Contents

1 Introduction to EverBEEN **3**

1.1 Foreword . . . 3
1.2 Case study . . . 3
1.3 Target audience . . . 4
1.4 Project history . . . 4
1.5 Project goals . . . 4

2 EverBEEN user guide **7**

2.1 EverBEEN requirements . . . 7
2.2 Deployment process . . . 7
2.3 EverBEEN controls . . . 7
2.4 Task and Benchmark API . . . 8
2.4.1 Maven Plugin and Packaging . . . 8
2.4.2 Descriptor Format . . . 9
2.4.3 Task API . . . 10
2.4.4 Task Properties . . . 11
2.4.5 Persisting Results . . . 11
2.4.6 Checkpoints and Latches . . . 11
2.4.7 Benchmark API . . . 12
2.4.8 Creating Task Contexts . . . 13
2.4.9 Resubmitting and Benchmark Storage . . . 13
2.4.10 Evaluators . . . 14
2.5 Software repository and BPKs . . . 14
2.6 Persistence layer . . . 15
2.6.1 Characteristics . . . 15
2.6.2 Components . . . 16
2.6.3 Persistence extension points . . . 16
2.7 EverBEEN configuration . . . 19
2.7.1 Configuration options . . . 19
2.8 EverBEEN best practices . . . 21

3 EverBEEN developer documentation **23**

3.1 Design goals . . . 23
3.2 Decision timeline . . . 23
3.3 EverBEEN architecture . . . 24
3.4 EverBEEN services . . . 24
3.4.1 Host Runtime . . . 24
3.4.2 Task Manager . . . 25
3.4.3 Software Repository . . . 28
3.4.4 Object Repository . . . 28
3.4.5 Map Store . . . 28
3.4.6 Web Interface . . . 29
3.5 Modular approach . . . 29
3.6 Used technologies . . . 29
3.6.1 Hazelcast . . . 30
3.6.2 0MQ . . . 30
3.6.3 Apache Maven . . . 30
3.6.4 Apache Commons Exec . . . 30
3.6.5 Apache Commons . . . 30
3.6.6 Apache HTTP Core/Components . . . 30
3.6.7 Bootstrap . . . 30
3.6.8 Hazelcast . . . 30
3.6.9 0MQ . . . 30
3.6.10 Jackson . . . 30
3.6.11 JAXB . . . 31
3.6.12 Logback (logging impl) . . . 31
3.6.13 MongoDB . . . 31
3.6.14 SLF4J . . . 31
3.6.15 Tapestry . . . 31
3.6.16 Other . . . 31
3.7 Principal features . . . 31
3.8 Current limitations . . . 32
3.9 In-code documentation . . . 32

Chapter 1

Introduction to EverBEEN

1.1 Foreword

- common testing methods
- – unit testing
- – integration testing
- regression benchmarking
- – not so common
- – mostly project specific
- generic framework for regression benchmarking
- – required properties
- – previous research (thesis etc)

1.2 Case study

- regression benchmarking lifecycle
- – back-to-back measures between revisions of the same software
- – granularity refinement on anomaly must be possible
- common use-cases
- – push-oriented (CIT->CRT)
- – pull-oriented (repo-spanning)

1.3 Target audience

- required knowledge
- – of tested software
- – of benchmarking practices
- regression benchmarking audience
- – developers/testers of a software project
- – outputs presentable outside these circles

1.4 Project history

- previous attempts at the project
- – BEEN
- – WillBEEN
- extending approach
- – features added in second incarnation
- – lack of technological innovation
- WillBEEN state overview in 2012
- – deployment difficulties
- – feature usability
- – code obsoletion (lack of delegation to libraries, obsolete package versions)
- – overhaul necessity (non-maintainable code - size and spaghetti factor)

1.5 Project goals

- formal goals
- – refactoring focus
- – code reuse
- refactoring clash points
- – non-modular architecture
- – RMI
- – code scatter (multiple implementation of the same functionality)
- – failure to delegate (custom implementations of logging, communication etc. over library use)
- goal adaptation
- – preserve the concept (time-proven)

- – revamp the code using modern technologies and practices
- – make it scale
- – make it deployable
- – make it usable (simplify task api)

Chapter 2

EverBEEN user guide

2.1 EverBEEN requirements

What a user needs to run BEEN * JRE 1.7 on every node * Mongo, or implement some stuff and use another dbase * some recommended hardware specs maybe?

2.2 Deployment process

- get a set of machines interconnected with a network
- deploy Mongo on 1 or more of them
- create clustering configurations
- – publish or otherwise distribute them
- assign a webapp container
- assign data nodes (explain node types here)
- assign service runner nodes (SWRepo and Repo)
- run nodes
- run webapp and connect

2.3 EverBEEN controls

- overview description
- viewing service status
- how to submit task to SW repo
- how to submit task/context/benchmark
- viewing outcomes & getting to logs
- explain how to clean up after something (& what does it delete)
- ...
- any other tutorials that come to mind

2.4 Task and Benchmark API

TODO: AQL (description of the abstract querying language API)

One of the main goals of the current BEEN project was making the task API as simple as possible and to minimize the amount of work needed to create the whole benchmark. This was of the biggest problems with the previous BEEN versions as writing a complete and efficient benchmark required a tremendous amount of time both to study the provided API with the related Java classes and to implement the benchmark itself.

BEEN works with three different concepts of user-supplied code and configuration:

- **Task**, which is an elementary unit of code that can be submitted and run by BEEN. Tasks are created by subclassing the abstract `Task` class and implementing the appropriate methods. Each task has to be described by a XML **task descriptor** which specifies the main class to run and parameters of the task.
- **Task context** is a container for multiple tasks that can interact together, pass data to each other and synchronize among themselves. Tasks contexts don't contain any user-written code, they only serve as a wrapper for the contained tasks. Each task context is described by a XML **task context descriptor** that specifies which tasks should be contained within the context.
- **Benchmark** is a first-class object that *generates* task contexts based on its **generator task**, which is again a user-written code created by subclassing the abstract `Benchmark` class. Each benchmark is described by a XML **benchmark descriptor** which specifies the main class to run and parameters of the benchmark. A benchmark is different from a task, because its API is provides features for generating task contexts and it can also persist its state so it can be re-run when an error occurs and the generator task fails.

All these three concepts can be submitted to BEEN and run individually, if you only want to test a single task, you can submit it without providing a task context or a whole benchmark.

2.4.1 Maven Plugin and Packaging

The easiest way to create a submittable item (e.g. a task) is by creating a Maven project and adding a dependency on the appropriate BEEN module (e.g. `task-api`) in `pom.xml` of the project:

```
<dependency>
    <groupId>cz.cuni.mff.d3s.been</groupId>
    <artifactId>task-api</artifactId>
    <version>3.0.0</version>
</dependency>
```

Tasks, contexts and benchmark must be packaged into a BPK file, which can then be uploaded to the BEEN cluster. Each BPK package can contain multiple submittable items and multiple XML descriptors. The problem of packaging is made easier by the supplied `bpk-plugin` Maven plugin. To preferred way to use it is to add this plugin to the `package` Maven goal in `pom.xml` of the project:

```
<plugin>
    <groupId>cz.cuni.mff.d3s.been</groupId>
    <artifactId>bpk-plugin</artifactId>
    <version>3.0.0</version>
    <executions>
        <execution>
            <goals>
```

```
                <goal>buildpackage</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        ...
    </configuration>
</plugin>
```

In the plugin's configuration the user must specify at least one descriptor of a task, a context or a benchmark. To add a descriptor into the BPK, it should be added as a standard Java resource file and then referenced in the plugin configuration in `pom.xml` by using `<taskDescriptors>` or `<taskContextDescriptors>` element. For example the provided sample benchmark called `nginx-benchmark` uses this configuration:

```
<configuration>
    <taskDescriptors>
        <param>src/main/resources/cz/cuni/mff/d3s/been/nginx/NginxBenchmark.td.xml</param>
    </taskDescriptors>
</configuration>
```

This specifies that the package should publish a single descriptor named `NginxBenchmark.td.xml` which is located in the specified resource path. With such a configuration, creating the BPK package is simply a matter of invoking `mvn package` on this project – this will produce a `.bpk` file that can be uploaded into BEEN.

2.4.2 Descriptor Format

There are two types of descriptors, task descriptors and task context descriptors. Note that benchmarks don't have a special descriptor format, instead you only provide a task descriptor for a generator task of the benchmark. These descriptors are written in XML and they must conform to the supplied XSD definitions (`task-descriptor.xsd` and `task-context-descriptor.xsd`).

The recommended naming practice is to name your task descriptors with the filename ending with `.td.xml` and your task context descriptors ending with `.tcd.xml`.

A simple task descriptor for a single task can look like this:

```
<?xml version="1.0"?>
<taskDescriptor xmlns="http://been.d3s.mff.cuni.cz/task-descriptor"
                groupId="my.sample.benchmark" bpkId="hello-world" version="3.0.0-SNAPSHOT"
                name="hello-world-task" type="task">
    <java>
        <mainClass>my.sample.benchmark.HelloWorldTask</mainClass>
    </java>
</taskDescriptor>
```

This specifies the main class and package that should be used to run the task. Apart from this, you can specify what parameters the task should receive and their default values:

```
<properties>
    <property name="key">value</property>
</properties>
```

These properties will be presented to the user in the web interface before submitting the task and the user can modify them. Next, you can specify command line arguments passed to Java:

```
<arguments>
    <argument>-Xms4m</argument>
    <argument>-Xmx8m</argument>
</arguments>
```

For debugging purposes, you can specify the `<debug>` element which will enable remote debugging when running the task. With the `<hostRuntimes>` element you can filter on which host runtimes can the task be run. The value of this setting is an XPath expression.

2.4.3 Task API

To create a task submittable into BEEN, you should start by subclassing the `Task` abstract class. To do this, you only need to provide a single method called `run` which will optionally receive string arguments.

BEEN uses `slf4j` as its logging mechanism and provides a logging backend for all user-written code. This means that you can simply use the standard loggers and any logs with be automatically stored in the BEEN cluster.

Knowing this, the simplest task that will only log a single string can look something like this:

```
package my.sample.benchmark;

import cz.cuni.mff.d3s.been.taskapi.Task;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorldTask extends Task {
    private static final Logger log = LoggerFactory.getLogger(HelloWorldTask.class);

    @Override
    public void run(String[] args) {
        log.info("Hello, world!");
    }
}
```

If this class is in a Maven project as described in the previous section, it can be packaged into a BPK package by invoking `mvn package`. This package can be uploaded and run either from the web interface or client submitter.

BEEN provides several APIs for user-written tasks:

- *Properties* – Tasks are configurable either from their descriptors or by the benchmark that generated them. These properties are again configurable by the user before submitting the task. All properties have a name and a simple string value and these can be accessed via the `getProperty` method of the abstract `Task` class.
- *Result storing* – Each task can persist a result that it has gathered by using the API providing access to the persistence layer. To store a result, use a `ResultPersister` object, which can be created by using the method `createResultPersister` from the `Task` abstract class.

- *Synchronization and communication* – When multiple tasks run in a task context, they can interact with each other either for synchronization purposes or to exchange data. API for these jobs are provided by the `CheckpointController` class. BEEN provides the concepts of **checkpoints** and **latches**. Latches serve as context-wide atomic numbers with the methods for setting a value, decreasing the value by one and waiting until the latch reaches zero. Checkpoint are also waitable objects, but they can also provide a value that was previously set to the checkpoint.

2.4.4 Task Properties

Every tasks has a key-value property storage. These properties can be set from various places: From the XML descriptor, from the user when submitting, inherited from a task context, set from a benchmark when it generates a task context. To access these values, you can use the `getProperty` method of the `Task` class:

2.4.5 Persisting Results

XXX TODO

Persisting a result:

```
SampleResult result = ...;
EntityID eid = new EntityID().withKind("result").withGroup("sample");
ResultPersister rp = results.createResultPersister(eid));
rp.persist(result);
```

2.4.6 Checkpoints and Latches

Checkpoints present a powerful mechanism for synchronization and communication between tasks. When tasks run in a task context, they share all their checkpoints and they can set a value to a checkpoint and another can wait for the checkpoint. This waiting is passive and once a value is assigned to a checkpoint, the waiter will receive it.

To use checkpoints, create a `CheckpointController`, which is an `AutoCloseable` object so the preferred way to use it is inside a try-catch block to ensure the object will be properly destroyed:

```
try (CheckpointController requestor = CheckpointController.create()) {
    ...
} catch (MessagingException e) {
    ...
}
```

Each checkpoint has a name, which is context-wide. All communication between tasks can only be done inside a single task context. You don't have to explicitly create a checkpoint, it will be created automatically once a task uses it. Setting a value to a checkpoint can be done with:

```
requestor.checkPointSet("mycheckpoint", "the value");
```

A typical scenario is that one tasks wants to wait for another to pass a value. To wait until a value is set and also to receive the value you can use:

```
String value = requestor.checkPointWait("mycheckpoint");
```

This call passively waits (possibly indefinitely) until a value is set to the checkpoint. There is also a variant of this method that takes another argument specifying a timeout, after which the call will throw an exception. Another method called `checkPointGet` can be used to retrieve the current value of a checkpoint without waiting.

Checkpoints initially don't have any value, and once a value is set, it cannot be changed. They work as a proper synchronization primitive, and setting a value is an atomic operation. The semantics don't change if you start waiting before or after the value is set.

Another provided synchronization primitive is a *latch*. They work best for counting values and for implementing rendez-vous synchronization. A latch provides a method to set an integer value:

```
requestor.latchSet("mylatch", 5);
```

Another task can then call an atomic method to decrease the value of the latch:

```
requestor.latchCountDown("mylatch");
```

You can then wait until the value reaches zero:

```
requestor.latchWait("mylatch");
```

All operations on latches are atomic and the waiting is passive. Latches initially have a value of zero.

2.4.7 Benchmark API

Writing a benchmark's generator task is similar to writing an ordinary task in the sense that you have to write a subclass, package it and then it will be submitted and run on a host runtime. However, the benchmark API is completely different, because the purpose of the benchmark is to provide a long-running code that will eventually generate new task contexts whenever there is all data for it available.

To create a benchmark, subclass the abstract `Benchmark` class and implement the appropriate methods. The main method to implement is `generateTaskContext` which is called periodically by BEEN and it is expected to return a newly generated task context. This context is then submitted and run. When the context finishes, this method is called again. This loop is ended whenever this method returns `null`.

This approach is chosen to cover a lot of possible use cases. When the benchmark doesn't have data for a new task context, it can simply block until it is possible to create a new context. On the other hand, the benchmark cannot overhaul the cluster by submitting too many contexts. Instead, it's up to the cluster to call the `generateTaskContext` method whenever it seems fit.

For creating task contexts you should use the provided `ContextBuilder` class. This supports loading a task context from a XML file, modifying it and setting values of properties inside the context descriptor. If you have a prepared `.tcd` file with a context descriptor, a sample benchmark that will indefinitely generate this context can look like this:

```
package my.sample.benchmark;

import cz.cuni.mff.d3s.been.benchmarkapi.Benchmark;
import cz.cuni.mff.d3s.been.benchmarkapi.BenchmarkException;
import cz.cuni.mff.d3s.been.benchmarkapi.ContextBuilder;
import cz.cuni.mff.d3s.been.core.task.TaskContextDescriptor;
import cz.cuni.mff.d3s.been.core.task.TaskContextState;
```

```
public class HelloWorldBenchmark extends Benchmark {
    @Override
    public TaskContextDescriptor generateTaskContext() throws BenchmarkException {
        ContextBuilder contextBuilder = ContextBuilder.createFromResource(HelloWorldBenchmark.class, "c
        TaskContextDescriptor taskContextDescriptor = contextBuilder.build();
        return taskContextDescriptor;
    }

    @Override
    public void onResubmit() { }

    @Override
    public void onTaskContextFinished(String s, TaskContextState taskContextState) { }
}
```

Notice the methods `onResubmit` and `onTaskContextFinished` which are used as notifications for the benchmark. You can use these methods for whatever error handling or logging you need.

You are supposed to implement the logic for generating the contexts. When your benchmark is done and it will not generate any more contexts, return `null` from the `generateTaskContext` method.

2.4.8 Creating Task Contexts

The preferred way of creating task contexts is to use the `ContextBuilder` class to load a XML file representing the context descriptor from a resource. This class also provides various methods for modifying the context descriptor and the contained tasks.

You can add tasks into the context via the `addTask` method, these tasks can be created using the `newEmptyTask`. The context descriptor can also provide *task templates* which can be used to create tasks.

However, when it's sufficient, you should create the whole descriptor in the XML file and only use `setProperty` to set the parameters to this task contexts. When the descriptor is done, you can generate it by calling `build` and return the resulting task context descriptor.

2.4.9 Resubmitting and Benchmark Storage

Benchmarks are supposed to be long-running and BEEN provides a mechanism to keep benchmarks running even after a failure occurs. When a generator task exits with an error (e.g. power outage), it will get resubmitted and the benchmark will continue. To support this behavior, you should use the provided benchmark key-value storage for the internal state of the benchmark and not use any instance variables.

The `Benchmark` abstract class provides methods `storageGet` and `storageSet` which will use the cluster storage for the benchmark state. This storage will be restored whenever the generator task is resubmitted. The implementation of a benchmark that uses this storage can look like this:

```
@Override
public TaskContextDescriptor generateTaskContext() throws BenchmarkException {
    int currentRun = Integer.parseInt(this.storageGet("i", "0"));
    TaskContextDescriptor taskContextDescriptor;
    if (currentRun < 5) {
        // generate a regular context
        taskContextDescriptor = ...;
    } else {
        // we're done
```

```
        taskContextDescriptor = null;
    }

    currentRun++;
    this.storageSet("i", Integer.toString(currentRun));

    return taskContextDescriptor;
}
```

2.4.10 Evaluators

BEEN provides a special task type called **evaluator**. The purpose of such a task is to query the stored results, perform and statistical analyses and return an interpretation of the data that can be shown back to the user via web interface. Evaluators are again tasks and they can be run manually (as a single task) or within a benchmark or a context. It's up to the user when and how to run an evaluator.

To create an evaluator, subclass the abstract class `Evaluator` and implement the method `evaluate`. This method is supposed to return a `EvaluatorResult` object which will then be stored in the database. This object holds a byte array of data and a MIME type of it. BEEN supports a few MIME types with which it can directly work, e.g. if the result is a JPEG image, it will show it directly in the web interface. The list of supported MIME types can be found defined as contants inside the `EvaluatorResult` class.

An evaluator needs to retrieve data from the persistence layer, and it can do so using the provided `ResultFacade` interface. This object is available as an instance method on the `Task` superclass. Queries can be build using the `QueryBuilder` object which supports various conditions and query parameters. A simple query that will retrieve a collection of results can have this form:

```
Query query = new QueryBuilder().on(...).with(...).fetch();
Collection<MyResult> data = results.query(query, MyResult.class);
```

For example code of a simple evaluator that output a plot chart with the measured data and error intervals, see the sample `nginx-benchmark`.

2.5 Software repository and BPKs

- what's a BPK
- – what is it used for
- – short description on the philosophy of storing BPKs
- – versioning
- how to create a BPK
- – describe plugin configuration
- – BPK contents (current use - full-classpath BPKs) - for suicidals interested in manual createion

2.6 Persistence layer

EverBEEN persistence layer functions as a bridge between EverBEEN distributed memory and a database of choice, rather than a direct storage component. This enables EverBEEN to run without a persistence layer, at the cost of heap space and a risk of data loss in case of an unexpected cluster-wide shutdown. EverBEEN doesn't *need* a persistence layer per-se at any given point in time. User tasks, however, might attempt to work with previously acquired results. Such attempts will result in task-scope failures if the persistence layer is not running. Log archives, too, will be made unavailable if the persistence layer is offline.

2.6.1 Characteristics

Follows an overview of the main characteristics of EverBEEN's persistence layer.

2.6.1.1 Bridging

The EverBEEN persistence layer doesn't offer any means of storing the objects per se. It only functions as an abstract access layer to an existing storage component (e.g. a database). EverBEEN comes with a default implementation of this bridge for the MongoDB database, but it is possible to port it to a different database (see extension point notes for more details). The user is responsible for setting up, running and maintaining the actual storage software.

2.6.1.2 Eventual persistence

As mentioned above, object-persisting commands (result stores, logging) do not, by themselves, execute insertions into the persistence layer. They submit objects into EverBEEN's distributed memory. When a persistence layer node is running, it continually drains this distributed memory, enacting the actual persistence of drained objects. This offers the advantage of being able to pursue persisting operations even in case the persistence layer is currently unavailable.

The downside of the bridging approach is that persisted objects might not find their way into the actual persistence layer immediately. It also means that should a cluster-wide shutdown occur while some objects are still in the shared memory, these objects will get lost. All that can be guaranteed is that submitted objects will eventually be persisted, provided that some data nodes and a persistence layer are running. This being said, experience shows that the transport of objects through the cluster and to the persistence layer is a matter of fractions of a second.

2.6.1.3 Scalability

As mentioned above, EverBEEN does not strictly rely on the existence of a persistence node for running user code, only to present the user with the data he requires. That being said, EverBEEN can also run multiple persistence nodes. In such case, it is the user's responsibility to set up these nodes in a way that makes sense.

While running multiple nodes, please keep in mind that these storage components will be draining the shared data structures concurrently and independently. It is entirely possible to setup EverBEEN to run two persistence nodes on two completely separate databases, but it will probably not result in any sensibly expectable behavior, as potentially related data will be scattered randomly across two isolated database instances.

Generally speaking, having multiple persistence layer nodes is only useful if you:

- Have highly limited resources for each persistence node and wish to load-balance accesses to the same database

- Have a synchronization/sharding strategy set up

Additional use-cases may arise if you decide to write your own database adapter. In that case, consult the extension point for more detail.

2.6.1.4 Automatic cleanup

To prevent superfluous information from clogging the data storage, the Object Repository runs a Janitor component that performs database cleanup on a regular basis. The idea is to clean all old data for failed jobs and all metadata for successful jobs after a certain lifecycle period has passed. For lifecycle period and cleanup frequency adjustment, see the janitor configuration section.

2.6.2 Components

Follows a brief description of components that contribute to forming the EverBEEN persistence layer.

- Object Repository
- Storage
- MapStore

2.6.2.1 Object Repository

It goes without saying that EverBEEN needs some place to store all the data your tasks will produce. That's what the Object Repository is for. Each time a task issues a command to submit a result, or logs a message, this information gets dispatched to the cluster, along with the associated object. The Object Repository provides a functional endpoint for this information. It effectively concentrates distributed data to its intended destination (a database, most likely). In addition, the Object Repository is also in charge of dispatching requested user data back.

2.6.2.2 Storage

The Storage component supplies the concrete database connector implementation. All communication between the Object Repository and the database is done through the Storage API.

The Storage component gets loaded dynamically by the Object Repository at startup. If you want to use a different database than MongoDB, this is the component you'll be replacing (along with the MapStore, potentially).

2.6.2.3 MapStore

Where the ObjectRepository stores user data, the MapStore is used to map EverBEEN cluster memory to a persistent storage, which enables EverBEEN to preserve job state memory through cluster-wide restarts. The MapStore runs on all *data nodes* (see deployment for more information on node types).

2.6.3 Persistence extension points

As mentioned above, EverBEEN comes with a default persistence solution for MongoDB. We realize, however, that this might not be the ideal use-case for everyone. Therefore, the MongoDB persistence layer is fully replaceable if you provide your own database implementation.

There are two components you might want to override - the Storage and the MapStore.

If your goal is to relocate EverBEEN user data (benchmark results, logs etc.) to your own database and don't mind running a MongoDB as well for EverBEEN service data, you'll be fine just overriding the Storage. If you want to completely port all of EverBEEN's persistence, you'll have to override the MapStore as well.

2.6.3.1 Storage

As declared above, the *Storage* component is fully replacable by a different implementation than the default MongoDB adapter. However, we don't feel comfortable with letting you plunge into this extension point override without a few warnings first.

2.6.3.1.1 Override warning The issue with *Storage* implementation is that the persistence layer is designed to be completely devoid of any type knowledge. The reason for this is that *Storage* is used to persist and retrieve objects from user tasks. Should the *Storage* have any RTTI knowledge of the objects it works with, imagine what problems could arise when two tasks using two different versions of the same objects would attempt to use the same *Storage*.

To avoid this, the *Storage* only receives the object JSON and some information about the object's placement. This being said, the *Storage* still needs to perform effective querying based on some attributes of the objects it is storing.

This is generally not an issue with NoSQL databases or document-oriented stores, but it can be quite hard if you use a traditional ORM. The ORM approach additionally presents the aforementioned class version problem, which you would need to solve somehow. If ORM is the way you want to go, be prepared to run into the following:

- **EverBEEN classes** - You will probably need to map some of these in your ORM
- **User types** - You will likely need to share a user-type library with your co-developers to consent on permitted result objects
- **User type versions** - Should the version of this user-type library change, you will need to restart the *Storage* before running any new tasks on EverBEEN. Restarting EverBEEN will likely result in the dysfunction of tasks using an older version of the user-type library

2.6.3.1.2 Override implementation overview If your intention is not to use ORM for *Storage* implementation, or you have really thought the consequences through, keep reading. To successfully replace the *Storage* implementation, you'll need to implement the following:

- Storage
- StorageBuilder

Additionally, you'll need to create a **META-INF/services** folder in the jar with your implementation, and place a file named **cz.cuni.mff.d3s.been.storage.StorageBuilder** in it. You'll need to put a single line in that file, containing the full class name of your StorageBuilder implementation.

We also strongly recommend that you implement these as well:

- QueryRedactorFactory (along with QueryRedactor implementations)
- QueryExecutorFactory (along with QueryExecutor implementations)

The general idea is for you to implement the *Storage* component and to provide the *StorageBuilder* service, which configures and instantiates your *Storage* implementation. The **META-INF/services** entry is for the

ServiceLoader EverBEEN uses to recognize your *StorageBuilder* implementation on the classpath. EverBEEN will then pass the *Properties* from the *been.conf* file (see configuration) to your *StorageBuilder*. That way, you can use the common property file for your *Storage*'s configuration.

The Storage interface is the main gateway between the Object Repository and the database. When overriding the Storage, there will be two major use-cases you'll have to implement: the asynchronous persist and the synchronous query.

2.6.3.1.3 Asynchronous persist All *persist* requests in EverBEEN are funneled through the store method. You'll receive two parameters in this method:

entityId The *entityId* is meant to determine the location of the stored entity. For example, if you're writing an SQL adapter, it should determine the table where the entity will be stored. For more information on the *entityId*, see persistent object info

JSON A serialized JSON representation of the object to be stored.

Generally, you'll need to decide where to put the object based on its *entityId* and then somehow map and store it using its *JSON*.

The store method is asynchronous. It doesn't return any outcome information, but be sure to throw a *DAOException* when the persist attempt fails. That way, you'll make sure the *ObjectRepository* knows that the operation failed and will take action to prevent data loss.

2.6.3.1.4 Query / Answer

2.6.3.1.5 General persistent object info Although the *Storage* doesn't implicitly know any RTTI on the object it's working with, there are some safe assumptions you can make based on the *entityId* that comes with the object.

The *entityId* is composed of *kind* and *group*. The *kind* is supposed to represent what the persisted object actually is (e.g. a log message). These kinds are currently recognized by EverBEEN:

- **log** - log messages and host load monitoring
- **result** - stored task results
- **descriptor** - *task*/*context* configurations; used to store parameters with which a *task* or *context* was run
- **named-descriptor** - *task*/*context* configurations; user-stored configuration templates for *task* or *context* runs
- **evaluation** - output of evaluations performed on task results; these objects contain serialized BLOBs - see evaluations for more detail
- **outcome** - meta-information about the state and outcome of jobs in EverBEEN; these are used in automatic cleanup

The *group* is supposed to provide a more granular grouping of objects and depends entirely on the object's *kind*.

If you need more detail on objects that you can encounter, be sure to also read the ORM special, which denotes what EverBEEN classes can be expected where and what *entityIds* can carry user types.

2.6.3.1.6 The ORM special

2.6.3.2 MapStore

2.7 EverBEEN configuration

- one configuration file
- how to generate one easily
- distribution via URL

2.7.1 Configuration options

Follows detailed description of available configuration options of the EverBEEN framework. Default value for each configuration option is provided

2.7.1.1 Cluster Configuration

Cluster configuration manages how nodes will form a cluster and how the cluster will behave. The configuration is directly mapped to Hazelcast configuration. These options are applicable only to *DATA* nodes.

It is essential that cluster nodes use the same configuration for these options, otherwise they may not form a cluster.

`been.cluster.group=`***dev*** Group to which the nodes belong. Nodes with different group will not form a cluster.

`been.cluster.password=`***dev-pass*** Password for the group. If different password is used among nodes the will not for a cluster.

`been.cluster.join=`***multicast*** Manages how nodes form the cluster. Two values are possible:

- *multicast* - only `been.cluster.multicast.*` options will be used
- *tcp* - only `been.cluster.tcp.members` option will be used

`been.cluster.multicast.group=`***224.2.2.3*** Specifies multicast group to use

`been.cluster.multicast.port=`***54327*** Specifies multicast port to use

`been.cluster.tcp.members=`***localhost:5701*** Semicolon separated list of `[ip|host][:port]` nodes to connect to.

`been.cluster.port=`***5701*** Port on which the node will listen to.

`been.cluster.interfaces=` Semicolen separated list of interfaces Hazelcast should bind to, '*' wildcard can be use, e.g. *10.0.1.**

`been.cluster.preferIPv4Stack=`***true*** Whether to prefer IPv4 stack over IPv6

`been.cluster.backup.count=`*1* : How many backups should the cluster keep.

`been.cluster.logging=`*false* : Enables/Disables logging of Hazelcast messages. Note that if enabled messages will not appear among service logs.

`been.cluster.mapstore.use=`***true*** Wheather to use MapStore to persist cluster runtime information

been.cluster.mapstore.write.delay=0

`been.cluster.mapstore.factory=`***cz.cuni.mff.d3s.been.mapstore.mongodb.MongoMapStoreFactory*** Implementation of the MapStore, must be on the classpath when starting a node.

`been.cluster.socket.bind.any=`***true*** Whether to bind to local interfaces

2.7.1.2 Cluster Client Configuration

```
been.cluster.client.members=localhost:5701
been.cluster.client.timeout=120
```

2.7.1.3 Task Manager Configuration

```
been.cluster.resubmit.maximum-allowed=10
been.tm.scanner.delay=15
been.tm.scanner.period=30
```

2.7.1.4 Cluster Persistence Configuration

```
been.cluster.persistence.query-processing-timeout=5
been.cluster.persistence.query-timeout=10
```

2.7.1.5 Persistence Janitor Configuration

```
been.repository.janitor.finished-longevity=96
been.repository.janitor.failed-longevity=48
been.repository.janitor.cleanup-interval=10
```

2.7.1.6 Monitoring Configuration

```
been.monitoring.interval=5000
```

2.7.1.7 Host Runtime Configuration

```
hostruntime.tasks.max=15
hostruntime.tasks.memory.threshold=90
hostruntime.wrkdir.name=.HostRuntime
hostruntime.tasks.wrkdir.maxHistory=4
```

2.7.1.8 MapStore Configuration

```
been.cluster.mapstore.db.hostname=localhost
been.cluster.mapstore.db.username=null
been.cluster.mapstore.db.password=null
been.cluster.mapstore.db.dbname=BEEN
```

2.7.1.9 Mongo Storage Configuration

```
mongodb.password=null
mongodb.dbname=BEEN
mongodb.username=null
mongodb.hostname=localhost
```

2.7.1.10 File System Based Store Configuration

```
hostruntime.swcache.folder=.swcache
swrepository.persistence.folder=.swrepository
hostruntime.swcache.maxSize=1024
```

2.7.1.11 Software Repository Configuration

```
swrepository.port=8000
swrepository.serviceInfoDetectionPeriod=30
swrepository.serviceInfoTimeout=45
```

2.8 EverBEEN best practices

A.K.A. how to avoid major mishaps. Will be assembled incrementally.

Chapter 3

EverBEEN developer documentation

3.1 Design goals

- scalability
- – decentralize decision-making
- – use a non-SPOF comm protocol (replacing TM and RMI)
- genericity (in reaction to previous experience with RR)
- – provide extension points
- – resulting abstraction
- – dynamic implementation loading
- ease of use
- – heavy Maven integration into Task API (BPK plugin), subsequent Java task focus
- – node service composition over separate processes

3.2 Decision timeline

- Apache Maven (vs. Apache Ant)
- – a no-brainer - the project was so big it just needed modules
- – describe mavenization attempts
- SLF4J (vs. nothing, really)
- – another no-brainer, as it's the only way to enable implementation swapping
- Hazelcast (vs. JMS, JGroups) over a generic cluster API
- – JMS had SPOF and JGroups was too low-level
- Full overhaul
- – attempts on refactoring standing RMI code to use Hazelcast were catastrophic

- How generic cluster API got impossible
- – would disable access to virtually every high-level function of the current cluster tech
- Separate services from tasks
- – chicken/egg problem with software bundles
- HTTP protocol for SWRepo
- – likeliness of large file transport (obsolete decision from current POV)
- Jackson as a serialization library
- – Universal serialization was needed
- – JSON is more traffic-economic than XML
- – Jackson has got cool mapping features for objects
- 0MQ for inter-process communication
- – It will probably take MS to describe why 0MQ (srsly IDK :D)
- MongoDB as storage
- – Full abstraction of user types (to avoid byte-code trafficking nonsense)
- – We were using JSON anyway
- – MongoDB is web-scale :D (OK srsly not this one, but I had to put it here just for kicks)
- Tapestry as WI framework
- – Less code time than pure JSP and TP knew how to use it

3.3 EverBEEN architecture

- mostly true to the original BEEN
- – principal components stayed the same (SWRepository, RRepository, HostRuntimes)
- – Tasks are still separate processes
- – Put some cool pictures and thesis links here
- cluster adaptation
- – TaskManager ascended to hive-mind
- – SWRepository emits its location to the cluster
- – RRepository is not centralized anymore (can run on multiple nodes if need be)

3.4 EverBEEN services

3.4.1 Host Runtime

- how it only helps when you want to run tasks
- why does it make sense to run nodes without it

3.4.2 Task Manager

The Task Manager is at the heart of the EverBEEN framework, its responsibilities include:

- task scheduling
- context scheduling
- benchmark scheduling
- context state changes
- detection and correction of error states (benchmark failures, Host Runtimes failures, etc.)

Main characteristic:

- event-driven
- distributed
- redundant (in default configuration)

3.4.2.1 Distributed approach to scheduling

The most important characteristic of the Task Manger is that the computation is event-driven and distributed among the *DATA* nodes. The implication from such approach is that there is no central authority, bottleneck or single point of failure. If a data node disconnects (or crashes) its responsibilities,along with data, are transparently taken over by the rest of the cluster.

Distributed architecture is the major difference from previous versions of the BEEN framework.

3.4.2.2 Implementation

The implementation of the Task Manager is heavily dependant on Hazelcast distributed data structures and its semantics, especially the `com.hazelcast.core.IMap`.

3.4.2.3 Workflow

The basic event-based workflow

1. Receiving asynchronous Hazelcast event
2. Generating appropriate message describing the event
3. Generating appropriate action from the message
4. Executing the action

Handling of internal messages is also message-driven, based on the 0MQ library, somewhat resembling the Actor model. This has the advantage of separating logic of message receiving and handling. Internal messages are executed in one thread, which also removes the need for explicit locking and synchronization (which happens, but is not responsibility of the Task Manager developer).

3.4.2.4 Data ownership

An important notion to remember is that an instance of the Task Manager handles only entries which it owns, whenever possible (e.g. task entries). Ownership of data means that it is stored in local memory and the node is responsible for it. The design of Task Manager takes advantage of the locality and most operations are local with regard to data ownership. This is highly desirable for the Task Manger to scale.

3.4.2.5 Main distributed structures

- BEEN_MAP_TASKS - map containing runtime task information
- BEEN_MAP_TASK_CONTEXTS - map containing runtime context information
- BEEN_MAP_BENCHMARKS - map containing runtime context information

These distributed data structures are also backed by the MapStore (if enabled).

3.4.2.6 Task scheduling

The Task Manager is responsible for scheduling tasks - finding a Host Runtime on which the task can run. Description of possible restrictions can be found at [Host Runtime] .

A distributed query is used to find suitable Host Runtimes, spreading the load among `DATA` nodes.

An appropriate Host Runtime is also chosen based on Host Runtime utilization, less overloaded Host Runtimes are preferred. Among equal hosts a Host Runtime is chosen randomly.

The lifecycle of a task is commenced by inserting a `cz.cuni.mff.d3s.been.core.task.TaskEntry` into the task map with a random UUID as the key and in the SUBMITTED state . Inserting a new entry to the map causes an event which is handled by the owner of the key - the Task Manager responsible for the key. The event is converted to the `cz.cuni.mff.d3s.been.manager.msg.NewTaskMessage` and sent to the processing thread. The handling logic is separated in order not to block the Hazelcast service threads. In this regard handling of messages is serialized on the particular node. The message then generates `cz.cuni.mff.d3s.been.manager.action.ScheduleTaskAction` which is responsible for figuring out what to do. Several things might happen

- the task cannot be run because it's waiting on another task, the state is changed to WAITING
- the task cannot be run because there is no suitable Host Runtime for it, the state is changed to WAITING
- the task can be scheduled on a chosen Host Runtime, the state is changed to SCHEDULED and the runtime is notified.

If the task is scheduled, the chosen Host Runtime is responsible for the task until it finishes or fails.

WAITING tasks are still responsibility of the Task Manager which can try to reschedule when an event happen, e.g.:

- another tasks is removed from a Host Runtime
- a new Host Runtime is connected

3.4.2.7 Benchmark Scheduling

Benchmark tasks are scheduled the same way as other tasks. The main difference is that if a benchmark task fails (i.e. Host Runtime failure, but also programming error) the framework can re-schedule the task on a different Host Runtime.

A problem can arise from re-scheduling an incorrectly written benchmark which fails too often. There is a configuration option which controls how many re-submits to allow for a benchmark task.

Future implementation could deploy different heuristics to detect defective benchmark tasks, such as failure-rate.

3.4.2.8 Context Handling

Contexts are not scheduled as an entity on Host Runtimes as they are containers for related tasks. The Task Manager handles detection of contexts state changes. The state of a contexts is decided from the states of its tasks.

Task context states:

- WAITING - for future use
- RUNNING - contained tasks are running, scheduled or waiting to be scheduled
- FINISHED - all contained tasks finished without an error
- FAILED - at least one task from the context failed

Future improvements may include heuristics for scheduling contexts as an entity (i.e. detection that the context can not be scheduled at the moment, which is difficult because of the distributed nature of scheduling. Any information gathered might be obsolete by the time its read).

3.4.2.9 Handling exceptional events

The current Hazelcast implementation (as of version 2.6) has one limitation. When a key migrates the new owner does not receive any event (`com.hazelcast.partition.MigrationListener` is not much useful in this regard since it does not contain enough information). This might be a problem if e.g. a node crashes and an event of type "new task added" is lost. To mitigate the problem the Task Manager periodically scans (`cz.cuni.mff.d3s.been.manager.LocalKeyScanner`) its *local keys* looking for irregularities. If it finds one it creates a message to fix it.

There are several situations this might happen:

- Host Runtime failure
- key migration
- cluster restart

Note that this is a safe net - most of the time the framework will receive an event on which it can react appropriately (e.g. Host Runtime failed).

In the case of cluster restart there might be stale tasks which does not run anymore, but the state loaded from the MapStore is inconsistent. Such situation will be recognized and corrected by the scan.

3.4.2.10 Hazelcast events

These are main sources of cluter-wide events, received from Hazelcast:

- Task Events - `cz.cuni.mff.d3s.been.manager.LocalTaskListener`
- Host Runtime events - `cz.cuni.mff.d3s.been.manager.LocalRuntimeListener`
- Contexts events - `cz.cuni.mff.d3s.been.manager.LocalContextListener`

3.4.2.11 Task Manger messages

Main interface `cz.cuni.mff.d3s.been.manager.msg.TaskMessage`, messages are created through the `cz.cuni.mff.d3s.been.manager.msg.Messages` factory.

Overview of main messages:

- `AbortTaskMessage`
- `ScheduleTaskMessage`
- `CheckSchedulabilityMessage`
- `RunContextMessage`

Detailed description is part of the source code nad Javadoc.

3.4.2.12 Task Manager actions

Main interface `cz.cuni.mff.d3s.been.manager.action.TaskAction`, actions are created through the `cz.cuni.mff.d3s.been.manager.action.Action` factory.

Overview of actions

- `AbortTaskAction`
- `ScheduleTaskAction`
- `RunContextAction`
- `NullAction`

Detailed description is part of the source code nad Javadoc.

3.4.2.13 Locking

3.4.3 Software Repository

- functional necessities (availability from all nodes)
- why it uses HTTP and how (describe request format)

3.4.4 Object Repository

- queue drains
- async persist queue
- abstract query machinery (query queue handling, effective querying without user type knowledge)

3.4.5 Map Store

The MapStore allows the EverBEEN to persist runtime information, which can be restored after restart or crash of the framework.

3.4.5.1 Role of the MapStore

EverBEEN runtime information (such as tasks, contexts and benchmarks, etc.) are persisted through the MapStore. This adds overhead to working with the distributed objects, but allows restoring of the state after a cluster restart, providing an user with more concise experience.

The implementation is build atop of Hazelcast Map Store - mechanism for storing/loading of Hazelcast distributed objects to/from a persistence layer. The EverBEEN team implemented a mapping to the MongoDB.

The main advantage of using the MapStore is transparent and easy access to Hazelcast distributed structures with the ability to persist them - no explicit actions are needed.

3.4.5.2 Difference between the MapStore and the Object repository

Both mechanism are used to persist objects - the difference is in the type of objects being persisted. The Object repository stores user generated information, whereas the MapStore handles (mainly) BEEN runtime information - information essential to proper working of the framework.

The difference is also in level of transparency for users. Object persistence happens on behalf of an user explicit request, the MapStore works "behind the scene".

Even though both implementations currently us MongoDB, in future the team envisage implementations serving different needs (such as load balancing, persistence guarantees, data ownership, data access, etc.)

3.4.5.3 Extension point

Adapting the layer to different persistence layer (such as relational database) is relatively easy. By implementing the `com.hazelcast.core.MapStore` interface and specifying the implementation to use at runtime, an user of the framework has ability to change behaviour of the layer.

3.4.5.4 Configuration

The layer can be configured to accommodate different needs:

- specify connection options (hostname, user, etc.)
- enable/disable
- change implementation
- write-through and write-back modes

Detailed description of configuration can be found at Confiuration.

3.4.6 Web Interface

- why it's not actually a service (but more like a client)
- cluster client connection mechanism

3.5 Modular approach

- module overview, could use the info from mvn site
- – add a short description to each module to indicate what it does and how
- module interactions
- – this will need a lot of cool pictures
- extensions
- – what can be plugged out and what needs to be plugged back instead

3.6 Used technologies

The reasoning why we chose this or that tech is already done in the decision timeline, this should be more of a list of all the stuff we used and what we used it for (including the technologies that have no real repercussions on the project but we just needed them). Consider a table with a lot of fancy logos...

3.6.1 Hazelcast

3.6.2 0MQ

0MQ is a message passing library which can also act as a concurrency framework. It supports many advanced features. Best source to learn more about the library is the 0MQ Guide:

The EverBEEN team chose the library as the primary communication technology between a Host Runtime and its tasks, especially because of:

- focus on message passing
- multi-platform support
- ease-of-use compared to plain sockets
- extensive list of language bindings
- support for different message passing patters
- extensive documentation

We decided to use the Pure Java implementation of libzmq because of easier integration with the project without the need to either compile the C library for each supported platform or add external dependency on it.

As an experiment the Task Manager's internal communication has been implemented on top of the library as well using the inter-process communication protocol, somewhat resembling the Actor concurrency model.

3.6.3 Apache Maven

3.6.4 Apache Commons Exec

The previous version of the BEEN framework chose to implement executing of tasks using basic primitives found in the Java SE (which is known to be hard). The realization was buggy, confusing and fragile. Instead of re-inventing the wheel once more the team decided to use time-proven Apache Commins Exec library.

3.6.5 Apache Commons

- (virtually everything around IO and compression)

3.6.6 Apache HTTP Core/Components

- (HTTP server)

3.6.7 Bootstrap

- (cool skins, save time)

3.6.8 Hazelcast

3.6.9 0MQ

3.6.10 Jackson

- (JSON serialization for inter-process data transport and user type abstraction)

3.6.11 JAXB

- (serializable POJO generation)

3.6.12 Logback (logging impl)

3.6.13 MongoDB

- (store all kinds of stuff)

3.6.14 SLF4J

- (logging unification of custom logging implementations and standard libraries)

3.6.15 Tapestry

3.6.16 Other

- I definitely forgot about a half of these, feel free to complete this, just maintain the cool alphabetic ordering

3.7 Principal features

- scalability
- – you can actually add and remove nodes to improve shared memory and computational capacity
- – you can do it without messing everything up instantly
- – you can (presumably) have multiple persistence end-points over multiple Mongo shards
- user type transparency
- – you can put anything in a result and it gets serialized anyway
- – BEEN doesn't even care
- – If you change a result's version, BEEN doesn't explode in your face (although your task might, if you're not careful)
- extensibility - you can classpath-swap implementations of these:
- – logging
- – persistence
- – software cache
- – software store
- ease deployment
- – deployment is reasonably easy (once you configure the cluster, that is)
- – easy configuration (all in one place and you can generate the file)
- – remote configuration (load config from a URL)

- – the web interface manipulation is pretty straightforward
- easy measures
- – task implementation time significantly reduced
- – task contexts templating allows for quick customization
- – configurable benchmarks with a straightforward goal - task context creation
- straightforward in-task result manipulation
- – the user doesn't have to worry about serialization (if he uses Java) - he works with his own types
- – decent retrieval API that hides all the cluster hassle
- ...
- feel free to add more, there's never enough glory

3.8 Current limitations

- native task support got kind of crushed
- – we've done so much support for Java using Maven it's not exactly advantageous not to use it
- – although we maintain the theoretical use-case
- task dependencies (formerly intended) are not used
- – again, there is support for huge improvements, but not enough manpower
- command-line client disappeared
- – was there ever a use-case for batch runs, what with Java-code generator tasks?
- generic evaluators are gone
- – the price you pay for totally generic user-types and storage
- – plans for future improvements (user-aided type inference)
- dbase triggers are gone, too
- – they don't make much sense with the absence of generic evaluators
- – it's kind of unclear how this would work over the persistence layer abstraction
- ...
- probably a lot more here, too; needs reordering

3.9 In-code documentation

Probably just put a link here that opens a browser to the JavaDoc html