Lecture 8: System Design and Architecture

Building the Blueprint for Your Software

State University of Zanzibar (SUZA)
BSc Computer Science

Contents

1 Introduction to System Design

1.1 What is System Design?

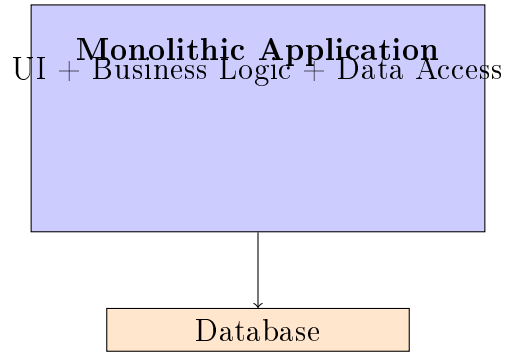
System design is the process of defining the architecture, components, modules, interfaces, and data flow of a system to satisfy specified requirements.

1.2 Why Design Matters

- Provides a roadmap for development
- Identifies potential problems early
- Enables parallel development by teams
- Facilitates maintenance and scalability
- Ensures all team members have shared understanding

2 Architecture Patterns

2.1 Monolithic Architecture

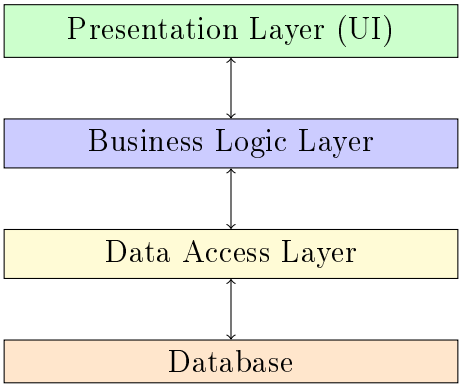


Characteristics:

- Single deployable unit
- All components tightly coupled
- Simple to develop and deploy initially

Best For: Small projects, student projects, MVPs
Drawbacks: Hard to scale, difficult to maintain as it grows

2.2 Layered (N-Tier) Architecture



Layers:

1. **Presentation Layer:** User interface (HTML, React, etc.)
2. **Business Logic Layer:** Application rules and processing
3. **Data Access Layer:** Database operations
4. **Database Layer:** Data storage

Best For: Most web applications, enterprise software

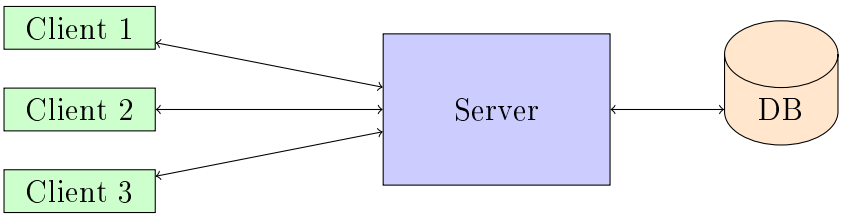
2.3 MVC (Model-View-Controller)

Components:

- **Model:** Data and business logic
- **View:** User interface display
- **Controller:** Handles user input, updates model

Popular Frameworks: Django (Python), Laravel (PHP), Ruby on Rails

2.4 Client-Server Architecture



Best For: Web applications, mobile apps with backend

3 Technology Stack Selection

3.1 Frontend Technologies

Technology	Type	Best For
HTML/CSS/JS	Basics	Simple websites
React	Library	Dynamic SPAs
Vue.js	Framework	Progressive apps
Angular	Framework	Enterprise apps
Bootstrap	CSS Framework	Responsive design
Tailwind CSS	CSS Framework	Custom designs

3.2 Backend Technologies

Technology	Language	Best For
Node.js/Express	JavaScript	APIs, real-time apps
Django	Python	Rapid development
Flask	Python	Lightweight APIs
Spring Boot	Java	Enterprise apps
Laravel	PHP	Web applications
Ruby on Rails	Ruby	Startups, MVPs

3.3 Database Selection

Relational Databases (SQL):

- PostgreSQL - Feature-rich, open source
- MySQL - Popular, easy to use
- SQLite - Lightweight, file-based

NoSQL Databases:

- MongoDB - Document-based, flexible schema
- Redis - In-memory, caching
- Firebase - Real-time, mobile apps

When to Use What:

- **SQL:** Structured data, complex queries, transactions
- **NoSQL:** Flexible schema, scalability, unstructured data

4 Database Design

4.1 Entity-Relationship Diagram (ERD)

An ERD shows entities (tables) and their relationships.

Components:

- **Entity:** A table (e.g., User, Product, Order)
- **Attribute:** Column in a table (e.g., name, email)
- **Relationship:** Connection between tables

4.2 Relationship Types

- **One-to-One (1:1):** User has one Profile
- **One-to-Many (1:N):** User has many Orders
- **Many-to-Many (M:N):** Students enroll in Courses

4.3 Example: E-commerce Database

```
Table: users
- id (PK)
- username
- email
- password_hash
- created_at

Table: products
- id (PK)
- name
- description
- price
- stock_quantity

Table: orders
- id (PK)
- user_id (FK -> users)
- total_amount
- status
- created_at

Table: order_items
- id (PK)
- order_id (FK -> orders)
- product_id (FK -> products)
- quantity
- price
```

4.4 Normalization

Purpose: Reduce data redundancy and improve data integrity.

Normal Forms:

1. **1NF:** No repeating groups, atomic values
2. **2NF:** 1NF + no partial dependencies
3. **3NF:** 2NF + no transitive dependencies

5 API Design

5.1 What is an API?

Application Programming Interface - a way for software components to communicate.

5.2 REST API Principles

- **Stateless:** Each request contains all needed information
- **Resource-based:** URLs represent resources
- **HTTP Methods:** Use appropriate methods for actions
- **JSON:** Standard data format

5.3 HTTP Methods

Method	Action	Example
GET	Read/Retrieve	GET /users
POST	Create	POST /users
PUT	Update (full)	PUT /users/1
PATCH	Update (partial)	PATCH /users/1
DELETE	Delete	DELETE /users/1

5.4 RESTful URL Design

```
Good URLs:
GET    /users            - List all users
GET    /users/1          - Get user with id 1
POST   /users            - Create new user
PUT    /users/1          - Update user 1
DELETE /users/1          - Delete user 1
GET    /users/1/orders   - Get orders for user 1

Bad URLs:
GET /getUsers
GET /getUserById?id=1
POST /createNewUser
GET /deleteUser/1
```

5.5 API Response Format

```
Success Response:
{
    "success": true,
    "data": {
        "id": 1,
        "name": "John Doe",
        "email": "john@example.com"
    }
}

Error Response:
{
    "success": false,
    "error": {
        "code": 404,
        "message": "User not found"
    }
}
```

5.6 HTTP Status Codes

Code	Meaning
200 OK	Success
201 Created	Resource created
400 Bad Request	Invalid input
401 Unauthorized	Authentication required
403 Forbidden	Not allowed
404 Not Found	Resource doesn't exist
500 Server Error	Internal error

6 User Interface Design

6.1 UI/UX Principles

- **Consistency:** Similar elements behave similarly
- **Feedback:** Show users what's happening
- **Simplicity:** Don't overwhelm users
- **Accessibility:** Design for all users
- **Mobile-first:** Design for small screens first

6.2 Wireframes vs Mockups

- **Wireframe:** Low-fidelity, shows layout and structure

- **Mockup:** High-fidelity, shows visual design
- **Prototype:** Interactive, shows user flow

6.3 Tools for UI Design

- Figma (recommended, free)
- Adobe XD
- Sketch
- Balsamiq (wireframes)
- Paper and pencil!

7 Security Design

7.1 Authentication vs Authorization

- **Authentication:** Who are you? (Login)
- **Authorization:** What can you do? (Permissions)

7.2 Authentication Methods

- **Session-based:** Server stores session data
- **Token-based (JWT):** Client stores token
- **OAuth:** Third-party authentication (Google, GitHub)

7.3 Security Best Practices

1. Hash passwords (use bcrypt)
2. Use HTTPS everywhere
3. Validate all user input
4. Prevent SQL injection (use parameterized queries)
5. Prevent XSS (escape output)
6. Implement rate limiting
7. Keep dependencies updated

8 Design Document Template

Your System Design Document should include:

1. **Introduction:** Purpose and scope
2. **Architecture Overview:** High-level diagram
3. **Technology Stack:** Frontend, backend, database
4. **Database Design:** ERD and schema
5. **API Design:** Endpoints and examples
6. **UI Design:** Wireframes/mockups
7. **Security:** Authentication and authorization
8. **Deployment:** How it will be hosted

9 Summary

- Choose architecture pattern based on project needs
- Select appropriate technology stack
- Design database with proper normalization
- Follow REST principles for APIs
- Consider security from the start
- Document your design decisions