Fundamentals of Web-Based Applications with Spring Boot, REST API, and Microservices

Author: Anand Prakash
Date: 08 June 2025
Email: adprakash2006@gmail.com

June 8, 2025

1 Fundamentals of Web-Based Applications

A **web-based application** is software that runs on a web server and is accessed by users through a web browser (e.g., Chrome, Firefox) over the internet. Unlike traditional desktop applications, web applications do not require installation on the user's device; they rely on a client-server architecture where:

- **Client**: The user's device (browser, mobile app) sends requests and displays responses.
- **Server**: The web server processes requests, performs computations, and interacts with databases or other services.
- **Database**: Stores and manages the application's data.

Web applications typically follow a **request-response model** using the HTTP protocol. The client sends an HTTP request (e.g., GET, POST), the server processes it, and returns an HTTP response (e.g., HTML, JSON).

2 Key Concepts in the Context of Java

2.1 Spring Boot

Spring Boot is an open-source Java framework that simplifies the development of web applications and microservices. It builds on the **Spring Framework**, which provides dependency injection, MVC architecture, and other tools for enterprise applications. Spring Boot's key features include:

- **Auto-Configuration**: Automatically configures components (e.g., database connections, web servers) based on dependencies.
- **Embedded Server**: Includes built-in servers like Tomcat or Jetty, so you don't need a separate server setup.
- **Simplified Dependency Management**: Uses Maven or Gradle to manage libraries (e.g., for REST APIs, databases).

Author: Anand Prakash
Date: 08 June 2025
Email: adprakash2006@gmail.com

- **Production-Ready Features**: Offers monitoring, metrics, and health checks out of the box.

Why Use Spring Boot? It reduces boilerplate code, speeds up development, and makes it easy to create standalone, scalable applications, especially RESTful microservices.

2.2 REST API

A **REST API** (Representational State Transfer Application Programming Interface) is a set of rules for building web services that allow systems to communicate over HTTP. REST APIs are stateless, meaning each request contains all the information needed to process it. Key characteristics:

- **Resources**: Data entities (e.g., users, orders) are represented as resources, identified by URLs (e.g., `/api/users`).
- **HTTP Methods**: Standard methods like GET (retrieve), POST (create), PUT (update), and DELETE (remove) manipulate resources.
- **JSON/XML**: Data is typically exchanged in JSON or XML format.
- **Statelessness**: No client state is stored on the server between requests.

Example: A GET request to `/api/users/123` might return a JSON object like:

```
{
  "id": 123,
  "name": "Alice",
  "email": "alice@example.com"
}
```

In Java, Spring Boot makes it easy to create REST APIs using annotations like `@RestController` and `@GetMapping`.

2.3 Microservices

Microservices is an architectural style where an application is broken into small, independent services that communicate over a network (e.g., via REST APIs). Each service focuses on a single business function, runs in its own process, and can be deployed independently. Benefits include:

- **Scalability**: Scale individual services as needed.
- **Flexibility**: Use different technologies for different personal data like name, email address, etc.
- **Resilience**: Failure in one service doesn't necessarily affect others.

Example: An e-commerce application might have separate microservices for user management, order processing, and payment handling.

In Java, Spring Boot is ideal for microservices because it supports modular development, containerization (e.g., Docker), and integration with tools like Spring Cloud for service discovery and load balancing.

Author: Anand Prakash
Date: 08 June 2025
Email: adprakash2006@gmail.com

3 Complete Flow: Client to Web Server to Database Server

Let's walk through the flow of a web-based application built with Spring Boot, exposing a REST API, and using a microservices architecture. We'll use a simple example: a user management system where a client retrieves user details.

3.1 Client Request

- **Who**: The client is typically a web browser, mobile app, or another system.
- **What**: The client sends an HTTP request to the server. For example, a GET request to `http://localhost:8080/api/users/123` to fetch user details.
- **How**: The request is sent over the internet using HTTP/HTTPS. The URL specifies the endpoint (`/api/users/123`), and headers may include metadata (e.g., authentication tokens).

Example:

```
GET /api/users/123 HTTP/1.1
Host: localhost:8080
Authorization: Bearer <token>
Accept: application/json
```

3.2 Web Server (Spring Boot Application)

The web server, running a Spring Boot application, receives the request. Here's what happens:

- **Embedded Server**: Spring Boot's embedded Tomcat or Jetty server listens on a port (e.g., 8080) and routes the request to the appropriate handler.
- **Routing**: Spring's `DispatcherServlet` maps the request to a controller based on the URL and HTTP method. For example:

```
@RestController
@RequestMapping("/api/users")
public class UserController {
    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        // Logic to fetch user
    }
}
```

- **Business Logic**:
  - The controller calls a **service** layer, which contains the business logic. For example:

Author: Anand Prakash
Date: 08 June 2025
Email: adprakash2006@gmail.com

```java
@Service
public class UserService {
    private final UserRepository repository;

    public UserService(UserRepository repository) {
        this.repository = repository;
    }

    public User getUser(Long id) {
        return repository.findById(id)
                .orElseThrow(() -> new UserNotFoundException(id));
    }
}
```

- The service interacts with a **repository** (data access layer) to query the database.
- **Dependency Injection**: Spring Boot injects dependencies (e.g., `UserRepository` into `UserService`) using annotations like `@Autowired` or constructor injection.

3.3 Database Server

The database server stores the application's data (e.g., user records). Here's how it fits into the flow:

- Repository Layer: The repository, typically using **Spring Data JPA**, translates Java method calls into SQL queries. For example:

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // Spring Data JPA generates queries automatically
}
```

- **Database Connection**: Spring Boot configures a connection to the database (e.g., PostgreSQL, MySQL) using properties in `application.properties`:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/mydb
spring.datasource.username=admin
spring.datasource.password=secret
spring.jpa.hibernate.ddl-auto=update
```

- **Query Execution**: The repository executes a query (e.g., `SELECT * FROM users WHERE id = 123`) via a **JDBC** driver or **Hibernate** (an ORM). The database returns the result (e.g., user data).
- **Data Mapping**: Hibernate maps the database row to a Java object (e.g., a `User` entity):

Author: Anand Prakash
Date: 08 June 2025
Email: adprakash2006@gmail.com

```java
@Entity
public class User {
    @Id
    private Long id;
    private String name;
    private String email;
    // Getters and setters
}
```

3.4 Response to Client

- **Data Transformation**: The service returns the `User` object to the controller, which converts it to JSON using Spring's **Jackson** library.
- **HTTP Response**: The controller sends an HTTP response back to the client:

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

{
  "id": 123,
  "name": "Alice",
  "email": "alice@example.com"
}
```

- **Client Rendering**: The client (e.g., a web browser) receives the JSON and renders it (e.g., displays user details on a webpage).

3.5 Microservices Context

In a microservices architecture, the user management system might be one of many services. Other services (e.g., order service, payment service) run independently, each with its own Spring Boot application and database. They communicate via REST APIs or message queues (e.g., Kafka, RabbitMQ). For example:

- The client requests user details from the user service (`/api/users/123`).
- The user service might call the order service (`/api/orders?userId=123`) to fetch related data.
- **Service Discovery** (e.g., Spring Cloud Netflix Eureka) helps services find each other.
- **API Gateway** (e.g., Spring Cloud Gateway) routes client requests to the appropriate microservice, handling authentication and load balancing.

Author: Anand Prakash
Date: 08 June 2025
Email: adprakash2006@gmail.com

4 Complete Flow Diagram

Here's a simplified flow for a GET `/api/users/123` request in a Spring Boot microservice:

1. **Client**: Browser sends `GET /api/users/123` to `http://localhost:8080`.
2. **API Gateway** (optional): Routes the request to the user service.
3. **Web Server (Spring Boot)**:
   - Tomcat receives the request.
   - `DispatcherServlet` routes to `UserController`.
   - `UserController` calls `UserService`.
   - `UserService` calls `UserRepository`.
4. **Database Server**:
   - `UserRepository` queries the database (e.g., PostgreSQL).
   - Database returns user data.
5. **Web Server**:
   - `UserRepository` maps data to a `User` object.
   - `UserService` returns the `User` to `UserController`.
   - `UserController` converts `User` to JSON.
6. **Client**: Receives JSON response and displays it.

5 Example: Radar Simulation Microservice

Let's relate this to the Java radar simulation microservice:

- **Client**: A web app sends a GET request to `/api/radar/power` to fetch power vs. distance data.
- **Spring Boot Web Server**:
  - The `RadarController` handles the request:

```
@GetMapping("/power")
public PowerResponse getPowerVsDistance() {
    RadarSystem radar = new RadarSystem();
    AtmosphericModel atmosphere = new AtmosphericModel(-39e-6);
    double[] distances = linspace(1e3, 3e6, 1000);
    RadarCoverageCalculator calculator = new RadarCoverageCalculator
       (radar, atmosphere, distances);
    // Compute and return data
}
```


Author: Anand Prakash
Date: 08 June 2025
Email: adprakash2006@gmail.com

- No database is used in this case; calculations are performed in-memory using the `RadarCoverageCalculator`.
- **Response**: The controller returns a JSON object with distances, received power, and dB values.
- **Client**: The web app uses the JSON to plot a graph (e.g., with Chart.js).

If a database were added (e.g., to store radar parameters), the flow would include a `RadarRepository` querying a database, as described above.

6 Additional Considerations

- **Security**: Use Spring Security to add authentication (e.g., JWT tokens) and authorization to protect endpoints.
- **Error Handling**: Handle exceptions (e.g., user not found) with `@ExceptionHandler` to return meaningful error responses.
- **Scalability**: Deploy microservices in containers (e.g., Docker) and use Kubernetes for orchestration.
- **Monitoring**: Use Spring Boot Actuator for health checks and metrics.
- **Caching**: Use Spring Cache to store frequently accessed data (e.g., user details).

7 Summary

- **Spring Boot**: Simplifies Java web development with auto-configuration and embedded servers.
- **REST API**: Enables communication between clients and servers using HTTP and JSON.
- **Microservices**: Breaks applications into independent, scalable services.
- **Flow**: Client → API Gateway (optional) → Spring Boot Web Server (Controller → Service → Repository) → Database → Response back to Client.

This flow ensures modularity, scalability, and maintainability, making Spring Boot a powerful choice for web-based applications and microservices in Java.

Author: Anand Prakash
Date: 08 June 2025
Email: adprakash2006@gmail.com