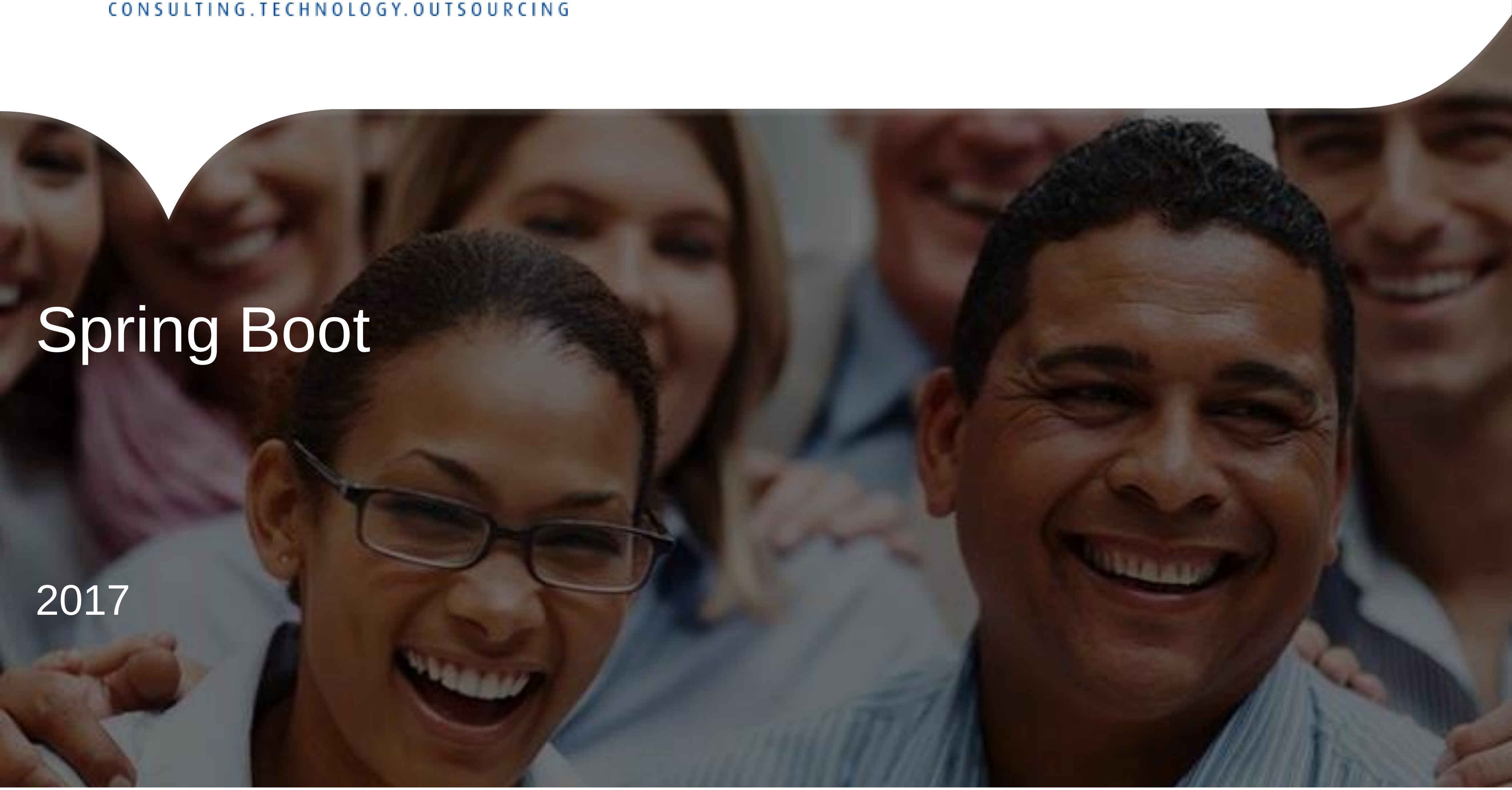
CONSULTING.TECHNOLOGY.OUTSOURCING

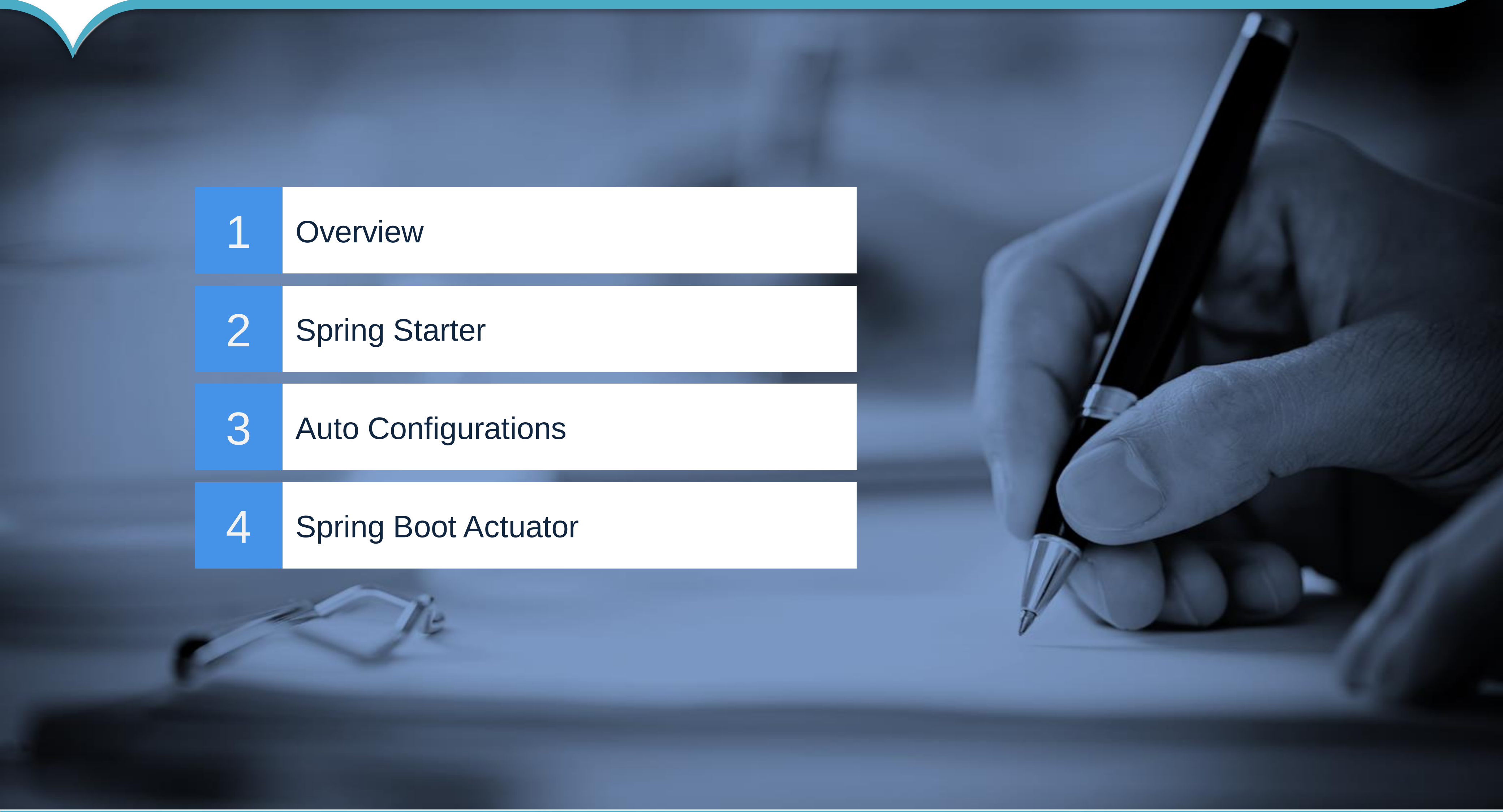


Spring Boot

2017

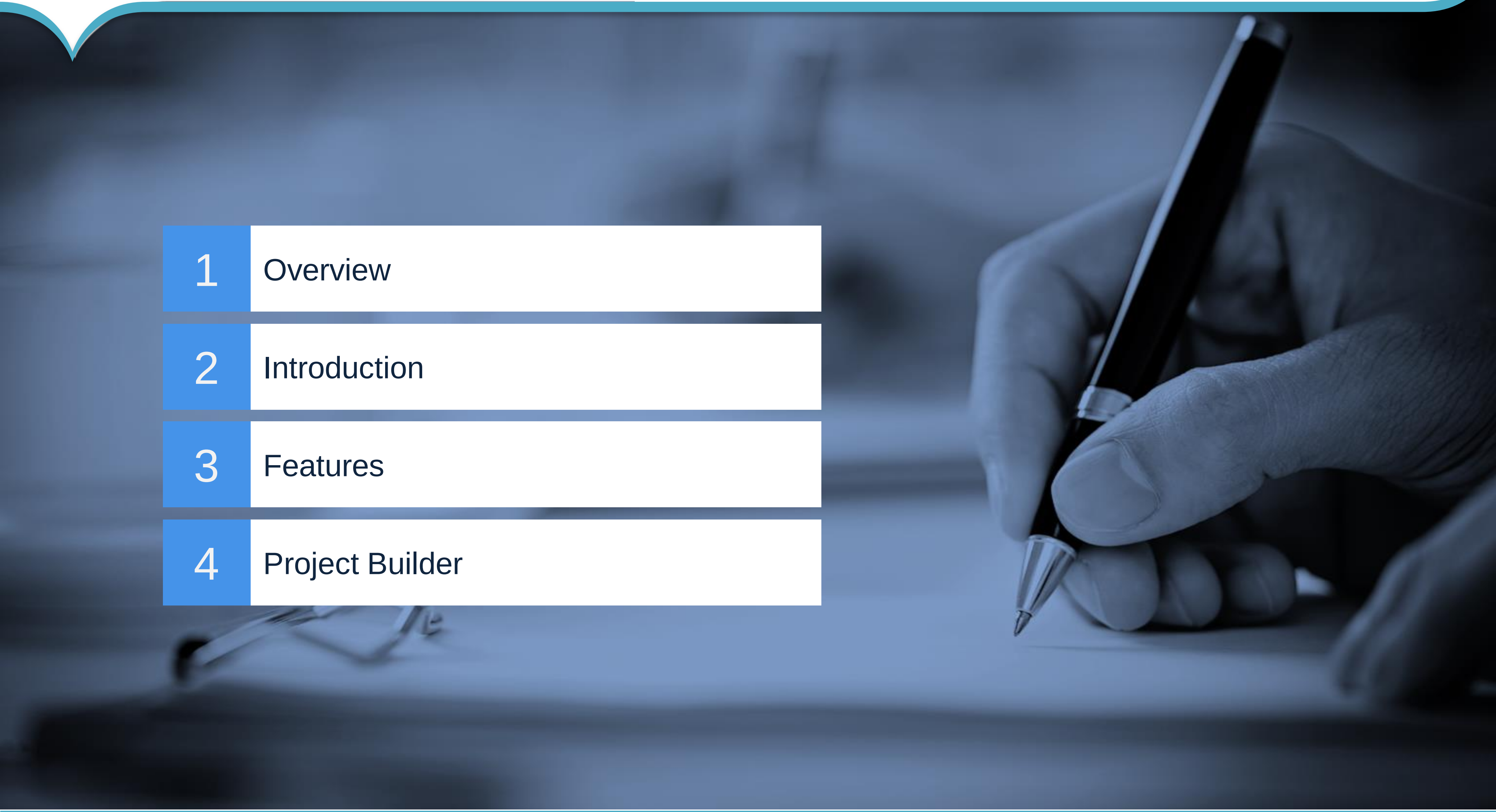
People matter, results count.

Agenda



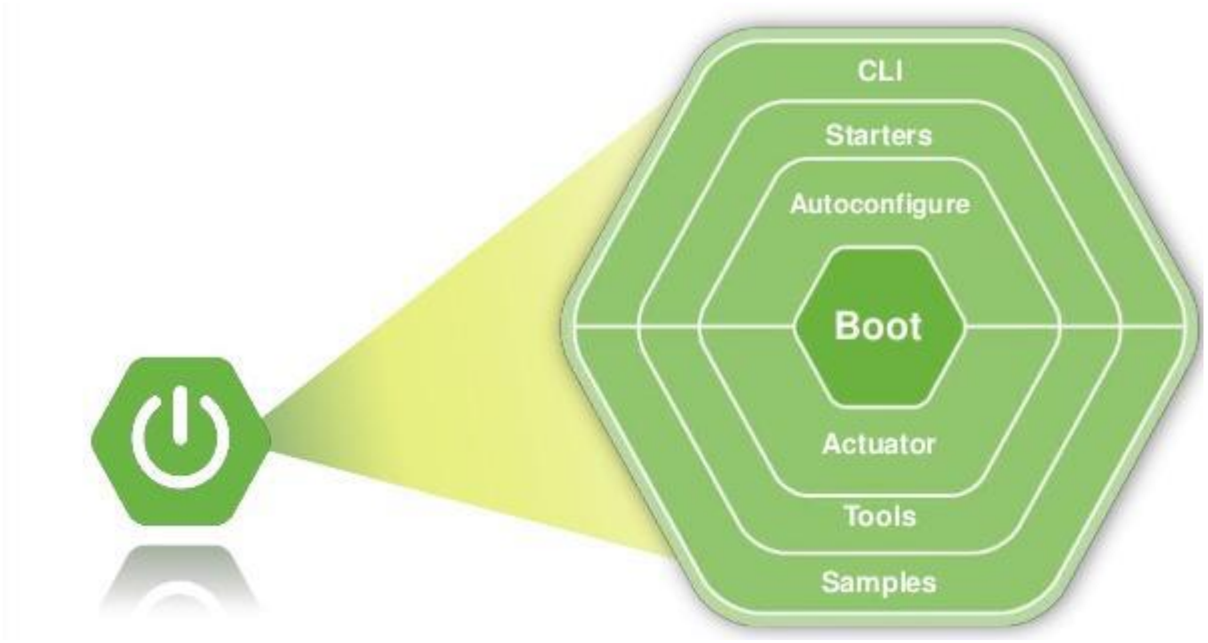
Spring Boot - Overview

Outline



Overview

- Spring is a fantastic open source framework, addresses the complexity of application development. One of the chief advantages of the Spring framework is its layered architecture, which allows you to be selective about which of its components you use. Spring is a cohesive framework for J2EE application development.
- Spring boot is developed on top of the Spring Framework; it's a modular framework which helps in building application infrastructure, taking care of CLI(command line interface), boot strapping, dependencies, framework integrations, testing, tools, auto-configuration and actuator. It favours convenience over configuration and so it's a great way to get quickly ride over spring platform.

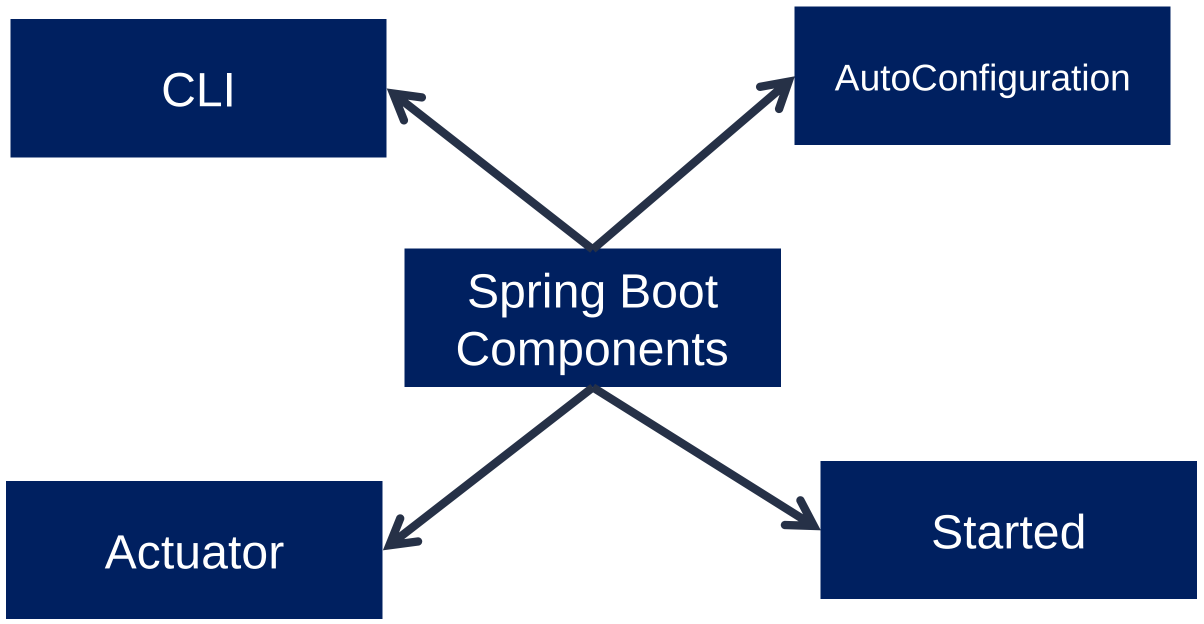


Introduction

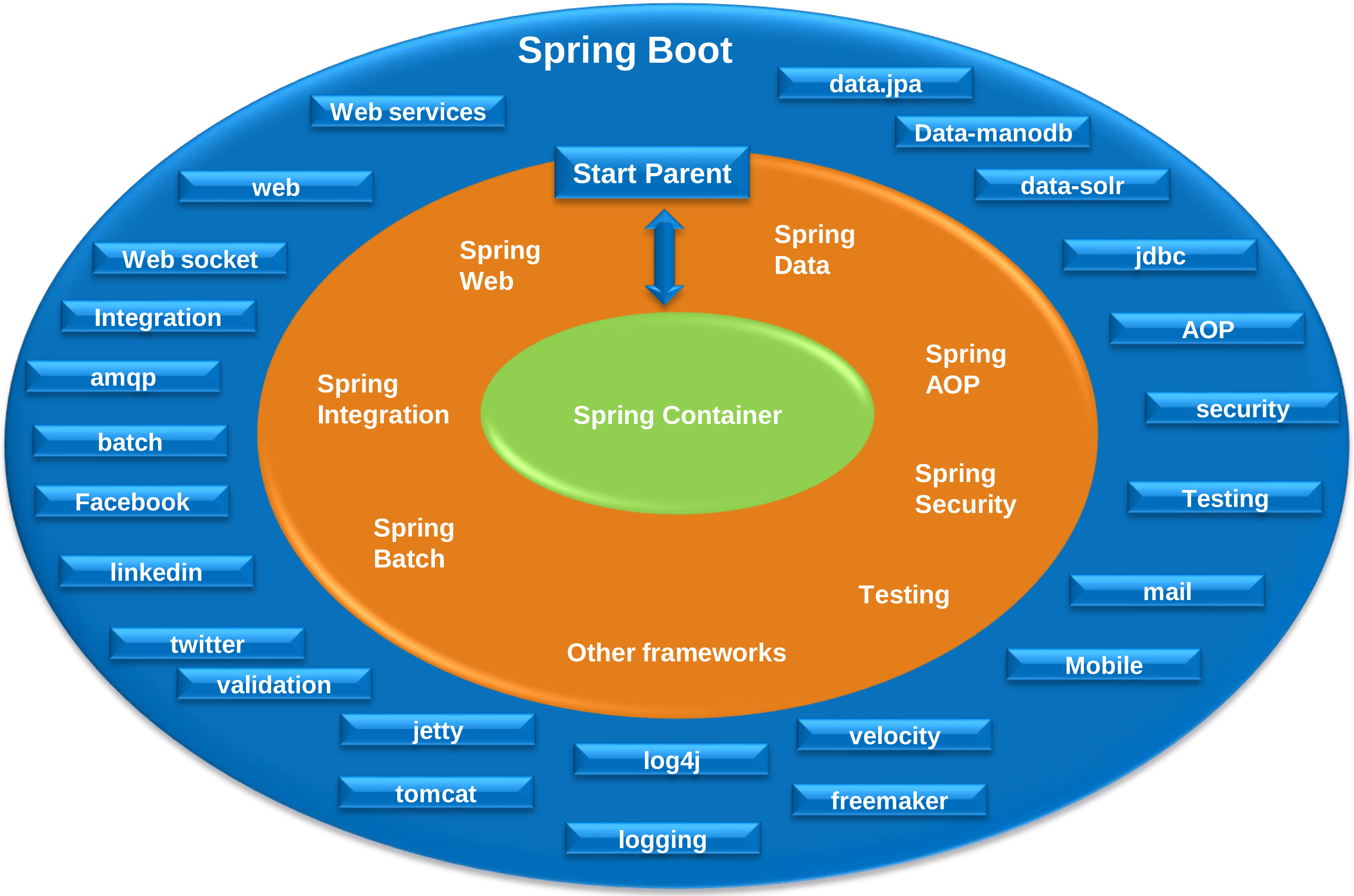
- Spring boot encapsulates spring and many other frameworks and tools like apache velocity, log4j, tomcat and provide starters to take care of these frameworks and tools integration, the figure 1.0 shows boot Starter.
- The spring-boot-starter-parent root module hooks your application with spring boot; and rest of the starter modules can be included as needed.
- Spring Boot support xml configuration to get up with legacy applications and favours Java-based configuration.

Overview

- Key Components of Spring Boot Framework:
  - Spring Boot Starters
  - Spring Boot AutoConfigurator
  - Spring Boot CLI
  - Spring Boot Actuator



Introduction



Features

- Faster
  - Generate project scaffolding using Initializr and your IDE
  - Spring Boot startup from 10 seconds to 2 during development
- Smarter
  - Learn how @EnableAutoConfiguration works
    - Tune or disable to your needs
    - Write your own
- Easier
  - Streamline the configuration of our application
    - Relaxed bindings
    - YAML
    - Eliminate casting using typesafe @ConfigurationProperties
    - Integrate with your IDE
- Cloudier
  - Deploy your Spring Boot apps to the cloud using Docker
  - Monitor your app in the cloud or on your servers using built in Actuators
  - Write your own custom monitoring Actuators
  - Remote debug and development with auto restart

Project Builder

- Spring projects can be build in various ways:
  1. Manual Build
  2. Spring Initializr
  3. Eclipse STS (Spring Tool Suite)
  4. Spring Boot CLI
  5. Maven Build
  6. Gradle Build

Spring Boot CLI

- Use the below link to download CLI
  - https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started-installing-spring-boot.html#getting-started-installing-the-cli

- Execute through CLI
  - spring run hello.groovy
    - Takes default port called 8080
  - spring run hello.groovy -- --server.port=9000
    - If you change port

Sample Code:

```
@RestController
class hello{
    @RequestMapping("/")
    String home() {
        "Hello World!"
    }
}
```

Components

- **Spring Boot CLI**
  - Spring Boot CLI(Command Line Interface) is a Spring Boot software to run and test Spring Boot applications from command prompt.

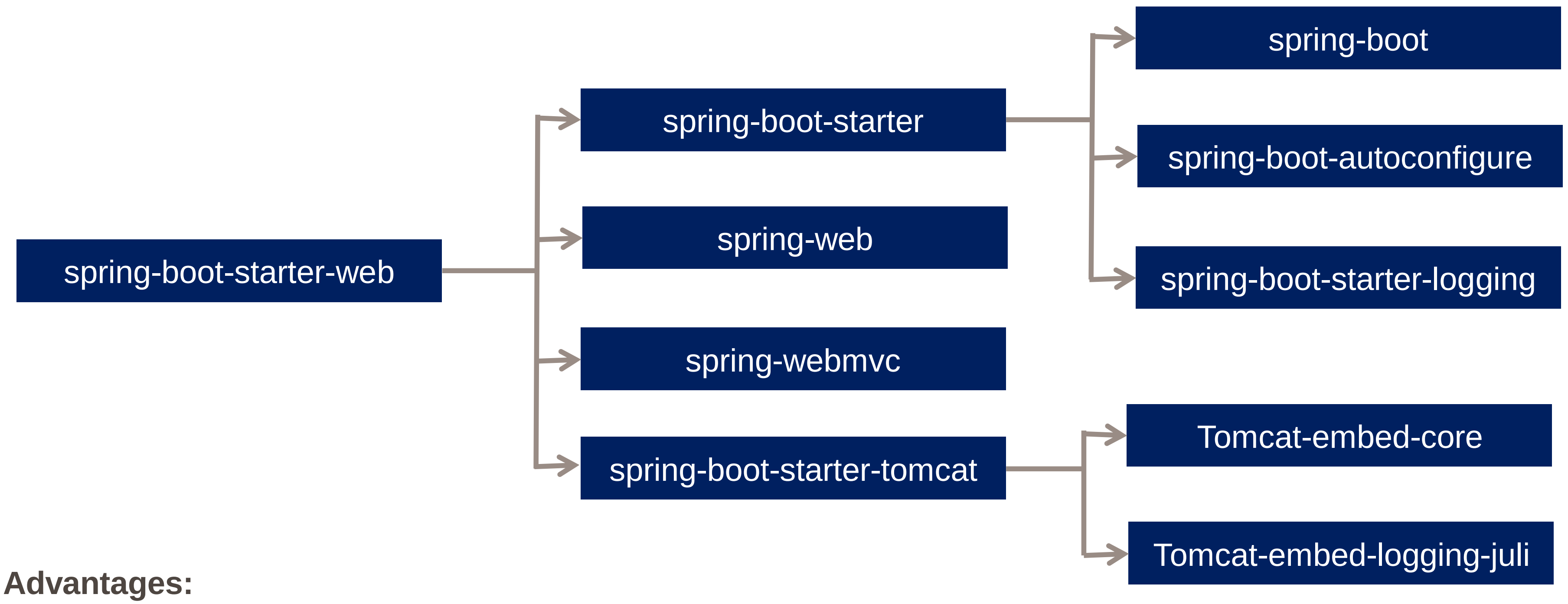
- **spring command example:**
  - spring run HelloWorld.groovy

Project Builder

- **Spring Boot Initilizr With ThirdParty Tools**
  - CURL Tool
  - HTTPie Tool

Components

- **Spring Boot Starter**


spring-boot-starter-web
spring-boot-starter
spring-web
spring-webmvc
spring-boot-starter-tomcat
spring-boot
spring-boot-autoconfigure
spring-boot-starter-logging
Tomcat-embed-core
Tomcat-embed-logging-juli

- **Advantages:**
  - Spring Boot Starter reduces defining many dependencies simplify project build dependencies.
  - Spring Boot Starter simplifies project build dependencies.

Components

- **Spring Boot AutoConfigurator**
  - AutoConfigurator is to reduce the Spring Configuration
  - No need to define single XML configuration and almost no or minimal Annotation configuration.

@SpringBootApplication = @Configuration + @ComponentScan + @EnableAutoConfiguration

```
@Target(value=TYPE)
@Retention(value=RUNTIME)
@Documented
@Inherited
@Configuration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication
```

Components

```
package com.capgemini;
@EnableAutoConfiguration
public class Foo {
...
}
```

Package has significance

Intelligent and seemingly "magical" annotation that enables features and configures functionality

```
package com.capgemini;
@Configuration
@ComponentScan
@EnableAutoConfiguration
public class Foo {
...
}
```

3 very common annotations in Spring Boot apps

Components

```
package com.capgemini;
@SpringBootApplication
public class Foo {
...
}
```

Package has significance

Introduced in Spring Boot 1.2.
Really three annotations in one.

Intelligent Decision Making Based on Conditions

- Presence / Absence of Jars
- Presence / Absence of Beans
- Presence / Absence of Property

Components

- **Externalized Configurations:**
  - Enhanced configuration
    - YAML
    - Typesafe configuration
    - Resolving configuration

YAML

✓Pronounced YAM-EL, rhymes with Camel

✓Data serialization language / format

✓Since 2001

✓Ruby, Python, ElasticSearch, MongoDb

YAML vs Properties

YAML

- Defined spec: http://yaml.org/spec/
- Human readable
- key/value (Map), Lists, and Scalar types
- Used in many languages
- Hierarchical
- Doesn’t work with @PropertySource
- Multiple Spring Profiles in default
- Config

.properties

- java.util.Properties Javadoc is spec
- Human readable
- key/value (Map) and String types
- Used primarily in Java
- Non-hierarchical
- Works with @PropertySource
- One Spring Profile per config

Switch Server – tomcat, jetty and undertow

```
<dependencies>
  <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
    <exclusion>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
    </exclusions>
  </dependency>
  ………..
  </dependencies>
```

Exclude the default tomcat server using <exclusions> in pom.xml

Switch Server – tomcat, jetty and undertow

Include jetty dependency in pom.xml

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-jetty</artifactId>
<scope>provided</scope>
</dependency>
```

This will start jetty server in place of default tomcat server.

Autoconfiguration

Spring Boot looks at

a) Frameworks available on the CLASSPATH
b) Existing configuration for the application.

Based on these, Spring Boot provides basic configuration needed to configure the application with these frameworks. This is called Auto Configuration.

Autoconfiguration

- As soon as we added in Spring Boot Starter Web as a dependency in our project, Spring Boot Autoconfiguration sees that Spring MVC is on the classpath. It autoconfigures dispatcherServlet, a default error page and findbyfirstlast.

- If you add Spring Boot Data JPA Starter, you will 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. All auto configuration logic for mvc, data, jms and other frameworks is present in a single jar.

Autoconfiguration

DataSourceAutoConfiguration:

Typically all Auto Configuration classes look at other classes available in the classpath.

Specific classes are available in the classpath, then configuration for that functionality is enabled through auto configuration.

Annotations like @ConditionalOnClass, @ConditionalOnMissingBean help in providing these features!

@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })

This configuration is enabled only when these classes are available in the classpath.

Autoconfiguration

```
@Configuration
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@EnableConfigurationProperties(DataSourceProperties.class)
@Import({ Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class
})
public class DataSourceAutoConfiguration {
```

Components

- **Spring Boot Actuator**
  - Spring Boot Actuator components gives many features, but two major features are
    - Providing Management EndPoints to Spring Boot Applications.
    - Spring Boot Applications Metrics.

Summary

Overview

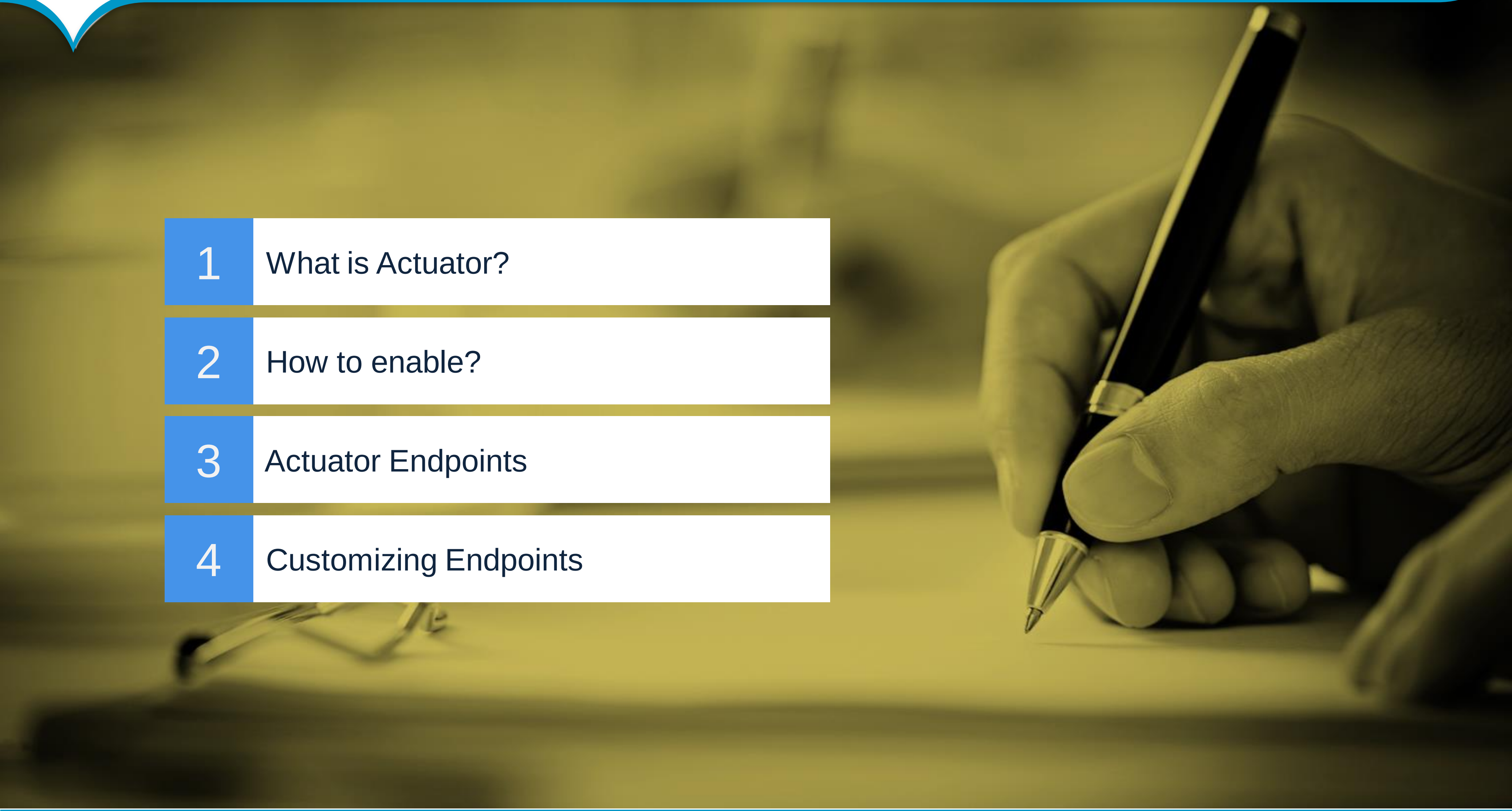
Features

Key Components

CLI

Spring Boot Actuator

Outline



What is Actuator?

- Spring Boot Actuator is a Sub-Project of Spring Boot.
- Spring Boot Actuator includes a number of additional features to help you monitor and manage your application when it's pushed to production.
- You can choose to manage and monitor your application using HTTP Endpoints, with JMX or even by remote shell (SSH or TELNET). Auditing, Health and Metrics gathering can be automatically applied to your application.
- Actuator is supported out of the box within spring boot applications. You just have to add the dependency to enable the actuator. The default configurations are enabled if you are not providing any application specific configurations.
- Main purpose of this feature is to provide various useful metrics about the applications. It is very helpful in the production environment to check the various metrics like **health of your application, configurations, error page, version details, etc.**

How to enable?

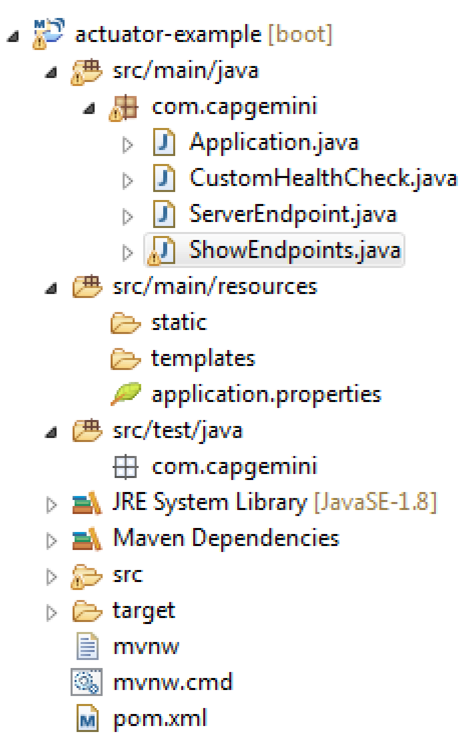
```
<dependency>
 <groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Actuator Endpoints

- HTTP Endpoints are enabled for accessing the various information about your application. List of 15 endpoints that are currently supported by spring boot actuator module is:
- ***actuator*****:** It is the default endpoint to list the available endpoints exposed to the user. It is enabled only when HATEOAS available in your classpath.
- ***autoconfig*** **:** Report on auto-configuration details.
- **beans :** This endpoint lists all the beans loaded by the application.
- ***configprops*** **:** This endpoint shows configuration properties used by your application.
- ***dump*** **:** Performs a thread dump.
- ***env*** **:** Exposes spring's properties from the configurations.
- ***health*** **:** Health of the application.
- ***info*** **:** Displays application information like version, description, etc.
- ***metrics*** **:** Metrics about memory, heap, etc. for the currently running application
- ***mappings*** **:** Displays a list of all @RequestMapping paths.
- ***shutdown*** **:** This endpoint allows to shutdown the application. This is not enabled by default.
- ***trace*** **:** Displays trace information for the current application.
- ***logfile*** **:** Provides access to the configured log files (This feature supported since Spring Boot 1.3.0).
- ***flyway*** **:** This endpoint provides the details of any flyway database migrations have been applied (This feature supported since Spring Boot 1.3.0).
- ***liquibase*** **:** This endpoint provides the details of any liquibase database migrations have been applied (This feature supported since Spring Boot 1.3.0).

Customizing Endpoints

- Just enabling the endpoints may not be sufficient in most of the real time applications. You would like to update application specific configurations.
  - Spring Boot configuration File **application.properties**



```
actuator-example [boot]
  src/main/java
    com.capgemini
      Application.java
      CustomHealthCheck.java
      ServerEndpoint.java
      ShowEndpoints.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
    com.capgemini
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

Customizing Endpoints

- The most common settings are:
  - management.port=8081 – change the port of the actuator endpoints in your server
  - management.address=127.0.0.1 – restrict to run only on localhost
  - management.context-path=/details – change the context path for actuator
  - endpoints.health.enabled=false – enable or disable the endpoints.

Summary

Endpoints

Server

HealthCheckUp

Info

Beans

Spring Boot Auto-configuration

What is Auto-Configuration?

- Spring Boot autoconfiguration represents a way to automatically configure a Spring application based on the dependencies that are present on the classpath.
- Make development faster and easier by eliminating the need for defining certain beans that are included in the auto-configuration classes.

@EnableAutoConfiguration Parameters

The following are the parameters that can be passed inside this annotation:

exclude – Exclude the list of classes from the auto configuration.
excludeNames – Exclude the list of fully qualified class names from the auto configuration. This parameter added since spring boot 1.3.0.

@SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan.

Creating a Custom Auto-Configuration

- **Annotate as @*Configuration***
- **Register that with** *org.springframework.boot.autoconfigure.EnableAutoConfiguration*
  - *resources/META-INF/spring.factories*:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.cap.autoconfiguration.MySQLAutoconfiguration
```

    – *@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)*

- Auto-configuration is designed using classes and beans marked with *@Conditional* annotations so that the auto-configuration or specific parts of it can be replaced.

- *Note that the auto-configuration is only in effect if the auto-configured beans are not defined in the application. If you define your bean, then the default one will be overridden.*

Class Conditions

Class conditions allow us to **specify that a configuration bean will be included if a specified class is present** using the *@ConditionalOnClass* annotation, **or if a class is absent** using the *@ConditionalOnMissingClass* annotation.

- @ConditionalOnClass annotation
- @ConditionalOnMissingClass annotation.

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

Bean Conditions

- If we want to **include a bean only if a specified bean is present or not**, we can use the *@ConditionalOnBean* and *@ConditionalOnMissingBean* annotations.

```
@Bean
@ConditionalOnBean(name = "dataSource")
@ConditionalOnMissingBean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean em
      = new LocalContainerEntityManagerFactoryBean();
    em.setDataSource(dataSource());
    em.setPackagesToScan("com.baeldung.autoconfiguration.example");
    em.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
    if (additionalProperties() != null) {
        em.setJpaProperties(additionalProperties());
    }
    return em;
}
```

```
@Bean
@ConditionalOnMissingBean(type = "JpaTransactionManager")
JpaTransactionManager transactionManager(EntityManagerFactory
entityManagerFactory) {
    JpaTransactionManager transactionManager = new
JpaTransactionManager();
    transactionManager.setEntityManagerFactory(entityManagerFactory);
    return transactionManager;
}
```

Property Condition

- The *@ConditionalOnProperty* annotation is used to **specify if a configuration will be loaded based on the presence and value of a Spring Environment property**.

```
@PropertySource("classpath:mysql.properties")
public class MySQLAutoconfiguration {
    //...
}
```

```
@Bean
@ConditionalOnProperty(
  name = "usemysql",
  havingValue = "local")
@ConditionalOnMissingBean
public DataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();

    dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/myDb?createDatabaseIfNotExist=true");
    dataSource.setUsername("mysqluser");
    dataSource.setPassword("mysqlpass");

    return dataSource;
}
```

Resource Conditions

- Adding the *@ConditionalOnResource* annotation means that the **configuration will only be loaded when a specified resource is present**.

```
@ConditionalOnResource(
  resources = "classpath:mysql.properties")
@Conditional(HibernateCondition.class)
Properties additionalProperties() {
    Properties hibernateProperties = new Properties();

    hibernateProperties.setProperty("hibernate.hbm2ddl.auto",  env.getProperty("mysql-hibernate.hbm2ddl.auto"));
    hibernateProperties.setProperty("hibernate.dialect",  env.getProperty("mysql-hibernate.dialect"));
    hibernateProperties.setProperty("hibernate.show_sql",  env.getProperty("mysql-hibernate.show_sql") != null
      ? env.getProperty("mysql-hibernate.show_sql") : "false");
    return hibernateProperties;
}
```

```
mysql-hibernate.dialect=org.hibernate.dialect.MySQLDialect
mysql-hibernate.show_sql=true
mysql-hibernate.hbm2ddl.auto=create-drop
```

Custom Conditions

- **Define custom conditions by extending the *SpringBootCondition* class and overriding the *getMatchOutcome()* method**.

```
static class HibernateCondition extends SpringBootCondition {

    private static String[] CLASS_NAMES
      = { "org.hibernate.ejb.HibernateEntityManager",   "org.hibernate.jpa.HibernateEntityManager" };

    @Override
    public ConditionOutcome getMatchOutcome(ConditionContext context,
      AnnotatedTypeMetadata metadata) {

        ConditionMessage.Builder message  = ConditionMessage.forCondition("Hibernate");
        return Arrays.stream(CLASS_NAMES)
          .filter(className -> ClassUtils.isPresent(className, context.getClassLoader()))
          .map(className -> ConditionOutcome
            .match(message.found("class")
            .items(Style.NORMAL, className)))
          .findAny()
          .orElseGet(() -> ConditionOutcome
            .noMatch(message.didNotFind("class", "classes")
            .items(Style.NORMAL, Arrays.asList(CLASS_NAMES))));
    }
}
```

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

Example

```
import org.slf4j.Logger; import org.slf4j.LoggerFactory;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
@ConditionalOnClass({ String.class })
public class ConfigureDefaults {
    Logger logger = LoggerFactory.getLogger(ConfigureDefaults.class);

@Bean
public String cacheManager() {
    logger.info("Configure Defaults");
    return new String("test");
}
}
```

Create spring.factories file as below and put it under the META-INF folder:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
Org.cap.spring.data.autoconfigure.ConfigureDefaults
```

Application Conditions

- **Specify that the configuration can be loaded only inside/outside a web context**, by adding the
- *@ConditionalOnWebApplication*
- *@ConditionalOnNotWebApplication* annotation.

Testing the Auto-Configuration

```
@Entity
public class MyUser {
    @Id
    private String email;

    // standard constructor, getters, setters
}
```

```
public interface MyUserRepository
  extends JpaRepository<MyUser, String> { }
```

- To enable auto-configuration, we can use one of the *@SpringBootApplication* or *@EnableAutoConfiguration* annotations:

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

Testing the Auto-Configuration

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(
  classes = AutoconfigurationApplication.class)
@EnableJpaRepositories(
  basePackages = { "com.baeldung.autoconfiguration.example" })
public class AutoconfigurationTest {

    @Autowired
    private MyUserRepository userRepository;

    @Test
    public void whenSaveUser_thenOk() {
        MyUser user = new MyUser("user@email.com");
        userRepository.save(user);
    }
}
```

Disabling Auto-Configuration Classes

- If we wanted to **exclude the auto-configuration from being loaded**, we could add the *@EnableAutoConfiguration* annotation with *exclude* or *excludeName* attribute to a configuration class:

```
@Configuration
@EnableAutoConfiguration(
  exclude={MySQLAutoconfiguration.class})
public class AutoconfigurationApplication {
    //...
}
```

- Another option to disable specific auto-configurations is by setting the *spring.autoconfigure.exclude* property:

```
spring.autoconfigure.exclude=com.baeldung.autoconfiguration.MySQLAutoconfiguration
```

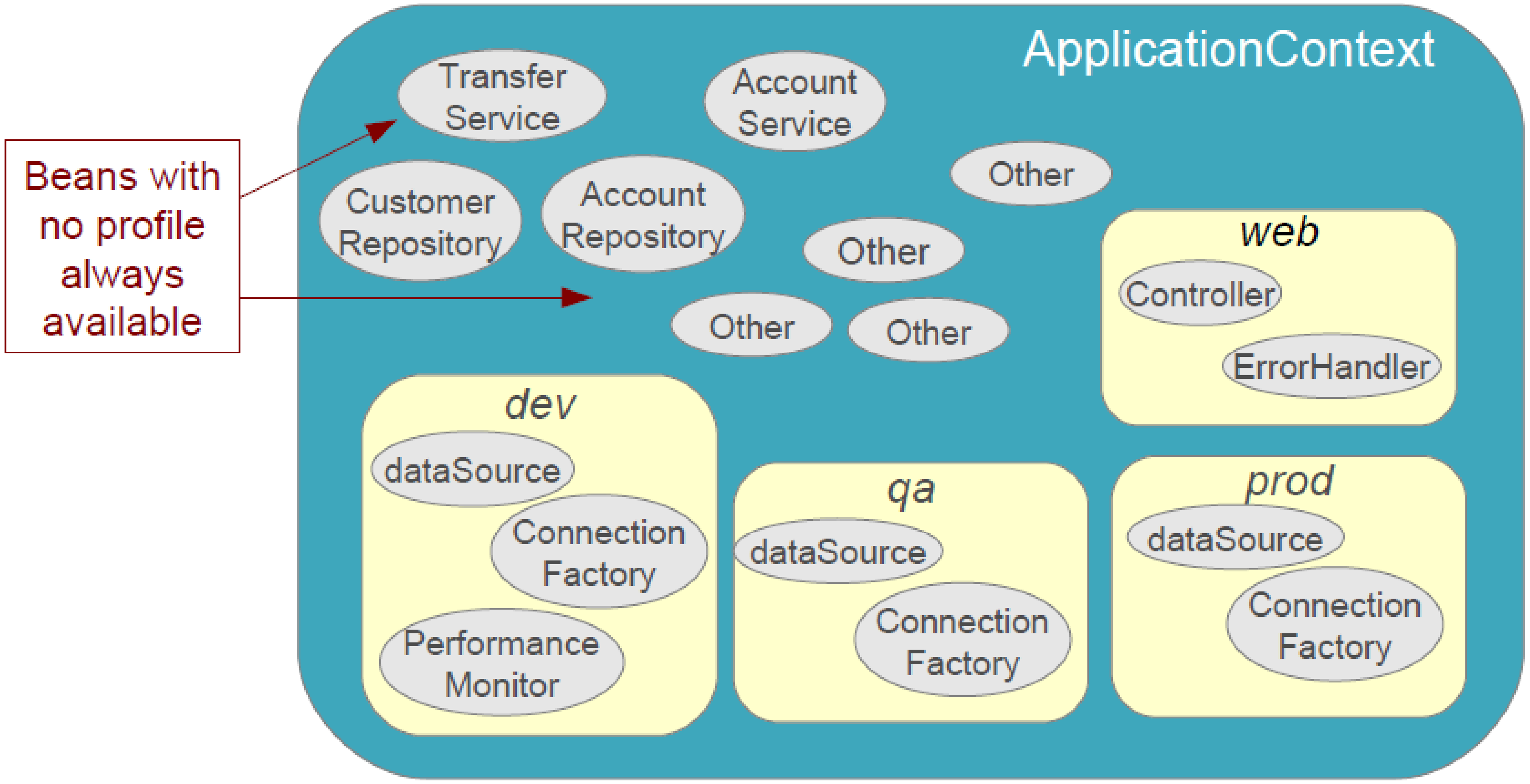
SPRING PROFILES

Spring Profiles

- Part of Spring Framework since 3.0
  - Allow multiple different configurations
  - Select the ones you want by selecting one or more profiles
  - integrated into Spring Testing framework also

- Beans can be grouped into Profiles
  - Profiles can represent purpose: "web", "offline"
  - Or environment: "dev","qa","uat","prod","cloud"
  - Or implementation: "jdbc","jpa"
  - Beans included /excluded based on profile membership

Example Profiles



Defining profiles

- Add @Profile annotation to component or configuration
- Or qualify <beans> in XML

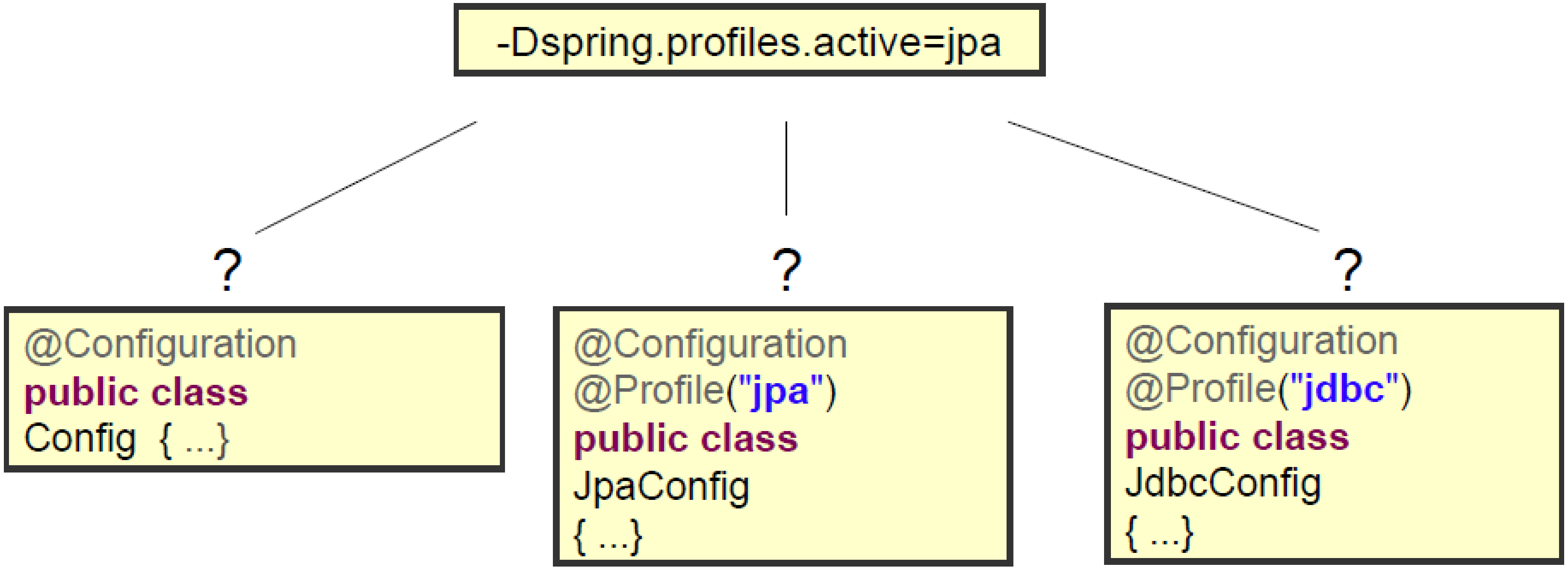
```
@Configuration
@Profile("dev")
public class DevConfig {

   @Bean
   public DataSource dataSource() {
      ...
   }
}
```

```
@Repository
@Profile("jdbc")
public class
JdbcAccountRepository
  { ...}
```

```
<beans xmlns=...>
   <!-- Available to all profiles →
   <bean id="transferService" … />
   ...
   <beans profile="jdbc">
      <bean id="dataSource" … />
   </beans>
   <beans profile="jpa"> ... </beans>
</beans>
```

QUIZ

- Which of the following is/are selected?



```
-Dspring.profiles.active=jpa
```

```
@Configuration
public class
Config  { ...}
```

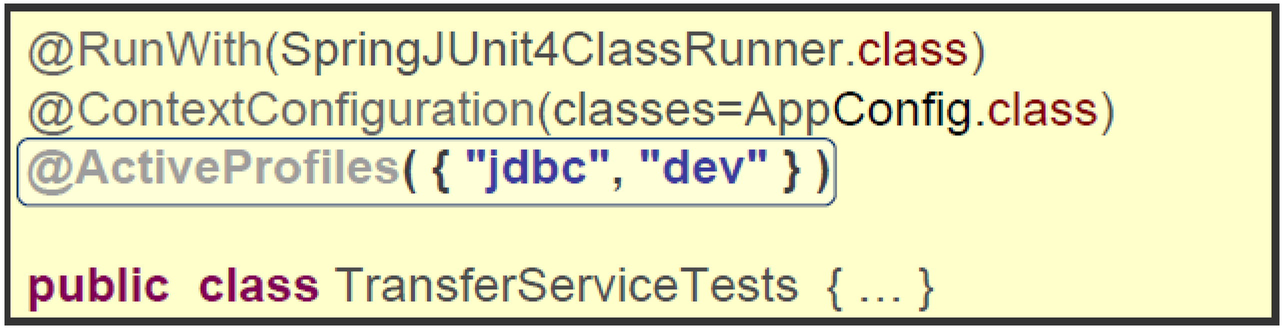
```
@Configuration
@Profile("jpa")
public class
JpaConfig
{ ...}
```

```
@Configuration
@Profile("jdbc")
public class
JdbcConfig
{ ...}
```

Activating Profiles For a Test

- @ActiveProfiles inside Spring-driven test class
  - Define one or more profiles
  - Beans associated with that profile are instantiated
  - Also beans not associated with any profile

- Example: Two profiles activated -jdbc and dev



```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=AppConfig.class)
@ActiveProfiles( { "jdbc", "dev" } )

public  class TransferServiceTests  { … }
```

Active Profiles

- Profiles may be activated at execution-time
  - System property

```
-Dspring.profiles.active=dev,jpa
```

- Profiles activated via Cloud Foundry environment variable
  - CLI or manifest

```
cf  set-env <app>  spring.profiles.active  dev
```

- Java buildpac
  - You can a

Creating war in Spring Boot

Create WAR File in Spring Boot

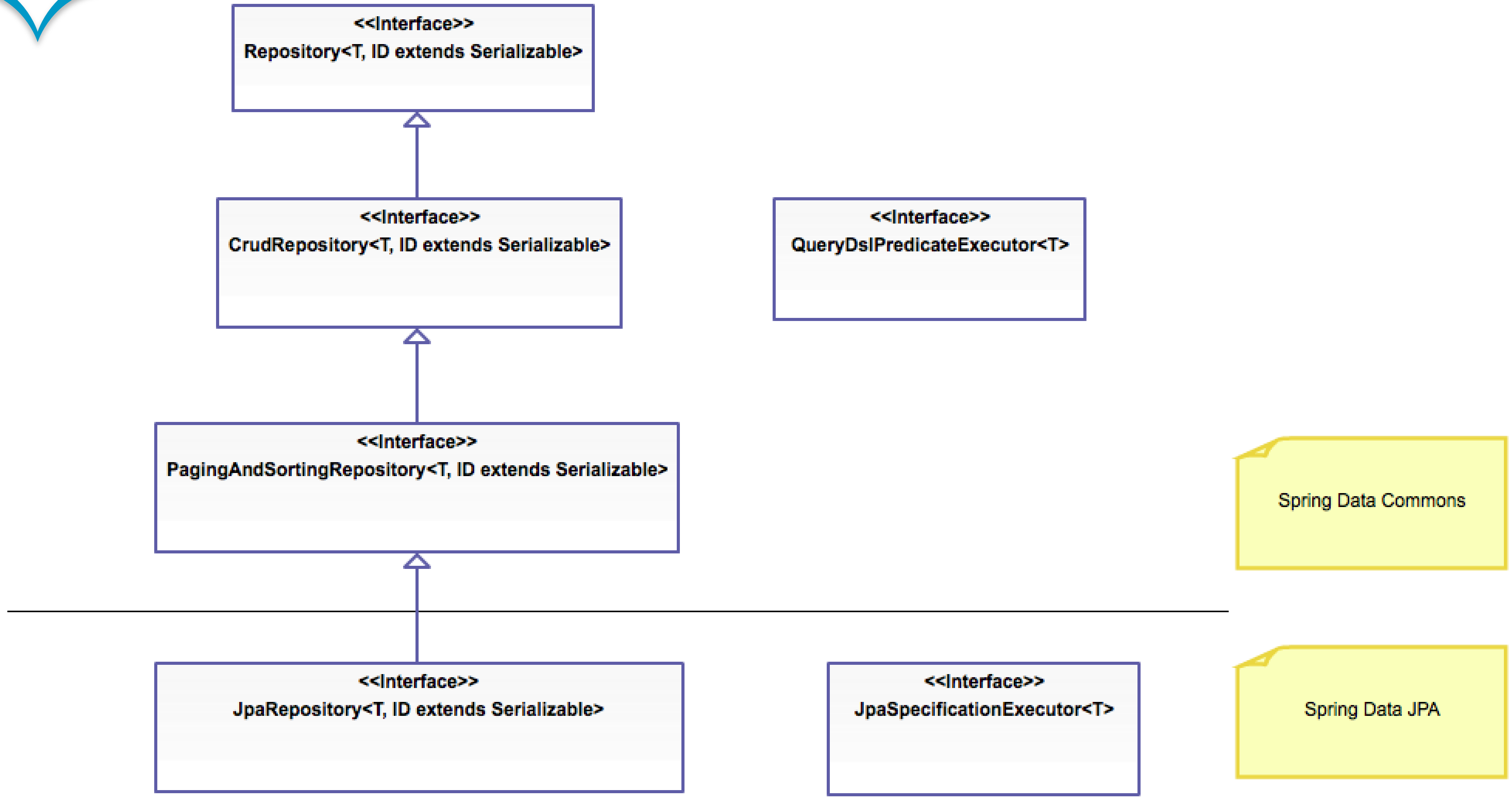
- **Extending Main Class**
- **Overriding configure method**

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer{

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

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return builder.sources(Application.class);
    }
}
```

- **Configure Packaging to WAR**
  - **<packaging>war</packaging>** in your pom.xml
  - Then build your application using **mvn package.**

Spring Data JPA

Spring Data API


<<Interface>>
Repository<T, ID extends Serializable>
<<Interface>>
CrudRepository<T, ID extends Serializable>
<<Interface>>
QueryDslPredicateExecutor<T>
<<Interface>>
PagingAndSortingRepository<T, ID extends Serializable>
Spring Data Commons
<<Interface>>
JpaRepository<T, ID extends Serializable>
<<Interface>>
JpaSpecificationExecutor<T>
Spring Data JPA

Spring Data

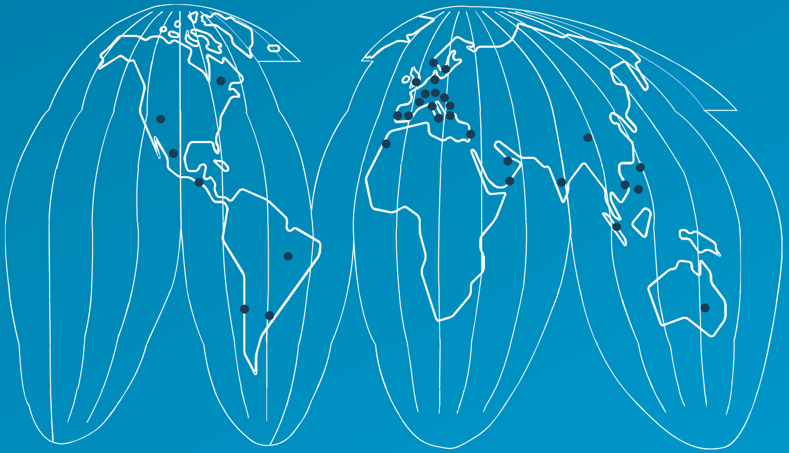
- The Spring Data generated DAO – No More DAO Implementations
- Custom Access Method and Queries
- Automatic Custom Queries
- Manual Custom Queries
- Transaction Configuration

People matter, results count.

About Capgemini

With more than 145,000 people in 40 countries, Capgemini is one of the world's foremost providers of consulting, technology and outsourcing services. The Group reported 2014 global revenues of EUR 10.5 billion.
Together with its clients, Capgemini creates and delivers business and technology solutions that fit their needs and drive the results they want. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.

Rightshore® is a trademark belonging to Capgemini



www.capgemini.com