

Chapter 2

Fundamentals of Backend Architecture

Understanding how backend systems are structured before writing any code.



2.1 Introduction

- Backend = invisible core of applications.
- Performs logic, data handling, and enforces business rules.
- Supports frontend systems that users interact with.

🧩 **Key idea:** Architecture helps developers build scalable, maintainable, and efficient systems.



2.2 The Role of Backend in a System

- A software system consists of two main sides:

| Layer | Description | Example |
|----------|--|------------------------------|
| Frontend | What users see and interact with | HTML, React, Android |
| Backend | Handles logic, data, and communication | Spring Boot, Node.js, Django |

💡 The **frontend** is the face; the **backend** is the brain.



Conceptual View

- **Frontend (Client side):** Handles UI, input, and user interaction.
- **Backend (Server side):** Handles logic, storage, and rules.

 Communication flow:

```
[User Interface]
↓ Request (HTTP)
[Backend Server]
↓ Query / Logic
[Database]
↑ Response (Data)
[Frontend Display]
```



Separation of Concerns

| Layer | Concern | Description |
|----------|--------------|----------------------------------|
| Frontend | Presentation | How info is shown to users |
| Backend | Logic & Data | How info is processed and stored |

- Enables **modularity, security, and scalability**
- Basis for **multi-tier architecture**



Analogy — The Restaurant Model 🍴

| Role | Software Equivalent | Function |
|-----------|---------------------|---------------------------|
| Customer | User / Client | Places an order |
| Waiter | Frontend | Communicates with kitchen |
| Kitchen | Backend | Prepares dishes |
| Inventory | Database | Stores ingredients |



Backend Responsibilities

1. Processing Requests

e.g., authentication, calculations

2. Validating & Transforming Data

Converts input into internal format

3. Interacting with Databases

CRUD operations with abstraction

4. Returning Structured Responses

e.g., JSON or XML to frontend



2.3 Client–Server Model

The foundation of all modern web systems.

- Defines how *clients* communicate with *servers*
- Every web app follows this model

```
[Client] → Request → [Server] → [Database]
← Response ←
```



Key Characteristics

1. **Request–Response Cycle** — Client initiates, server responds
2. **Statelessness (HTTP)** — Each request is independent
3. **Scalability** — Multiple clients can access one or many servers
4. **Loose Coupling** — Each side can evolve independently



Example — Web Browsing

1. Browser sends HTTP GET request
2. Server retrieves HTML and responds
3. Browser renders page

```
GET /index.html  
→ 200 OK (HTML Response)
```

 Happens within milliseconds!



Advantages & Challenges

✓ Advantages

- Centralized control
- Easy maintenance
- Scalable & secure

⚠ Challenges

- Network dependency
- Server overload
- Latency under high load



2.4 REST Architecture

REST (Representational State Transfer) defines how web services communicate.

- Proposed by *Roy Fielding (2000)*
- Core of all modern APIs
- Resource-oriented, stateless, and simple



Principles of REST

1. Client–Server Separation
2. Stateless Communication
3. Resource-Based Design
4. Uniform Interface
5. HTTP Methods



Standard HTTP Methods

| Method | Action | Example |
|--------|-----------------|---------------------------|
| GET | Retrieve data | <code>/api/users</code> |
| POST | Create resource | <code>/api/users</code> |
| PUT | Update resource | <code>/api/users/1</code> |
| DELETE | Delete resource | <code>/api/users/1</code> |

🧠 Each request returns a *status code* like `200 OK` or `404 Not Found`.



Example Interaction

```
Client: GET /api/books/1
Server: 200 OK
{
  "id":1,
  "title":"Clean Code",
  "author":"Robert C. Martin"
}
```

💬 No state is stored — every request is independent.




REST vs SOAP vs RPC

| Aspect | REST | SOAP | RPC |
|-------------|----------|-----------|--------|
| Protocol | HTTP | HTTP/SMTP | Custom |
| Format | JSON/XML | XML | Varies |
| Coupling | Loose | Tight | Tight |
| Scalability | High | Medium | Low |



REST in Practice (Spring Boot)

```
@RestController
@RequestMapping("/api/books")
public class BookController {
    @GetMapping("/{id}")
    public Book getBook(@PathVariable int id) {
        return new Book(id, "Clean Code", "Robert C. Martin");
    }
}
```

 REST + Java + Spring Boot = clean, maintainable backend.



2.5 Layered Architecture

Divide and conquer — structure backend systems into layers.

Typical Layers

| Layer | Responsibility | Example |
|----------------|----------------------|------------|
| Presentation | Handles input/output | Controller |
| Business Logic | Implements rules | Service |
| Data Access | Connects to DB | Repository |
| Database | Stores data | SQL/NoSQL |



Flow of Control

Client → Controller → Service → Repository → Database

↑ Response flows back up the chain.



Benefits

- ✓ Separation of concerns
 - ✓ Flexibility to change UI or DB independently
 - ✓ Easier testing and debugging
 - ✓ Code reuse across platforms
- ⚠ *Be careful of too many layers → complexity overhead.*



Spring Example

```
@RestController
public class UserController {
    @Autowired private UserService service;

    @GetMapping("/users/{id}")
    public User getUser(@PathVariable Long id) {
        return service.getUserById(id);
    }
}
```

🧩 Layer mapping:

Controller → Service → Repository → Database



2.6 MVC Pattern (Model–View–Controller)

Structure presentation logic for clarity and collaboration.

Core Components

| Component | Purpose | Example |
|------------|------------------------|-------------------|
| Model | Data and rules | User, Product |
| View | UI or output | HTML, JSON |
| Controller | Input and coordination | REST API handlers |



Data Flow

User → Controller → Model → View → User

🧠 Keeps logic, data, and presentation cleanly separated.



MVC in Practice (Spring Example)

```
@Controller
public class LoginController {
    @PostMapping("/login")
    public String login(@RequestParam String username,
                       @RequestParam String password, Model model) {
        boolean success = authService.verify(username, password);
        if (success) return "dashboard";
        else return "login";
    }
}
```



Benefits of MVC

- ✓ Clear separation of concerns
- ✓ Easy testing and maintenance
- ✓ Parallel teamwork (UI, backend, DB)
- ✓ Scalable for large projects



2.7 Introduction to Spring Framework

Turning theory into real-world implementation.

What is Spring?

- A **modular Java framework** for backend development.
- Promotes *Inversion of Control*, *Dependency Injection*, and *Layered Design*.
- Foundation for **Spring Boot**, used widely in enterprise systems.



Core Concepts

| Concept | Description |
|-----------------------------------|--|
| IoC (Inversion of Control) | Framework manages object creation. |
| DI (Dependency Injection) | Objects receive dependencies automatically. |
| AOP (Aspect-Oriented Programming) | Handles cross-cutting concerns like logging. |
| Spring Boot | Simplifies setup and deployment. |



Example — Hello REST Controller

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, Backend!";
    }
}
```

🧩 **Layers in action:**

Controller → REST Endpoint → Response to Client



Spring Aligns with Architecture

| Concept | Spring Implementation |
|----------------|---|
| Client–Server | Acts as the server |
| REST | <code>@RestController</code> , <code>@GetMapping</code> |
| Layered Design | Controller, Service, Repository |
| MVC | Built-in support |



Why Learn Spring?

-  Industry standard framework
-  Bridges theory → implementation
-  Reinforces backend architecture concepts
-  Ideal foundation for enterprise or cloud development



Summary

| Concept | Purpose |
|----------------------|--------------------------------|
| Client–Server | Foundation of web systems |
| REST | Standard for APIs |
| Layered Architecture | Organizes backend logic |
| MVC | Structures presentation flow |
| Spring | Implements these ideas in Java |

Backend architecture connects theory with practice — turning design principles into working systems.

