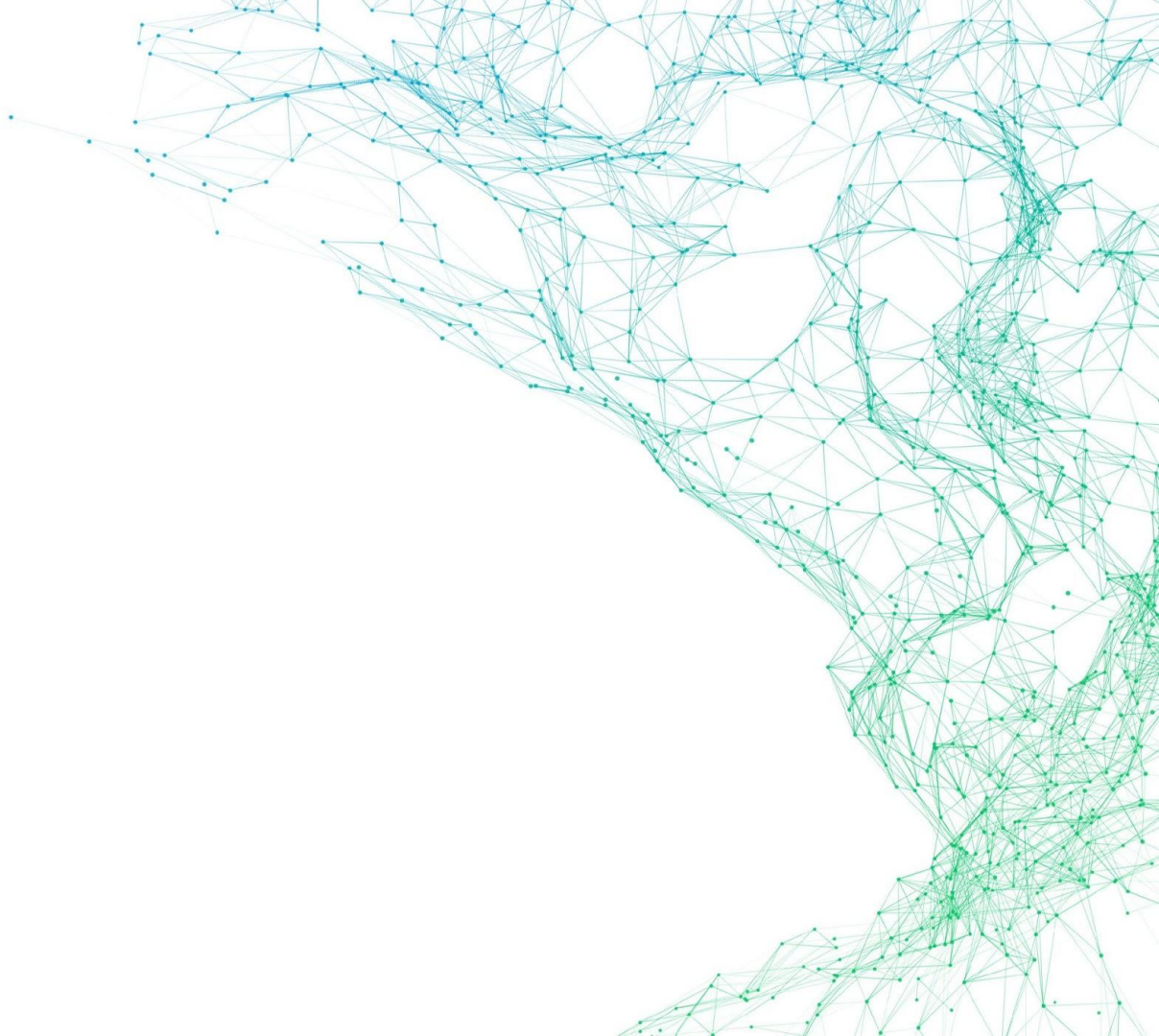


APPDYNAMICS

Hackathon

17. 04. 2023

Agenda

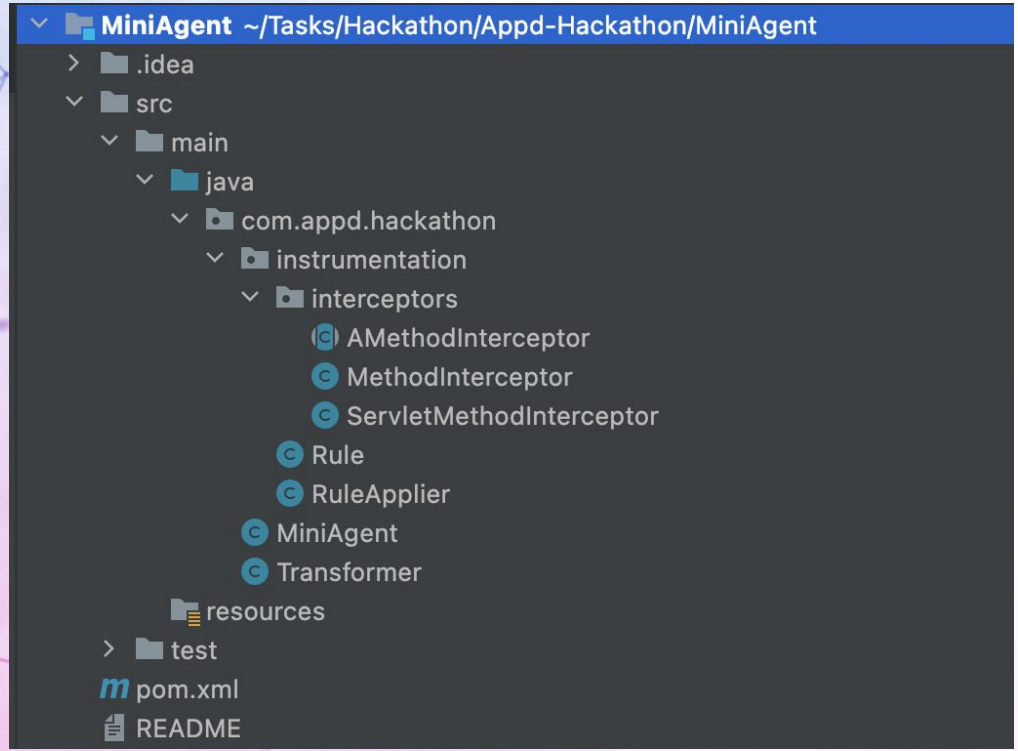
- Introduction
- Template description
- Hackathon requirements
- Time to work!

Introduction

Paweł Węglarczyk
Konrad Gębczyński
Dominik Zabłotny

Template description

Agent project structure



Template description

com.appd.hackathon.MiniAgent

```java
public class MiniAgent {
    public static void premain(String args, Instrumentation inst) throws Exception {
        System.out.println("--| Starting MiniAgent |--");
        inst.addTransformer(new Transformer());
    }
}
```

The premain is a mechanism associated with the java.lang.instrument package, used for loading "Agents" which make byte-code changes in Java programs.

Doc: https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html

Template description

com.appd.hackathon.Transformer/transform

```java
public class Transformer implements ClassFileTransformer {

    1 usage
    private final RuleApplier ruleApplier = new RuleApplier();

    @Override
    public byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
                            ProtectionDomain protectionDomain, byte[] classfileBuffer) throws IllegalClassFormatException {

        if (className == null) {
            return new byte[0];
        }

        try {
            List<Rule> rules = ruleApplier.matches(className);

            if (!rules.isEmpty()) {
                CtClass cc = getClass(className);

                for (Rule rule : rules) {
                    this.applyInterceptor(cc, rule);
                }
                return cc.toBytecode();
            }

        } catch (NotFoundException | RuntimeException | CannotCompileException | IOException e) {
            System.err.printf("%s class was not modified\nSome error exception occur: %s%n",
                    className, e.getMessage());
        }
        return new byte[0];
    }
```

Transformer/transform will be called for every new class definition and every class redefinition.

It's used in our case to find matching rules for loaded classes and apply bytecode agent instrumentation by executing **this.applyInterceptor**() method.

Template description

com.appd.hackathon.Transformer/applyInterceptor

```
private void applyInterceptor(CtClass cc, Rule rule) throws NotFoundException, CannotCompileException,
        IOException {
    String interceptorClass = rule.getInterceptor().getCanonicalName();
    String interceptorClassVariable = "interceptorClassVariable_" + new Random().nextInt(Integer.MAX_VALUE);
    CtMethod cm = cc.getDeclaredMethod(rule.getMethod());

    System.out.printf("Applying interceptor: %s on: %n",interceptorClass);
    System.out.printf(" cc: %s ", cc.getName());
    System.out.printf("cm: %s ", cm.getName());

    CtField f = CtField.make( src: "private " + interceptorClass + " " + interceptorClassVariable + ";", cc);
    cc.addField(f);


    String insertBefore = String.format("this.%s = new %s(); this.%s.onMethodBegin($0, \"%s\", \"%s\", $args); ",
            interceptorClassVariable, interceptorClass, interceptorClassVariable, cc.getName(), cm.getName());
    System.out.println(insertBefore);
    cm.insertBefore(insertBefore);

    String insertAfter = String.format("this.%s.onMethodEnd($0, \"%s\", \"%s\", $args,$_); ",
            interceptorClassVariable, cc.getName(), cm.getName());
    System.out.println(insertAfter);

    cm.insertAfter(insertAfter);
}
```

applyInterceptor method applies interceptor by performing the following.

- Obtains the declared method with the CtClass object usage.
- Creates a new field for interceptor called *interceptorClassVariable_<generated_Id>* to the method.
- Adds to the method begin the execution of agent interceptor onMethodBegin() method - content of insertBefore variable.
- Adds to the method end the execution of agent interceptor onMethodEnd() method - content of insertAfter variable.

Template description

Transformation result example

```
1  public abstract class HttpServlet extends GenericServlet {
2
3  ...
4  private com.appd.hackathon.instrumentation.interceptors.ServletMethodInterceptor interceptorClassVariable_123;
5  ...
6
7      protected void service(HttpServletRequest req, HttpServletResponse resp)
8          throws ServletException, IOException
9      {
10         // Content of insertBefore
11         {
12             this.interceptorClassVariable_123 = new %s();
13             this.interceptorClassVariable_123.onMethodBegin(this, "javax.servlet.http.HttpServlet", "service", paramValues)
14         }
15
16         ... // Originating service() method body
17
18         // Content of insertAfter
19         {
20             this.interceptorClassVariable_123.onMethodEnd(this, "javax.servlet.http.HttpServlet", "service", paramValues, returnValue)
21         }
22
23     }
24 ...
25 }
26
```

Template description

com.appd.hackathon.instrumentation.RuleApplied

```java
public class RuleApplier {

    3 usages
    private List<Rule> rules = new ArrayList<>();

    //Servlet rule
    1 usage
    private static String SERVLET_CLASS = "javax.servlet.http.HttpServlet";
    1 usage
    private static String SERVLET_METHOD = "service";
    1 usage
    private static Class<? extends AMethodInterceptor> SERVLET_INTERCEPTOR = ServletMethodInterceptor.class;

    1 usage
    public RuleApplier() { this.generateRules(); }

    1 usage
    public void generateRules() {
        {
            Rule rule = new Rule(SERVLET_CLASS, SERVLET_METHOD, SERVLET_INTERCEPTOR);
            this.rules.add(rule);
            System.out.println(this.rules.toString());
        }
    }

    1 usage
    public List<Rule> matches(String className) {
        return rules.stream()
                .filter(r -> r.getCl()
                .equals(className.replace( target: "/", replacement: ".")))
                .collect(Collectors.toList());
    }
}
```

RuleApplier is responsible for storing instrumentations rules and finding match rules for the loaded classes.

You can add new rules to the list in order to perform instrumentation task like:

- Measuring method execution time
- Reading HTTP parameters

Template description

Compile and instrument Tomcat

To compile project use the below command (requires Maven):

```
> mvn package
```

Add the packaged Agent JAR (with dependencies) to Tomcat startup script.

For Windows:

```
<package>/apache-tomcat-9.0.73/bin/catalina.bat
```

Add the following line:

```
set CATALINA_OPTS="%CATALINA_OPTS% -javaagent:<drive>:\<package>\MiniAgent-1.0-with-dependencies.jar"
```

For Linux:

```
<package>/apache-tomcat-9.0.73/bin/catalina.sh
```

Add the following line:

```
CATALINA_OPTS="$CATALINA_OPTS -javaagent:<package>/MiniAgent-1.0-with-dependencies.jar"
```

Run Tomcat by executing the `catalina.sh`/`catalina.bat` script.

Hackathon requirements

- Measure the method execution time in milliseconds.
- Aggregate the method execution time metric based on the Servlet's URI during runtime.
- Collect HTTP headers.
- Collect method invocation parameters and its types.
- Log the above-mentioned metrics/values along with the time when the metric was collected.
- (optional) HTTP correlation by injecting agent header to the HTTP request object and reading it on a downstream application
- (optional) Plot method execution time depending on time

Feel free to modify example agent concept code.

Hackathon

Agent Template, Tomcat and Presentation available on:

tinyurl.com/5xyumnc2

Leader sends result to the email: **kgebczyn@cisco.com**

- Achieved requirements list
- zip of agent source code
- Compiled agent jar with dependencies