Software Architecture

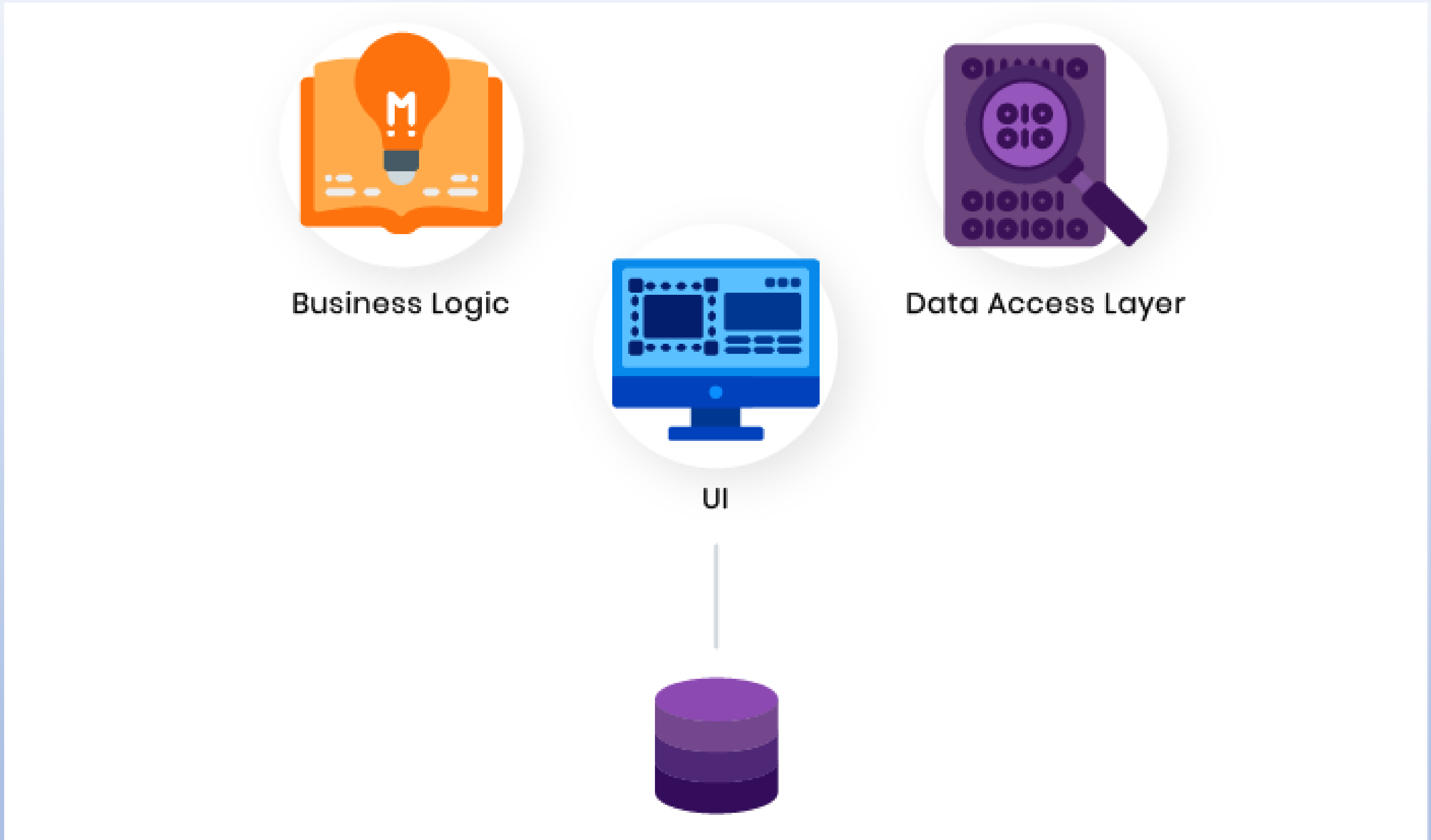
@Amit Kumar

Types of software architectures

SOA vs Monolith

Monolithic Architecture

Monolith is an ancient word referring to a huge single block of stone. In software engineering, a monolithic pattern refers to a single indivisible unit. The concept of monolithic software lies in different components of an application being combined into a single program on a single platform. Usually, a monolithic app consists of a database, client-side user interface, and server-side application.



Monolithic Architecture

Pros of Monolithic

- **Simpler development and deployment**
- There are lots of tools you can integrate to facilitate development. In addition, all actions are performed with one directory, which provides for easier deployment.
- **Fewer cross-cutting concerns**
- Most applications are reliant on a great deal of cross-cutting concerns, such as audit trails, logging, rate limiting, etc. Monolithic apps incorporate these concerns much easier due to their single code base.
- **Better performance**
- If built properly, monolithic apps are usually more performant than microservice-based apps. An app with a microservices architecture might need to make 40 API calls to 40 different microservices to load each screen, for example, which obviously results in slower performance. Monolithic apps, in turn, allow faster communication between software components due to shared code and memory.

Cons of Monolithic

- **Codebase gets cumbersome over time**
- In the course of time, most products develop and increase in scope, and their structure becomes blurred. The code base starts to look really massive and becomes difficult to understand and modify, especially for new developers. It also gets harder to find side effects and dependencies.
- **Difficult to adopt new technologies**
- If there’s a need to add some new technology to your app, developers may face barriers to adoption. Adding new technology means rewriting the whole application, which is costly and time-consuming.
- **Limited agility**
- In monolithic apps, every small update requires a full redeployment. Thus, all developers have to wait until it’s done. When several teams are working on the same project, agility can be reduced greatly.

The bottom line

- The monolithic model isn’t outdated, and it still works great in some cases. Some giant companies like **Etsy** stay monolithic despite today’s popularity of microservices. Monolithic software architecture can be beneficial if your team is at the founding stage, you’re building an unproven product, and you have no experience with microservices. Monolithic is perfect for start-ups that need to get a product up and running as soon as possible. However, certain issues mentioned above come with the monolithic package.

SOA Architecture

A service-oriented architecture (SOA) is a software architecture style that refers to an application composed of discrete and loosely coupled software agents that perform a required function. SOA has two main roles: a service provider and a service consumer. Both of these roles can be played by a software agent. The concept of SOA lies in the following: an application can be designed and built in a way that its modules are integrated seamlessly and can be easily reused.

Pros of SOA

Reusability of services

Due to the self-contained and loosely coupled nature of functional components in service-oriented applications, these components can be reused in multiple applications without influencing other services.

Better maintainability

Since each software service is an independent unit, it’s easy to update and maintain it without hurting other services. For example, large enterprise apps can be managed easier when broken into services.

Higher reliability

Services are easier to debug and test than are huge chunks of code like in the monolithic approach. This, in turn, makes SOA-based products more reliable.

Parallel development

As a service-oriented architecture consists of layers, it advocates parallelism in the development process. Independent services can be developed in parallel and completed at the same time. Below, you can see how SOA app development is executed by several developers in parallel.

Cons of SOA

Complex management

The main drawback of a service-oriented architecture is its complexity. Each service has to ensure that messages are delivered in time. The number of these messages can be over a million at a time, making it a big challenge to manage all services.

High investment costs

SOA development requires a great upfront investment of human resources, technology, and development.

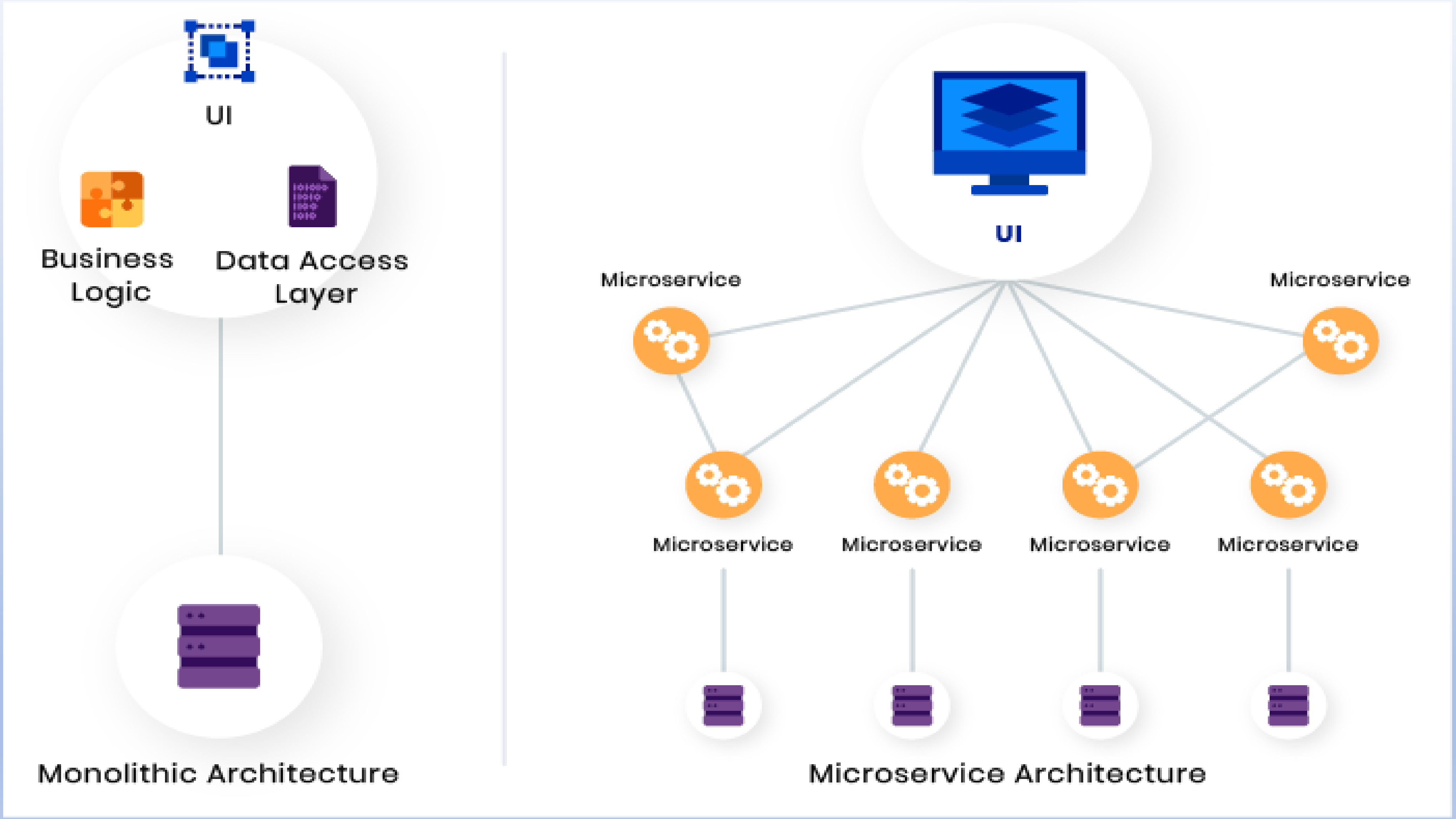
Extra overload

In SOA, all inputs are validated before one service interacts with another service. When using multiple services, this increases response time and decreases overall performance.

The bottom line

The SOA approach is best suited for complex enterprise systems such as those for banks. A banking system is extremely hard to break into microservices. But a monolithic approach also isn’t good for a banking system as one part could hurt the whole app. The best solution is to use the SOA approach and organize complex apps into isolated independent services.

Why Microservices?

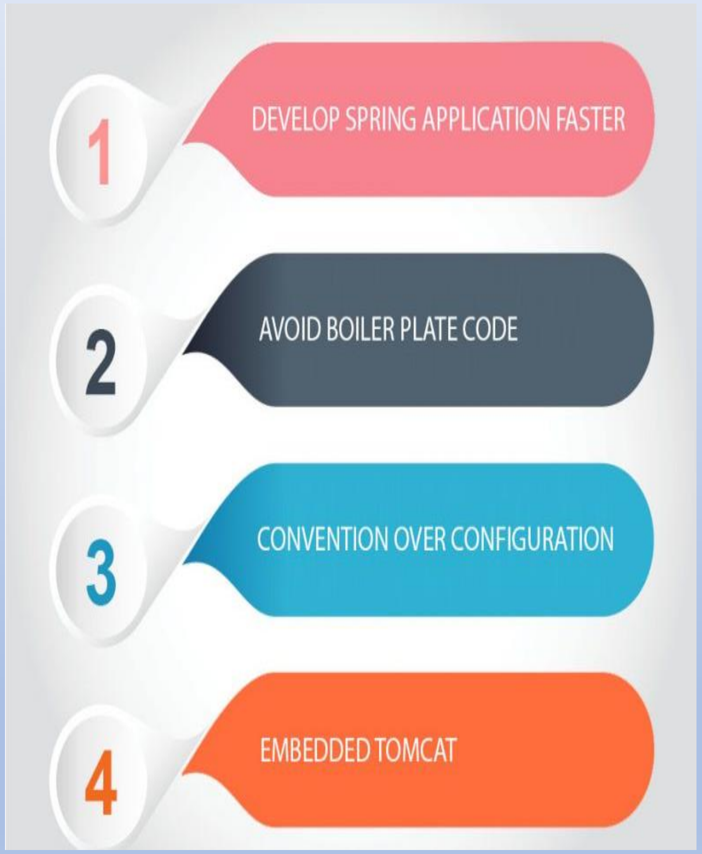

UI
Business Logic
Data Access Layer
Monolithic Architecture
UI
Microservice
Microservice
Microservice
Microservice
Microservice
Microservice
Microservice Architecture

@Amit Kumar

Spring boot

Spring Boot provides a way to create production-ready Spring applications with minimal setup time. The primary goals behind the creation of the Spring Boot project are central to the idea that users should be able to get up and running quickly with Spring. Spring Boot builds upon the Spring ecosystem and third-party libraries, adding in opinions and establishing conventions to streamline the realization of production-ready applications.

Spring Boot is an opinionated framework that helps developers build Spring-based applications quickly and easily. **The main goal of Spring Boot is to quickly create Spring-based applications without requiring developers to write the same boilerplate configuration again and again.**


1
DEVELOP SPRING APPLICATION FASTER
2
AVOID BOILER PLATE CODE
3
CONVENTION OVER CONFIGURATION
4
EMBEDDED TOMCAT

Spring vs Spring boot

Spring boot helps you to create Spring projects faster by eliminating boilerplate configurations.

It takes an opinionated view of Spring platform to bootstrap application faster. Spring boot can reduce your efforts to bootstrapping any spring application

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 server** 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 pom.xml file that internally takes care of downloading the dependencies **JARs** based on Spring Boot Requirement.

Spring Boot Primary Goals

- Provide a radically faster and widely accessible getting-started experience for all Spring development.
- Be opinionated out of the box but get out of the way quickly as requirements start to diverge from the defaults.
- Provide a range of non-functional features that are common to large classes of projects (such as embedded servers, security, metrics, health checks, and externalized configuration).
- Absolutely no code generation and no requirement for XML configuration.

Key Spring Boot features

List of a few key features of the Spring boot:

1. Spring Boot starters
2. Spring Boot autoconfiguration
3. Elegant configuration management
4. Spring Boot actuator
5. Easy-to-use embedded servlet container support

Spring Boot Starters

- Spring Boot offers many starter modules to get started quickly with many of the commonly used technologies, like **SpringMVC, JPA, MongoDB, Spring Batch, SpringSecurity, Solr, ElasticSearch**, etc. These starters are pre-configured with the most commonly used library dependencies so you don't have to search for the compatible library versions and configure them manually.

- For example, the *`spring-boot-starter-data-jpa`* starter module includes all the dependencies required to use Spring Data JPA, along with Hibernate library dependencies, as Hibernate is the most commonly used JPA implementation.

- One more example, when we add the `spring-boot-starter-web` dependency, it will by default pull all the commonly used libraries while developing Spring MVC applications, such as `spring-webmvc`, `jackson-json`, `validation-api`, and `tomcat`.

- Not only does the spring-boot-starter-web add all these libraries but it also configures the commonly registered beans like *`DispatcherServlet`*, *`ResourceHandlers`*, *`MessageSource`*, etc. with sensible defaults.

Spring Boot Autoconfiguration

Spring Boot addresses the problem that Spring applications need complex configuration by eliminating the need to manually set up the boilerplate configuration.

Spring Boot takes an opinionated view of the application and configures various components automatically, by registering beans based on various criteria. The criteria can be:

- Availability of a particular class in a classpath
- Presence or absence of a Spring bean
- Presence of a system property
- An absence of a configuration file

For example, if you have the *`spring-webmvc`* dependency in your classpath, Spring Boot assumes you are trying to build a SpringMVC-based web application and automatically tries to register *`DispatcherServlet`* if it is not already registered. If you have any embedded database drivers in the classpath, such as H2 or HSQL, and if you haven't configured a *`DataSource`* bean explicitly, then Spring Boot will automatically register a *`DataSource`* bean using in-memory database settings.

Elegant Configuration Management

Spring supports externalizing configurable properties using the **@PropertySource** configuration. Spring Boot takes it even further by using the sensible defaults and powerful type-safe property binding to bean properties. Spring Boot supports having separate configuration files for different profiles without requiring many configurations.

Example:

```
@Configuration
@PropertySource("classpath:config.properties")
```

Spring Boot Actuator

Being able to get the various details of an application running in production is crucial to many applications. The Spring Boot actuator provides a wide variety of such production-ready features without requiring developers to write much code. Some of the Spring actuator features are:

- Can view the application bean configuration details
- Can view the application URL mappings, environment details, and configuration parameter values
- Can view the registered health check metrics

Embedded Servlet Container Support

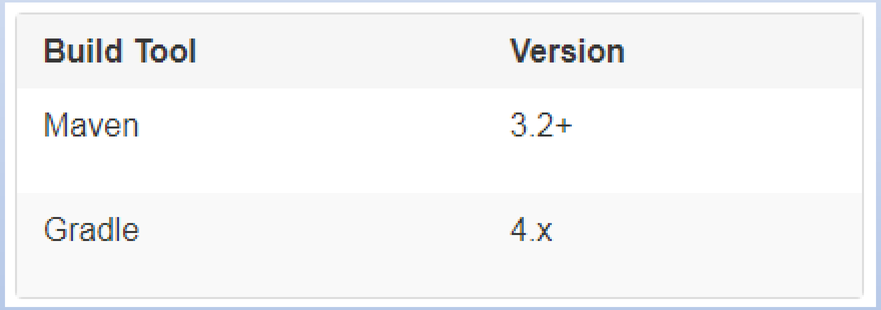
Traditionally, while building web applications, you need to create WAR type modules and then deploy them on external servers like Tomcat, WildFly, etc. But by using Spring Boot, you can create a JAR type module and embed the servlet container in the application very easily so that the application will be a self-contained deployment unit.

Also, during development, you can easily run the Spring Boot JAR type module as a Java application from the IDE or from the command-line using a build tool like Maven or Gradle.

Spring Boot System Requirement

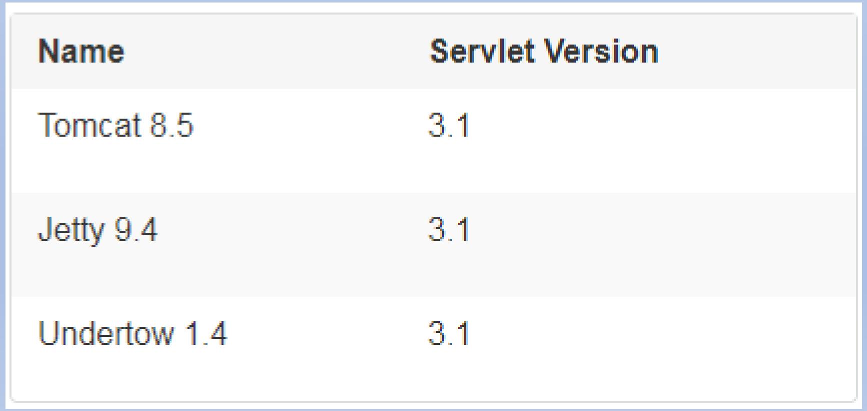
Spring Boot 2+ requires Java 8 or 9 and Spring Framework 5.1.0.RELEASE or above.

Explicit build support is provided for the following build tools:



Build Tool	Version
Maven	3.2+
Gradle	4.x

Servlet Containers/Servers

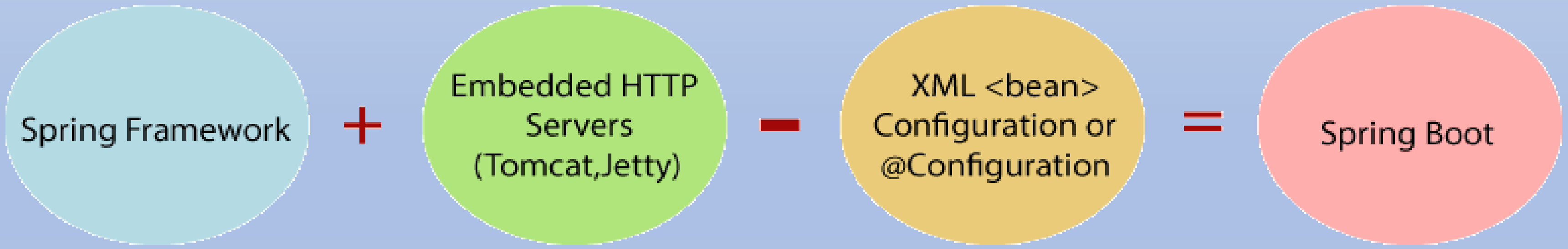


Name	Servlet Version
Tomcat 8.5	3.1
Jetty 9.4	3.1
Undertow 1.4	3.1

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.


Spring Framework
+
Embedded HTTP Servers (Tomcat,Jetty)
-
XML <bean> Configuration or @Configuration
=
Spring Boot

Spring boot

In Spring Boot, there is no requirement for XML configuration (deployment descriptor). It uses convention over configuration software design paradigm that means it decreases the effort of the developer.

- We can use Spring **STS IDE** or **Spring Initializr** to develop Spring Boot Java applications.
- **Why should we use Spring Boot Framework?**
- 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.

Spring boot

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.

Spring boot advantage

- 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 the 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.

Spring boot features

- Web Development
- Spring Application
- Application events and listeners
- Admin features
- Externalized Configuration
- Properties Files
- YAML Support
- Type-safe Configuration
- Logging
- Security

Spring boot features

- **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**
- The 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.

```
public static void main(String[] args)
{
SpringApplication.run(ClassName.class, args);
}
```

Spring boot features

Application Events and Listeners

- Spring Boot uses events to handle the variety of tasks. It allows us to create factories file that is used to add listeners. We can refer it to using the **ApplicationListener key**.
- Always create factories file in META-INF folder like **META-INF/spring.factories**.

Admin Support

- 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 we can work with the same application 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.

Spring boot features

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 of properties file.
- **ype-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.

Spring boot 2.0

- **What's New**
  - Infrastructure Upgrade
  - Spring Framework 5
- **What's Changed**
  - Configuration Properties
  - Gradle Plugin
  - Actuators endpoints
- **What's Evolving**
  - Security
  - Metric

Spring boot 2.0

The pivotal team has upgraded the **infrastructure** in which the following tools are involved:

- Supports **Java 8** or above versions
- Supports Apache **Tomcat 8** or above versions
- Supports **Thymeleaf 3**
- Supports **Hibernate 5.2**
- In **Spring Framework 5**, the Pivotal team upgraded the following:
- Reactive Spring
    - Servlet Container
    - Servlet API
    - Spring MVC
- Functional API
- Kotlin Support

Spring boot 2.0

Dependency upgrades:

- Mongodb 3.11.2
- Spring Security 5.2.1.RELEASE
- Slf4j 1.7.29
- Hibernate Validator 6.0.18.Final
- Hibernate 5.4.8.Final
- Spring Framework 5.2.1
- Spring AMQP 2.2.1
- Spring Security 5.2
- Spring Batch 4.2

Some important and widely used third-party dependencies are upgraded in this release are as follows:

- Micrometer 1.3.1
- Flyway 6.0.7
- Elasticsearch 6.8.4
- JUnit 5.5
- Jackson 2.10

Spring boot Architecture

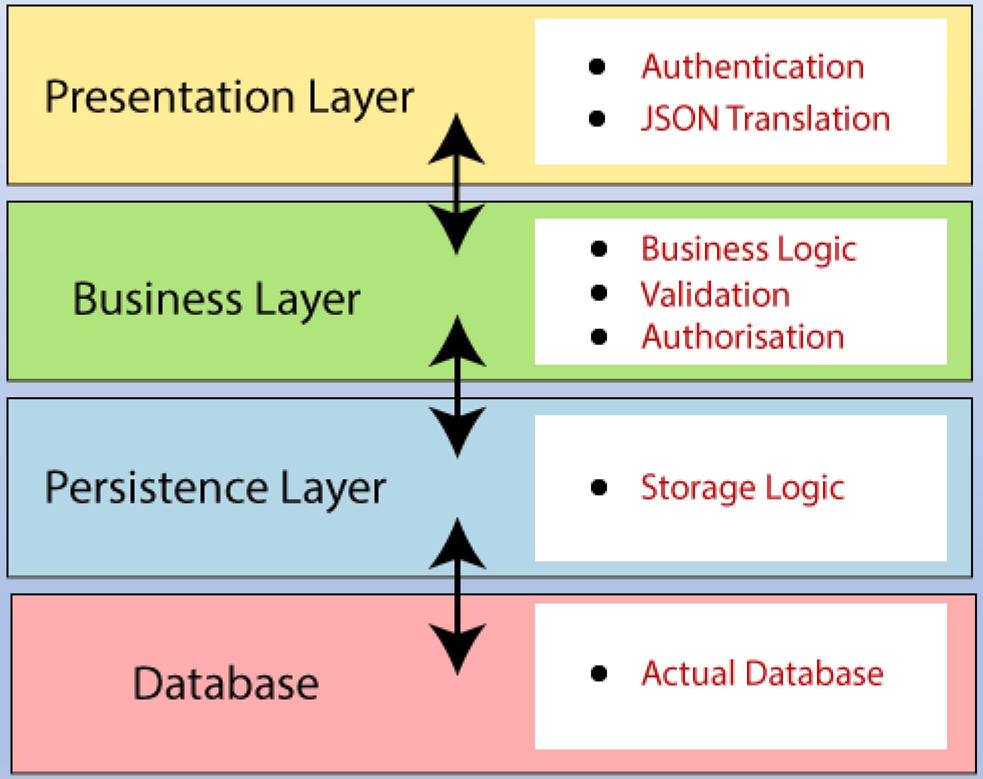
Presentation Layer: The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of **views** i.e., frontend part.

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 **authorization** and **validation**.

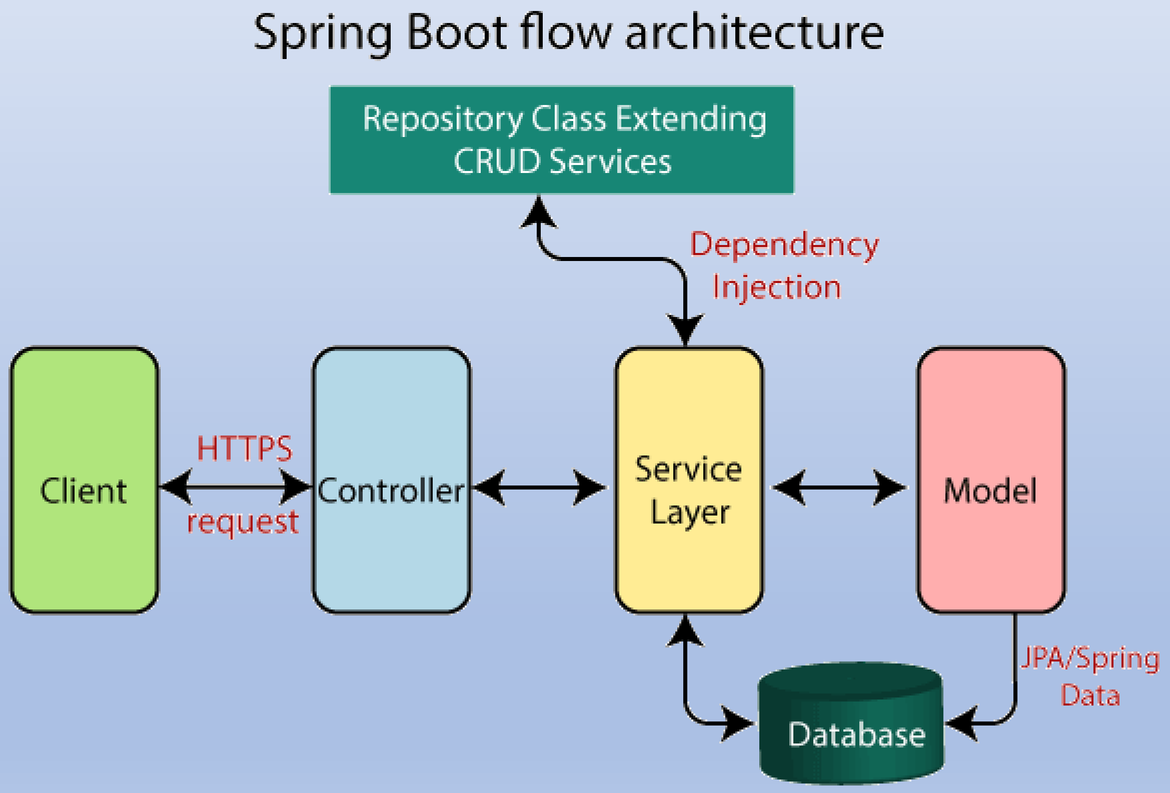
Persistence Layer: The persistence layer contains all the **storage logic** and translates business objects from and to database rows.

Database Layer: In the database layer, **CRUD** (create, retrieve, update, delete) operations are performed.

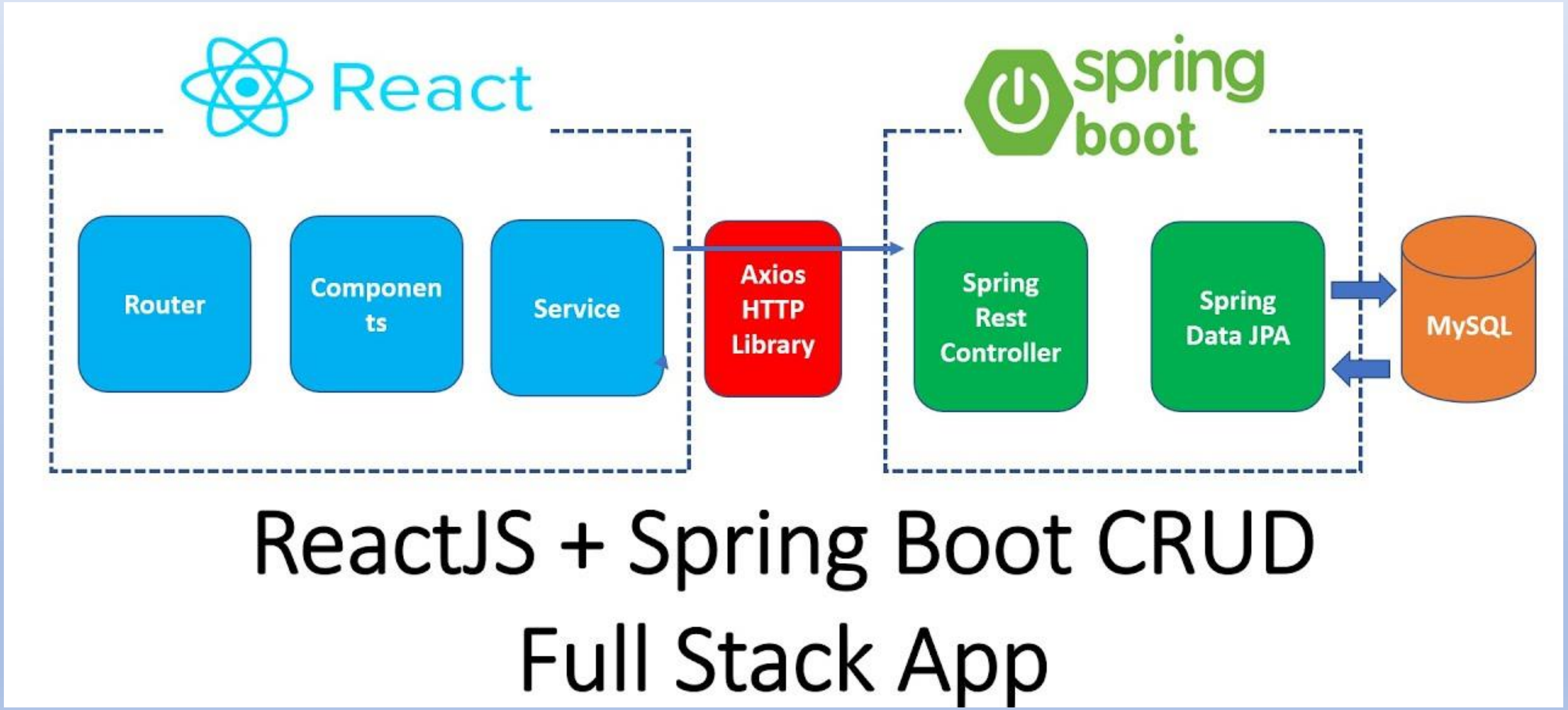
Spring boot Architecture



Spring boot Architecture



Full Stack Architecture

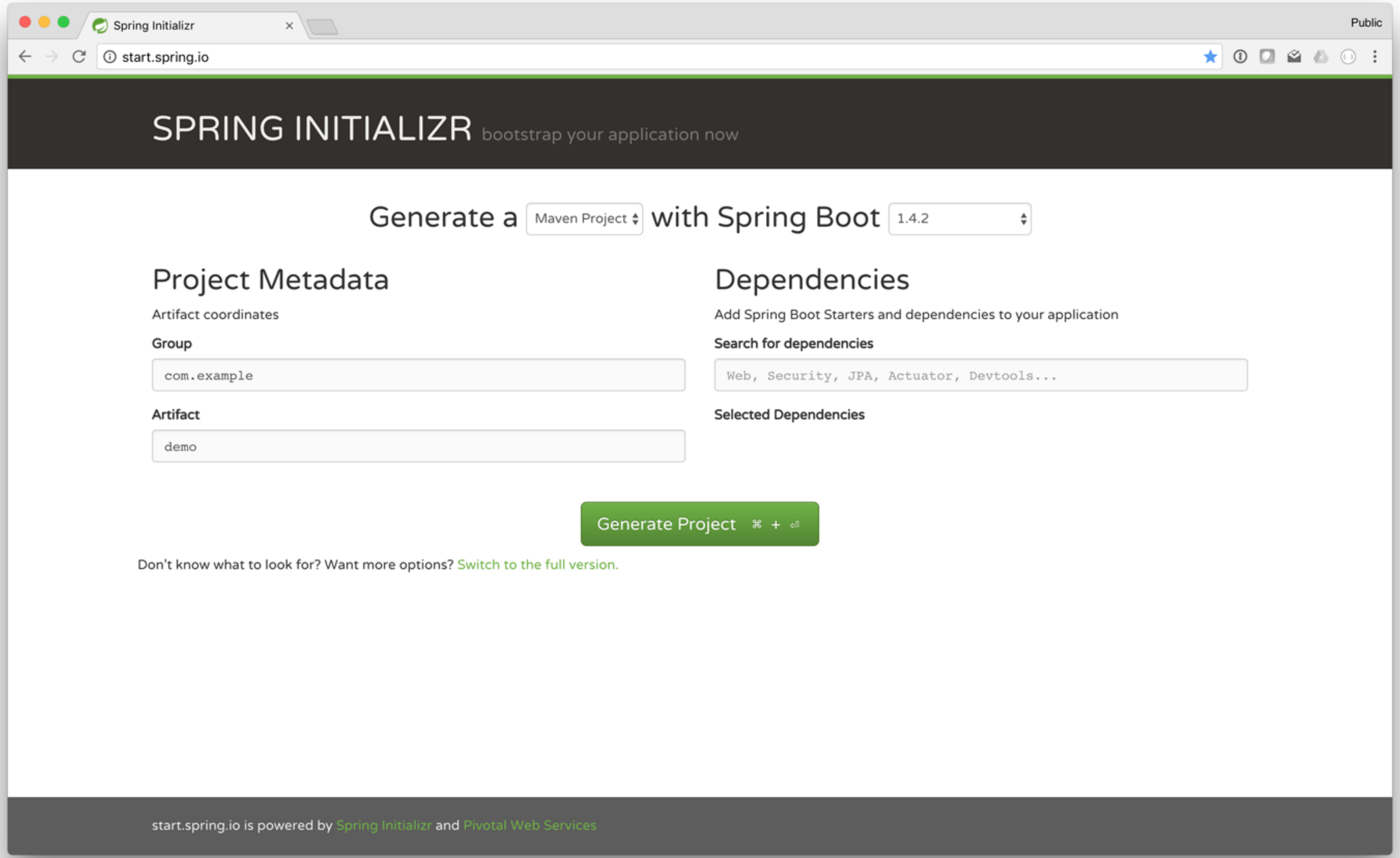


Spring Initializr

The Spring Initializr is an open source project and tool in the Spring ecosystem that helps you quickly generate new Spring Boot applications. Pivotal runs an instance of the Spring Initializr hosted on Pivotal Web Services. It generates Maven and Gradle projects with any specified dependencies, a skeletal entry point Java class, and a skeletal unit test.

In the world of monolithic applications, this cost may be prohibitive, but it’s easy to amortize the cost of initialization across the lifetime of the project. When you begin migrating to a cloud native architecture, you’ll find yourself needing to create more and more applications. Because of this, the friction of creating a new application in your architecture should be reduced to a minimum. The Spring Initializr helps reduce that up-front cost. It’s both a web application that you can consume from your web browser and a REST API that will generate new projects for you.

Spring Initializr



Spring Initializr

start.spring.io

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Spring Boot 1.4.2

Project Metadata

Artifact coordinates

Group

com.example

Artifact

demo

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Don't know what to look for? Want more options? Switch to the full version.

start.spring.io is powered by Spring Initializr and Pivotal Web Services

Spring Initializr

Spring Project	Starter Projects	Maven artifactId
Spring Data JPA	JPA	spring-boot-starter-data-jpa
Spring Data REST	REST Repositories	spring-boot-starter-data-rest
Spring Framework (MVC)	Web	spring-boot-starter-web
Spring Security	Security	spring-boot-starter-security
H2 Embedded SQL DB	H2	h2

Spring boot

Spring Boot automatically configures your application based on the dependencies you have added to the project by using **@EnableAutoConfiguration** annotation. For example, if MySQL database is on your class path, but you have not configured any database connection, then Spring Boot auto-configures an in-memory database.

The entry point of the spring boot application is the class contains **@SpringBootApplication** annotation and the main method.

Spring Boot automatically scans all the components included in the project by using **@ComponentScan** annotation.

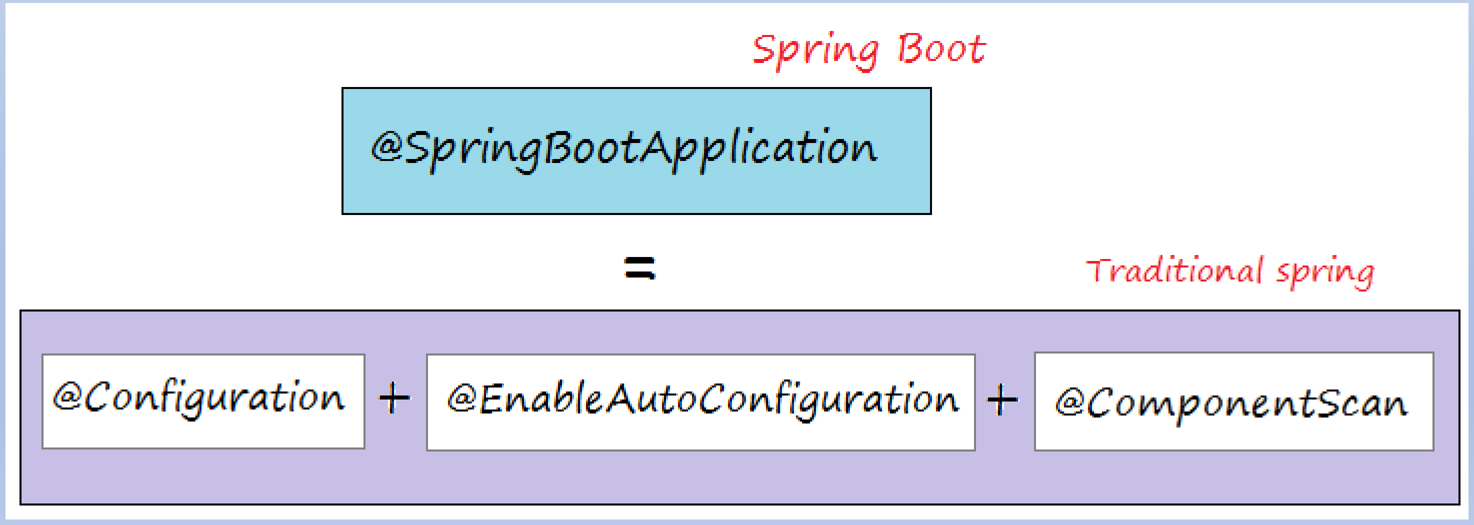
Spring boot

The entry point of the Spring Boot Application is the class contains @SpringBootApplication annotation. This class should have the main method to run the Spring Boot application. **@SpringBootApplication** annotation includes Auto- Configuration, Component Scan, and Spring Boot Configuration.

If you added **@SpringBootApplication** annotation to the class, you do not need to add the **@EnableAutoConfiguration, @ComponentScan** and **@SpringBootConfiguration** annotation. The **@SpringBootApplication** annotation includes all these annotations.

Spring boot Component Scan

Spring Boot application scans all the beans and package declarations when the application initializes. You need to add the **@ComponentScan** annotation for your class file to scan your components added in your project.



Spring boot Run Method internal flow

1. create application context

2. check Application Type

3. Register the annotated class beans with the context

4. Creates an instance of TomcatEmbeddedServletContainer : and adds the context. Used to deploy our jar automatically.

SpringApplication.class

And find here run(String… args) method inside this method you will see the method createApplicationContext() so first it will create application context and inside createApplicationContext() method it will check application type it is SERVLET type Or REACTIVE or DEFAULT context type based on this it will return context.

Now in DEFAULT_CONTEXT_CLASS you will see the class AnnotationConfigApplicationContext.class

```
public AnnotationConfigApplicationContext(Class… annotatedClasses) {
this();
register(annotatedClasses);
refresh();
}
```

SpringApplication.class

- open this class its constructor is used to Register the annotated class beans with the context.
- The classes which are annotated with **@Component, @Service, @Configuration** etc. will be register to the context.
- And in the finally run(-) method auto deploy the **jar/war** to server.
- **@Configuration :**
  It will behave act as bean.
  @EnableAutoConfiguartion :
  it will enable bean based on some condition that we have discussed above.
  @ComponentScan :
  It is mainly used to scan the classes and packages to create the bean.

SpringApplication.class

-
  ```
  @SpringBootApplication
  public class Application {
  public static void main(String[] args) {
  SpringApplication.run(Application.class, args);
  }
  }
  ```

-
  ```
  @SpringBootConfiguration
  @EnableAutoConfiguration
  @ComponentScan(excludeFilters = {
  @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
  @Filter(type = FilterType.CUSTOM, classes =
  AutoConfigurationExcludeFilter.class) })
  public @interface SpringBootApplication {
  ```

Spring Annotation

There are 4 types of annotations:

- Spring Core Annotations
- Spring Web Annotations
- Spring Scheduling Annotations
- Spring Boot Annotations

Spring Core Annotations

Spring core annotations are used in **Spring DI** and **Spring IOC** and belongs to org.springframework.beans.factory.annotation and org.springframework.context.annotation packages.

Annotations:

1. @Autowired
2. @Bean
3. @Qualifier
4. @Required
5. @Value
6. @DependsOn
7. @Lazy
8. @Primary
9. @Scope
10. @Profile
11. @Import
12. @ImportResource
13. @PropertySoruce
14. @PropertySoruces

Spring Core Annotations

@Autowired:

We can use the *@Autowired* to **mark a dependency which Spring is going to resolve and inject**. We can use this annotation with a constructor, setter, or field injection.

Constructor:

```
@RestController
public class CustomerController {
    private CustomerService customerService;

    @Autowired
    public CustomerController(CustomerService customerService) {
        this.customerService = customerService;
    }
}
```

Spring Core Annotations

Setter:

```
@RestController
public class CustomerController {
    private CustomerService customerService;

    @Autowired
    public void setCustomerService(CustomerService customerService) {
        this.customerService = customerService;
    }
}
```

Field:

```
@RestController
public class CustomerController {
    @Autowired
    private CustomerService customerService;
}
```

Spring Core Annotations

Setter:

```
@RestController
public class CustomerController {
    private CustomerService customerService;

    @Autowired
    public void setCustomerService(CustomerService customerService) {
        this.customerService = customerService;
    }
}
```

Field:

```
@RestController
public class CustomerController {
    @Autowired
    private CustomerService customerService;
}
```

Spring Core Annotations

Field:

```
@RestController
public class CustomerController {
   @Autowired(required=true)
   private CustomerService customerService;
}
```

@Autowired has a *boolean* argument called *required* with a default value of *true*. It tunes Spring's behavior when it doesn't find a suitable bean to wire. When *true*, an exception is thrown, otherwise, nothing is wired.

Note, that if we use constructor injection, all constructor arguments are mandatory.

Starting with version 4.3, we don't need to annotate constructors with *@Autowired* explicitly unless we declare at least two constructors.

Spring Core Annotations

@Qualifier:

```
 // setter based DI
   @Autowired
   @Qualifier("emailService")
   public void setMessageService(MessageService messageService) {
       this.messageService = messageService;
   }
   // constructor based DI
   @Autowired
   public MessageProcessorImpl(@Qualifier("emailService") MessageService messageService) {
       this.messageService = messageService;
   }
```

This annotation helps fine-tune annotation-based autowiring. There may be scenarios when we create more than one bean of the same type and want to wire only one of them with a property. This can be controlled using @Qualifier annotation along with the @Autowired annotation.

Spring Core Annotations

@Required:

```
@Required
void setColor(String color) {
   this.color = color;
}
```

The @Required annotation is method-level annotation and applied to the setter method of a bean.

This annotation simply indicates that the setter method must be configured to be dependency-injected with a value at configuration time.

Spring Core Annotations

@Value:

```
@Value("${java.home}")
private String javaHome;
```

```
@Value("${HOME}")
private String homeDir;
```

Spring @Value annotation is used to assign default values to variables and method arguments. We can read spring environment variables as well as system variables using @Value annotation.

Spring @Value annotation also supports SpEL.

```
@Value("#{systemProperties['java.home']}")
private String javaHome;
```

Spring Core Annotations

@DependsOn:

```
@Bean
@DependsOn("fuel")
Engine engine() {
    return new Engine();
}
```

We can use this annotation to make Spring **initialize other beans before the annotated one**. Usually, this behavior is automatic, based on the explicit dependencies between beans.

We only need this annotation **when the dependencies are implicit**, for example, JDBC driver loading or static variable initialization.

We can use *@DependsOn* on the dependent class specifying the names of the dependency beans. The annotation's *value* argument needs an array containing the dependency bean names:

Spring Core Annotations

@Lazy:

```
@Configuration
@Lazy
class VehicleFactoryConfig {

    @Bean
    @Lazy(false)
    Engine engine() {
        return new Engine();
    }
}
```

We use *@Lazy* when we want to initialize our bean lazily. By default, Spring creates all singleton beans eagerly at the startup/bootstrapping of the application context.

However, there are cases when **we need to create a bean when we request it, not at application startup**.

Spring Core Annotations

@Primary:

```
@Component
@Primary
class Car implements Vehicle {}
@Component
class Bike implements Vehicle {}
@Component
class Driver {
    @Autowired
    Vehicle vehicle;
}
@Component
class Biker {
    @Autowired
    @Qualifier("bike")
    Vehicle vehicle;
}
```

Sometimes we need to define multiple beans of the same type. In these cases, the injection will be unsuccessful because Spring has no clue which bean we need.

We already saw an option to deal with this scenario: marking all the wiring points with @Qualifier and specify the name of the required bean.

However, most of the time we need a specific bean and rarely the others. We can use @Primary to simplify this case: if we mark the most frequently used bean with @Primary it will be chosen on unqualified injection points:

Spring Core Annotations

@Scope:

```
@Component
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
public class TwitterMessageService implements MessageService {
}

@Component
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class TwitterMessageService implements MessageService {
}
```

We use *@Scope* to define the scope of a *@Component* class or a *@Bean* definition*.* It can be either *singleton, prototype, request, session, globalSession* or some custom scope.

Spring Core Annotations

@Profile:

```
@Component
@Profile("sportDay")
class Bike implements Vehicle {}
```

If we want Spring to use a *@Component* class or a @Bean method only when a specific profile is active, we can mark it with @Profile. We can configure the name of the profile with the value argument of the annotation.

Spring Core Annotations

@Import:

```
@Configuration
public class ConfigA {

    @Bean
    public A a() {
        return new A();
    }
}

@Configuration
@Import(ConfigA.class)
public class ConfigB {

    @Bean
    public B b() {
        return new B();
    }
}
```

The **@Import** annotation indicates one or more **@Configuration** classes to import.

Spring Core Annotations

@ImportResource:

```
@Configuration
@ImportResource({"classpath*:applicationContext.xml"})
public class XmlConfiguration {
}
```

Spring provides a **@ImportResource** annotation is used to load beans from an applicationContext.xml file into an ApplicationContext. For example: Consider we have applicationContext.xml spring bean configuration XML file on the classpath.

Spring Core Annotations

@PropertySource and @PropertySources:

```
@Configuration
@PropertySource("classpath:/annotations.properties")
@PropertySource("classpath:/vehicle-factory.properties")
class VehicleFactoryConfig {}
```

```
@Configuration
@PropertySources({
   @PropertySource("classpath:/annotations.properties"),
   @PropertySource("classpath:/vehicle-factory.properties")
})
class VehicleFactoryConfig {}
```

@PropertSoruce and @PropertySources used to define property files for application settings.

Spring Boot Annotations

Spring boot provides many annotations for autoconfiguration from the *org.springframework.boot.autoconfigure* and *org.springframework.boot.autoconfigure.condition* packages.

@SrpingBootApplication

@EnableAutoConfiguration

Auto-Configuration Conditions:

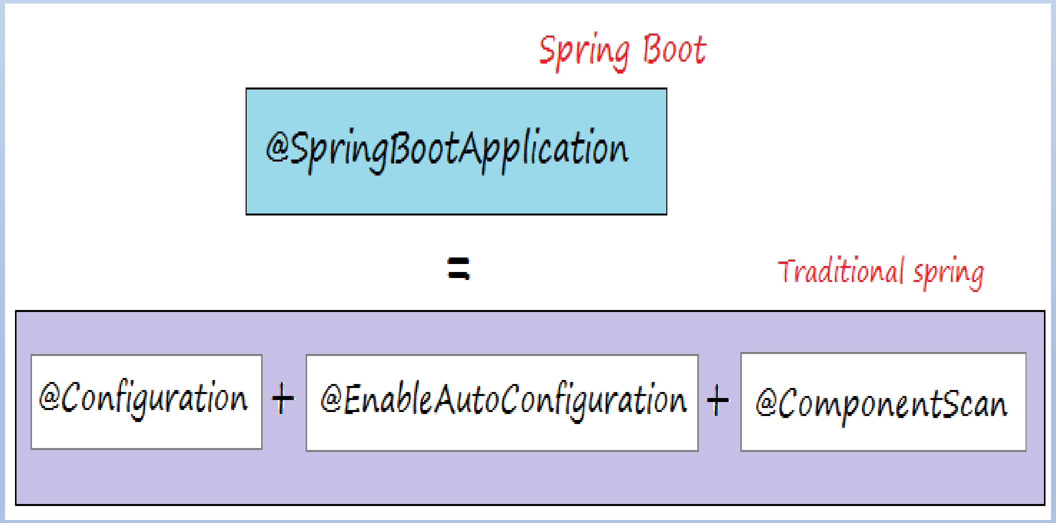
- **@ConditionalOnClass and @ConditionalOnMissingClass**
- **@ConditionalOnBean and @ConditionalOnMissingBean**
- **@ConditionalOnProperty**
- **@ConditionalOnResource**
- **@ConditionalOnWebApplication and @ConditionalOnNotWebApplication**
- **@ConditionalExpression**
- **@Conditional**

Spring Boot Annotations

@SrpingBootApplication:

@SpringBootApplication annotation indicates a configuration class that declares one or more @Bean methods and also triggers auto-configuration and component scanning.

```
// same as @Configuration @EnableAutoConfiguration @ComponentScan
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```



Spring Boot Annotations

@SrpingBootApplication:

SpringBootApplication Annotation Optional Elements:

- Class<?>[] exclude - Exclude specific auto-configuration classes such that they will never be applied.
- String[] excludeName - Exclude specific auto-configuration class names such that they will never be applied.
- Class<?>[] scanBasePackageClass - A type-safe alternative to scanBasePackages() for specifying the packages to scan for annotated components.
- String[] scanBasePackages - Base packages to scan for annotated components.

Spring Boot Annotations

@EnableAutoConfiguration:

@EnableAutoConfiguration annotation tells Spring Boot to "guess" how you want to configure Spring, based on the jar dependencies that you have added. Since spring-boot-starter-web dependency added to classpath leads to configure Tomcat and Spring MVC, the auto-configuration assumes that you are developing a web application and sets up Spring accordingly.

@EnableAutoConfiguration Annotation Optional Elements:

- *Class<?>[] exclude* - Exclude specific auto-configuration classes such that they will never be applied.
- *String[] excludeName* - Exclude specific auto-configuration class names such that they will never be applied.

```
@EnableAutoConfiguration
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Spring Boot Annotations

Auto-Configuration Conditions:

- Usually, when we write our **custom auto-configurations**, we want Spring to **use them conditionally**. We can achieve this with the annotations in this section.
- We can place the annotations in this section on *@Configuration* classes or *@Bean* methods.

```
@Configuration
@ConditionalOnClass(DataSource.class)
class MySQLAutoconfiguration {
   //...
}
```

Spring Boot Annotations

@ConditionalOnBean and @ConditionalOnMissingBean:

- These annotations belong to Class conditions.
  The *@ConditionalOnClass* and *@ConditionalOnMissingClass* annotations let configuration be included based on the presence or absence of specific classes.
- **Example:** In below example, using these conditions, Spring will only use the marked auto-configuration bean if the class in the annotation's argument is present/absent:

```
• @Configuration
• @ConditionalOnClass(LogApi.class)
• public class LogGenAutoConfiguration {
```

Spring Boot Annotations

@ConditionalOnBean and @ConditionalOnMissingBean:

- The *@ConditionalOnBean* and *@ConditionalOnMissingBean* annotations let a bean be included based on the presence or absence of specific beans.
- ***@ConditionalOnBean*** **annotation example:** Use when we want to define conditions based on the presence or absence of a specific bean:

```
@Bean
@ConditionalOnBean(name = "dataSource")
LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    // ...
}
```

- ***@ConditionalOnMissingBean*** **annotation example:** When placed on a @Bean method, the target type defaults to the return type of the method, as shown in the following example:

```
@Configuration
public class MyAutoConfiguration {
 @Bean
 @ConditionalOnMissingBean
 public MyService myService() { ... }
}
```

The *myService* bean is going to be created if no bean of type *MyService* is already contained in the *ApplicationContext*.

Spring Boot Annotations

@ConditionalOnProperty:

- The *@ConditionalOnProperty* annotation lets configuration be included based on a *Spring Environment* property.
- ***@ConditionalOnProperty* annotation example:** With this annotation, we can make conditions on the values of properties:

```
@Bean
@ConditionalOnProperty(
   name = "usemysql",
   havingValue = "local"
)
DataSource dataSource() {
   // ...
}
```

Spring Boot Annotations

@ConditionalOnResource:

- The *@ConditionalOnResource* annotation lets configuration be included only when a specific resource is present:

```
@ConditionalOnResource(resources = "classpath:mysql.properties")
Properties additionalProperties() {
   // ...
}
```

Spring Boot Annotations

@ConditionalOnWebApplication and @ConditionalOnNotWebApplication:

- The *@ConditionalOnWebApplication* and *@ConditionalOnNotWebApplication* annotations let configuration be included depending on whether the application is a "web application". A web application is an application that uses a Spring *WebApplicationContext*, defines a session scope, or has a *StandardServletEnvironment*.
- ***@ConditionalOnWebApplication*** **annotation sample code:** With these annotations, we can create conditions based on if the current application is or isn't a web application:

```
@ConditionalOnWebApplication
HealthCheckController healthCheckController() {
   // ...
}
```

Spring Boot Annotations

@ConditionalExpression:

We can use this annotation in more complex situations. Spring will use the marked definition when the *SpEL expression* is evaluated to true:

```
@Bean
@ConditionalOnExpression("${usemysql} && ${mysqlserver == 'local'}")
DataSource dataSource() {
   // ...
}
```

Spring Boot Annotations

@Conditional:

For even more complex conditions, we can create a class evaluating the custom condition. We tell Spring to use this custom condition with *@Conditional*:

```
@Conditional(HibernateCondition.class)
Properties additionalProperties() {
   //...
}
```