





EMB: A Curated Corpus of Web/Enterprise Applications And Library Support for Software Testing Research

Andrea Arcuri^{*†}, Man Zhang^{*†}, Amid Golmohammadi^{*†}, Asma Belhadi^{*†},

Juan P. Galeotti^{‡§}, Bogdan Marculescu^{*†}, Susruthan Seran^{*†}

^{*}Kristiania University College, Oslo, Norway

[†]OsloMet, Oslo, Norway

[‡]University of Buenos Aires, Buenos Aires, Argentina

[§]CONICET, Argentina

Abstract—Web Services like REST, GraphQL and RPC APIs are widely used in industry. They form the backends of modern Cloud Applications. In recent years, there has been an increase interest in the research community about fuzzing web services. However, there is no clear, common benchmark in the literature that can be used for comparing techniques and ease experimentation. Even if nowadays it is not so difficult to find web services on open-source repositories such as GitHub, quite a bit of work might be required to setup databases and authentication information (e.g., hashed passwords). Furthermore, how to start and stop the applications might vary greatly among the different frameworks (e.g., Spring and DropWizard) used to implement such services. For all these reasons, since 2017 we have created and maintained a corpus of web services called EMB, together with all the tooling and configurations needed to run software testing experiments. Originally, EMB was created for evaluating the fuzzer EVOMASTER, but it can be (and has been) used by other tools/researchers as well. This paper discusses how EMB is designed and how its libraries can be used to run experiments on these APIs. An introductory video for EMB can be currently accessed at <https://youtu.be/wJs34ATgLEw>

Index Terms—Benchmark, REST, GraphQL, RPC, API, Fuzzing, Test Generation

I. INTRODUCTION

Writing test cases by hand is costly and time consuming, especially when dealing with web/enterprise applications [1]–[3]. The backends of Cloud Applications are often developed with Web Services, including REST [4], GraphQL [5] and RPC (e.g., gRPC [6]) APIs. Considering their wide use in industry [7], [8], there has been a recent increase of interest in the research community about how to design new techniques to automatically test these Web APIs.

In the recent years, many techniques and tools have been proposed to fuzz RESTful APIs, including (in alphabetic order) for example bBOXRT [9], EVOMASTER [10], RestCT [11], RESTest [12], Restler [13], RestTestGen [14], and Schemathesis [15]. However, less attention in the literature

has been spent on the fuzzing of other kinds of web services, like GraphQL [16]–[20] and RPC [21] APIs.

We are the authors of EVOMASTER [10], [22], a search-based fuzzer which supports all these kinds of web services, i.e., REST [23], GraphQL [20] and RPC [21] APIs. EVOMASTER supports both white-box and black-box testing [20], [24]. As far as we know, among the different tools in the literature, it seems that EVOMASTER is still the only one that supports white-box testing (i.e., all the other tools are black-box fuzzers).

Since its inception as an open-source project in 2016 [25], one challenge has been how to find suitable APIs for experimentation. In our first published work on EVOMASTER in 2017 [26], we used only 3 APIs: *features-service* and *scout-api* from GitHub, and an API from one of our industrial partners. At that time, finding RESTful APIs in open-source repositories was not trivial, as all the search functionality we can access today were not available at that time (e.g., to search for projects using specific libraries). As we were doing experiments on white-box testing, which requires code analyses, we focused on one programming language only, i.e., Java. This restricted what could be used for experimentation (e.g., no API written in Python nor Ruby). There were several potential APIs that could be selected for experimentation, but the large majority of them had either no documentation, would not compile or required a lot of manual effort to just get them started (e.g., build WAR files to manually upload to a JEE container like JBoss).

For doing white-box experiments, and to enable the generation of actual test cases (e.g., in JUnit format) that could be used as well for *regression testing* [27], there is a need to be able to programmatically *start*, *stop* and *reset* (e.g., the state in a SQL database) the tested APIs. For each system under test (SUT) used in our experiments, we wrote *driver classes* to carry out those tasks programmatically (discussed in more details in Section IV). These classes also have to provide all the necessary information for the fuzzers to effectively test the APIs, like for example any authentication information (e.g., user-names and passwords) if required. Writing these driver

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research, innovation programme (grant agreement No 864972), and partially by UBACyT 2020 20020190100233BA and PICT-2019-01793.

classes require some manual effort, but for other researchers and practitioners it seems not a too complicated task (e.g., [28], [29]). This is particularly the case when practitioners use fuzzers like EVOMASTER in industry on large microservice architectures with hundreds of distinct web services, like for example at Meituan [21] (a large e-commerce enterprise with more than 600 million customers). As often the web services used in a microservice architecture share the same technologies, once a driver class is written for one web service, it can be trivial to do the same for all the others [21].

Finding and preparing SUTs for experiments on web services is very, very different from what we did for example when preparing experiments on *unit testing* with EvoSuite [30]. When we prepared the SF110 corpus [31], we simply took 100 projects at random, plus the 10 most popular projects (at that time, the main open-source repository was SourceForge). For experiments on unit testing we only needed to be able to *compile* those projects, and the projects could be of any kind. There was no need to be able to *run* those applications. For example, when doing *system testing* of a RESTful API, if this API is using a SQL database such as Postgres, this needs to be up and running. Fortunately, nowadays this can be fully automated with tools like Docker [32] and libraries like Testcontainers [33] (and this is something we do in our driver classes).

For all these reasons, considering the issues in finding and preparing suitable SUTs for experimentation in fuzzing web services, since 2017 we have collected all our SUTs and driver classes for them in a single GitHub repository, called EMB [34]. The driver classes rely on some functionality and utilities that we have published as a library on Maven Central (for JVM languages) and NPM (for NodeJS languages). Each year, we have added new SUTs to EMB, to provide a larger and more variegated corpus of SUTs for experimentation, which we have used to better evaluate novel techniques we designed in EVOMASTER (e.g., for SQL handling [35] and *adaptive hypermutation* [36]).

Originally, EMB stood for “EVOMASTER Benchmark”, as it was prepared to support our experiments with EVOMASTER. But, our driver classes can be technically used by any fuzzer, e.g., to programmatically setup all experiment configurations, and to be able to generate self-contained test cases that can be used for regression testing. For example, EVOMASTER uses these driver classes in its generated test suite files, where (1) the SUT is *started* before any test case is run (e.g., in a JUnit `@BeforeAll` function); (2) it is *reset* before each test case execution; and (3) finally *stopped* once all test cases in the test suite are executed.

Besides being used by our group for experiments on EVOMASTER, also other research groups have used EMB for their studies. These for example include extensions of EVOMASTER (e.g., [37], [38]), as well as being used for comparisons of different fuzzers (e.g., [28]).

This paper provides the following contributions:

- We present EMB, a curated corpus of web services that can be used for experimentation in software testing

research. It has been in use already for few years, since 2017, extended each year with new SUTs.

- We provide implementation of driver classes (involving thousands of lines of code), together with library support and explanation on how to use them, to enable other fuzzers besides EVOMASTER to generate self-contained test cases.

An introductory video for EMB can be currently accessed at <https://youtu.be/wJs34ATgLEw>

II. RELATED WORK

In the literature, there have been already a few cases of collecting different artifacts for easing experimentation and improve replicability of experiments in software testing research.

One of the earliest cases in the 90s was the Siemens Benchmark Suite [39] for testing C programs. This included 7 small C programs, up to 512 LOCs.

A decade later, the Software-Artifact Infrastructure Repository (SIR) [40] provided not only C/Java programs, but also different versions of these programs having known, cataloged faults. It included 81 subjects.

Another decade later, a very popular collection of Java programs with known, cataloged faults was Defect4J [41]. It included 5 Java libraries with 357 faults. For experiments on unit test generation for Java programs, there is also the SF110 corpus [31] we discussed in the introduction.

More recent examples of curated software corpora for software testing experimentation in the last few years include FuzzBench [42] for libraries/parsers, GBGallery [43] for video games, and BugsJS [44] for JavaScript programs. But there are more, e.g., Artho et al. [45] provide a survey of 23 existing benchmark projects.

As far as we know, there is no other dedicated corpus in the literature for web services. As already discussed in the introduction, it is not just a matter of collecting such APIs, but also provide all the configurations needed to build, start, stop and reset them.

Note that, in recent years, it has become a common practice to provide *replication packages* when publishing research articles. Those replication packages can include the SUTs used in the experiments. For papers published on fuzzing RESTful APIs, for example this was the case for [28], [46]. Replication packages are very useful, and can be a good starting point for new research efforts (e.g., for collecting artifacts for experimentation). However, there is quite a difference in size and scope between a replication package aimed at replicating a single study and a curated corpus, with documentation, examples and library support aimed at being used by different researchers with different tools.

III. APPLICATIONS IN EMB

At the time of this writing, EMB contains 20 RESTful APIs (14 for the JVM and 6 for NodeJS), 7 GraphQL APIs (5 for JVM and 2 for NodeJS), and 2 RPC APIs (for the JVM). There are also some APIs for C#, but their support is not

TABLE I: Statistics on the 29 APIs currently in EMB. We report their name, type, programming language, number of source-code files (not including any third-party library), their number of lines (LOCs), if they use any database, and the URL of their original repository they were took from.

SUT	Type	Language	#Files	#LOCs	Database	URL
<i>timbuctoo</i>	GraphQL	Java	1113	107729	Neo4j	https://github.com/HuygensING/timbuctoo
<i>patio-api</i>	GraphQL	Java	178	18048	PostgreSQL	https://github.com/patio-team/patio-api
<i>petclinic-graphql</i>	GraphQL	Java	89	5212	PostgreSQL	https://github.com/spring-petclinic/spring-petclinic-graphql
<i>graphql-scs</i>	GraphQL	Kotlin	13	577	-	-
<i>graphql-ncs</i>	GraphQL	Kotlin	8	548	-	-
<i>react-finland</i>	GraphQL	TypeScript	461	16206	-	https://github.com/ReactFinland/graphql-api
<i>ecommerce-server</i>	GraphQL	TypeScript	49	1815	PostgreSQL	https://github.com/react-shop/react-ecommerce
<i>languagetool</i>	REST	Java	1385	174781	-	https://github.com/languagetool-org/languagetool
<i>ocvn-rest</i>	REST	Java	526	45521	H2, MongoDB	https://github.com/devgateway/ocvn
<i>genome-nexus</i>	REST	Java	405	30004	MongoDB	https://github.com/genome-nexus/genome-nexus
<i>market</i>	REST	Java	124	9861	H2	https://github.com/aleksey-lukyanets/market
<i>scout-api</i>	REST	Java	93	9736	H2	https://github.com/mikaelsvensson/scout-api
<i>catwatch</i>	REST	Java	106	9636	H2	https://github.com/zalando-incubator/catwatch
<i>proxyprint</i>	REST	Java	73	8338	H2	https://github.com/ProxyPrint/proxyprint-kitchen
<i>cwa-verification</i>	REST	Java	47	3955	H2	https://github.com/corona-warn-app/cwa-verification-server
<i>gestaohospital-rest</i>	REST	Java	33	3506	MongoDB	https://github.com/ValchanOfficial/Gestaohospital
<i>features-service</i>	REST	Java	39	2275	H2	https://github.com/JavierMF/features-service
<i>restcountries</i>	REST	Java	24	1977	-	https://github.com/apilayer/restcountries
<i>rest-scs</i>	REST	Java	13	862	-	-
<i>rest-ncs</i>	REST	Java	9	605	-	-
<i>cyclotron</i>	REST	JavaScript	25	5803	MongoDB	https://github.com/ExpediaInceCommercePlatform/cyclotron
<i>spacex-api</i>	REST	JavaScript	63	4966	MongoDB	https://github.com/r-spacex/SpaceX-API
<i>disease-sh-api</i>	REST	JavaScript	57	3343	Redis	https://github.com/disease-sh/API
<i>js-rest-scs</i>	REST	JavaScript	13	1046	-	-
<i>js-rest-ncs</i>	REST	JavaScript	8	775	-	-
<i>rest-news</i>	REST	Kotlin	11	857	H2	https://github.com/arcuri82/testing_security_development_enterprise_systems
<i>realworld-app</i>	REST	TypeScript	37	1229	MySQL	https://github.com/lujakob/nestjs-realworld-example-app
<i>rpc-thrift-scs</i>	RPC-Thrift	Java	14	772	-	-
<i>rpc-thrift-ncs</i>	RPC-Thrift	Java	9	585	-	-
Total 29			5025	470568	18	

fully completed, and so they will not be discussed further in this paper. Table I shows some statistics on those APIs.

The projects were selected based on searches using keywords on GitHub APIs, using convenience sampling. Several SUTs were looked at, in which we discarded the ones that would not compile, would crash at startup, would use obscure/unpopular libraries with no documentation to get them started, are too trivial, student projects, etc. Where possible, we tried to prioritize/sort based on number of stars on GitHub.

This process was repeated a few times throughout the years (i.e., since 2017). We also added interesting SUTs used in different studies (e.g., the recently added *genome-nexus* and *market* were first used by Kim et al. in [28]). Others were based on their popularity. For example, when during the Covid world-pandemic it was in the news that the Covid-App used in Germany was open-source, we added one of the RESTful APIs from its backend (i.e., *cwa-verification*).

Note that some of these open-source projects might be no longer supported, whereas others are still developed and updated. Once a system is added to EMB, we do not modify nor keep it updated with its current version under development. The reason is that we want to keep an easy to use, constant set of case studies for experimentation that can be reliably used throughout the years.

The SUTs called NCS (Numerical Case Study) and SCS (String Case Study) are artificial, developed by us. They are

based on numerical and string-based functions previously used in the literature of unit test generation [47], [48]. We just re-implemented them in different languages, and put them behind different types of web services. The reasoning here was to validate white-box heuristics when used in system testing in Web APIs. If these kinds of functions can be fully covered when doing search-based unit test generation (e.g., with EvoSuite [30]), then an API fuzzer should aim at being able to do so as well.

Table I shows that the different APIs can vary greatly in size. Some are small, less than 1 000 LOCs, whereas others are more than 100 000 LOCs. In total, the LOCs over whole EMB is more than 470 000. Such variety is important for a curated corpus, as it enables to evaluate the *scalability* of novel techniques on SUTs of very different sizes and complexity.

One common critique, from academic reviewers, about the earliest uses of EMB as case study in our previous work was on the size of the SUTs. A few times, they were considered as too small. This is one reason for adding large SUTs (e.g., *languagetool* and *timbuctoo*) in the recent years. However, there are some counter-arguments that can be made to support the use of “small” SUTs.

First, what reported in Table I does **NOT** include third-party libraries. When running a Web API, its business logic (which we want to maximize code coverage on and find faults in it, if any is there) only covers a tiny proportion of the whole code

of the application. Consider *rest-ncs*, with its 9 files and 605 LOCs. Those files, when compiled and compressed, take 12 KB. However, the executable JAR file *rest-ncs-sut.jar* takes 24 729 KB (recall that JAR files are compressed). In terms of compiled size, the business logic counts for just 0.04% of the whole application, where the rest is made by third-party libraries (e.g., HTTP servers such as Tomcat and application frameworks such as Spring). Given these values, it can be roughly estimated that the source-code of these third-party libraries consists of more than 1 million LOCs. However, not all such code in third-party libraries do influence the execution flow in the business logic of the SUT. Some third-party libraries might be used directly by the business logic, whereas others do not (e.g., especially all the code related to reading HTTP requests from the TCP sockets and parse them before calling the appropriate functions in the business logic to deal with those requests). So, it is hard to quantify in an objective way what is the impact of these third-party libraries on the testing efforts. Furthermore, a SUT might just need one single function from a library, but still the whole library would need to be imported. There are techniques to prune, at build time, any unused third-party code from the self-executable JAR files, but those do not work when the SUT (or any of its employed libraries) uses *reflection*, which is very common (e.g., with frameworks such as Spring). Looking at and analyzing which third-party library classes are actually loaded at runtime in the JVM by the SUT would not help much either, for two reasons. First, unless 100% code coverage is achieved, such values would be an underestimation, as classes are loaded dynamically based on when they are first accessed. If a third-party function is used inside a block in a *if* statement, but such block is never executed during testing, then such class would not be loaded. Second, some frameworks such as Spring and JEE might scan and load all classes on the classpath, to check for annotations (e.g., used to determine which classes to instantiate as beans). They might end up loading all classes from the third-party libraries regardless of whether they are actually in use or not. For all these reasons, we do not attempt to provide information on the used third-party libraries in these SUTs. However, one has to be aware of their presence and potential impact on system-level test case generation.

Another argument to support the validity of using “small” SUTs, and not just large ones, relies on the practice of *microservice architectures* in industry [7], [8]. Large enterprise applications, with possibly millions of lines of code, are often split from *monoliths* into many Web APIs. This is for example the case for Meituan, a large e-commerce enterprise with microservice architectures involving more than a thousand distinct RPC APIs [21], [29]. Each API would not be particularly large (e.g., between 1 and 50 thousand LOCs [21]), where the complexity would strongly depend on the number and type of connections between the different services. On this type of highly interconnected industrial systems, LOCs of the business logic might not be the best indicator for how a fuzzer would fare on them.

The current selection of APIs in EMB does not represent a statistically representative sample of open-source Web APIs [49], nor it needs to be. The vast majority of Web APIs are developed in industry, and not as open-source projects. There are cases in which some entities in the public administration do open-source their developed systems, but those are a tiny minority. Ideally, academic results should have impact on practice [3], [50]–[52]. For such a goal, industry-academia collaborations are a must [53], like for example we are doing with Meituan [21], [29]. However, in most cases industrial case studies cannot be shared, which makes replicating studies practically impossible. Having a selection of non-trivial open-source projects, besides using industrial systems, is hence useful for scientific purposes. Existing fuzzers do not achieve full coverage on EMB [28], [54]. So, EMB can be considered as a challenge to overcome while addressing as well industrial APIs. Several of their research problems that need to be solved will likely be similar.

IV. LIBRARY SUPPORT FOR TEST DRIVERS

For each SUT in EMB, we have written *driver classes*. Those are used for two main reasons: (1) how to handle the SUT; and (2) to provide useful information to the fuzzers to be able to test these SUTs. Each such driver class extends the class *EmbeddedSutController*, which is part of the open-source library *evomaster-client-java-controller*, published on Maven Central. Note: for simplicity, for the rest of this section we will focus only on the Java version of this library. There is also an equivalent library for JavaScript/TypeScript, used for the SUTs running on NodeJS.

To handle the SUT, there are three main aspects to consider: how to *start* it, how to *reset* it and how to *stop* it. These functionalities are captured in the interface *SutHandler*, whose three main methods are shown in Figure 1, including their current JavaDoc documentation. One interesting point to make here is that the *start* method returns a string, which should specify the URL of where the API is running. This is done to allow the API to bind to an ephemeral TCP port, instead of a fixed one like 8080. This is not only needed when test cases are run in parallel or on a Continuous Integration server to avoid port allocation conflicts, but it is also essential for when running scientific experiments. Several instances of the same API can be run and fuzzed in parallel without the need of worrying of TCP port conflicts. If the API needs to use some external database like PostgreSQL or MongoDB, those can be automatically started with Docker [32], using the Testcontainers [33] library.

Figure 2 shows an excerpt from a generated JUnit test by EVOMASTER on the *rest-ncs* API. Here, the driver class *EmbeddedEvoMasterController* (which extends *EmbeddedSutController*) is instantiated (Line 3), and saved in a variable named *controller* of type *SutHandler*. Before any test is run, in a *@BeforeClass* function the *controller* is used to start the application (Line 9). The base URL of where the API is currently running

```

1 /**
2  * <p>
3  * Start a new instance of the SUT.
4  * </p>
5  *
6  * <p>
7  * This method must be blocking until the SUT is
   initialized.
8  * </p>
9  *
10 * <p>
11 * How this method is implemented depends on the
   library/framework in which
12 * the application is written.
13 * For example, in Spring applications you can use
   something like:
14 * {@code SpringApplication.run()}
15 * </p>
16 *
17 *
18 * @return the base URL of the running SUT, eg "http
   ://localhost:8080"
19 */
20 String startSut();
21
22 /**
23  * <p>
24  * Stop the SUT.
25  * </p>
26  *
27  * <p>
28  * How to implement this method depends on the
   library/framework in which
29  * the application is written.
30  * For example, in Spring applications you can save
   in a variable the {@code
   ConfigurableApplicationContext}
31  * returned when starting the application, and then
   call {@code stop()} on it here.
32  * </p>
33  */
34 void stopSut();
35
36 /**
37  * <p>
38  * Make sure the SUT is in a clean state (eg, reset
   data in database).
39  * </p>
40  *
41  * <p>
42  * A possible (likely very inefficient) way to
   implement this would be to
43  * call {@code stopSUT} followed by {@code startSUT
   }.
44  * </p>
45  *
46  * <p>
47  * When dealing with databases, you can look at the
   utility functions from
48  * the class {@link DbCleaner}.
49  * How to access the database depends on the
   application.
50  * To access a {@code java.sql.Connection}, in
   Spring applications you can use something like:
51  * {@code ctx.getBean(JdbcTemplate.class).
   getDataSource().getConnection()}.
52  * </p>
53  */
54 void resetStateOfSUT();

```

Fig. 1: Excerpt from the interface `SutHandler`.

is saved in the variable called `baseUrlOfSut`. Then, before each test execution, in a `@Before` function the controller is used to reset the state of the application (Line 27). When in the test cases HTTP calls are made toward the SUT (e.g., using the popular library `RestAssured` [55]), the variable `baseUrlOfSut` is used as a base to form the URL to query (Line 34). Once all test cases marked with `@Test` are executed, the API is shut down in a `@AfterClass` block (Line 22). This makes the test case *self-contained*, and usable for regression testing, as it does not need any manual intervention to deal with the API.

This test case was generated by EVOMASTER, but how such driver classes can be used in the generated tests is completely independent from the used fuzzers. Other fuzzers could use the library `evomaster-client-java-controller` to generate test cases with this kind of scaffolding using `SutHandler`.

Although the functions in `SutHandler` are needed when running the generated tests, there is also other important information that a fuzzer needs to be able to generate test cases. For example, for a RESTful API, where is the OpenAPI/Swagger schema located? If the SUT requires some authentication, what information should be used? If the fuzzer is doing white-box testing, how can the classes of business logic be differentiated from the third-party libraries? (e.g., to measure code coverage). Accessing this information is not needed when running a generated test (e.g., the authentication information would have been used in the generated tests to setup the right HTTP headers), but it is essential for using a fuzzer. For these reasons, the abstract class `EmbeddedSutController` (which implements `SutHandler`) has a further list of abstract methods that must be implemented in the driver classes. Figure 3 shows three of such methods: `getPackagePrefixesToCover()` (Line 2) used to specify which packages constitute the business logic of the API; `getProblemInfo()` (Lines 7) that specifies the type of problem (e.g., REST or GraphQL), and specific information about it (e.g., for REST APIs the location of where the OpenAPI/Swagger schema can be found); `getInfoForAuthentication()` (Lines 14) provides authentication information, if any is needed. All these configuration objects (e.g., `ProblemInfo` and `AuthenticationDto`) are part of the `evomaster-client-java-controller` library, including JavaDoc documentation.

Regarding authentication, there are many ways in which requests towards an API can be authenticated. We provide different descriptive objects to represent different kinds of authentication, including static tokens and cookie handling, together with a few utility functions to simplify how to instantiate them (e.g., `AuthUtils` as can be seen in Figure 4). Note that a fuzzer that reads one of these `AuthenticationDto` objects would still need to use such information to authenticate the requests. We simply provide all needed information in object data structures, but still it is up to the fuzzers to use such information to authenticate. For example, `EVOMASTER`

```

1 public class EM__MIO_1_Test {
2
3     private static final SutHandler controller = new em.embedded.org.restnecs.EmbeddedEvoMasterController();
4
5     private static String baseUrlOfSut;
6
7     @BeforeClass
8     public static void initClass() {
9         controller.setupForGeneratedTest();
10        baseUrlOfSut = controller.startSut();
11        assertNotNull(baseUrlOfSut);
12        RestAssured.enableLoggingOfRequestAndResponseIfValidationFails();
13        RestAssured.useRelaxedHTTPSValidation();
14        RestAssured.urlEncodingEnabled = false;
15        RestAssured.config = RestAssured.config()
16            .jsonConfig(JsonConfig.jsonConfig().numberReturnType(JsonPathConfig.NumberReturnType.DOUBLE))
17            .redirect(redirectConfig().followRedirects(false));
18    }
19
20    @AfterClass
21    public static void tearDown() {
22        controller.stopSut();
23    }
24
25    @Before
26    public void initTest() {
27        controller.resetStateOfSUT();
28    }
29
30    @Test
31    public void test_0() throws Exception {
32
33        given().accept("application/json")
34            .get(baseUrlOfSut + "/api/fisher/502/173/0.14986444170152968")
35            .then()
36            .statusCode(200)
37            .assertThat()
38            .contentType("application/json")
39            .body("'resultAsInt'", nullValue())
40            .body("'resultAsDouble'", numberMatches(6.863317496222495E-17));
41    }
42 }

```

Fig. 2: Excerpt from a generated test suite with EVOMASTER for the API *rest-ncs*, showing just one test case.

```

1 @Override
2 public String getPackagePrefixesToCover() {
3     return "org.devgateway.";
4 }
5
6 @Override
7 public ProblemInfo getProblemInfo() {
8     return new RestProblem(
9         "http://localhost:" + getSutPort() + "/v2/api-docs?group=locDashboardsApi",
10        null);
11 }
12
13 @Override
14 public List<AuthenticationDto> getInfoForAuthentication() {
15     return Arrays.asList(
16         AuthUtils.getForDefaultSpringFormLogin("ADMIN", "admin", "admin"));
17 }

```

Fig. 3: Excerpt from driver class for *ocvn-rest* API.


```

1 /**
2  * DTO representing the use of authentication via a X-WWW-FORM-URLENCODED POST submission.
3  * Assuming default names and endpoint used in SpringSecurity for default formLogin() configuration.
4  *
5  * When using this kind of DTO, EM will first do a POST on such endpoint with valid credentials,
6  * and then use the resulting cookie for the following HTTP requests.
7  *
8  * @param dtoName a name used to identify this dto. Mainly needed for debugging
9  * @param username the id of a user
10 * @param password password for that user
11 * @return a DTO
12 */
13 public static AuthenticationDto getForDefaultSpringFormLogin(String dtoName, String username, String
    password) {
14
15     CookieLoginDto cookie = new CookieLoginDto();
16     cookie.httpVerb = CookieLoginDto.HttpVerb.POST;
17     cookie.contentType = CookieLoginDto.ContentType.X_WWW_FORM_URLENCODED;
18     cookie.usernameField = "username";
19     cookie.passwordField = "password";
20     cookie.loginEndpointUrl = "/login";
21     cookie.username = username;
22     cookie.password = password;
23
24     AuthenticationDto dto = new AuthenticationDto(dtoName);
25     dto.cookieLogin = cookie;
26
27     return dto;
28 }

```

Fig. 4: Snippet code of the function `getForDefaultSpringFormLogin` in `AuthUtils`, from the library `evomaster-client-java-controller`. It is used to create authentication information for the default configuration of login-based form authentication in Spring Security.

uses this information to make a login call at each test case execution, and then add the received HTTP Cookie in all the HTTP calls in the test case.

For the APIs running on the JVM, we provide two different kinds of driver classes: `EmbeddedSutController` and `ExternalSutController`. The difference is how the SUT is started. In the former case (i.e., `EmbeddedSutController`), the API is started embedded in the same JVM of the driver class, where the API itself is added as a library and started programmatically. In the latter case (i.e., `ExternalSutController`) the API is spawned in a different process. In such case, the driver class needs to know where the executable uber-jar is located.

The driver classes for `EmbeddedSutController` are easier to write and use (e.g., when running the generated tests with a debugger), and are those we recommend to use and write. However, for cases of applications that are not easy to start programmatically (e.g., JEE containers), `ExternalSutController` provides an alternative option.

At the time of this writing, considering the 29 APIs currently in EMB, there are $8 + 21 \times 2 = 50$ driver classes that we have written (the 8 NodeJS APIs do not have `ExternalSutController` drivers). For this reason, throughout the years we have made a major effort in trying to make sure that the drivers are *backward compatible* at each new release of `evomaster-client-java-controller`. Otherwise, at each breaking change, we would have to manually update 50

driver classes, which is time consuming.

V. EMB PROJECT STRUCTURE

It is important to understand how the EMB project is structured, to be able to effectively navigate through it. Each folder represents a set of SUTs (and drivers) that can be built using the same tools. For example, the folder `jdk_8_maven` contains all the SUTs that need JDK 8 and are built with Maven. On the other hand, the SUTs in the folder `jdk_11_gradle` require JDK 11 and Gradle. Each SUT is grouped by type (e.g., `rest` and `graphql`).

For the JVM, each module has 2 submodules, called `cs` (short for “Case Study”) and `em` (short for “EvoMaster”). The module `cs` contains all the source code of the different SUTs, whereas `em` contains all the drivers. This separation was made explicitly to enable to use the APIs *as they are*, without dependencies with the driver classes.

Regarding JavaScript, unfortunately NodeJS does not have a good handling of multi-module projects. Each SUT has to be built separately. However, for each SUT, we put its source code under a folder called `src`, whereas all the code related to the drivers is under `em`, to explicitly differentiate them. Currently, both NodeJS 14 and 16 should work on these SUTs.

The driver classes for the JVM are all called `EmbeddedEvoMasterController` and `ExternalEvoMasterController`. For JavaScript, they are in a script file called `app-driver.js`.

Everything can be setup and built by running the Python script `scripts/dist.py`. Note that you will need installed at least JDK 8, JDK 11 and NPM, as well as Docker. Also, you will need to setup environment variables like `JAVA_HOME_8` and `JAVA_HOME_11`. The script will issue error messages if any prerequisite is missing. Once the script is completed, all the built SUTs will be available under the `dist` folder. Note that here the drivers will be built as well besides the SUTs, and the SUT themselves will also have an instrumented version (for white-box testing heuristics) for EVOMASTER (this is for JavaScript, whereas instrumentation for JVM is done at runtime, via an attached JavaAgent).

For running experiments with EVOMASTER, you can also “start” each driver directly from an IDE (e.g., IntelliJ). Each of these drivers has a “main” method that is running a REST API (binding on default port 40100), where each operation (like start/stop/reset the SUT) can be called via an HTTP message by EVOMASTER. For JavaScript, you need to use the files `em-main.js` under the `instrumented/em` folders.

Each release of EMB is linked to the releases of `evomaster-client-java-controller`. They are kept in sync, and use the same version numbers. The master branch of the Git repository of EMB points to the latest commit of the latest published release. All further development is done in a `develop` branch (including adding new SUTs), till the next release. Using GitHub Actions, each new release is automatically uploaded to Zenodo for long term storage (e.g., version 1.5.0 [56]). This means that cloning and building the master branch of EMB is the same as using the latest version on Zenodo. However, building the `develop` branch is more complicated, as the snapshot version of `evomaster-client-java-controller` needs to be built and installed manually (this is discussed in more details in our documentation).

VI. EXPERIMENT EXAMPLE

In Table II we show the results of EVOMASTER applied on EMB, when run for 10 minute budget. To take into account the randomness of the employed algorithms, each experiment was repeated 10 times. Reported coverage results are based on EVOMASTER’s own instrumentation. Detected faults are for example based on 500 HTTP status codes, and detected mismatches from the API schemas.

On some APIs, quite high coverage can be achieved, whereas on others coverage values are low. This shows that there are still many research challenges that need to be overcome to achieve better results [57]. However, still it is possible to find many faults in these APIs [58].

When looking at the results in Table II we need to make two disclaimers. First, coverage results for NodeJS APIs are an *underestimation*, as currently for NodeJS we do not collect the achieved coverage at boot-time (i.e., when the API starts, before we make any HTTP call towards it). Second, fault results for GraphQL APIs are an *overestimation*, as in GraphQL currently there is no clear way to automatically distinguish between user and server errors [19], [20].

TABLE II: Example of results obtained for EVOMASTER applied on EMB, averaged out of 10 runs, each one for 10 minutes.

SUT	Line Coverage %	# Detected Faults
<i>catwatch</i>	50.2%	24.3
<i>cwa-verification</i>	46.6%	3.8
<i>cyclotron</i>	30.9%	30.0
<i>disease-sh-api</i>	19.3%	32.1
<i>ecommerce-server</i>	7.7%	25.2
<i>features-service</i>	80.0%	29.0
<i>genome-nexus</i>	33.4%	16.8
<i>gestaohospital-rest</i>	39.4%	21.6
<i>graphql-ncs</i>	77.6%	12.9
<i>graphql-scs</i>	75.9%	11.0
<i>js-rest-ncs</i>	83.2%	6.0
<i>js-rest-scs</i>	75.0%	1.0
<i>languagetool</i>	37.6%	4.2
<i>market</i>	46.3%	17.9
<i>ocvn-rest</i>	17.4%	241.1
<i>patio-api</i>	34.8%	43.2
<i>petclinic-graphql</i>	56.0%	18.0
<i>proxyprint</i>	46.6%	69.0
<i>react-finland</i>	2.3%	33.2
<i>realworld-app</i>	24.4%	27.4
<i>rest-ncs</i>	93.0%	6.0
<i>rest-news</i>	63.6%	7.7
<i>rest-scs</i>	83.5%	10.8
<i>restcountries</i>	73.4%	2.0
<i>rpc-thrift-ncs</i>	91.5%	12.1
<i>rpc-thrift-scs</i>	79.9%	13.0
<i>scout-api</i>	49.5%	74.0
<i>spacex-api</i>	40.1%	43.3
Average	52.1%	29.9

VII. CONCLUSIONS

In this paper, we have presented EMB, a curated corpus of Web APIs. We have been using and extending this corpus since 2017, for running experiments with the EVOMASTER fuzzer. An important aspect here is how to configure and run all of these APIs, especially when they use external databases such as PostgreSQL and MongoDB, and/or need authentication information. We provide configuration files (called *driver classes*) to handle all of them, together with a library support (including documentation) to ease the use of such files by other fuzzers. EMB has already been used by other research groups, like for example in [28], [37], [38].

To avoid developing novel techniques that just overfit a given corpus of artifacts, it will be important to keep extending EMB with new APIs each year. This would be needed as well to take into account new libraries and frameworks that become widespread in industry. For example, currently none of the SUTs in EMB use Kafka [59], an open-source distributed event

streaming platform used by more than 80% of all Fortune 100 companies [59]. How to automatically generate events from Kafka to cover different execution paths in the code of the tested APIs will be a further research challenge to address. When adding new APIs to EMB, searching for and prioritizing APIs using Kafka (or other popular tools/frameworks currently missing in EMB) would be useful.

Besides isolated Web APIs, there are other types of Web/Enterprise applications that could be added to EMB. This could be whole microservice architectures (e.g., several interconnected APIs running behind a Gateway), as well as systems with web frontends that run on the browser. Currently, EVOMASTER does not generate tests for this kind of systems. When we will extend EVOMASTER to support them, the selected case studies to evaluate it will be added to EMB.

EMB is published on GitHub [34], as well as on Zenodo for long term storage [56].

REFERENCES

- [1] G. Canfora and M. Di Penta, "Service-oriented architectures testing: A survey," in *Software Engineering*. Springer, 2009, pp. 78–105.
- [2] 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.
- [3] A. Arcuri, "An experience report on applying software testing academic results in industry: we need usable automated test generation," *Empirical Software Engineering*, vol. 23, no. 4, pp. 1959–1981, 2018.
- [4] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.
- [5] "Graphql foundation," <https://graphql.org/foundation/>.
- [6] "grpc," <https://grpc.io/>.
- [7] S. Newman, *Building Microservices*. "O'Reilly Media, Inc.", 2015.
- [8] R. Rajesh, *Spring Microservices*. Packt Publishing Ltd, 2016.
- [9] N. Laranjeiro, J. Agnelo, and J. Bernardino, "A black box tool for robustness testing of rest services," *IEEE Access*, vol. 9, pp. 24 738–24 754, 2021.
- [10] A. Arcuri, "EvoMaster: Evolutionary Multi-context Automated System Test Generation," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018.
- [11] H. Wu, L. Xu, X. Niu, and C. Nie, "Combinatorial testing of restful apis," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2022.
- [12] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "REStTest: Automated Black-Box Testing of RESTful Web APIs," in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2021, pp. 682–685.
- [13] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful REST API fuzzing," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2019, p. 748–758.
- [14] E. Viglianisi, M. Dallago, and M. Ceccato, "Resttestgen: Automated black-box testing of restful apis," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2020.
- [15] Z. Hatfield-Dodds and D. Dygalo, "Deriving semantics-aware fuzzers from web api schemas," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2022, pp. 345–346.
- [16] D. M. Vargas, A. F. Blanco, A. C. Vidaurre, J. P. S. Alcocer, M. M. Torres, A. Bergel, and S. Ducasse, "Deviation testing: A test case generation technique for graphql apis," in *11th International Workshop on Smalltalk Technologies (IWST)*, 2018, pp. 1–9.
- [17] S. Karlsson, A. Čaušević, and D. Sundmark, "Automatic property-based testing of graphql apis," *arXiv preprint arXiv:2012.07380*, 2020.
- [18] L. Zetterlund, D. Tiwari, M. Monperrus, and B. Baudry, "Harvesting production graphql queries to detect schema faults," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 365–376.
- [19] A. Belhadi, M. Zhang, and A. Arcuri, "Evolutionary-based Automated Testing for GraphQL APIs," in *Genetic and Evolutionary Computation Conference (GECCO)*, 2022.
- [20] —, "White-box and black-box fuzzing for graphql apis," 2022. [Online]. Available: <https://arxiv.org/abs/2209.05833>
- [21] M. Zhang, A. Arcuri, Y. Li, Y. Liu, and K. Xue, "White-box fuzzing rpc-based apis with evomaster: An industrial case study," 2022. [Online]. Available: <https://arxiv.org/abs/2208.12743>
- [22] A. Arcuri, J. P. Galeotti, B. Marculescu, and M. Zhang, "Evomaster: A search-based system test generation tool," *Journal of Open Source Software*, vol. 6, no. 57, p. 2153, 2021.
- [23] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, p. 3, 2019.
- [24] —, "Automated black-and white-box testing of restful apis with evomaster," *IEEE Software*, vol. 38, no. 3, pp. 72–78, 2020.
- [25] "EvoMaster," <https://github.com/EMResearch/EvoMaster>.
- [26] A. Arcuri, "REStful API Automated Test Case Generation," in *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 9–20.
- [27] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability (STVR)*, vol. 22, no. 2, pp. 67–120, 2012.
- [28] M. Kim, Q. Xin, S. Sinha, and A. Orso, "Automated test generation for rest apis: No time to rest yet," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 289–301. [Online]. Available: <https://doi.org/10.1145/3533767.3534401>
- [29] M. Zhang, A. Arcuri, Y. Li, K. Xue, Z. Wang, J. Huo, and W. Huang, "Fuzzing microservices in industry: Experience of applying evomaster at meituan," 2022. [Online]. Available: <https://arxiv.org/abs/2208.03988>
- [30] 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.
- [31] —, "A large-scale evaluation of automated unit test generation using EvoSuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 8, 2014.
- [32] "Docker," <https://www.docker.com>.
- [33] "Testcontainers," <https://github.com/testcontainers/testcontainers-java>.
- [34] "Evomaster benchmark (emb)," <https://github.com/EMResearch/EMB>.
- [35] A. Arcuri and J. P. Galeotti, "Handling sql databases in automated system test generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–31, 2020.
- [36] M. Zhang and A. Arcuri, "Adaptive hypermutation for search-based system test generation: A study on rest apis with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, 2021.
- [37] O. Sahin and B. Akay, "A discrete dynamic artificial bee colony with hyper-scout for restful web service api test suite generation," *Applied Soft Computing*, vol. 104, p. 107246, 2021.
- [38] D. Stallenberg, M. Olsthoorn, and A. Panichella, "Improving test case generation for rest apis through hierarchical clustering," *arXiv preprint arXiv:2109.06655*, 2021.
- [39] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria," in *Proceedings of 16th International conference on Software engineering*. IEEE, 1994, pp. 191–200.
- [40] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.
- [41] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.
- [42] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1393–1403.
- [43] Z. Li, Y. Wu, L. Ma, X. Xie, Y. Chen, and C. Fan, "Gbgallery: A benchmark and framework for game testing," *Empirical Software Engineering*, vol. 27, no. 6, pp. 1–27, 2022.

- [44] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszédes, R. Ferenc, and A. Mesbah, "Bugsjs: a benchmark of javascript bugs," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 90–101.
- [45] C. Artho, A. Benali, and R. Ramler, "Test benchmarks: Which one now and in future?" in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 328–336.
- [46] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato, "Automated black-box testing of nominal and error scenarios in RESTful APIs," *Software Testing, Verification and Reliability*, p. e1808, 2022.
- [47] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness?" in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2011, pp. 265–275.
- [48] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators," *Software Testing, Verification, and Reliability*, vol. 16, no. 3, pp. 175–203, 2006.
- [49] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2012, pp. 178–188.
- [50] V. Garousi, M. Felderer, M. Kuhrmann, and K. Herkiloğlu, "What industry wants from academia in software testing?: Hearing practitioners' opinions," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2017, pp. 65–69.
- [51] V. Garousi and M. Felderer, "Worlds apart: a comparison of industry and academic focus areas in software testing," *IEEE Software*, vol. 34, no. 5, pp. 38–45, 2017.
- [52] V. Garousi, M. M. Eskandar, and K. Herkiloğlu, "Industry-academia collaborations in software testing: experience and success stories from canada and turkey," *Software Quality Journal*, pp. 1–53, 2016.
- [53] V. Garousi, D. Pfahl, J. M. Fernandes, M. Felderer, M. V. Mäntylä, D. Shepherd, A. Arcuri, A. Coşkunçay, and B. Tekinerdogan, "Characterizing industry-academia collaborations in software engineering: evidence from 101 projects," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2540–2602, 2019.
- [54] M. Zhang and A. Arcuri, "Open problems in fuzzing restful apis: A comparison of tools," 2022. [Online]. Available: <https://arxiv.org/abs/2205.05325>
- [55] "RestAssured," <https://github.com/rest-assured/rest-assured>.
- [56] A. Arcuri, ZhangMan, A. Gol, asmab89, and J. P. Galeotti, "Emresearch/emb:," Jun. 2022. [Online]. Available: <https://doi.org/10.5281/zenodo.6651791>
- [57] M. Zhang and A. Arcuri, "Open problems in fuzzing restful apis: A comparison of tools," *arXiv preprint arXiv:2205.05325*, 2022.
- [58] B. Marculescu, M. Zhang, and A. Arcuri, "On the faults found in rest apis by automated test generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–43, 2022.
- [59] "Kafka," <https://kafka.apache.org/>.