Maintainability Measurement Report on

TechEmpower – FrameworkBenchmarks

Dr. Adam Vanya

FrontEndART Ltd.

15-02-2021

Table of Contents

Table of Figures

1 Introduction

1.1 The software analyzed

The TechEmpower – FrameworkBenchmarks GitHub project (https://github.com/TechEmpower/FrameworkBenchmarks/tree/master) provides representative performance measures across a wide field of web application frameworks. The project presently includes frameworks on many languages including Go, Python, Java, Ruby, PHP, C#, F#, Clojure, Groovy, Dart, JavaScript, Erlang, Haskell, Scala, Perl, Lua, C, and others. The current tests exercise plaintext responses, JSON serialization, database reads and writes via the object-relational mapper (ORM), collections, sorting, server-side templates, and XSS counter-measures.

The popularity of this open source GitHub project can be described with more than 5600 stars (indication of how many people like this project), more than 1600 forks (indication of how often the codebase was accessed) and more than 230 watches (indication of how many people follow this project).

1.2 Goal and motivation

Web application frameworks are continuously evolving to meet their new requirements. For example, to be more performant and to use the latest features of the programming languages they are implemented in. Consequently, the contributors of TechEmpower – FrameworkBenchmarks need to update their framework related implementations and tests to keep them relevant and useful for their community.

It is important that the necessary changes to FrameworkBenchmarks are introduced as fast as possible so that its community can assess the performance of the new framework versions not long after there are released. This way, the community can make their decisions on time. Next to that, the tests should be reliable and therefore free from bugs since the community will make some of their key decisions based on the outcome of the FrameworkBenchmarks tests. To achieve this, the implementation of the tests needs to be highly maintainable.

By executing a **source code quality assessment** on FrameworkBenchmarks, FrontEndART Ltd. intends to help the contributors and the community of FrameworkBenchmarks with identifying the maintainability related technical risks FrameworkBenchmarks has and with providing pieces of expert advice for mitigating those risks. These risks and pieces of advice can then be used to increase the maintainability of the test implementations. An improved maintainability will help to limit the number of bugs introduced and the time it takes to create future versions for the community.

Next to that, FrontEndART Ltd. also finds this a great opportunity to showcase their source code quality assessment service. This report documents the results of this service.

1.3 Report structure

The rest of this report is structured as follows:

- **Section 2** details the scope of the analysis executed.
- **Section 3** describes the measurement results together with the detailed technical findings.
- **Section 4** summarizes the technical findings.
- **Section 5** suggests improvement steps to be taken.

2 Analysis scope

Version 2.2 of QualityGate was used to execute the analysis. This source code quality management system analyzed each of the main branch commits of the codebase and future ones are planned to be analyzed too. The subject of the current report is the commit which was created on 18-12-2020 (SHA: a85daeb3bd198aa7e30baf91cbaf319a6a7e046b). The scope of the inspection was maintainability assessment. During the assessment, all Java files were analyzed including test, generated and 3rd party code. This is in line with how all the other systems in the FrontEndART benchmark were analyzed. Based on the analysis results, the expert assessment was concluded on 15-02-2021.

The following system-level measurements are characterizing the codebase analyzed:

System-level Metric	Value
Number of Packages	**164**
Number of Methods	**1569**
Logical Lines of Code	**16488**
API Documentation Coverage	**16%**
Clone Coverage	**5%**
Clone Instances	**97**
Warnings – Blocker	**21**
Warnings – Critical	**140**
Warnings – Major	**587**
Warnings – Minor	**1637**
Warnings – Info	**13**

3 Measurement results and technical findings

The maintainability of the TechEmpower – FrameworkBenchmarks codebase is **7.06** on the scale from 0 (score of the least maintainable system) to 10 (score of the most maintainable system). This score means that the maintainability of the TechEmpower – FrameworkBenchmarks codebase is **GOOD** considering the FrontEndART benchmark which includes hundreds of open source and closed source systems. The score of the snapshot analyzed may change somewhat over time since the benchmark changes too.

The following figure indicates the metrics and quality attributes based on which the maintainability score of the codebase was calculated:

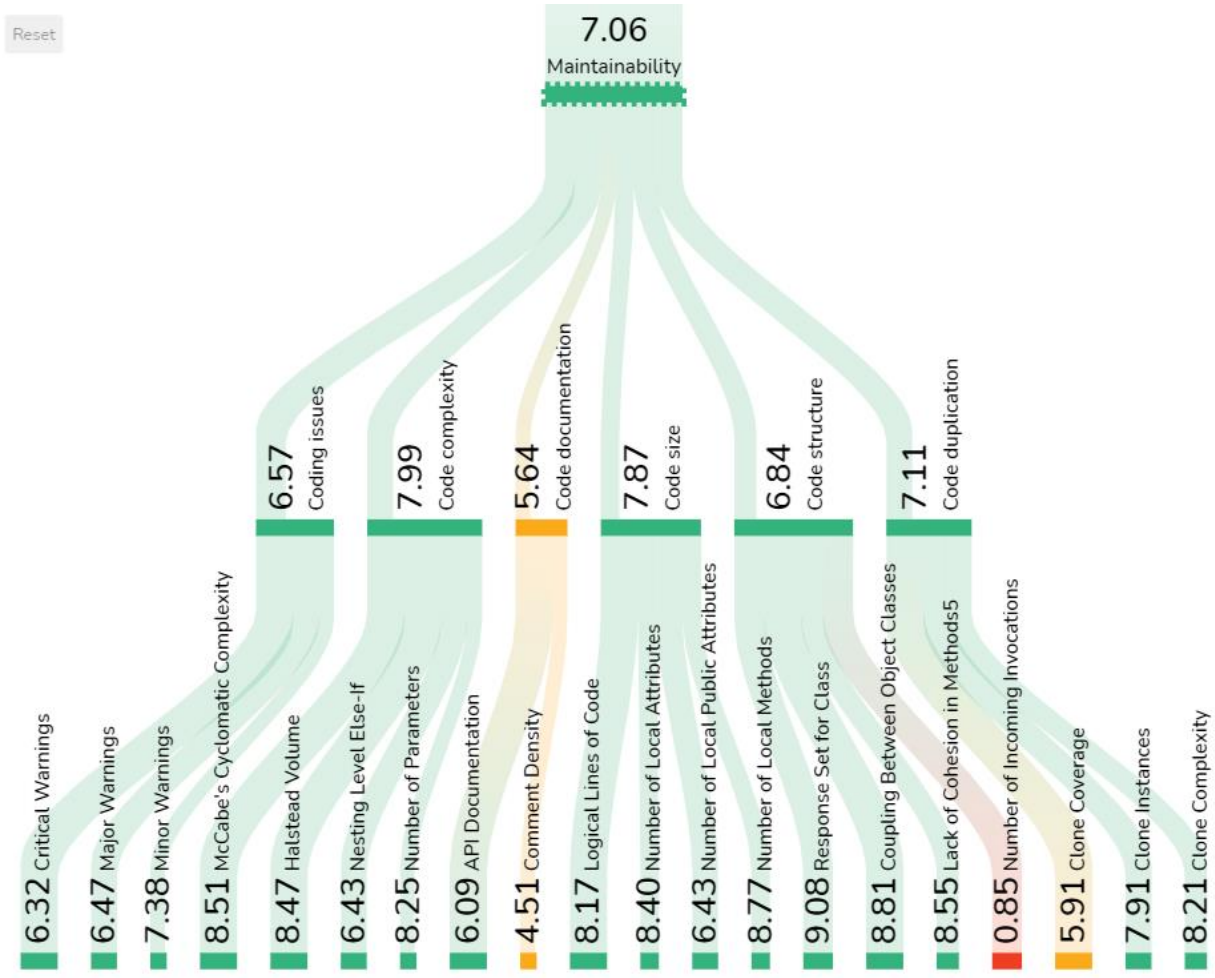

Figure 1*:* **FrontEndART Maintainability Model applied to TechEmpower – FrameworkBenchmarks**

This section goes through the FrontEndART maintainability model described above and explains the elements of that model: 6 quality attributes and 20 metrics. For each model element, this section also details the measurement results and the underlying technical findings for the TechEmpower – FrameworkBenchmarks codebase. Those metrics where the TechEmpower – FrameworkBenchmarks codebase performs less than ABOVE MARKET AVERAGE are given extra attention when detailing measurement results.

3.1 Coding Issues

The Coding Issues quality attribute indicates the extent to which a given software system is implemented using programming best practices as compared to other systems in the FrontEndART benchmark. The TechEmpower – FrameworkBenchmarks codebase scores **6.57** (ABOVE MARKET AVERAGE) on this quality attribute. This codebase implements more best practices than most of the other systems in the benchmark.

3.1.1 WarningCritical, WarningMajor and WarningMinor

Coding violations are created by programmers when they do not follow programming best practices. In such cases, warnings can be created. Based on the effect of a violation on maintainability, one can distinguish between critical, major and minor warnings. The score of a warning metric is calculated by measuring the number coding violations for each class.

The codebase has in total 567 classes and these classes contain 140 critical violations, 587 major violations and 1637 minor violations. Based on the measurements of each class and the FrontEndART benchmark, the WarningCritical, WarningMajor and WarningMinor metric scores are **6.32** (ABOVE MARKET AVERAGE), **6.47** (ABOVE MARKET AVERAGE) and **7.38** (GOOD), respectively.

A representative example of a critical violation can be found in the **armeria** component involving the frameworks/Java/armeria/src/main/java/hello/services/**PostgresFortunesService.java** file. In this file, there is a method called **buildMustacheTemplate(List<Fortune> fortunes)** which does not close the **bytes** resource. The following figure shows the content of this method:

```
86      private byte[] buildMustacheTemplate(List<Fortune> fortunes) { FH UPM
87        Mustache mustache = mustacheFactory.compile("fortunes.mustache");
88        ByteArrayOutputStream bytes = new ByteArrayOutputStream(); FH CIR
89
90        try (Writer writer = new OutputStreamWriter(bytes, StandardCharsets.UTF_8)) {
91          mustache.execute(writer, fortunes);
92        } catch (IOException e) {
93          throw new UncheckedIOException(e);
94        }
95
96        return bytes.toByteArray();
97      }
98    }
```

Figure 2: **Critical violation example**

To decrease the negative effect of coding violations on maintainability the number of critical and minor violations should further be decreased by fixing those violations. The full list of the detected violations can be found on the website of the QualityGate system.

3.2 Code Complexity

The Coding Complexity quality attribute indicates the extent to which a given software system is easy to understand, modify and test as compared to other systems in the FrontEndART benchmark. The TechEmpower – FrameworkBenchmarks codebase scores **7.99** (GOOD) on this quality attribute. The code of this codebase is less complex than the code of most of the other systems in the benchmark.

3.2.1 McCabe's Cyclomatic Complexity

The McCabe's cyclomatic complexity metric counts all the possible execution paths for each methods of the system. This is achieved by counting the decision points in the code of those methods. A method with a high amount of execution paths is difficult to understand and test for functional suitability. For a maintainable system, the number of execution paths in a method must be kept minimal.

The codebase has in total 1569 methods and the cyclomatic complexity of those methods reaches a maximum of 10 McCabe points. Based on the measurements of each method and the FrontEndART benchmark, the McCabe's Cyclomatic Complexity metric score is **8.51** (OUTSTANDING).

The methods of the codebase contain methods with a low McCabe complexity. This leads to methods that are easy to understand and test.

3.2.2 Halstead Volume

The Halstead volume metric evaluates the size of each method in terms of their length and vocabulary. Length and vocabulary are counted based on operators and operands. The methods with a high volume are difficult to read through, understand, test and reuse. Therefore, a maintainable system should contain methods with a low volume.

The codebase has in total 1569 methods and the Halstead volume of those methods reaches a maximum of 1845.9. Based on the measurements of each method and the FrontEndART benchmark, the Halstead Volume metric score is **8.47** (OUTSTANDING).

The methods of the codebase are short and use a limited vocabulary. This contributes to methods that are easy to understand and test.

3.2.3 Nested Level Else-If

This metric expresses the complexity of methods as the depth of the maximum embeddedness of its conditional and iteration block scopes, where in the if-else-if construct only the first if instruction is considered. A method with deeply embedded programming constructions is difficult to understand and test for functional suitability. For a maintainable system, the depth of embeddedness must be kept minimal.

The codebase has in total 1569 methods and the maximum level of program construction embeddedness in those methods reaches a maximum of 1. Based on the measurements of each method and the FrontEndART benchmark, the Nested Level Else-If metric score is **6.43** (ABOVE MARKET AVERAGE).

The methods of the codebase do not contain deep nesting of programming constructions, like loops and conditional statements. This contributes to methods that are easy to understand and test.

3.2.4 Number of Parameters

This metric measures how many parameters each method of the system has. Methods which have many parameters impose a risk of calling that method incorrectly and it may also indicate that the method implements more than just a single concept. Therefore, for a maintainable software one needs to keep the number of parameters in the signature of the methods low.

The codebase has in total 1569 methods and the maximum number of parameters methods use is 2. Based on the measurements of each method and the FrontEndART benchmark, the Number of Parameters metric score is **8.25** (OUTSTANDING).

The methods of the codebase use just a few parameters in their signature. With this, calling a method is easy, less error prone and it is more likely that those methods implement just a single concept.

3.3 Code Documentation

The Coding Documentation quality attribute indicates the extent to which a given software system is documented as compared to other systems in the FrontEndART benchmark. The TechEmpower – FrameworkBenchmarks codebase scores **5.64** (MARKET AVERAGE) on this quality attribute. This codebase belongs to one of the best documented systems in the benchmark.

3.3.1 API Documentation

This metric calculates the ratio of the number of documented public methods in the class (+1 if the class itself is documented) to the number of all public methods in the class (+ 1 the class itself); however, the nested, anonymous, and local classes are not included. The result of this calculation shows how well the public API of a class is documented. API documentation contributes a lot to understanding how that API should be used. Therefore, for an easy-to-understand and maintainable system the APIs must be properly documented.

The codebase has a total of 567 classes and the ratio of the documented public methods ranges between 0% and 100%. Based on the measurements of each class and the FrontEndART benchmark, the API Documentation metric score is **6.09** (ABOVE MARKET AVERAGE).

Public methods are properly documented. This helps the users of the APIs understand how those APIs should be used properly.

3.3.2 Comment Density

This metric measures the ratio of the comment lines of the class to the sum of its comment and logical lines of code. In other words, the percentage of comment lines is calculated for each class. Documentation can help to make the code of a class more understandable and maintainable because it provides extra information. The importance of comments increases with the size of the class.

The codebase has 567 classes in total and the ratio of comment lines in classes ranges between 0% and 67%. Based on the measurements of each class and the FrontEndART benchmark, the Comment Density metric score is **4.51** (MARKET AVERAGE).

An example of a class with a low percentage of comment lines can be found in the Java/Wicket component involving the frameworks/Java/wicket/src/main/java/hellowicket/dbupdates/**HelloDbUpdatesResource.java** file. Only 8% of the **HelloDbUpdatesResource** class implementation contains comments. This class has a single long method called **respond** with **54** LLOC. This method is difficult to understand because of its sheer size. Refactoring to smaller methods would be the ultimate solution to improve maintainability, but at least some helping comment should be added to the current version.

The following figure shows the first part of the implementation of this class:

```java
public class HelloDbUpdatesResource implements IResource
{
  private static final long serialVersionUID = 1L;

  private static final int DB_ROWS = 10000; PMD UUINL

  @Override
  public void respond(Attributes attributes)
  {
    int _queries = attributes.getRequest().getQueryParameters().getParameterValue("queries").toInt(1);
    if (_queries < 1)
    {
      _queries = 1;
    }
    else if (_queries > 500)
    {
      _queries = 500;
    }
    final int queries = _queries;

    try
    {
      final ThreadLocalRandom random = ThreadLocalRandom.current();
      DataSource dataSource = WicketApplication.get().getDataSource();

      World[] worlds = new World[queries];
      try (Connection connection = dataSource.getConnection();
           PreparedStatement query = connection.prepareStatement(
                   "SELECT * FROM World WHERE id = ?",
                   ResultSet.TYPE_FORWARD_ONLY,
                   ResultSet.CONCUR_READ_ONLY);
           PreparedStatement update = connection.prepareStatement(
                   "UPDATE World SET randomNumber = ? WHERE id= ?"))
      {
        for (int i = 0; i < queries; i++)
        {
          query.setInt(1, random.nextInt(DB_ROWS) + 1);
          World world;
          try (ResultSet resultSet = query.executeQuery())
          {
            resultSet.next();
            world = new World(
                resultSet.getInt("id"),
```

Figure 3: **Example for a class with low Comment Coverage**

To decrease the negative effect of purely commented long classes on maintainability, the problematic ones need to be refactored, for example by providing more and useful comments. The full list of classes with a low comment density can be found on the website of the QualityGate. From these, the following list contains the longest classes with no comments at all.

Class	Logical LOC	Comment Density
frameworks/Java/netty/src/main/java/hello/**HelloServerHandler**.java	81	0%
frameworks/Java/restexpress/src/main/java/hello/config/**Configuration**.java	80	0%
frameworks/Java/minijax/src/main/java/com/techempower/minijax/**MinijaxBenchmark**.java	70	0%
frameworks/Java/spring-webflux/src/main/java/benchmark/web/**WebfluxHandler**.java	69	0%
frameworks/Java/helidon/src/main/java/io/helidon/benchmark/services/**DbService**.java	67	0%
frameworks/Java/spring-webflux/src/main/java/benchmark/config/**PgClientConfig**.java	63	0%
frameworks/Java/wizzardo-http/src/main/java/com/wizzardo/techempower/**DBController**.java	62	0%
frameworks/Java/proteus/src/main/java/io/sinistral/models/**MessageEncoder**.java	59	0%
frameworks/Java/restexpress/src/main/java/hello/config/**MysqlConfig**.java	55	0%
frameworks/Java/quarkus/pgclient/src/main/java/io/…/resource/pgclient/**DbResource**.java	54	0%

Figure 4**: Top 10 longest classes without any comments**

3.4 Code Size

The Code Size quality attribute indicates the extent to which a given software system is implemented using small and therefore easy-to-understand code elements as compared to other systems in the FrontEndART benchmark. The TechEmpower – FrameworkBenchmarks codebase scores **7.87** (GOOD) on this quality attribute. This codebase in general has smaller code elements than most of the other systems in the benchmark.

3.4.1 Logical Lines of Code

This metric measures the size of the methods by counting the number of non-empty and non-comment code lines of the method; however, its anonymous and local classes are not included. Long methods are more difficult to understand and they often contain implementation of multiple concepts. Such contained concept implementations cannot be reused. For sound maintainability the size of the system methods must be kept small.

The codebase has in total 1569 methods and the maximum number non-empty and non-comment lines in the methods is 63. Based on the measurements of each method and the FrontEndART benchmark, the Logical Lines of Code metric score is **8.17** (OUTSTANDING).

The methods of the codebase contain a low number of lines of code with are not empty and not comment lines. Small methods are easier to read through, understand, test and reuse.

3.4.2 Number of Local Attributes

This metric measures the amount of responsibility each class has by counting the number of local attributes for those classes. The more local attributes a class has, the more difficult concept that class implements. It can also indicate that the class implements multiple concepts. Classes with difficult concept implementations are difficult-to-understand and test. Therefore, from a maintainability point of view, the number of local attributes must be kept low.

The codebase has in total 567 classes and the maximum number of local attributes within a class is 27. Based on the measurements of each class and the FrontEndART benchmark, the Number of Local Attributes metric score is **8.4** (OUTSTANDING).

The number of local attributes in the classes of the codebase are low. This indicates that the responsibility of those classes is likely to be low as well and that the separation of concepts is correctly implemented.

3.4.3 Number of Local Public Attributes

This metric measures how well one of the fundamental concepts of objected-oriented programming, namely encapsulation, is implemented by a class. The more local public attributes a class has, the more open the implementation of that class is and the less information hiding, or encapsulation is achieved. Open class implementations can quickly lead to unwanted couplings and code that is difficult to understand and test. To avoid this, the number of public local class attributes must be kept low.

The codebase has in total 567 classes and the maximum number of local attributes within a class is 7. Based on the measurements of each class and the FrontEndART benchmark, the Number of Local Public Attributes metric score is **6.43** (ABOVE MARKET AVERAGE).

The number of local public attributes in the classes of the codebase are low. This indicates that the encapsulation within the classes is correctly implemented.

3.4.4 Number of Local Methods

This metric inspects the diversity of a class's behavior by counting the number of methods of that class. Classes with many methods are difficult to understand and to identify what exactly that class is responsible for. In such cases it is wise to re-think what that class should do and execute the necessary changes.

The codebase has in total 567 classes and the maximum number for local methods within a class is 21. Based on the measurements of each class and the FrontEndART benchmark, the Number of Local Methods metric score is **8.77** (OUTSTANDING).

The number of local methods in the classes of the codebase are low. This indicates that the responsibility of those classes is likely to be low as well and that the separation of concepts is correctly implemented.

3.5 Code Structure

The Code Structure quality attribute indicates the extent to which a given software system is structured as compared to other systems in the FrontEndART benchmark. The TechEmpower – FrameworkBenchmarks codebase scores **6.84** (ABOVE MARKET AVERAGE) on this quality attribute. This codebase is structured better than an average system in the benchmark.

3.5.1 Response Set for Class

This metric measures the testability and analyzability of classes by counting the sum of the methods to which that class refers to. The more methods supply data to a class, the more difficult it is to test and

analyze that class. To implement a maintainable system, the number of referred methods must be kept low.

The codebase has in total 567 classes and the maximum response set size for a class is 24. Based on the measurements of each class and the FrontEndART benchmark, the Response Set for Class metric score is **9.08** (EXCEPTIONAL).

The number of methods called by the classes of the codebase is typically low. This means that those classes are typically compact, and their lifecycle is less likely to be connected to those of other classes.

3.5.2 Coupling Between Object Classes

This metric measures the size of the environment of a class by counting the number of referred classes. These references can be related to inheritance, method call, type reference, etc. Classes which depend on many other classes are more difficult to test and to reuse. Moreover, these classes are overly sensitive to the changes in their environment. It is therefore needed to keep weak couplings between classes.

The codebase has in total 567 classes and the maximum the classes a class uses is 10. Based on the measurements of each class and the FrontEndART benchmark, the Coupling Between Object Classes metric score is **8.81** (OUTSTANDING).

The number of classes a class references in the codebase is typically low. This means that those classes are less likely to be impacted by a change in their environment.

3.5.3 Lack of Cohesion in Methods 5

This metric is used to identify the number of functionalities of a class. One of the basic principles of object-oriented programming is encapsulation, meaning that attributes belonging together and the operations that use them should be organized into one class, and one class shall implement only one functionality, i.e. its attributes and methods should be coherent.

This metric measures the lack of cohesion and computes into how many coherent classes the class could be split. It is calculated by taking a non-directed graph, where the nodes are the implemented local methods of the class and there is an edge between the two nodes if and only if a common (local or inherited) attribute or abstract method is used or a method invokes another. The value of the metric is the number of connected components in the graph not counting those, which contain only constructors, destructors, getters, or setters. A maintainable system should contain classes with a single functionality.

The codebase has a total of 567 classes and the maximum number for local attributes within a class is 8. Based on the measurements of each class and the FrontEndART benchmark, the Lack of Cohesion in Methods 5 metric score is **8.58** (OUTSTANDING).

The number of functionalities the classes of the codebase implement is typically low. This means that those classes have a high cohesion.

3.5.4 Number of Incoming Invocations

This metric measures the number of other methods and attribute initializations which directly call the local methods of a class. If a method is invoked several times from the same method or attribute initialization, it is counted only once. The more call a class has to its methods, the bigger the calling environment of that class is. Changes to a class with a big calling environment has risks, because they may imply changes to a high number of other classes. Those changes are typically difficult to perform and may lead to inconsistencies and bugs. An easily maintainable software has a low amount of incoming invocations for its classes, especially for those that are bigger in size.

The codebase has in total 567 classes and the maximum number of incoming invocations for a class is 10. Based on the measurements of each class and the FrontEndART benchmark, the Number of Incoming Invocations metric score is **0.85** (NEARLY IMPOSSIBLE).

An example of a class which has many invocations to its local methods can be found in the **Java/vertx-web** component involving the frameworks/Java/vertx-web/src/main/java/io/vertx/benchmark/**Helper .java** file. The local methods of the Helper class in this file are called 10 times. The following figure shows the first couple of lines of this class's implementation:

```
7   public final class Helper {
8
9     private Helper() {
10    }
11
12    /**
13     * Returns the value of the "queries" getRequest parameter, which is an integer
14     * bound between 1 and 500 with a default value of 1.
15     *
16     * @param request the current HTTP request
17     * @return the value of the "queries" parameter
18     */
19    static int getQueries(HttpServerRequest request) { PMD_DP FH_TMR 412~CloneInstance
20      String param = request.getParam("queries");
21
22      if (param == null) {
23        return 1;
24      }
```

Figure 5: **Example of a class with a high number of incoming invocations**

To decrease the negative effect of classes with many incoming invocations on maintainability, the problematic classes need to be refactored, for example by splitting them up. The full list of classes with a high number of incoming invocations can be found on the website of the QualityGate. From these the following list contains the ones with the most incoming invocations.

Class	Number of Incoming invocations
frameworks/Java/vertx-web/src/main/java/io/vertx/benchmark/**Helper**.java	10
frameworks/Java/greenlightning/src/main/java/com/javanut/gl/benchmark/**ResultObject**.java	10
frameworks/Java/greenlightning/src/main/java/com/javanut/gl/benchmark/**PoolManager**.java	6
frameworks/Java/dropwizard/src/main/java/com/example/helloworld/resources/**Helper**.java	6
frameworks/Java/proteus/src/main/java/io/sinistral/controllers/**Benchmarks**.java	6
frameworks/Java/dropwizard/src/main/java/com/example/helloworld/db/model/**World**.java	6
frameworks/Java/revenj-jvm/src/main/java/hello/**Utils**.java	5
frameworks/Java/wizzardo-http/src/main/java/com/wizzardo/techempower/**DBService**.java	5
frameworks/Java/quarkus/hibernate/src/main/java/io/quarkus/benchmark/…/**WorldRepository**.java	4
frameworks/Java/jooby/src/main/java/com/techempower/**Json**.java	4

Figure 6**: Top 10 classes with the highest number of incoming invocations**

3.6 Code Duplication

The Code Duplication quality attribute indicates the extent to which the code of a given software system is duplicated as compared to other systems in the FrontEndART benchmark. The TechEmpower – FrameworkBenchmarks codebase scores **7.11** (GOOD) on this quality attribute. The duplication related technical risks for this codebase is less severe than those for most of the other systems in the benchmark.

3.6.1 Clone Coverage

Two code segments are considered to be duplicates of each other when their code structures are the same (type 2 clones). The clone coverage metric measures the ratio of duplicated code within each class. To determine those ratios, syntactic entities (statements, expressions, etc.) are counted. The higher the clone coverage is within a class; the more code must be maintained in a consistent way. Maintaining duplicated code is error prone due to the likely inconsistent changes. It takes also more time and leads to an unnecessary increase of system size. Furthermore, duplicated code cannot be reused unless the entire class or method is a duplicate.

The codebase has in total 567 classes and the ratio of duplicated code in classes goes up to 100% in some cases. Based on the measurements of each class and the FrontEndART benchmark, the Clone Coverage metric score is **5.91** (MARKET AVERAGE).

An example of a class with a high clone coverage ratio can be found in the **Java/undertow-jersey** component involving the frameworks/Java/undertow-jersey/src/main/java/hello/**WorldResource.java** file. In this file **85%** of the implementation of the **WorldResource** class is covered by duplicates. The following figure shows the first method from this class:

```
25      @Singleton
26      @Produces(MediaType.APPLICATION_JSON)
27      @Path("/db")
28      public class WorldResource {
29        @Inject
30        private EntityManagerFactory emf; FH UPF
31
32        @GET 604~CloneInstance
33        public Object db() throws InterruptedException, ExecutionException { FH SMN
34          Callable<World> callable = () -> {
35            EntityManager em = emf.createEntityManager();
36            Session session = em.unwrap(Session.class);
37            session.setDefaultReadOnly(true);
38            try {
39              return (World) session.byId(World.class).load(randomWorld());
40            } finally {
41              session.close();
42            }
43          };
44          Future<World> futureWorld = Common.EXECUTOR.submit(callable); FH NFSVMBSB
45          return futureWorld.get();
46        }
```

Figure 7: **Example of a class with a high percentage of clone coverage**

By relying on and using the fundamental concepts of object-orientation (for example: inheritance), or common method creation one could reduce the redundancy introduced. The full list of the detected class-level duplications can be found on the website of the QualityGate. From these the following list contains the ones with the highest clone coverage.

Class	Clone Coverage
frameworks/Java/officefloor/src/woof_benchmark_raw/src/main/java/net/officefloor/benchmark/RawOfficeFloorMain.java:**DefaultMustacheFactory**	100%
frameworks/Java/grizzly/src-jersey/main/java/hello/WebServer.java:**SimplifyAddOn**	96%
frameworks/Java/grizzly/src/main/java/org/glassfish/grizzly/bm/Server.java:**SimplifyAddOn**	96%
frameworks/Java/tapestry/src/main/java/hello/services/**DevelopmentModule**.java	95%
frameworks/Java/play2-java/play2-java-ebean-hikaricp/app/startup/**Startup**.java	90%
frameworks/Java/play2-java/play2-java-jooq-hikaricp/app/startup/**Startup**.java	90%
frameworks/Java/play2-java/play2-java-jpa-hikaricp/app/startup/**Startup**.java	90%
frameworks/Java/proteus/src/main/java/io/sinistral/proteus/controllers/handlers/**BenchmarksRouteSupplier**.java	89%
frameworks/Java/undertow-jersey/src/main/java/hello/**WorldResource**.java	85%
frameworks/Java/grizzly/src-jersey/main/java/hello/**WorldResource**.java	85%

Figure 8: **Top 10 classes with the highest percentage of clone coverage**

3.6.2 Clone Instances

This metric measures the number of clone instances for each class of the system. Each clone instance is a piece of code that can be found somewhere else in the system with the same structure (type 2 duplicates). Clone instances can exist in the same class too. The more clone instances there are in a class, the higher

the chance is that changing another class with the same clone instances will lead to a change to that class. In other words, high number of clone instances in a class results in strong couplings between the class and its environment. Since there are no evident signs of these couplings, maintaining them are risky and can cause inconsistent changes.

The codebase has in total 567 classes and the maximum number of clone instances within a class is 11. Based on the measurements of each class and the FrontEndART benchmark, the Clone Instances metric score is **7.91** (GOOD).

The number of clone instances in the classes of the codebase are low. This is an indication that common code segments are properly separated and reused.

3.6.3 Clone Complexity

The clone complexity metric measures how complex the implementation of a class is that belongs to the clone instances of that class. This is done by taking the sum of McCabe complexity of the clone instances in the class. Complex clones are difficult to understand and to refactor. High complexity clones also increase the chance that the related clone instances will not be changed in a consistent way and that therefore a bug will be introduced.

The codebase has in total 567 classes and the maximum level of clone complexity within a class is 12 McCabe points. Based on the measurements of each class and the FrontEndART benchmark, the Clone Complexity metric score is **8.21** (OUTSTANDING).

Even in those classes where clone instances are present, the complexity of the code within those clone instances are low. Therefore, the effects of the existing clone instances are less severe.

4 Technical Summary

The maintainability of the TechEmpower – FrameworkBenchmarks codebase is **7.06** on the scale from 0 (score of the least maintainable system) to 10 (score of the most maintainable system). This score means that the maintainability of the TechEmpower – FrameworkBenchmarks codebase is GOOD considering the FrontEndArt benchmark which includes hundreds of open source and closed source systems.

From the 6 quality attributes of the FrontEndART Maintainability Model, the TechEmpower – FrameworkBenchmarks codebase performs the worst on the Code Documentation quality attribute with a score of **5.64**. However, even this score indicates that the documentation of the codebase's code is at the MARKET AVERAGE level based on the FrontEndART benchmark.

At the level of metrics, there are 3 metrics where the TechEmpower – FrameworkBenchmarks codebase performs less than ABOVE MARKET AVERAGE as compared to the FrontEndART banchmark. These are the following:

Metric	Score
Number of Incoming Invocations	**0.85**
Comment Density	**4.51**
Clone Coverage	**5.91**

This means that in the codebase analyzed, there are classes with a high number of incoming invocations, low percentage of comment coverage and a high percentage of clone coverage to be refactored. Where to look for the most problematic classes are described in Sections 3.5.4, 3.3.2 and 3.6.1.

All the other 17 metrics indicate that from their perspective the maintainability of the TechEmpower – FrameworkBenchmarks codebase is ABOVE MARKET AVERAGE or even better.

5 Technical Advice

To improve the maintainability of the TechEmpower – FrameworkBenchmarks codebase even further we advise addressing those issues which belong to the metrics with a less than ABOVE MARKET AVERAGE score. These metrics are enumerated in Section 5. Based on this, FrontEndART provides the following piece of advice:

- **Decrease the number of incoming invocations of the classes with the highest number of incoming calls. For example, by redesigning class responsibilities and splitting up class implementations.**

- **Provide more comments for long classes and methods to make them better to understand. Even better, refactor long classes and methods into smaller ones.**

- **Decrease the clone coverage percentage of the classes from the Java Officefloor, Grizzly, Tapestry, Play2, Proteus and Undertow-jersey components by factoring out and reusing duplicated code segments.**

To make sure that the maintainability improves as required and stays at an acceptable level, we suggest monitoring the TechEmpower – FrameworkBenchmarks codebase continuously. Such a monitoring service is present within the service portfolio of FrontEndART Ltd. and we look forward to helping you manage maintainability on the long run.