Magical Beans and Mystery

Tiffany Jernigan
Senior Developer Advocate, www.tiffanyfay.dev
@tiffanyfayj

Timo Salm
Senior Lead Tanzu DevX Solution Engineer, VMware Tanzu
@salmto

May 2024

Who We Are

Tiffany Jernigan

Senior Developer Advocate

X (Twitter): @tiffanyfayj
LinkedIn: https://linkedin.com/in/tiffanyfayj

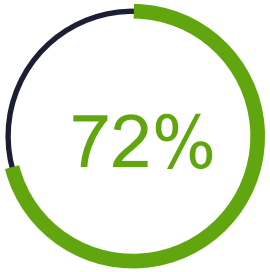
Timo Salm

Senior Lead Tanzu DevX Solution Engineer – EMEA
Broadcom

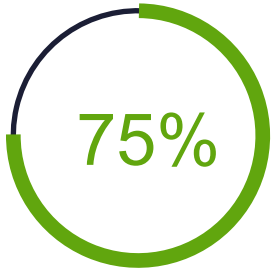
X (Twitter): @salmto
LinkedIn: https://linkedin.com/in/timosalm

spring® by VMware Tanzu

The most popular application development framework on the JVM



of Java developers utilize Spring Boot



like Spring because it's very stable, scalable, and secure

Sources: https://www.jetbrains.com/lp/devecosystem-2023/java/
https://tanzu.vmware.com/content/ebooks/the-state-of-spring-2022

History of Spring

Created in 2003 as a lightweight alternative to address the complexity of the early J2EE specifications. Spring and Java/Jakarta EE are not in competition, they are complementary.

Why Spring?

- Focused on speed, simplicity, and productivity
- Enterprise and production ready
- Extensive ecosystem
- Large, active developer community

Official Spring Projects

spring.io/projects

Spring Framework

Provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.

Spring Boot

Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.

Spring Session

Provides an API and implementations for managing a user's session information.

Spring Integration

Supports the well-known Enterprise Integration Patterns through lightweight messaging and declarative adapters.

Spring Data

Provides a consistent approach to data access – relational, non-relational, map-reduce, and beyond.

Spring Cloud

Provides a set of tools for common patterns in distributed systems. Useful for building and deploying microservices.

Spring HATEOAS

Simplifies creating REST representations that follow the HATEOAS principle.

Spring Modulith

Spring Modulith allows developers to build well-structured Spring Boot applications and guides developers in finding and working with application modules driven by the domain.

Spring Cloud Data Flow

Provides an orchestration service for composable data microservice applications on modern runtimes.

Spring Security

Protects your application with comprehensive and extensible authentication and authorization support.

Spring REST Docs

Lets you document RESTful services by combining hand-written documentation with auto-generated snippets produced with Spring MVC Test or REST Assured.

Spring AI

Spring AI is an application framework for AI engineering.

Spring Authorization Server

Provides a secure, light-weight, and customizable foundation for building OpenID Connect 1.0 Identity Providers and OAuth2 Authorization Server products.

Spring for GraphQL

Spring for GraphQL provides support for Spring applications built on GraphQL Java.

Spring Batch

Simplifies and optimizes the work of processing high-volume batch operations.

Spring CLI

A CLI focused on developer productivity

AND Spring AMQP, Spring Flo, Spring for Apache Kafka, Spring for Apache Pulsar, Spring LDAP, Spring Shell, Spring Statemachine, Spring Web Flow, Spring Web Services

Spring Framework

Core Functionality

Inversion of Control and Dependency Injection

What is Inversion of Control (IoC)?

Design principle that transfers object creation and management from application code to a container or framework.

Decouples the execution of a task from its implementation.

What is Dependency Injection (DI)?

Design pattern to implement IoC that provides an object with its dependencies, rather than the object creating them itself.

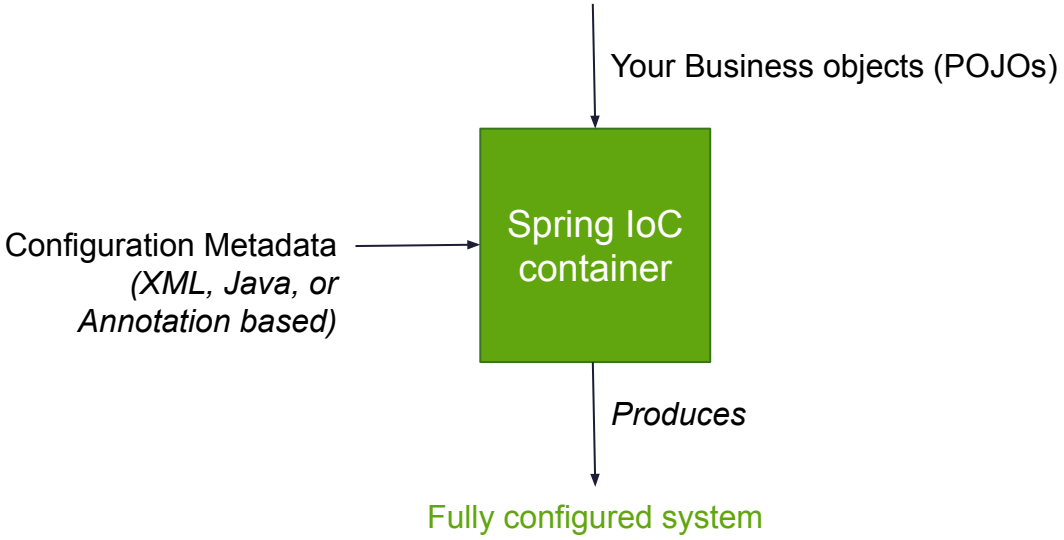
```java
public class Service {
    private final Repository repository;

    public Service(Repository repository) {
        this.repository = repository;
    }
}
```

Example: Dependency provided through a class constructor

Key Benefits

- Loose Coupling
- Enhanced Testability
- Improved Maintainability
- Increased Flexibility:

Spring IoC Container and Beans



Spring IoC container: Manages POJOs based on configuration metadata and uses dependency injection to achieve inversion of control

Spring Beans: Objects the Spring IoC container instantiates, assembles, and manages

BeanFactory: Root interface for accessing the Spring container. It provides basic functionalities for managing beans

ApplicationContext: Sub-interface of the BeanFactory that offers additional functionality like resolving messages, supporting internationalization, publishing events, and application-layer specific contexts

Spring IoC Container and Beans

Configuration Metadata can be defined in the following forms:

- XML
- Java
- Annotations (drive only the dependency injection)
- Stereotyped annotations with auto detection
- Groovy Bean Definition DSL

```java
@Configuration
public class AppConfig {
     @Bean
     public MyService myService() {
          return new MyServiceImpl();
     }
}
```

Example: Bean configuration in Java

Bean definitions include metadata like the Java class, name, constructor arguments, properties, and scope.

Bean scopes control how objects are created from a bean definition:

- singleton: (Default) Creates only a single object instance for each bean definition
- prototype: Creates any number of object instances for each bean definition
- With a web-aware Spring ApplicationContext, there are also request, session, application, and websocket scopes supported

Environment Abstraction

Introduction and Profiles

The **Environment** abstraction is integrated in the IoC container and provides the profiles and properties key aspects of the application environment.

Enables you to separate configuration from code to run your application in different environments without code changes.

Profiles allow the registration of different beans in different environments.
The `@Profile` annotation lets you define which components will get registered for an active profile.

```java
@Bean
@Profile("development")
public DataSource dataSource() { ... }

@Bean("dataSource")
@Profile("production")
public DataSource prodDataSource() { ... }
```

Example: Bean configuration for different profiles

Profiles can be activated programmatically (Environment API) or declaratively with the `spring.profiles.active` property.

Multiple profiles can be activated at once.

Environment Abstraction

Properties

A **PropertySource** is a simple abstraction over any source of key-value pairs.

The Environment abstraction provides search operations over a configurable hierarchy of them.

The `StandardEnvironment` is configured with two PropertySource objects for JVM system properties and system environment variables.

PropertySources and their hierarchy are configurable programmatically (Environment API) and they can be added declaratively with the **`@PropertySource`** annotation.

Another option to access the properties is to inject them via the **`@Value`** annotation.

```java
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig {

 @Bean
 public String recipesEnv(Environment env) {
      return env.getProperty("recipes");
 }

 @Bean
 public String recipesAtValue(@Value("${recipes}")
String recipesPropertyValue) {
      return recipesPropertyValue;
 }
}
```

Example: Access properties via the Environment API and @Value annotation

Resources

The **`Resource`** interface is an abstraction to access low-level resources.

Spring includes several built-in Resource implementations like:

- UrlResource
- ClassPathResource
- FileSystemResource

The **`ResourceLoader`** interface is an abstraction for loading resources.

ApplicationContexts implement the `ResourceLoader` interface, and can be used to obtain `Resource` instances.

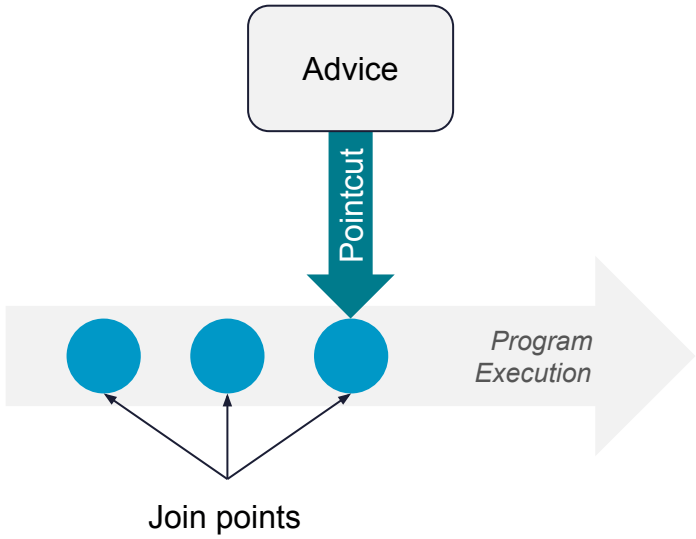
JAVA

```java
public interface Resource extends InputStreamSource
{
   boolean exists();
   File getFile() throws IOException;
   byte[] getContentAsByteArray() throws IOException
   …
}

public interface ResourceLoader {
   Resource getResource(String location);
   ClassLoader getClassLoader();
}

Resource template =
ctx.getResource("classpath:application.properties");
```

Example: Available interfaces to access resources

Aspect Oriented Programming (AOP)

Aspects enable the modularization of concerns that cut across multiple types and objects (e.g. transaction management).



Join point: A point during the execution of a program

Pointcut: A predicate that matches join points.

Advice: An action taken by an aspect that runs at any join point matched by the pointcut

Custom aspects can be implemented via a schema-based approach or the `@AspectJ` annotation style

Spring Framework

Beyond the Core Functionality

Unit and Integration Testing

Support for unit and integration testing is provided in the spring-test module.

Unit Testing

Dependency injection enables us to test most of our code independent of the Spring IoC Container with **JUnit** or **TestNG.**

Spring also provides **mock implementations** like `MockEnvironment` and `MockPropertySource` for code that depends on environment-specific properties, **helper classes** like `AopTestUtils` or `ReflectionTestUtils`.

Integration Testing

The **Spring TestContext Framework** provides generic, annotation-driven testing support that is agnostic of the testing framework in use.

It manages Spring IoC container caching between tests, provides transaction management and more.

Web Applications

The **spring-webmvc** module is a web framework in Spring built on the Servlet API.

In addition, Spring Framework 5.0 introduced the reactive-stack web framework module spring-webflux built on a Reactive Streams API.

In Spring Web MVC, the **DispatcherServlet** provides a shared algorithm for request processing, actual work is performed by delegate components.

To enable MVC configuration, the **@EnableWebMvc annotation** has to be added to an `@Configuration` class.

Annotations like **@Controller** or **@RestController** are used to configure request mappings, request input, exception handling, and more.

JAVA

```java
@RestController
public class RecipesResource {

  @GetMapping("/api/v1/recipes")
  public ResponseEntity<List<Recipe>> fetchRecipes()
  {
     return ResponseEntity.ok(
       Collections.singletonList(new Recipe(".."))
     );
  }
}
```

Example: Request mapping configuration

Data Access

Transaction Management

The Spring Framework provides an abstraction for transaction management with a consistent programming model across different transaction APIs like JDBC, Hibernate, and JPA.

Transactions can be declared programmatically, or declaratively (recommend) with the **`@Transactional`** annotation based on Spring AOP.

The **`@EnableTransactionManagement`** annotation To enables transaction management for the declarative approach.

A key part of the Spring transaction abstraction is the **transaction strategy**, which is defined by a **`TransactionManager`**.

JAVA

```java
@Service
public class RecipeService {
    private final RecipeRepository repository;
    public RecipeService(
      RecipeRepository repository
    ) {
      this.repository = repository;
    }

    @Transactional
    public void addRecipes(List<Recipe> recipes) {
      recipes.forEach(r ->
        repository.insertRecipe(r.name())
      );
    }
}
```

Example: Declaring a transaction

Data Access

JDBC

JDBC(Java Database Connectivity) is an API for universal data access, and Spring takes care of all the low-level details of it, like transaction handling.

`JdbcTemplate` is the classic, “lowest-level” approach to use JDBC with Spring.

Introduced in version 6.1, `JdbcClient` is more focused and convenient facade on top of it.

The spring-jdbc module also provides **support for Embedded Databases** like HSQL or H2 for development or integration tests.

JAVA

```java
jdbcClient.sql("SELECT * FROM RECIPE")
    .query(
        (rs, rowNum) -> new Recipe(rs.getInt("id"), rs.getString("name"))
    )
    .list();

jdbcClient.sql("INSERT INTO recipe (name) VALUES (:name)")
    .param("name", name)
    .update();
```

Example: Using the JdbcClient to get and insert recipes

More Integrations

The Spring Framework also provides core support for number of other technologies:

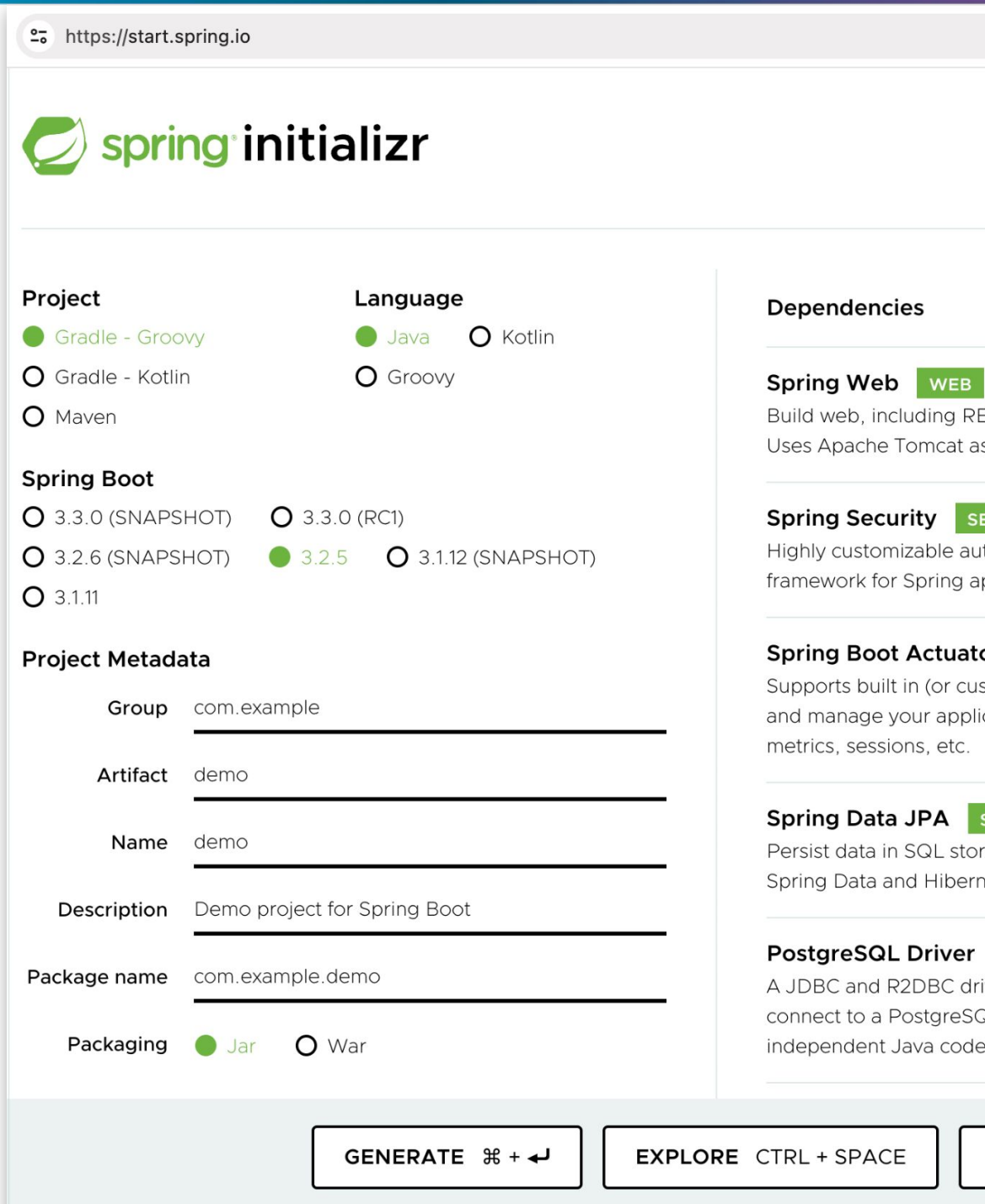
- REST Clients
- JMS (Java Message Service)
- JMX (Java Management Extensions)
- Email
- Task Execution and Scheduling
- Cache Abstraction
- Observability Support
- JVM Checkpoint Restore
- CDS

Spring Boot

Build anything with Spring Boot

Easily create self-contained, production-grade, 12-Factor applications that you can "just run".

- Get started in seconds using Spring Initializr, CLI, or your favorite IDE
- Code with zero to little configuration
- Production-ready features such as micrometer, tracing, metrics, and health status
- Autoconfigure and embed Tomcat, Jetty, and Undertow
- Developer-friendly tooling (such as auto-restart, SSH, and live reload)



Dependency Management and Starters

Management of dependencies is simplified by providing a curated list of compatible dependencies for each Spring Boot version that can be used with Maven and Gradle.

GRADLE

```
plugins {
 id 'java'
 id 'org.springframework.boot' version '3.3.0'
 id 'io.spring.dependency-management' version '1.1.5'
}
```

MAVEN

```
<parent>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-parent</artifactId>
 <version>3.3.0</version>
 <relativePath/>
</parent>
```

Example: How to configure dependency management for Gradle and Maven

Starters are a consistent and supported set of managed transitive dependencies to simplifying the process of setting up projects.

All official starters follow the naming pattern **`spring-boot-starter-*`**, where * is a particular type of application.

GRADLE

```
dependencies {
 implementation
  'org.springframework.boot:spring-boot-starter-web'

 testImplementation
  'org.springframework.boot:spring-boot-starter-test'
}
```

Example: Starters for Web Applications and Testing

Auto-Configuration and the @SpringBootApplication Annotation

Spring Boot automatically configures your app based on added jar dependencies.

The `@EnableAutoConfiguration` or **`@SpringBootApplication`** annotations enable auto-configuration.

The `@SpringBootApplication` annotation additionally enables `@Component` scan and registration of extra beans in the context or the import of additional configuration classes.

The static **`SpringApplication.run`** method provides a convenient way to start a Spring application from a `main()` method.

JAVA

```java
@SpringBootApplication(exclude = {
  DataSourceAutoConfiguration.class
})
public class RecipeFinderApplication {
  public static void main(String[] args) {
    SpringApplication.run(
      RecipeFinderApplication.class,args
    );
  }
}
```

Example: Simplified setup of a Spring Boot application with auto configuration

Externalized Configuration

Spring Boot configures **several additional PropertySources.**

One of them automatically finds and loads `application.properties` or `application.yaml` config data files from e.g. the root of the classpath or current directory.

They can be profile specific with the `application-{profile}` naming convention.

YAML

```yaml
# application.yaml
spring:
  application.name: recipe-finder
  datasource.url: jdbc:hsqldb:mem:db
# application-production.yaml
spring.datasource.url:jdbc:postgresql://db:5432/db
```

Example: Configuration for different environments.

In addition to the `@Value` annotation for a single property, with the `@ConfigurationProperties` you can inject multiple properties as a strongly typed bean.

Packaging and Running Spring Boot Applications

Development

By default, the application is packaged as an **executable JAR** with an **embedded HTTP server.**

Spring Boot Maven and Gradle plugins provide an improved UX for compiling and running your app.

The `spring-boot-devtools` module adds automatic restart, live-reload and more capabilities.

```shell
$ java -jar my-application.jar
# Maven
$ mvn spring-boot:run
# Gradle
$ gradle bootRun
```

Example: How to run an app packaged as JAR and with Maven and Gradle

Production

An **executable JAR** is a valid option.

Unpacking the JAR can be an advantage like for start-up time improvements.

```shell
$ jar -xf myapp.jar
$ java -cp "BOOT-INF/classes:BOOT-INF/lib/*"
com.example.MyApplication
```

For start-up time improvements and other benefits, Spring also provides support for **GraalVM Native Images**, **Project CraC**, and **Class Data Sharing**.

For container image building Spring supports **Cloud Native Buildpacks**, that automatically apply best practices.

Unit and Integration Testing with Spring Boot

The **`spring-boot-starter-test`** module provides some additional utilities and annotations for testing.

The **`@SpringBootTest`** annotation is a replacement for the `@ContextConfiguration` annotation when you need Spring Boot features.

By default, `@SpringBootTest` provides a mock web environment. Embedded servers can be configured with its `webEnvironment` attribute.

`@*Test` annotations search for configuration automatically without explicit definition. Customization can be applied with a nested **`@TestConfiguration`** class.

A Mockito mock for a bean can be defined with a **`@MockBean`** annotation.

Testcontainers for Integration Testing

Spring Boot's support for Testcontainers enables you to start required services for your integration tests in containers before the tests run.

Monitoring, Metrics, And More

The **`spring-boot-actuator`** module provides all of Spring Boot's "production-ready features".

Actuator endpoints let you monitor and interact with your applications and are available via HTTP or JMX. Several are available ootb.

The endpoints have to be manually exposed via configuration (`management.endpoint.<id>.enabled`), and are for HTTP by default available under `/actuator` path.

Observability (logging, metrics and traces)

Spring Boot provides support for common monitoring systems with its dependency management and auto-configuration via the vendor-independent **Micrometer** project.

Micrometer Observation is used for metrics and the **Micrometer Tracing** extension for traces,

Spring Boot for Web Applications and Data Access

Dependency management and auto configuration support is provided via related stater dependencies.

Spring Boot Starter for Spring Web MVC

The auto-configuration replaces the need for `@EnableWebMvc`.

It also registers additional (HttpMessage)Converters, Formatter beans, and MessageCodesResolver, adds static index.html support, and more.

Spring Boot Starter for Spring Data JDBC

External configuration properties in `spring.datasource.*` make it easier to configure a `DataSource`.

In addition to using the `JdbcClient`, the `spring-boot-starter-data-jdbc` provides a **`CrudRepository`** interface that provides methods for common database operations and automatically generates the SQL them.

For more advanced queries, a **`@Query`** annotation is provided.

SECURE ANYTHING WITH SPRING SECURITY

Spring Security is a powerful and highly customizable authentication and access-control framework.

It is the standard for securing Spring-based applications.

It provides support for authentication, authorization, and protection against common exploits.

As authentication mechanisms, it supports for example Basic Auth, OAuth, and SAML.

Spring Boot Starter for Spring Security

Secure your application with minimal effort and configuration.

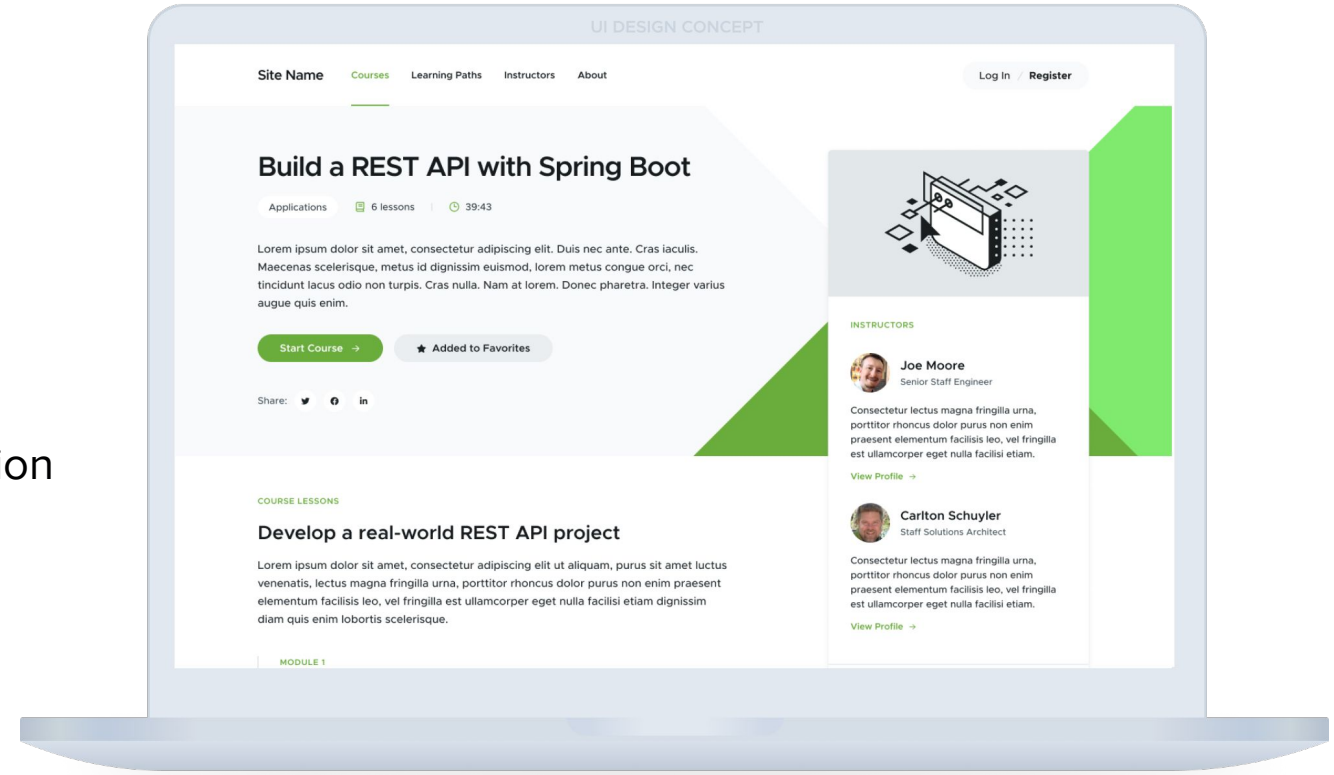
Auto-configuration is based on the Spring Security modules on the classpath, like special configuration if the `spring-security-oauth2-client` module is detected.

spring ACADEMY

spring.academy

Free on-demand education developed and curated by the world's foremost experts in Spring.

- Guides, Courses, and Learning Paths
- Interactive Learning Environments
- Everything you need for Spring Certification



Thank You

Follow us
@salmto
@tiffanyfayj