- **Interactive documentation** with Swagger UI
- **Code generation** for clients and servers
- **Validation** of requests and responses
- **Testing** capabilities built-in

SOA Security Standards

WS-Security : Security framework for SOAP web services

- **Message-level security** (not just transport)
- **Digital signatures** for message integrity
- **Encryption** for confidentiality
- **Authentication tokens** (username, X.509, SAML)

OAuth 2.0 : Authorization framework for API access

Flow Example:

1. Client requests authorization from user
2. User grants authorization
3. Client receives authorization code
4. Client exchanges code for access token
5. Client uses token to access protected resources

Token Types:

- **Access Token**: Short-lived, grants access to resources
- **Refresh Token**: Long-lived, used to obtain new access tokens
- **ID Token**: Contains user identity information (OpenID Connect)

10. Industry Case Studies

Netflix: Microservices at Scale

Architecture Evolution

Before (2008):

- Monolithic architecture hosted on physical servers
- Single point of failure affecting entire service
- Difficult to scale individual components

After (2012+):

- 1000+ microservices running on AWS cloud
- Each service owns specific functionality (user profiles, recommendations, video encoding)
- Independent scaling and deployment

Key Innovations

Chaos Engineering:

- **Chaos Monkey**: Randomly terminates services to test resilience
- **Chaos Gorilla**: Simulates entire data center failures
- **Chaos Kong**: Tests multi-region disaster scenarios

Service Reliability:

- **Circuit Breaker Pattern**: Prevents cascade failures
- **Bulkhead Pattern**: Isolates critical resources
- **Retry and Timeout**: Handles transient failures gracefully

Results

- **99.99% uptime** despite individual service failures
- **Global expansion** to 190+ countries
- **Rapid feature deployment** with multiple releases per day
- **Independent team scaling** with 1000+ engineers

Amazon: SOA Transformation

The Mandate (Early 2000s)

Jeff Bezos issued a company-wide directive:

1. All teams must expose functionality through service interfaces
2. Teams must communicate only through these interfaces
3. No direct linking, shared memory, or backdoors allowed
4. Technology choice is irrelevant (HTTP, CORBA, etc.)
5. All service interfaces must be designed to be externally accessible

Implementation Strategy

Service Decomposition:

- **Product Catalog Service**: Product information and search
- **Inventory Service**: Stock levels and availability
- **Pricing Service**: Dynamic pricing algorithms
- **Recommendation Service**: Personalized product suggestions
- **Order Service**: Order processing and fulfillment
- **Payment Service**: Transaction processing and billing

Benefits Realized:

- **Team autonomy**: Each service owned by dedicated team
- **Technology diversity**: Teams choose best tools for their domain
- **Faster innovation**: Independent development and deployment
- **Better fault isolation**: Service failures don't cascade

AWS Birth

Internal Platform → External Service:

- Amazon's internal SOA infrastructure became AWS services
- EC2, S3, SQS, and other services originated from internal needs
- External customers could use the same reliable infrastructure

Key Success Factors

- **Leadership commitment**: Top-down mandate for SOA adoption
- **Gradual migration**: Incremental transformation over years
- **Investment in infrastructure**: Significant tooling and platform development
- **Team restructuring**: Conway's Law - teams structured around services

PickMe: Sri Lankan Success Story

Business Context

- Leading ride-hailing service in Sri Lanka
- Competing with global players like Uber
- Needed to scale rapidly while maintaining service quality
- Required integration with local payment systems and regulations

SOA Implementation

Core Services Architecture:

```
[Mobile Apps] → [API Gateway] → [Authentication Service]
                    ↓              [Booking Service]
               [Load Balancer] → [Driver Service]
                    ↓              [Fare Calculation Service]
               [Service Mesh] → [Route Optimization Service]
                    ↓              [Payment Service]
               [Monitoring] → [Notification Service]
```

Service Breakdown:

1. **User Management Service**
   - Registration, profiles, preferences
   - Authentication and authorization
   - Rating and feedback management
2. **Booking Service**
   - Ride request processing
   - Driver matching algorithms
   - Trip state management
3. **Geolocation Service**
   - Real-time GPS tracking
   - Route calculation and optimization
   - Traffic data integration
4. **Fare Calculation Service**
   - Dynamic pricing algorithms
   - Surge pricing during peak hours
   - Discount and promotion handling
5. **Payment Service**
   - Multiple payment method support (cards, mobile money, cash)
   - Integration with local banks
   - Transaction processing and reconciliation
6. **Notification Service**
   - Push notifications to mobile apps
   - SMS alerts for booking confirmations
   - Email receipts and trip summaries

Technical Achievements

Scalability:

- Handles **100,000+ daily rides** across multiple cities
- **Sub-second response times** for booking requests
- **99.9% uptime** with redundant service deployment

Performance Optimizations:

- **Caching layer**: Redis for frequently accessed data
- **Database sharding**: Horizontal scaling of user and trip data
- **CDN integration**: Fast delivery of mobile app assets
- **Async processing**: Queue-based handling of non-critical operations

Local Adaptations:

- **Sinhala/Tamil language support**: Localized user interfaces
- **Cash payment handling**: Integration with driver cash collection
- **Local bank integration**: Support for Lankan payment methods
- **Regulatory compliance**: Adherence to transportation regulations

Business Impact

- **Market leadership**: Largest ride-hailing service in Sri Lanka
- **Rapid expansion**: Coverage across major cities and towns
- **Employment creation**: Thousands of driver-partners onboarded
- **Technology export**: Platform components reused in other markets

Traditional Bank SOA Migration Case Study

Initial State: Legacy Monolith

- **Core Banking System**: 30-year-old COBOL mainframe
- **Branch Applications**: Desktop applications with direct database access
- **Online Banking**: Separate system with batch synchronization
- **Mobile App**: Basic functionality with limited integration

Problems Identified

- **Slow time-to-market**: New features took 6-12 months
- **Maintenance nightmare**: Changes required extensive regression testing
- **Limited scalability**: Monolith couldn't handle peak loads
- **Technology debt**: Difficulty finding COBOL developers
- **Customer experience**: Inconsistent data across channels

SOA Migration Strategy

Phase 1: Service Extraction

- Extract core services from monolith without changing interfaces
- Create facade services to maintain backward compatibility
- Implement API gateway for unified access

Phase 2: Channel Integration

- Develop RESTful APIs for mobile and web channels
- Implement real-time data synchronization
- Create consistent customer experience across touchpoints

Phase 3: Modernization

- Replace legacy components with cloud-native services
- Implement event-driven architecture for real-time processing
- Add analytics and AI services for personalization

Service Architecture

Account Services:

- Account management and balance inquiries
- Transaction history and statements
- Account opening and closure workflows

Transaction Services:

- Payment processing and transfers
- Bill payment and standing orders
- Currency exchange and foreign transactions

Customer Services:

- Customer onboarding and KYC
- Profile management and preferences
- Customer support and communication

Analytics Services:

- Fraud detection and prevention
- Risk assessment and credit scoring
- Customer behavior analysis and recommendations

Results After Migration

Business Metrics:

- **Time-to-market**: New features deployed in 2-4 weeks
- **Customer satisfaction**: 40% improvement in app ratings
- **Operational efficiency**: 60% reduction in manual processes
- **Revenue growth**: 25% increase in digital banking adoption

Technical Metrics:

- **System availability**: 99.95% uptime (from 98.5%)
- **Response times**: Sub-second API responses
- **Scalability**: Automatic scaling during peak periods
- **Development velocity**: 3x faster feature delivery

Critical Success Factors

- **Executive sponsorship**: Strong leadership commitment to transformation
- **Phased approach**: Gradual migration minimizing business disruption
- **Team training**: Extensive upskilling of development teams
- **Regulatory compliance**: Ensuring all changes meet banking regulations
- **Customer communication**: Transparent communication about changes

s

11. SOA Design Best Practices

Service Design Principles

1. Single Responsibility Principle

- Each service should have one clear business purpose
- Avoid creating services that handle multiple unrelated functions
- Example: Separate OrderService and InventoryService rather than OrderInventoryService

2. Business Capability Alignment

- Design services around business capabilities, not technical functions
- Services should reflect how the business operates
- Example: CustomerService handles all customer-related operations

3. Data Ownership

- Each service should own its data and database
- Avoid shared databases between services
- Use APIs for data access between services

4. Autonomous Teams

- Organize teams around service boundaries
- Teams should be able to develop, deploy, and maintain their services independently
- Follow Conway's Law: System design reflects organizational structure

API Design Guidelines

RESTful API Best Practices:

http

```
# Good: Resource-based URLs
GET /api/customers/123
POST /api/customers
PUT /api/customers/123
DELETE /api/customers/123
```

```
# Bad: Action-based URLs
GET /api/getCustomer?id=123
POST /api/createCustomer
POST /api/updateCustomer
POST /api/deleteCustomer
```

HTTP Status Code Usage:

- **200 OK**: Successful GET, PUT, PATCH
- **201 Created**: Successful POST with resource creation
- **202 Accepted**: Async processing initiated
- **204 No Content**: Successful DELETE or PUT without response body
- **400 Bad Request**: Client error in request format
- **401 Unauthorized**: Authentication required
- **403 Forbidden**: Access denied
- **404 Not Found**: Resource doesn't exist
- **409 Conflict**: Resource conflict (duplicate creation)
- **500 Internal Server Error**: Server-side error

Error Response Format:

json

```json
{
  "error": {
    "code": "INVALID_EMAIL",
    "message": "The provided email address is not valid",
    "details": {
      "field": "email",
      "value": "invalid-email",
      "constraint": "Must be valid email format"
    },
    "timestamp": "2024-01-15T10:30:00Z",
    "requestId": "req-12345"
  }
}
```

Service Documentation

OpenAPI Specification Best Practices:

yaml

```yaml
openapi: 3.0.0
info:
  title: Customer Management API
  version: 2.1.0
  description: |
    Comprehensive API for managing customer data and operations.

    ## Authentication
    All endpoints require Bearer token authentication.

    ## Rate Limiting
    Requests are limited to 1000 per hour per API key.

  contact:
    name: API Support
    email: api-support@company.com
    url: https://docs.company.com/support
  license:
    name: MIT
    url: https://opensource.org/licenses/MIT
servers:
  - url: https://api.company.com/v2
    description: Production server
  - url: https://staging-api.company.com/v2
    description: Staging server
```

Documentation Requirements:

- **Clear descriptions** for all endpoints and parameters
- **Example requests and responses** for complex operations
- **Error scenarios** with appropriate status codes
- **Authentication and authorization** requirements
- **Rate limiting** and usage guidelines
- **Changelog** for version updates

Security Best Practices

Authentication and Authorization

OAuth 2.0 Implementation:

javascript

```javascript
// Client credentials flow for service-to-service
const tokenResponse = await fetch('/oauth/token', {
  method: 'POST',
  headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
  body: 'grant_type=client_credentials&client_id=service1&client_secret=secret123'
});

const { access_token } = await tokenResponse.json();

// Use token for API calls
const apiResponse = await fetch('/api/customers', {
  headers: { 'Authorization': `Bearer ${access_token}` }
});
```

JWT Token Best Practices:

- **Short expiration times**: 15-60 minutes for access tokens
- **Secure storage**: Use httpOnly cookies or secure token storage
- **Token rotation**: Implement refresh token mechanism
- **Payload minimization**: Include only necessary claims
- **Signature verification**: Always validate JWT signatures

API Security

Input Validation:

javascript

```javascript
// Example validation middleware
function validateCustomer(req, res, next) {
  const schema = {
    name: { type: 'string', minLength: 2, maxLength: 100 },
    email: { type: 'string', format: 'email' },
    age: { type: 'integer', minimum: 18, maximum: 120 }
  };

  const { error } = validate(req.body, schema);
  if (error) {
    return res.status(400).json({ error: error.details });
  }
  next();
}
```

Rate Limiting:

javascript

```javascript
// Express rate limiting middleware
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 from this IP',
  standardHeaders: true,
  legacyHeaders: false
});

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

HTTPS Enforcement:

javascript

```javascript
// Redirect HTTP to HTTPS
app.use((req, res, next) => {
  if (req.header('x-forwarded-proto') !== 'https') {
    res.redirect(`https://${req.header('host')}${req.url}`);
  } else {
    next();
  }
});
```

Monitoring and Observability

Logging Best Practices

Structured Logging:

javascript

```javascript
// Good: Structured logging with context
logger.info('Customer created', {
  customerId: customer.id,
  email: customer.email,
  requestId: req.id,
  timestamp: new Date().toISOString(),
  source: 'customer-service'
});

// Bad: Unstructured logging
console.log('Customer created: ' + customer.id);
```

Log Levels Usage:

- **ERROR**: System errors requiring immediate attention
- **WARN**: Potentially harmful situations
- **INFO**: General information about application flow
- **DEBUG**: Detailed information for debugging

Health Checks

Service Health Endpoint:

javascript

```javascript
app.get('/health', async (req, res) => {
  const health = {
    status: 'healthy',
    timestamp: new Date().toISOString(),
    version: process.env.APP_VERSION,
    checks: {
      database: await checkDatabase(),
      redis: await checkRedis(),
      externalApi: await checkExternalService()
    }
  };

  const isHealthy = Object.values(health.checks).every(check => check.status === 'healthy');
  res.status(isHealthy ? 200 : 503).json(health);
});
```

Metrics and Monitoring

Key Performance Indicators (KPIs):

- **Response Time**: Average, 95th, 99th percentile
- **Throughput**: Requests per second
- **Error Rate**: Percentage of failed requests
- **Availability**: Uptime percentage
- **Resource Utilization**: CPU, memory, disk usage

Alerting Strategy:

- **Error Rate**: Alert when > 5% for 5 minutes
- **Response Time**: Alert when 95th percentile > 2 seconds
- **Availability**: Alert when < 99.9% uptime
- **Resource Usage**: Alert when CPU > 80% for 10 minutes

Deployment and DevOps Best Practices

Containerization with Docker

Dockerfile Best Practices:

dockerfile

```
# Use specific version tags, not 'latest'
FROM node:18.17.0-alpine

# Create non-root user for security
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nodejs -u 1001

# Set working directory
WORKDIR /app

# Copy package files first for better caching
COPY package*.json ./
RUN npm ci --only=production && npm cache clean --force

# Copy application files
COPY --chown=nodejs:nodejs . .

# Switch to non-root user
USER nodejs

# Expose port
EXPOSE 3000

# Health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:3000/health || exit 1

# Start application
CMD ["npm", "start"]
```

CI/CD Pipeline

Pipeline Stages:

1. **Source**: Code commit triggers pipeline
2. **Build**: Compile code and run unit tests
3. **Test**: Integration and contract testing
4. **Security**: Vulnerability scanning and code analysis
5. **Package**: Create container images
6. **Deploy**: Progressive deployment to environments
7. **Monitor**: Post-deployment verification

Example GitHub Actions Pipeline:

yaml

```
name: CI/CD Pipeline

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '18'
      - run: npm ci
      - run: npm test
      - run: npm run lint

  security:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run security audit
        run: npm audit --audit-level high

  build:
    needs: [test, security]
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Build Docker image
        run: docker build -t app:${{ github.sha }} .
      - name: Push to registry
        run: docker push app:${{ github.sha }}

  deploy:
    needs: build
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'
    steps:
      - name: Deploy to staging
        run: kubectl set image deployment/app app=app:${{ github.sha }}
```

Performance Optimization

Caching Strategies

Cache Patterns:

javascript

```
// Cache-aside pattern
async function getCustomer(id) {
  // Try cache first
  let customer = await cache.get(`customer:${id}`);

  if (!customer) {
    // Fetch from database
    customer = await database.customers.findById(id);

    // Store in cache with TTL
    await cache.set(`customer:${id}`, customer, { ttl: 300 });
  }

  return customer;
}

// Write-through pattern
async function updateCustomer(id, data) {
  // Update database
  const customer = await database.customers.update(id, data);

  // Update cache
  await cache.set(`customer:${id}`, customer, { ttl: 300 });

  return customer;
}
```

Cache Invalidation:

javascript

```
// Event-driven cache invalidation
eventBus.on('customer-updated', async (customerId) => {
  await cache.delete(`customer:${customerId}`);
  await cache.delete(`customer-orders:${customerId}`);
});
```

Database Optimization

Connection Pooling:

javascript

```
// PostgreSQL connection pool
const pool = new Pool({
  host: 'localhost',
  port: 5432,
  database: 'myapp',
  user: 'dbuser',
  password: 'dbpass',
  max: 20, // Maximum number of connections
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 2000
});
```

Query Optimization:

sql

```
-- Good: Use indexes and specific columns
SELECT id, name, email
FROM customers
WHERE email = $1 AND status = 'active'
LIMIT 10;

-- Bad: Select all columns without proper indexing
SELECT * FROM customers WHERE email LIKE '%@gmail.com%';
```

Load Balancing

Nginx Configuration:

nginx

```
upstream app_servers {
    least_conn;
    server app1:3000 weight=3;
    server app2:3000 weight=2;
    server app3:3000 weight=1;
    keepalive 32;
}

server {
    listen 80;
    server_name api.example.com;

    location / {
        proxy_pass http://app_servers;
        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_connect_timeout 30s;
        proxy_send_timeout 30s;
        proxy_read_timeout 30s;
    }
}
```