

Feasibility Study - DQGUI

Team Considerations

Work Policies

A classical division of labor into backend and frontend is hard to incorporate into this project as the backend already exists and our task is to create a graphical user interface for it.

Some work regarding the backend will be duo though as we have to implement two IQM4HD interfaces and a custom client tailored to our application. The workload for that however is not high enough to justify an entire backend team dedicated towards building the client and implementing the interfaces. We already have access to the tool that we build our application upon - so we shall use CI (Continuous Integration) for integrating new functionality into our application.

Our GUI prototype is still incomplete and will be updated at the beginning of the first sprint to fit the new backlog and Mr. Heines improvements that we received via mail. Once updated all GUI components that are not linked towards functionality yet will redirect to a message box informing the user that no functionality is to be expected here. Developers will then replace these placeholders with their implemented features so after every sprint we can ship a working application.

The following categories from our backlog will have to be satisfied:

1. File system based operations
2. Database based operations
3. IQM4HD based operations
4. UI Design and UI based operations

Breaking up our points into these categories will also have the side effect that specialists for each category will emerge while the project progresses.

To avoid pushing new, untested and work in progress integrations directly into our master branch we aim to create different branches within our repository. In theory this should also keep the master branch stable. There could be a development branch for example that contains the latest addition even though they might not work correctly yet. It is not expected that synchronization issues will appear as developers can see via Trello who is working on which backlog item. GUI skeleton implementations however will always be pushed into the latest master.

Division of Labor

It is planned to form 3 subteams to complete the tasks from the backlog:

Team US :	Kevin and Grant
Team GER1 :	Marc
Team GER2 :	Julian and Daniel

The division is based upon the physical proximity of the respective team members as well as experiences regarding JavaFX. Team US has some experience with JavaFX and can help each other with training. Team GER2 is similar: Daniel and Julian have already worked on a JavaFX project but still feel like beginners in that field. Marc has the most experience with JavaFX.

Technical Considerations

Work Environment / Required Software

IDE: Eclipse (default), IntelliJ (if preferred)

Technologies: JSON for local user settings and export of saved data, SQLite for saving report results

Platform: Java 8 (Oracle JDK) with JavaFX 8 and JavaFXs FXML technology

VCS: GitHub Private Repository

Database: SQLite for local storage operations

Further Tools: SceneBuilder, TeamSpeak 3, TeamViewer, Slack, Trello, StarUML, WhatsApp

Build tool: Gradle

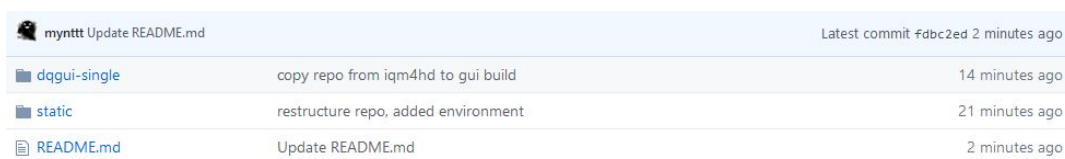
Usage of Java 8 or Java 9+ is still being actively discussed. It's currently defaulted to Java 8.

We will work on this project within Linux and Windows environments. Java and JavaFX is cross-platform so this should and will not cause any issues. JavaFXs cross-platform issues are mainly apparent when it comes to playing proprietary media codecs and complex GUI operations that only really occur in games.

Eclipse IDE, SceneBuilder and JavaFX + the required build tool (gradle) and libraries have been installed on our machines. Thus the development environment is set up. IQM4HD runs properly on our systems and we already tested it with the *NotNullCheckProbParkingPlate* action.

Gradle has also been tested successfully and all dependencies have been resolved. We have a simple HelloWorld application in our repo that we will transform into our application with the start of the first sprint. IQM4HD has been added to our project as a local jar file.

The git repository will be structured as following:



mynttt Update README.md		Latest commit fdbc2ed 2 minutes ago
dqgui-single	copy repo from iqm4hd to gui build	14 minutes ago
static	restructure repo, added environment	21 minutes ago
README.md	Update README.md	2 minutes ago

Fig. 1: Git Repo

static: This folder will contain documents, sample data and the IQM4HD codebase.

dqgui-single: This folder will be our working folder for this application. It is supported by gradle as the build tool. The .gitignore has been fed with settings for Eclipse and IntelliJ which means that the usage of both IDEs is possible and everyone should use what they feel the most comfortable with. The projects README.md shall be used as a landing point for developers and to share project related tips and tricks.

Languages / Libraries and Frameworks / APIs and their Rationales

We decided on using Java and JavaFX to build the tool and decided against Swing as JavaFX is more modern and suits the Zeitgeist better (more abstraction than Swing, more modern architecture). JavaFX features styling via CSS which is way faster, easier and abstracted from Swings "draw override" approach. JavaFX also allows us to build the GUI layout with a software called SceneBuilder, which is efficient and allows us to separate between GUI Design and Controller code. These SceneBuilder files are just customized XML and will then be parsed by JavaFXs FXML module. An advantage of this is that once the GUI design is relatively finalized these files can just sit within the repository and will not be touched by other developers resulting in less chaos on commits to the same file. Merely the Controllers will see some activity as they glue the UI elements together with the actual code. Using Java is an easily accomplished task since IQM4HD is written in Java as well.

We will interact with the underlying IQM4HD software via the interfaces supplied through the API package. `RuleService` and `DatabaseService` will be implemented by us and then used by a customized client utilizing the functions defined in the `Iqm4hdAPI` class. We will orientate ourselves by overlooking the usage of function calls supplied via the `SampleClient`. JDBC and the Java implementation of the MongoDB Client shall be used to interact with the databases.

[RichTextFX](#) (additional JavaFX components library) is our choice when it comes to the editing features of our project. Reinventing the wheel is one of the worst traits in software development unless explicitly used for learning purposes, this projects goal however is to write a development environment for the IQM4HD project and not to build a complex rich text panel. It features syntax highlighting and a way to implement keyword suggestions while typing. Suggestions of variables is too complex and will not be implemented as it would require to parse the language instead of just checking the defined keywords.

[controlsfx](#) (additional JavaFX components library) will help us as well to extend the range of JavaFXs existing GUI elements to create a swift and user-friendly experience.

Gradle shall be the build tool of choice to manage the build process and dependencies as its a powerful and user friendly tool and nobody wants to see giant XML files anymore (looking at the pom.xml).

Architecture

DQGUI will be realized as a fat client since we implement the `DatabaseService` interface and are thus responsible to access the database directly and execute the queries + transform the queried data to `DatabaseEntryIterator` objects that IQM4HD can work with.

The following architecture diagram will elucidate on the entire infrastructure and the protocols used to access the different components.

DQGUI (our application) will stay in JVM space together with the IQM4HD project. We will interact with IQM4HD via the API and its implementable interfaces. The `Rserve` instance that IQM4HD requires for some calculations will be started and stopped via the already supplied start shell script. Relational databases will be accessed through JDBC and NoSQL/Document based databases through their respective client implementation.

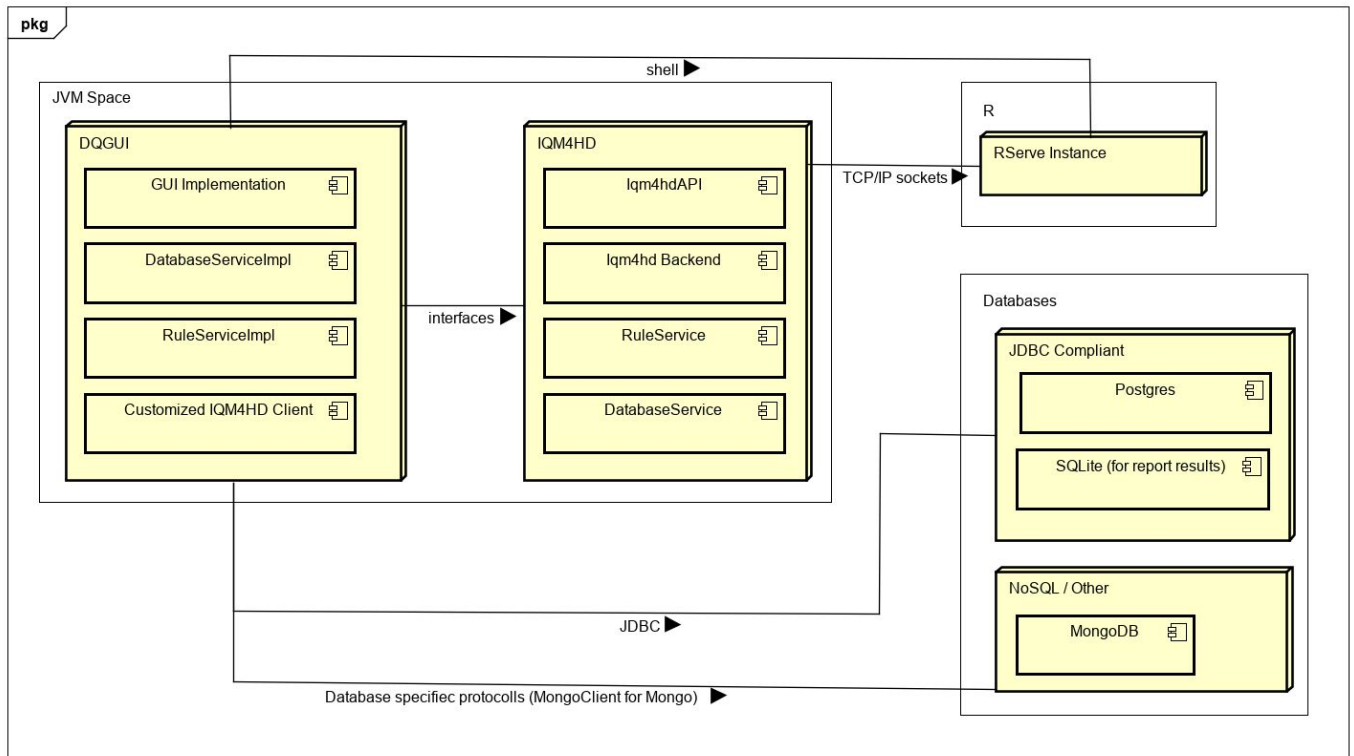


Fig. 2: Architecture of our Application

User Experience

In order to make the future usage of the IDE as pleasant as possible for its users we focus on creating an organized and simplistic interface which allows to conveniently and intuitively manage database environments / connections, DQ rules and their execution. Our focus hereby lies on a clear overview and simple menu navigation - each operation will be easily accessible so that the user can find it as quickly as possible.

The text editor for the rules will be tabbed so multiple instances of rules can be opened and switched through easily. Executed actions will run in the background so long tasks will not end up blocking the user having to wait for the report result. This will also allow for multiple tasks as long as their number stays under a customizable threshold so they do not end up starving each other of processing power.

These tasks can be viewed from a separate window. A use case of this behavior is on a multi-monitor system where that window can be placed on the second monitor while the user edits rules in the first window on the primary monitor.

Optional notifications will be implemented as well notifying a user once a report has finished. We do not plan to do some extensive styling on the application. It will be kept simple and easy on the eyes.

The next page contains a simple GUI prototype that has been created for the feasibility study in order to show the direction the application moves into.

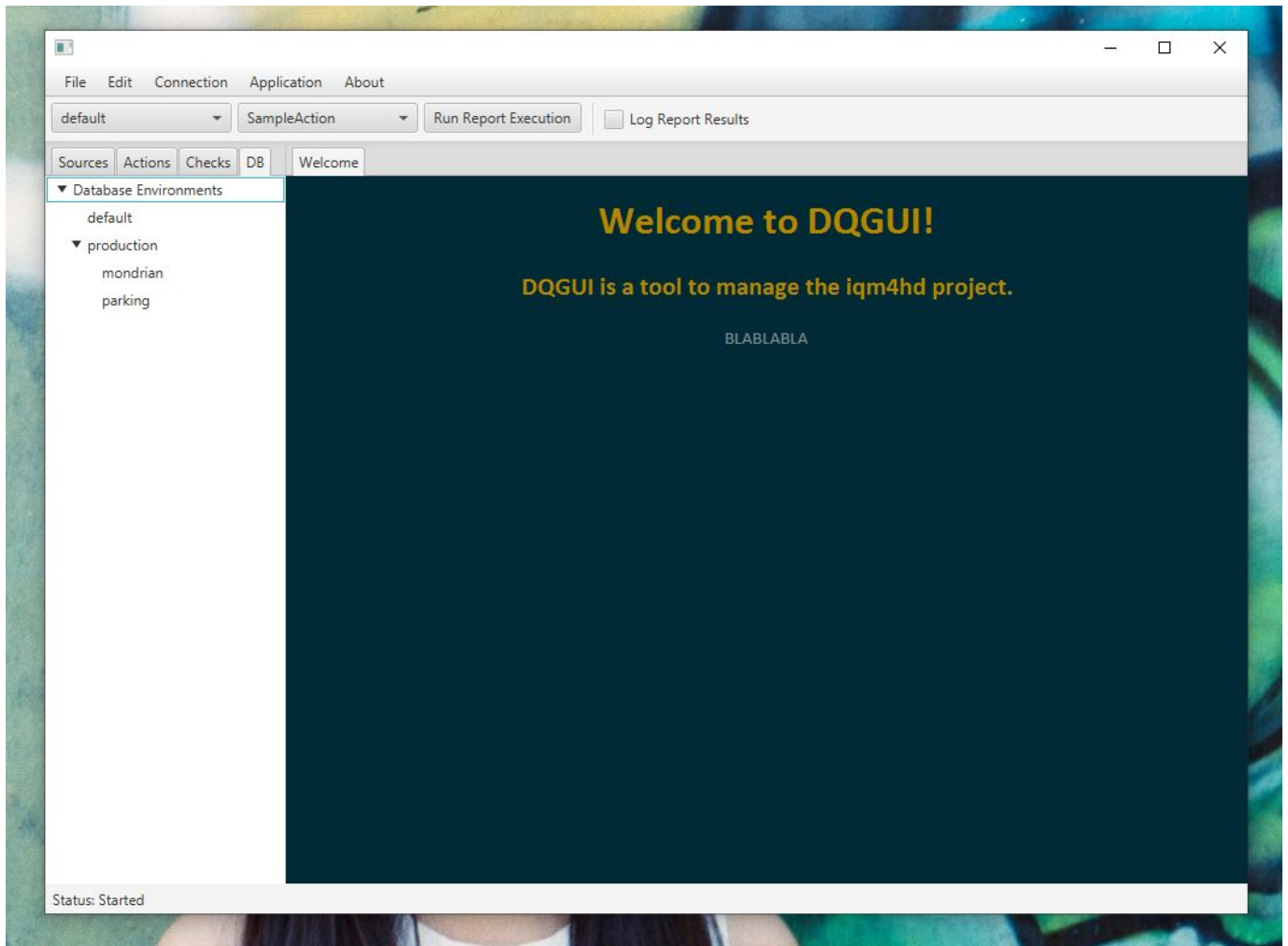


Fig. 3: GUI Prototype of our Application

Database connections shall be able to be created and tested with a wizard that will guide a user through the process of submitting all the data needed to successfully fetch data for IQM4HD. This wizard will be framework-like so it is possible to later add custom pages for previously unsupported database types.

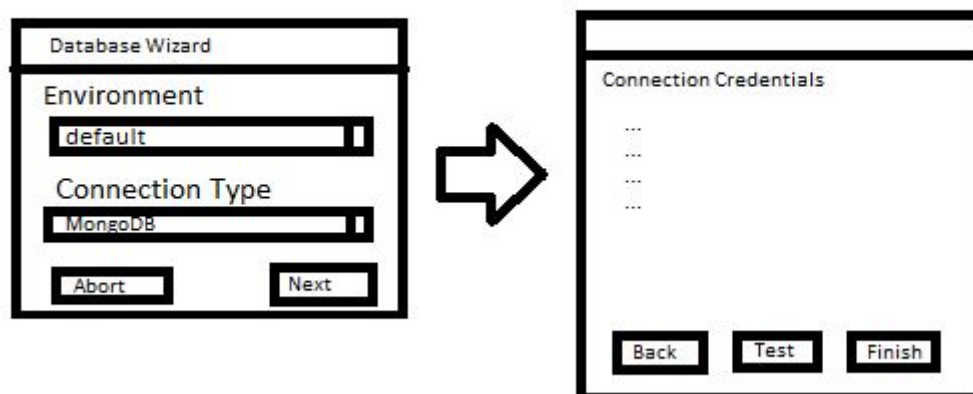


Fig. 4: Sketchup of the wizard helping to add database connections

Data storage / deployment considerations

Users preferences, database environments and their contained database connections will be stored as JSON in a folder on the local filesystem. The log file that we use for debugging will be written in the same folder as the application. IQM4HD reports shall be written into a local SQLite database placed next to the database connection files.

Solutions to Non-Trivial Problems

#1: Profiling

Profiling is not an implementation problem but more of an issue with our understanding of it. We do not know what makes a good and a bad profiling service and best practices regarding it. As of Mr. Heines mail this will be discussed separately and later on. Profiling will be added once all basic features of our application work as it would not make sense to implement profiling on a system that is not even able to store historic report results. The *NotNullCheckProbParkingPlate* has run successfully so we do not know yet how to model our SQLite database schema but we have looked into the `ExecutionIssue` class and it should be possible to map this to a relational database system. Using the file system would lose us the ability to easily query and filter reports and a NoSQL database is just overkill for such a small use case.

#2: Management of Database Connections / Database Environments and interaction with IQM4HD in a way that allows easy addition of previously unsupported databases

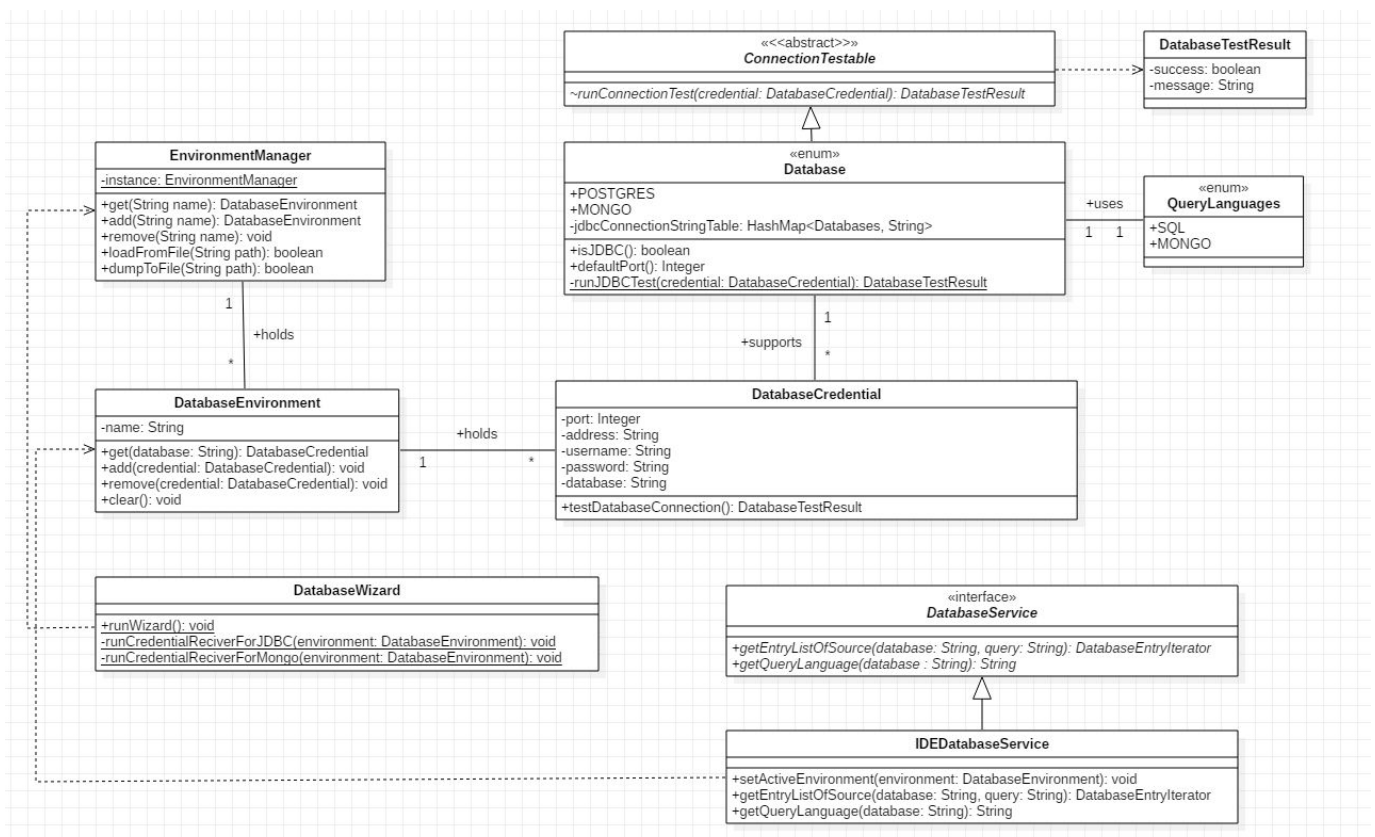


Fig. 5: UML Diagram for a possible solution to #2

The solution displayed above should allow us to interact between IQM4HD and our application in a comfortable way. It is fully extensible and new database types can be added without rewriting parts of the previous implementation. In this case we differ between JDBC supported systems as they can be treated as the same with different connection strings and default ports and other databases like MongoDB and other NoSQL database systems.

Supporting a new database would go as following:

1. Add an Enum constant to `QueryLanguages` in case it uses a different language that IQM4HD needs to know.
2. Create an Enum constant in `Database` and supply its constructor with a boolean indicating if it is a JDBC supported database, its default port and the language enum constant.
 - a. In case it is a JDBC supported database also add a formattable String to the internal HashMap that will help creating a connection string.
3. Implement a test for the database connection via the `ConnectionTestable` interface.
 - a. In case it is a JDBC supported database just redirect to the private static `runJDBCTest` method.
4. In case the database needs more information than Port, Username, Password, Address and Database extend the `DatabaseCredential` class.
5. Create your own wizard to collect credentials. This part will simply be integrated in the existing wizard when your new database is selected.
6. In case it is not a JDBC supported database you will also have to implement a conversion between your query results and the `DatabaseEntryIterator` in the `IDEDatabaseService`.

Test Plan / Basic Test Concept

It is planned to test per module: CREATION, MODIFICATION and DELETION of data on the used databases (MongoDB, Postgres). The types of tests will evolve with the project. When testing the individual modules, it will be important to understand the following interactions of the functionalities:

- module to itself
- module to other modules
- module to GUI
- module to backend
- have the other modules remained unaffected?

JUnit is the test suite within our repository. It can be voluntarily used by the team members for small independant GUI functionalities and will definitely be in use for more complicated failure prone operations that are easily affected by active development. We will however not strive for a 100% test coverage of the project.

Sprint Planning

Initial Burndown Chart

	Sprint	Due date	Ideal	Actual
Semester 1:	Start	15.10.2019	100	100
	1	04.11.2019	86	
	2	25.11.2019	72	
	3	09.12.2019	58	
Semester 2:	4	TBA	44	
	5	TBA	30	
	6	TBA	16	
	7	TBA	0	

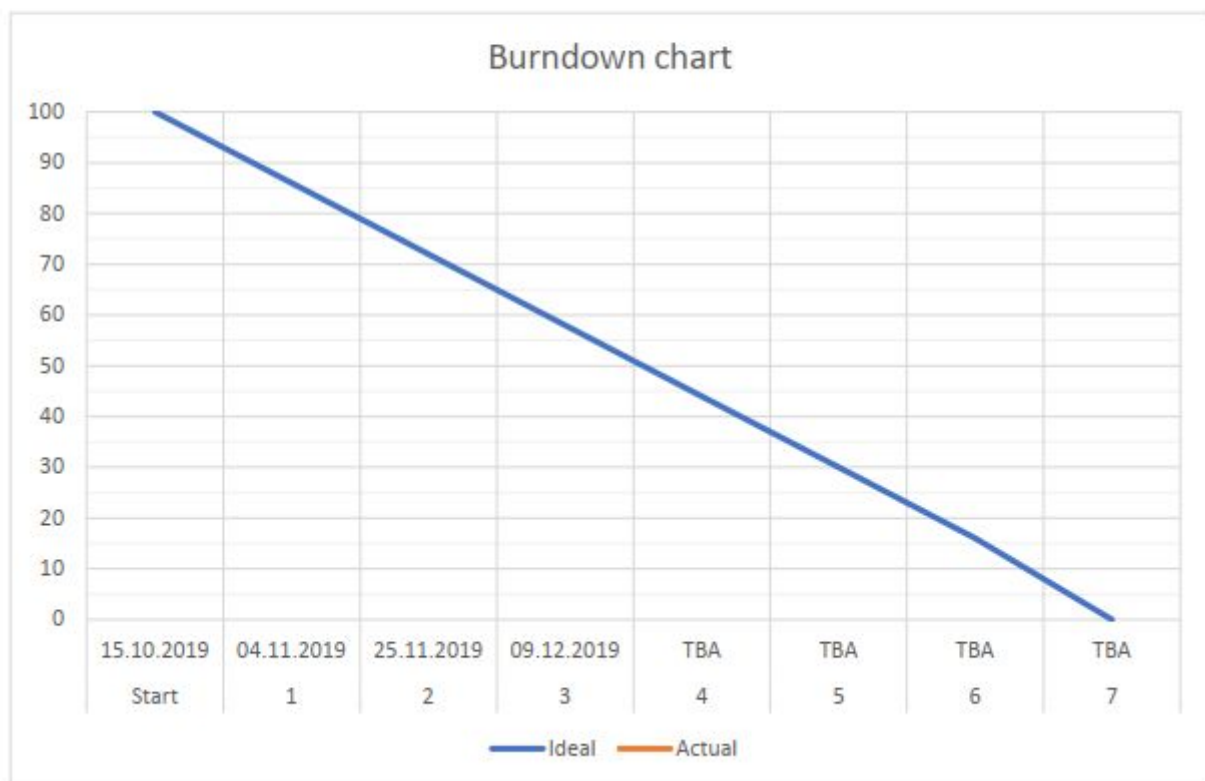


Fig. 6: Burndown Chart

Since we cannot foresee specific priorities or workload at this point, we have equally weighted the bullet points of our backlog. This may and probably will change as our project progresses. In order to estimate the workload for each sprint we used following calculations:

35 bullet points (without sub-items) total in the backlog.

$100\% / 35 = \sim 2,85\%$ per bulletpoint

35 b.p. / 7 sprints = 5 bullet points per sprint

$5 * \sim 2,85\% = \sim 14,25\%$ workload per sprint

Furthermore we will set up a Trello board where we will categorize all of our backlog items. This shall act as our digital backlog.

Four categories will be introduced:

- [Open](#)
- [WIP \(work in progress\)](#)
- [Postponed \(due to problems or the like\)](#)
- [Completed](#)

This approach is helpful to keep track of the sprint and enables us to prioritize issues early on, distribute them between our team and opens up the possibility of adding further items at any given time.

Each member will be able to pick a task and mark it appropriately according to its current status, thus allowing us to work in an agile manner. Flexibility is the key here.

Proposed sprint plan:

Sprint plans for the sprints id > 1 will be created in the next sprint report once the sprint has been completed as it is impossible to foresee how the backlogs progress will turn out from this position.

*The sprints will be planned extremely narrowly so it can be entirely possible that not all of sprints proposed items will be finished. This is to prevent a sprint from having too **less** items in it.*

This view of the sprint is simplified for the sake of understandability. The real backlog with all the tiny sub-points will be added to Trello. Low priority items will move up if not completed for the next sprint.

Priority	Item
HIGH	Implementation of proposed database connection management “backend” system (see UML).
HIGH	Ability to manage rules on the filesystems (RuleService Interface implementation)
MEDIUM	Ability to manage rules within the gui. (Repo Wizard + creation)
MEDIUM	Ability to manage connections within the gui. (Edit/Wizard/Test)
MEDIUM	GUI Skeleton finished regarding elementary operations
MEDIUM	Basic integration of the text editor functionality
LOW	API to manage user settings and user stored values like DB Connections / Environments / ...
LOW	Basic IQM4HD test run

Milestones

1. GUI is built and has access to IQM4HD via implemented interfaces / client.
2. Storage of historic data and a profiling service using that data.
3. All items are implemented and tested successfully.

Deliverables

Each Sprint will require the following deliverables:

- Sprint report and presentation
- Updated burndown chart
- Potentially shippable product according to current state of development
- Peer evaluation report