



CORPOSITORY

— A Yubi Company —

Monday 3-June-2024

Subham Modi

Chennai

Dear Subham,

Further to your candidature & interactions with us, It is my pleasure to extend an offer to you for the position of **Software Engineer 1**, at Corpository (Bluevine Technologies Private Limited).

We would like you to join us on or before **Wednesday 05-June-2024**. Your work location will be **Chennai** and you may be required to travel to other locations as deemed by business needs.

Your Cost to company will be **INR 12,72,000** as per the break up provided in Annexure A below.

This offer letter is not an employment contract. You will be provided with an employment agreement when you join. To indicate your acceptance of this offer, please acknowledge and accept the offer within 2 business days. Please note that your salary details are strictly private and confidential and should not be disclosed or discussed with others.

We are very pleased to have you join Corpository and look forward to working with you. We look forward to your continued contribution and commitment to ensuring the success of the company in the coming years.

Janaki Parikh

Manager – Human Resources
Bluevine Technologies Private Limited

BLUEVINE TECHNOLOGIES PRIVATE LIMITED

CIN: U72900GJ2015PTC084737

Office No.2001-2020, 20th Floor, B – Block, Navratna Corporate Park, Ambli Bopal Road, Ahmedabad-380058

Email: info@corpository.com | **Web:** www.corpository.com

A project report on

BUILDING MICROSERVICES APPLICATION USING SPRING BOOT

Submitted in partial fulfillment for the award of the degree of

**Bachelor of Technology in Computer Science and Engineering with
Specialization with Business System**

by

SUBHAM MODI (20BBS0166)



VIT[®]

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

June, 2024

BUILDING MICROSERVICES APPLICATION USING SPRING BOOT

Submitted in partial fulfillment for the award of the degree of

**Bachelor of Technology in Computer Science and Engineering with
Specialization with Business System**

by

SUBHAM MODI (20BBS0166)



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

June, 2024

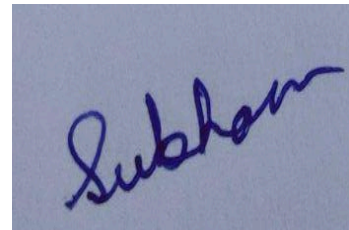
DECLARATION

I hereby declare that the thesis entitled “BUILDING MICROSERVICES APPLICATION USING SPRING BOOT ” submitted by me, for the award of the degree of Specify the name of the degree VIT is a record of bonafide work carried out by me under the supervision of Dani Vijay (Yubi Employee)

I further declare that the work reported in this thesis has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place: Vellore

Date: 29 June, 2024

A handwritten signature in blue ink, appearing to read "Subham", is written on a light blue background.

ABSTRACT

Microservices architecture has gained significant popularity due to its ability to enhance scalability, maintainability, and agility in modern software development. Spring Boot, a widely adopted Java-based framework, provides robust support for building microservices, offering developers a comprehensive ecosystem to create scalable and resilient applications.

This paper explores the fundamental principles and practical considerations involved in designing and implementing a microservices-based application using Spring Boot. It begins by elucidating the core concepts of microservices architecture, emphasizing the advantages of decoupling monolithic applications into smaller, independent services. Key architectural patterns such as service discovery, load balancing, and fault tolerance are examined in the context of Spring Boot's capabilities, illustrating how these patterns contribute to building resilient microservices.

Furthermore, the paper delves into the implementation details, covering essential components such as Spring Boot's dependency management, auto-configuration, and embedded server capabilities. It discusses best practices for designing RESTful APIs within microservices, leveraging Spring Boot's powerful features such as Spring Data and Spring Cloud for seamless integration with databases and external services.

The practical aspects of deployment and monitoring are also addressed, highlighting strategies for containerization with Docker and orchestration using Kubernetes. Additionally, the paper explores methods for ensuring operational excellence through centralized logging, metrics collection, and distributed tracing, all facilitated by Spring Boot's support for industry-standard observability tools.

To illustrate these concepts, a sample application is developed throughout the paper, demonstrating step-by-step how to implement microservices using Spring Boot. This application showcases various aspects including service registration with Netflix Eureka, load balancing with Ribbon, and fault tolerance with Hystrix. Code snippets and configuration examples are provided to guide developers in effectively leveraging Spring Boot's capabilities to build robust microservices architectures.

In conclusion, this paper serves as a comprehensive guide for developers and architects looking to adopt microservices architecture using Spring Boot. By combining theoretical insights with practical implementation guidelines, it equips readers with the knowledge and skills necessary to successfully design, develop, deploy, and maintain microservices-based applications that are scalable, resilient, and efficient.

ACKNOWLEDGEMENT

It is my pleasure to express with deep sense of gratitude to Dani Vijay, Manager, for his constant guidance, continual encouragement, understanding; more than all, he taught me patience in my endeavor. My association with him / her is not confined to academics only, but it is a great opportunity on my part of work with an intellectual and expert in the field of Development.

I would like to express my gratitude to Dr. G Viswanathan, Mr. Sankar Viswanathan, Dr. Sekar Viswanathan, Dr.G V Selvam, DR. V. S. Kanchana Bhaaskaran, Dr. Partha Sharathi Mallick, and Dr. Ramesh Babu K, School of Computer Science and Engineering (SCOPE), for providing with an environment to work in and for his inspiration during the tenure of the course.

In jubilant mood I express ingeniously my whole-hearted thanks to Dr. R. Sujatha (Program Chair and Associate Professor) , all teaching staff and members working as limbs of our university for their not-self-centered enthusiasm coupled with timely encouragements showered on me with zeal, which prompted the acquirement of the requisite knowledge to finalize my course study successfully. I would like to thank my parents for their support.

It is indeed a pleasure to thank my friends who persuaded and encouraged me to take up and complete this task. At last but not least, I express my gratitude and appreciation to all those who have helped me directly or indirectly toward the successful completion of this project.

Place: Vellore

Date:29 June, 2024

Subham Modi

CONTENTS

CONTENTS	iii
LIST OF FIGURES	vi
LIST OF TABLES	vii
LIST OF ACRONYMS	viii
CHAPTER 1 INTRODUCTION	1
1.1 WHY ARE MICROSERVICES USED?	1
1.2 WHAT EXACTLY ARE MICROSERVICES?	2
1.3 DIFFERENCES BETWEEN TRADITIONAL ARCHITECTURE AND MICROSERVICES	3
1.4 ARCHITECTURE OF MICROSERVICES	3
1.5 FEATURES OF MICROSERVICES	4
1.6 ADVANTAGES OF MICROSERVICES	5
CHAPTER 2 THE 12-FACTOR APPROACH FOR MICROSERVICES	6
2.1 CODEBASE	6
2.2 DEPENDENCIES	7
2.3 CONFIG	7
2.4 BACKING SERVICES	8
2.5 BUILD, RELEASE, RUN	8
2.6 PROCESSES	8
2.7 PORT BINDING	8
2.8 CONCURRENCY	9
2.9 DISPOSABILITY	9
2.10 DEV/PROD PARITY	10
2.11 LOGS	10
2.12 ADMIN PROCESSES	10
CHAPTER 3 MICROSERVICES AND APIs	11

3.1 WHAT IS AN API?	11
3.2 WHAT ARE APIs USED FOR?	12
3.3 REST APIs	12
3.4 WHERE ARE APIs USED IN MICROSERVICES?	13
3.5 MICROSERVICES VS API	14
CHAPTER 4 INTRODUCTION TO SPRING BOOT	16
4.1 WHAT IS SPRING BOOT	16
4.2 ADVANTAGES OF SPRING BOOT	17
4.3 GOALS OF SPRING BOOT	18
4.4 FEATURES OF SPRING BOOT	18
4.5 SPRING vs SPRING BOOT	19
4.6 SPRING BOOT vs SPRING MVC	20
CHAPTER 5 SPRING BOOT ARCHITECTURE	22
5.1 SPRING BOOT FLOW ARCHITECTURE	23
CHAPTER 6 SPRING BOOT PROJECT COMPONENTS	24
6.1 SPRING BOOT ANNOTATIONS	24
6.1.1 CORE SPRING FRAMEWORK ANNOTATIONS	24
6.1.2 SPRING FRAMEWORK STEREOTYPE ANNOTATIONS	25
6.1.3 SPRING BOOT ANNOTATIONS	25
6.1.4 SPRING MVC AND REST ANNOTATIONS	26
6.2 SPRING BOOT DEPENDENCY MANAGEMENT	27
6.2.1 MAVEN DEPENDENCY MANAGEMENT SYSTEM	27
6.3 SPRING BOOT APPLICATION PROPERTIES	29
6.3.1 SPRING BOOT PROPERTY CATEGORIES	29
6.3.2 APPLICATION PROPERTIES TABLE	30
6.4 SPRING BOOT STARTERS	34
6.4.1 THIRD-PARTY STARTERS	34
6.4.2 SPRING BOOT PRODUCTION STARTERS	38

6.4.3 SPRING BOOT TECHNICAL STARTERS	39
6.5 SPRING BOOT STARTER PARENT	39
6.6 SPRING BOOT STARTER WEB	40
6.7 SPRING BOOT STARTER DATA JPA	41
6.7.1 SPRING DATA JPA FEATURES	41
6.7.2 SPRING DATA REPOSITORY	42
6.7.3 SPRING BOOT STARTER DATA JPA	43
6.7.4 HIBERNATE vs JPA	43
6.8 SPRING BOOT STARTER ACTUATOR	43
6.8.1 SPRING BOOT ACTUATOR FEATURES	44
6.8.2 SPRING BOOT ACTUATOR ENDPOINTS	44
6.8.3 SPRING BOOT ACTUATOR PROPERTIES	46
6.9 SPRING BOOT STARTER TEST	46
6.10 SPRING BOOT DEVTOOLS	47
6.10.1 SPRING BOOT DEVTOOLS FEATURES	47
6.11 MULTI-MODULE PROJECT	49
6.12 SPRING BOOT PACKAGING	50
6.13 SPRING BOOT AUTO-CONFIGURATION	51
6.13.1 DISABLE AUTO-CONFIGURATION CLASSES	52
6.13.2 NEED OF AUTO-CONFIGURATION	52
CHAPTER 7 INTRODUCTION TO RESTFUL WEB SERVICES	54
7.1 RESTFUL SERVICE CONSTRAINTS	55
7.2 ADVANTAGES OF RESTFUL WEB SERVICES	55
7.3 RESTFUL WEB SERVICES BEST PRACTICES	55
CHAPTER 8 CONCLUSION & FUTURE WORK	57
REFERENCES	58

LIST OF FIGURES

Figure 1.1: Challenges of Monolithic Architecture	2
Figure 1.2: Representation of Microservices	2
Figure 1.3: Differences Between Monolithic Architecture and Microservices	3
Figure 1.4: Architecture of Microservices	3
Figure 1.5: Features of Microservices	4
Figure 1.6: Advantages of Microservices	5
Figure 3.1: HTTP methods	11
Figure 3.2: E-commerce application	13
Figure 3.3: Microservices vs API	14
Figure 4.1: What is Spring Boot	16
Figure 5.1: Spring Boot Architecture	22
Figure 5.2: Spring Boot Flow Architecture	23
Figure 6.1: spring-boot-starter-parent	28
Figure 6.2: Using the <scope> tag	28
Figure 6.3: Example of application.properties	29
Figure 6.4: Spring Boot Packaging	50
Figure 6.5: Configuring dispatcher servlet	52
Figure 6.6: Configuring datasource	53
Figure 6.7: Configuring entity manager factory	53
Figure 6.8: Configuring transaction manager	53

LIST OF TABLES

Table 4.1: Spring vs Spring Boot	19
Table 4.2: Spring Boot vs Spring MVC	21
Table 6.1: Application Properties table	30
Table 6.2: Third-Party Starters	35
Table 6.3: Spring Boot Production Starters	38
Table 6.4: Spring Boot Technical Starters	39
Table 6.5: Spring Boot Actuator Endpoints	44
Table 6.6: Spring MVC Actuator Endpoints	46

LIST OF ACRONYMS

REST	Representational State Transfer
API	Application programming interface
SAAS	Software as a service
OS	Operating System
CI/CD	Continuous Integration and Continuous Delivery/Continuous Deployment
HTTP	Hypertext Transfer Protocol
UNIX	UNiplexed Information Computing System
NoSQL	Not Only SQL
CRUD	Create, Read, Update and Delete
RAD	Rapid Application Development
XML	Extensible Markup Language
STS	Spring Tool Suite
IDE	Integrated Development Environment
JPA	Java Persistence API
ORM	Object–Relational Mapping
JAR	Java ARchive
WAR	Web Application Resource or Web application ARchive
CLI	Command-Line Interface
JSON	JavaScript Object Notation
IoC	Inversion of Control
Java EE	Java Platform, Enterprise Edition
MVC	Model-View-Controller
DAO	Data Access Object
JSP	JavaServer Pages
URL	Uniform Resource Locator
UTF	Unicode Transformation Format

SMTP	Simple Mail Transfer
Protocol SSL	Secure Sockets Layer
JMS	Java Message Service
JDBC	Java Database Connectivity
HATEOAS	Hypermedia as the Engine of Application
State JAX-RS	Java API for RESTful Web Services
LDAP	Lightweight Directory Access
Protocol AOP	Aspect-Oriented Programming
AMQP	Advanced Message Queuing
Protocol JTA	Java Transaction API
jOOQ	jOOQ Object Oriented Querying
SSH	Secure Shell or Secure Socket
Shell	
SQL	Structured Query Language
JMX	Java Management Extensions
JDWP	Java Debug Wire Protocol
EJP	Enterprise Java Bean
SOAP	Simple Object Access
Protocol IT	Information Technology

Chapter 1

Introduction to Microservices

Microservices, also called microservice architecture, is a style of architecture that builds an application as a group of services that are:

- Easy to maintain and test
- Loosely connected
- Can be used on its own
- Organized around how a business works
- It is owned by a small group

1.1 WHY ARE MICROSERVICES USED?

Microservices were used to overcome the challenges of monolithic architecture that prevailed initially in the market and also enables to deploy independent services. So, before we move on to what is Microservices in simple words, let's see the architecture that prevailed initially in the market i.e. the Monolithic Architecture. In layman's terms, the Monolithic Architecture is similar to a big container wherein all the software components of an application are assembled together and tightly packaged.

Listed below are the challenges of Monolithic Architecture:

- **Inflexible** – Monolithic applications cannot be built using different technologies
- **Unreliable** – Even if one feature of the system does not work, then the entire system does not work
- **Unscalable** – Applications cannot be scaled easily since each time the application needs to be updated, the complete system has to be rebuilt
- **Blocks Continous Development** – Many features of the applications cannot be built and deployed at the same time
- **Slow Development** – Development in monolithic applications takes a lot of time to be built since each and every feature has to be built one after the other
- **Not Fit For Complex Applications** – Features of complex applications have tightly coupled dependencies

The above challenges were the main reasons that led to the evolution of microservices.

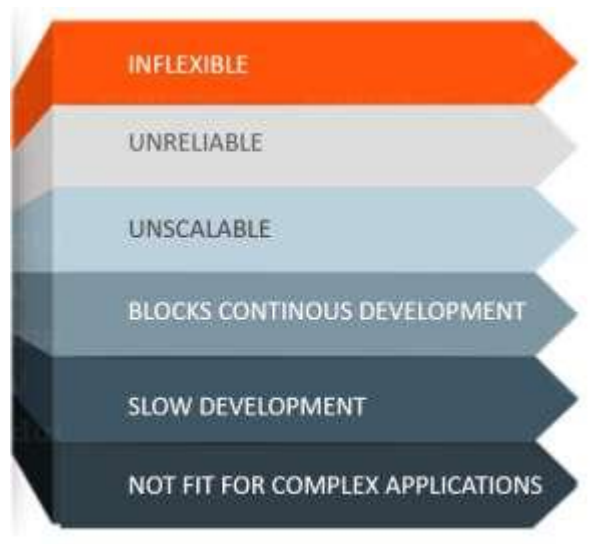


Figure 1.1: Challenges of Monolithic Architecture

1.2 WHAT EXACTLY ARE MICROSERVICES?

Microservices is an architectural style that structures an application as a collection of small autonomous services, modelled around a business domain.



Figure 1.2: Representation of Microservices

In the given Architecture, each service is self-contained and implements a single business capability.

1.3 DIFFERENCES BETWEEN TRADITIONAL ARCHITECTURE AND MICROSERVICES

Let us consider an example of an E-commerce application as a use case to understand the difference between Monolithic Architecture and Microservices.

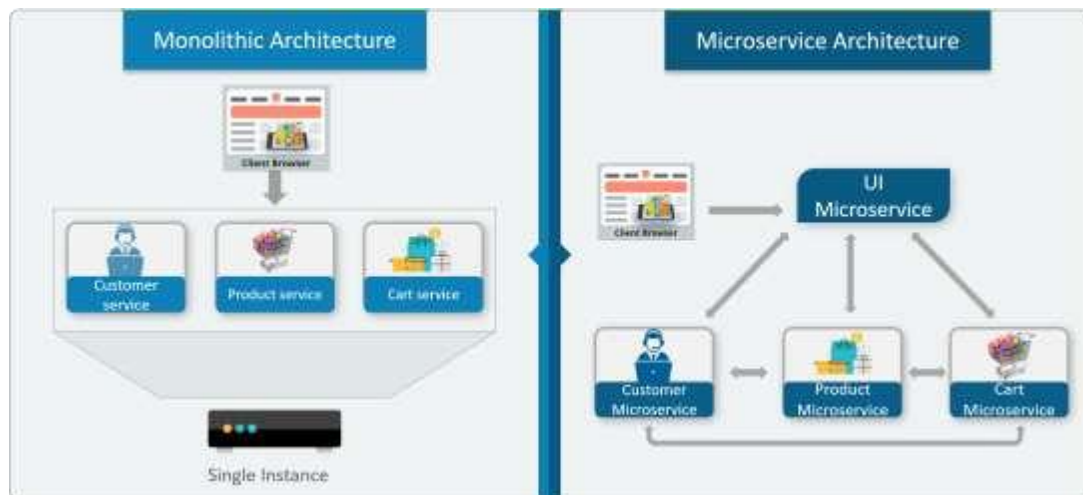


Figure 1.3: Differences Between Monolithic Architecture and Microservices

The main difference we observe in the above diagram is that all the features initially were under a single instance sharing a single database. But then, with microservices, each feature was allotted a different microservice, handling its own data, and performing different functionalities.

1.4 ARCHITECTURE OF MICROSERVICES

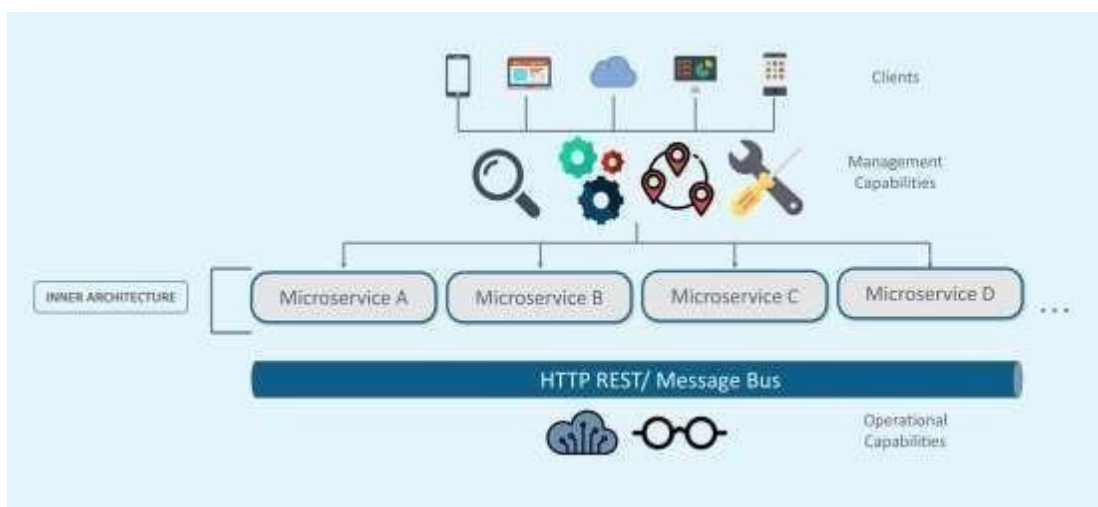


Figure 1.4: Architecture of Microservices

- Different clients from different devices try to use different services like search, build, configure and other management capabilities
- All the services are separated based on their domains and functionalities and are further allotted to individual microservices
- These microservices have their own load balancer and execution environment to execute their functionalities & at the same time capture data in their own databases
- All the microservices communicate with each other through a stateless server which is either REST or Message Bus
- Microservices know their path of communication with the help of Service Discovery and perform operational capabilities such as automation, monitoring
- Then all the functionalities performed by microservices are communicated to clients via API Gateway
- All the internal points are connected from the API Gateway. So, anybody who connects to the API Gateway automatically gets connected to the complete system

1.5 FEATURES OF MICROSERVICES



Figure 1.5: Features of Microservices

- **Decoupling** – Services within a system are largely decoupled. So the application as a whole can be easily built, altered, and scaled
- **Componentization** – Microservices are treated as independent components that can be easily replaced and upgraded
- **Business Capabilities** – Microservices are very simple and focus on a single

capability

- **Autonomy** – Developers and teams can work independently of each other, thus increasing speed
- **Continuous Delivery** – Allows frequent releases of software, through systematic automation of software creation, testing, and approval
- **Responsibility** – Microservices do not focus on applications as projects. Instead, they treat applications as products for which they are responsible
- **Decentralized Governance** – The focus is on using the right tool for the right job. That means there is no standardized pattern or any technology pattern. Developers have the freedom to choose the best useful tools to solve their problems
- **Agility** – Any new feature can be quickly developed and discarded again

1.6 ADVANTAGES OF MICROSERVICES

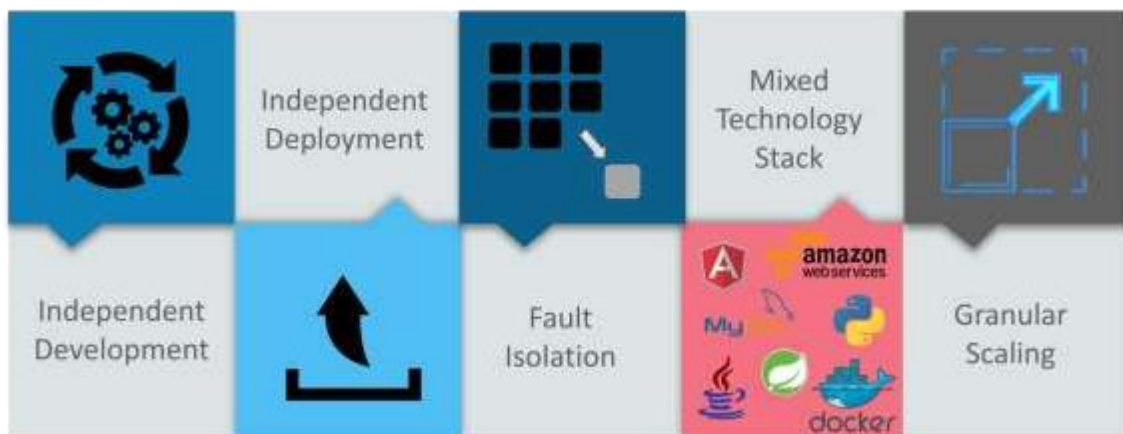


Figure 1.6: Advantages of Microservices

- **Independent Development** – All microservices can be easily developed based on their individual functionality
- **Independent Deployment** – Based on their services, they can be individually deployed in any application
- **Fault Isolation** – Even if one service of the application does not work, the system still continues to function
- **Mixed Technology Stack** – Different languages and technologies can be used to build different services of the same application
- **Granular Scaling** – Individual components can scale as per need, there is no need to scale all components together

Chapter 2

The 12-Factor Approach for Microservices

The Twelve-Factor App methodology was created by developers at Heroku who observed that successful apps share some common principles. While the original principles were based on apps developed on the Heroku platform, they're technology- and language-agnostic. They work well for breaking monolithic apps into microservices and for developing cloud-native apps from scratch using microservices. The following provides an overview of using the Twelve-Factor App methodology with some information specific to microservices.

In the modern era, the software is commonly delivered as a service: called web apps, or software-as-a-service (SAAS). The twelve-factor app is a methodology for building SAAS apps that:

- Use declarative formats for setup automation, to minimize time and cost for new developers joining the project;
- Have a clean contract with the underlying operating system, offering maximum portability between execution environments;
- Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration;
- Minimize divergence between development and production, enabling continuous deployment for maximum agility;
- And can scale up without significant changes to tooling, architecture, or development practices.

The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

2.1 CODEBASE

One codebase tracked in revision control, many deploys

There is only one codebase per app — in microservices architecture, it's one codebase per service — but there can be numerous deploys. A deploy is a running instance of the app. This is usually a production site and one or more staging sites. A copy of an app running in a developer's local development environment is also considered a deploy. The codebase must

be the same across all deployments, although different versions may be active in each deploy. For this reason, it's essential to track the app in a version control system, such as Subversion and Git.

2.2 DEPENDENCIES

Explicitly declare and isolate dependencies

Declare all dependencies precisely via a dependency declaration manifest. Use a dependency isolation tool during execution to ensure that no implicit dependencies enter from the surrounding system. This applies to both production and development. Don't rely on the implicit existence of any system-wide site-packages or site tools. While these tools may exist on most systems, there's no guarantee they'll exist on the systems where the app runs in the future. Nor is it a sure thing that the version found on a future system will be compatible with the app.

2.3 CONFIG

Store config in the environment

All configuration data should be stored separately from the code – in the environment as variables and not in the code repository — and read in by the code at runtime. A separate config file makes it easy to update the config values without touching the actual codebase, eliminating the need for redeploying apps when config values are changed.

The following are some additional best practices to consider:

- Use an environment variable for anything that can change at runtime, and for secrets that shouldn't be committed to a shared repository.
- Use non version-controlled .env files for local development.
- Keep all .env files in a secure storage system so they're available to the development teams, but not committed to the code repository being used.
- Once an app is deployed to a delivery platform, use the platform's mechanism for managing environment variables.

Configs stored as variables are unlikely to be checked into the repository accidentally. Another bonus is that the configs are independent of language and OS.

2.4 BACKING SERVICES

Treat backing services as attached resources

Backing services are usually managed locally by the same systems administrators who deploy the app's runtime. The app may also have services provided and managed by third parties. For microservices, anything external to a service is treated as an attached resource. This ensures that every service is completely portable and loosely coupled to the other resources in the system. In addition, the strict separation increases flexibility during development; developers only need to run the services they're modifying.

2.5 BUILD, RELEASE, RUN

Strictly separate build and run stages

Use a continuous integration/continuous delivery (CI/CD) tool to automate builds and support strict separation of build, release, and run stages. Docker images make it easy to separate the build and run stages. Images should be created from every commit and treated as deployment artifacts. Start the build process by storing the app in source control and then build out its dependencies. Separating the config information enables you to combine it with the build for the release stage. It's then ready for the run stage. Each release should have a unique ID.

2.6 PROCESSES

Execute the app as one or more stateless processes

Ensuring the app is stateless makes it easy to scale a service horizontally by simply adding more instances of that service. Store any stateful data or data that needs to be shared between instances, in a backing service such as a database. The process' memory space or filesystem can be used as a brief, single-transaction cache. Session state data is a good candidate for a datastore that offers time expiration. Never assume that anything cached in memory or on disk will be available on a future request or job.

2.7 PORT BINDING

Export services via port binding

The app should be completely self-contained and not rely on the runtime injection of a

webserver into the execution environment to create a web-facing service. The web app exports HTTP as a service by binding to a port, and listening to requests coming in on that port. Nearly any kind of server software can be run via a process binding to a port and awaiting incoming requests. To make the port binding factor more useful for microservices, allow access to the persistent data owned by a service only by way of the service's API. This prevents implicit service contracts between microservices. It also ensures they can't become tightly coupled. In addition, data isolation allows the developer to choose the type of data store for each service, that best suits its needs.

2.8 CONCURRENCY

Scale-out via the process model

Supporting concurrency means that different parts of an app can be scaled up to meet the need at hand. When you develop the app to be concurrent, you can spin up new instances to the cloud effortlessly. This principle draws from the UNIX model for running service daemons, enabling an app to be architected to handle diverse workloads by assigning each type of work to a process type. To do this, organize processes according to their purpose and then separate them so that they can be scaled up and down according to need. Docker and other containerized services provide service concurrency.

2.9 DISPOSABILITY

Maximize robustness with fast startup and graceful shutdown

An app's processes should be disposable so they can be started, stopped, and redeployed quickly with no loss of data. This facilitates fast elastic scaling, rapid deployment of code and config changes, and robustness of production deploys. Services deployed in Docker containers do this automatically, as it's an inherent feature of containers that they can be stopped and started instantly. The concept of disposable processes means that an app can die at any time, but it won't affect the user—the app can be replaced by other apps, or it can start right up again. Building disposability into your app ensures that the app shuts down gracefully: it should clean up all utilized resources and shut down smoothly. When designed this way, the app comes back up again quickly. Likewise, when processes terminate, they should finish their current request, refuse any incoming request, and exit.

2.10 DEV/PROD PARITY

Keep development, staging, and production as similar as possible

Continuous deployment requires continuous integration based on matching environments to limit deviation and errors. As such, dev, staging, and production should be as similar as possible. Containers work well for this as they enable you to run the exact same execution environment all the way from local development through production. Note: the differences in the underlying data can still cause differences at runtime. Don't use different backing services between development and production, even when adapters theoretically abstract away any differences. Even minor deviations can cause incompatibilities to crop up that can cause code that worked in development or staging to fail in production.

2.11 LOGS

Treat logs as event streams

Stream logs to a chosen location rather than dumping them into a log file. Logs can be directed anywhere. For example, they could be directed to a database in NoSQL, to another service, to a file in a repository, to a log-indexing-and-analysis system, or to a data-warehousing system. Use a log-management solution for routing or storing logs. The process for routing log data needs to be separate from processing log data. Define your logging strategy as part of the architecture standards, so all services generate logs in a similar fashion.

2.12 ADMIN PROCESSES

Run admin/management tasks as one-off processes

The idea here is to separate administrative tasks from the rest of the app to prevent one-off tasks from causing issues with your running apps. Containers make this easy, as you can spin up a container just to run a task and then shut it down. Examples include doing data cleanup, running analytics for a presentation, or turning on and off features for A/B testing. Though the admin processes are separate, you must continue to run them in the same environment and against the base code and config of the app itself. Shipping the admin tasks code alongside the app prevents drift.

Chapter 3

Microservices and APIs

Microservices is an approach to building an application that breaks its functionality into modular components. APIs are part of an application that communicates with other applications. So, APIs can be used to enable microservices. As a result, they make it easier to create software.

3.1 WHAT IS AN API?

An application programming interface, or API, is the part of an application that communicates with other applications. More technically speaking, an API is a group of protocols and methods that define how two applications share and modify each other's data. APIs are necessary for our modern digital infrastructure because they enable standardized and efficient communication between applications which might differ in function and construction. An API sits between a software's core components and the public, and external developers can access certain parts of an application's backend without needing to understand how everything works inside the app. This is what makes an API an interface for programmers.

Now, while building and using applications, CRUD operations are required. The different operations in CRUD are creating a resource, reading a resource, updating a resource and deleting a resource. So, APIs are generally developed by using the RESTful style, and these methods are nothing but the methods of HTTP. The methods associated with the HTTP actions are as shown in the figure:

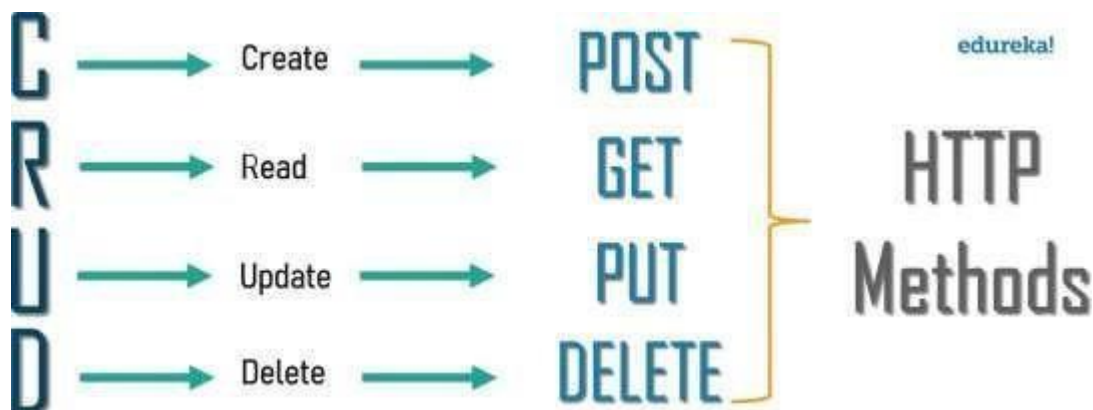


Figure 3.1: HTTP methods

The above methods help in standardizing a way in which actions will be performed on various applications having different interfaces. Also, with the help of these methods, developers can easily understand the inference of the actions taken across the different interfaces.

3.2 WHAT ARE APIs USED FOR?

If software is being used, then APIs are also being used. That's because APIs enable software integrations — they allow otherwise separate software entities to share information and function together.

Imagine you're online shopping and ready to check out. You see that the store you're on gives the option to pay through PayNow, a payment processor, and you already have an account on PayNow.com with your payment info set up. How convenient! Since PayNow is a different company from the store you're currently on, an API facilitates interaction between the store and PayNow. First, the store uses PayNow's payment gateway API to request your payment information. Next, the PayNow API fields the request, validates it, fetches the information from its customer database, and sends it back to the store. Finally, the store uses your card information to complete the transaction. Thanks to PayNow, your store gets all the info it needs to complete your checkout without having to access PayNow's private database itself, and without requiring you to navigate off the store website.

Exchanges like this occur almost any time two separate applications work together. Some other real-world examples include an embedded YouTube video, a travel website using an airline's API to access flight times and prices, a website using one of Google's and/or Facebook's APIs to allow social login, and a navigation app accessing a public transit system's API for real-time transportation data.

3.3 REST APIs

Since an API is more of a concept, programmers can build an API for their app however they please. However, most rely on frameworks to create them. REST, which stands for Representational State Transfer, is a framework for developing APIs, and APIs that conform to REST are called REST APIs. REST APIs are the most common type of API for cross-platform integrations and are also used in microservices.

REST outlines a set of constraints for building APIs that make them efficient and secure.

REST APIs work by fielding HTTP requests and returning responses in JSON (JavaScript Object Notation) format. HTTP (Hypertext Transfer Protocol) is already the standard protocol for web-based data transfer. So, with some knowledge of HTTP, developers can easily learn how to build or interact with a REST API.

3.4 WHERE ARE APIs USED IN MICROSERVICES?

Consider the scenario of an e-commerce application built using Microservices. Assume there are three services, i.e. the customer service, cart service, and products service. Now, these services communicate with each other to process the client's request through the APIs'. So, each of these microservices will have its own APIs' to communicate with the other services.

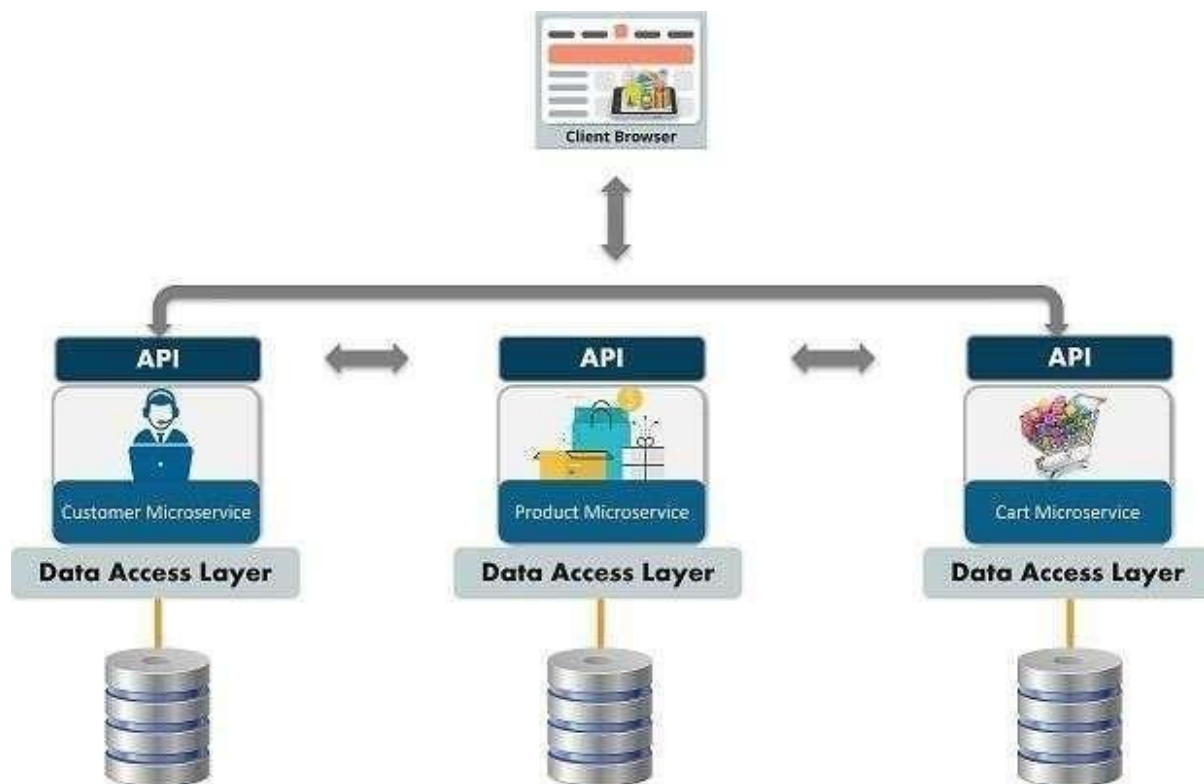


Figure 3.2: E-commerce application

Now, even if one microservice, does not work, then the application will not go down. Instead, only that particular feature will not be working, and once it starts working, APIs' can process the request again and send the required response, back to the client.

3.5 MICROSERVICES VS API

- An API is a part of a web application that communicates with other applications. A software's API defines a set of acceptable requests to be made to the API and responses to these requests.
- A microservice is an approach to building an application that breaks down an application's functions into modular, self-contained programs. Microservices make it easier to create and maintain software.

While different things, microservices and APIs are frequently paired together because services within a microservice use APIs to communicate with each other. Similar to how an application uses a public API to integrate with a different application, one component of a microservice uses a private API to access a different component of the same microservice. Within a microservice, each service has its own API which determines what requests it may receive and how it responds. These APIs typically follow REST principles. Below is a visual example of a basic microservice held together with internal APIs:

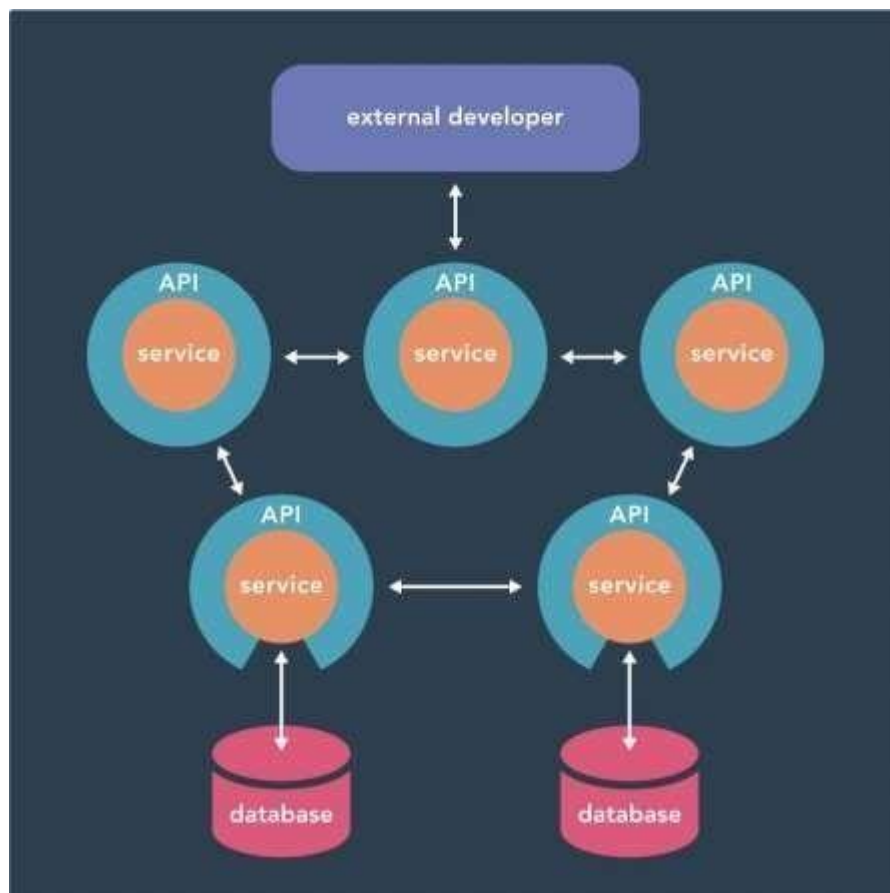


Figure 3.3: Microservices vs API

In the above figure, it can be noted that only one module interacts with third-party developers. In our example, this particular service handles integrations with other applications. So, that particular API is public-facing, while all other APIs in this microservice are private. It's important to note that no two microservices are alike, and all utilize APIs differently. Some might assign multiple APIs to one service, or use a single API for accessing multiple services. It should be noted that not every application follows a one-to-one API-to-service pairing. APIs have uses beyond microservices. Web APIs enable data-sharing between systems, which is necessary for many web applications. Also, APIs can be used internally but without a microservice implementation.

Thus it can be concluded that APIs are a part of microservices and help these services in communicating with each other. However, while communicating with the other services, each service can have its own CRUD operations to store the relevant data in its database. Not only this but while performing CRUD operations, APIs generally accept and return parameters based on the request sent by the user. For example, if the customer wants to know the order details, then product details will be fetched from the product service, the billing address and contact details will be fetched from the customer service, and the product purchased will be fetched from the cart service.

Chapter 4

Introduction to Spring Boot

Spring Boot is an open-source Java-based framework used to create a micro Service. It is developed by the Pivotal Team and is used to build stand-alone and production-ready spring applications. It basically is a Spring module that provides the RAD (Rapid Application Development) feature to the Spring framework.

4.1 WHAT IS SPRING BOOT

Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications. It is a Spring module that provides the RAD (Rapid Application Development) feature to the Spring Framework. It is used to create a stand-alone Spring-based application that you can just run because it needs minimal Spring configuration.

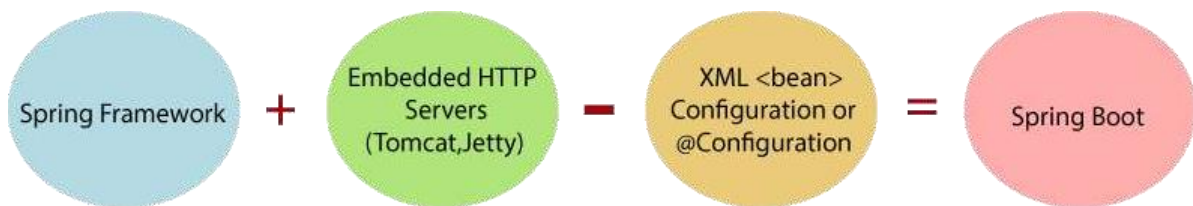


Figure 4.1: What is Spring Boot

In short, Spring Boot is the combination of Spring Framework and Embedded Servers. In Spring Boot, there is no requirement for XML configuration (deployment descriptor). It uses convention over configuration software design paradigm which means it decreases the effort of the developer. We can use Spring STS IDE or Spring Initializr to develop Spring Boot Java applications.

We should use Spring Boot Framework because:

- The dependency injection approach is used in Spring Boot.
- It contains powerful database transaction management capabilities.
- It simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts, etc.

- It reduces the cost and development time of the application.

Along with the Spring Boot Framework, many other Spring sister projects help to build applications addressing modern business needs. There are the following Spring sister projects are as follows:

- **Spring Data** – It simplifies data access from the relational and NoSQL databases.
- **Spring Batch** – It provides powerful batch processing.
- **Spring Security** – It is a security framework that provides robust security to applications.
- **Spring Social** – It supports integration with social networking like LinkedIn.
- **Spring Integration** – It is an implementation of Enterprise Integration Patterns. It facilitates integration with other enterprise applications using lightweight messaging and declarative adapters.

4.2 ADVANTAGES OF SPRING BOOT

- It creates stand-alone Spring applications that can be started using Java -jar.
- It tests web applications easily with the help of different Embedded HTTP servers such as Tomcat, Jetty, etc. We don't need to deploy WAR files.
- It provides opinionated 'starter' POMs to simplify our Maven configuration.
- It provides production-ready features such as metrics, health checks, and externalized configuration.
- There is no requirement for XML configuration.
- It offers a CLI tool for developing and testing the Spring Boot application.
- It offers a number of plug-ins.
- It also minimizes writing multiple boilerplate codes (the code that has to be included in many places with little or no alteration), XML configuration, and annotations.
- It increases productivity and reduces development time.

One of the major limitations of Spring Boot is that it can use dependencies that are not going to be used in the application. These dependencies increase the size of the application.

4.3 GOALS OF SPRING BOOT

The main goal of Spring Boot is to reduce development, unit test, and integration test time.

- Provides Opinionated Development approach
- Avoids defining more Annotation Configuration
- Avoids writing lots of import statements
- Avoids XML Configuration.

By providing or avoiding the above points, Spring Boot Framework reduces Development time, Developer Effort, and increases productivity.

4.4 FEATURES OF SPRING BOOT

Given below are the main features of Spring Boot:

- **Web Development** – It is a well-suited Spring module for web application development. We can easily create a self-contained HTTP application that uses embedded servers like Tomcat, Jetty, or Undertow. We can use the spring-boot-starter-web module to start and run the application quickly.
- **SpringApplication** – SpringApplication is a class that provides a convenient way to bootstrap a Spring application. It can be started from the main method. We can call the application just by calling a static run() method.
- **Application events and listeners** – Spring Boot uses events to handle a variety of tasks. It allows us to create factories file that is used to add listeners. We can refer it tousing the ApplicationListener key. Always create factories file in the META-INF folder like META-INF/spring.factories.
- **Admin features** – Spring Boot provides the facility to enable admin-related features for the application. It is used to access and manage applications remotely. We can enable it in the Spring Boot application by using spring.application.admin.enabled property.
- **Externalized Configuration** – Spring Boot allows us to externalize our configuration so that the same application can be worked in different environments. The application uses YAML files to externalize configuration.
- **Properties Files** – Spring Boot provides a rich set of Application Properties. So, we can use that in the properties file of our project. The properties file is used to set

properties like server-port =8082 and many others. It helps to organize application properties.

- **YAML Support** – It provides a convenient way of specifying the hierarchical configuration. It is a superset of JSON. The SpringApplication class automatically supports YAML. It is an alternative to the properties file.
- **Type-safe Configuration** – The strong type-safe configuration is provided to govern and validate the configuration of the application. Application configuration is always a crucial task which should be type-safe. We can also use annotation provided by this library.
- **Logging** – Spring Boot uses Common logging for all internal logging. Logging dependencies are managed by default. We should not change logging dependencies if no customization is needed.
- **Security** – Spring Boot applications are spring bases web applications. So, it is secure by default with basic authentication on all HTTP endpoints. A rich set of Endpoints is available to develop a secure Spring Boot application.

4.5 SPRING vs SPRING BOOT

- **Spring:** Spring Framework is Java's most popular application development framework. The main feature of the Spring Framework is dependency Injection or Inversion of Control (IoC). With the help of Spring Framework, we can develop a loosely coupled application. It is better to use if application type or characteristics are purely defined.
- **Spring Boot:** Spring Boot is a module of Spring Framework. It allows us to build a stand-alone application with minimal or zero configurations. It is better to use if we want to develop a simple Spring-based application or RESTful services.

The primary comparison between Spring and Spring Boot are discussed below:

Spring	Spring Boot
Spring Framework is a widely used Java EE framework for building applications.	Spring Boot Framework is widely used to develop REST APIs.
It aims to simplify Java EE development that makes developers more productive.	It aims to shorten the code length and provide the easiest way to develop Web

	Applications.
The primary feature of the Spring Framework is dependency injection.	The primary feature of Spring Boot is Autoconfiguration. It automatically configures the classes based on the requirement.
It helps to make things simpler by allowing us to develop loosely coupled applications.	It helps to create a stand-alone application with less configuration.
The developer writes a lot of code (boilerplate code) to do the minimal task.	It reduces boilerplate code.
To test the Spring project, we need to set up the sever explicitly.	Spring Boot offers embedded servers such as Jetty and Tomcat, etc.
It does not provide support for an in-memory database.	It offers several plugins for working with an embedded and in-memory database such as H2.
Developers manually define dependencies for the Spring project in pom.xml.	Spring Boot comes with the concept of starter in the pom.xml file that internally takes care of downloading the dependencies JARs based on Spring Boot requirements.

Table 4.1: Spring vs Spring Boot

4.6 SPRING BOOT vs SPRING MVC

- **Spring Boot:** Spring Boot makes it easy to quickly bootstrap and start developing a Spring-based application. It avoids a lot of boilerplate code. It hides a lot of complexity behind the scene so that the developer can quickly get started and develop Spring-based applications easily.
- **Spring MVC:** Spring MVC is a Web MVC Framework for building web applications. It contains a lot of configuration files for various capabilities. It is an HTTP oriented web application development framework.

Spring Boot and Spring MVC exist for different purposes. The primary comparison between Spring Boot and Spring MVC is discussed below:

Spring Boot	Spring MVC
Spring Boot is a module of Spring for packaging the Spring-based application with sensible defaults.	Spring MVC is a model view controller-based web framework under the Spring framework.
It provides default configurations to build a Spring-powered framework.	It provides ready to use features for building a web application.
There is no need to build configuration manually.	It requires manual build configurations.
There is no requirement for a deployment descriptor.	A Deployment descriptor is required.
It avoids boilerplate code and wraps dependencies together in a single unit.	It specifies each dependency separately.
It reduces development time and increases productivity.	It takes more time to achieve the same.

Table 4.2: Spring Boot vs Spring MVC

Chapter 5

Spring Boot Architecture

Spring Boot is a module of the Spring Framework. It is used to create stand-alone, production- grade Spring Based Applications with minimum effort. It is developed on top of the core Spring Framework. Spring Boot follows a layered architecture in which each layer communicates with the layer directly below or above (hierarchical structure) it.

Before understanding the Spring Boot Architecture, we must know the different layers and classes present in it. There are four layers in Spring Boot as follows:

- Presentation Layer
- Business Layer
- Persistence Layer
- Database Layer

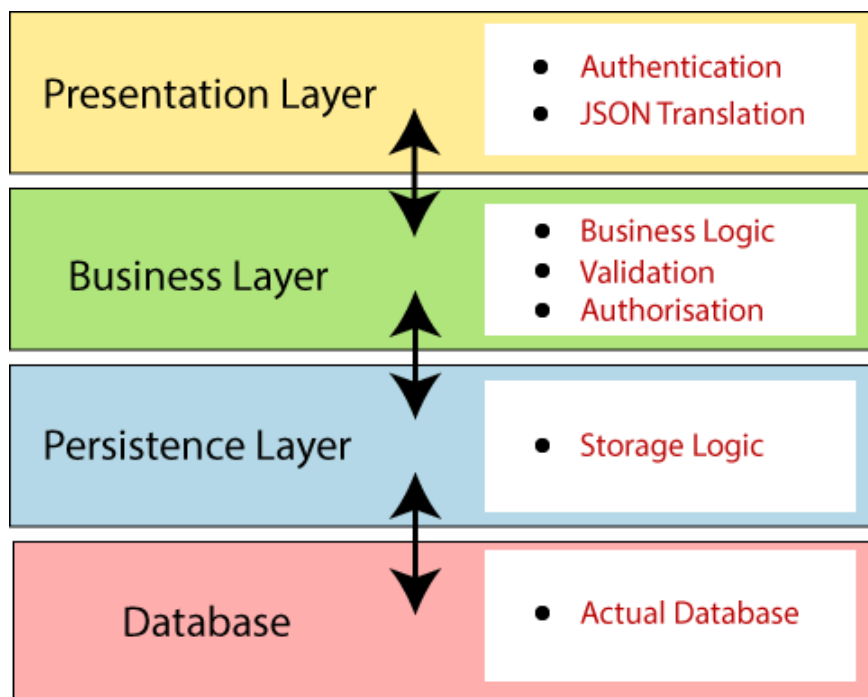


Figure 5.1: Spring Boot Architecture

- **Presentation Layer** – The presentation layer handles the HTTP requests, translates the JSON parameter to the object, and authenticates the request and transfers it to the business layer. In short, it consists of views i.e., the frontend part.

- **Business Layer** – The business layer handles all the business logic. It consists of service classes and uses services provided by data access layers. It also performs authorisation and validation.
- **Persistence Layer** – The persistence layer contains all the storage logic and translates business objects from and to database rows.
- **Database Layer** – In the database layer, CRUD (create, retrieve, update, delete) operations are performed.

5.1 SPRING BOOT FLOW ARCHITECTURE

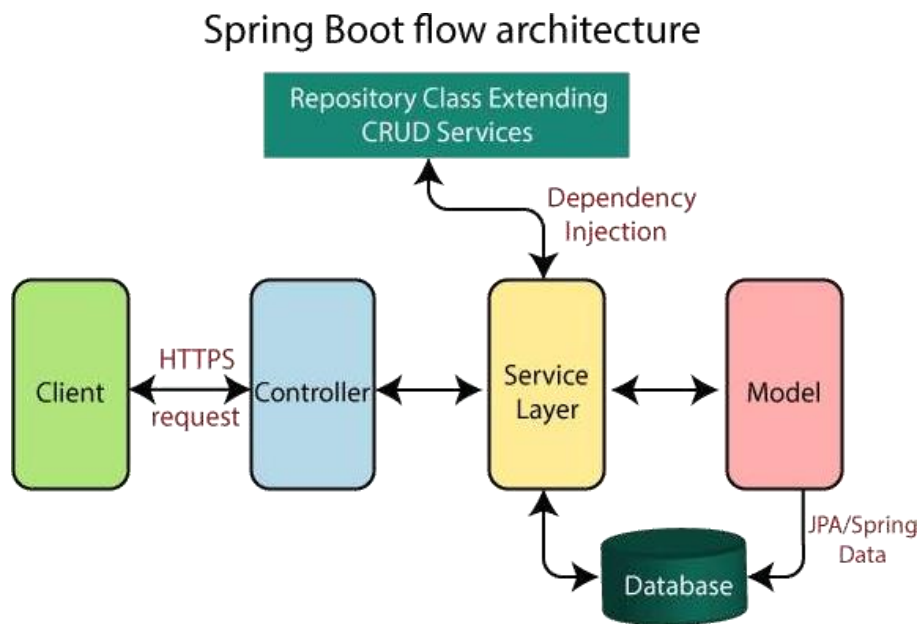


Figure 5.2: Spring Boot Flow Architecture

- Now we have validator classes, view classes, and utility classes.
- Spring Boot uses all the modules of Spring-like Spring MVC, Spring Data, etc. The architecture of Spring Boot is the same as the architecture of Spring MVC, except for one thing: there is no need for DAO and DAOImpl classes in Spring boot.
- Creates a data access layer and performs CRUD operation.
- The client makes the HTTP requests (PUT or GET).
- The request goes to the controller, and the controller maps that request and handles it. After that, it calls the service logic if required.
- In the service layer, all the business logic performs. It performs the logic on the data that is mapped to JPA with model classes.
- A JSP page is returned to the user if no error occurred.

Chapter 6

Spring Boot Project Components

There are a few project components required for a Spring Boot project. The required project components are as follows:

- Spring Boot Annotations
- Spring Boot Dependency Management
- Spring Boot Application Properties
- Spring Boot Starters
- Spring Boot Starter Parent
- Spring Boot Starter Web
- Spring Boot Starter Data JPA
- Spring Boot Starter Actuator
- Spring Boot Starter Test
- Spring Boot DevTools
- Multi-Module Project
- Spring Boot Packaging
- Spring Boot Auto-configuration

6.1 SPRING BOOT ANNOTATIONS

Spring Boot Annotations is a form of metadata that provides data about a program. In other words, annotations are used to provide supplemental information about a program. It is not a part of the application that we develop. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

6.1.1 CORE SPRING FRAMEWORK ANNOTATIONS

- **@Required** – It applies to the bean setter method. It indicates that the annotated bean must be populated at configuration time with the required property, else it throws an exception `BeanInitializationException`.
- **@Autowired** – Spring provides annotation-based auto-wiring by providing this annotation. It is used to autowire spring bean on setter methods, instance variable, and constructor. When we use `@Autowired` annotation, the spring container auto-wires the bean by matching the data type.

- **@Configuration** – It is a class-level annotation. The class annotated this annotation is used by Spring Containers as a source of bean definitions.
- **@ComponentScan** – It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.
- **@Bean** – It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.

6.1.2 SPRING FRAMEWORK STEREOTYPE ANNOTATIONS

- **@Component** – It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with @Component is found during the classpath. The Spring Framework pick it up and configures it in the application context as a Spring Bean.
- **@Controller** – The @Controller is a class-level annotation. It is a specialization of @Component. It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with @RequestMapping annotation.
- **@Service** – It is also used at the class level. It tells the Spring that the class contains the business logic.
- **@Repository** – It is a class-level annotation. The repository is a DAOs (Data Access Object) that access the database directly. The repository does all the operations related to the database.

6.1.3 SPRING BOOT ANNOTATIONS

- **@EnableAutoConfiguration** – It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. @SpringBootApplication.
- **@SpringBootApplication** – It is a combination of the following three annotations @EnableAutoConfiguration, @ComponentScan, and @Configuration.

6.1.4 SPRING MVC AND REST ANNOTATIONS

- **@RequestMapping** – It is used to map the web requests. It has many optional elements like consumes, header, method, name, params, path, produces, and value. We use it with the class as well as the method.
- **@GetMapping** – It maps the HTTP GET requests on the specific handler method. It is used to create a web service endpoint that fetches. It is used instead of using: `@RequestMapping(method = RequestMethod.GET)`
- **@PostMapping** – It maps the HTTP POST requests on the specific handler method. It is used to create a web service endpoint that creates. It is used instead of using: `@RequestMapping(method = RequestMethod.POST)`
- **@PutMapping** – It maps the HTTP PUT requests on the specific handler method. It is used to create a web service endpoint that creates or updates. It is used instead of using: `@RequestMapping(method = RequestMethod.PUT)`
- **@DeleteMapping** – It maps the HTTP DELETE requests on the specific handler method. It is used to create a web service endpoint that deletes a resource. It is used instead of using: `@RequestMapping(method = RequestMethod.DELETE)`
- **@PatchMapping** – It maps the HTTP PATCH requests on the specific handler method. It is used instead of using: `@RequestMapping(method = RequestMethod.PATCH)`
- **@RequestBody** – It is used to bind HTTP requests with an object in a method parameter. Internally it uses HTTP MessageConverters to convert the body of the request. When we annotate a method parameter with `@RequestBody`, the Spring framework binds the incoming HTTP request body to that parameter.
- **@ResponseBody** – It binds the method return value to the response body. It tells the Spring Boot Framework to serialize and return an object into JSON and XML format.
- **@PathVariable** – It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple `@PathVariable` in a method.
- **@RequestParam** – It is used to extract the query parameters from the URL. It is also known as a query parameter. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.
- **@RequestHeader** – It is used to get the details about the HTTP request headers. We use this annotation as a method parameter. The optional elements of the annotation are

name, required, value, and defaultValue. For each detail in the header, we should specify separate annotations. We can use it multiple time in a method

- **@RestController** – It can be considered as a combination of @Controller and @ResponseBody annotations. The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.
- **@RequestAttribute** – It binds a method parameter to the request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server side.

6.2 SPRING BOOT DEPENDENCY MANAGEMENT

Spring Boot manages dependencies and configuration automatically. Each release of Spring Boot provides a list of dependencies that it supports. The list of dependencies is available as a part of the Bills of Materials (spring-boot-dependencies) that can be used with Maven. So, we need not specify the version of the dependencies in our configuration. Spring Boot manages itself. Spring Boot upgrades all dependencies automatically in a consistent way when we update the Spring Boot version.

The following are the advantages of dependency management:

- It provides the centralization of dependency information by specifying the Spring Boot version in one place. It helps when we switch from one version to another.
- It avoids the mismatch of different versions of Spring Boot libraries.
- We only need to write a library name specifying the version. It is helpful in multi-module projects.

6.2.1 MAVEN DEPENDENCY MANAGEMENT SYSTEM

The Maven project inherits the following features from spring-boot-starter-parent:

- The default Java compiler version
- UTF-8 source encoding
- It inherits a Dependency Section from the spring-boot-dependency-pom. It manages the version of common dependencies. It ignores the <version> tag for that dependencies.

- Dependencies inherited from the spring-boot-dependencies POM
- Sensible resource filtering
- Sensible plugin configuration

The following spring-boot-starter-parent inherits automatically when we configure the project.

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.2.BUILD-SNAPSHOT</version>    <!-- lookup parent from repository -->
<relativePath/>
</parent>
```

Figure 6.1: spring-boot-starter-parent

To add another dependency with the same artifact that we have injected already, inject that dependency again inside the `<properties>` tag to override the previous one. We can also add the Maven plugin to our pom.xml file. It wraps the project into an executable jar file.

By using the `<scope>` tag, we can take advantage of dependency management even without the spring-boot-starter-parent dependency.

```
<dependencyManagement>
<dependencies>
<dependency><!-- Import dependency management from Spring Boot -->
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-dependencies</artifactId>
<version>2.2.2.RELEASE</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

Figure 6.2: Using the `<scope>` tag

The above dependency does not allow overriding. To achieve the overriding, we need to add an entry inside the <dependencyManagement> tag of our project before the spring-boot-dependencies entry.

6.3 SPRING BOOT APPLICATION PROPERTIES

Spring Boot Framework comes with a built-in mechanism for application configuration using a file called application.properties. It is located inside the src/main/resources folder.

Spring Boot provides various properties that can be configured in the application.properties file. The properties have default values. We can set a property(s) for the Spring Boot application. Spring Boot also allows us to define our property if required. The application.properties file allows us to run an application in a different environment. In short, we can use the application.properties file to:

- Configure the Spring Boot framework
- Define our application's custom configuration properties

```
#configuring application name
spring.application.name = demoApplication
#configuring port
server.port = 8081
```

Figure 6.3: Example of application.properties

In the above example, the application name and port have been configured. Port 8081 denotes that the application runs on port 8081.

6.3.1 SPRING BOOT PROPERTY CATEGORIES

There are sixteen categories of Spring Boot properties as follows:

- Core Properties
- Cache Properties
- Mail Properties
- JSON Properties
- Data Properties

- Transaction Properties
- Data Migration Properties
- Integration Properties
- Web Properties
- Templating Properties
- Server Properties
- Security Properties
- RSocket Properties
- Actuator Properties
- DevTools Properties
- Testing Properties

6.3.2 APPLICATION PROPERTIES TABLE

Property	Default Values	Description
Debug	false	It enables debug logs.
spring.application.name		It is used to set the application name.
spring.application.admin.enabled	false	It is used to enable admin features of the application.
spring.config.name	application	It is used to set the config file name.
spring.config.location		It is used to config the file name.
server.port	8080	Configures the HTTP server port
server.servlet.context-path		It configures the context path of the application.
logging.file.path		It configures the location of

		the log file.
spring.banner.charset	UTF-8	Banner file encoding.
spring.banner.location	classpath:banner.txt	It is used to set banner file location.
logging.file		It is used to set the log file name. For example, data.log.
spring.application.index		It is used to set the application index.
spring.application.name		It is used to set the application name.
spring.application.admin.enabled	false	It is used to enable admin features for the application.
spring.config.location		It is used to config the file locations.
spring.config.name	application	It is used to set the config file name.
spring.mail.default-encoding	UTF-8	It is used to set default MimeMessage encoding.
spring.mail.host		It is used to set the SMTP server host. For example, smtp.example.com.
spring.mail.password		It is used to set the login password of the SMTP server.

spring.mail.port		It is used to set the SMTP server port.
spring.mail.test-connection	false	It is used to test that the mail server is available onstartup.
spring.mail.username		It is used to set the login user of the SMTP server.
spring.main.sources		It is used to set sources for the application.
server.address		It is used to set the network address to which the server should bind.
server.connection-timeout		It is used to set the time in milliseconds that connectors will wait for another HTTP request before closing the connection.
server.context-path		It is used to set the context path of the application.
server.port	8080	It is used to set the HTTP port.
server.server-header		It is used for the Server response header (no header is sent if empty)
server.servlet-path	/	It is used to set the path of the main dispatcher servlet

server.ssl.enabled		It is used to enable SSL support.
spring.http.multipart.enabled	True	It is used to enable support of multi-part uploads.
spring.servlet.multipart.max-file-size	1MB	It is used to set max file size.
spring.mvc.async.request-timeout		It is used to set time in milliseconds.
spring.mvc.date-format		It is used to set date format. For example, dd/MM/yyyy.
spring.mvc.locale		It is used to set the locale for the application.
spring.social.facebook.app-id		It is used to set application's Facebook App ID.
spring.social.linkedin.app-id		It is used to set application's LinkedIn App ID.
spring.social.twitter.app-id		It is used to set application's Twitter App ID.
security.basic.authorize-mode	role	It is used to set security to authorize mode to apply.
security.basic.enabled	true	It is used to enable basic authentication.
Spring.test.database.replace	any	Type of existing DataSource to replace.
Spring.test.mockmvc.print	default	MVC Print option

spring.freemarker.content-type	text/html	Content-Type value
server.server-header		Value to use for the server response header.
spring.security.filter.dispatcher-type	async, error, request	Security filter chain dispatcher types.
spring.security.filter.order	-100	Security filter chain order.
spring.security.oauth2.client.registration.*		OAuth client registrations.
spring.security.oauth2.client.provider.*		OAuth provider details.

Table 6.1: Application Properties table

6.4 SPRING BOOT STARTERS

Spring Boot provides a number of starters that allow us to add jars in the classpath. Spring Boot built-in starters make development easier and rapid. Spring Boot Starters are the dependency descriptors. In the Spring Boot Framework, all the starters follow a similar naming pattern: spring-boot-starter-*, where * denotes a particular type of application. For example, if we want to use Spring and JPA for database access, we need to include the spring-boot-starter-data-jpa dependency in our pom.xml file of the project.

6.4.1 THIRD-PARTY STARTERS

We can also include third party starters in our project. But we do not use spring-boot-starter for including third party dependency. The spring-boot-starter is reserved for official Spring Boot artifacts. The third-party starter starts with the name of the project. For example, the third-party project name is abc, then the dependency name will be abc-spring-boot-starter.

The Spring Boot Framework provides the following application starters under the org.springframework.boot group.

Name	Description
spring-boot-starter-thymeleaf	It is used to build MVC web applications using Thymeleaf views.
spring-boot-starter-data-couchbase	It is used for the Couchbase document-oriented database and Spring Data Couchbase.
spring-boot-starter-artemis	It is used for JMS messaging using Apache Artemis.
spring-boot-starter-web-services	It is used for Spring Web Services.
spring-boot-starter-mail	It is used to support Java Mail and Spring Framework's email sending.
spring-boot-starter-data-redis	It is used for Redis key-value data store with Spring Data Redis and the Jedis client.
spring-boot-starter-web	It is used for building the web application, including RESTful applications using Spring MVC. It uses Tomcat as the default embedded container.
spring-boot-starter-data-gemfire	It is used to GemFire distributed data store and Spring Data GemFire.
spring-boot-starter-activemq	It is used in JMS messaging using Apache ActiveMQ.
spring-boot-starter-data-elasticsearch	It is used in Elasticsearch search and analytics engine and Spring Data Elasticsearch.
spring-boot-starter-integration	It is used for Spring Integration.
spring-boot-starter-test	It is used to test Spring Boot applications

	with libraries, including JUnit, Hamcrest, and Mockito.
spring-boot-starter-jdbc	It is used for JDBC with the Tomcat JDBC connection pool.
spring-boot-starter-mobile	It is used for building web applications using Spring Mobile.
spring-boot-starter-validation	It is used for Java Bean Validation with Hibernate Validator.
spring-boot-starter-hateoas	It is used to build a hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS.
spring-boot-starter-jersey	It is used to build RESTful web applications using JAX-RS and Jersey. An alternative to spring-boot-starter-web.
spring-boot-starter-data-neo4j	It is used for the Neo4j graph database and Spring Data Neo4j.
spring-boot-starter-data-ldap	It is used for Spring Data LDAP.
spring-boot-starter-websocket	It is used for building the WebSocket applications. It uses Spring Framework's WebSocket support.
spring-boot-starter-aop	It is used for aspect-oriented programming with Spring AOP and AspectJ.
spring-boot-starter-amqp	It is used for Spring AMQP and Rabbit MQ.
spring-boot-starter-data-cassandra	It is used for Cassandra distributed database and Spring Data Cassandra.

spring-boot-starter-social-facebook	It is used for Spring Social Facebook.
spring-boot-starter-jta-atomikos	It is used for JTA transactions using Atomikos.
spring-boot-starter-security	It is used for Spring Security.
spring-boot-starter-mustache	It is used for building MVC web applications using Mustache views.
spring-boot-starter-data-jpa	It is used for Spring Data JPA with Hibernate.
spring-boot-starter	It is used for core starter, including auto-configuration support, logging, andYAML.
spring-boot-starter-groovy-templates	It is used for building MVC web applications using Groovy Template views.
spring-boot-starter-freemarker	It is used for building MVC web applications using FreeMarker views.
spring-boot-starter-batch	It is used for Spring Batch.
spring-boot-starter-social-linkedin	It is used for Spring Social LinkedIn.
spring-boot-starter-cache	It is used for Spring Framework's caching support.
spring-boot-starter-data-solr	It is used for the Apache Solr search platform with Spring Data Solr.
spring-boot-starter-data-mongodb	It is used for MongoDB document-oriented database and Spring Data MongoDB.
spring-boot-starter-jooq	It is used for jOOQ to access SQL databases. An alternative to

	spring-boot-starter-data-jpa or spring-boot-starter-jdbc.
spring-boot-starter-jta-narayana	It is used for Spring Boot Narayana JTA Starter.
spring-boot-starter-cloud-connectors	It is used for Spring Cloud Connectors that simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku.
spring-boot-starter-jta-bitronix	It is used for JTA transactions using Bitronix.
spring-boot-starter-social-twitter	It is used for Spring Social Twitter.
spring-boot-starter-data-rest	It is used for exposing Spring Data repositories over REST using Spring Data REST.

Table 6.2: Third-Party Starters

6.4.2 SPRING BOOT PRODUCTION STARTERS

Name	Description
spring-boot-starter-actuator	It is used for Spring Boot's Actuator that provides production-ready features to help you monitor and manage your application.
spring-boot-starter-remote-shell	It is used for the CRaSH remote shell to monitor and manage your application over SSH. Deprecated since 1.5.

Table 6.3: Spring Boot Production Starters

6.4.3 SPRING BOOT TECHNICAL STARTERS

Name	Description
spring-boot-starter-undertow	It is used for Undertow as the embedded servlet container. An alternative to spring-boot-starter-tomcat.
spring-boot-starter-jetty	It is used for Jetty as the embedded servlet container. An alternative to spring-boot-starter-tomcat.
spring-boot-starter-logging	It is used for logging using Logback. Default logging starter.
spring-boot-starter-tomcat	It is used for Tomcat as the embedded servlet container. Default servlet container starter used by spring-boot-starter-web.
spring-boot-starter-log4j2	It is used for Log4j2 for logging. An alternative to spring-boot-starter-logging.

Table 6.4: Spring Boot Technical Starters

6.5 SPRING BOOT STARTER PARENT

The spring-boot-starter-parent is a project starter. It provides default configurations for our applications. It is used internally by all dependencies. All Spring Boot projects use spring-boot-starter-parent as a parent in pom.xml file. Parent Poms allow us to manage the following things for multiple child projects and modules:

- Configuration – It allows us to maintain consistency of Java Version and other related properties.
- Dependency Management: It controls the versions of dependencies to avoid conflict.
- Source encoding
- Default Java Version
- Resource filtering
- It also controls the default plugin configuration.

The `spring-boot-starter-parent` inherits dependency management from `spring-boot-dependencies`. We only need to specify the Spring Boot version number. If there is a requirement of the additional starter, we can safely omit the version number.

6.6 SPRING BOOT STARTER WEB

There are two important features of `spring-boot-starter-web`:

- It is compatible for web development
- Auto configuration

The `spring-boot-starter-web` dependency has to be added in the `pom.xml` file to make use of spring boot starter web. Starter of Spring web uses Spring MVC, REST and Tomcat as a default embedded server. The single `spring-boot-starter-web` dependency transitively pulls in all dependencies related to web development. It also reduces the build dependency count.

The `spring-boot-starter-web` transitively depends on the following:

- `org.springframework.boot:spring-boot-starter`
- `org.springframework.boot:spring-boot-starter-tomcat`
- `org.springframework.boot:spring-boot-starter-validation`
- `com.fasterxml.jackson.core:jackson-databind`
- `org.springframework:spring-web`
- `org.springframework:spring-webmvc`

By default, the `spring-boot-starter-web` contains the following tomcat server dependency (`spring-boot-starter-tomcat`). The `spring-boot-starter-web` auto-configures the following things that are required for the web development:

- Dispatcher Servlet
- Error Page
- Web JARs for managing the static dependencies
- Embedded servlet container

Each Spring Boot application includes an embedded server. Embedded server is embedded as a part of deployable application. The advantage of embedded server is, we do not require pre-installed server in the environment. With Spring Boot, default embedded server is Tomcat. Spring Boot also supports another two embedded servers:

- Jetty Server
- Undertow Server

For servlet stack applications, the `spring-boot-starter-web` includes Tomcat by including `spring-boot-starter-tomcat`. The `spring-boot-starter-jetty` or `spring-boot-starter-undertow` instead can also be used. For reactive stack applications, the `spring-boot-starter-webflux` includes Reactor Netty by including `spring-boot-starter-reactor-netty`, but we can use `spring-boot-starter-tomcat`, `spring-boot-starter-jetty`, or `spring-boot-starter-undertow` instead.

6.7 SPRING BOOT STARTER DATA JPA

Spring Data is a high-level Spring Source project. Its purpose is to unify and easy access to the different kinds of persistence stores, both relational database systems, and NoSQL data stores. When we implement a new application, we should focus on the business logic instead of technical complexity and boilerplate code. That's why the Java Persistent API (JPA) specification and Spring Data JPA are extremely popular.

Spring Data JPA adds a layer on the top of JPA. It means, Spring Data JPA uses all features defined by JPA specification, especially the entity, association mappings, and JPA's query capabilities. Spring Data JPA adds its own features such as the no-code implementation of the repository pattern and the creation of database queries from the method name.

Spring Data JPA handles most of the complexity of JDBC-based database access and ORM (Object Relational Mapping). It reduces the boilerplate code required by JPA. It makes the implementation of your persistence layer easier and faster. Spring Data JPA aims to improve the implementation of data access layers by reducing the effort to the amount that is needed.

6.7.1 SPRING DATA JPA FEATURES

There are three main features of Spring Data JPA are as follows:

- No-code repository – It is the most popular persistence-related pattern. It enables us to implement our business code on a higher abstraction level.
- Reduced boilerplate code – It provides the default implementation for each method by its repository interfaces. It means that there is no longer need to implement read and write operations.
- Generated Queries – Another feature of Spring Data JPA is the generation of database

queries based on the method name. If the query is not too complex, we need to define a method on our repository interface with the name that starts with `findBy`. After defining the method, Spring parses the method name and creates a query for it. For example:

```
public interface EmployeeRepository extends CrudRepository<Employee, Long>
{
    Employee findByName(String name);
}
```

In the above example, we extend the `CrudRepository` that uses two generics: `Employee` and `Long`. The `Employee` is the entity that is to be managed, and `Long` is the data type of primary key. Spring internally generates a JPQL (Java Persistence Query Language) query based on the method name. The query is derived from the method signature. It sets the bind parameter value, execute the query, and returns the result.

There are some other features as follows:

- It can integrate custom repository code.
- It is a powerful repository and custom object-mapping abstraction.
- It supports transparent auditing.
- It implements a domain base class that provides basic properties.
- It supports several modules such as Spring Data JPA, Spring Data MongoDB, Spring Data REST, Spring Data Cassandra, etc.

6.7.2 SPRING DATA REPOSITORY

Spring Data JPA provides three repositories as follows:

- `CrudRepository` – It offers standard create, read, update, and delete. It contains methods like `findOne()`, `findAll()`, `save()`, `delete()`, etc.
- `PagingAndSortingRepository` – It extends the `CrudRepository` and adds the `findAll` methods. It allows us to sort and retrieve the data in a paginated way.
- `JpaRepository` – It is a JPA specific repository. It is defined in Spring Data Jpa. It extends the both repository `CrudRepository` and `PagingAndSortingRepository`. It adds the JPA-specific methods, like `flush()` to trigger a flush on the persistence context.

6.7.3 SPRING BOOT STARTER DATA JPA

Spring Boot provides spring-boot-starter-data-jpa dependency to connect Spring application with relational database efficiently. The spring-boot-starter-data-jpa internally uses the spring-boot-jpa dependency (since Spring Boot version 1.5.3). The databases are designed with tables/relations. Earlier approaches (JDBC) involved writing SQL queries. In the JPA, we will store the data from objects into table and vice-versa. However, JPA evolved as a result of a different thought process. Before JPA, ORM was the term more commonly used to refer to these frameworks. It is the reason Hibernate is called the ORM framework.

JPA allows us to map application classes to table in the database.

- Entity Manager: Once we define the mapping, it handles all the interactions with the database.
- JPQL (Java Persistence Query Language): It provides a way to write queries to execute searches against entities. It is different from the SQL queries. JPQL queries already understand the mapping that is defined between entities. We can add additional conditions if required.
- Criteria API: It defines a Java-based API to execute searches against the database.

6.7.4 HIBERNATE vs JPA

Hibernate is the implementation of JPA. It is the most popular ORM framework, while JPA is an API that defines the specification. Hibernate understands the mapping that we add between objects and tables. It ensures that data is retrieved/ stored from the database based on the mapping. It also provides additional features on the top of the JPA.

6.8 SPRING BOOT STARTER ACTUATOR

Spring Boot Actuator is a sub-project of the Spring Boot Framework. It includes a number of additional features that help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live). We can use HTTP and JMX endpoints to manage and monitor the Spring Boot application. If we want to get production-ready features in an application, we should use the Spring Boot actuator. The actuator can be enabled by injecting the dependency spring-boot-starter-actuator in the pom.xml file.

6.8.1 SPRING BOOT ACTUATOR FEATURES

There are three main features of Spring Boot Actuator:

- Endpoints
- Metrics
- Audit

Endpoint – The actuator endpoints allows us to monitor and interact with the application. Spring Boot provides a number of built-in endpoints. We can also create our own endpoint. We can enable and disable each endpoint individually. Most of the application choose HTTP, where the Id of the endpoint, along with the prefix of /actuator, is mapped to a URL. For example, the /health endpoint provides the basic health information of an application. The actuator, by default, mapped it to /actuator/health.

Metrics – Spring Boot Actuator provides dimensional metrics by integrating with the micrometer. The micrometer is integrated into Spring Boot. It is the instrumentation library powering the delivery of application metrics from Spring. It provides vendor-neutral interfaces for timers, gauges, counters, distribution summaries, and long task timers with a dimensional data model.

Audit – Spring Boot provides a flexible audit framework that publishes events to an AuditEventRepository. It automatically publishes the authentication events if spring-security is in execution.

6.8.2 SPRING BOOT ACTUATOR ENDPOINTS

The actuator endpoints allow us to monitor and interact with our Spring Boot application. Spring Boot includes number of built-in endpoints and we can also add custom endpoints in Spring Boot application. The following table describes the widely used endpoints:

Id	Usage	Default
actuator	It provides a hypermedia-based discovery page for the other endpoints. It requires Spring HATEOAS to be on the classpath.	True
auditevents	It exposes audit events information for the current	True

	application.	
autoconfig	It is used to display an auto-configuration report showing all auto-configuration candidates and the reason why they 'were' or 'were not' applied.	True
beans	It is used to display a complete list of all the Spring beans in your application.	True
configprops	It is used to display a collated list of all @ConfigurationProperties.	True
dump	It is used to perform a thread dump.	True
env	It is used to expose properties from Spring's ConfigurableEnvironment.	True
flyway	It is used to show any Flyway database migrations that have been applied.	True
health	It is used to show application health information.	False
info	It is used to display arbitrary application info.	False
loggers	It is used to show and modify the configuration of loggers in the application.	True
liquibase	It is used to show any Liquibase database migrations that have been applied.	True
metrics	It is used to show metrics information for the current application.	True
mappings	It is used to display a collated list of all @RequestMapping paths.	True
shutdown	It is used to allow the application to be gracefully shutdown.	True

trace	It is used to display trace information.	True
-------	--	------

Table 6.5: Spring Boot Actuator

Endpoints For Spring MVC, the following additional endpoints are used:

Id	Usage	Default
docs	It is used to display documentation, including example requests and responses for the Actuator's endpoints.	False
heapdump	It is used to return a GZip compressed hprof heap dump file.	True
jolokia	It is used to expose JMX beans over HTTP (when Jolokia is on the classpath).	True
logfile	It is used to return the contents of the logfile.	True
prometheus	It is used to expose metrics in a format that can be scraped by a prometheus server. It requires a dependency on micrometer-registry-prometheus.	

Table 6.6: Spring MVC Actuator Endpoints

6.8.3 SPRING BOOT ACTUATOR PROPERTIES

Spring Boot enables security for all actuator endpoints. It uses form-based authentication that provides user Id as the user and a randomly generated password. We can also access actuator-restricted endpoints by customizing basic auth security to the endpoints. We need to override this configuration by management.security.roles property.

6.9 SPRING BOOT STARTER TEST

The spring-boot-starter-test is the primary dependency for the test. It contains the majority of elements required for our tests. There are several different types of tests that we can write to help test and automate the health of an application. Before starting any testing, we need to

integrate the testing framework.

With Spring Boot, we need to add starter to our project, for testing we only need to add the spring-boot-starter-test dependency. It pulls all the dependencies related to test. After adding it, we can build up a simple unit test. We can either create the Spring Boot project through IDE or generate it using Spring Initializr. When we create a simple Spring Boot application, by default, it contains the test dependency in the pom.xml file and ApplicationNameTest.java file under in the folder src/test/java.

6.10 SPRING BOOT DEVTOOLS

Spring Boot 1.3 provides another module called Spring Boot DevTools. DevTools stands for Developer Tool. The aim of the module is to try and improve the development time while working with the Spring Boot application. Spring Boot DevTools pick up the changes and restart the application. We can implement the DevTools in our project by adding the following dependency in the pom.xml file (spring-boot-devtools).

6.10.1 SPRING BOOT DEVTOOLS FEATURES

Spring Boot DevTools provides the following features:

- Property Defaults
- Automatic Restart
- LiveReload
- Remote Debug Tunneling
- Remote Update and Restart

Property Defaults – Spring Boot provides templating technology Thymeleaf that contains the property spring.thymeleaf.cache. It disables the caching and allows us to update pages without the need of restarting the application. But setting up these properties during the development always creates some problems. When we use the spring-boot-devtools module, we are not required to set properties. During the development caching for Thymeleaf, Freemarker, Groovy Templates are automatically disabled.

Automatic Restart – Auto-restart means reloading of Java classes and configure it at the server-side. After the server-side changes, it deployed dynamically, server restarts happen, and load the modified code. It is mostly used in microservice-based applications. Spring Boot uses two types of ClassLoaders:

- The classes that do not change (third-Jars) are loaded in the base ClassLoader.
- The classes that we are actively developing are loaded in the restart ClassLoader.

When the application restarts, the restart ClassLoader is thrown away, and a new one is populated. Therefore, the base ClassLoader is always available and populated. We can disable the auto-restart of a server by using the property `spring.devtools.restart.enabled` to false.

The following points has to be kept in mind:

- The DevTools always monitors the classpath resources.
- There is only a way to trigger a restart is to update the classpath.
- DevTools required a separate application classloader to work properly. By default, Maven fork the application process.
- Auto-restart works well with LiveReload.
- DevTools depends on the application context's shutdown hook to close it during the restart.

LiveReload – The Spring Boot DevTools module includes an embedded server called LiveReload. It allows the application to automatically trigger a browser refresh whenever we make changes in the resources. It is also known as auto-refresh.

We can disable the LiveReload by setting the property `spring.devtools.livereload.enabled` to false. It provides browser extensions for Chrome, Firefox, and Safari. By default, LiveReload is enabled. The LiveReload works on the following path:

- `/META-INF/maven`
- `/META-INF/resources`
- `/resources`
- `/static`
- `/public`
- `/templates`

We can also disable auto-reload in browser by excluding the above paths as follows:

```
spring.devtools.restart.exclude=public/**, static/**, templates/**
```

The following points has to be kept in mind:

- We can run one LiveReload server at a time.
- Before starting the application, ensure that no other LiveReload server is running.
- If we start multiple applications from IDE, it supports only the first LiveReload.

Remote Debug Tunneling – Spring Boot can tunnel JDWP (Java Debug Wire Protocol) over HTTP directly to the application. It can even work application deployment to Internet Cloud providers that only expose port 80 and 443.

Remote Update and Restart – There is another trick that DevTools offers is: it supports remote application updates and restarts. It monitors local classpath for file changes and pushes them to a remote server, which is then restarted. We can also use this feature in combination with LiveReload.

6.11 MULTI-MODULE PROJECT

A Spring Boot project that contains nested maven projects is called the multi-module project. In the multi-module project, the parent project works as a container for base maven configurations. In other words, a multi-module project is built from a parent pom that manages a group of submodules. Or A multi-module project is defined by a parent POM referencing one or more submodules. The parent maven project must contain the packaging type pom that makes the project as an aggregator. The pom.xml file of the parent project consists the list of all modules, common dependencies, and properties that are inherited by the child projects. The parent pom is located in the project's root directory. The child modules are actual Spring Boot projects that inherit the maven properties from the parent project. When we run the multi- module project, all the modules are deployed together in an embedded Tomcat Server. We can deploy an individual module, also.

The parent POM defines the Group ID, Artifact ID, version, and packaging. In the multi-module project, the parent POM defines the packaging pom. Splitting the project into multiple modules is useful and easy to maintain. We can also easily edit or remove modules in the project without affecting the other modules. It is useful when we required to deploy modules individually. We only need to specify all the dependencies in the parent pom. All the other modules share the same pom, so we need not to specify the same dependency in each module separately. It makes the code easier to keep in order with a big project.

6.12 SPRING BOOT PACKAGING

In the J2EE application, modules are packed as JAR, WAR, and EAR. It is the compressed file formats that is used in the J2EE. J2EE defines three types of archives:

- WAR
- JAR
- EAR

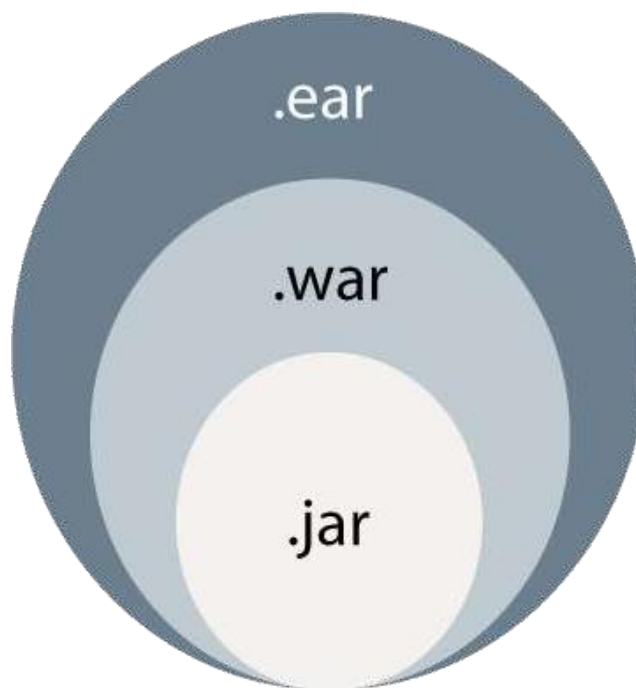


Figure 6.4: Spring Boot Packaging

WAR – WAR stands for Web Archive. WAR file represents the web application. Web module contains servlet classes, JSP files, HTML files, JavaScripts, etc. are packaged as a JAR file with .war extension. It contains a special directory called WEB-INF.

WAR is a module that loads into a web container of the Java Application Server. The Java Application Server has two containers: Web Container and EJB Container. The Web Container hosts the web applications based on Servlet API and JSP. The web container requires the web module to be packaged as a WAR file. It is a WAR file special JAR file that contains a web.xml file in the WEB-INF folder. An EJB Container hosts Enterprise Java beans based on EJB API. It requires EJB modules to be packaged as a JAR file. It contains an ejb-jar.xml file in the META-INF folder. The advantage of the WAR file is that it can be

deployed easily on the client machine in a Web server environment. To execute a WAR file, a Web server or Web container is required. For example, Tomcat, Weblogic, and Websphere.

JAR – JAR stands for Java Archive. An EJB (Enterprise Java Beans) module that contains bean files (class files), a manifest, and EJB deployment descriptor (XML file) are packaged as JAR files with the extension .jar. It is used by software developers to distribute Java classes and various metadata. In other words, A file that encapsulates one or more Java classes, a manifest, and descriptor is called JAR file. It is the lowest level of the archive. It is used in J2EE for packaging EJB and client-side Java Applications. It makes the deployment easy.

EAR – EAR stands for Enterprise Archive. EAR file represents the enterprise application. The above two files are packaged as a JAR file with the .ear extension. It is deployed into the Application Server. It can contain multiple EJB modules (JAR) and Web modules (WAR). It is a special JAR that contains an application.xml file in the META-INF folder.

6.13 SPRING BOOT AUTO-CONFIGURATION

Spring Boot auto-configuration automatically configures the Spring application based on the jar dependencies that we have added. For example, if the H2 database Jar is present in the classpath and we have not configured any beans related to the database manually, the Spring Boot's auto-configuration feature automatically configures it in the project. We can enable the auto-configuration feature by using the annotation `@EnableAutoConfiguration`. But this annotation does not use because it is wrapped inside the `@SpringBootApplication` annotation. The annotation `@SpringBootApplication` is the combination of three annotations: `@ComponentScan`, `@EnableAutoConfiguration`, and `@Configuration`. However, we use `@SpringBootApplication` annotation instead of using `@EnableAutoConfiguration`.

`@SpringBootApplication=@ComponentScan+@EnableAutoConfiguration+@Configuration`

When we add the spring-boot-starter-web dependency in the project, Spring Boot auto-configuration looks for the Spring MVC is on the classpath. It auto-configures `dispatcherServlet`, a default error page, and web jars. Similarly, when we add the spring-boot-starter-data-jpa dependency, we see that Spring Boot Auto-configuration, auto-configures a `datasource` and an `Entity Manager`. All auto-configuration logic is implemented in `spring-boot- autoconfigure.jar`.

6.13.1 DISABLE AUTO-CONFIGURATION CLASSES

We can also disable the specific auto-configuration classes, if we do not want to be applied. We use the `exclude` attribute of the annotation `@EnableAutoConfiguration` to disable the auto-configuration classes. We can use the attribute `excludeName` of the annotation `@EnableAutoConfiguration` and specify the qualified name of the class, if the class is not on the class path. We can exclude any number of auto-configuration classes by using the property `spring.autoconfigure.exclude`.

6.13.2 NEED OF AUTO-CONFIGURATION

Spring-based application requires a lot of configuration. When we use Spring MVC, we need to configure dispatcher servlet, view resolver, web jars among other things.

```
<servlet>
<servlet-name>dispatcher</servlet-name>
<servlet-class>
org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<init-param>
<param-name>contextConfigLocation</param-name>
<param-value>/WEB-INF/todo-servlet.xml</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>dispatcher</servlet-name>
<url-pattern>/</url-pattern>
</servlet-mapping>
```

Figure 6.5: Configuring dispatcher servlet

Similarly, when we use Hibernate/ JPA, we need to configure datasource, a transaction manager, an entity manager factory among a host of other things.

```

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
destroy-method="close">
<property name="driverClass" value="${db.driver}" />
<property name="jdbcUrl" value="${db.url}" />
<property name="user" value="${db.username}" />
<property name="password" value="${db.password}" />
</bean>
<jdbc:initialize-database data-source="dataSource">
<jdbc:script location="classpath:config/schema.sql" />
<jdbc:script location="classpath:config/data.sql" />
</jdbc:initialize-database>

```

Figure 6.6: Configuring datasource

```

<bean
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
id="entityManagerFactory">
<property name="persistenceUnitName" value="hsq_lpu" />
<property name="dataSource" ref="dataSource" />
</bean>

```

Figure 6.7: Configuring entity manager factory

```

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
<property name="entityManagerFactory" ref="entityManagerFactory" />
<property name="dataSource" ref="dataSource" />
</bean>
<tx:annotation-driven transaction-manager="transactionManager"/>

```

Figure 6.8: Configuring transaction manager

Chapter 7

Introduction to RESTful Web Services

REST stands for REpresentational State Transfer. It is developed by Roy Thomas Fielding, who also developed HTTP. The main goal of RESTful web services is to make web services more effective. RESTful web services try to define services using the different concepts that are already present in HTTP. REST is an architectural approach, not a protocol. It does not define the standard message exchange format. We can build REST services with both XML and JSON. JSON is more popular format with REST. The key abstraction is a resource in REST. A resource can be anything. It can be accessed through a Uniform Resource Identifier (URI).

For example, The resource has representations like XML, HTML, and JSON. The current state capture by representational resource. When we request a resource, we provide the representation of the resource. The important methods of HTTP are:

- GET: It reads a resource.
- PUT: It updates an existing resource.
- POST: It creates a new resource.
- DELETE: It deletes the resource.

For example, if we want to perform the following actions in the social media application, we get the corresponding results.

- POST /users: It creates a user.
- GET /users/{id}: It retrieves the detail of a user.
- GET /users: It retrieves the detail of all users.
- DELETE /users: It deletes all users.
- DELETE /users/{id}: It deletes a user.
- GET /users/{id}/posts/post_id: It retrieve the detail of a specific post.
- POST / users/{id}/ posts: It creates a post of the user.

HTTP also defines the following standard status code:

- 404: RESOURCE NOT FOUND
- 200: SUCCESS

- 201: CREATED
- 401: UNAUTHORIZED
- 500: SERVER ERROR

7.1 RESTFUL SERVICE CONSTRAINTS

The following are the restful service constraints:

- There must be a service producer and service consumer.
- The service is stateless.
- The service result must be cacheable.
- The interface is uniform and exposing resources.
- The service should assume a layered architecture.

7.2 ADVANTAGES OF RESTFUL WEB SERVICES

The following are the advantages of restful web services:

- RESTful web services are platform-independent.
- It can be written in any programming language and can be executed on any platform.
- It provides different data format like JSON, text, HTML, and XML.
- It is fast in comparison to SOAP because there is no strict specification like SOAP.
- These are reusable.
- They are language neutral.

7.3 RESTFUL WEB SERVICES BEST PRACTICES

Given below are some of the best practices to be followed while developing RESTful web services:

- The first and last best practice is the Consumer First. It means, we need to always think about our consumers. Before naming the resources, think from the perspective of the customers, what do they think about those resources? Will they be able to understand these resources?
- We must have excellent documentation for our API. Swagger is one of the most popular documentation standard for RESTful API. Make sure that the consumer understands the documentation that we have produced.
- The next best practice is to make the best use of HTTP. RESTful web services are

based on HTTP. Make the best use of the request methods. Use the right request method (GET, POST, PUT, and DELETE) appropriate for our specific action and ensure that we are sending a proper response status back. For example, when a RESOURCE NOT FOUND, don't send the SERVER ERROR. When a resource is CREATED, don't send SUCCESS, send CREATED back.

- Ensure that there is no secure information in the URI. Think about what you are putting in your URI. Ensure that there is nothing secure that is going in the URIs.
- Always use plurals. In the previous examples, we have used /users instead of using /user. Similarly, for accessing a resource, we have used /users/1 not /user/1. It is more readable than using the singular.
- When we think about resources, always use nouns for resources. But it is not always possible. There are always exception scenarios. For all these exceptions scenarios, define a consistent approach if we are searching through user use /user/search. For example, if we put a star on the gists the Github sends the request to the resource of the gists (/gists/{id}) and sends a put request with the star in the URI.

Chapter 8

Conclusion & Future Work

Companies are attempting to capitalize on the greater agility and scalability that microservices provide. According to Market Forecast, the worldwide cloud microservices market will increase by 22.5%, while the US market will rise by 27.4%. The future of microservices will emerge in a variety of interesting and challenging ways.

By integrating microservices and an event-driven architecture, developers may create distributed, scalable, fault-tolerant, and extendable systems that ingest and process huge volumes of real-time event information. Microservices enable development teams to add new features without changing or rewriting substantial portions of current code on the fly. Monitoring microservices helps you to validate the architecture and performance of your service while also detecting possible troubleshooting difficulties. Companies understand the benefits of microservice design. The use of hybrid clouds by various end-users and sectors is a significant factor impacting the growth of the microservices industry.

Serverless architectures brought the central cloud and microservice patterns full circle. A measured and organic approach will aid in the organization of procedures required to manage an Information Technology (IT) Cloud with microservices without losing control and reverting to the monolithic past.

As a conclusion, I write with hope and optimism. We have seen enough of the microservice concept to feel it is a viable option. We do not really know where we will end up, however one of the challenges of application development is that we really can only make decisions based on information we have right now.

REFERENCES

1. M. Klymash, I. Tchaikovskiy, O. Hordiichuk-Bublivska and Y. Pyrih, "Research of Microservices Features in Information Systems Using Spring Boot," 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T), 2020, pp. 507-510, doi: 10.1109/PICST51311.2020.9467911.
2. O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), 2018, pp. 000149-000154, doi: 10.1109/CINTI.2018.8928192.
3. A. Akbulut and H. G. Perros, "Software Versioning with Microservices through the API Gateway Design Pattern," 2019 9th International Conference on Advanced Computer Information Technologies (ACIT), 2019, pp. 289-292, doi: 10.1109/ACITT.2019.8779952.
4. M. A. Jamil, M. Arif, N. S. A. Abubakar and A. Ahmad, "Software Testing Techniques: A Literature Review," 2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M), 2016, pp. 177-182, doi: 10.1109/ICT4M.2016.045.
5. P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," in IEEE Software, vol. 35, no. 3, pp. 24-35, May/June 2018, doi: 10.1109/MS.2018.2141039.
6. K. Guntupally, R. Devarakonda and K. Kehoe, "Spring Boot based REST API to Improve Data Quality Report Generation for Big Scientific Data: ARM Data Center Example," 2018 IEEE International Conference on Big Data (Big Data), 2018, pp. 5328-5329, doi: 10.1109/BigData.2018.8621924.
7. L. xuchen and L. chaoyu, "Design and Implementation of a Spring Boot-Based Data Collection System," 2020 12th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC), 2020, pp. 236-239, doi: 10.1109/IHMSC49165.2020.00059.
8. Z. Wang, F. Tang and Z. L. Yu, "Design and Implementation of a Health Status

Reporting System Based on Spring Boot," 2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE), 2020, pp. 453-457, doi: 10.1109/ICAICE51518.2020.00095.

9. O. Al-Debagy and P. Martinek, "Extracting Microservices' Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach," 2020 IEEE 15th International Conference of System of Systems Engineering (SoSE), 2020, pp. 289-294, doi: 10.1109/SoSE50414.2020.9130466.
10. R. Yilmaz and F. Buzluca, "A Fuzzy Quality Model to Measure the Maintainability of Microservice Architectures," 2021 2nd International Informatics and Software Engineering Conference (IISEC), 2021, pp. 1-6, doi: 10.1109/IISEC54230.2021.9672417.