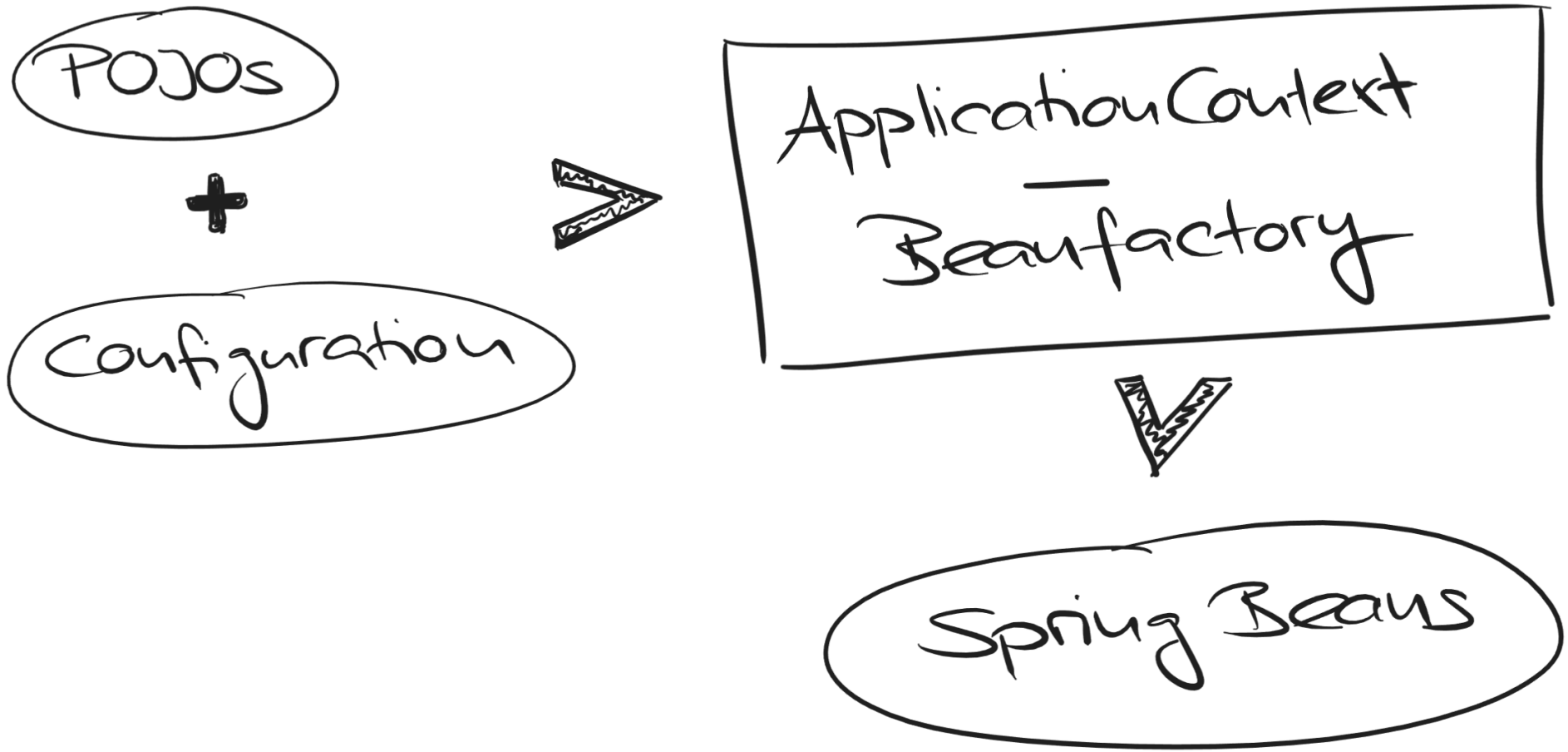
Spring Boot Configuration

Spring Configuration

ApplicationContext


POJOs
+
Configuration
ApplicationContext
=
BeanFactory
Spring Beans

Creating an ApplicationContext

Plain Java

```
ApplicationContext context =
    new AnnotationConfigApplicationContext(MyApplicationConfig.class);
```

or

```
ApplicationContext context =
    new AnnotationConfigWebApplicationContext(MyApplicationConfig.class);
```

Programmatically

```
context = new AnnotationConfigApplicationContext();
context.register(MyApplicationConfig.class);

// do more stuff

context.refresh();
```

Component Scanning

```
@Configuration
@ComponentScan(basePackages="com.fortytwotalents.app")
public class MyApplicationConfig { … }
```

or

```
context = new AnnotationConfigApplicationContext();
context.scan("com.fortytwotalents.app");
context.refresh();
```

Spring Boot

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

Spring Test Context Framework

```
// Junit 5
 @ExtendWith(SpringExtension.class)
 @ContextConfiguration(classes=MyApplicationConfig.class)
 public class MyApplicationTest {}
```

or

```
// Junit 5
 @SpringJunitConfig(MyApplicationConfig.class)
 public class MyApplicationTest {}
```

Spring Boot Test

```
@SpringBootTest
public class MyApplicationTest {}
```

or

```
@SpringBootTest(classes=MyApplicationConfig.class)
public class MyApplicationTest {}
```

What is a Spring Bean?

Attributes of a Spring Bean

- Name / ID
- Type
- Dependencies
- Target Class / Factory Method
- Initializers / Destructors

Bean Scopes

- Singleton
- Prototype*
- Request
- Session
- Custom

} Clean-up
⇒ @PreDestroy

* no clean-up

Defining Spring Beans

Java Configuration

Instantiating a Spring Bean

```
@Configuration
public class MyApplicationConfig {

    @Bean
    public BlogPostService blogPostService() {
        return new BlogPostServiceImpl(postRepository());
    }

    @Bean(name = "postRepository", initMethod="load")
    public PostRepository postRepository() {
        return new JdbcPostRepository(dataSource());
    }

    @Bean
    public DataSource dataSource() {
        SimpleDataSource dataSource = new SimpleDataSource();
        dataSource.setUrl("jdbc:mysql://localhost/mydb");
        dataSource.setUsername("root");
        dataSource.setPassword("secret");
        return dataSource;
    }
}
```

Dependency Injection

```java
@Configuration
public class MyApplicationConfig {

    @Bean
    public BlogPostService blogPostService(PostRepository postRepository) {
        return new BlogPostServiceImpl(postRepository);
    }

    @Bean(name = "postRepository", initMethod="load")
    public PostRepository postRepository(DataSource dataSource) {
        return new JdbcPostRepository(dataSource);
    }
}
```

```java
@Configuration
public class InfrastructureConfig {
    @Bean
    public DataSource dataSource() {
        SimpleDataSource dataSource = new SimpleDataSource();
        // …
        return dataSource;
    }
}
```

Importing Configurations

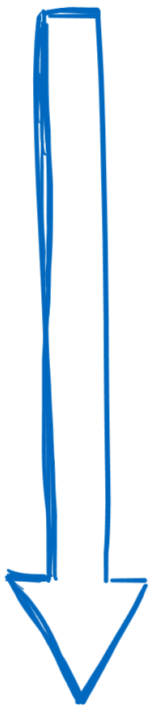
```
@Configuration
@Import({WebConfig.class, SecurityConfig.class})
public class MyApplicationConfig { … }

@Configuration
public class WebConfig { … }

@Configuration
public class SecurityConfig { … }
```

or

```
@Configuration
@ImportResource("xml-legacy-configuration.xml")
public class MyApplicationConfig { … }
```

Environment Abstraction

Environment Abstraction



- ...
- Java System Properties
  ⇒ java -Dserver.port=9999 -jar my.jar
- OS Environment
  ⇒ JAVA_HOME=...
- Java Property Files
  ⇒ @PropertySource

PropertySources

```
@Configuration
@PropertySource("classpath:jdbc.properties")
public class MyApplicationConfig {

    @Value("${main.jdbc.url}")
    private String jdbcUrl;

    @Bean
    public DataSource dataSource() {
        return new SimpleDataSource(jdbcUrl);
    }

    @Bean
    public DataSource customerDataSource(@Value("${customer.jdbc.url}") String jdbcUrl) {
        return new SimpleDataSource(jdbcUrl);
    }
}
```

```
# my-application.properties
main.jdbc.url=jdbc:mysql://localhost/mydb
customer.jdbc.url=jdbc:mysql://localhost/mydb
```

Defining Spring Beans

Annoation Based Configuration

Annotation Based Configuration

- Rapid prototyping through component scanning at runtime
- An alternative to XML and Java Config setups
  - Less verbose, no explicit bean definitions like Java Configuration
  - Nice for your code, is not used within the framework
  - Application startup takes more time and depends on package restrictions

Stereotype Annotations

Stereotype Annotations

@Controller
@RestController

@Service

@Repository

@Configuration

@...

@Component

Component Scanning

Basic Configuration

```
@Component
public class BlogPostServiceImpl implements BlogPostService {
}

@Configuration
@ComponentScan(basePackages="com.fortytwotalents.app")
public class MyApplicationConfig {
}
```

or

```
@Component("blogPostService")
public class BlogPostServiceImpl implements BlogPostService {
}
```

Value Injection

```
@Component
public class BlogPostServiceImpl implements BlogPostService {

    @Value("${provider.api.url}")
    private String apiUrl;

    @Value("${provider.api.username}")
    private String apiUsername;

    @Value("${provider.api.password}")
    private String apiPassword;

    @Value("${provider.api.retries:10}")
    private int numOfRetries;
}
```

```
# my-application.properties
provider.api.username=superuser
provider.api.password=secret
```

Constructor Injection

```
@Service
public class BlogPostServiceImpl implements BlogPostService {

    private final PostRepository postRepository;

    @Autowired // Not needed if it is the only constructor
    public BlogPostServiceImpl(PostRepository postRepository) {
      this.postRepository = postRepository;
    }

    …

}
```

Field injection

```
@Service
public class BlogPostServiceImpl implements BlogPostService {

    @Autowired
    private PostRepository postRepository;

    …
}
```

See also: https://odrotbohm.de/2013/11/why-field-injection-is-evil/

Setter Injection

```
@Service
public class BlogPostServiceImpl implements BlogPostService {

    private PostRepository postRepository;

    @PostConstruct
    public void init() { … }

    @Autowired(required = false)
    public void setPostRepository(PostRepository postRepository) {
        this.postRepository = postRepository;
    }

    @Autowired
    public void load(PostRepository postRepository, TimelineService timelineService) { … }

    …
}
```

Dependency Injection By Name

```
@Service
public class BlogPostRepository implements PostRepository {

    @Autowired
    @Qualifier("mainDataSource")
    private DataSource mainDS;

    @Autowired
    @Qualifier("accountDataSource")
    private DataSource accountDS;

}
```

Dependency Injection By Name Fallback

```
@Service
public class BlogPostRepository implements PostRepository {

    @Autowired
    private DataSource mainDataSource;

    @Autowired
    private DataSource accountDataSource;

}
```

Basic Annotations

Spring and Standard Annotations

- Spring Annotations
  - @Autowired / @Value
  - @Qualifier
  - @Required
- JSR 250
  - @Resource
  - @PostConstruct
  - @PreDestroy

Basic Annotations

Even more

- Context
  - @Scope
  - @Bean
  - @DependsOn
  - @Lazy
- Transactional
  - @Transactional
- ...

Spring Boot Auto Configuration

Spring Boot Auto Configuration

- Auto-configuration of Spring Beans
- Creating Spring Beans by convention
  - Checking the classpath for dependencies
  - Reading properties from `application.properties` or `application.yml`
  - Creating beans based on other beans
  - ...

Spring Boot Starters

Spring Boot Starter POM

- Standard Maven POMs
- Contains transient dependencies
- Parent POM optional / import via `<DependencyManagement>`
- Available for web, batch, integration, data, security, aop, jdbc, ...

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

See also: https://repo1.maven.org/maven2/org/springframework/boot/spring-boot-dependencies/3.0.4/spring-boot-dependencies-3.0.4.pom

AutoConfiguration

@SpringBootApplication

- Creates a running ApplicationContext

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

- Or one without Spring Boot Banner

```
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication context =
            new SpringApplication(DemoApplication.class);
        context.setBannerMode(Mode.OFF);
        context.run(args); }}
```

@EnableAutoConfiguration

```
@SpringBootApplication
public class DemoApplication {}
```

Resolves to

```
@SpringBootConfiguration
@ComponentScan
@EnableAutoConfiguration
public class DemoApplication {}
```

Resolves to

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
public class DemoApplication {}
```

`@EnableAutoConfiguration`

- Tries to auto-configure your application
- Does not do anything if you define your own beans
- Regular `@Configuration` classes
- Usually done with `@ConditionalOnClass` and `@ConditionalOnMissingBean` or `@ConditionalOnProperty` and many more …

AutoConfiguration

Currently Available AutoConfigured Behavior

- Developer Tools
- Web
- Template Engines
- Security
- SQL
- NoSQL
- Messaging
- I/O
- OPS
- Observability
- Testing
- Spring Cloud
- Cloud Platforms

Import Candidates in META-INF/spring/

org.springframework.boot.autoconfigure.AutoConfiguration.imports

```
...
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration
...
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration
org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration
org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration
org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration
org.springframework.boot.autoconfigure.neo4j.Neo4jAutoConfiguration
org.springframework.boot.autoconfigure.netty.NettyAutoConfiguration
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration
org.springframework.boot.autoconfigure.quartz.QuartzAutoConfiguration
org.springframework.boot.autoconfigure.r2dbc.R2dbcAutoConfiguration
org.springframework.boot.autoconfigure.r2dbc.R2dbcTransactionManagerAutoConfiguration
```

Auto Configuration

```
@AutoConfiguration(before = SqlInitializationAutoConfiguration.class)
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@ConditionalOnMissingBean(type = "io.r2dbc.spi.ConnectionFactory")
@EnableConfigurationProperties(DataSourceProperties.class)
@Import(DataSourcePoolMetadataProvidersConfiguration.class)
public class DataSourceAutoConfiguration {

  @Configuration(proxyBeanMethods = false)
  @Conditional(EmbeddedDatabaseCondition.class)
  @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
  @Import(EmbeddedDataSourceConfiguration.class)
  protected static class EmbeddedDatabaseConfiguration {
  }

  @Configuration(proxyBeanMethods = false)
  @Conditional(PooledDataSourceCondition.class)
  @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
  @Import({ DataSourceConfiguration.Hikari.class, DataSourceConfiguration.Tomcat.class,
    DataSourceConfiguration.Dbcp2.class, DataSourceConfiguration.OracleUcp.class,
    DataSourceConfiguration.Generic.class, DataSourceJmxConfiguration.class })
  protected static class PooledDataSourceConfiguration {
  }
```

Conditions

- Configuration with `@Conditional`
  - OnClass / OnMissingClass
  - OnBean / OnMissingBean
  - OnProperty
  - OnResource
  - OnExpression
  - OnJava
  - OnJndi
  - OnSingleCandidate
  - OnWarDeployment
  - OnWebApplication / OnNotWebApplication
  - OnCloudPlatform

Property Sources

Environment and Profile

- Every `ApplicationContext` has an `Environment`
- Abstraction for key/value pairs from multiple sources
- Used to manage `@Profile` switching
- `@Value` reads property from Environment
- Always available: System properties and OS ENV variables
- Spring Boot extends the Environment with many other PropertySources

See also: https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-external-config

Externalizing Configuration to Properties Files

- Put `application.properties` in one of the following locations:
  - A `/config` sub-directory of the current directory
  - The current directory
  - classpath `/config` package
  - The root classpath
- Properties can be overridden
  - Command line arg > file > classpath
  - Locations higher in the list override lower items

Configuration Properties

Binding Properties To Beans

```
@Component
public class MyProperties {

    @Value("${private.location}")
    private Resource location;

    @Value("${private.skip:true}")
    private boolean skip;

    // … getters and setters
}
```

```
# application.properties
private.location = classpath:mine.xml
private.skip = false
```

Binding to Components (only Spring Boot!)

```
@Component
@ConfigurationProperties(prefix="private")
public class MyProperties {
    private Resource location;
    private boolean skip = true;
    // … getters and setters
}
```

```
# application.properties
private.location = classpath:mine.xml
private.skip = false
```

Enabling Configurationtion Properties

```
@Configuration
@EnableConfigurationProperties(MyProperties.class)
public class MyApplicationConfig { … }
```

```
@Configuration
@ConfigurationPropertiesScan(basePackages="…")
public class MyApplicationConfig { … }
```

```
@Configuration
@ComponentScan(basePackages="")
@EnableConfigurationProperties
public class MyApplicationConfig { … }
```

Property Binding on Java Config Beans

```
@Configuration
public class MyApplicationConfig {

    @Bean
    @ConfigurationProperties(prefix="spring.datasource")
    public DataSource dataSource() {
        return new HikariDataSource();
    }
}
```

Validating Configurationtion Properties

```
@Component
@ConfigurationProperties
@Validated
public class GlobalProperties {

    @Max(5)
    @Min(0)
    private int threadPoolSize;

    @NotEmpty
    private String email;
    //getters and setters
}
```

Configuration Name & Location

- `spring.config.name` - default `application`, can be comma-separated list
- `spring.config.location` - a Resource path, ends with `/` to define a directory, otherwise overrides

```
java -jar myApp.jar --spring.config.name=production
java -jar myApp.jar --spring.config.location=classpath:/cfg/
java -jar myApp.jar --spring.config.location=classpath:/cfg.yml
java -jar myApp.jar --spring.config.location=optional:/etc/config/application.properties
```

- Import `application.properties` or `application.yaml` as they are discovered


```
spring.config.import=optional:file:./dev.properties
```

Disabling AutoConfiguration

Disabling AutoConfiguration

- Via Annotation

```
@Configuration
@EnableAutoConfiguration(excludeName="…", exclude=DataSourceAutoConfiguration.class})
public class AppConfiguration { … }
```

- Or via property

```
spring.autoconfigure.exclude = org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
```