Web Architecture & Design Patterns

Web Technologies — Lecture 2

Masoud Hamad

State University of Zanzibar (SUZA)

Semester II, 2025/2026

Outline

Architecture vs Design vs Code

- **Architecture** — the highest-level decisions: components, boundaries, protocols, data flow. "Hard to change later."
- **Design** — how each component works internally: classes, modules, patterns
- **Code** — the concrete implementation

Why architecture matters

Wrong architecture = the whole house leans. You can paint the walls (refactor code), but you can't easily move the foundation.

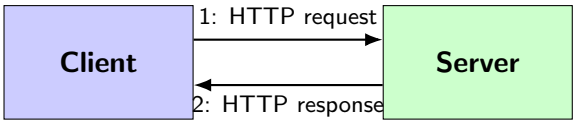
Architectural Quality Attributes

Good architecture optimises for measurable qualities:

- **Performance** — latency, throughput
- **Scalability** — handles 10x users without redesign
- **Availability** — $\sim$99.9% uptime
- **Security** — defends against known attacks
- **Maintainability** — easy to change & test
- **Cost** — runs within budget

Trade-offs are inevitable — e.g., more security $\Rightarrow$ slower performance.

Client–Server Model



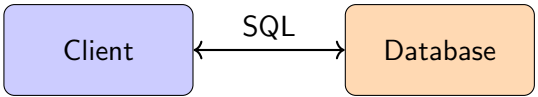
- Foundational pattern of *every* web app
- Server is **passive** — waits for requests
- Client is **active** — initiates communication
- Stateless protocol (HTTP) keeps things simple

Why Tiers?

Break a system into **layers** (logical) deployed as **tiers** (physical), each with one responsibility.

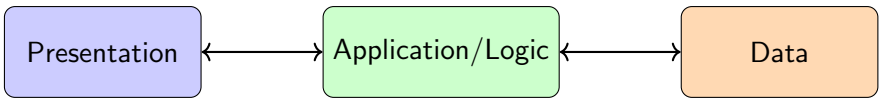
- Manage complexity by abstraction
- Replace one tier without touching others
- Different teams own different tiers
- Easier to scale each tier independently

2-Tier — Classic Client–Server



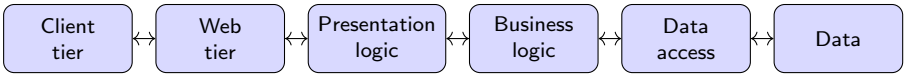
- Fat client connects directly to database
- Examples: old desktop apps, simple internal tools
- **Problems:** all logic in client; can't scale; deployment is painful

3-Tier — The Modern Default



- **Presentation:** browser, mobile app, CLI
- **Application:** business rules, request handling, validation
- **Data:** database, file storage, external APIs

6-Tier — Real-World Web App



- **Client** — browser UI components
- **Web** — HTTP server (nginx, Apache)
- **Presentation logic** — server-side templates / SSR
- **Business logic** — domain rules, workflows
- **Data access** — ORM, query layer
- **Data** — database, cache, search index

Monolith vs Microservices

	Monolith	**Microservices**
Deployment unit	one app	many small services
Tech stack	one language/framework	polyglot allowed
Communication	in-process function calls	HTTP/gRPC/messages
Database	usually one shared DB	one DB per service
Best for	small/medium teams	large orgs at scale
Complexity	low at start, grows	high from day one

Default for new

projects: start with a well-structured monolith. Move to microservices *only* when team size or scale forces it.

Serverless / Functions-as-a-Service

- Write small functions; cloud runs them on demand
- Pay per invocation (millisecond billing)
- No server to manage
- Examples: AWS Lambda, Cloudflare Workers, Vercel Functions

Pros: cheap at low scale, auto-scales, no ops.
Cons: cold starts, vendor lock-in, harder to test locally.

SPA vs SSR vs MPA

- **MPA** (Multi-Page App) — server renders HTML, browser navigates pages. Classic. Good for SEO.
- **SPA** (Single-Page App) — one HTML page; JS rewrites the DOM. Smooth UX, harder SEO. (React, Vue)
- **SSR** (Server-Side Rendering) — best of both. Server sends fully-rendered HTML, then JS "hydrates" for interactivity. (Next.js, Nuxt, Remix)

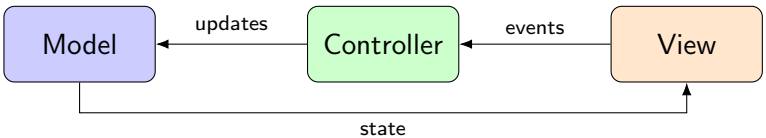
What is a Design Pattern?

Definition

A **design pattern** describes interacting classes/objects that solve a general design problem within a particular context.

- Patterns $\neq$ algorithms or libraries — they are *templates*
- Give engineers a **shared vocabulary**
- Captured in the famous "Gang of Four" (GoF) book (1994)
- Web frameworks bake in many patterns by default

MVC — Model–View–Controller



- **Model** — data + business rules
- **View** — presentation (HTML/CSS)
- **Controller** — handles input, mediates between model & view
- Used by: Rails, Django, Spring MVC, Laravel, ASP.NET MVC

REST — Architectural Style for APIs

REST (Representational State Transfer) is the dominant style for web APIs.

- Resources identified by **URLs**: `/api/students/42`
- Actions via **HTTP verbs**: `GET`, `POST`, `PUT`, `DELETE`
- **Stateless** — each request carries all context
- Representations: JSON, XML, HTML

```
GET    /api/students          -> list all students
POST   /api/students          -> create new student
GET    /api/students/42       -> read student 42
PUT    /api/students/42       -> update student 42
DELETE /api/students/42       -> delete student 42
```

Other Web-Relevant Patterns

Pattern	**Where you'll see it**
Front Controller	Single entry point routes all requests (Express app)
Repository	Hides DB queries behind a clean API
DAO	Same idea, OO-flavour (Spring Data)
DTO	Typed payload between layers
Singleton	DB connection pool, app-wide config
Observer / Pub-Sub	Real-time events, websockets, message queues
Strategy	Pluggable algorithms (auth, payment)
Decorator / Middleware	Logging, auth, CORS in HTTP pipeline
Factory	Create objects without specifying exact class
Adapter	Wrap third-party API to fit your interface

The Modern Web Stack (Reference)

CDN (Cloudflare, CloudFront) — static assets, edge caching

Load Balancer (nginx, ALB) — TLS termination, routing

Web/API Servers (Node, Spring, Django) — behind autoscaling

Cache (Redis, Memcached) — hot data, session store

Database (Postgres, MySQL) — primary + read replicas

Queue (RabbitMQ, SQS) — async work

Object Storage (S3, R2) — images, files, backups

Choosing an Architecture — Heuristics

- **Start simple.** Monolith + Postgres handles surprising scale.
- **Optimise for change.** Layers should let you swap one part without rewriting others.
- **Pay for complexity only when needed.** Don't add Kubernetes for 100 users.
- **Make boundaries explicit.** Clear interfaces $\rightarrow$ replaceable parts.
- **Document the "why".** Future-you won't remember the trade-off.

Summary

- Architecture = high-level decisions that are **hard to change**
- Tier the system: presentation $\rightarrow$ logic $\rightarrow$ data
- Patterns (MVC, REST, Repository, Observer...) give shared vocabulary
- Default: well-structured monolith; reach for microservices only at real scale
- SPA vs SSR vs MPA $\rightarrow$ pick per use case (SEO, UX, complexity)

Next: Agile methodology — how the team *works* on this architecture.

Questions?