Collaborative Project
Holistic Benchmarking of Big Linked Data

Project Number: 688227  Start Date of Project: 2015/12/01  Duration: 36 months

Deliverable 2.2.1
First 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 15, 28/02/2017</td>
</tr>
<tr>
<td>Actual Submission Date</td>
<td>Month 15, 28/02/2017</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>43</td>
</tr>
</tbody>
</table>

Abstract: This deliverable presents the first 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>26/01/2017</td>
<td>First draft created</td>
<td>Michael Röder</td>
</tr>
<tr>
<td>0.2</td>
<td>30/01/2017</td>
<td>Added user guide</td>
<td>Roman Korf</td>
</tr>
<tr>
<td>0.9</td>
<td>07/02/2016</td>
<td>Beta version created</td>
<td>Michael Röder</td>
</tr>
<tr>
<td>0.10</td>
<td>22/02/2016</td>
<td>Reviewed version created</td>
<td>Martin Strohbach</td>
</tr>
<tr>
<td>1.0</td>
<td>27/02/2016</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>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 first version of the platform in detail. It gives a brief introduction before it describes 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 4.
Contents

1 Introduction 5

2 Architecture 5
  2.1 Benchmarking a System 6
  2.2 Platform Components 8
    2.2.1 Platform Controller 8
      2.2.1.1 Persistent Status 8
      2.2.1.2 Queue 8
      2.2.1.3 Benchmark Execution 9
      2.2.1.4 Interaction with Analysis Component 9
      2.2.1.5 Docker Container Creation 9
    2.2.2 Storage 9
    2.2.3 Front End 10
      2.2.3.1 Roles 10
    2.2.4 User Management 10
    2.2.5 Repository 10
    2.2.6 Analysis 10
    2.2.7 Message Bus 10
    2.2.8 Logging 11
      2.2.8.1 Logging inside Components 12
  2.3 Benchmark Components 12
    2.3.1 Benchmark Controller 12
      2.3.1.1 Workflow 12
      2.3.1.2 Error Handling 13
    2.3.2 Data Generator 14
    2.3.3 Task Generator 14
    2.3.4 Evaluation Storage 15
    2.3.5 Evaluation Module 15
  2.4 Benchmarked System Components 16
  2.5 Ontology 16
    2.5.1 Classes 18
      2.5.1.1 hobbit:Benchmark 18
2.5.2 Properties ......................................................... 22

3 User Manuals ....................................................... 23

  3.1 Register ......................................................... 23
  3.2 Upload a system .................................................. 24
    3.2.1 Uploading the Docker image ............................... 24
    3.2.2 System metadata file ..................................... 26
  3.3 Upload a benchmark ............................................. 28
  3.4 Benchmark a system ............................................ 30
  3.5 Explore experiment results ................................... 33
  3.6 Register for a challenge ...................................... 34
  3.7 Create a challenge ............................................ 36

4 Evaluation .......................................................... 42

References ............................................................ 42
1 Introduction

This document describes the first 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>
</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. Finally, the evaluation of the platform is described in Section 4. 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.

2 Architecture

The Hobbit platform comprises 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, i.e., the benchmark components as well as 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 data flow shown in the overview picture is carried out using the RabbitMQ message bus. However, the Hobbit platform is modular in a way that allows the benchmark to use a different

\[https://docker.com/\]

\[https://www.rabbitmq.com/\]
middleware. Thus, a benchmark might use network sockets or a streaming framework for the communication between the benchmark components and the communication with the system. For the communication with the platform controller component 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.

![Diagram of platform components](image)

Figure 1: Overview of the platform components.

### 2.1 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 meta data 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 generates the system that should be benchmarked.
   - 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 generates the benchmark controller.
   - The benchmark controller generates the data and task generators as well as the evaluation storage.
   - It 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 mimicking algorithms to create the 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 store 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 creates the evaluation module.

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

Figure 2: Overview of the general benchmarking workflow. For keeping simplicity, the benchmarked system as well as the front end are left out.
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.

Beside 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.

2.2 Platform Components

The platform has several components that are started and stopped together with the complete platform.

2.2.1 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, the starting and stopping of benchmarks, the observing of the cluster health status and the triggering of the analysis component.

2.2.1.1 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 the current status, e.g., benchmarks that have been configured and scheduled inside the queue.

2.2.1.2 Queue

User configured experiments get a unique HOBBIT ID and are put into a 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.
2.2.1.3 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 generates the system.
3. The platform controller generates the benchmark controller.
4. 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.
5. 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.
6. The platform controller receives the results from the benchmark controller.
7. 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.

2.2.1.4 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.

2.2.1.5 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.

2.2.2 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. The storage component offers a public SPARQL Endpoint with read-only access.

2.2.3 Front End

The front end component handles the interaction with the user. It offers different functionalities to the 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.

2.2.3.1 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.

2.2.4 User Management

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

2.2.5 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.

2.2.6 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.

2.2.7 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:

---

\(^3\) [http://www.keycloak.org/](http://www.keycloak.org/)

\(^4\) [https://about.gitlab.com/](https://about.gitlab.com/)
- **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 the benchmark. However, if the benchmark relies on RabbitMQ, the benchmark implementation has to add the ID of the experiment to the queue name to enable a parallel execution of benchmarks.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hobbit.command</td>
<td>Bus</td>
<td>Broadcasts commands to all connected components.</td>
</tr>
<tr>
<td>hobbit.storage</td>
<td>RPC</td>
<td>The storage component accepts reading and writing SPARQL queries. The response contains the query result.</td>
</tr>
<tr>
<td>hobbit.frontend-controller</td>
<td>RPC</td>
<td>Used by the front end to interact with the controller.</td>
</tr>
<tr>
<td>hobbit.controller-analysis</td>
<td>Simple</td>
<td>Used by the controller to send the URIs of finished experiments to the analysis component which uses them for further analysis.</td>
</tr>
</tbody>
</table>

Table 2: List of queues used inside the platform.

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 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.

### 2.2.8 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.
<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`

2.2.8.1 Logging inside Components

The single components should write log messages to the standard output. The docker containers will be configured in a way that sends these 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 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.

2.3 Benchmark Components

These components are part of the benchmark. They 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.\(^6\)

2.3.1 Benchmark Controller

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

2.3.1.1 Workflow

1. Is created by the platform controller with the following parameters
   - **Hobbit** ID of the experiment
   - benchmark specific parameters
   - the Docker container ID of the system

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

\(^5\)http://www.slf4j.org/

\(^6\)The 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).
3. The benchmark controller waits for the system to send the start signal (hobbit.command).

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

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

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

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

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

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

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

11. It creates the evaluation module (Docker).

12. It waits for the evaluation module to finish (Docker).

13. It sends a terminate signal to the 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 exits with code 0.

2.3.1.2 Error Handling

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

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

The benchmark controller will be informed by the platform controller that one of its component exited with an error code. The benchmark controller has to send a stop signal to all 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 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 stores the error in the storage to give the feedback to the user that an error occurred.

If the benchmark controller has a faulty implementation and does not handle the error in the way described above the platform controller implements a fall back strategy. Firstly, a benchmark that does not terminate during a preconfigured 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 preconfigured run time has 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.
### 2.3.2 Data Generator

The data generator contains a mimicking algorithm implementation that is able to generate the data needed for the evaluation. For benchmarking a system based on big data, the data generator might be instantiated several times. Each instance will receive an ID that should be used to generate different data.

Additionally, the 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
   - **HOBBIT ID** of the experiment
   - **ID** of this particular generator instance 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.

### 2.3.3 Task Generator

The task generator gets the data from the data generator and generates tasks that can be identified with an ID. This ID is needed to map the system responses to the 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 response is sent to the evaluation storage.

During a benchmark experiment, the task generator typically performs the following steps

1. It is created by the benchmark controller with the following parameters
   - **HOBBIT ID** of the experiment
   - **ID** of this particular generator instance 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 tasks.
It reads incoming data from the data generators.
• It generates a task and the expected solution.
• It sends the task to the 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 status code 0.

2.3.4 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 result calculated by the benchmarked system.

During a benchmark experiment, the task generator typically performs the following steps

1. It is created by the benchmark controller with the HOBBIT ID of the experiment.
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 reacts to all incoming queues and stores (expected) results. 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 the result pairs. Every 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.

We offer a default implementation of this component written in Java based on the Riak\(^7\) key-value store.

2.3.5 Evaluation Module

The evaluation module evaluates the result generated by the 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.

1. It is created by the benchmark controller with the HOBBIT ID of the experiment.
2. It initializes itself and connects to the hobbit.command queue.

\(^7\)http://docs.basho.com/riak/latest/
3. It requests result pairs from the evaluation storage and evaluates 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.

### 2.4 Benchmarked System Components

The Hobbit platform does not have any requirements, how a system is structured internally. However, a system that should be benchmarked using the Hobbit platform 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 the system image, the user has to upload meta data about the system, e.g., the systems name. This data might contain parameters of the system that could be used by the analysis component.

2. During the creation 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 experiment, the system is taking part of.

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 which has to be used by the system.

### 2.5 Ontology

The experiments as well as the 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
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/rdf-syntax-ns#">http://www.w3.org/1999/02/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>

- 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.

Figure 3: Overview of the ontology including the benchmark, the system, their parameters and the experiment.
In the following sub sections, the single classes and properties of the ontology are described.

2.5.1 Classes

2.5.1.1 hobbit:Benchmark

Instances of this class define a particular benchmark. This includes the parameters the benchmark has, the KPIs it will create and the API that the benchmarked systems will have to implement to be benchmarked by it.

Datatype properties

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

Object properties

- hobbit:measuresKPI: The benchmark measures the KPI and will return it as result.
- hobbit:hasAPI: Connects the benchmark with the API the benchmarked system will have to implement to be benchmarked.
- hobbit:hasParameter: Connects the benchmark to the parameters with which the benchmark can be configured.

2.5.1.2 hobbit:Challenge

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

Datatype properties

- rdfs:label: The label of the Challenge.
rdfs:comment A description of the challenge.

hobbit:executionDate The date at which the execution of the experiments of this challenge starts.

hobbit:publicationDate The data at which the challenge results will be copied into the public graph, i.e., be publicly available.

hobbit:organizer The user identifier of the challenge organizer.

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

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

2.5.1.3 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 The 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 defining the 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 this challenge task.

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

hobbit:isTaskOf The challenge this task is part of.

2.5.1.4 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 The 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 the API it implements.
- `hobbit:instanceOf` Connects the instance to the system of which it is a configured instance.

### 2.5.1.5 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.

### 2.5.1.6 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` The timestamp of the start of the experiment.
- `hobbit:endTime` The 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 the `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` The 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.
2.5.1.7 **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.

2.5.1.8 **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.

2.5.1.9 **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.

2.5.1.10 **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.
2.5.1.11 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.

2.5.1.12 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.

2.5.2 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:defaultValue A default value of a hobbit:Parameter.
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: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.

3 User Manuals

In this section, the interaction of a user with the platform is described based on the user roles described in Section 2.2.3.1.

3.1 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 organiser. To become a challenge organiser the 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 the user has a login already she can login with her credentials. If she intends to use the HOBBIT benchmarking system only as guest she can press the Log in as Guest button and will be redirected to the HOBBIT benchmark page as Guest with restricted access rights.

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 Organiser and add and manage challenges the user needs to contact the platform host or administrator which can change the user’s roll from ordinary registered user to Challenge Organiser.

One can logout from the HOBBIT benchmark system using the logout option (cf. Figure 7).
3.2 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 systems 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 8).

3.2.1 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

```bash
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 of the project to which the image should be uploaded. Note that the letters should be lowercased even if the real user name and project name include upper case letters. Using non-alphanumeric characters is not allowed as it might create problems to use names with non-alphanumeric characters.
Figure 6: Hobbit registration page.

Figure 7: Hobbit logout.

Figure 8: Hobbit upload system or benchmark menu.
After the successful completion of the build process, the image can be pushed to the git project

```bash
1 docker login git.project-hobbit.eu:4567
2 docker push git.project-hobbit.eu:4567/maxpower/mysystem
3 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. Please note that Gitlab follows a "one image per project" philosophy. That means that for every Docker image, a new git project has to be created.

### 3.2.2 System meta data file

The system meta data 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.\(^8\) 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...").
- 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.

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

\(^8\)https://www.w3.org/TR/turtle/
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.
3.3 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 meta data 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 meta data file comprising information needed by the HOBBIT platform. The file contains the meta data 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 meta data 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.

```turtle
@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;
  rdfs:comment "This is my own example benchmark..."@en;
  hobbit:imageName "git.project-hobbit.eu:4567/maxpower/mybenchmarkcontroller";
  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.

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.


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.

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

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

10. rdfs:label "F-measure"@en ;
11. rdfs:comment "F-measure is the harmonic mean of precision and recall."@en ;
12. rdfs:range xsd:float .`

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.

2. rdfs:label "Dataset size"@en ;
3. rdfs:comment "The size of the generated dataset counted in triples"@en ;
4. rdfs:range xsd:unsignedInt ;
5. hobbit:defaultValue "10000"^^xsd:unsignedInt .`

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.

2. rdfs:label "Graph pattern"@en ;
3. rdfs:comment "The graph pattern used to generate the test data"@en ;
4. rdfs:range <http://www.example.org/exampleBenchmark/GraphPattern> ;
5. hobbit:defaultValue <http://www.example.org/exampleBenchmark/Star> .`
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.

2. `rdfs:label "Amount of blank nodes"@en`;
3. `rdfs:comment "The amount of blank nodes that have been created during the graph generation."@en`;
4. `rdfs:range xsd:float`.

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 the 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.

### 3.4 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 9) where at first the benchmark is selected to be used for the experiment. The drop down menu displays all possible benchmarks.

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 10). 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
Figure 9: HOBBIT configure benchmark experiment. (a) Select the benchmark.

Figure 10: HOBBIT configure benchmark experiment. (b) Select the system and configure experiment.

Submission details is presented (see Figure 11) to the user.
### Benchmark Submitted

**Benchmark**
http://w3id.org/gerbil/hobbit/vocab#GerbilBenchmark - GERBIL Benchmark

**System**
ex:DummySystem - Example GERBIL System

**Configuration Parameters**
- **Number of documents**: 100
- **Type of experiment**: gerbitA2KB

Submitted successfully HOBBIT ID: 1485780190338 at 2017-01-30T13:43:10.395+01:00

---

**Figure 11**: HOBBIT configure benchmark experiment. (c) Submission details.
3.5 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).

![Hobbit selecting result page.](image)

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

![Hobbit experiments result page.](image)

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. One additional option is to see all results of a specific challenge task. This is done via the selection of the challenges task name at the bottom of the challenge page.

The results are displayed next to each other (see Figure 15) for easy comparison with the parameter- (KPI-) names displayed on the left side of the table. If the experiment caused and logged an error, it is displayed as well. The error will be highlighted.
3.6 Register for a challenge

System providers can register for challenge tasks in a 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 organiser the page might vary (see Section 3.7). The registration of a system for a challenge task is done via the Challenges menu item (see Figure 16). The challenge page displays the challenges accessible by the user.

To register a system the user selects the challenge and is then redirected to the challenge page (see Figure 17). 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 18). 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 17). This will redirect the user to the challenge task details page (see Figure 19).
3.7 Create a challenge

A challenge organiser can create challenges. In order to do so the challenge organiser needs to be logged in (see Section 3.1). When selecting the Challenges menu item the challenge organiser, other than a conventional user, additionally sees the Add Challenge button (see Figure 20).

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 21). The challenge organiser is required to enter a challenge name, some description, an execution date (see Figure 23) and some publishing date (see Figure 21). 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 the challenge organiser can indicate if a challenge is Visible for everyone. Not setting this flag can be used, e.g., in order to save a challenge and re-edit it late on. In this case it will not be visible to anyone else. 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 a challenge the challenge is registered for execution at the platform. When pressing the button a dialogue comes up asking for closing the challenge (see Figure 22) to ensure this does not happen accidentally.

When all parameters are filled out, the challenge organiser is required to save the challenge (press...
the Save button) before adding tasks to the challenge.
Figure 21: HOBBIT new challenge form page.

Figure 22: HOBBIT closing a challenge.
This will redirect the challenge organiser to the challenge list page (see Figure 24). By clicking on the newly created challenge in the list the challenge organiser can change the challenge details and add tasks (see Figure 25). In order to add tasks the challenge organiser presses the *Add Task* button.

The challenge organiser is then redirected to the challenge task detail page (see Figure 26). Here the challenge organiser can name the task and give some more details via the description. Additionally the organiser is required to select and configure a benchmark that implements the described task (see Figure 27).

After having added the challenge tasks to the challenge (see Figure 27) 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.
Figure 25: **HOBBIT** partly filled challenge form page.

Figure 26: **HOBBIT** change challenge task details form page.

before saving the challenge.
Figure 27: HOBBIT change challenge task details form page with configured task benchmark.

Figure 28: HOBBIT filled challenge form page with challenge tasks.
4 Evaluation

The evaluation of the platform is focused on its usage in a common environment. Its goal is to show that it is possible to use the open-source platform even without a server environment. The scenario focuses on the sending of SPARQL queries and result sets during the evaluation phase of a SPARQL endpoint. Messages, that are transferred:

- from data generator to task generator: SPARQL query + result set
- from task generator to system: SPARQL query
- from task generator to evaluation storage: result set
- from system to evaluation storage: result set

From the Linked SPARQL Queries Dataset [1] it is known that the average size of a SPARQL query is 545.45 characters. A result set comprises 122.45 single results on average. Unfortunately, we do not know the average size of a single result. However, from several example queries we gathered the assumption that on average a result has 100 to 120 characters.

The platform was deployed on a small machine with an Intel Core i5 processor (2 cores with 2.5 GHz) and 2 GB RAM for the sake of the evaluation (note that a deployment on a cluster can be found at the URL of the demonstrator). The single benchmark runs are listed in Table 5. We executed the benchmark with three different numbers of queries that were generated with an expected response by two data generators. Our results show that the platform can run even within this minimalistic environment, hence proving (together with our development tests), that it is highly flexible. The results also clearly indicate the necessity to deploy the platform in a large-scale environment to test some of the Big Linked Data systems. In particular, our results indicate that the average runtime per query grows as well as the standard deviation of the runtime grows with an increase of the amount of queries. This clearly shows that generating too many query can lead to a traffic jam in the queues of the message bus. Still, we can conclude (1) that the platform can be executed even on systems with limited resources. However, (2) more powerful hardware is needed for benchmarking Big Linked Data systems.

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Experiment 1</th>
<th>Experiment 2</th>
<th>Experiment 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data generators</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Task generators</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>
</tr>
<tr>
<td>Avg. query runtime (in ms)</td>
<td>10,414</td>
<td>39,789</td>
<td>60,009</td>
</tr>
<tr>
<td>Query runtime std. dev.</td>
<td>2,086</td>
<td>2,536</td>
<td>7,832</td>
</tr>
<tr>
<td>Overall runtime (in s)</td>
<td>17.9</td>
<td>54.0</td>
<td>140.9</td>
</tr>
<tr>
<td>Queries per second</td>
<td>55.9</td>
<td>37.0</td>
<td>35.5</td>
</tr>
</tbody>
</table>

Table 5: Results of the platform benchmark on a single machine.
References