Collaborative Project
Holistic Benchmarking of Big Linked Data
Project Number: 688227  Start Date of Project: 2015/12/01  Duration: 36 months

Deliverable 2.2.2
Second Version of the HOBBIT Platform

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Due Date of Deliverable</td>
<td>Month 27, 28/02/2018</td>
</tr>
<tr>
<td>Actual Submission Date</td>
<td>Month 27, 28/02/2018</td>
</tr>
<tr>
<td>Work Package</td>
<td>WP2 - Benchmarking Platform</td>
</tr>
<tr>
<td>Task</td>
<td>T2.2</td>
</tr>
<tr>
<td>Type</td>
<td>Demonstrator</td>
</tr>
<tr>
<td>Approval Status</td>
<td>Final</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Number of Pages</td>
<td>62</td>
</tr>
</tbody>
</table>

Abstract: This deliverable presents the second version of the HOBBIT platform.

The information in this document reflects only the author’s views and the European Commission is not liable for any use that may be made of the information contained therein. The information in this document is provided "as is" without guarantee or warranty of any kind, express or implied, including but not limited to the fitness of the information for a particular purpose. The user thereof uses the information at his/ her sole risk and liability.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 688227.
History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason</th>
<th>Revised by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>24/01/2018</td>
<td>First draft created</td>
<td>Michael Röder</td>
</tr>
<tr>
<td>0.2</td>
<td>20/02/2018</td>
<td>Reviewed version created</td>
<td>Pavel Smirnov</td>
</tr>
<tr>
<td>1.0</td>
<td>23/02/2018</td>
<td>Final version created</td>
<td>Michael Röder</td>
</tr>
</tbody>
</table>

Author List

<table>
<thead>
<tr>
<th>Organization</th>
<th>Name</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>InfAI</td>
<td>Michael Röder</td>
<td><a href="mailto:roeder@informatik.uni-leipzig.de">roeder@informatik.uni-leipzig.de</a></td>
</tr>
<tr>
<td>USU</td>
<td>Roman Korf</td>
<td><a href="mailto:r.korf@usu.de">r.korf@usu.de</a></td>
</tr>
<tr>
<td>InfAI</td>
<td>Axel-Cyrille Ngonga Ngomo</td>
<td><a href="mailto:ngonga@informatik.uni-leipzig.de">ngonga@informatik.uni-leipzig.de</a></td>
</tr>
<tr>
<td>AGT</td>
<td>Martin Strohbach</td>
<td><a href="mailto:MStrohbach@agtinternational.com">MStrohbach@agtinternational.com</a></td>
</tr>
<tr>
<td>AGT</td>
<td>Pavel Smirnov</td>
<td><a href="mailto:PSmirnov@agtinternational.com">PSmirnov@agtinternational.com</a></td>
</tr>
<tr>
<td>OpenLink</td>
<td>Milos Jovanovik</td>
<td><a href="mailto:mjovanovik@openlinksw.com">mjovanovik@openlinksw.com</a></td>
</tr>
</tbody>
</table>
Executive Summary

This document describes the second version of the HOBBIT platform in detail. It gives a brief introduction before it gives an overview over the architecture in 2. In Section 3, the single work flows for benchmarking a system or executing a challenge are described. Finally, the evaluation of the platform is described in Section 5.
Contents

1 Introduction  5

2 Architecture  5
  2.1 Benchmarking a System ................................................. 6
  2.2 Platform Components .................................................. 8
    2.2.1 Platform Controller ............................................. 8
    2.2.2 Storage ............................................................ 10
    2.2.3 Front End ......................................................... 10
    2.2.4 User Management ............................................... 11
    2.2.5 Repository ....................................................... 11
    2.2.6 Analysis .......................................................... 11
    2.2.7 Message Bus ..................................................... 11
    2.2.8 Logging ........................................................... 12
  2.3 Benchmark Components ............................................... 13
    2.3.1 Benchmark Controller .......................................... 13
    2.3.2 Data Generator .................................................. 14
    2.3.3 Task Generator ................................................... 15
    2.3.4 Evaluation Storage ............................................. 16
    2.3.5 Evaluation Module ............................................. 16
  2.4 Benchmarked System Components .................................... 17
  2.5 Ontology ..................................................................... 17
    2.5.1 Classes ............................................................ 18
    2.5.2 Properties .......................................................... 24

3 User Manuals  25
  3.1 Register ................................................................. 26
  3.2 Upload a system ...................................................... 27
    3.2.1 Uploading the Docker image ..................................... 27
    3.2.2 System metadata file ............................................ 28
  3.3 Upload a benchmark .................................................. 30
  3.4 Benchmark a system ................................................ 33
  3.5 Experiment status ..................................................... 36
  3.6 Explore experiment results ........................................... 37
3.7 Register for a challenge .................................................. 37
3.8 Create a challenge .......................................................... 39

4 Java SDK ............................................................................. 50

4.1 Architecture .................................................................... 50
  4.1.1 RabbitMQ Dockerizer ..................................................... 50
  4.1.2 Command Queue Listener .............................................. 51
  4.1.3 Components Executor .................................................. 51
4.2 Class diagram .................................................................... 51
  4.2.1 Dockerizer .................................................................... 52
  4.2.2 Pull-based dockerizer .................................................... 52
  4.2.3 Build-based dockerizer .................................................. 52
  4.2.4 Dockerizer Builder ........................................................ 52
  4.2.5 Component-specific DBs ................................................. 52
4.3 Development using the SDK ............................................. 52
  4.3.1 Test benchmark components as pure java codes ................. 53
  4.3.2 Test dockerized benchmark components .......................... 54
  4.3.3 Test a benchmarked system under third-party benchmark .... 55

5 Evaluation ............................................................................ 58

5.1 Triple store benchmark use case ......................................... 58
5.2 Knowledge Extraction benchmark use case ........................ 59

References .............................................................................. 60

6 Appendix A: changes from version 1 to version 2 .................... 61

6.1 Features ............................................................................ 61
6.2 Fixes ................................................................................. 61
Introduction

This document describes the second version of the HOBBIT platform. The platform serves as a framework for benchmarking Linked Data systems. Both benchmarks focusing the evaluation of the quality of a system using single consecutive requests as well as benchmarks aiming at the efficiency (e.g., by generating a lot of parallel requests leading to a high work load) can be run on the platform. Especially for the latter case, the platform supports the handling of Big Linked Data to make sure that even for scalable systems a maximum load can be generated.

The HOBBIT platform included in the HOBBIT project aims at two goals. Firstly, we offer an open source evaluation platform that can be downloaded and executed locally. The open-source projects related to the platform are listed in Table 1. Secondly, we offer an online instance of the platform for a) running public challenges and b) making sure that even people without the required infrastructure are able to run the benchmarks they are interested in. The online instance (i.e., the demonstrator) can be accessed via http://master.project-hobbit.eu. Its Gitlab instance has the address https://git.project-hobbit.eu while the public SPARQL endpoint offering the data is available at http://db.project-hobbit.eu/.

<table>
<thead>
<tr>
<th>Project</th>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td><a href="https://github.com/hobbit-project/core">https://github.com/hobbit-project/core</a></td>
<td>Library containing core functionalities that ease the integration into the platform</td>
</tr>
<tr>
<td>Platform</td>
<td><a href="https://github.com/hobbit-project/platform">https://github.com/hobbit-project/platform</a></td>
<td>The HOBBIT platform and a wiki containing tutorials.</td>
</tr>
<tr>
<td>Ontology</td>
<td><a href="https://github.com/hobbit-project/ontology">https://github.com/hobbit-project/ontology</a></td>
<td>The HOBBIT ontology described in Section 2.5</td>
</tr>
<tr>
<td>Default evaluation storage</td>
<td><a href="https://github.com/hobbit-project/evaluation-storage">https://github.com/hobbit-project/evaluation-storage</a></td>
<td>A default implementation of the evaluation storage component which can be used by benchmark developers.</td>
</tr>
</tbody>
</table>

Table 1: List of open source projects created during T2.2.

This document is structured in the following way. The following Section 2 describes the architecture of the platform. In Section 3, the single work flows for benchmarking a system or executing a challenge are described. In Section 4, the usage of the Java SDK is explained. Finally, the evaluation of the platform is described in Section 5. In the appendix, a list of major differences between the first and second version of the platform can be found. Throughout the document, the term "system" refers to the system that is benchmarked and an "experiment" is a single execution of a benchmark to evaluate a single system.

Architecture

The HOBBIT platform comprises of several components that are deployed as Docker containers. These components can be separated into two groups. The first group comprises of platform components that are always running. The second group contains all components that belong to a certain experiment.

---

1[^docker]: https://docker.com/
i.e., components of the benchmark as well as components of the benchmarked system. Figure 1 provides an overview over the components and their relationships. Platform components are marked blue while the orange components are the components of a certain benchmark.

The dataflow shown in the overview picture is carried out using the RabbitMQ message bus.\footnote{https://www.rabbitmq.com/} However, the HOBBIT platform is modular in a way that allows the benchmark to use a different middleware. For the communication between the benchmark components as well as for the communication with the benchmarked system the benchmark may use network sockets or a streaming framework. For the communication with the platform controller it has to use the offered RabbitMQ message queues. However, throughout the document it is assumed that the benchmark components rely on RabbitMQ as well.

![Figure 1: Overview of the platform components.](image)

**Benchmarking a System**

In this section the general workflow of benchmarking a system is described. For keeping the workflow simple, the creation of Docker containers has been simplified and it is assumed that all necessary images and metadata is available in the repository of the platform. Figure 2 shows a sequence diagram containing the steps as well as the type of communication that is used. However, the orchestration of the single benchmark components is part of the benchmark and might be different.

1. The platform controller makes sure that a benchmark can be started. This includes a check to make sure that all nodes of the cluster are available.

2. The platform controller spawns the components of benchmarked system.
   - The system initializes itself and makes sure that it is working properly.
   - It sends a message to the platform controller to indicate that it is ready.

3. The platform controller spawns the benchmark controller.
Figure 2: Overview of the general benchmarking workflow. For keeping simplicity, internals of benchmarked system as well as the front end components of the platform are left out.

- The benchmark controller spawns the data and task generators as well as the evaluation storage.
- The benchmark controller sends a message to the platform controller to indicate that it is ready.

4. The platform controller waits until the system as well as the benchmark controller are ready.

5. The platform controller sends a start signal to the benchmark controller which starts the data generators.

6. The data generators start their algorithms to create the input data.
   - The data is sent to the system and to the task generators.
   - The task generators generate the tasks and send it to the system.
   - The systems response is sent to the evaluation storage.

7. The task generators send the expected result in the evaluation storage.

8. After the data and task generators finished their work the benchmarking phase ends and the generators as well as the benchmarked system terminate.
9. After that the terminated components are discarded and the benchmark controller spawns the evaluation module.

10. The evaluation module loads results from the evaluation storage. This is done by requesting the results pairs, i.e., expected result and actual result received from the system for a single task, from the storage. The evaluation module uses these pairs to evaluate the system’s performance and calculate the Key Performance Indicators (KPIs). The results of this evaluation are returned to the benchmark controller before the evaluation module terminates.

11. The benchmark controller sends the signal to the evaluation storage to terminate.

12. The benchmark controller sends the evaluation results to the platform controller and terminates.

13. After the benchmark controller has finished its work, the platform controller can add additional information to the result, e.g., the configuration of the hardware, and store the result. After that, a new evaluation could be started.

14. The platform controller sends the URI of the experiment results to the analysis component.

15. The analysis component reads the evaluation results from the storage, processes them and stores additional information in the storage.

Apart of the described orchestration scheme the platform will support other schemes as well. For example, it will be possible to generate all the data in a first step before the task generators start to generate their tasks based on the complete data. In another variant, the task generators are not only sending the generated task but are waiting for the response before sending the next task.

Platform Components

The platform comprises several different components.

Platform Controller

The platform controller is the central component of the HOBBIT platform coordinating the interaction of other components if needed. This mainly includes the handling of requests that come from the front end component, starting and stopping of benchmarks, observing of cluster health status and triggering of the analysis component.

Persistent Status

The internal status of the platform controller is stored in a Redis database. This enables the shut down or restart of a platform controller without loosing current status, e.g., benchmarks that have already been configured and pushed into the execution queue.

Queue

Experiments submitted by users obtain a unique HOBBIT ID and are put into the execution queue managed by the platform controller. This queue is a "First in, First out" queue, i.e., experiments are executed in the same order in which they have been added. However, the platform controller
guarantees, that the experiments of a certain challenge are executed on a certain date. Thus, the order of the experiments change as soon as the execution date of a challenge has come. In this case, the experiments of the challenge are executed first.

Benchmark Execution

The platform controller contains a queue of experiments, i.e., benchmark and system combinations. If there is no running experiment and the queue is not empty, the platform controller initiates the execution of an experiment in the following way:

1. The platform controller makes sure that a benchmark can be started. This includes a check to make sure that the system/cluster is healthy.
2. The platform controller loads the maximum runtime the benchmark should get.
3. The platform controller spawns the benchmarked system. This includes a check whether the benchmark defines parameters that should be forwarded to the system.
4. The platform controller spawns the benchmark controller.
5. The system as well as the benchmark controller load all the data that they need and make sure that they are working properly. Afterwards they send a message to the platform controller using the command queue to indicate that they are ready. The platform controller is waiting for these messages.
6. After both the system and the benchmark controller are ready, the platform controller sends a start signal to the benchmark controller. With this signal, the platform controller stops observing the system container. It is assumed that the benchmark controller takes care of sending a termination signal to the system.
7. The platform controller receives the results from the benchmark controller.
8. The results are extended with additional information about the hardware before they are send to the storage.

The platform controller observes the state of the benchmark controller. If the experiment takes more time than a configured maximum, the platform controller terminates the benchmark controller and all the containers that belong to it. Since the platform controller manages the creation of containers it has a list of created containers that belong to the currently running benchmark. This list is used to make sure that all resources are freed and the cluster is ready for the next experiment.

Challenge management

The platform controller manages several actions necessary to support the workflow of challenges. Firstly, it makes sure that if a challenge is closed every system is benchmarked for the challenge task it has been registered for.

Secondly, it takes care of the publication of challenge results. The platform offers the definition of a distinct publication date for a challenge. Before this date is reached, the complete set of results is only visible for the organizer of a challenge. When the date is reached, the results are made public. This mechanism ensures that the challenge organizer can already gather the winners of the single task.
challenges for announcing them during a ceremony while nobody else can get the complete list of results.

Thirdly, the platform controller takes care of executing repeatable challenges. These challenges are special in the way that there is no one single date at which the registration of systems closes, but there are several dates at which the registered systems are benchmarked. This allows the challenge organizer to run its challenge over several months with intermediate results and a public leaderboard.

Interaction with Analysis Component

The platform controller triggers the analysis component sending the URI of an terminated experiment to inform the analysis component that new results for this combination are available.

Docker Container Creation

The platform controller is the only component that has direct access to the docker daemon. If another component would like to start a docker container, it has to send a request to the platform controller containing the image name and parameters. Thus, the platform controller offers the central control of commands that are sent to the docker daemon which increases the security of the system.

Resource usage gathering

As a part of the API offered by the platform controller the resource usage information of the currently benchmarked system can be queried. This enables the benchmarks to measure the systems efficiency not only based on the response times but also based on the resource statistics. This statistics contains the sum of a) CPU time, b) main memory and c) disk space that the containers of the system are consuming.

Storage

The storage component contains the experiment results. It comprises two containers—a Virtuoso triple store that uses the HOBBIT ontology to describe the results and a Java program that handles the communication between the message bus and the triple store. This communication can be encrypted to make sure that no invasiv benchmark or system image can change the data in the storage. The storage component offers a public SPARQL Endpoint with read-only access.

Front End

The front end component handles the interaction with the user. It offers different functionalities to different user groups. Thus, it contains a user management that allows different roles for authenticated users as well as a guest role for unauthenticated users.

Roles

**Guest.** A guest is an unauthenticated user. This type of user is only allowed to read the results of experiments and analysis.
Registered user. This user is allowed to upload system images and start benchmarks for his own systems. Additionally, the user can register its systems for challenge tasks.

Challenge Organizer. This user is allowed to organize a challenge, i.e., define challenge tasks with a benchmark and a certain date at which the experiments of the challenge will be executed.

User Management

For the user management, we use Keycloak\(^3\). It stores the user account information and handles the authentication.

Repository

The platform comprises a Gitlab\(^4\) instance. This instance hosts the meta data files of systems and benchmarks. Additionally, it is used to enable the users to upload their Docker images of their systems and benchmarks.

Analysis

This component is triggered after an experiment has been carried out successfully. Its task is to enhance the benchmark results by combining them with the features of the benchmarked system and the data or task generators. These combination can lead to additional insights, e.g., strengths and weaknesses of a certain system.

Message Bus

This component contains the message bus system. There are several messaging systems available from which we chose RabbitMQ. Table 2 shows the different queues that are used by the platform components. We will use the following queue types supported by RabbitMQ:

- **Simple.** A simple queue has a single sending component and a single receiving consumer.
- **Bus.** Every component connected to this queue receives all messages sent by one of the other connected components.
- **RPC.** The queue has one single receiving consumer that handles incoming requests, e.g., a SPARQL query, and sends a response containing the result on a second parallel queue back to the emitter of the request.

Queues that are used exclusively by the benchmarking components or the benchmarked system are not listed in table 2, since their usage depends on a particular benchmark. However, if the benchmark relies on RabbitMQ, the benchmark implementation has to add the ID of the experiment to the execution queue in order to be executed in parallel with other submitted benchmarks.

The **hobbit.command** queue is used to connect the loosely coupled components and orchestrate their activities. Since the platform should be able to support the execution of more than one experiment in parallel, we will use a simple addressing scheme to be able to distinguish between platform

\(^3\)http://www.keycloak.org/
\(^4\)https://about.gitlab.com/
components and benchmark components of different experiments. Table 3 shows the structure of a command message. Based on the Hobbit ID at the beginning of the message, a benchmark component can decide whether the command belongs to its experiment. The message contains a byte that encodes the command that is sent. Based on this command a component that belongs to the addressed experiment decides whether it has to react to this message. Additional data can be appended as well if necessary.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..3</td>
<td>int</td>
<td>Length $h$ of the Hobbit ID.</td>
</tr>
<tr>
<td>4..$h + 3$</td>
<td>String</td>
<td>Hobbit ID of the experiment this command belongs to.</td>
</tr>
<tr>
<td>$h + 4$</td>
<td>byte</td>
<td>ID of the command.</td>
</tr>
<tr>
<td>$&gt; h + 4$</td>
<td>byte[]</td>
<td>Additional data (optional)</td>
</tr>
</tbody>
</table>

Table 3: Structure of a message of the hobbit.command.

Logging

The logging comprises of three single components—Logstash, Elasticsearch and Kibana. While Logstash is used to collect the log message from the single components, Elasticsearch is used to store them inside a fulltext index. Kibana offers the front end for accessing this index.

Logging inside Components

Single components should write log messages to a standard output. Docker containers will be configured to send the standard outputs to the Logstash instance.

We encourage the usage of a facade, e.g., slf4j\(^5\), in the program code. This offers the advantage

\(^5\)http://www.slf4j.org/
that a certain implementation, e.g., log4j, can be chosen without influence on the already written code. Additionally, all components should share a single pattern for the log messages to ease the analysis of log messages and the configuration of Kibana.

**Benchmark Components**

These components are instantiated for a particular experiment and should be destroyed when the experiment terminates.

The benchmark components are—like all other components—encapsulated into Docker containers. Thus, the different containers are decoupled from each other and can be implemented in different languages. However, to ease the implementation of benchmarks, we offer abstract Java classes that implement the basic workflow of the single components.⁶

**Benchmark Controller**

A benchmark controller is the central component of an experiment. It creates and controls data generators, task generators, evaluation storage and evaluation module.

**Workflow**

1. The platform controller spawns a benchmark controller the with the following parameters:
   - An identifier of an experiment (HOBBIT ID)
   - An identifier of a Docker container of the system (ID)
   - A benchmark specific parameters.

2. The benchmark controller initializes itself and connects to the `hobbit.command` queue.

3. The benchmark controller waits for the system to send the start signal (`hobbit.command`).

4. It creates data and task generators as well as the evaluation storage (Docker). It waits for single components to report that they are ready (`hobbit.command`).

5. It sends a start signal to the spawned benchmark components (`hobbit.command`).

6. It waits for all the spawned data generators to end (Docker).

7. It sends a signal to the spawned task generators that all data generators have terminated (`hobbit.command`).

8. It waits for all the spawned task generators to end (Docker).

9. It sends a signal to the system that all the spawned task generators have terminated (`hobbit.command`).

10. It waits for the system to end (Docker).

11. It spawns an evaluation module (Docker).

---

⁶The basic functionalities can be used by including the core library that can be found at [https://github.com/hobbit-project/core](https://github.com/hobbit-project/core).
12. It waits for the spawned evaluation module to finish (Docker).

13. It sends a termination signal to the spawned evaluation module (hobbit.command) and waits for it to finish (Docker).

14. It sends the results received from the evaluation module to the platform controller and finishes with an exit code (0 in case of success).

Error Handling

The error handling inside benchmark components is important since the occurrence of an error can lead to a wrong evaluation result. Thus, benchmark components need to have exact rules how they should act in case of a severe error.

All benchmark components are observed by a special component that waits for their termination. Thus, if some benchmark component encounters a severe error that can not be handled inside the component and will lead to a wrong benchmark result, the component will be terminated with an exit code $> 0$.

A benchmark controller will be informed by a platform controller that one of components exited with an error code. The benchmark controller has to send a stop signal to all of the benchmark components and wait for their termination. If the benchmark controller has been initialized and already sent the message to the platform controller that it is ready, it has to stop the benchmarked system as well. Before it terminates the benchmark controller should send the error to the platform controller and terminate with a status $> 0$. The platform controller will save the error in the storage to give the feedback to the user that an error occurred.

If a benchmark controller has a faulty implementation and does not handle the error as described above the platform controller implements a fallback strategy. Firstly, a benchmark that does not terminate during a configured time frame is stopped by the platform controller. Thus, even if the combination of a crashed benchmark component and the faulty error handling lead to a deadlock of the benchmark, the platform itself will continue as soon as the configured timeout will been reached. Secondly, the platform controller comprises a mapping of the benchmark components to the Hobbit ID of the experiment. Thus, even if the benchmark controller is not able to terminate all its components, the platform controller will be able to stop the components using this mapping.

Data Generator

The data generator contains some data generation algorithm (e.g. static file-based, dynamic mimicking, etc.) and able to generate the data needed for the evaluation. In case of benchmarking of Big Data engines a data generator have to provide a highly intensive input stream and thus it might be instantiated several times. Each instance will receive an identifier (ID) that should be used to generate different data.

Additionally, a data generation needs to be repeatable, i.e., a data generator that is run a second time with the same configuration has to produce exactly the same data as it did during its first execution. Thus, if the generation is relying on pseudo-random processes a seed should be used.

During a benchmark experiment, the data generator typically performs the following steps:

1. It is created by the benchmark controller with the following parameters:
• An identifier of the experiment (HOBBIT ID)
• An identifier of the particular generator instance (ID) as well as the number of generators
• benchmark specific parameters (optional)

2. It initializes itself and connects to the hobbit.command queue.
3. It sends the ready signal to the benchmark controller (hobbit.command).
4. It waits for the start signal (hobbit.command).
5. It generates data based on the given parameters.
6. It terminates with status code 0.

**Task Generator**

Task generators receive data from data generators and generate tasks that can be identified with an identifier (taskID). The taskID is needed to map system’s responses to expected responses during the evaluation, i.e., a task comprising a SPARQL query should have the same task ID as the expected result of the query. The generated tasks are sent to the system while the expected responses are sent to a spawned evaluation storage.

During a benchmarking experiment the task generator typically performs the following steps:

1. It is spawned by the benchmark controller with the following parameters:
   • An identifier of the experiment (HOBBIT ID)
   • An identifier of this particular generator instance (ID) as well as the number of generators
   • benchmark specific parameters (optional)

2. It initializes itself and connects to a hobbit.command queue.
3. It sends the ready signal to the benchmark controller (hobbit.command).
4. It waits for the start signal (hobbit.command).
5. It generates tasks.
   • It receives data tuples, which come from data generators.
   • It generates a task and the expected solution.
   • It sends the task to a benchmarked system.
   • It sends the expected solution to the evaluation storage.

6. If the DATA_GEN_TERMINATE signal is received (hobbit.command) it consumes all data that is still available.

7. It terminates with exit code (0 in case of success).
Evaluation Storage

The evaluation storage is a component that stores the gold standard results as well as the responses of the benchmarked system during the computation phase. During the evaluation phase it sends this data to the evaluation module. The ID of the task is used as the key, while the value comprises the expected result as well as the actual result returned by a benchmarked system.

During a benchmarking experiment, the evaluation storage typically performs the following steps:

1. It is spawned by the benchmark controller with an identifier of an experiment (HOBBIT ID).
2. It initializes itself and connects to a `hobbit.command` queue.
3. It sends a ready signal to the benchmark controller (`hobbit.command`).
4. It reacts to all incoming queues and stores both types of results (expected and actual ones). It might add a timestamp to the results received from the system to enable time measurements.
5. After the evaluation module has been started, the evaluation storage will receive a request to iterate accumulated result pairs. Every incoming request will be answered with the next result pair.
6. If a request from the evaluation module is received but can not be answered because all result pairs have been sent, an empty response is sent.
7. If the signal to terminate is received from the benchmark controller it terminates with status code (0 in case of success).

We offer a default implementation of this component written in Java based on the Riak key-value store.

Evaluation Module

The evaluation module evaluates results generated by a benchmarked system. Depending on the goals of the benchmark this might be accomplished by using a comparison with the expected results or based on time measurements. During a benchmarking experiment, an evaluation module typically performs the following steps:

1. It is created by a benchmark controller with the HOBBIT ID of the experiment.
2. It initializes itself and connects to the `hobbit.command` queue.
3. It requests result pairs from the evaluation storage and evaluates (compares) them.
4. After the last pair has been received and evaluated, the evaluation results are summarized and sent to the benchmark controller.
5. It terminates with status code (0 in case of success).

\[^{7}\text{http://docs.basho.com/riak/latest/}\]
Benchmarked System Components

The **HOBBIT** platform does not have any requirements how a benchmarked system should be structured internally. However, the system has to implement a certain API. Depending on the implementation of the system and the benchmark that will be used, a system could implement the system API directly or use a system adapter that might be executed in an additional Docker container.

The API a system has to implement can be separated into two parts. One part is the communication with the **HOBBIT** platform while the other part is the communication with the benchmark.

The communication with the **HOBBIT** platform comprises the following three aspects.

1. With uploading a Docker image of the benchmarked system the user has to upload a metadata, which contains system’s name, input parameter model and other parameters of the system that could be used by the analysis component.
2. During the spawning of the systems Docker container the platform controller sets environment variables which have to be used by the system (e.g., the ID of the current benchmarking experiment).
3. The system has to connect to the `hobbit.command` queue and send a signal that indicates that it is ready for being benchmarked.

The second part of the API relies on the benchmark. As described above the data generators might send data while the task generators might send tasks. The responses generated by the system should be send to the evaluation storage. This communication might be carried out using RabbitMQ. However, as described before, the benchmark implementation might use a different communication technology which has to be used by the system.

Ontology

Experiments as well as created challenges are described using the **HOBBIT** ontology. The ontology offers classes and properties to describe:

- Benchmarks with their parameters, features and KPIs,
- Systems and their features,
- Experiments in which a single system is benchmarked with a certain benchmark,
- The results of an experiment,
- Challenges and their single challenge tasks as well as
- Additional information that is needed to rerun the same experiment again, e.g., the hardware on which the experiment has been carried out.

Table 4 lists the prefixes and their URIs that are used in this section. `exp` is the namespace for experiment instances while `error` is the namespace for error instances. The namespace of the **HOBBIT** Ontology is `http://w3id.org/hobbit/vocab#`. The Figures 3 and 4 give an overview over the classes of the ontology and their relations.

In the following sub sections, the single classes and properties of the ontology are described.
Table 4: Prefixes used in this section.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>hobbit</td>
<td><a href="http://w3id.org/hobbit/vocab#">http://w3id.org/hobbit/vocab#</a></td>
</tr>
<tr>
<td>error</td>
<td><a href="http://w3id.org/hobbit/error#">http://w3id.org/hobbit/error#</a></td>
</tr>
<tr>
<td>exp</td>
<td><a href="http://w3id.org/hobbit/experiments#">http://w3id.org/hobbit/experiments#</a></td>
</tr>
<tr>
<td>owl</td>
<td><a href="http://www.w3.org/2002/07/owl#">http://www.w3.org/2002/07/owl#</a></td>
</tr>
<tr>
<td>qb</td>
<td><a href="http://purl.org/linked-data/cube#">http://purl.org/linked-data/cube#</a></td>
</tr>
<tr>
<td>rdf</td>
<td><a href="http://www.w3.org/1999/02/22-rdf-syntax-ns#">http://www.w3.org/1999/02/22-rdf-syntax-ns#</a></td>
</tr>
<tr>
<td>rdfs</td>
<td><a href="http://www.w3.org/2000/01/rdf-schema#">http://www.w3.org/2000/01/rdf-schema#</a></td>
</tr>
</tbody>
</table>

Figure 3: Overview of the ontology including the benchmark, the system, their parameters and the experiment.

Classes

**hobbit:Benchmark**

Instances of this class define a particular benchmark. This includes parameters the benchmark has, KPIs it will create and an API that the benchmarked systems will have to implement in order to be benchmarked by it.
Figure 4: Overview of the ontology including the Challenge as well as the ChallengeTask classes and their relations.

**Datatype properties**

- **rdfs:label**: A label of the benchmark.
- **rdfs:comment**: A description of the benchmark.
- **hobbit:imageName**: A name of docker image of the benchmark controller.
- **hobbit:version**: A version of the benchmark.

**Object properties**

- **hobbit:measuresKPI**: the KPI and will return it as a result.
- **hobbit:hasAPI**: Connects the benchmark with an API of a particular benchmarked system.
- **hobbit:hasParameter**: Connects the benchmark to a number of parameters with which have to be configured.

**hobbit:Challenge**

Instances of this class define a challenge with its single hobbit:ChallengeTasks.

**Datatype properties**

- **rdfs:label**: A label of the challenge.
- **rdfs:comment**: A description of the challenge.
hobbit:executionDate  A date when an execution of experiments belonging to a particular challenge starts.

hobbit:publicationDate  A date when challenge results will be copied into the public graph, i.e., will become publicly available.

hobbit:organizer  The user identifier of the challenge organizer.

hobbit:closed  A flag that shows whether the challenge has been closed, i.e., no new participants will be able to register for it.

hobbit:visible  A flag that shows whether the challenge is visible to all users.

hobbit:ChallengeTask
A challenge task is a part of a challenge that involves a benchmark with a certain parameterization and a list of systems.

Datatype properties

rdfs:label  A label of the challenge task.

rdfs:comment  A description of the challenge task.

Additionally a challenge task might have properties that are instances of the hobbit:Parameter class, which defines a configuration of the benchmark for this challenge task.

Object properties

hobbit:involvesBenchmark  Connects the challenge task to the hobbit:Benchmark instance that will be used for the challenge task.

hobbit:involvesSystemInstance  Connects an experiment to one or more hobbit:SystemInstance entities that will be benchmarked during a challenge.

hobbit:isTaskOf  Defines a challenge, to which the task belongs to.

hobbit:rankingKPIs  Connects the hobbit:KPISeq, i.e. the ranking of the benchmark KPIs starting with the most important KPI to this particular challenge task.

hobbit:SystemInstance
Instances of this class are configured instances of a system that can be mapped to a docker image. They might have additional information about their configuration using the hobbit:hasParameter property.

Datatype properties
rdfs:label A label of the system.
rdfs:comment A description of the system instance.
hobbit:imageName The docker image name of the system.

Additionally, a system instance might have properties that are instances of the `hobbit:Parameter` class defining the configuration of this instance of the system.

**Object properties**

hobbit:implementsAPI Connects the instance of the system with an API it implements.
hobbit:instanceOf Connects the instance to the system of which it is a configured instance.

**hobbit:System**

Instances of this class might be used to express the connection between single system instances that represent different configurations of the same system. The definition of the system can comprise the definition of the parameters of this configuration. These parameter properties can be used to add the values of the parameters to the single `hobbit:SystemInstance` entities.

**Object properties**

hobbit:hasParameter Connects the system to one of its parameters.

**hobbit:Experiment**

An experiment is the benchmarking of a particular `hobbit:SystemInstance` at a particular time with a `hobbit:Benchmark` using a concrete configuration of the benchmark using the instances of the `hobbit:Parameter` properties of the benchmark. Additionally, the hardware on which the experiment has been carried out can be added.

**Datatype properties**

hobbit:startTime A timestamp of the start of the experiment.
hobbit:endTime A timestamp of the end of the experiment.
hobbit:hobbitPlatformVersion The version of the HOBBIT platform that has been used to carry out the experiment.

Additionally, an experiment might have properties that are instances of the `hobbit:Parameter` class defining the configuration of the benchmark for this experiment.

**Object properties**
hobbit:involvesBenchmark     Connects the experiment to a hobbit:Benchmark instance that has been used for the experiment.

hobbit:involvesSystemInstance     Connects the experiment to the hobbit:SystemInstance that has been benchmarked in this experiment.

hobbit:isPartOf     A challenge task this experiment is part of.

hobbit:terminatedWithError     If the experiment terminated with an error code, the error is linked to the experiment using this property.

hobbit:API

An API is a resource that is used to find systems that can be benchmarked by a given benchmark. It is used as a simple identifier and does not have additional properties. It might have a description (i.e., rdfs:comment) but it is not necessary.

hobbit:Error

An Error is a resource that is used to express the reason why an experiment did not terminate in a healthy state. There will be several instances for this class and each of them describes a reason why an experiment could have crashed. Each error should have a label and might have a description.

Datatype properties

rdfs:label     The label of the API.

rdfs:comment     A description of the API.

Instances

error:BenchmarkCrashed     The benchmark terminated with an error.

error:BenchmarkImageMissing     The benchmark image could not be loaded.

error:BenchmarkCreationError     The benchmark could not be created.

error:ClusterNotHealthy     The cluster was not in a healthy state when executing the experiment.

error:Experiment Took Too Much Time     The experiment took too much time.

error:SystemCrashed     The benchmarked system terminated with an error.

error:SystemImageMissing     The benchmarked system image could not be loaded.

error:SystemCreationError     The benchmarked system could not be created.

error:TerminatedByUser     The experiment has been terminated by the user.

error:UnexpectedError     An unexpected error occurred.
hobbit:Parameter

The parameter class is a subclass of rdf:Property. Instances of this class define a parameter property with a label, a description and a value type (using a range definition). The properties can have either a hobbit:SystemInstance or a hobbit:Experiment as domain. An instance of a parameter should define a range. This can be either a literal datatype or a resource class, e.g., defining a set of predefined values of which the user has to choose one.

**Datatype properties**

- **rdfs:label** The label of the parameter.
- **rdfs:comment** A description of the parameter.

**Object properties**

- **rdfs:range** A literal type or a resource class.

hobbit:ConfigurableParameter

This subclass of the hobbit:Parameter and qb:DimensionProperty classes is used to define benchmark parameters that should be configured by the user. An instance of hobbit:Experiment class involving a benchmark that has defined these parameters should use the instances to define the values of the parameters. An instance of a hobbit:ConfigurableParameter can define a default value. This can be helpful for inexperienced users to choose a good parameter value.

**Datatype properties**

- **hobbit:defaultValue** An (optional) default value for this parameter.

hobbit:FeatureParameter

The parameter class is a subclass of hobbit:Parameter. Instances of this class define a parameter that can be used as features of a system or a benchmark during later analysis. Note that in contrast to a hobbit:ConfigurableParameter instance the feature parameter can be created at runtime. This is helpful if a feature of a randomly generated dataset can not be determined before the experiment has been started.

hobbit:ForwardedParameter

The parameter class is a subclass of hobbit:Parameter. Instances of this class define a parameter that is defined for a benchmark but are also forwarded to the system when it is created. That allows benchmark developers to define to define parameters as part of their benchmark API which for example can be configured by the user and are given to the system.
hobbit:KPI

The KPI class (Key Performance Indicator) is a sub class of rdfs:Property. Instances of this class define a measure property with a label, a description and a value range. The domain of properties of this class is the hobbit:Experiment class.

**Datatype properties**

- rdfs:label: The label of the KPI.
- rdfs:comment: A description of the KPI.

**Object properties**

- rdfs:range: The literal type of the KPI.
- hobbit:ranking: The ranking of this KPI. At the moment, the platform implementation supports the two different instances hobbit:AscendingOrder and hobbit:DescendingOrder.

hobbit:KPISeq

The KPISeq class is a sub class of rdf:Seq. It is used as an ordered list of KPIs. The order expresses the importance of the KPIs for the ranking of experiment results, i.e., the experiment results are sorted using their value for the first KPI from that list. If two experiment results are equal for the first KPI, the value of the second KPI is taken into account. This ordering is used to generate the leaderboards of the challenge tasks.

**Object properties**

- rdf:__n: The KPI at the $n$-th position (where $n > 0$ following the definition of rdf:Seq).

**Properties**

In this section, the properties are listed and briefly explained.

- hobbit:closed: A flag that shows whether a hobbit:Challenge has been closed, i.e., no new participants will be able to register for it.
- hobbit:endTime: The timestamp of the end of a hobbit:Experiment.
- hobbit:hasAPI: Connects the hobbit:benchmark with the API the benchmarked system will have to implement to be benchmarked.
hobbit:hasParameter
Connects the hobbit:Benchmark to the hobbit:Parameter with which the benchmark can be configured.

hobbit:hobbitPlatformVersion
The version of the HOBBIT platform that has been used to carry out the experiment.

hobbit:imageName
The docker image name of a hobbit:Benchmark or a hobbit:SystemInstance.

hobbit:implementsAPI
Connects the hobbit:SystemInstance with the API it implements.

hobbit:instanceOf
Connects the hobbit:SystemInstance to the hobbit:System of which it is a configured instance.

hobbit:involvesBenchmark
Connects the hobbit:experiment or hobbit:ChallengeTask to the hobbit:Benchmark instance that has been used for the experiment.

hobbit:involvesSystemInstance
Connects the hobbit:experiment or hobbit:ChallengeTask to the hobbit:SystemInstance that has been benchmarked in this experiment.

hobbit:isPartOf
Connects the hobbit:experiment to a hobbit:ChallengeTask

hobbit:isTaskOf
Connects a hobbit:ChallengeTask to a hobbit:Challenge.

hobbit:measuresKPI
The hobbit:benchmark measures the KPI and will return it as result.

hobbit:ranking
The ranking of a hobbit:KPI. At the moment, the platform implementation supports the two different instances hobbit:AscendingOrder and hobbit:DescendingOrder.

hobbit:rankingKPIs
Connects a hobbit:KPISeq expressing a ranking of single KPIs to a certain challenge task.

hobbit:startTime
The timestamp of the start of the hobbit:Experiment.

hobbit:terminatedWithError
Connects the hobbit:Experiment to the description of the error occurred.

hobbit:version
The version of the hobbit:Benchmark.

hobbit:visible
A flag that shows whether a hobbit:Challenge is visible to all users.

**User Manuals**

In this section, the interaction of a user with the platform is described based on the user roles described in Section 2.2.3.
Register

Apart from exploring the public results of experiments, a guest has no further possibilities to interact with the platform. However, a user can get access to further functionalities by registering herself. A registered user can upload systems for benchmarking, register for challenges and see results published for the user. A special role is the challenge organizer. To become a challenge organizer a user has to contact the host of the HOBBIT benchmark page.

When accessing the HOBBIT benchmark page the user is forwarded to the HOBBIT login and registration page (cf. Figure 5). If a user has a login already he can login with her credentials. If the user intends to use the HOBBIT benchmarking system only as guest the user can press the Log in as Guest button and will be redirected to the HOBBIT benchmark page as Guest with restricted access rights.

![Login page](image.png)

Figure 5: Login page.

When registering the registration page (cf. Figure 6) asks certain information to be filled in. If a user already registered with the entered name or email address a warning message will show up e.g.: Email already registered. In this case a different name or email address is required for registration. With a successful registration the user is logged into the platform and will be redirected to the HOBBIT benchmark page. To become Challenge Organizer and add an manage challenges the user needs to contact the platform host or administrator which can change the user’s role from ordinary registered user to Challenge Organizer.

One can logout from the HOBBIT benchmark system using the logout option.
Upload a system

A registered user has access to the platform and the repository, i.e., the Gitlab instance of the platform. For benchmarking a system, the user has to upload a file containing the system’s metadata and the Docker image of the system.

Within the HOBBIT benchmarking platform one can access the pages explaining the uploading process via the **Upload** menu (cf. Figure 7).

Uploading the Docker image

The Docker container image for the system needs to be available to the HOBBIT platform. Thus, it needs to be uploaded to a Docker repository which can be accessed by the HOBBIT platform. If the image is not uploaded to an official repository, the Gitlab instance can be used to upload the images.

Let’s assume we have a system called "MySystem" and a HOBBIT user profile with the name MaxPower. We already created a git project for our system in the HOBBIT Gitlab (with the same name as our system). We build the image with the command

```
$ docker build -t git.project-hobbit.eu:4567/maxpower/mysystem .
```

It can be seen, that the address of the HOBBIT Gitlab is part of the image name as well as the name.
Pushing multiple images

It is possible to upload multiple images. This is needed for uploading a benchmark or in cases in which the a benchmarked system comprises more than a single Docker container. Although Gitlab follows a "one image per project" philosophy it is possible to upload multiple images into a single git project. Gitlab supports this with up to one additional hierarchy, e.g., it can distinguish between the two images

```
1 docker login git.project-hobbit.eu:4567
2 docker push git.project-hobbit.eu:4567/maxpower/mysystem
```

System metadata file

The system metadata file comprises meta data about the uploaded system that is needed by the HOBBIT platform. The file contains the meta data as RDF triples in the Turtle format. The simplest file defines only the necessary information of a single `hobbit:SystemInstance`. First, the user has to collect the following data:

- The system needs a unique identifier, i.e., a URI. In our example, we choose `http://www.example.org/exampleSystem/MySystem`.
- The system needs a name ("MySystem") and a short description ("This is my own example system...").

8https://www.w3.org/TR/turtle/
• The name of the uploaded docker image is needed ("git.project-hobbit.eu:4567/maxpower/mysystem").

• The URI of the benchmark API, the system implements (http://benchmark.org/MyNewBenchmark/BenchmarkApi, should be provided by the benchmark description page).

The example meta data file comprising the necessary information has the following content.

```rdfs namespace: <http://www.w3.org/2000/01/rdf-schema#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix hobbit: <http://w3id.org/hobbit/vocab#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://www.example.org/exampleSystem/MySystem> a hobbit: SystemInstance ;
  rdfs:label "MySystem"@en ;
  rdfs:comment "This is my own system defined in a simple way"@en ;
  hobbit:imageName "git.project-hobbit.eu:4567/maxpower/mysystem" ;

Some systems offer one or more parameters which can be used to adapt the system for certain scenarios. The Hobbit platform can analyse the influence of a parameter on the systems performance using its analysis component. To enable this analysis, it is necessary to define the parameters as well as the single parameterizations of a system in the meta data file. To this end, a hobbit:System has to be defined that is connected to hobbit:FeatureParameter instances. The single parameterised versions of the system are defined as hobbit:SystemInstance objects with a single value for each parameter.

```
In the example, "MySystem" is defined as a `hobbit:System` with a threshold parameter attached to it using the `hobbit:hasParameter` property. The threshold parameter is an instance of `hobbit:FeatureParameter`, has a label as well as a description and defines the range of its values as floating point number. The two instances "MySystem (0.6)" and "MySystem (0.7)" are connected to the system definition using the `hobbit:instanceOf` property and define a certain value for the threshold parameter. This example results in two system instances listed as systems for benchmarking in the platforms front end and the knowledge about the threshold as a feature that can be used in the analysis component.

To ease the usage of parameters, the meta data of a system instance is given to its Docker container when it is started, i.e., the user does not have to define a single Docker image for every parameterization but can read the parameters from an environmental variable at runtime.

**Upload a benchmark**

Uploading a benchmark works similar to uploading a system. The user needs to upload the Docker image(s) necessary for executing the benchmark as well as defining the metadata of the benchmark in a benchmark.ttl file. The uploading of Docker images for a benchmark works exactly as the uploading of a system image explained in Section 3.2.1.

Every benchmark uploaded to the platform has to be described with a benchmark metadata file comprising information needed by the Hobbit platform. The file contains the metadata as RDF triples in the Turtle format. It needs to be uploaded to the git instance of the platform, e.g., into the root directory of one of the projects that host one of the benchmark components, e.g., the benchmark controller. For creating a metadata file the user has to collect the following data:

- The benchmark needs a unique identifier, i.e., a URI. In our example, we choose `http://www.example.org/exampleBenchmark/MyOwnBenchmark`.
- The benchmark needs a name ("MyBenchmark") and a short description ("This is my own example benchmark...").
- The name of the uploaded benchmark controller Docker image ("git.project-hobbit.eu:4567/maxpower/mybenchmarkcontroller").
- The URI of the benchmark API.
- The version number of our benchmark implementation.
- The parameters of the benchmark, their ranges and their default values.
- The KPIs measured by the benchmark and their ranges.
The basic description of the example benchmark has the following triples.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix hobbit: <http://w3id.org/hobbit/vocab#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://www.example.org/exampleBenchmark/MyOwnBenchmark> a hobbit: Benchmark ;
rdfs: label "MyBenchmark" @en ;
hobbit: imageName "git.project-hobbit.eu:4567/maxpower/mybenchmarkcontroller" ;
hobbit: usesImage "git.project-hobbit.eu:4567/maxpower/mydatagenerator" ;
hobbit: usesImage "git.project-hobbit.eu:4567/maxpower/mytaskgenerator" ;
hobbit: version "v1.1" @en ;
hobbit: measuresKPI <http://www.example.org/exampleBenchmark/precision> ;
hobbit: measuresKPI <http://www.example.org/exampleBenchmark/recall> ;
hobbit: measuresKPI <http://www.example.org/exampleBenchmark/fmeasure> ;
hobbit: hasApi <http://www.example.org/exampleBenchmark/API> ;
hobbit: hasParameter <http://www.example.org/exampleBenchmark/size> ;
hobbit: hasParameter <http://www.example.org/exampleBenchmark/queryScenario> ;
hobbit: hasParameter <http://www.example.org/exampleBenchmark/amountOfNullExamples> .
```

It can be seen that apart from the label, the description, a version and the image name the example benchmark has an API, several KPIs and parameters assigned to it which will be explained in the following. Additionally, we listed to other images using the `hobbit:usesImage` property to enable the platform to prefetch these images.

Inside the meta data files, the API is a URI that works as a simple identifier to be able to map systems to benchmarks. A benchmark can only be executed together with a system if both share the same API.

```
<http://www.example.org/exampleBenchmark/API> a hobbit: API .
```

A Key Performance Indicator (KPI) is a value that shows the performance of the benchmarked system. It should have a label, a description and a range. The example benchmark has the three KPIs precision, recall and F-measure.

```
<http://www.example.org/exampleBenchmark/precision> a hobbit: KPI ;
rdfs: label "Precision" @en ;
rdfs: comment "Precision = TP / (TP + FP)" @en ;
rdfs: range xsd: float .

<http://www.example.org/exampleBenchmark/recall> a hobbit: KPI ;
rdfs: label "Recall" @en ;
rdfs: comment "Recall = TP / (TP + FN)" @en ;
rdfs: range xsd: float .

<http://www.example.org/exampleBenchmark/fmeasure> a hobbit: KPI ;
rdfs: label "F-measure" @en ;
rdfs: comment "F-measure is the harmonic mean of precision and recall" .
```
There are two (overlapping) groups of parameters. Configurable parameters can be configured by the user when starting a benchmark. Feature parameters are used for deeper analysis by the analysis component. Every parameter should have a label, a description and a value range. Configurable parameters should have a default value that can be used by non-expert users to get the benchmark running.

Apart from parameters with simple values, HOBBIT supports parameters that you have to choose from a given set of values. To use this feature, the parameter needs to have a class as `rdfs:range` and the single choosable values need to be instances of that class.

In the example, the benchmark offers different graph patterns (star, grid and clique) from which the user will have to choose one.

In some cases, the benchmark might define a parameter that should not be set by the user but is still interesting for later analysis, e.g., if the benchmark is based on data that is generated on the fly. This can be achieved by defining a parameter only as `hobbit:FeatureParameter` and not as configurable.
In the example, the benchmark would generate a graph for benchmarking a system. After the generation and the evaluation of the systems results, the benchmark could add an amount of blank nodes that have been created to the result. Note that this is not a KPI since it is not bound to the performance of the system.

**Benchmark a system**

Being a registered user of the platform and having uploaded a system which conforms the specification (API) of one of the benchmarks allows the user to benchmark the system. The benchmarking of a system is done via the *Benchmarks* menu (see Figure 8) where at first the benchmark is selected to be used for the experiment. The drop down menu displays all possible benchmarks.

![Figure 8: Configuration of a benchmarking experiment. (a) Select the benchmark.](image)

Having selected the benchmark, the system to be benchmarked is selected. Only systems uploaded by the user and fitting the API of the chosen benchmark are displayed. Then the benchmark experiment is configured by setting the benchmark specific parameters (see Figure 9). These might vary amongst the different benchmarks due to their different nature. Parameters can be e.g. numeric values, string values, dates or even nominal with pre-defined values that can be selected by a drop-down box. Some of the values might also have restrictions.

When the experiment is configured it can be submitted via the *Submit* button. This button is inactive as long as the configuration is not completed. After successful submission a page with the submission details is presented (see Figure 10) to the user.
Figure 9: Configuration of a benchmarking experiment. (b) Select the system and configure experiment.
Figure 10: Configuration of a benchmarking experiment. (c) Submission details.
Experiment status

Any user can see the current status of the platform and the configured experiments, e.g., for checking whether the own experiment is already being executed. The status page can be reached by clicking on Experiments and Experiment status.

![Status page of the HOBBIT platform.](image)

Figure 11: Status page of the HOBBIT platform.
The page shows the experiment that is executed at the moment. This includes general information as well as the phase the experiment is in (as described in Section 2.1) and the maximum run time that this experiment will take. If the user started this experiment, a Delete button is displayed which can be used to stop the experiment.

Below the executed experiment, the queue of waiting experiments is shown. Experiments configured by the user can be removed from the list using the Delete button.

**Explore experiment results**

Any user can explore published results of experiments. Additionally, registered users can explore unpublished results for their systems, e.g., results in a challenge that has not published its complete result list. Challenge owners are allowed to see the results of all experiments that belong to their challenge allowing them to see and announce the best performing system even if the results haven’t been published.

In order to see the results the user selects the Experiment Results page of the Experiments menu item (see Figure 12).

![Figure 12: Navigation to result page.](image)

The results page displays the details of finished experiments (see Figure 13) the user is allowed to explore.

One can select various experiments in order to compare the results of these experiments (see Figure 14) or only select one experiment to explore its result. The results are displayed after pressing the Show Details button. The list of experiments can be filtered to search for certain systems, benchmarks or challenge tasks. To see all results of a specific challenge task it is also possible to simply select the task at the bottom of the page.

The results are displayed next to each other (see Figures 15 and 16). The result table comprises five parts: 1) General information about the experiment, 2) the measured KPIs, 3) experiment logs, 4) benchmark parameters and 5) the visualisation of result sets as diagrams. If the experiment caused and logged an error, it is displayed as well. The error will be highlighted.

**Register for a challenge**

System providers can register for challenge tasks in challenges. To register for a challenge task the user needs to be logged in into the HOBBIT benchmarking page (see Section 3.1). If the user is logged in as challenge organizer the page might vary (see Section 3.8). The registration of a system for a challenge task is done via the Challenges menu item (see Figure 17). The challenge page displays the
challenges accessible by the user. The first table contains challenges which are currently running while the second table contains challenges that already ended. Both tables allow the filtering and sorting of their content.

To register a system the user selects the challenge and is then redirected to the challenge page (see Figure 18). The page displays various details of the challenge. The bottom of the page displays the tasks of the challenge. To register a system press the **Register System** button at the **Tasks** section of the page.

A new page will display tasks of the challenge and the systems of the user matching the task
requirements (see Figure 19). Only the user’s systems are displayed. The user can select the systems for the challenge. When finished the user is required to press the Done button on top of the page to finalise the registration.

To see the challenge task details like the configuration of the benchmark performed by the challenge task, the user can select the challenge task in the challenge page 18). This will redirect the user to the challenge task details page (see Figure 20).

For every challenge task, the challenge offers a leaderboard. Those leaderboards can be listed by the user using the leaderboard button of the challenge (see Figure 18). On the leaderboard screen, the user can select the single tasks and see the systems which participated in the challenge sorted by their performance (see Figure 21). By clicking on the systems, the details of the experiments can be viewed. This view is very helpful for repeatable challenges to see the best performing systems that took part in the challenge so far.

Create a challenge

A challenge organizer can create challenges. In order to do so the challenge organizer needs to be logged in (see Section 3.1). When selecting the Challenges menu item the challenge organizer, other than a conventional user, additionally sees the Add Challenge button (see Figure 22).

To add a challenge the user presses the Add Challenge button and is re-directed to the page providing a form to add information on the challenge (see Figure 23). The challenge organizer is required to enter a challenge name, some description, an execution date and some publishing date. While the first date marks the point in time at which the platform will start to execute the experiments of the challenge, the second date is the day from which on the results will be published, i.e., they are made visible to everybody. Additionally, a challenge homepage can be entered which will be rendered as link for the other users. Additionally the challenge organizer can indicate if a challenge is Visible for everyone. Not setting
When all parameters are filled out, the challenge organizer is required to save the challenge (press the Save button) before adding tasks to the challenge.

This will redirect the challenge organizer to the challenge list page. By clicking on the newly created challenge in the list the challenge organizer can change the challenge details and add tasks (see Figure 24). In order to add tasks the challenge organizer presses the Add Task button.

The challenge organizer is then redirected to the challenge task detail page (see Figure 25). Here the challenge organizer can name the task and give some more details via the description. Additionally the organizer is required to select and configure a benchmark that implements the described task.

Figure 15: Upper part of result comparison (tables have been shortened for this picture).
After having added the challenge tasks to the challenge the challenge needs to be saved again. In order to allow people to register it needs to be Visible for everyone by setting this flag before saving the challenge.

The Close Challenge button closes a challenge, i.e., no more registrations are possible and the challenge is waiting for its execution date to arrive for running the experiments. It should only be pressed when the registration for the challenge is over. After closing the evaluation of the registered systems starts as soon as the execution date is reached. When pressing the button a dialogue comes up asking for closing the challenge (see Figure 26) to ensure this does not happen accidentally.
Figure 17: Lists of challenges.
Figure 18: HOBBIT challenge detail page.
Figure 19: Challenge task registration page.
Figure 20: Challenge task details page.
Figure 21: Challenge leaderboards page.

Figure 22: Challenge page header for a user with the challenge organizer role.
Figure 23: New challenge form page.
Figure 24: Filled challenge form without tasks.

Figure 25: Edit challenge task details form page.
Figure 26: Dialog for closing a challenge.
Java SDK

In order to make the development of HOBBIT-compatible applications easier the software library called "HOBBIT Java SDK" was developed.

The library allows benchmark designers and systems developers to execute HOBBIT-compatible components (benchmarks and benchmarked systems) locally without having a running instance of the platform. The SDK provides an ability to emulate a running instance of the HOBBIT platform for debugging purposes and prevent developers from any deployment and configuration efforts. When the debugging process is finished developers are free to upload tested Docker images to the running instance of the platform (e.g. the online platform) and run them without any changes.

The SDK provides developers with two abilities to test their components: execute as pure java codes and execute being packed into Docker containers. Execution as pure java codes makes possible to debug an internal logic of components in any integrated development environment (IDE) by putting breakpoints and using an integrated debugger. Execution as Docker containers makes possible to test whether (a) Docker images for the components are building correctly, (b) the components correctly spawning, executing and finishing inside Docker containers. All the logs, produced by the components inside Docker containers are captured by the SDK and provided into console output in real time, so it makes the process of testing less time consuming and more efficient.

Architecture

Key components of the SDK and their communication with other HOBBIT-related components are presented at Fig. 27. The purpose of the components are described below.

![Architecture of the SDK and interaction with HOBBIT components.](image)

**Figure 27:** Architecture of the SDK and interaction with HOBBIT components.

**RabbitMQ Dockerizer**

The component spawns the Docker image with RabbitMQ server in order to provide the queuing service for other HOBBIT components during local debugging process. The required Docker image will
downloaded automatically.

Command Queue Listener

The component attaches to the main command queue and responds the incoming requests during the runtime. It submits the required components (e.g. data generators, task generators, evaluation storage and evaluation module) associated with image names, according to image name of the particular incoming request.

Components Executor

The component executes the submitted components in parallel fashion. RabbitMQ Dockerizer, Benchmark controller and Benchmarked system are submitted to the component at the beginning of test execution, while all other components (data generators, task generators, etc.) will be submitted by the Command Queue Listener during a runtime.

Class diagram

![Class diagram](image_url)

Figure 28: Class diagram of key classes of the Java SDK.

Key classes of the SDK which used to describe dockerized components are presented at Fig. 28. The descriptions of the classes are presented below.
Dockerizer

Dockerizer is an abstract class for serving Docker containers, which implements the Component interface provided by the Hobbit Core library. Dockerizer provides functionality for Docker container management (create, run, read logs, stop, remove, etc...) except preparation of Docker images, which is can be different (e.g. build, pull, etc.) and supposed to be implemented by other extending classes.

Pull-based dockerizer

Pull-based dockerizer is an implementation of the Dockerizer class for serving Docker containers for which Docker images have to be downloaded (pulled) from third-party repositories, specified in the URL of particular image. RabbitMQ dockerizer is an example of such type of Dockerizer.

Build-based dockerizer

Build-based dockerizer is an implementation of the Dockerizer class for serving Docker containers for which Docker which images have to be built from Docker file. Static or dynamically generated docker files can be used for this type of Dockerizer.

Dockerizer Builder

DockerizerBuilder (DB) is a basic class used for building Dockerizer instances, which is used as a basis of components-specific builders. Developers are free to choose a type of Dockerizer (build-based or pull-based) for any component, so it makes possible to run benchmark components without source codes (i.e. using images from third-party repositories).

Component-specific DBs

Component-specific dockerizer builders (e.g. Benchmark DB, Data Generator DB, Task Generator DB, EvalStorage DB, Eval System DB) define environment variables unique for each particular type of component. The defined parameters will be submitted to a docker container, when it will be launched during local execution.

Development using the SDK

Before start of development using the SDK a developer should make sure that Oracle Java 1.8 or higher as well as Docker v.17 or higher are installed. Also the line 127.0.0.1 rabbit should be added to a hosts file of development OS. The SDK library should be installed to a local maven repository by executing the mvn validate command just after the clone of starter kit repository, described below.

In order to start development using the SDK a developer should clone the SDK starting kit https://github.com/hobbit-project/java-sdk-example. The distribution (see Fig. 29) includes basic implementations of all types of Hobbit-compatible components (e.g. benchmark controller, data generator, system adapter, etc.) and unit tests build around that components. Basic implementations of components are supposed to be extended by logic of a particular benchmark and checked/debugged by the unit tests included. The unit tests are intended to be the main working tool for developers to test/debug the developing component and supposed to be executed manually from any IDE.
Before running tests a set of constants components’ image names and other project-related values (see Fig. 30) should be specified in file Constants.java. Part of these constants (image names) will be associated with the particular executable components (see Fig. 34), which will be submitted to the components executor, when the particular image name will be requested to be spawned.

Test benchmark components as pure java codes

In order to run components as pure java codes a developer can run the `checkHealth()` method from the `ExampleBenchmarkTest.java` (see 31).

The standard testing pipeline includes:
1. Initialization of RabbitMQ dockerizer,

2. Specification of environment variables including parameter models for the benchmark and the benchmarked system (see Fig. 32),

3. Initialization of components to be submitted (see Fig. 33),

4. Initialization of SDK components, submission of components and association with image names (see Fig. 34).

Test dockerized benchmark components

In order to run components as docker containers the checkHealthDockerized() method from the ExampleBenchmarkTest.java should be executed (see Fig. 35).

The Docker containers will be executed and images preparation (pulling/building) will be performed automatically, if some images do not exist. Developers are free build/rebuild images on demand by executing the buildImages() method from the ExampleBenchmarkTest.java (see Fig. 36).

In order to build Docker images from the components a jar package should be created under the target folder. The package may be created by the mvn package -DskipTests=true command. The package and images should be rebuilt each time when any changes in code have been done. The checkHealthDockerized() shares the same pipeline as checkHealth(), but submitted components are reinitialized with Dockerizers (see Fig. 37).

```java
public class Constants {
    public static String GIT_REPO_PATH = "git.project-hobbit.eu:4567/";
    public static String PROJECT_NAME = "sdk-examples/"
    public static final String BENCHMARK_IMAGE_NAME = GIT_REPO_PATH+PROJECT_NAME + "benchmark-controller";
    public static final String TASKGEN_IMAGE_NAME = GIT_REPO_PATH+PROJECT_NAME + "taskgen";
    public static final String EVAL_STORAGE_IMAGE_NAME = GIT_REPO_PATH+PROJECT_NAME + "eval-storage";
    public static final String EVALMODULE_IMAGE_NAME = GIT_REPO_PATH+PROJECT_NAME + "eval-module";
    public static final String SYSTEM_IMAGE_NAME = GIT_REPO_PATH+PROJECT_NAME + "system";

    public static final String BENCHMARK_URI = "http://project-hobbit.eu"+PROJECT_NAME;
    public static final String SYSTEM_URI = "http://project-hobbit.eu"+PROJECT_NAME;

    public static final String SDK_BUILD_DIR_PATH = ".";
    public static final String SDK_JAR_FILE_PATH = "target/sdk-example-benchmark-1.0.jar";

    @Test
    public void checkHealth() throws Exception {
        checkHealth(dockerized: false);
    }
}
```

Figure 30: Project-related constants

Figure 31: Screenshot of the check health method
Dockerizers will be created from component-specific builders, which use build-based dockers builder based on project-related constants (see Fig. 30). For any component-specific Docker builder the developer is free specify (a) avoidance of image rebuilding at every run – by calling the `useCachedImage()` method, (b) file or folder with data which should be placed into a particular image – by executing the `addFileOrFolder()` method (c) a custom dockerFile for a particular docker image – by executing the `customDockerFileReader()` method (see ExampleDockersBuilder at Fig. 38).

Test a benchmarked system under third-party benchmark

The SDK allows system developers (challenge participants) to debug their systems under third-party benchmark’s workload. The example of the test can be found in the `ExampleSystemTest.java` file, which is also included into the starting kit. Pull-based dockerizers are used to execute the benchmark components, which are supposed to be downloaded (pulled) from the URLs provided by benchmark owner. Owner of the benchmark have to make the images publicly accessible and provide the URLs of benchmark images to systems developers. System developers are able to test their systems in the same manner, as benchmark developers (i.e. pure and dockerized mode) as described above.
```java
commandQueueListener = new CommandQueueListener();
componentsExecutor = new ComponentsExecutor(commandQueueListener, environmentVariables);
rabbitMqDockerizer.run();

commandQueueListener.setCommandReactions(
    new MultipleCommandReactions(componentsExecutor, commandQueueListener)
        .dataGenerator(dataGen).dataGeneratorImageName(DATAGENERATOR_IMAGE_NAME)
        .taskGenerator(taskGen).taskGeneratorImageName(TASKGEN_IMAGE_NAME)
        .evalStorage(evalStorage).evalStorageImageName(EVAL_STORAGE_IMAGE_NAME)
        .evalModule(evalModule).evalModuleImageName(EVALMODULE_IMAGE_NAME)
        .systemContainerId(SYSTEM_IMAGE_NAME)
);

componentsExecutor.submit(commandQueueListener);
commandQueueListener.waitForInitialization();

componentsExecutor.submit(benchmarkController);
componentsExecutor.submit(systemAdapter, SYSTEM_IMAGE_NAME);
```

Figure 34: Screenshot of components submission

```java
@Test
public void checkHealthDockerized() throws Exception {
    checkHealth(dockerized: true);
}
```

Figure 35: Check health dockerized test.

```java
@Ignore
public void buildImages() throws Exception {
    init(useCachedImage: false);
    benchmarkBuilder.build().prepareImage();
    dataGeneratorBuilder.build().prepareImage();
    taskGeneratorBuilder.build().prepareImage();
    evalStorageBuilder.build().prepareImage();
    evalModuleBuilder.build().prepareImage();
    systemAdapterBuilder.build().prepareImage();
}
```

Figure 36: Build images test

```java
benchmarkController = benchmarkBuilder.build();
dataGen = dataGeneratorBuilder.build();
taskGen = taskGeneratorBuilder.build();
evalStorage = evalStorageBuilder.build();
evalModule = evalModuleBuilder.build();
systemAdapter = systemAdapterBuilder.build();
```

Figure 37: Dockerized components initialization
benchmarkBuilder = new BenchmarkDockerBuilder(
    new ExampleDockerBuilder(BenchmarkController.class, BENCHMARK_IMAGE_NAME)
        .useCachedImage(useCachedImage));

dataGeneratorBuilder = new DataGenDockerBuilder(
    new ExampleDockerBuilder(DataGenerator.class, DATAGEN_IMAGE_NAME)
        .useCachedImage(useCachedImage)
        .addFileOrFolder("data"));

taskGeneratorBuilder = new TaskGenDockerBuilder(
    new ExampleDockerBuilder(TaskGenerator.class, TASKGEN_IMAGE_NAME)
        .useCachedImage(useCachedImage)
        .customDockerFileReader(new FileReader(fileName: "path"));

evalStorageBuilder = new EvalStorageDockerBuilder(
    new ExampleDockerBuilder(EvalStorage.class, EVAL_STORAGE_IMAGE_NAME)
        .useCachedImage(useCachedImage));

systemAdapterBuilder = new SystemAdapterDockerBuilder(
    new ExampleDockerBuilder(SystemAdapter.class, SYSTEM_IMAGE_NAME)
        .useCachedImage(useCachedImage));

evalModuleBuilder = new EvalModuleDockerBuilder(
    new ExampleDockerBuilder(EvalModule.class, EVALMODULE_IMAGE_NAME)
        .useCachedImage(useCachedImage));

Figure 38: Example of component-specific dockerizer builders
Table 5: Results of the platform benchmark on a single machine (1 – 3) and on a server cluster (4, 5).

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Exp. 1</th>
<th>Exp. 2</th>
<th>Exp. 3</th>
<th>Exp. 4</th>
<th>Exp. 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data generators</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Task generators</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Queries</td>
<td>1,000</td>
<td>2,000</td>
<td>5,000</td>
<td>100,000</td>
<td>300,000</td>
</tr>
<tr>
<td>Avg. query runtime</td>
<td>7,058</td>
<td>17,309</td>
<td>33,561</td>
<td>38,810</td>
<td>59,828</td>
</tr>
<tr>
<td>Query runtime std. dev.</td>
<td>686</td>
<td>4,493</td>
<td>3,636</td>
<td>22,517</td>
<td>24,540</td>
</tr>
<tr>
<td>Overall runtime</td>
<td>11.2</td>
<td>32.4</td>
<td>51.5</td>
<td>2,086</td>
<td>2,536</td>
</tr>
<tr>
<td>Queries per second</td>
<td>44.9</td>
<td>31.0</td>
<td>48.6</td>
<td>865.1</td>
<td>774.2</td>
</tr>
</tbody>
</table>

Evaluation

The HOBBIT platform has already been used successfully in a large number of challenges. Still, we evaluated our architecture in two different respects. First, we simulated benchmarking triple stores using HOBBIT. These experiments had two goals. First, we wanted to prove that the HOBBIT platform can be used on single, lightweight hardware (e.g., for development purposes or for benchmarks where the scalability and runtime are not of importance) as well as in a distributed environment. Second, we wanted to evaluate the throughput of storage benchmarks. In addition, we benchmarked several knowledge extraction tools and studied the runtime performance of these systems for the first time.

Triple store benchmark use case

To configure our simulation, we derived message characteristics from real data using the Linked SPARQL Queries Dataset [7]–a collection of SPARQL query logs. This collection of real query logs suggests that (1) the average length of a SPARQL query has a length of 545.45 characters and (2) the average result set comprises 122.45 bindings. We assumed that the average size of a single result is 100 characters leading to a result set size of approximately 12,200 characters.

The platform was deployed on a small machine\(^9\) and on a server cluster.\(^10\) The single benchmark runs are shown in Table 5. We executed the benchmark with three different numbers of queries on the smaller machine and two larger numbers of queries on the cluster. Our results show that the platform can run even on the minimalistic single machine chosen for our evaluation. Hence, the HOBBIT platform can be used locally for smoke tests and development tests. In addition, our results also clearly indicate the need for a platform such as HOBBIT by pointing to the necessity to deploy benchmarking platforms in a large-scale environment to test some of the BLD systems. Experiments with 5000 queries run on the small machine clearly show an increase in the average runtime per query and the standard deviation of the query runtimes due to a traffic jam in the message bus queues. In contrast, our results on the

\(^9\)Dual Intel Core i5 , 2.5 GHz, 2 GB RAM
\(^10\)1 master server (1xE5-2630v4 10-cores, 2.2GHz, 128GB RAM) hosting platform components (including RabbitMQ message broker), 1 data server (1xE5-2630v3 8-cores, 2.4GHz, 64GB RAM) hosting storages, 6 nodes (2xE5-2630v3 8-cores, 2.4GHz, 256GB RAM) divided into two groups hosting either components of the benchmark or the benchmarked system.
Knowledge Extraction benchmark use case

For our second evaluation, we used Task 1B of the Open Knowledge Extraction challenge 2017 [9] as use case. This task comprises the problem of spotting named entities from a given text and linking them to a given knowledge base. All experiments were run on our cluster. We benchmarked the following named entity recognition tools: (1) FOX [8], (2) the Ottawa Baseline Information Extraction (Balie) [5], (3) the Illinois Named Entity Tagger (Illinois) [6], (5) the Apache OpenNLP Name Finder (OpenNLP) [1], (5) the Stanford Named Entity Recognizer (Stanford) [2], and (6) DBpedia Spotlight (Spotlight) [3]. The entities that were found in the text by any of the tools are linked to a given knowledge base using AGDISTIS [4]. In our experiment, we used DBpedia 2015\(^{11}\) as the reference knowledge base.

The aim of the benchmark was to measure the scalability and the accuracy of these systems under increasing load, an experiment which was not possible with existing benchmarking solutions. We used a gold standard made up of 10,000 documents generated using the BENGAL generator\(^{12}\) included in the HOBBIT platform. The evaluation module was based on the evaluation used in [9] and measured the run time for single documents as well as the result quality in terms of micro-precision, recall and F1-measure. We used 1 data and 1 task generator for our benchmark. The data generator was configured to run through 5 velocity phases (2000 documents/phase) with differing delays between single documents in each phase. The delays between the documents were set to \{1s, \(\frac{1}{2}\)s, \(\frac{1}{4}\)s, \(\frac{1}{8}\)s, 0s\} leading to an increasing workload of \{1, 2, 4, 8, \(\approx\)800\} documents per second.

The results in Table 6 and Figure 39 show that all approaches scale well when provided with enough hardware. As expected, FOX is the slowest solution as it relies on calling 5 underlying fully-fledged

---

11\text{http://dbpedia.org}
12\text{http://github.com/dice-group/bengal}
entity recognition tools and merging their results. Our results also indicate that a better load balancing could lead to even better runtimes. In particular, the runtime per documents starts to increase as soon as the tool cannot handle the incoming amount of documents in time and the documents start to be queued (see Phase 2 to 4).

References


Appendix A: changes from version 1 to version 2

In this appendix, we briefly summarize the major changes that have been made to the first version of the platform when developing the second version. The list of changes is divided into features that have been added to the platform and problems, which were present in version 1 and have been fixed (either in task T2.2 or T2.3 of the project).

Features

- Improved the usability of the platform for normal users.
  - Added filtering options for list of experiments and challenges.
  - Added log file download for system and benchmark developers.
  - Better visualization of the platform status including information about the maximum runtime of the currently running experiment.
  - Enabled the deletion of experiments from the queue by the user who created them.
  - Implemented the visualization of result sets as diagrams.

- Improved the support of challenges.
  - Added support for repeatable challenges.
  - Added leaderboards for challenges.
  - Added a field to add a link to a homepage to a challenge.

- Improved the usability of the platform for local deployments.
  - Added an optional configuration file which can be used to define the maximum runtime of experiments based on the benchmark used in the experiment and whether it belongs to a challenge.
  - Deactivation of `docker pull` commands for local development.
  - Eased the deployment process of the platform.

- Offered a Java SDK to ease the development of benchmarks and system adapters.

- Added support of the forwarding of parameters from the benchmark to the system.

- Added support for gathering the amount of resources used by the benchmarked system.

- Added support for streaming data removing the limitation of single tasks or results being bound to a maximum size of 1GB.

Fixes

- Increased the robustness of several parts including
  - Gitlab crawling,
  - User input processing,
  - Missing Docker images,
- Docker container creation, manipulation and observation.
  
  * Complete rework of the graphical user interface including
    
    - Fixed problems when changing challenge configurations.
    - Fixed problems when registering systems for a challenge.
    - Fixed problems when displaying large KPI results.
    - Fixed visibility problems of systems and benchmarking results.
    - Added the missing displaying of benchmark parameters and challenge information on the experiment result page.
    - Added more explaining messages, e.g., when an experiment with a given ID couldn’t be found.
    - Added a default sorting to tables and drop down menus.
    - Caching problems in the users browser.

  - Added an authentication to the platform to enable the access to private Docker images in the name of the user.
  
  * Fixed several problems regarding pulling Docker images.
  
  * Enabled the gelf logging adapter for containers created by the platform.
  
  * Fixed the accidental resubmission of created experiments when refreshing the submit page.
  
  * Enabled the separation of result models if they can not be stored using one large SPARQL insert query.
  
  * Fixed the order of experiments in the experiment queue.
  
  * Fixed the problem that a system received the complete system.ttl file with all systems defined in that file.
  
  * Fixed communication problems between Keycloak and the platform front end which where depending on the OS on which the platform has been deployed.
  
  * While a user created a new experiment, the queue status could not be loaded (and not be shown to other users).
  
  * Fixed problems with Docker swarm which was not correctly updating the status of containers.
  
  * Fixed problems with the default evaluation storage that is offered to other benchmark developers.