**Javith’s 3-Month Tech Upgrade Roadmap**

**🔹 Month 1: Foundation & Positioning**

**Goal**: Modernize your backend skills and prepare your personal brand.

**✅ Skills to Learn**

* **Spring Boot Advanced**: RESTful APIs, exception handling, validation, and testing.
* **Microservices Architecture**: Service registry, config server, circuit breaker.
* **GitHub Portfolio Setup**: Start with 2–3 mini projects (e.g., Employee API, Order Service).

**📘 Resources**

* Spring Boot Masterclass (Udemy or Coursera)
* GitHub + LinkedIn optimization guides

**🧠 Weekly Targets**

* Week 1: Spring Boot deep dive
* Week 2: Microservices with Spring Cloud
* Week 3: Build & deploy one REST API
* Week 4: Polish GitHub + update LinkedIn headline

**🔹 Month 2: Cloud & DevOps**

**Goal**: Become cloud-ready and deployment-savvy.

**✅ Skills to Learn**

* **AWS or Azure Fundamentals**: EC2, S3, IAM, Lambda, API Gateway
* **Docker & Kubernetes**: Containerize your microservices
* **CI/CD Pipelines**: Jenkins or GitHub Actions

**📘 Resources**

* AWS Cloud Practitioner or Azure Fundamentals course
* Docker + K8s hands-on labs (Katacoda or Play with Docker)

**🧠 Weekly Targets**

* Week 1: Cloud basics + deploy REST API to cloud
* Week 2: Dockerize your app
* Week 3: Learn Kubernetes basics
* Week 4: Set up CI/CD pipeline

**🔹 Month 3: System Design & Interview Prep**

**Goal**: Think like an architect and ace interviews.

**✅ Skills to Learn**

* **System Design Patterns**: Load balancing, caching, sharding, queues
* **API Design Best Practices**: Versioning, throttling, security
* **Behavioral Interview Prep**: STAR method, leadership stories

**📘 Resources**

* Grokking the System Design Interview
* Tech Dummies YouTube (for mock interviews)

**🧠 Weekly Targets**

* Week 1: Design scalable REST API system
* Week 2: Practice 3 system design problems
* Week 3: Mock interviews with peers or mentors
* Week 4: Apply to 5 top companies

**🎯 Bonus Tips**

* **Blog Weekly**: Share what you learn—build visibility.
* **Connect with Recruiters**: Target roles like “Senior Java Developer,” “Cloud Engineer,” “API Architect.”
* **Stay Consistent**: Even 1 hour/day will compound into mastery.

**Step-by-Step Setup Guide for Month 1 Tasks**

**🔹 1. Install Java Development Kit (JDK)**

* Download **JDK 17** (recommended for Spring Boot 3.x) from Oracle or AdoptOpenJDK.
* After installation, verify with:

bash

java -version

**🔹 2. Install IntelliJ IDEA (or Eclipse)**

* Recommended: IntelliJ IDEA Community Edition
* Features: Smart code completion, Spring Boot support, Git integration

**🔹 3. Install Maven**

* Spring Boot uses Maven for dependency management.
* Download from Apache Maven
* Verify with:

bash

mvn -version

**🔹 4. Install Git**

* Download from Git SCM
* Set up GitHub account and configure Git:

bash

git config --global user.name "javith"

git config --global user.email "your-email@example.com"

**🔹 5. Install Postman**

* For testing REST APIs
* Download from Postman

**🔹 6. Install Docker Desktop (Optional for Week 3)**

* Download from Docker
* Helps you containerize and deploy your apps

**🔹 7. Create Your First Spring Boot Project**

* Use Spring Initializr
  + Project: Maven
  + Language: Java
  + Spring Boot: 3.x
  + Dependencies: Spring Web, Spring Data JPA, H2 Database, Lombok

**🔹 8. Set Up GitHub Repository**

* Create a new repo: employee-management-api
* Push your Spring Boot project:

Bash

Username:javithrasid-hash

git init

git remote add origin https://github.com/your-username/employee-management-api.git

git add .

git commit -m "Initial commit"

git push -u origin master

Refined:

# 1. Initialize the repo

git init

# 2. Enable long path support (avoids filename too long errors)

git config core.longpaths true

# 3. Set line ending handling (avoids LF/CRLF warnings)

git config core.autocrlf true

# 4. Add remote origin

git remote add origin <https://github.com/your-username/employee-management-api.git>

# 5. Create a .gitignore to exclude system files

echo "AppData/" >> .gitignore

echo "\*.jtx" >> .gitignore

echo "\*.lst" >> .gitignore

echo "node\_modules/" >> .gitignore # if using Node.js

# 6. Stage only safe files

git add .

# 7. Commit your changes

git commit -m "Initial commit"

# 8. Push to GitHub (use main if your repo defaults to main)

git push -u origin master

**🔹 9. Test Your API**

* Run your app in IntelliJ
* Use Postman to test endpoints like:
  + GET /employees
  + POST /employees
  + PUT /employees/{id}
  + DELETE /employees/{id}

**🔹 10. Document with Swagger**

* Add Swagger dependency:

xml

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-ui</artifactId>

<version>1.6.14</version>

</dependency>

* Access docs at: http://localhost:8080/swagger-ui.html

Once you finish this setup, you’ll be ready to start building and deploying your first project. Want help designing your Employee API schema or writing your first controller class? I can walk you through it.

**How your Spring Boot application executes step by step when you run it inside IntelliJ, based on your folder structure and class layout.**

springboot-rest-api/

├── src/

│ ├── main/

│ │ ├── java/

│ │ │ └── com/

│ │ │ └── spring/

│ │ │ └── api/

│ │ │ ├── controller/ # REST controllers (API layer)

│ │ │ │ └── BookController.java

│ │ │ ├── service/ # Interfaces for business logic

│ │ │ │ └── BookService.java

│ │ │ ├── service/impl/ # Service implementations

│ │ │ │ └── BookServiceImpl.java

│ │ │ ├── provider/ # DAO layer (JPA Repositories)

│ │ │ │ └── ProviderApplication.java

│ │ │ │ └── ServletInitializer.java

│ │ │ ├── model/ # Domain models / entities

│ │ │ │ └── Book.java

│ │ └── resources/

│ │ ├── application.properties # Config file

│ │ └── static/ # Static assets (if any)

│ │ └── templates/ # Thymeleaf templates (if used)

│ └── test/

│ └── java/

│ └── com.spring.api.provider/

│ └── ProviderApplicationTests.java # Unit and integration tests

├── pom.xml # Build configuration

└── README.md

**🚀 Step-by-Step Execution Flow**

**🟢 1. Start with ProviderApplication.java**

This is your entry point:

java

@SpringBootApplication

public class ProviderApplication {

public static void main(String[] args) {

SpringApplication.run(ProviderApplication.class, args);

}

}

* The compiler starts here.
* @SpringBootApplication triggers component scanning, auto-configuration, and bootstrapping.
* It scans all subpackages under com.spring.api—so your controller, service, impl, and model packages are included.

**🟡 2. Spring Boot Auto-Configuration**

* Spring Boot configures embedded Tomcat (default port 8080 from application.properties).
* It sets up the web context, loads beans, and prepares the application for HTTP requests.

**🟠 3. Component Scanning and Bean Creation**

Spring scans for annotations like:

* @RestController → BookController is registered as a web controller.
* @Service → BookServiceImpl is registered as a service bean.
* @Autowired → Injects BookServiceImpl into BookController.

**🔵 4. @PostConstruct in BookServiceImpl Runs**

java

@PostConstruct

public void init() {

books.add(new Book(1, "Spring Boot in Action", "Craig Walls"));

books.add(new Book(2, "Effective Java", "Joshua Bloch"));

}

* This method runs **after the bean is created**.
* It populates your in-memory books list before any API call is made.

**🟣 5. Application Ready for Requests**

Once everything is initialized:

* You can hit endpoints like GET http://localhost:8080/api/books/1
* The flow is:
  1. Request hits BookController
  2. Controller calls BookService.findBookById(id)
  3. BookServiceImpl searches the list and returns the result
  4. Spring serializes the Book object into JSON and sends it back

**🧠 Bonus: ServletInitializer**

Your ServletInitializer.java is only used if you deploy the app as a WAR to an external servlet container (like Tomcat or WebLogic). For IntelliJ and embedded Tomcat, it’s not invoked.

Would you like me to generate a visual flow diagram of this execution path or help you add logging to trace each step in the console? I can also show how to extend this with a database layer using Spring Data JPA.