# System Design Update: Integrating Spring Boot and Maven into the Java/Playwright Automation Tool

## I. Introduction

This report outlines an updated system design for the Java/Playwright web form automation tool. The primary objective is to modernize the application's structure and build process by incorporating Apache Maven as the build system and leveraging the Spring Boot framework for managing the application lifecycle, configuration, and dependencies. This transition aims to enhance maintainability, testability, and deployment simplicity by producing a single, self-contained executable JAR file. The following sections detail the refined application structure, component management, configuration strategy, logging integration, Playwright resource management, Maven build configuration, command-line argument handling, and the resulting component interactions.

## II. Refined Application Structure with Spring Boot

The core application structure will be refactored to align with Spring Boot conventions, moving away from a potentially manual setup towards a managed application context.

**(a) Main Application Class (@SpringBootApplication)**

The entry point of the application will be a class annotated with @SpringBootApplication. This annotation is a convenience annotation that combines @Configuration, @EnableAutoConfiguration, and @ComponentScan. It marks the class as a primary Spring configuration class and enables Spring Boot's auto-configuration mechanism and component scanning.

package com.yourcompany.automation;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication // Enables Auto-config, Component Scan, and marks as Config class  
public class AutomationToolApplication {  
  
 public static void main(String args) {  
 // Launches the Spring Boot application  
 SpringApplication.run(AutomationToolApplication.class, args);  
 }  
  
}

**(b) Implementing Application Logic with Runners (CommandLineRunner/ApplicationRunner)**

The previous CommandLineApp logic, which likely contained the main execution flow initiated directly from the main method, will be migrated into a Spring bean that implements either the CommandLineRunner or ApplicationRunner interface. These interfaces provide a run method that Spring Boot executes automatically *after* the application context is fully loaded and initialized, but *before* the application finishes starting (unless it's a web application waiting for requests).

* **CommandLineRunner:** Provides void run(String... args) method, receiving raw command-line arguments as a string array.
* **ApplicationRunner:** Provides void run(ApplicationArguments args) method, receiving a parsed ApplicationArguments object which offers more structured access to both option (--name=value) and non-option arguments.

**Recommendation:** Use ApplicationRunner as it provides cleaner access to arguments, especially non-option arguments like the input file path required for this tool.

The chosen runner bean (e.g., AutomationRunner) will be annotated with @Component (or a specialization like @Service) to be managed by Spring. Dependencies required for the automation logic (like the Orchestrator) will be injected via the constructor. The run method within this runner will serve as the main execution block, parsing necessary arguments (like the input file path) and initiating the automation process by calling the Orchestrator.

package com.yourcompany.automation.runner;  
  
import com.yourcompany.automation.orchestrator.Orchestrator;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.boot.ApplicationArguments;  
import org.springframework.boot.ApplicationRunner; // Using ApplicationRunner  
import org.springframework.stereotype.Component;  
  
import java.util.List;  
  
@Component // Marks this as a Spring-managed bean  
public class AutomationRunner implements ApplicationRunner { // Implements ApplicationRunner  
  
 private static final Logger log = LoggerFactory.getLogger(AutomationRunner.class);  
  
 private final Orchestrator orchestrator; // Dependency injected via constructor  
  
 public AutomationRunner(Orchestrator orchestrator) {  
 this.orchestrator = orchestrator;  
 }  
  
 @Override  
 public void run(ApplicationArguments args) throws Exception {  
 log.info("Automation Tool starting execution via ApplicationRunner...");  
  
 // Access non-option arguments (e.g., input file path)  
 List<String> nonOptionArgs = args.getNonOptionArgs(); // [span\_253](start\_span)[span\_253](end\_span)  
 if (nonOptionArgs.isEmpty() |  
| nonOptionArgs.get(0) == null |  
| nonOptionArgs.get(0).isBlank()) {  
 log.error("FATAL: Input data file path is required as a command-line argument.");  
 // Consider throwing an exception or using ExitCodeGenerator for cleaner exit  
 System.exit(1);  
 return; // Exit if file path is missing  
 }  
 String inputFilePath = nonOptionArgs.get(0);  
 log.info("Using input data file: {}", inputFilePath);  
  
 // Add logic to handle potential option arguments if needed  
 // e.g., args.containsOption("debug"), args.getOptionValues("output-dir")  
  
 try {  
 // Initiate the main automation logic  
 orchestrator.executeAutomation(inputFilePath);  
 log.info("Automation Tool execution finished successfully.");  
 } catch (Exception e) {  
 log.error("Automation Tool execution failed.", e);  
 // Consider more specific exception handling or exit codes  
 System.exit(1);  
 }  
 }  
}

This runner acts as the application's main operational entry point after Spring Boot has completed its setup.

## III. Component Implementation as Spring Beans

All core components of the automation tool will be defined as Spring beans, managed by the Spring Inversion of Control (IoC) container. This promotes loose coupling and simplifies dependency management.

**(a) Defining Components as Beans**

Components such as DataReader (with implementations like ExcelDataReader), WebAutomationEngine, Orchestrator, ReportGenerator (with specific implementations), and the configuration properties class (e.g., AppConfigProperties) should be annotated with Spring stereotype annotations. Common choices include:

* @Component: Generic stereotype for any Spring-managed component.
* @Service: Indicates a service layer component, often holding business logic (suitable for Orchestrator, WebAutomationEngine).
* @Repository: Indicates a data access component (potentially applicable to DataReader implementations if they involve more complex data access logic, though @Component or @Service might suffice here).
* @Configuration: Used for classes defining beans via @Bean methods or for @ConfigurationProperties classes (as discussed later).

Example:

package com.yourcompany.automation.orchestrator;  
  
import com.yourcompany.automation.config.AppConfigProperties;  
import com.yourcompany.automation.data.DataReader;  
import com.yourcompany.automation.engine.WebAutomationEngine;  
import com.yourcompany.automation.report.ReportGenerator;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.stereotype.Service; // Use @Service for business logic layer  
  
import java.util.List;  
import java.util.Map;  
  
@Service // Marks Orchestrator as a service bean  
public class Orchestrator {  
 //... dependencies and logic...  
}  
  
package com.yourcompany.automation.data;  
  
import org.springframework.stereotype.Component; // Or @Repository if more data-access specific  
  
@Component // Example implementation bean  
public class ExcelDataReader implements DataReader {  
 //... implementation...  
}  
  
// Similar annotations for WebAutomationEngine, ReportGenerator implementation, etc.

**(b) Dependency Injection (DI)**

Dependencies between these beans will be managed by Spring's DI container. The preferred method is **constructor injection**. Spring automatically resolves and injects the required dependencies when creating the bean instance.

* **Mechanism:** Define dependencies as parameters in the constructor of the dependent bean. Spring's container will find the required bean instances (e.g., DataReader, WebAutomationEngine, ReportGenerator, AppConfigProperties) and pass them to the Orchestrator's constructor. The @Autowired annotation on the constructor is implicit and generally not required if the class has only one constructor.
* **Benefits:** Makes dependencies explicit, improves testability (dependencies can be easily mocked), and encourages immutable design patterns.

Example (Orchestrator with constructor injection):

package com.yourcompany.automation.orchestrator;  
  
import com.yourcompany.automation.config.AppConfigProperties;  
import com.yourcompany.automation.data.DataReader;  
import com.yourcompany.automation.engine.WebAutomationEngine;  
import com.yourcompany.automation.report.ReportGenerator;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.stereotype.Service; // Or @Component  
  
import java.util.List;  
import java.util.Map;  
  
@Service  
public class Orchestrator {  
  
 private static final Logger log = LoggerFactory.getLogger(Orchestrator.class);  
  
 private final DataReader dataReader;  
 private final WebAutomationEngine webAutomationEngine;  
 private final ReportGenerator reportGenerator;  
 private final AppConfigProperties configProperties; // Inject configuration properties bean  
  
 // Dependencies injected via constructor  
 public Orchestrator(DataReader dataReader,  
 WebAutomationEngine webAutomationEngine,  
 ReportGenerator reportGenerator,  
 AppConfigProperties configProperties) {  
 this.dataReader = dataReader;  
 this.webAutomationEngine = webAutomationEngine;  
 this.reportGenerator = reportGenerator;  
 this.configProperties = configProperties;  
 }  
  
 public void executeAutomation(String inputFilePath) {  
 log.info("Starting orchestration process for file: {}", inputFilePath);  
 log.info("Target URL from config: {}", configProperties.getTargetUrl()); // Example using config  
  
 try {  
 // 1. Read Data  
 List<Map<String, String>> inputData = dataReader.readData(inputFilePath);  
 log.info("Read {} records from input file.", inputData.size());  
  
 // 2. Process Forms via Web Automation  
 Map<String, String> results = webAutomationEngine.processForms(inputData);  
 log.info("Web automation processing completed.");  
  
 // 3. Generate Report  
 reportGenerator.generateReport(results, inputData);  
 log.info("Report generated successfully.");  
  
 } catch (Exception e) {  
 log.error("Error during orchestration", e);  
 // Handle exception (e.g., generate error report, re-throw)  
 throw new RuntimeException("Orchestration failed", e);  
 }  
 }  
}

This declarative approach, where components and their dependencies are defined and automatically wired by the framework, significantly simplifies the application's setup and reduces boilerplate code compared to manual instantiation and wiring.

## IV. Spring Boot Configuration Management

The custom ConfigManager component will be replaced entirely by Spring Boot's robust and standardized externalized configuration mechanism.

**(a) Leveraging Spring Boot Configuration**

Spring Boot provides built-in support for loading configuration from various sources, including properties files, YAML files, environment variables, and command-line arguments, eliminating the need for a custom configuration loading solution.

**(b) Configuration File (application.properties or application.yml)**

Configuration settings should be stored in application.properties or application.yml located in the src/main/resources directory. YAML (.yml) is often preferred for its hierarchical structure and readability, especially for nested configurations.

Example src/main/resources/application.yml:

app:  
 target-url: "https://example.com/form"  
 # input-data-dir: "/path/to/input" # Input path likely comes from CLI arg now  
 output-report-dir: "/path/to/reports"  
 report-filename-prefix: "automation-report"  
  
playwright:  
 browser: "chromium" # Options: chromium, firefox, webkit  
 headless: true  
 default-timeout-ms: 30000 # 30 seconds  
 navigation-timeout-ms: 60000 # 60 seconds  
  
# Locators might be better managed elsewhere if complex, but can be here for simple cases  
locators:  
 username: "#usernameField"  
 password: "#passwordField"  
 submit-button: "button[type='submit']"  
 success-message: ".success-msg"  
# Add other configurations as needed

**(c) Mapping Properties to Java Objects (@ConfigurationProperties vs. @Value)**

Spring Boot offers two primary ways to inject configuration values into beans: @ConfigurationProperties and @Value.

* **Primary Recommendation: @ConfigurationProperties** This annotation provides a powerful, type-safe mechanism to bind external configuration properties (structured in files like YAML or properties) to Java objects (POJOs or Records).
  + **Implementation:**
    1. Create a dedicated Java class (e.g., AppConfigProperties) annotated with @ConfigurationProperties(prefix = "your-prefix"). The prefix corresponds to the top-level key in the configuration file (e.g., app, playwright).
    2. Define fields in the class matching the property keys. Spring Boot uses "relaxed binding," allowing variations like target-url (kebab-case) in YAML to map to targetUrl (camelCase) in Java. Nested properties in the configuration file map to nested classes or records within the main properties class.
    3. Enable the properties class for Spring management. This can be done by:
       - Adding @Configuration to the properties class itself.
       - Adding @EnableConfigurationProperties(AppConfigProperties.class) to another @Configuration class (often the main @SpringBootApplication class).
       - Using @ConfigurationPropertiesScan on a @Configuration class (requires Spring Boot 2.2+).
    4. Inject the fully populated properties bean (e.g., AppConfigProperties) into other beans requiring configuration values, preferably using constructor injection.
  + **Validation:** @ConfigurationProperties beans can be validated using JSR-380 annotations (e.g., @NotBlank, @Min, @Max, @Pattern, @Length) by adding @Validated to the properties class and including the spring-boot-starter-validation dependency. This ensures configuration validity at application startup.

Example (AppConfigProperties.java):package com.yourcompany.automation.config;  
  
import jakarta.validation.constraints.Max;  
import jakarta.validation.constraints.Min;  
import jakarta.validation.constraints.NotBlank;  
import jakarta.validation.constraints.NotNull;  
import org.hibernate.validator.constraints.Length; // Example validation constraint  
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.validation.annotation.Validated; // Required for validation activation  
  
// Option 1: Add @Configuration here (if not using @EnableConfigurationProperties or @ConfigurationPropertiesScan elsewhere)  
// @Configuration  
@ConfigurationProperties(prefix = "app") // Matches 'app' prefix in YAML/properties  
@Validated // Enables validation of fields annotated with JSR-380 constraints  
public class AppConfigProperties {  
  
 @NotBlank // Ensures value is not null and not empty  
 private String targetUrl;  
 // inputDataDir likely comes from CLI arg, remove if not needed here  
 // private String inputDataDir;  
 @NotBlank  
 private String outputReportDir;  
 private String reportFilenamePrefix = "automation-report"; // Example default value  
  
 @NotNull // Ensures the nested object itself is not null  
 private PlaywrightProperties playwright;  
  
 @NotNull  
 private LocatorProperties locators;  
  
 // Nested static class for Playwright properties  
 // No @ConfigurationProperties needed if field name matches prefix implicitly  
 @Validated // Can also validate nested properties  
 public static class PlaywrightProperties {  
 @NotBlank  
 private String browser;  
 private boolean headless = true; // Default value  
 @NotNull @Min(1000) // Timeout in milliseconds  
 private Integer defaultTimeoutMs;  
 @NotNull @Min(5000)  
 private Integer navigationTimeoutMs;  
  
 // Getters and Setters...  
 public String getBrowser() { return browser; }  
 public void setBrowser(String browser) { this.browser = browser; }  
 public boolean isHeadless() { return headless; }  
 public void setHeadless(boolean headless) { this.headless = headless; }  
 public Integer getDefaultTimeoutMs() { return defaultTimeoutMs; }  
 public void setDefaultTimeoutMs(Integer defaultTimeoutMs) { this.defaultTimeoutMs = defaultTimeoutMs; }  
 public Integer getNavigationTimeoutMs() { return navigationTimeoutMs; }  
 public void setNavigationTimeoutMs(Integer navigationTimeoutMs) { this.navigationTimeoutMs = navigationTimeoutMs; }  
 }  
  
 // Nested static class for Locators  
 @Validated  
 public static class LocatorProperties {  
 @NotBlank  
 private String username;  
 @NotBlank  
 private String password;  
 @NotBlank  
 private String submitButton;  
 @NotBlank  
 private String successMessage;  
  
 // Getters and Setters...  
 public String getUsername() { return username; }  
 public void setUsername(String username) { this.username = username; }  
 public String getPassword() { return password; }  
 public void setPassword(String password) { this.password = password; }  
 public String getSubmitButton() { return submitButton; }  
 public void setSubmitButton(String submitButton) { this.submitButton = submitButton; }  
 public String getSuccessMessage() { return successMessage; }  
 public void setSuccessMessage(String successMessage) { this.successMessage = successMessage; }  
 }  
  
  
 // Getters and Setters for AppConfigProperties...  
 public String getTargetUrl() { return targetUrl; }  
 public void setTargetUrl(String targetUrl) { this.targetUrl = targetUrl; }  
 public String getOutputReportDir() { return outputReportDir; }  
 public void setOutputReportDir(String outputReportDir) { this.outputReportDir = outputReportDir; }  
 public String getReportFilenamePrefix() { return reportFilenamePrefix; }  
 public void setReportFilenamePrefix(String reportFilenamePrefix) { this.reportFilenamePrefix = reportFilenamePrefix; }  
 public PlaywrightProperties getPlaywright() { return playwright; }  
 public void setPlaywright(PlaywrightProperties playwright) { this.playwright = playwright; }  
 public LocatorProperties getLocators() { return locators; }  
 public void setLocators(LocatorProperties locators) { this.locators = locators; }  
}  
*Note: Ensure the spring-boot-starter-validation dependency is included in pom.xml for validation to work.*

* **Alternative: @Value** This annotation injects individual property values directly into fields or constructor parameters using the property key.
  + **Pros:** Simple for injecting one or two unrelated properties. Supports Spring Expression Language (SpEL) for more complex value derivations.
  + **Cons:** Can lead to scattered configuration logic across multiple classes. Lacks the type safety of mapping to a dedicated object. Validation needs to be performed manually. Less resilient to refactoring (property keys are strings). Does not easily handle complex or nested structures. Can become verbose if many properties are needed.
* **Comparison and Recommendation Rationale:** @ConfigurationProperties is strongly preferred for managing groups of related configuration settings, as required in this automation tool (e.g., all app.\* settings, all playwright.\* settings). Its benefits of type safety, built-in validation support, handling of complex/nested types, and better organization significantly improve code maintainability and robustness compared to scattering @Value annotations throughout the codebase. @Value should be reserved for injecting single, isolated properties where the overhead of a dedicated @ConfigurationProperties class is not justified.

| Feature | @ConfigurationProperties | @Value |
| --- | --- | --- |
| **Type Safety** | High (binds to typed Java fields/records) | Lower (injects String, requires conversion) |
| **Validation (JSR-380)** | Built-in support via @Validated | Manual validation required |
| **Relaxed Binding** | Yes (e.g., target-url -> targetUrl) | No (requires exact property key match) |
| **Complex/Nested Types** | Excellent support (nested classes/records) | Difficult / Manual parsing required |
| **Spring Expression Language (SpEL)** | No | Yes |
| **Use Case Recommendation** | Groups of related properties, structured config | Single, isolated properties |
| **Refactoring Ease** | High (refactoring Java fields) | Lower (string-based keys are brittle) |

**(d) Environment-Specific Configurations (Profiles)**

Spring Profiles provide a standard mechanism for managing configurations across different deployment environments (e.g., development, testing, production).

* **Mechanism:** Create profile-specific configuration files named application-{profile}.yml (or .properties) in src/main/resources. For example, application-dev.yml, application-prod.yml.
* **Overrides:** Properties defined in an active profile's file will override the corresponding properties in the default application.yml (or application.properties).
* **Activation:** Profiles can be activated using various methods, including:
  + Environment variable: SPRING\_PROFILES\_ACTIVE=dev
  + System property: -Dspring.profiles.active=dev
  + Command-line argument: --spring.profiles.active=dev
  + Programmatically via SpringApplication.setAdditionalProfiles("dev")

Example (application-dev.yml):

# src/main/resources/application-dev.yml  
# Overrides for the 'dev' profile  
app:  
 target-url: "http://localhost:8080/test-form" # Use local test URL  
  
playwright:  
 headless: false # Run browser with UI visible for easier debugging during development  
 default-timeout-ms: 60000 # Longer timeout for debugging

When the application is run with the dev profile active (e.g., java -jar your-app.jar --spring.profiles.active=dev), the target-url will be http://localhost:8080/test-form and Playwright will run headful.

**Configuration Source Precedence:** It is crucial to understand that Spring Boot loads configuration from multiple sources in a specific, hierarchical order. Later sources override earlier ones. Key sources in descending order of precedence include: Command-line arguments, Environment variables, Profile-specific application-{profile}.yml, Profile-specific application-{profile}.properties, Default application.yml, Default application.properties. This predictable hierarchy allows for flexible configuration overrides. For instance, a sensitive password might be set via an environment variable in production , overriding any value potentially present in configuration files, while a developer might override the target-url using a command-line argument for a single test run without modifying any files.

## V. Spring Boot Logging Integration

Spring Boot provides excellent integration with common logging frameworks, simplifying configuration and removing the need for custom setup code.

**(a) Default Logging Framework (Logback)**

By default, Spring Boot utilizes Logback as its logging implementation when using the standard starters (spring-boot-starter, spring-boot-starter-web, etc.). It automatically configures console logging with sensible defaults, including color-coded output where supported. The underlying logging API used should be SLF4J (Simple Logging Facade for Java). Using the SLF4J API (org.slf4j.Logger, org.slf4j.LoggerFactory) in application code ensures decoupling from the specific logging implementation (Logback), allowing for easier migration to other frameworks like Log4j2 if ever needed.

**(b) Replacing Custom LoggingSetup**

The previously required custom LoggingSetup component becomes redundant and should be removed. Spring Boot's auto-configuration and externalized configuration properties provide standard, powerful mechanisms for managing logging levels, output destinations (console, file), formatting, and rolling policies.

**(c) Configuration Methods**

Logging can be configured primarily through two standard Spring Boot mechanisms:

1. **Via application.properties or application.yml:** This method is suitable for common configuration needs.
   * **Log Levels:** Control the verbosity for specific packages/classes or the root logger using the logging.level. prefix (e.g., logging.level.root=WARN, logging.level.com.yourcompany.automation=DEBUG).
   * **File Logging:**
     + Specify a log file name and path: logging.file.name=logs/automation.log.
     + Specify only a path (uses spring.log as filename): logging.file.path=/var/log/automation/. (logging.file.name takes precedence if both are set ).
   * **Rolling Policies (Logback):** Configure file rotation based on size and/or time using logging.logback.rollingpolicy.\* properties.
     + file-name-pattern: Defines the pattern for archived files (e.g., automation-%d{yyyy-MM-dd}.%i.log.gz). %d for date, %i for index.
     + max-file-size: Maximum size for each log file before rolling (e.g., 10MB).
     + total-size-cap: Maximum total size of all archived log files (e.g., 500MB).
     + max-history: Maximum number of archive files or days to keep (e.g., 7).
     + clean-history-on-start: Clean archives on application start (e.g., true).
   * **Log Pattern:** Customize the format of log messages for console or file output using logging.pattern.console or logging.pattern.file.

**Table: Common Logging Configuration Properties**

| Property Key | Description | Example Value |
| --- | --- | --- |
| logging.level.root | Root logger level | INFO |
| logging.level.<logger-name> | Level for a specific logger (package/class) | logging.level.com.yourcompany=DEBUG |
| logging.file.name | Full path and name of the log file | logs/app.log |
| logging.file.path | Path to store log file (default name spring.log) | /var/log/app/ |
| logging.pattern.console | Log format pattern for console output | "%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n" |
| logging.pattern.file | Log format pattern for file output | "%d %-5level [%thread] %logger : %msg%n" |
| logging.logback.rollingpolicy.file-name-pattern | Pattern for archived log file names (requires Logback) | logs/app-%d{yyyy-MM-dd}.%i.log.gz |
| logging.logback.rollingpolicy.max-file-size | Maximum size of a log file before rolling (requires Logback) | 10MB |
| logging.logback.rollingpolicy.total-size-cap | Total size cap for all archived logs (requires Logback) | 1GB |
| logging.logback.rollingpolicy.max-history | Maximum number of archived log files/days to keep (requires Logback) | 7 |

1. **Via logback-spring.xml (or logback.xml):** This method provides maximum flexibility and control over Logback's features.
   * **Location:** Place the file in src/main/resources. Using logback-spring.xml is preferred as it enables Spring Boot extensions, such as profile-specific configurations using <springProfile> tags and access to Spring Environment properties using <springProperty>.
   * **Overrides:** This configuration file completely overrides any logging settings defined in application.properties/yml.
   * **Structure:** Define <appender> elements (e.g., ConsoleAppender, RollingFileAppender) and <logger> elements within the root <configuration> tag. You can include Spring Boot's default configurations using <include resource="org/springframework/boot/logging/logback/base.xml"/> or more specific includes like defaults.xml, console-appender.xml, file-appender.xml to reuse standard patterns and appenders.
   * **Example (logback-spring.xml for Size and Time Based Rolling):**  
     <?xml version="1.0" encoding="UTF-8"?>  
     <configuration>  
      <include resource="org/springframework/boot/logging/logback/defaults.xml"/>  
      <include resource="org/springframework/boot/logging/logback/console-appender.xml" />  
       
      <springProperty scope="context" name="LOG\_PATH" source="logging.file.path" defaultValue="./logs"/>  
      <springProperty scope="context" name="LOG\_FILE\_NAME" source="logging.file.name" defaultValue="automation-tool.log"/>  
      <property name="LOG\_FILE" value="${LOG\_PATH}/${LOG\_FILE\_NAME}"/>  
      <property name="FILE\_LOG\_PATTERN" value="%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n"/>  
      <property name="ARCHIVE\_PATTERN" value="${LOG\_PATH}/archived/${LOG\_FILE\_NAME}.%d{yyyy-MM-dd}.%i.log.gz"/>  
       
      <appender name="ROLLING\_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">  
      <file>${LOG\_FILE}</file> <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy"> <fileNamePattern>${ARCHIVE\_PATTERN}</fileNamePattern>  
      <maxFileSize>10MB</maxFileSize> <maxHistory>7</maxHistory> <totalSizeCap>1GB</totalSizeCap> </rollingPolicy>  
      <encoder>  
      <pattern>${FILE\_LOG\_PATTERN}</pattern>  
      </encoder>  
      </appender>  
       
      <root level="INFO"> <appender-ref ref="CONSOLE" />  
      <appender-ref ref="ROLLING\_FILE" />  
      </root>  
       
      <logger name="com.yourcompany.automation" level="DEBUG" additivity="false">  
      <appender-ref ref="CONSOLE" />  
      <appender-ref ref="ROLLING\_FILE" />  
      </logger>  
       
      <logger name="com.microsoft.playwright" level="WARN"/>  
       
     </configuration>

**Recommendation:** Begin with configuration in application.properties/yml for simplicity. Only introduce a logback-spring.xml file if advanced features like multiple distinct appenders, complex filtering, profile-specific appenders, or configurations not exposed via properties are required. The XML approach offers maximum control but increases configuration complexity.

## VI. Playwright Lifecycle Management in Spring

Properly managing the lifecycle of Playwright objects (Playwright instance, Browser instance) is critical, as they control external browser processes and consume system resources. Spring's bean lifecycle management provides a robust mechanism for this.

**(a) Managing Playwright Objects within Spring Beans**

The Playwright and Browser instances should be managed within a Spring-managed bean. This could be the WebAutomationEngine itself or a dedicated PlaywrightManager bean if separation is desired.

* **Bean Scope:** A @Scope("singleton") (the default scope) is generally appropriate for these objects within a CLI application context. This means a single Playwright instance and potentially a single Browser instance will be created and reused for the duration of the application run. While Playwright supports multiple BrowserContexts for isolation within a single Browser instance , creating a fresh Browser instance per execution (e.g., inside the Orchestrator.executeAutomation method) is possible but adds significant startup overhead and is likely unnecessary for this tool's typical single-run nature.
* **Initialization (@PostConstruct):** Use the @PostConstruct annotation on a method within the managing bean. This method will be called by Spring after the bean has been created and dependencies (like AppConfigProperties) have been injected. Inside this method, initialize the Playwright instance (Playwright.create()) and launch the Browser instance (playwright.chromium().launch(...), configured using injected properties). This ensures Playwright is ready before the application logic attempts to use it.
* **Cleanup (@PreDestroy / DisposableBean):** It is crucial to properly close the Browser and Playwright instances to terminate browser processes and release resources.
  + **@PreDestroy:** Annotate a cleanup method within the managing bean with @PreDestroy. Spring guarantees to call this method during application context shutdown, just before the bean is destroyed. This method should contain the logic to close the browser (browser.close()) and the Playwright instance (playwright.close()). This is the generally preferred approach as it avoids coupling the bean code to Spring-specific interfaces.
  + **DisposableBean:** Alternatively, the bean can implement the DisposableBean interface and place the cleanup logic within the overridden destroy() method.

Example Snippet (within WebAutomationEngine):

package com.yourcompany.automation.engine;  
  
import com.microsoft.playwright.\*;  
import com.microsoft.playwright.options.BrowserType;  
import com.yourcompany.automation.config.AppConfigProperties;  
import jakarta.annotation.PostConstruct; // Standard Java annotation  
import jakarta.annotation.PreDestroy; // Standard Java annotation  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.stereotype.Service;  
// import org.springframework.beans.factory.DisposableBean; // Alternative  
  
import java.util.List;  
import java.util.Map;  
  
@Service  
public class WebAutomationEngine /\* implements DisposableBean \*/ { // Alternative  
  
 private static final Logger log = LoggerFactory.getLogger(WebAutomationEngine.class);  
  
 private final AppConfigProperties configProperties;  
 private Playwright playwright;  
 private Browser browser;  
  
 public WebAutomationEngine(AppConfigProperties configProperties) {  
 this.configProperties = configProperties;  
 }  
  
 @PostConstruct // Called after bean creation and DI  
 public void initializePlaywright() {  
 log.info("Initializing Playwright and launching browser...");  
 try {  
 // Ensure Playwright binaries are downloaded/managed appropriately  
 // Consider playwright.CLI.main(new String{"install", configProperties.getPlaywright().getBrowser()});  
 // if needed, though often handled by Playwright library itself or build process.  
  
 playwright = Playwright.create();  
 AppConfigProperties.PlaywrightProperties pwConfig = configProperties.getPlaywright();  
 BrowserType.LaunchOptions launchOptions = new BrowserType.LaunchOptions()  
 .setHeadless(pwConfig.isHeadless())  
 .setTimeout(pwConfig.getDefaultTimeoutMs().doubleValue()); // Browser launch timeout  
  
 log.info("Launching {} browser (headless: {})", pwConfig.getBrowser(), pwConfig.isHeadless());  
 switch (pwConfig.getBrowser().toLowerCase()) {  
 case "firefox":  
 browser = playwright.firefox().launch(launchOptions);  
 break;  
 case "webkit":  
 browser = playwright.webkit().launch(launchOptions);  
 break;  
 case "chromium":  
 default:  
 browser = playwright.chromium().launch(launchOptions);  
 break;  
 }  
 log.info("Playwright initialized successfully. Browser connected: {}", browser.isConnected());  
 } catch (Exception e) {  
 log.error("FATAL: Failed to initialize Playwright or launch browser", e);  
 // Consider using ApplicationEventPublisher to signal failure or using ExitCodeGenerator  
 throw new RuntimeException("Playwright initialization failed", e);  
 }  
 }  
  
 @PreDestroy // Called before bean destruction  
 public void closePlaywright() {  
 log.info("Closing Playwright resources...");  
 if (browser!= null && browser.isConnected()) {  
 log.info("Closing browser instance...");  
 try {  
 browser.close();  
 log.info("Browser closed successfully.");  
 } catch (PlaywrightException e) {  
 log.warn("Error closing browser instance: {}", e.getMessage());  
 }  
 } else {  
 log.info("Browser was null or not connected.");  
 }  
 if (playwright!= null) {  
 log.info("Closing Playwright instance...");  
 try {  
 playwright.close();  
 log.info("Playwright instance closed successfully.");  
 } catch (PlaywrightException e) {  
 log.warn("Error closing Playwright instance: {}", e.getMessage());  
 }  
 }  
 log.info("Playwright resources cleanup finished.");  
 }  
  
 /\* Alternative using DisposableBean  
 @Override  
 public void destroy() throws Exception { // [span\_168](start\_span)[span\_168](end\_span)[span\_170](start\_span)[span\_170](end\_span)  
 closePlaywright();  
 }  
 \*/  
  
 public Map<String, String> processForms(List<Map<String, String>> data) {  
 if (browser == null ||!browser.isConnected()) {  
 log.error("Browser is not initialized or not connected. Cannot process forms.");  
 throw new IllegalStateException("Playwright browser is not available.");  
 }  
  
 // Use a new BrowserContext for each run to ensure isolation [span\_171](start\_span)[span\_171](end\_span)[span\_172](start\_span)[span\_172](end\_span)  
 log.info("Creating new browser context for processing {} records.", data.size());  
 try (BrowserContext context = browser.newContext()) {  
 Page page = context.newPage();  
 // Apply timeouts from configuration  
 page.setDefaultTimeout(configProperties.getPlaywright().getDefaultTimeoutMs());  
 page.setDefaultNavigationTimeout(configProperties.getPlaywright().getNavigationTimeoutMs());  
  
 log.info("Navigating to target URL: {}", configProperties.getTargetUrl());  
 page.navigate(configProperties.getTargetUrl()); // [span\_254](start\_span)[span\_254](end\_span)[span\_255](start\_span)[span\_255](end\_span)  
  
 // --- Main Automation Logic ---  
 // Iterate through 'data', use locators from configProperties.getLocators()  
 // Fill form fields: page.locator(...).fill(...) [span\_173](start\_span)[span\_173](end\_span)[span\_174](start\_span)[span\_174](end\_span)  
 // Handle checkboxes/radios: page.locator(...).check() [span\_175](start\_span)[span\_175](end\_span)[span\_176](start\_span)[span\_176](end\_span)[span\_177](start\_span)[span\_177](end\_span)[span\_178](start\_span)[span\_178](end\_span)  
 // Handle dropdowns: page.locator(...).selectOption(...) [span\_179](start\_span)[span\_179](end\_span)[span\_180](start\_span)[span\_180](end\_span)[span\_181](start\_span)[span\_181](end\_span)  
 // Click submit: page.locator(...).click() [span\_182](start\_span)[span\_182](end\_span)[span\_183](start\_span)[span\_183](end\_span)[span\_184](start\_span)[span\_184](end\_span)[span\_185](start\_span)[span\_185](end\_span)  
 // Wait for navigation/results: page.waitForURL(...), expect(page.locator(...)).toBeVisible() [span\_186](start\_span)[span\_186](end\_span)[span\_187](start\_span)[span\_187](end\_span)[span\_188](start\_span)[span\_188](end\_span)[span\_189](start\_span)[span\_189](end\_span)[span\_190](start\_span)[span\_190](end\_span)[span\_191](start\_span)[span\_191](end\_span)  
 // Implement robust error handling for Playwright actions [span\_192](start\_span)[span\_192](end\_span)[span\_193](start\_span)[span\_193](end\_span)[span\_194](start\_span)[span\_194](end\_span)[span\_195](start\_span)[span\_195](end\_span)[span\_196](start\_span)[span\_196](end\_span)  
  
 log.info("Simulating form processing for {} records...", data.size());  
 // Placeholder for actual processing and result collection  
 Map<String, String> results = Map.of("status", "Processed " + data.size() + " records successfully.");  
 // --- End Automation Logic ---  
  
 log.info("Closing page.");  
 page.close();  
 log.info("Page closed.");  
 return results;  
  
 } catch (PlaywrightException e) {  
 log.error("Playwright error during form processing: {}", e.getMessage(), e);  
 // Implement more specific error handling/reporting [span\_197](start\_span)[span\_197](end\_span)[span\_198](start\_span)[span\_198](end\_span)[span\_199](start\_span)[span\_199](end\_span)  
 throw new RuntimeException("Form processing failed due to Playwright error", e);  
 } finally {  
 log.info("Browser context closed.");  
 }  
 }  
 //... other methods  
}

By placing Playwright initialization (@PostConstruct) and cleanup (@PreDestroy) within the same Spring bean responsible for using these resources (like WebAutomationEngine), the lifecycle management becomes tightly coupled and reliable. Spring's container guarantees the invocation of these lifecycle methods , ensuring that external browser processes are properly started and, more importantly, terminated, preventing resource leaks.

Furthermore, even within a single execution of the CLI tool, using a new BrowserContext for each logical processing unit (e.g., each call to processForms, potentially processing multiple data rows) provides crucial isolation. Similar to an incognito window, a BrowserContext prevents state like cookies or local storage from leaking between the processing of different data entries, which is important if the target web application maintains session state. Using a try-with-resources block ensures the context and its pages are closed automatically after use.

## VII. Maven Build Configuration (pom.xml)

Apache Maven will manage the project's build lifecycle, dependencies, and packaging. The pom.xml file is central to this configuration.

**(a) Necessary Dependencies**

The pom.xml must declare dependencies required by the application. Key dependencies include:

* **Spring Boot Starter:** Provides core Spring Boot functionality, auto-configuration, and logging (Logback + SLF4J).
  + groupId: org.springframework.boot
  + artifactId: spring-boot-starter
* **Playwright:** The core library for browser automation.
  + groupId: com.microsoft.playwright
  + artifactId: playwright
* **Excel Data Reading (Apache POI):** For reading data from .xlsx files.
  + groupId: org.apache.poi
  + artifactId: poi-ooxml (Includes poi transitively)
* **Validation:** Required if using JSR-380 validation annotations with @ConfigurationProperties.
  + groupId: org.springframework.boot
  + artifactId: spring-boot-starter-validation
* **Lombok (Optional):** Reduces boilerplate code (getters, setters, etc.) in POJOs.
  + groupId: org.projectlombok
  + artifactId: lombok
  + Scope should typically be provided or optional=true as it's mainly a compile-time tool.

**Table: Key Maven Dependencies**

| GroupId | ArtifactId | Description | Required/Optional |
| --- | --- | --- | --- |
| org.springframework.boot | spring-boot-starter | Core Spring Boot, Auto-config, Logging (Logback) | Required |
| com.microsoft.playwright | playwright | Playwright browser automation library | Required |
| org.apache.poi | poi-ooxml | Read/Write OOXML (.xlsx) Excel files | Required |
| org.springframework.boot | spring-boot-starter-validation | Bean validation (JSR-380) support | Required (if using validation) |
| org.projectlombok | lombok | Reduces boilerplate code (getters, setters, etc.) | Optional |

**(b) spring-boot-maven-plugin for Executable JAR**

To create a single, executable JAR file containing the application code and all its dependencies (a "fat JAR" or "uber JAR"), the spring-boot-maven-plugin is essential.

* **Purpose:** This plugin repackages the standard JAR produced by Maven. It embeds dependencies into a specific structure (e.g., BOOT-INF/lib/) and adds a special Spring Boot loader (org.springframework.boot.loader.\*) that understands how to load classes from these nested JARs. It also correctly configures the MANIFEST.MF file to make the JAR runnable via java -jar.
* **Configuration:**
  + Include the plugin in the <build><plugins> section of pom.xml.
  + Define an <execution> block bound to the package phase, specifying the repackage goal.
  + **(Optional) mainClass:** Explicitly specify the fully qualified name of the class containing the main method (the one annotated with @SpringBootApplication). While the plugin can often find it automatically, explicit configuration is more robust.
  + **(Optional) classifier:** If set (e.g., exec), the plugin generates a separate executable JAR (e.g., myapp-exec.jar) alongside the original thin JAR. If omitted (default), the plugin replaces the original JAR with the executable fat JAR. For a standalone CLI tool, replacing the original is usually the desired behavior.

Example pom.xml Snippet:

<project...>  
 <parent> <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>3.2.5</version> <relativePath/> </parent>  
  
 <groupId>com.yourcompany.automation</groupId>  
 <artifactId>automation-tool</artifactId>  
 <version>1.0.0-SNAPSHOT</version>  
 <name>Web Form Automation Tool</name>  
 <description>Java/Playwright tool for web form automation using Spring Boot</description>  
  
 <properties>  
 <java.version>17</java.version> <playwright.version>1.43.0</playwright.version> <poi.version>5.2.5</poi.version> </properties>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>com.microsoft.playwright</groupId>  
 <artifactId>playwright</artifactId>  
 <version>${playwright.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.apache.poi</groupId>  
 <artifactId>poi-ooxml</artifactId>  
 <version>${poi.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-validation</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <optional>true</optional>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 </dependencies>  
  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 <executions>  
 <execution>  
 <goals>  
 <goal>repackage</goal> </goals>  
 </execution>  
 </executions>  
 <configuration>  
 <mainClass>com.yourcompany.automation.AutomationToolApplication</mainClass>  
 </configuration>  
 </plugin>  
 </plugins>  
 </build>  
</project>

Using the spring-boot-starter-parent POM is highly recommended as it provides dependency management for common libraries and configures the spring-boot-maven-plugin with sensible defaults, reducing boilerplate in the project's pom.xml. The repackage goal ensures the creation of a self-contained, executable JAR, simplifying distribution and execution (java -jar your-app.jar), which is ideal for a command-line utility.

## VIII. Command-Line Argument Handling with Spring Boot

The application needs to accept the path to the input Excel file as a command-line argument. Spring Boot provides mechanisms to access these arguments within the application context, particularly within the ApplicationRunner or CommandLineRunner.

**(a) Accessing Non-Option Arguments**

The primary command-line argument for this tool is the input file path, which is expected to be a non-option argument (i.e., not prefixed with --).

* **Using ApplicationRunner and ApplicationArguments (Recommended):** When implementing ApplicationRunner, the run method receives an ApplicationArguments object. This object provides convenient methods to distinguish between option and non-option arguments. The args.getNonOptionArgs() method returns a List<String> containing only the arguments passed without the -- prefix. This is the cleanest and most reliable way to retrieve the input file path.  
  // Inside ApplicationRunner's run(ApplicationArguments args) method:  
  List<String> nonOptionArgs = args.getNonOptionArgs();  
  if (nonOptionArgs.isEmpty()) {  
   log.error("Missing required argument: Input Excel file path.");  
   // Use Spring Boot's ExitCodeGenerator or System.exit  
   throw new IllegalArgumentException("Input file path argument is required.");  
  }  
  String inputFilePath = nonOptionArgs.get(0); // Assuming the first non-option arg is the file path  
  log.info("Processing input file: {}", inputFilePath);  
  // Pass inputFilePath to the Orchestrator  
  orchestrator.executeAutomation(inputFilePath);
* **Using CommandLineRunner:** The run(String... args) method of CommandLineRunner receives all arguments, including options, as a single String array. Identifying the non-option file path requires manual parsing, such as iterating through the array or assuming its position (e.g., the last argument), which can be error-prone if option arguments are also used.  
  // Inside CommandLineRunner's run(String... args) method:  
  String inputFilePath = null;  
  if (args.length > 0) {  
   // Simplistic: assumes last argument is the file path. Needs better parsing if options exist.  
   inputFilePath = args[args.length - 1];  
  }  
  if (inputFilePath == null |

| inputFilePath.startsWith("--")) { // Basic check log.error("Missing or invalid input file path argument."); throw new IllegalArgumentException("Input file path argument is required."); } log.info("Processing input file: {}", inputFilePath); orchestrator.executeAutomation(inputFilePath); ```

**Argument Injection Timing:** Spring Boot processes command-line arguments early in the startup sequence. It populates an ApplicationArguments bean which is then available for injection into other beans, including the ApplicationRunner or CommandLineRunner. This ensures that when the runner's run method is executed, the command-line arguments have already been processed and are accessible.

## IX. Updated Component Interactions

The integration of Spring Boot fundamentally changes how components interact, shifting towards a dependency-injection-managed architecture.

* **Central Role of Spring IoC Container:** The Spring container becomes the core manager. It instantiates beans (classes annotated with @Component, @Service, etc.), manages their lifecycles (including calling @PostConstruct and @PreDestroy methods), and injects dependencies.
* **Application Entry Point:** Execution begins with AutomationToolApplication.main(), which calls SpringApplication.run(). This initializes the Spring context. Once the context is ready, Spring invokes the run method of the registered ApplicationRunner bean (e.g., AutomationRunner).
* **Dependency Injection:** Instead of manual instantiation (e.g., new Orchestrator(...)), dependencies are declared via constructors. Spring automatically injects the required bean instances. For example, the AutomationRunner receives an Orchestrator instance, and the Orchestrator receives instances of DataReader, WebAutomationEngine, ReportGenerator, and AppConfigProperties. Arrows in a conceptual diagram would represent these injected dependencies.
* **Configuration Flow:** Components no longer rely on a custom ConfigManager. Beans needing configuration (like WebAutomationEngine needing browser settings or Orchestrator needing the target URL) receive the AppConfigProperties bean via constructor injection. Spring populates this bean from application.yml/properties and other sources (environment variables, command-line args) according to its precedence rules.
* **Logging:** The custom LoggingSetup is removed. All components obtain an SLF4J logger (LoggerFactory.getLogger(...)). Logging behavior (level, file output, rotation) is configured centrally via Spring Boot's mechanisms (application.yml or logback-spring.xml).
* **Playwright Lifecycle:** The Playwright and Browser objects' lifecycles are managed within their containing Spring bean (e.g., WebAutomationEngine) using @PostConstruct for setup and @PreDestroy for teardown, orchestrated by the Spring container.

This architectural shift moves from a potentially more procedural setup, where the main method might explicitly wire objects, to a **declarative model**. Developers *declare* components, dependencies, and configurations using annotations and property files. The Spring framework then interprets these declarations to assemble and manage the application. This declarative approach enhances modularity, reduces boilerplate code, simplifies testing, and makes the overall application structure more maintainable and aligned with modern Java practices.

## X. Conclusion

This updated system design integrates Apache Maven and Spring Boot into the Java/Playwright web form automation tool, resulting in a more robust, maintainable, and deployable application. Key transformations include:

1. **Maven Build System:** Provides standardized dependency management and build lifecycle control, configured via pom.xml.
2. **Spring Boot Application Structure:** Leverages the Spring IoC container for bean management and dependency injection (via constructor injection), using @SpringBootApplication and ApplicationRunner as the core structure.
3. **Executable Fat JAR:** The spring-boot-maven-plugin's repackage goal produces a single, self-contained executable JAR, simplifying deployment and execution (java -jar).
4. **Standardized Configuration:** Replaces custom configuration management with Spring Boot's externalized configuration using application.yml (or .properties) and type-safe binding via @ConfigurationProperties. Spring Profiles enable environment-specific settings.
5. **Integrated Logging:** Utilizes Spring Boot's default Logback integration, configured via application properties or logback-spring.xml, leveraging the SLF4J API.
6. **Managed Playwright Lifecycle:** Employs Spring bean lifecycle callbacks (@PostConstruct, @PreDestroy) for reliable initialization and cleanup of Playwright resources.
7. **Simplified Argument Handling:** Uses ApplicationArguments injected into the ApplicationRunner for clean access to command-line arguments.

These changes collectively lead to a system that is easier to develop, test, configure, and run. The adoption of established frameworks like Maven and Spring Boot aligns the tool with industry best practices, enhancing its long-term viability and maintainability.

Next steps involve implementing these design changes, developing corresponding unit and integration tests leveraging Spring Boot's testing support, and configuring any necessary build and deployment pipelines.

#### Quellenangaben

1. Spring Boot: ApplicationRunner and CommandLineRunner - DZone, https://dzone.com/articles/spring-boot-applicationrunner-and-commandlinerunne 2. Spring Boot Runners - Tutorialspoint, https://www.tutorialspoint.com/spring\_boot/spring\_boot\_runners.htm 3. How to use CommandLineRunner in Spring Boot Application? - Jhooq, https://jhooq.com/commandlinerunner-spring-boot/ 4. Spring Boot CommandLineRunner Example - Mkyong.com, https://mkyong.com/spring-boot/spring-boot-commandlinerunner-example/ 5. CommandLineRunner (Spring Boot 3.4.4 API), https://docs.spring.io/spring-boot/api/java/org/springframework/boot/CommandLineRunner.html 6. Spring Boot Runners: CommandLine vs Application - JavaDZone, https://javadzone.com/spring-boot-runners-commandline-vs-application/ 7. ApplicationRunner VS CommandLineRunner - java - Stack Overflow, https://stackoverflow.com/questions/65980516/applicationrunner-vs-commandlinerunner 8. Spring Boot – CommandLineRunner and ApplicationRunner - GeeksforGeeks, https://www.geeksforgeeks.org/spring-boot-commandlinerunner-and-applicationrunner/ 9. Parsing Arguments in Commandline Applications with Spring Boot - codeboje, https://codeboje.de/spring-boot-commandline-app-args/ 10. Which is better approach ? : r/SpringBoot - Reddit, https://www.reddit.com/r/SpringBoot/comments/1gutmkt/which\_is\_better\_approach/ 11. Python project structure - Python discussion forum, https://discuss.python.org/t/python-project-structure/36119 12. Externalized Configuration :: Spring Boot, https://docs.spring.io/spring-boot/reference/features/external-config.html 13. 24. Externalized Configuration - Spring, https://docs.spring.io/spring-boot/docs/2.1.7.RELEASE/reference/html/boot-features-external-config.html 14. java - @ConfigurationProperties vs @PropertySource vs @Value - Stack Overflow, https://stackoverflow.com/questions/58691325/configurationproperties-vs-propertysource-vs-value 15. Working with Python Configuration Files: Tutorial & Best Practices - Configu, https://configu.com/blog/working-with-python-configuration-files-tutorial-best-practices/ 16. Spring Boot – @ConfigurationProperties | GeeksforGeeks, https://www.geeksforgeeks.org/spring-boot-configurationproperties/ 17. Guide to @ConfigurationProperties in Spring Boot | Baeldung, https://www.baeldung.com/configuration-properties-in-spring-boot 18. ConfigurationProperties versus @Value - Learning Spring Boot 2.0 - O'Reilly, https://www.oreilly.com/library/view/learning-spring-boot/9781786463784/404e3bec-c18d-4c6e-b4ce-b1ad036ad534.xhtml 19. python-dotenv - PyPI, https://pypi.org/project/python-dotenv/ 20. Using Py Dotenv (python-dotenv) Package to Manage Env Variables - Configu, https://configu.com/blog/using-py-dotenv-python-dotenv-package-to-manage-env-variables/ 21. Python YAML configuration with environment variables parsing - DEV Community, https://dev.to/mkaranasou/python-yaml-configuration-with-environment-variables-parsing-2ha6 22. Settings Management - Pydantic, https://docs.pydantic.dev/latest/concepts/pydantic\_settings/ 23. 83.1 Configure Logback for Logging - Spring, https://docs.spring.io/spring-boot/docs/2.1.8.RELEASE/reference/html/howto-logging.html 24. Logging in Spring Boot - Baeldung, https://www.baeldung.com/spring-boot-logging 25. Using Logback with Spring Boot, https://springframework.guru/using-logback-spring-boot/ 26. Logging :: Spring Boot, https://docs.spring.io/spring-boot/reference/features/logging.html 27. How to Configure Default Log Files - Spring Boot Logging | SigNoz, https://signoz.io/guides/default-logging-file-for-spring-boot-application/ 28. Springboot Logging Pattern For Rolling Logs - Stack Overflow, https://stackoverflow.com/questions/74068787/springboot-logging-pattern-for-rolling-logs 29. How to use Logback in Spring Boot – Rolling Files Example - CodeJava.net, https://www.codejava.net/frameworks/spring-boot/logback-rolling-files-example 30. How to configure rolling file appender within Spring Boot's application.yml - Stack Overflow, https://stackoverflow.com/questions/29918323/how-to-configure-rolling-file-appender-within-spring-boots-application-yml 31. A Guide to Rolling File Appenders - Baeldung, https://www.baeldung.com/java-logging-rolling-file-appenders 32. Chapter 4: Appenders - Logback, https://logback.qos.ch/manual/appenders.html 33. Logback Xml Configuration Examples - Java Logback Config - JavaTechOnline, https://javatechonline.com/logback-xml-configuration-examples/ 34. How to pass properties from application.properties to logback config file - Stack Overflow, https://stackoverflow.com/questions/40497947/how-to-pass-properties-from-application-properties-to-logback-config-file 35. Isolation | Playwright Python, https://playwright.dev/python/docs/browser-contexts 36. Playwright vs Puppeteer vs Cypress vs Selenium (E2E testing) | Better Stack Community, https://betterstack.com/community/comparisons/playwright-cypress-puppeteer-selenium-comparison/ 37. Selenium vs Puppeteer vs Playwright: Choosing the Right Tool for Web Automation, https://dev.to/mechcloud\_academy/selenium-vs-puppeteer-vs-playwright-choosing-the-right-tool-for-web-automation-5el 38. Bean Life Cycle in Java Spring | GeeksforGeeks, https://www.geeksforgeeks.org/bean-life-cycle-in-java-spring/ 39. Customizing the Nature of a Bean :: Spring Framework, https://docs.spring.io/spring-framework/reference/core/beans/factory-nature.html 40. Bean Lifecycle Management Spring Boot - java - Stack Overflow, https://stackoverflow.com/questions/39370021/bean-lifecycle-management-spring-boot 41. Spring PostConstruct and PreDestroy Annotations | Baeldung, https://www.baeldung.com/spring-postconstruct-predestroy 42. SpringBoot bean lifecycle management - Stack Overflow, https://stackoverflow.com/questions/57137250/springboot-bean-lifecycle-management 43. 15.Spring Bean Life Cycle methods @PostConstruct and @PreDestroy and their usage | Zest Prime - YouTube, https://www.youtube.com/watch?v=8RkXXPdmjfI 44. Pandas or openpyxl : r/learnpython - Reddit, https://www.reddit.com/r/learnpython/comments/r23xne/pandas\_or\_openpyxl/ 45. Reading an excel file using Python openpyxl module | GeeksforGeeks, https://www.geeksforgeeks.org/python-reading-excel-file-using-openpyxl-module/ 46. Spring Boot Maven Plugin – spring-boot:repackage, https://docs.spring.io/spring-boot/docs/1.5.2.RELEASE/maven-plugin/repackage-mojo.html 47. Tutorial: Build Fat Jar with Maven Plugin & Include all Dependencies - Aegis Softtech, https://www.aegissofttech.com/articles/how-to-build-a-fatjar-using-maven-in-java.html 48. spring-boot-maven-plugin doesn't create fat jar - Stack Overflow, https://stackoverflow.com/questions/49590459/spring-boot-maven-plugin-doesnt-create-fat-jar 49. 68.2 Create an executable JAR with Maven - Spring, https://docs.spring.io/spring-boot/docs/1.1.2.RELEASE/reference/html/howto-build.html 50. Difference Between spring-boot:repackage and Maven package ..., https://www.baeldung.com/spring-boot-repackage-vs-mvn-package 51. Spring Sweets: Access Application Arguments With ApplicationArguments Bean - JDriven, https://jdriven.com/blog/2017/03/spring-sweets-access-application-arguments-applicationarguments-bean 52. Accessing Application Arguments :: Spring Boot - GitHub Pages, https://rwinch.github.io/spring-boot/features/spring-application/application-arguments.html 53. 23. SpringApplication, https://docs.spring.io/spring-boot/docs/1.3.0.M2/reference/html/boot-features-spring-application.html 54. how to use the ApplicationArguments in spring-boot - Stack Overflow, https://stackoverflow.com/questions/38802355/how-to-use-the-applicationarguments-in-spring-boot 55. Provide a way to inject parsed application arguments · Issue #1990 · spring-projects/spring-boot - GitHub, https://github.com/spring-projects/spring-boot/issues/1990