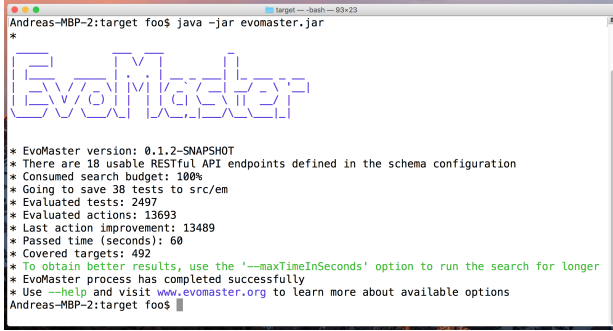


# EvoMaster: Evolutionary Multi-context Automated System Test Generation

Andrea Arcuri  
Westerdals Oslo ACT, Oslo, Norway  
and SnT, University of Luxembourg, Luxembourg  
Email: arcand@westerdals.no



```
Andreas-MBP-2:target foo$ java -jar evomaster.jar
*
EvoMaster

* EvoMaster version: 0.1.2-SNAPSHOT
* There are 18 usable RESTful API endpoints defined in the schema configuration
* Consumed search budget: 100%
* Going to save 38 tests to src/em
* Evaluated tests: 2497
* Evaluated actions: 13693
* Last action improvement: 13489
* Passed time (seconds): 60
* Covered targets: 492
* To obtain better results, use the '--maxTimeInSeconds' option to run the search for longer
* EvoMaster process has completed successfully
* Use --help and visit www.evomaster.org to learn more about available options
Andreas-MBP-2:target foo$
```

Fig. 1. Usage of EVOMASTER from command terminal.

**Abstract**---This paper presents EVOMASTER, an open-source tool that is able to automatically generate system level test cases using evolutionary algorithms. Currently, EVOMASTER targets RESTful web services running on JVM technology, and has been used to find several faults in existing open-source projects. We discuss some of the architectural decisions made for its implementation, and future work.

**Keywords:** REST, SBSE, SBST, SOA, Microservice, Web Service, Test Generation

## I. INTRODUCTION

There exist different tools that can automatically generate unit tests, using variants of random testing (e.g., Randoop [1]), evolutionary search (e.g., EvoSuite [2]) or dynamic symbolic execution (e.g., Pex/IntelliTest [3]). For smartphone applications [4], there are tools like Sapienz [5] that can generate sequences of events on the GUI. For web applications serving HTML pages, there are web crawler tools like Crawljax [6]. These crawlers can be used for testing of web applications, but they are black-box, and do not take into account the internal details of the server side code. Furthermore, little exists that is *available* (i.e., a tool that can be downloaded and used) for *white-box* system testing of enterprise applications [7], in particular RESTful web services.

This paper introduces EVOMASTER, a new tool that aims at test generation at system level using evolutionary techniques, in particular the MIO algorithm [8]. At the current stage, EVOMASTER targets RESTful APIs [9], [10] running on JVMs. However, EVOMASTER is architected in a way in which it can be extended for other languages and other system test contexts.

```
@Test
public void test12() throws Exception {

    String location_activities = "";

    String id_0 = given().accept("*/")
        .header("Authorization", "ApiKey administrator") // administrator
        .contentType("application/json")
        .body("{\"name\":\"Jig\",
            \"date_updated\":\"1968-7-28T10:40:58.000Z\",
            \"description_material\":\"CDasIs\",
            \"description_prepare\":\"VatRg\",
            \"description_main\":\"vbhUS\",
            \"description_safety\":\"mdMZKHw6Ac0L7\",
            \"age_min\":-1639552914,
            \"age_max\":-546,
            \"participants_min\":-166,
            \"time_max\":-728,
            \"featured\":true,
            \"activity\":{\"ratings_sum\":-2169794882535544017,
                \"favourites_count\":2018287764382358555,
                \"ratings_average\":0.7005221066369205,
                \"related\":{\"5230990194698818394,
                    4025421724722458028,
                    -1291838056,
                    -210322044}}}")
        .post(baseUrlOfSut + "/api/v1/activities")
        .then()
        .statusCode(200)
        .extract().body().path("id").toString();

    location_activities = "/api/v1/activities/" + id_0;

    given().accept("*/")
        .header("Authorization", "ApiKey administrator") // administrator
        .contentType("application/json")
        .body("{\"rating\":\"7126434\", \"favourite\":false}")
        .post(resolveLocation(location_activities,
            baseUrlOfSut + "/api/v1/activities/-324163273/rating"))
        .then()
        .statusCode(204);

    given().accept("*/")
        .header("Authorization", "ApiKey administrator") // administrator
        .delete(resolveLocation(location_activities,
            baseUrlOfSut + "/api/v1/activities/-324163273/rating"))
        .then()
        .statusCode(204);
}
```

Fig. 2. Example of test generated by EVOMASTER.

Modern web applications often rely on external *web services*. Large and complex enterprise applications can be split into individual web service components, in what is typically called a *microservice* architecture [11]. The assumption is that individual components are easier to develop and maintain compared to a large monolithic application. The use of microservice applications is a very common practice in industry, done for example in companies like Netflix, Uber, Airbnb, eBay, Amazon, Twitter, Nike, etc [12].

Besides being used internally in many enterprise applications, there are many web services available on the Internet. Websites like *ProgrammableWeb*<sup>1</sup> currently list more than 16 thousand Web APIs. Many companies provide APIs to their tools and

<sup>1</sup><https://www.programmableweb.com/api-research>

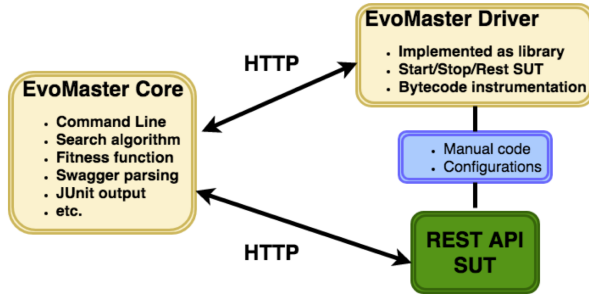


Fig. 3. High level architecture of EVOMASTER.

services using REST, which is currently the most common type of web service, like for example Google<sup>2</sup>, Amazon<sup>3</sup>, Twitter<sup>4</sup>, Reddit<sup>5</sup>, LinkedIn<sup>6</sup>, etc.

Testing web services, and in particular RESTful web services, does pose many challenges [13], [14]. Different techniques have been proposed. However, most of the work so far in the literature has been concentrating on black-box testing of SOAP web services, and not REST [15].

Figure 1 shows a use of EVOMASTER from command terminal, whereas Figure 2 shows an example of generated test in Java using the highly popular RestAssured<sup>7</sup> library (which helps in writing tests that require HTTP calls). Automatically generating tests for RESTful APIs is a complex task, because a test might require several HTTP calls. Each HTTP call might require to set up the right URL (path and query parameters), HTTP headers and an HTTP payload body. This latter can be particularly complex, as the RESTful API could take as input any arbitrary kind of data (usually in JSON or XML format). Furthermore, a HTTP call might require data from the output of a previous HTTP call. This is a typical example when a resource is created on the server with a HTTP POST request, and then the returned id of this resource is needed to have a GET request on such newly generated resource. A tool aiming at generating this kind of tests needs to be able to handle all of these cases.

Although EVOMASTER is still in an early phase of development (it was started in the late 2016), it has already been used to successfully find several bugs in existing open-source projects and in an industrial application [15]. EVOMASTER is released under the LGPL open-source license, and it is freely accessible on GitHub<sup>8</sup>.

## II. TOOL IMPLEMENTATION

EVOMASTER is composed of two main components: a *core* process responsible for the main functionalities (e.g., command-line parsing, search and generation of test files), and a *driver*

process. This latter is responsible to start/stop/reset the system under test (SUT) and instrument its source code, e.g., via automated bytecode manipulation, in a similar way of how unit test tools like EvoSuite [2] do. For example, you need to add probes in the bytecode to check which statements are executed, and also to define heuristics to help solving the predicates in the branch statements (e.g., the so called *branch distance* [16]). Such test execution information is then exported by the *driver* module (in JSON format) and used by the *core* process to generate new test cases. Figure 3 shows a high level overview of EVOMASTER’s architecture.

EVOMASTER implements different kinds of search algorithms for test suite generation (e.g., WTS [17] and MOSA [18]), where MIO [8] is the default one. EVOMASTER generates test suites with the goal of optimising white-box, code coverage metrics (e.g., statement and branch coverage) and fault detection (e.g., HTTP 5xx status codes can be used in some cases as automated oracles). Each test will be composed of one or more HTTP calls. The generated test files (e.g., using JUnit<sup>9</sup> and RestAssured<sup>10</sup> libraries) are self-contained, as using the EVOMASTER driver as a library to automatically start the SUT before running the tests (e.g., in JUnit this can be done in a `@BeforeClass` init method).

The *core* process of EVOMASTER is written in Kotlin, a new language that can compile into JVM bytecode. The choice of Kotlin was due to the fact that we consider it as the best language for developing tools like EVOMASTER. On the other hand, the drivers need to be implemented based on the target language of the SUT. Currently, we provide a driver only for JVM languages (e.g., Java and Kotlin). Adding support for a new language (e.g., C#) does not require any change in the *core* process, as communications between *core* and *driver* are program language agnostic (e.g., JSON over HTTP).

To use EVOMASTER on a given SUT, a test engineer has to provide in a configuration class some basic information, like for example where to find the SUT’s executable (e.g., a uber jar) and on which TCP port the started RESTful service will listen on. This is discussed in more details in the next section.

## III. MANUAL PREPARATIONS

In contrast to tools for unit testing like EvoSuite, which are 100% fully automated (a user just need to select in their IDE for which classes the tests should be generated), our tool EVOMASTER for system/integration testing of RESTful APIs does require some manual configuration. This is not a limitation of the tool, but rather one of challenges of system-level testing.

The developers of the RESTful APIs need to import our library (published on Maven Central Repository<sup>11</sup>), and then create a class that extends the *EmbeddedSutController* class in such library. The developers will be responsible to define how the SUT should be started, where the Swagger schema can be found (which defines what present in the API), which packages should be instrumented, etc. This will of course vary

<sup>2</sup><https://developers.google.com/drive/v2/reference/>

<sup>3</sup><http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html>

<sup>4</sup><https://dev.twitter.com/rest/public>

<sup>5</sup><https://www.reddit.com/dev/api/>

<sup>6</sup><https://developer.linkedin.com/docs/rest-api>

<sup>7</sup><https://github.com/rest-assured/rest-assured>

<sup>8</sup><https://github.com/EMResearch/EvoMaster>

<sup>9</sup><http://junit.org/junit4/>

<sup>10</sup><https://github.com/rest-assured/rest-assured>

<sup>11</sup><https://mvnrepository.com/artifact/org.evomaster/evomaster-client-java>

Fig. 4. Example of class that needs to be implemented by the developers of the SUT to enable the usage of our test case generation tool. In this particular case, the SUT is written with SpringBoot, where *Application* is the main entry point of the SUT.

```
public class EMController extends EmbeddedSutController {

    private ConfigurableApplicationContext ctx;
    private final int port;
    private Connection connection;

    public static void main(String[] args) {

        EMController controller = new EMController(port);
        InstrumentedSutStarter starter = new InstrumentedSutStarter(controller);

        starter.start();

    }

    public EMController(){this(0);}

    public EMController(int port) {
        this.port = port;
    }

    @Override public int getControllerPort(){
        return port;
    }

    @Override public String startSut() {

        ctx = SpringApplication.run(Application.class,
            new String[]{"--server.port="+port});

        if(connection != null){
            try { connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        JdbcTemplate jdbc = ctx.getBean(
            JdbcTemplate.class);

        try {
            connection = jdbc.getDataSource()
                .getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return "http://localhost:"+getSutPort();
    }

    protected int getSutPort(){
        return (Integer)((Map) ctx.getEnvironment()
            .getPropertySources().get("server.ports")
            .getSource()).get("local.server.port");
    }

    @Override public boolean isSutRunning() {
        return ctx!=null && ctx.isRunning();
    }

    @Override public void stopSut() { ctx.stop();}

    @Override public String getPackagePrefixesToCover() {
        return "org.javiermf.features.";
    }

    @Override public void resetStateOfSUT() {
        ScriptUtils.executeSqlScript(connection,
            new ClassPathResource("empty-db.sql"));
        ScriptUtils.executeSqlScript(connection,
            new ClassPathResource("data-test.sql"));
    }

    @Override public String getUriOfSwaggerJSON() {
        return "http://localhost:"+getSutPort()+
            "/swagger.json";
    }

    @Override public List<AuthenticationDto>
        getInfoForAuthentication(){
        return null;
    }

}
```

based on how the RESTful API is implemented, e.g., if with Spring<sup>12</sup>, DropWizard<sup>13</sup>, Play<sup>14</sup>, Spark<sup>15</sup> or Java EE.

Figure 4 shows an example of one such class we had to write for one of the SUTs in our empirical studies. That SUT uses SpringBoot. That class is quite small, and needs to be

written only once. It does not need to be updated when there are changes internally in the API. The code in the superclass *EmbeddedSutController* will be responsible to do the automatic bytecode instrumentation of the SUT, and it will also start a RESTful service to enable our testing tool to remotely call the methods of such class.

However, besides starting/stopping the SUT and providing other information (e.g., location of the Swagger file), there are two further tasks the developers need to perform:

- RESTful APIs are supposed to be stateless (so they can easily scale horizontally), but they can have side effects on external actors, such as a database. In such cases, before each test execution, we need to reset the state of the SUT environment. This needs to be implemented inside the *resetStateOfSUT()* method. In the particular case of the class in Figure 4, two SQL scripts are executed: one to empty the database, and one to fill it with some existing values. We did not need to write those scripts by ourself, as we simply re-used the ones already available in the manually written tests in that SUT. How to automatically generate such scripts would be an important topic for future investigations.
- If a RESTful API requires some sort of authentication and authorization, such information has to be provided by the developers in the *getInfoForAuthentication()* method. For example, even if a testing tool would have full access to the database storing the passwords for each user, it would not be possible to reverse engineer those passwords from the stored hash values. Given a set of valid credentials, the testing tool will use them as any other variable in the test cases, e.g., to do HTTP calls with and without authentication.

Once such class is implemented, it needs to be run as a process (see its *main* method). This can be easily done in an IDE like IntelliJ/Eclipse by right-clicking on it. Once this driver process is started, it will open a listening TCP port. We can then start the EVOMASTER executable from a command terminal (e.g., recall Figure 1), which will connect to the driver process via TCP, and start generating test cases. The documentation of EVOMASTER at [www.evomaster.org](http://www.evomaster.org) provides links to videos on how to do these steps.

To enable researchers to use EVOMASTER in their experiments, we have provided on GitHub<sup>16</sup> a set of open-source projects for which we maintain the EVOMASTER driver classes needed to use it. Note: as the *driver* modules provide test execution information and heuristics independently from the *core* process, such drivers can also be used in other system testing tools besides EVOMASTER. This is of particular importance, as writing a bytecode manipulation library is a complex task.

Besides *EmbeddedSutController*, users have also the option of rather extending the *ExternalSutController* class. This latter case is to handle situations in which it is not easy, or even possible, to start a web service directly from a class (e.g.,

<sup>12</sup><https://github.com/spring-projects/spring-framework>

<sup>13</sup><https://github.com/dropwizard/dropwizard>

<sup>14</sup><https://github.com/playframework/playframework>

<sup>15</sup><https://github.com/perwendel/spark>

<sup>16</sup><https://github.com/EMResearch/EMB>

Java EE). To handle these cases, we enable the option to start the SUT on a separate, external process from the driver one, instead of running the SUT embedded in the same process of the driver. To do so, we need the SUT to be packaged in a self-executable jar file. The EVOMASTER driver library will *automatically* handle all the necessary technical details on how to start/stop such process, enable JavaAgents, and collect statistics from these spawn processes.

#### IV. CONFIGURATIONS

EVOMASTER has several configurations, which can be set with command line options. For a practitioner, the main options are:

- `--help` : List all available options.
- `--maxTimeInSeconds <Int>` : Maximum number of seconds allowed for the search. The more time is allowed, the better results one can expect. But then the test generation will take longer.
- `--outputFolder <String>` : The path directory of where the generated test classes should be saved to.
- `--outputFormat <OutputFormat>` : Specify in which format the tests should be outputted. For example, `JAVA_JUNIT_5` or `JAVA_JUNIT_4`.
- `--testSuiteFileName <String>` : The name of the generated file with the test cases.

All options provide sensible default values. For example, by default the search lasts one minute.

For researchers, most of the internal settings of the search algorithms (e.g., population size) can be configured via command line options, like the different parameters used in the MIO algorithm [8].

#### V. CURRENT RESULTS

EVOMASTER was evaluated in [15] on three different RESTful APIs: two open-source, and one from our industrial partners. These APIs were between 2 and 10 thousand lines of Java code.

On such APIs, EVOMASTER found 38 unique bugs, where HTTP calls were generated in a way in which 5xx (server error, internal crash) HTTP codes were returned by the SUT responses. However, on such SUTs the statement code coverage was only between 20% and 40%. One main reason is that these SUTs (and RESTful APIs in general) interact with databases. Supporting databases in search-based software testing (e.g., heuristics based on the results of the SQL queries) is one of current main activities in the EVOMASTER development.

#### VI. CONCLUSION

In this paper, we have presented EVOMASTER, a new tool that aims at generating white-box, system-level test cases for enterprise/web applications. This type of systems are very common in industry. But, in contrast to unit and mobile testing, to the best of our knowledge there is no available existing *white-box* tool that addresses enterprise/web applications.

Internally, EVOMASTER uses evolutionary techniques, like the MIO algorithm [8]. Currently, EVOMASTER does target

RESTful APIs, but it is architected in a way in which it will be easily extended to other contexts. For example, the bytecode instrumentation is released as a library on Maven Central Repository, and can be integrated in other tools.

This paper describes some of the technical details of EVOMASTER, current results (e.g. bugs found in existing APIs) and future work (supporting SQL databases). To enable technology transfer from academic research to industrial practice, EVOMASTER is released with a permissive open-source license (LGPL v3.0), and published on GitHub. To learn more about EVOMASTER, visit our webpage at: [www.evomaster.org](http://www.evomaster.org)

#### ACKNOWLEDGMENT

This work is supported by the National Research Fund, Luxembourg (FNR/P10/03).

#### REFERENCES

- [1] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2007, pp. 75–84.
- [2] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2011, pp. 416–419.
- [3] N. Tillmann and N. J. de Halleux, "Pex --- white box test generation for .NET," in *International Conference on Tests And Proofs (TAP)*, 2008, pp. 134–253.
- [4] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet?" in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.
- [5] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, pp. 94–105.
- [6] A. Mesbah, A. Van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *TSE*, vol. 38, no. 1, pp. 35–53, 2012.
- [7] A. Arcuri, "An experience report on applying software testing academic results in industry: we need usable automated test generation," *Empirical Software Engineering (EMSE)*, pp. 1–23, 2018.
- [8] -----, "Many Independent Objective (MIO) Algorithm for Test Suite Generation," in *International Symposium on Search Based Software Engineering (SSBSE)*, 2017, pp. 3–17.
- [9] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.
- [10] S. Allamaraju, *Restful web services cookbook: solutions for improving scalability and simplicity*. " O'Reilly Media, Inc.", 2010.
- [11] S. Newman, *Building Microservices*. " O'Reilly Media, Inc.", 2015.
- [12] R. Rajesh, *Spring Microservices*. Packt Publishing Ltd, 2016.
- [13] G. Canfora and M. Di Penta, "Service-oriented architectures testing: A survey," in *Software Engineering*. Springer, 2009, pp. 78–105.
- [14] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing and verification in service-oriented architecture: a survey," *Software Testing, Verification and Reliability (STVR)*, vol. 23, no. 4, pp. 261–313, 2013.
- [15] A. Arcuri, "Restful api automated test case generation," in *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 9–20.
- [16] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.
- [17] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.
- [18] A. Panichella, F. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering (TSE)*, 2017.