Mauli Bhavsar

Plugin-development

Overview

Develop an IntelliJ plugin to detect and highlight probable deployment issues due to changes. This plugin can be further converted to GitLab MR pipeline components.

Goals achieved :

Plugin will be able to flag the following issues in real time therefore preventing the deployment from breaking:

1. **Circular dependency:** Detects circular dependencies during bean initialization, where two classes depend on each other for bean creation.
2. **No bean found because package is not specified in .xml:** Identify if a class from a particular package is initialised somewhere but is not mentioned in any .xml file, leading to a "no bean found" error. The plugin will provide the package name that is used but not declared in .xml.
3. **Not annotated with @Serive/@Component/@Repository and @Autowired:** Detects missing annotations on classes, fields, or parameters based on their usage. This ensures that all necessary Spring annotations are present.
4. **Duplicate qualifiers ,Multiple beans:** Identify situations where multiple beans are available for the same type and Spring needs clarification about which child class to use for bean initialization. This is crucial as instances of interfaces or abstract classes cannot be created directly.
5. **Rest API error, for GET methods while defining payload:** Validates all the GET methods and its parameters and detects if parameter has missing annotation.

Project approach till now

Traverse all the modules:

- Navigate through all the modules.
- Locate all `spring-*.xml` files.

Identify Packages and Classes:

- Within each `spring-*.xml` file, identify all the packages specified.
- Traverse through each package to locate all classes.

Check Class Annotations:

- For each class in the packages, check if the class is annotated with `@Service`, `@Repository`, or `@Component`.
- For classes annotated with `@Service`, `@Repository`, or `@Component`:

  A. Handling @Autowired Fields

- Collect all fields in the class that are annotated with `@Autowired`. These will be excluded from further checks.

  B. Checking Parent Classes of Fields

- For each field declared in the class:
  - Check if the parent class of the field is not an interface or abstract class.
  - Ensure the parent class is annotated with `@Service`.

  C. Analysing Constructors

  Check if the parameters of any constructors are annotated with `@Autowired`.

  If constructors have parameters annotated with `@Autowired`:

  - Skip further checks for those constructors.

  If constructors do not have parameters annotated with `@Autowired:`

- **Single Constructor**:
    - If a class has a single constructor, no annotation is required, but it's recommended to use `@Autowired` for clarity.
- **Multiple Constructors**:
    - **With a Default Constructor**:
        - If there is one default constructor (no-args constructor), no `@Autowired` annotation is necessary as it will be automatically used.
    - **Without a Default Constructor**:
        - If there are multiple constructors and none is a default constructor, at least one constructor must be annotated with `@Autowired`.
        - Check if the user meets these conditions and provide warnings if necessary.

Changes to Include Checks for Multiple Beans and Duplicate Qualifiers

F. Checking Multiple Beans and Duplicate Qualifiers

Constructor Parameter Checks

1. **Check for Constructor/Setter Parameters and field:**:
    - If parameters/fields are annotated with `@Autowired`, check if their parent class is an interface or abstract class.
2. **Single Child Class**:
    - If the parent class has a single child class, no further checks are needed.
3. **Multiple Child Classes**:
    - If no child class is annotated with `@Service`, log a warning to annotate at least one child class with `@Service`.
4. **More Than One Child Class with @Service Annotation**:
    - Check for `@Primary` annotation:
        - **Single @Primary Annotation**: No further checks are needed.
        - **More Than One @Primary Annotation**: Warn the user to use only one `@Primary` annotation.
        - **No @Primary Annotation**: Log a warning to use `@Qualifier` annotation.

        - If `@Qualifier` is not found, warn the user to use `@Qualifier`.
        - If `@Qualifier` is found, check for the name:
            - **Name Missing**: Warn the user to specify the name.
            - **Name Found**: Check the validity of the name.
                - **Invalid Name**: Warn the user to use a valid name.
                - **Valid Name**: Check if the valid class is annotated with `@Service`.
                    - If not, warn the user to annotate the class with `@Service`.

Changes to Include Circular Dependency Check

G. Checking Circular Dependencies

Setup for Circular Dependency Detection

1. **Initialize Sets and Arrays**:
    - Create a set of visited beans to store already visited classes.
    - Create a set of initialised beans to store classes whose beans have been initialised.
    - Create an array `currentPath` to store the current traversal path.

Circular Dependency Detection Method

2. **Circular Dependency Check Method**:
    - For each class annotated with `@Service`:

        - If the class is already in `initializedBeans`, return as no cycle found.
        - If the class is already in `currentPath`, print the path and return that a cycle is found.
        - If the class is already in `visitedBeans`, return as no cycle is found.
        - If none of the above conditions are met, add the class to `visitedBeans` and `currentPath`.
        - Traverse through each dependency of the class (constructor parameters):
            - If any dependency is annotated with `@Lazy`, skip further checks for it.
            - For each dependency, use one of the two methods below:
                1. **Non-Abstract Class or Non-Interface Parent Class**:
                    - Call the recursive method for the dependency's class.
                2. **Abstract Class or Interface Parent Class**:
                    - Look for the class name used with the `@Qualifier` annotation.
                    - Recursively call the method for the class specified by the `@Qualifier` annotation.

Circular Dependency Detection Implementation

3. **Recursive Method**:
    - Define a recursive method to detect circular dependencies:
        - Check if the current class is in `initializedBeans`, `currentPath`, or `visitedBeans`.

        - Add the current class to `visitedBeans` and `currentPath`.
        - Traverse each dependency and call the recursive method as outlined above.
        - Remove the class from `currentPath` and add it to `initializedBeans` after processing all dependencies.

Check for Package Not Found Error

H. Checking for Package Not Found Error

Collecting Packages

1. **Collect All Relevant Classes**:
    - Create a set to store all classes that are `@Autowired`, `@Service`, and those used in `@Qualifier` annotations.

Extracting and Comparing Package Names

2. **Extract Package Names**:
    - For each class in the set, extract its package name.
3. **Compare Packages**:
    - Compare the set of packages used in the classes with the set of packages found in the `.xml` files.
    - Warn the user if a package is used but not found in the `.xml` package set.

Check for Rest API errors

I. Check for GET methods:

Traversing classes

1. **All Relevant Classes**:
    - Check if class is annotated with `@Component` and `@Path`.

Traverse GET methods in class

2. **Find GET methods**:
    - For each class, traverse through all the methods and check which are annotated with `@GET`.
3. **Validate parameters**:
    - Check each parameter of the method is annotated with any of the annotation below:

      `@QueryParam/@RequestParam/@PathVariable/@RequestBody/@ModelAttribute/@FormParam`

Plugin.xml file:

1. Basic description of plugin which contains id(unique identifier of a plugin),plugin name and its description.Since it is an intelliJ plugin in java, it depends on their modules.

```
<idea-plugin>

    <id>plugin-dev</id>

    <name>Pre deployment Scan</name>

    <vendor>Your</vendor>

    <description><![CDATA[
   To develop plugin to verify various issues before run time.
  ]]></description>

    <depends>com.intellij.modules.platform</depends>
    <depends>com.intellij.java</depends>
```

2. It also describes how the notification will be shown on the user's screen and extends

```
<extensions defaultExtensions="com.intellij">

    <notificationGroup id="annotationCheckerGroup"
                       displayType="BALLOON"
                       toolWindowId="ToolWindowId"
    />
</extensions>
```

3. This is the main part which describes the action class and where the plugin button will be available. For eg: **Here the button would be inside Tools and name of the button would be Validate Beans.**

```
    <actions>
        <group id="org.example.plugindev" text="My Plugin">
            <add-to-group group-id="ToolsMenu" anchor="last"/>
            <action class="org.example.plugindev.StartPlugin" id="Mydemo.Actions.plugindev" text="Validate Beans"
                    description="Checks for annotations like @Service,@Autowired and @Qualifier and
                    also detects cyclic dependency in bean initialisation"/>
        </group>
    </actions>
</idea-plugin>
```

Further Implementations

User Options for Plugin Execution

1. **Execution Scope**:
    - Provide users with the option to run the plugin on:
        - The whole project.
        - Selected files.
        - Changed files.

Handling @Qualifier Annotations with Custom Names

2. **GitLab MR pipeline**