Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Stratégiai játék futtató webes rendszer

DIPLOMATERV

Készítette	*Konzulens*
Márkus Krisztián	dr. Dudás Ákos

December 6, 2018

Contents

HALLGATÓI NYILATKOZAT

Kivonat

Jelen projekt célja egy rugalmas, nyitott környezet kifejlesztése, mely lehetővé teszi különböző megbízhatatlan forrásokból származó, egymással összekapcsolt programok JVM–alapú biztonságos futtatását. Ismeretlen, ellenőrizetlen kód végrehajtása természetesen komoly biztonsági feladat, mely részletes tervezést igényel annak érdekében, hogy a rendszer hatásosan védekezni tudjon a lehetséges támadások és sebezhetőség kihasználások ellen. Dolgozatomban bemutatom egy ilyen rendszer részletes tervezési- és működési folyamatát, valamint az azt kiegészítő kezelő modulokét. A haszált védelmi mechanizmusok nagyrészt a java platform biztonsági lehetőségein alapulnak, kiegészítve azokat egyéni megoldásokkal.

A keretrendszer továbbá leírja a modellt, melyre alapozva intuitív módon készíthetőek idiomatikus kliensek a rendszerhez, a gyakori programnyelvek — és leginkább a gazda java nyelv — stílusának megfelelően. Egy példa projekt keretein belül bemutatok egy tipikus felhasználási módot, végigjárva annak a fejlesztés során felmerülő tervezési döntéseit és megvalósítási vonatkozásait.

Továbbá a projekt megvalósít egy, az imént említett biztosnágos futtatókörnyezetet felhasználó és működtető rendszert, mely kezelési- és információ megjelenítési lehetőségekkel látja el felhasználóit.

Abstract

This goal of this project is to develop a flexible, open runime that allows for the safe execution of distinct, connected programs from untrusted sources on the JVM. Executing unsafe, unverified code is, of course, a major security concern, which must be dealt with by carefully designing the system to guard effectively against possible attacks and exploits. In this thesis, I will describe the in-depth construction and working of such a system, and its supplementary management modules. Used security measures are based largely on the java platform's security capabilities, extended with custom solutions.

The framework also defines the model, based on which client programs may be created for it in an idiomatic, intuitive manner; fitting the control structure of most commonly used programming languages — especially the host java language. I will demonstrate a typical usage of the runtime via an example project, describing in detail the design decisions and implementational concerns that may arise during development.

Finally, the project realizes a control system that utilizes the aforementioned secure framework, and provides management- and information display capabilities toward end-users.

Chapter 1

Introduction

This project is an attempt to solve problems that commonly surface when learning programming languages, or when programming is pactised as a leisure activity.

1.1   Learning programming

When learning a programming language or programming itself — either via organized education or self-teaching — it is necessary to occasionally test one's acquired theoretical knowledge in action, and measure their progression. This may be done for many reasons, including general practicing, self-evaluation, or grading students of a course. However, in practice, testing may be difficult in all of these situations due to different inconveniences.

An autodidact may find it difficult to come up with problems matching their level of skills, and even evaluating their solution isn't as easy as it may seem. A lot of computational tasks require a verified, correctly working second program to compare results with, or a list of expected results for specific inputs, both of which require a second party to create them first. It is also easy to fall into the trap of believing in the correctness of your solution based on a limited set of test cases you have created on your own.

As for communal education — formal and non-formal alike —, while the educator may be proficient in coming up with good exercises, it is usually very challenging for them to evaluate and/or grade submitted works due to the number of students. This problem is present not just in schools and universities, but boot camps, online courses, and company training as well.

Thus, some sort of automated testing system becomes necessary. Many institutions, especially universities and online learning platforms already provide solutions for this problem, but these are often insecure, ad hoc in nature, and require the superfluous development of a relatively large, mostly analogous

overhead by all participants.

1.2 Recreational programming

The other important target subject of the project is the recreational programming scene. By its nature, this area can strongly overlap with the aforementioned educational programming — as free time programming can be viewed as a way of skill development —, but it also covers purely leisure activities. Several communities exists in the programming word that publish programming challenges of some sort, many of them mainly design games for which coders can write bots, either competing with each other or simply solving puzzles. But in general, open solutions are not commonplace, and existing frameworks and websites usually don't provide the ability to publish custom games.

1.3 Goal of the project

The goal of the project is to provide a safe runtime for untrusted programs that interact with each other on top of the JVM platform. This framework shall also allow the development of games/challenges and their bots in an easy to integrate and environment familiar way. These games should be able to effortlessly provide meaningful results, which can then be processed by a user of the runtime.

The framework must allow the development of both single-, and two-player games, so that it can satisfy the needs of both educational-, and recreational programming.

The runtime must be integrateable into different projects, and configured for diverse environments, but should also contain a default management interface for game and bot control.

The outcome of the development should be a server-side backend application and its frontend counterpart. They are to allow the uploading of games, and bots for these games. Through them users can browse games, bots, and their own bots' results.

The functional core of the project is the secure runtime, which must enable the safe execution of untrusted programs, provide communication between them, handle the control flow, and report the result of each run.

The backend shall use this framework as a library to execute matches and challenges between its managed bots. It handles game coordination, report generation, information provision for the frontend, and data persistence.

The frontend is a simple web-app, that — according to the end-users' need — manages games and bots, and shows valueable information to the players about the games played.

Chapter 2

Game Model

In order to be operatable by the runtime, the games and their respective bots must match certain type level requirements and abide some contracts.

2.1   Game API

The game api is a library which contains everything that is required or may help to develop bots for the game. This includes the bot interface, all types used in communicating between the engine and a bot, and various helpers and utilities to ease bot development.

Bot interface

The bot interface is a simple java interface that extends the `BotInterface` marker interface found in the universal game runtime api, and defines the operations a bot must support. Through its methods will the engine communicate with a bot, which must implement this interface. In order to ensure better compatibility with different programming paradigms (such as functional programming), and generally make bot development easier, it is advised to operate bots as stateless entities, and pass the current state of the game to them in these methods.

API types

These domain classes and interfaces are to represent the internal model of the game according to its logic. They can be passed to the bot through the bot interface's methods, or be returned from them. To allow the engine and the bots to run on separate virtual machines, these must be serializable. Due to their vm separation, when passing one of these objects to a bot, a replica will be created, and therefore changes made by the engine or bot to their objects will not be visible to the other's

instances. If changes are to be made on such parameters by the bot, then the changed value should be returned. It is generally a good idea to define these types as simple immutable value classes to help reduce potential bugs created as a result of improper understanding of the object sharing mechanism.

Utilities

As a game is generally created to be played by many bots, it is very helpful of the game developer to take their time and create well-usable utilities to help bot development efforts. These helpers usually operate on the domain classes and provide functionality that most bot developers would have to write on their own, because of their frequent necessity.

2.2 Game engine

The game engine is a library which contains the implementation of the game logic (the engine itself) and optionally other relevant classes and interfaces. There are three requirements for being a valid game engine:

- Be a concrete class that implements the generic `GameEngine` interface from the engine runtime api, with the proper bot interface defined in the game api being its type parameter.

- Have a public constructor that takes either one or two instances of the previously mentioned bot interface as parameters, based on whether the game is a challenge or a match.

- Have the realised `playGame` method return an instance of `ChallengeResult` or `MatchResult` based on whether the game is a challenge or a match.

However, in the engine runtime api two abstract classes are also defined to help (type-safe) game engine development, namely `ChallengeGameEngine` and `DuelGameEngine`.

A game engine is able to log messages during a game for both itself and the playing bot(s) as well. It can do so, by acquiring a `GameEngineLogger` instance, either using the `EngineLoggerFactory`'s static `getLogger` method, or if already extending one of the aforementioned abstract helpers, through its `logger` protected member.

ChallengeGameEngine

`ChallengeGameEngine` is an abstract class that ensures a proper return type (`ChallengeResult`), while also providing helper methods for easier result creation

and logging.

DuelGameEngine

Much like `ChallengeGameEngine`, `DuelGameEngine` is also an abstract helper with proper result safety, logging, and result creation helpers; however, it also implements a basic turn-based logic. Its extender only has to define the mechanism for game initialization and for a single turn of the game, keeping track of the active player is being taken care of.

Other classes

The library can contain any other classes that help the engine work, such as inner types, or utilities that it wishes not to share with the bots.

2.3 Bot

Bots are created as libraries that contain a concrete implementation of the specific game's bot interface, and have a public, no-argument constructor. The bot library may contain other helpers and utilities as well.

Much like game engines, bots can log messages during matches; however, they can only send these to themselves (so only their creator can view them). To create log entries with a bot, a `GameBotLogger` instance is needed. A bot can access one using the `BotLoggerFactory`'s static `getLogger` method.
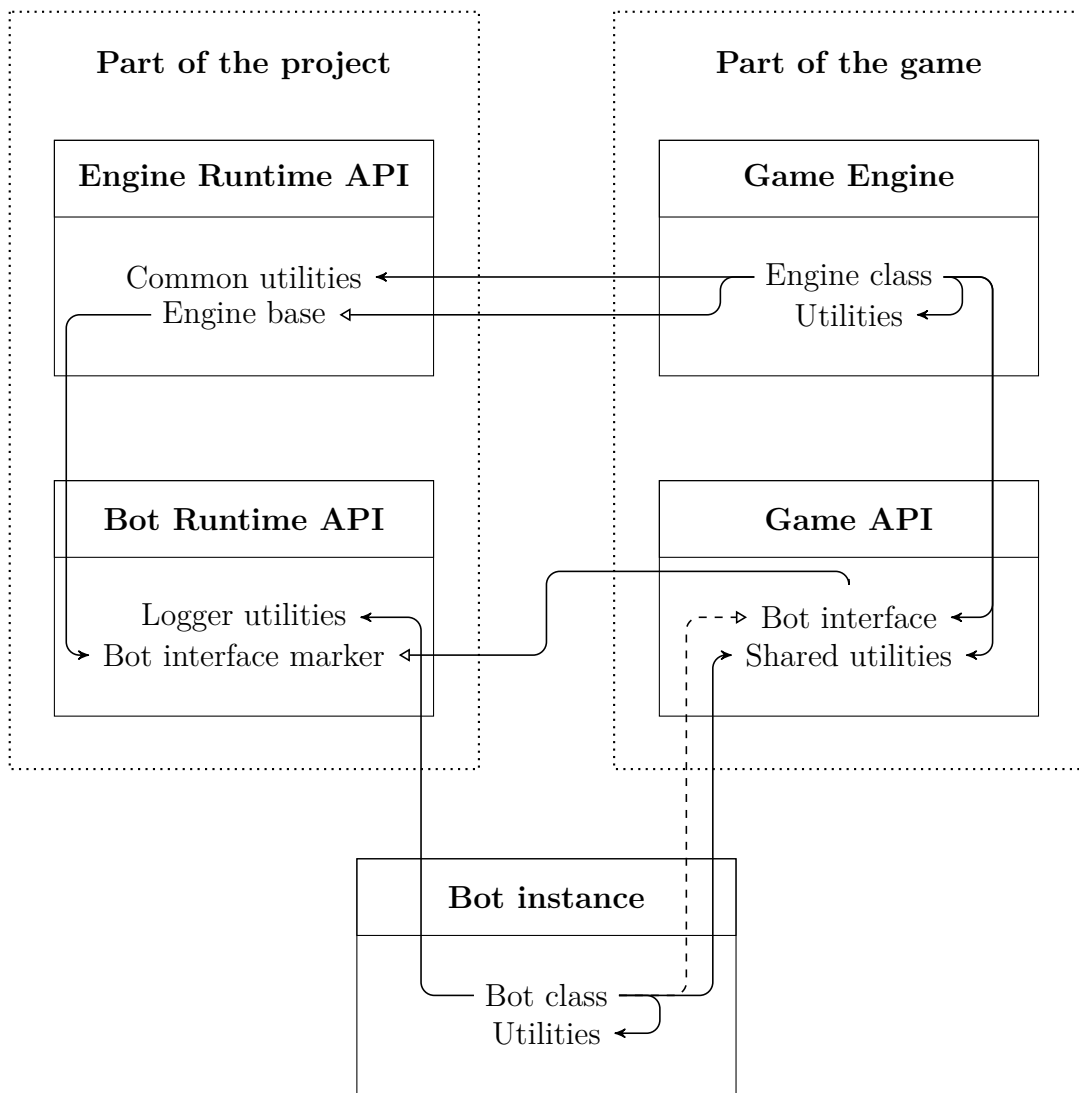
Game model components

Chapter 3

Environment

The backend system is written completely in the kotlin programming language with the exception of the game- and engine runtime APIs, which are written in java to allow game and bot development to happen also in java without unnecessary dependencies.

The system targets the Java Virtual Machine, and is therefore reliant on its features, such as OS independence and security. Despite being a primarily server side application, the system is not powered by Java Enterprise Edition, but the more accessible Java Standard Edition (version 9).

3.1 Database

The application can be configured to use any JDBC capable datasource, and as a default implementation, an in-process HSQLDB running in file mode is used as the database. For querying, standard SQL syntax is used; for the initialization of the database, custom create scripts may be necessary to be implemented if the selected database does not support the same types and options as HSQLDB.

3.2 Web backend

The supporting web application is based on the standard Java SE ecosystem as a standalone kotlin application. It uses a RESTful API between the JAX-RS defined backend and the SPA-based frontend.

3.3 Java Security Architecture

Since its early versions, the java platform has had built-in security measures to protect valuable resources, while also providing the ability to execute untrusted code. The original purpose of this was to allow safe usage of unsafe web applets,

which have since largely disappeared due to technological change. However, the java security model has remained in the language specification nonetheless, and through two main iterations have reached its current form in Java 2 (formerly known as Java 1.2).

The current model [8] allows the JVM to separately manage different code bases, all with their own unique set of permissions. These code units are loaded by one of possibly multiple classloaders, either from a local source, from a remote path, or whatever custom logic the specific classloader supports. It is also this classloader that assigns an appropriate identifier to these code collectives based on which permissions can be granted to them.

3.3.1   Access control

Access control is done by a security manager, which can be installed either programmatically or via a JVM startup argument. It is the job of this security manager instance then to check whether an action shall be allowed to happen in a given execution context. These permission checks are initiated by low level java APIs, such as I/O access methods or threading primitives; and include a vast number of different actions, such as specific file operations, configuration access, networking procedures, or replacing the active security settings (including the security manager itself). While the security manager can be customized to deal with these requests in an arbitrary way programmatically, it is usually easier to configure separate policy settings to the different code sources, and allow the default security manager to use these settings for access authorization.

The standard way of permission control is based on code sources and their respective set of permissions [5]. At the points of an access check (e.g. reading from a file or creating a network socket), the default security manager delegates the request to the built-in `AccessController`. That in turn first creates a view of the current security context represented by a call stack, where each frame is assigned the set of permissions of its code (based on the code source the called method is located in). Then the controller iterates through the stack and only allows the request to go through, if all frames would allow the request in isolation.

This behavior can be modified using the controller's set of `doPrivileged` methods that allows code to be executed if it would be allowed to run on its on right, effectively cutting the stack off under the specific method. This allows trusted code to provide operations on guarded resources to their untrusted counterparts. An example this might be useful for, is when a file should be kept secret from an untrusted method, however some properties (e.g. number of lines) of that same file should be allowed to be accessed. In this example a trusted method could be written, which reads the

file, and only returns the required property (line count). If this line counter method uses the access controller's privileged execution, then it would be allowed to be used from the original, untrusted code block without security exceptions being thrown, or the file becoming fully accessible from there.

Trusted.kt

```kotlin
object Trusted {
    fun guardedLineCount(): Int {
        return AccessController.doPrivileged {
            File("guarded.txt").lines().count()
        }
    }
}
```

Untrusted.kt

```kotlin
object Untrusted {
    fun processCount() {
        val lineCount = Trusted.guardedLineCount()
        println("Guarded file has $lineCount lines!")
    }
}
```

3.3.2 Security policy

Assigning permissions to the different code sources can be done programmatically at load time using the classloader, or in a declarative way via a policy file [9]. A policy file contains a set of rules that define a list of permissions to be granted to code sources satisfying a set of selectors. These selectors can filter based on the identifier of the code source; whether it was cryptographically signed by someone; and if it was, by whom was it signed by; or what organization (`Principal`) does it belong to.

The enforced policy file can be set much like the security manager, either programmatically or through a JVM argument.

3.3.3 Limitations

While the JVM has good facilities to protect resources, and guard certain operations, it is not equipped to resolve all possible exploits which may be caused by untrusted code. Due to java's shared-heap based memory model, the JVM is

inherently incapable of providing thread-based protection against deliberate or accidental causes of out of memory errors.

Furthermore, since errors — opposed to exceptions — are not necessarily catchable (as they usually signal unrecoverable mishaps), it would not be possible to reliably log which actor caused a crash. That would be particularly unfortunate in the case of matches, where one bot could cause any game to result in an error and the sinner would not be detectable.

These limitations make it necessary to take the actors — and their handlers — out of the common java runtime, and place them into separate JVMs. This method can defend against out-of-memory style-, and other unavoidable errors. In this case, the virtual machines are simply allowed to crash as a result of the errors, because their failure leaves the rest of the system unaffected. If such crash happens, the runtime will be notified through the cessation of communication with it, and it can react in an appropriate manner.

However, limiting the instances memory capacity and CPU time is still crucial, as burning through any of them may result in errors being thrown in another JVM instance. Fortunately, these can be managed either through JVM-internal settings (maxium memory limit through the `-XmX` flag) or through OS utilities (CPU time limiting using the process' 'nice' value, or through cgroups). These measures are set up by the starting scripts initializing the handlers.

Chapter 4

Runtime Architecture

Running games require the execution of untrusted code — in form of both the game engine and the bots — in a safe manner, which is complex task.

The limitations of java's security error handling make the JVM unable to provide a platform where a whole game — all of the actors and client runtimes — can be run as a single process. Therefore separate java processes are necessary to achieve a safe runtime environment.

4.1   Runtime model

The runtime model consists of two main types of entities: the runtime handler that acts as the controller of a played game, and actor client runtimes that manage the individual actor participants. They communicate with each other over standard TCP sockets organized into a star topology with the handler at its center.

4.2   Runtime handler

The runtime handler is the central managing entity of a given game, it initiates most operations through controlling the client runtimes, and processes their responses. Its job is mainly to:

- Provide the necessary binaries to the client runtimes, including the game api, engine, or bot libraries.

- Control the game flow based on a general ask–answer turn based system.

- Process log messages from the actors.

- Process error reports from the actor runtimes.

- Report the result of the game, whether the engine reported a 'normal' finish or some error occurred.

The handler is a library which can be used from other JVM-based programs, as it exposes functions to run games using provided actors.

4.3 Actor client runtimes

Actor client runtimes are executable java processes that are responsible for communicating with the runtime handler, providing a secure environment in which the given actor can be used, processing requests from the runtime handler, driving the actor in response to the received control messages, and reporting any problems that might occur during the game. They come in two forms, based on whether they manage a game engine or a bot, as the two tasks require different messages and control structures.

The runtimes are isolated programs, that only need to know the way of connecting to the handler to run, as they actually receive the actor to be managed and other libraries through standard messages after startup. This allows a flexible separation of the runtimes as long as a connection can be provided between them. If, for example, extra security or scalability is warranted, the design would allow running these clients on different physical- or virtual machines, or other types of containers.

All actor client runtimes provide logging facilities to their respective actors, which can be used to save messages to be viewed by the creator of the actor after the game has ended.

Beside direct client–handler messaging — such as logging or error reporting — , the runtime model also needs to manage inter-actor communication to allow an engine to interact with the bots. This is manifested as one directional method calls as — due to the logic of the system — only the engines have access to their bots and not in reverse. As these calls should happen between objects running on different java virtual machines, remote method invocation is necessary.

Engine client runtime

On top of the client runtimes' common tasks, an engine client runtime must also provide game bot proxies to the engine it manages. Through these stubs can the environment then convert the specific call requests to standard messages, which can in turn be understood, and routed by the runtime handler. After such call is successfully evaluated by the desired bot, the response is processed by the proxy and returned transparently to the engine.

Bot client runtime

As an engine runtime handles the stub side of the actor–bot remote communication, a bot runtime must provide a way of processing incoming invocation requests, and forward them to the bot it drives. After the target method has been executed, and some kind of result is produced, it is this skeleton's task to send the outcome back through the runtime handler.

4.4 Communication model

Communication between the various components of the runtime model are driven by discrete messages, whose content and sequence follow a well-defined protocol enforced by the runtime handler and clients. These messages either provide standalone information, or form a more complex transaction. Any message that breaks the expected flow of communication will trigger errors in the clients, but it is usually the handler that interrupts the game, should any discrepancy occur.

Some messages can be closely tied to actor operations, while others help to operate the runtime model.

Messages carry three types of information: their type, the type's optional parameters, and an optional payload as well. The type is to determine the objective of the message, while the type parameters and payload are to carry variable information of the instance of the message. The latter two's separation is an implementational decision that ensures safe message handling. As such, from the communication model's perspective, they can be viewed as a combined payload, and in the remainder of this chapter will be referred to as simlpy 'payload' or 'data'.

Simple messages

Simple messages carry pieces of information that are meaningful on their own, either from an actor client to the handler or in the opposite direction. They do not represent an answer for another message, and no such answer is expected to be replied to them.
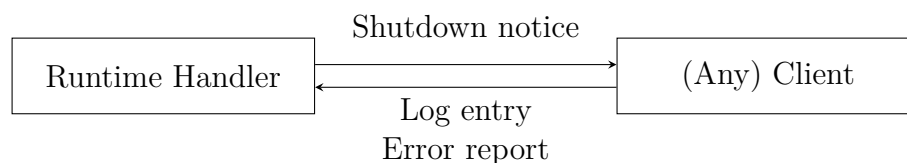
Possible simple messages between handler and bot- or engine client
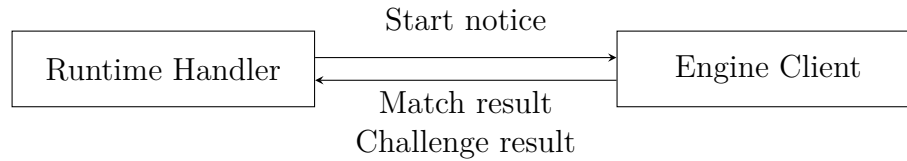
Possible simple messages between handler and engine client

Start notice   This message is only sent once per game to the engine client, as a signal that all actor clients have received all the data (api and actor binaries) that they needed to start working, and a game may be started. Following its reception, the engine client starts the game by invoking the engine object's `playGame` method.

Shutdown notice   This message can be sent at any time during the game, and signals its end — either due to some error happening or normal game completion — to the clients, who after receiving it should finish all running tasks, and stop themselves.

Log entry   Log messages are the only non-control messages, due to them not affecting the flow of the game in any way. They carry their target (the actor or actors whose log will contain the message) and their message itself as their payload. They are sent at any time by one of the actor clients, and are processed independently of the control messages by the runtime handler.

Error report   Much like log entries, error reports can be sent at any time; however, instead of being actor initiated, they are generated by the client runtime if any irregularity has happened. They can be caused by execptions thrown in the actors' code, unexpected runtime behavior, or attempted access to guarded resources by the actor. As they signal unrecoverable errors, the runtime handler will interrupt the game after receiving them, and choose a result fitting the context the report was created in and the information it carries. Based on these pieces of information, the handler will generally mark the game as ended with:

- An error caused defeat of the bot of the report's sender (e.g. in case of attempted permission violation or general exception in the bot's code).

- An error (no winner or loser) if the error occurred not in the actor's, but in the runtime's code

- Game error if the report was sent from the engine's client due to the former's failure.

Match result, Challenge result   Signals from the engine client that imply the engine object's `playGame` method successfully finished. They carry the data containing the game's result in accordance with the game model's description. If no error has happened, they act as the signal for the end of the game, and as such may be viewed to form a non-simple message pair with start notice.

Actor binary transfer

Actor binary messages serve as both parts of a request–response pair. When making a request, a client asks the handler for a type of binary resource required to do its job — the game's api, engine, or the managed bot. The handler in return sends back the requested binary, while also keeping track of all such requests. If all needed binaries have been sent to all clients, the handler will initialize the start of the game using a start notice message.
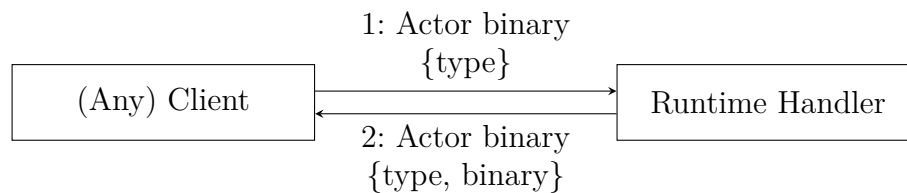
Actor binary request–response

Remote invocation

Remote invocation is the most complex communication operation as it involves routing by the handler, and multiple possible valid return types.

As the engine runtime's proxy captures a method call attempt by the engine object, the client sends a proxy call message containing the target bot, requested method identifier, and optional method parameters. This message is forwarded by the handler to the target client, who executes its bot's appropriate method. If this method produces a valid result under the configurable time limit, that value is returned wrapped into a call result message, all the way to the engine. If some error happens during the evaluation, (as usual) an error report is returned to the handler.

However, a special case is also possible, if the bot's method does not finish in time. If this happens, instead of an error report, a bot timeout message is sent back to the engine. This bot timeout acts as a 'recoverable error' — an event when the engine can choose to ignore the bot's shortcoming. If the engine was created without support for these occurrences, the default error processing takes place.

Engine Client			Engine Client	

1: Proxy call {method, params}   4: Call result {result}

1: Proxy call {method, params}   4: Bot timeout

Runtime Handler   Runtime Handler

2: Proxy call {method, params}   3: Call result {result}

2: Proxy call {method, params}   3: Bot timeout

(Target) Bot Client   (Target) Bot Client

Successful remote method invocation

Remote method invocation ended in bot timeout
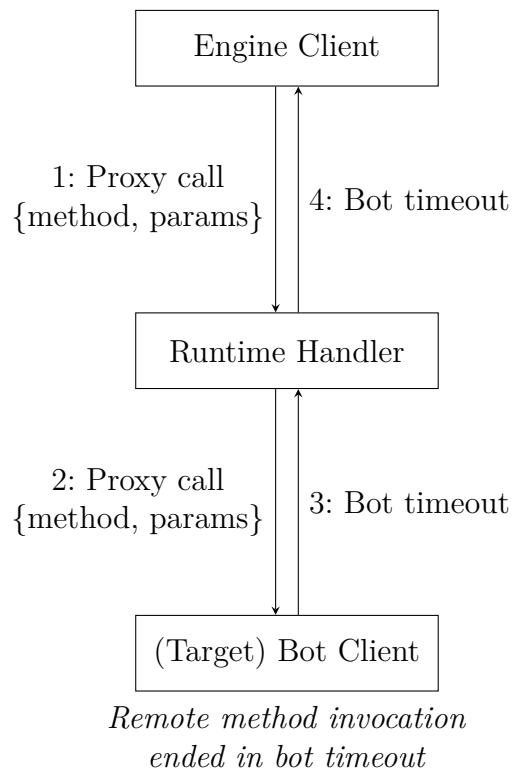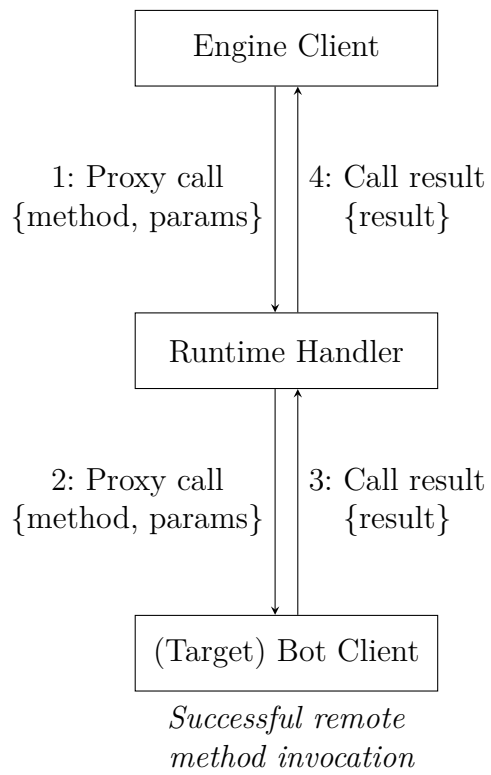
Chapter 5

Runtime Implementation

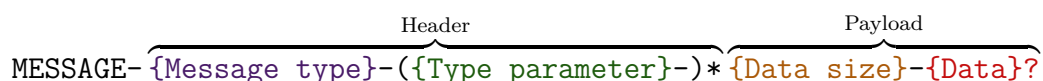## 5.1 Messaging

5.1.1 Format

All messages start with a fixed prefix (`MESSAGE`), end with a line break, and their different parts are separated by hyphens. Messages consist of two logical parts: the header and the payload. The former specifies the type of the message, and provides other information like the type parameters, which may be needed to process this kind of message; while the latter carries additional information in the form of a blob and its size in bytes.

$$\texttt{MESSAGE-}\overbrace{\texttt{\{Message type\}-(\{Type parameter\}-)*}}^{\text{Header}}\overbrace{\texttt{\{Data size\}-\{Data\}?}}^{\text{Payload}}$$

Structure of the messages

Type parameters and payload The difference between the type parameters and the payload data is that type parameters can be understood and processed without 'unsafe' operations, such as java deserialization. Their arity and types are determined by the message type, and their values may consist of primitives, strings, or values of enumerations. This — combined with a size limit on them — allows the runtime handler to always safely parse, process, and optionally, route the messages without requiring it to understand the potentially unsafe data of the payload.

The payload on the other hand deals with binary data produced by one of the actors, and as such should only be parsed by the actor clients, which are already equipped with the proper protective capabilities. The payloads size — in bytes — is also included in the message for safe message handling.

Both the type parameters and the payload are converted to base 64 strings, as to make message formatting and parsing easier — due to no necessary separator escaping.

Message type	Parameters	Payload
Start notice	————	————
Shutdown notice	————	————
Log entry	Target actor Log message	————
Error report	Error message	————
Challenge result	Points Max points	————
Match result	Three way result	————
Actor binary	Type	Binary (nothing on requests)
Proxy call	Target bot	Target method Call parameters
Call result	Called method	Result data
Bot timeout	————	————

Type parameters and payloads of message types

5.1.2 Conversion and serialization

Conversion and deconversion Conversion is the process during which domain representation of a message in the form of a `Message` object is converted into an intermediate state as it gets ready to be sent. Its result is a `MessageDTO`, an object which holds the same header the original message did, but has its payload — if any were present — transformed into a processable, base 64 encoded string representation. Deconversion is its inverse operation. It is a distinct process from the general serialization of messages, as to ensure independence between the algorithms used in the two processes.

This loose coupling is beneficial, as conversion is an unsafe routine that only actor clients should conduct. Message (de)conversion is required when dealing with remote proxy calls, and this breakup of concerns allows the runtime handler to parse and route a call request or call result message without attempting to decode its payload, which contains JVM objects as transformed by standard java serialization.

As object (de)serialization is dangerous — due to the construction of arbitrary large serialization 'bombs' —, it is best to pass these messages to the actor clients who are equipped with proper security measures to safely minister it.

Serialization and Deserialization Serialization takes payload-less messages or converted messages, and transforms them to the aforementioned general message format. This includes encoding the message header, and concatenating it to the prefix and optional (already converted) payload. This encoding of the header — and its pair decoding in deserialization — can be done safely and strictly, as the content of the header is specified by the message type, and its parameters are simple values that can be easily converted to strings.

As all parties require (de)serialization to communicate, the components providing these services are shared across all of them.

5.1.3 Communication

All messaging is done over point-to-point, full-duplex TCP channels between the actor clients and the runtime handler. These connections are initialized by the clients, who — as a command arguments — receive the host — usually localhost — and the port they should connect to. These ports are free ports chosen randomly by the handler, and reserved before the start of the clients.

Once these channels have been established, communication should begin as described by the communication model — starting with the clients requesting the game specific actor binaries.

If a game if finished — in a normal way or because of errors —, or a communication channel should fail (e.g. due to unexpected crash of an actor), the runtime notifies all (still reachable) clients with shutdown notices. After sending and receiving these, all participants are to assume the game has ended, the clients stop themselves, and the handler reports the result of the challenge or match.

Just like serialization, low level communication is helped by components shared across all members of the runtime model.

5.2 Runtime handler

A runtime handler is made up of a couple of active components and passive data structures. Active components are primarily based on their own threads, and communicate with each other through either direct calls, or the passive structures. There are two kinds of these active components, a — per runtime handler — singleton control handler, and actor handlers for each actor client runtime. Passive

components come in the form of a control queue and log stores for each actor.
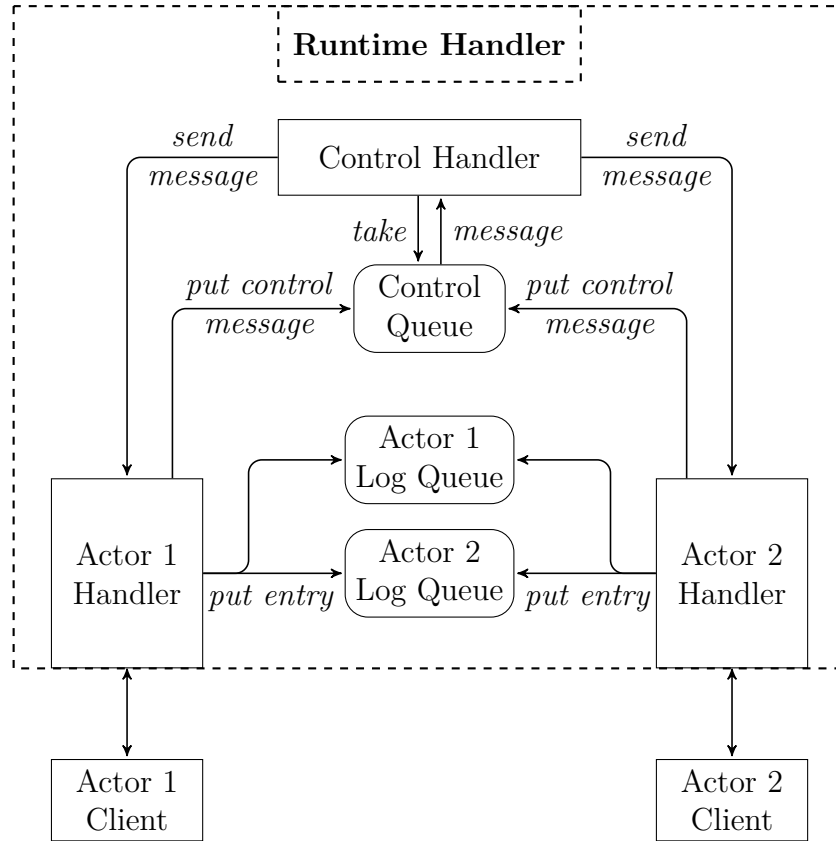
Runtime handler simplified component model
Challenge — two actors

5.2.1 Control handler

The control handler is the heart of the runtime handler, as its name implies, it directs the flow of the game, and manages the actor handlers. At its core it is a finite-state machine that processes control messages, and optionally instructs the actor handlers to send messages based on the actions.

It is important to note that the control messages it consumes are slightly different from the general messages sent between the members of the runtime model, as they have been preprocessed by the actor handlers. The two main changes are that log messages have been removed (those are processed by the actor handlers), and each message contains its source actor. The control handler acquires these messages from the control queue, which acts as a thread-safe buffer between the components. The possible control states are:

- **Connection await** The initial state, it signals that not all actor clients are ready to start, as not all have connected or received all the binaries they need.

- **Wait for Engine** The engine is working.

- **Wait for Bot** The engine had requested a proxy call, whose result has not been returned from the target bot's actor yet.

- **Finished** The game had finished normally. *End*

- **Crash** The game finished abnormally due to some error. *End*
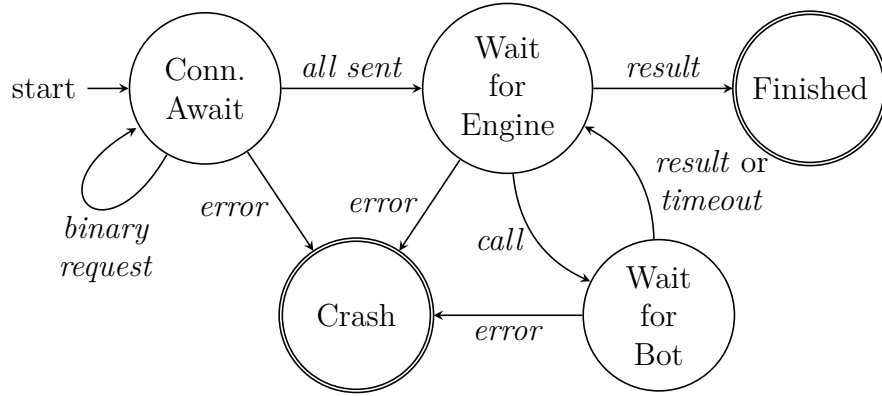
Control state machine

To simplify game flow management, the control handler is an extended state machine, which allows states to store variables as internal state. This data allows for selective transitions to happen, and help store results in the final (accepting) states. Each state may have its own set of parameters as follows:

Control state	Parameters
Connection await	sent binaries by receiver client
Wait for Engine	————
Wait for Bot	target bot, called method
Finished	challenge/match result
Crash	optional result, error description

Parameters of control states

Game flow The controller starts off in the `Connection await` state, and remains there until all actors have received their required binaries. It keeps track of the sent libraries, and after the last of these has been issued, a 'Start notice' message is sent to the engine client, the state is changed to `Wait for Engine`, and the game starts.

During the game, as the engine makes calls for which the bots respond, the state is bouncing between `Wait for Engine` and `Wait for Bot`. As a security measure, the target bot and called method are also stored, so that it can be verified that the right bot answered a call. As per the game model, 'Bot timeout' messages are also possible in the place of 'Call results'.

After the game had come to a conclusion, and the engine had sent the proper result message, the state reaches `Finished`, where it stays while keeping the game result.

Of course while in any active state, an error might also happen, in which case the controller changes to the `Crash` state. Although in this case a game result have not been reported by the engine, the controller might set a valid result anyway, based on the context in which the error occurred. If for example the game crashed due to a runtime exception while executing a code of a bot, the result of the challenge or match will be set to error from that bot — a result that is processed similarly to the opponent winning in the case of a match, or a 0 point run in the case of a challenge.
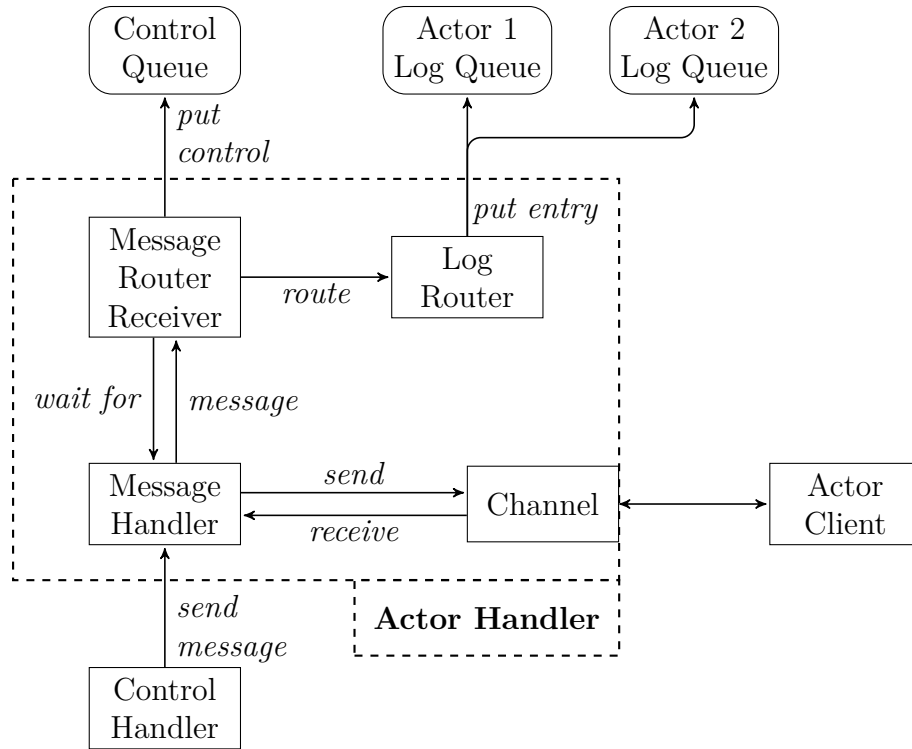
5.2.2 Actor handler

In contrast to the control handler, separate actor handlers are created for each actor. These composite components are responsible for communicating with the runtime clients, processing incoming messages, storing log entries, and forwarding control messages to the control handler in a digestible format.

Low level communication with the clients is done using a channel, an object which provides a safe, message-aware wrapper for the underlining network sockets. On top of this sits a message handler, which now provides operations based on the type of the clients (engine or bot). This handler is accessed by two components, the already mentioned control handler and the message router receiver. Both of these objects run on their own threads while using the message handler, but due to them only using the sending or receiving capabilities respectively (and the fact that the underlying architecture is full-duplex capable), these actions can happen independently. The message router receiver's thread can be considered the actor handler's main thread, as it performs most of the tasks of the handler. It blockingly waits for incoming messages, then routes them according their purpose.

Control messages get transcribed into the format the control handler can understand, and put onto the control queue from where they will be taken by the control handler. Log messages on the other hand are passed to a log router, an object which puts them into their target's log queue. As game engines have the power to simultaneously send info entries to multiple actors, this might mean more than one overall targets.

Actor handlers are designed to be as reusable as possible when dealing with different types of actors. This means that only the message handler and the log router differ based on actor kind — as they either must show different sending methods to the outside, or must work fundamentally differently while routing messages. Also their characteristics are hidden from the other components (except for the control handler) via abstraction achieved by the inheritance of a base type.

Actor handler model
Challenge — two actors

5.3 Actor runtime

The structure of actor runtimes is fairly simple, and is largely identical between engine- and bot clients. Their core component is the 'client', which provides the logic necessary to manage the actor assigned to them, and therefore of course contains actor role (engine or bot) specific code. The rest of the components involved in the communication process are equivalent in each case. The client is based on three active objects all running on separate threads — a message receiver, a message sender, and the aforementioned client. These elements are communicating with each other — in a one-way manner — over blocking queues.

The receiver waits for messages over the communication channel, processes them and places them on the 'In queue', from where the client can get them. It may also

initiate the exit of the client if a shutdown notice is received. Likewise, the sender sends messages placed onto the 'Out queue' by the client.
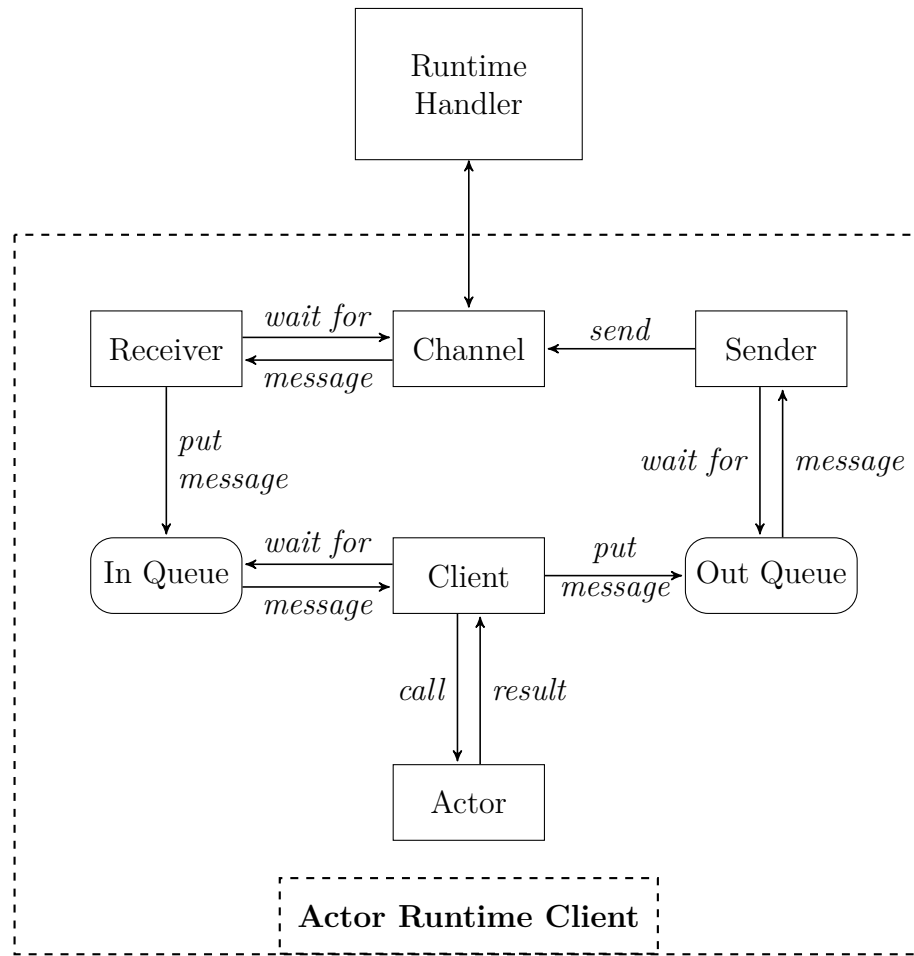
Actor runtime client model

5.3.1 Base client

As the client's workflow is dependent on the type of the actor it manages, it is realized as two different classes — the `EngineClient` and the `BotClient` —, but many common functions are outsourced from these to shared dependencies.

Class loading and security

The most important job of all actor clients is the provision of a safe runtime environment where the actor's and the game api's inherently untrusted code may be executed. This is supported by the standard solutions provided by the java virtual machine. The heart of guarding valuable resources (such as files or I/O capabilities), is the strict separation of safe- and unsafe code with their respective permissions assigned.

Safe code is constituted by all the sources of the runtime client and its dependencies, such as java libraries used — effectively the whole codebase of the runtime. This partition is granted all possible permissions using a policy file, which is — with the enforcing security manager — put into effect via program arguments passed to the JVM.

client.policy

```
grant codeBase "file:${trusted.codebase}" {
    permission java.security.AllPermission "", "";
};
```

This policy grants all permissions to classes whose source URI matches the file path also set as a vm argument through `-Dtrusted.codebase`. This property is in turn set by the starter script to the executable jar's path that not only contains the actor client, but also all of its necessary dependencies as well.

As a security manager is installed, other code sources have no granted permissions by default, therefore all other (unsafe) code parts need only not to be marked as being from the same source.

The loading of the user-defined code blocks is implemented by a custom class loader (`BinaryClassLoader`), which works on in-memory byte array representations of java archives (JARs). These libraries are sent over to the actor client in actor binary messages, then the client passes them to the classloader for processing. The loader iterates through the compiled referencial type entries of the archive (`.class` files), and defines them using the standard native class loader methods. The binary class loader also sets these classes' protection domain, which — due to different source definitions — provide no granted permissions to them.

Although this way no permissions are granted to the user defined programs, they would still be able to engage in unwanted behavior, namely multi-thread operations. The game model specifies strict 'ask–respond' style communication between the engine and the bots for clear gameplay and enforceable timeout limiting. This means that when — as a result of a proxy invocation — a method of a bot is called, it must finish all operations when that method returns. As threads cannot be safely killed once started, this effectively forbids custom thread creation, even in non-straightforward ways, such as the use of an executor framework. Fortunately, while 'no granted permissions' still allow thread creation, there is a safe way of disabling thread formation for unsafe code. At the initialization of a thread object, the environment does check access permission, but the default behavior is to allow thread creation if the thread is created into the current thread group. The look up of the current group is done by the set up security manager

using its `getThreadGroup` method. By overriding this method of the default security manager, we can return the root thread group, which will always be different from the caller's thread group, and thus thread creation will fail when attempted from unsafe code.

Actor initialization

Initializing an actor instance to manage starts with finding the game's bot interface. This is done by searching through the user defined types, and finding the appropriate class object. This capability is shared between the different types of runtime clients.

Engine client   After the bot interface has been found, the engine client creates one or two stub instances for this interface. This is done using the java runtime's built-in dynamic proxy creation capability through the `Proxy` class' static `newProxyInstance` method. This method takes an invocation handler — a callback that takes as its parameters the context in which a method of the proxy was called, and meta data about that same method. My implementation of this callback is to create and send — put into the outgoing message queue to be more precise — a proxy call message. This message will contain the target bot, a unique identifier of the method (full signature), and the method arguments serialized. After sending this message, the handler waits for a response (by blockingly querying the incoming message queue), unwraps the result, and returns it.

   After the stubs have been generated, the engine client searches through the user defined classes, looks for the engine, and creates an instance of it with the previously created proxies as the constructor parameters.

Bot client   The bot client finds the bot implementation that realizes the bot interface (and has a zero argument constructor), and instantizes it. Then the client uses the bot interface to create a skeleton for the remote invocation. This object holds all the bot interface's methods (represented by their unique identifiers, same as on the engine client's side) mapped to the aforementioned bot instance's corresponding methods. When a proxy call message is received, this skeleton executes a lookup of the method identifier, and calls the matching method of the bot, and sends the result back as a call result message.

Chapter 6

Kreator

Parallel to the development of the main project — and in coordination with it —, I have started working on a framework to allow well-integrated dependency injection for kotlin.

Although solutions of varying functionality and quality have existed for years — such as Kodein [2] or KOIN [1] for kotlin, and many others for the JVM in general [7] —, I have found them not to work in accordance of how I would prefer such a library or framework to behave.

6.1 Introduction

Dependency injection in object oriented environments is usually manifested in providing objects to other objects, where the latter use some accessible services of the former. This pattern can be immensely useful in reducing coupling and module dependency, which in turn allow for easier unit testing, better project transparency, and more.

Types of injection

Conventionally, dependency injection is separated into three main types: *constructor-*, *setter-*, and *interface injection* [4]. These cover most of the options commonly used in the java world, but — while using for-java libraries is possible, — due to more powerful language capabilities, solutions in the kotlin ecosystem use different methods. The most important features of kotlin that endorse different patterns are strong immutability support (`val`), simplified 'setter constructor' definition — both incentivizing an in-constructor initialization of dependencies — and first class singleton support (`object`) and top level functions, which allow for easier programmatic injection. Setter- and interface injection is, of course, still possible (through nullable or `lateinit` properties), but result in a reduction of

code quality.

Categorization

DI is compromised of two necessary phases: resource declaration and injection. The former is the process in which dependencies are identified for the framework, while injection is the practice of actually providing the requested dependencies, optionally initializing them in the process.

Most available dependency injection solutions work in what I would categorize into one of two ways: declarative- or imperative injection. Declarative injection uses annotations (or in the past, commonly XML configuration) to identify resources, and injects these at marked injection points, such as setters of annotated properties. This kind of injection relies on managed component handling to detect when an injection should occur. On the other hand, imperative injection frameworks require components to be registered at startup to be injectable, and they can be requested via functions.

6.2 Design

Kreator is based on a mixed system where resource identification is done in a declarative way using annotations, while resource injection is implemented in an imperative manner through global functions (`inject*`). This aims to capture the best of both words — easy to understand definition and idiomatic injection.

I have focused on supporting constructor injection, because as it allows for easy non-nullable, immutable dependency properties, it is the best type of injection. To achieve this, Kreator is heavily based on kotlin's default method parameters, which allow passing dependencies manually, or falling back to injection. The main design principle is that all components (classes) should take their dependencies as constructor parameters — usually with the type of their provided service interface, and usually using kotlin's unified constructor-property definition construct —, and provide default values for them via the injector functions. It is also an important thing that subtypes of classes shall have constructor parameters for their parents' constructor parameters as well (with default injection) to keep manual dependency passing possible. This is unfortunately easy to forget when extending a component, but is necessary to enable passing dependecies transitively. This design makes Kreator a fairly "magic-free" framework, where resource declarations are transparent and injection is cleanly done.

```kotlin
open class UserService(
    private val userRepo: UserRepo = inject(),
    private val accountRepo: AccountRepo = inject()
) {
    open fun create(name: String, balance: Int) {...}
}


class RemoteUserService(
    private val provider: ConnectionProvicder = inject(),

    // Always set parent's optional values
    // to allow manual dependency passing
    userRepo: UserRepo = inject(),
    accountRepo: AccountRepo = inject()
) : UserService(userRepo, accountRepo) {

    override fun create(name: String, balance: Int) {...}
}
```

Even though dependency injection is the goal of the framework, Kreator's injection mechanism is really just resource provision — provided resources are only "injected" to modules because the user connects the providers to the constructors. Because of this, Kreator can perhaps more correctly be referred to as a resource management- and provision framework. It is perfectly valid to use injector functions in method bodies or property initializers (or inside constructors), but doing so breaks the possibility of manual dependency passing, effectively hides these used services, and therefore is ill-advised. Of course it sometimes still can be useful, especially when initialization cannot happen by hand anyway, such as in singleton property initializers or in the main function.

Injection function

Injection is completed using three base functions, `inject`, `injectOpt`, and `injectAny`, whose behavior differ slightly. These functions all take advantage of kotlin's (limited) reified generics to provide type-safety. They are designed in a way to be easily extended by building atop of them in a third-party library. All three functions follow the same qualification filtering algorithm, but in the case of some edge cases return different results:

Standard injection   Using the standard injection logic, `inject` returns a non-null object, or throws an `InjectionException` if no resource matches the given injection qualifiers, or if multiple resources match the qualifiers with the same precedent.

Optional injection   Works like standard injection, except returns `null` if no matching resource can be found.

Any injection   Differs from the standard injection in that if the resource qualifiers are ambiguous, one of the matching resources are returned arbitrarily.

Arity

Each resource provider has an assigned arity that describes how it should be used by the injection framework.

- **Per request** The producer should be invoked on every injection request.

- **Singleton** The producer should be invoked at the time of the first request, the produced value should be stored and returned at future injection requests.

- **Singleton autostart** Similar to singleton, but the producer should be invoked at the time of the initialization (resource discovery phase) of the injection framework.

6.3   Injection qualifiers

Injection qualifiers are meta data connected to the injectable resources, which are used to select a resource for a given injection request. Such a request is a call of one of the injection functions. Its selector inputs are the implicitly defined — through generics — injection type, assigned environment, and an optional tag value.

Tag

Tags are the most straightforward qualifiers assigned to resources. They are custom strings that are tested based on their equality to the (optional) query tag, and a resource definition can have multiple of them. At injection, the caller may pass a tag with which resources must be marked to be eligible. As these string can have any value, it is up to the programmers to create standard, meaningful tags to be used in a project. Tags should be used as a way of describing different behavior between resources of a type, and therefore often can be grouped by some attribute.

Examples: *file*, *db*, *in-mem*; *direct*, *cached*; *blocking*, *async*; *secure*, *non-secure*

Environment

The environment describes a hierarchical (tree organized) setting, in which a given resource can be used. An environment value is also set for the program at runtime using the `KREATOR_ENV` environmental variable. This qualifier is unique, as it not only filters resources, but also groups them into classes. These classes are looked through in a specific order to find matching resources. The groups are as follows:

Env. group	Example resources	Lookup
Exact match	`"test.unit"`	First
Sub environment	`"test.unit.junit"`	Second
Sup environment	`"test"`, `""`	Third
Neither	`"dev"`, `"prod.local"`	Never

Groups and examples for the program environment `"test.unit"`

As the default environment (`""`) is a parent of all others, if no value is set, all — otherwise eligible — resources can be injected.

Environments are use to define environment dependent behavior, most commonly to create test stubs, fakes, and mocks that should not be used in production. They can also be used to make the program work seamlessly on any host, e.g. on different cloud hosts or operating systems. Environments can also be used to provide implementation with different error processing or logging configurations.

Test resource limitation is achieved in the main project — in part — using environments, as the maven surefire (tester) plugin is configured to execute unit tests under `"test.unit"`.

Default

The default flag is used to select a resource when multiple are eligible for injection. It is a simple boolean value, false by default. Default resources always take precedence over non-default ones.

6.4   Resource declaration

In contrast to some DI frameworks (such as most declarative ones, including java CDI [6]), not all classes that comply with some rules (e.g. having a no-arg constructor) are automatically eligible for injection. As more common in programmatic systems, resources must be marked explicitly as injectable — but

using annotations. The four annotations associated with resource declaration are `InjectableType`, `Injectable`, `TestInjectable`, and `NotInjectableFor`.

InjetableType This annotation marks types that act as a target for injection. It is intended to be used on interfaces that define the behavior of a type of services.

Injectable, TestInjectable These define resource providers with their respective configuration. They may be placed on no-argument functions, -constructors or classes with such a constructor. Their configuration defines the qualifiers of the resource (environment, tags, defaultness), its arity, and their injectable types that define for which types can the resource be provided. By default (if the types are left empty), a resource is injectable for its own type (without it needing to be marked injectable type) and all of its parent injectable types. If types are set explicitly, the resource can only by injected for the set types.

A single provider can have multiple injectable annotations, each with their own set of qualifiers to establish different roles for the resource.

`TestInjectable` is a simple type-safe helper whose only job is to prefix the environment with `"test"`, the most commonly used environment (other than `""`).

NotInjectableFor This annotation acts as a disqualifier for specific types. It is to be placed on providers that are annotated with `Injectbale` (or `TestInjectable`) and have a lot of injectable parent types, but are not intended to be injectable for all of them. It allows the type qualifier to stay non-defined (implicitly all injectable supertypes), and to list only a handful of supertypes as not injectable for.

6.5 Property injection

As an independent submodule, Kreator contains a property configuration framework that allows for string, integer, and boolean properties to be defined in various sources, such as environment variables, property files, or structured config files. Like the resource injection parts, this module is configured via environmental variables (source type and optionally source location), supports optional and non-optional properties, is type-safe, and provides default no-op sources if configuration is not present.

This system is relied on by the main project for environment dependent configuration of the data source, runtime actor handlers, test games, and test data manager scripts.

Sample config — `tulkas.conf`

```
Client {
    Log {
        base-dir: /tulkas/log/
    }
    Engine {
        script-path: /tulkas/engine-runtime-client/start.sh
        redirect-out: true
    }
    Bot {
        script-path: /tulkas/bot-runtime-client/start.sh
        redirect-out: true
    }
}

Server {
    Database {
        DriverClass: org.hsqldb.jdbc.JDBCDriver
        ConnectionString: jdbc:hsqldb:mem:tulkasDB
        Username: tulkas
        Password: tulkas

        Script {
            Init:  sql/create.sql
            Fill:  sql/insert_test_data.sql
            Clear: sql/clear_tables.sql
            Drop:  sql/delete.sql
        }
    }
}
```

```kotlin
@InjectableType
interface ConnectionSource {
    operator fun invoke(): Connection
}


@Injectable(
    arity=SINGLETON_AUTOSTART, default=true, tags=["jdbc"]
)
class JdbcConnectionSource(
    driverClass: String
            = property("Server.Database.DriverClass"),
    private val connectionString: String
            = property("Server.Database.ConnectionString"),
    private val username: String
            = property("Server.Database.Username"),
    private val password: String
            = property("Server.Database.Password")
) : ConnectionSource {
    init {
        Class.forName(driverClass) // Load external class
    }

    override operator fun invoke(): Connection =
        getConnection(connectionString, username, password)
}
```

6.6 Project structure

Kreator is structured as a maven project, and is separated into four modules all targeting their respective jar files — *annotation*, *api*, *core*, and *property*.

Annotation contains necessary types for resource declaration, it is a very minimal library that can be used by any framework or library that wishes to provide optional Kreator-based injection support. It only contains declarative parts, therefore if a dependency of a program is built on it, but the program itself decides not to use Kreator, no disadvantage will happen.

API defines the injetion functions with default no-op implementations, therefore it, too, can be used in a library without it affecting all users of that library, only the injections will fail (return null on optional injections).

Core   includes the default implementation of the injection logic defined by *api*, and is necessary for injection. It is designed to be easily replaceable by alternative implementations.

Property   defines and implements all the aforementioned property configuration capabilities.

The project is available as free software on GitHub [3].

Chapter 7

Example game: Battleship

The game is a version of the classic battleship puzzle game [10]. It is played by two players who both try to sink their opponents 'ships', while avoiding their own ones getting sunk. The game has two phases — the setup of the battlefield, and the shootouts. Initially both players have 7 ships, which have distinct lengths and are one unit in width.

Ship name	Size	Count
Battleship	4	1
Cruiser	3	2
Destroyer	2	3
Submarine	1	4

Ships of a player

During the setup phase, the players place all their ships on their 10×10 grid battlefield vertically or horizontally. The only limitation is that ships may not border each other over edges (they can touch diagonally). After this setup is complete, the players may not reposition their ships for the remainder of the game and their layout is kept secret.

Then the shootout begins. Taking turns, the players select targets (by their coordinates, such as 'A–5') and their opponent report them the result of the attack as either 'miss', 'hit', or 'sink'. Miss means no ship has been placed at that position, hit and sink reflect that a ship in fact occupied that place, sink also means that all of the ship's part have been hit previously. The player then marks the selected position on his own target field (as hit or miss), and if the result was not a miss, the other player marks the targeted part of the ship hit. The current player keeps shooting at new targets until he misses, or all the enemy ships have

sunk. After that — if not all ships are gone — a new turn starts, and the opponent starts shooting. The first player to destroy all of their adversary's vessels wins the game.

7.1 Design

Designing often involves taking a gameplay planned for real-world usage, and created with physical limitations in mind, and modifying it to naturally fit into a virtual environment. This, of course, is done any time a digital version of a game is being made, however, in this case a notable design consideration is the fact that the players (bots) and the core game (engine) must be separated from each other.

I have decided to follow the natural separation of game phases — setup and shootout.

The setup could be done through a number of ways, for example per-ship placement requests (engine asks the bot where each ship should go), distinct placements of ships (engine asks the bot to place one of the remaining ships), or a one-time setup of all ships (engine asks for a complete setup). I have opted to use the latter, as I felt it matches the layout planning the most. Since the ships have placement constraints between them, it is probable that a bot places all of them at once to ensure the validity of the design anyway.

The shootout part is basically the same operation done over and over on an ever-changing target map. It, again, could be designed in many ways, e.g. with a single-, or with result-dependent result callbacks (`result(...)`; `onSink(...)`, `onHit(...)`, `onMiss(...)`), or with the previous result sent at each new target request. I have elected to go with passing the results at every new request to simplify the bot interface. This means that at the first shooting, the previous status is a `null` value, which is not ideal due to java's lack of proper distinction between nullable and non-nullable types, but with good bot interface documentation it is not a problem.

7.2 Game API

The api of the game is fairly simple, mostly consists of model classes that help describe the game model in a type-safe manner. For a more complex game the creation of extra utilities would make sense as to make bot development easier.

7.2.1 Domain classes

These include any models that are used in the engine–bot communication or may be useful for bot development.

Position   A simple $(x; y)$ coordinate pair, also lets one check whether it is a valid position on the game field. *(data class)*

Direction   The orientation of a ship, may be vertical or horizontal. *(enumeration)*

Ship   Describes the properties of a ship, including its (start) position, direction, and size. *(data class)*

ShootResult   the result of a shot, may be a miss, a hit, or a sink. *(enumeration)*

TargetMap   the view of the opponent's field. Shows all the tried positions as shoot results. When a ship is sunk, its bordering water is marked missed, and the ship itself as sunk. Each position can be queried individually, and it also contains a helper method to list all non-yet-tried targets.

7.2.2   Bot interface

The bot interface (`BattleshipBot`) is based on the design of the communication between the engine and the bots, therefore it has two abstract methods: `setupField` and `nextShoot`.

`setupField` takes the sizes of the ships to be placed as its parameter, and must return a valid list of ships. It is called once, at the beginning of the game.

`nextShoot` is called on each turn and is used to determine the target of the next shot. Its arguments are the current value of the bot's target map, the previous target of the bot, and the result of the previous shot. The latter two are `null` on the first turn of the bot. The method must return a valid, not yet tried target to shoot at.

<div align="center">BattleshipBot.java</div>

```java
public interface BattleshipBot extends BotInterface {

    List<Ship> setupField(final List<Integer> shipSizes);

    Position nextShoot(
        final TargetMap targetMap,
        final Position previousTarget,
        final ShootResult previousResult
    );
}
```

7.2.3 Utility

As this game is fairly uncomplicated, I have create a single utility class for shared usage — `ShipUtil`. This helper has a simple public method, `randomSetup`, which places all the ships of a player at random — but valid — positions at arbitrary orientations. I have found this to be rather practical, as the placement of the ships — if not following some elaborate scheme — is often done randomly by the players, and is a non-trivial task (due to the rules of ship placement) to program.

7.3 Engine

The engine module contains the actual game engine class, as well as other resources that class uses, such as `ShipMap` and `ShipState`, which mutably hold information of the state of a player's field.

The game engine class' structure is quite typical. For per-bot data keeping it defines a static inner class to hold the field of the bot with its last target and the result of that shot (if any). For each bot, an instance of this gets created on the duel game engine's bot initialization phase, based on given bot's placement of its ships.

After that, the turn-based shooting starts. A turn here — by the duel game engine's definition — represents a set of operations that take place while one of the bots is working. This may include multiple shots, until the active bot finally misses or wins the game by sinking all enemy ships. After each shot attempt, the engine first validates the target by checking that it is a not yet tried position on the board. Based on the validation result, the engine either declares the game lost due to an error, or actually processes the changes, which lead to a shot result. The bot's previous shot data is always updated to the current target and result, then based on the shot result some action may follow:

- If the shot was a miss, the bot's turn is over.

- If the shot was a hit, the turn continues with a next shot.

- If the shot sank a ship, the engine checks if all enemy ships are sunk. If that is the case, the game is won by the current bot, otherwise the turn continues as with simple hits.

7.4 Random bot

I have found it often useful when implementing a game to first create a simple bot that plays the game 'dumb', but in accordance with the rules. This helps in both

testing the engine, and can be used to duel against a more competent opponent later on.

In the case of battleship, I have created a bot that places its ships randomly, then at the shootout phase chooses targets also randomly — of the not yet tried tiles —, regardless of its attacks' result.

This implementation, of course, performs very poorly ($\mathbb{E}(x) \approx 71.19$ turns to finish), but can still give a good idea of what functionality to build into the api, in this case the shared random ship placement was a result of this experiment.

RandomBot.java

```java
@Override
public List<Ship> setupField(List<Integer> shipSizes) {
    return ShipUtil.randomSetup(shipSizes);
}


@Override
public Position nextShoot(
        final TargetMap targetMap,
        final Position previousTarget,
        final ShootResult previousResult) {

    // all valid targets
    final List<Position> targets =
            targetMap.freePositions().collect(toList());

    // choose randomly
    return targets.get(random.nextInt(targets.size()));
}
```

7.5   A smarter bot

I have also created a better performing bot which can play the game almost as well as a human can (`SmartBot`). Just like the previous bot, it places its ships randomly, and also shoots at random if it hasn't discovered any ships yet. However, when it hits something it starts attacking that target consciously. This starts by shooting around the hit mark to find the orientation of the ship under attack, and when that becomes known, shooting at the end of a — randomly chosen — longitudinal side. It of course can take multiple turns to sink the ship, but it will get done sooner or later.

It is not an optimal player, because it does not select new targets intelligently (filtering based on where remaining ships can fit) when a ship is sunk, but it is fairly

performant due to the gameplay being inherently heavily based on sheer luck.

This smarter bot is naturally vastly superior to the previous random bot — I am yet to see it be defeated by that player.

Chapter 8

Managment framework

The core runtime constitutes the main part of the project, but in and of itself only allows running challenges and games given the necessary actors and game libraries. To make it accessible as a server side application, a standalone program is needed to store and manage actors and games, and also provide a web frontend for human interaction.

8.1 Persistence

Data persistence is by default handled by an in-process HSQLDB instance running in embedded file mode. The connection parameters and database location is configurable, and if the database is not found the program attempts to create it at startup. Database access is maintained over JDBC using standard SQL.

In lieu of an ORM system, a simple submodule is used for object persisting and querying. I have choosen this solution because I have felt the project does not require the complex solution of most ready-to-use libraries and a custom set of abstractions, and helpers would do. These resources are well integrated into the Kreator dependency injection framework, and match my preferred style of data handling. Repositories can be defined for the desired entites, that need only to define two-way object-row mapping to enable various data access functionality (result mapping, single- or multi entity selection, single- or multi entity persistence, deletion), atop which more behavior can be built. Transaction management is also handled through utilities which build on JDBC checkpoints and multiple connections to provide nested and guarded transactional behavior that is well-integrated into kotlin's capabilities, like with-receiver lambda expressions and extension methods.

8.2   Game management

Game management consists of queuing future games, running them using the runtime library, persisting the results, and ranking bots. Currently bot ranking is done based on the result of a single game in the case of challenges, and cross-matches of all submitted bots in the case of matches.

Challenges are thus fairly simple to manage, as if an unranked bot is found, the managment engine only has to run a single game with it. However, due to a possibly more complex game result (points out of maximum points, only points, variable maximum points), ranking based on the individual results takes some planning. Here I have decided to take all points/max points type results as a ratio in the $[0; 1]$ interval, and all single value results as their values unmodified (error is the same as a 0 point result). I have decided to do so because I have assumed consistent scoring within games, and so no mix-up between the types of results. After this, bots are simply ranked by their acquired points.

Match bot evaluations on the other hand are harder to organize, but — as their results are three-way outcomes — easier to evaluate. The current system here takes each newly uploaded bot, and runs them against all previous bots (two times, both bots being first and second player once). This ensures that all bots have played against each other in at least one pair of matches. If some bots have played multiple twin matches (e.g. due to unexpected restart of the program), then only their last match-pair counts. Each win (or opponent error) gives one, each loss gives minus one point to the bots, who are then ranked by their cumulative points.

If the system goes down for whatever reason during game managment, at the next startup the queue of games to be played is recreated, and evaluation continues. As result persistence is transactional, no incomplete result is stored in the database.
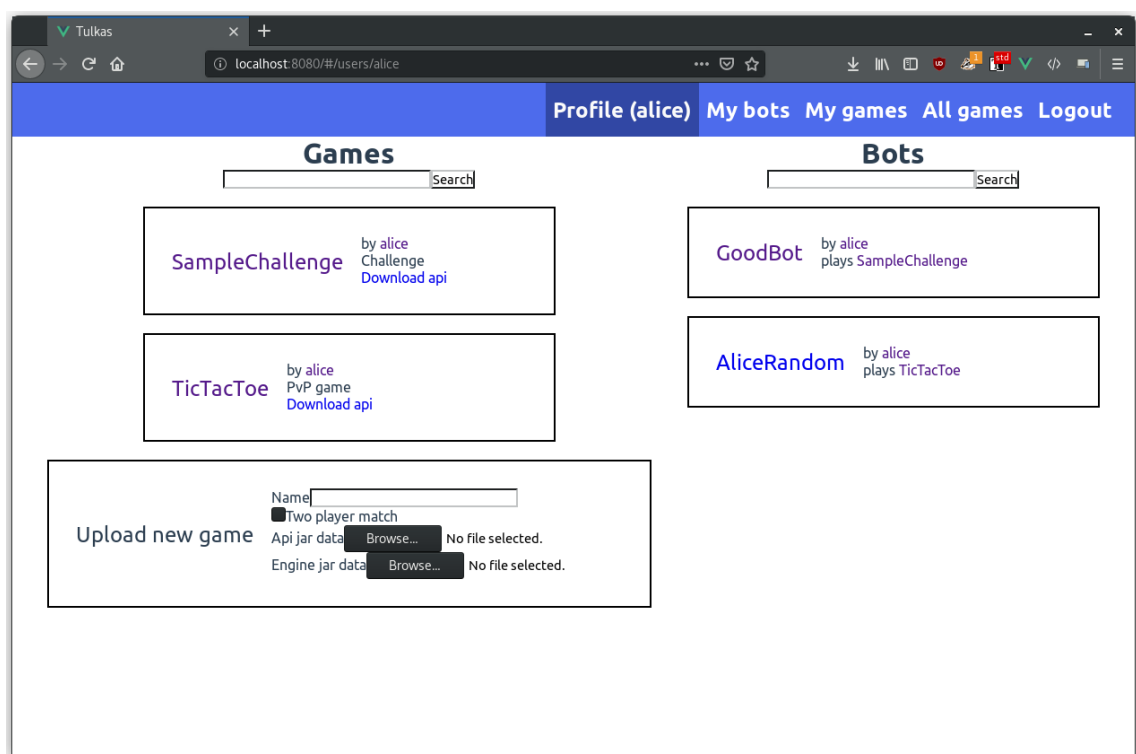
8.3   Web interface

The web backend is a kotlin standalone application which provides a RESTful API via JAX-RS with Jersey as its implementation. The program launches an embedded Apache Jetty web server to handle requests. Data serialization (JSON) and endpoint handling services are registered from JAR dependencies at startup.

The frontend web application is written in javascript using Vue.js. It is a single page application that uses Vue Router. The site is based on reuseable, single file components that — following Vue's guidelines — describe, implement, and control an element, such as a bot- or game information card, or a user's profile.

The site is minimalistic in design as it serves mostly as a demonstration of what capabilities a support interface should have — bot- and game upload, game api

publishing, game log viewing, etc.

Web interface user profile screen
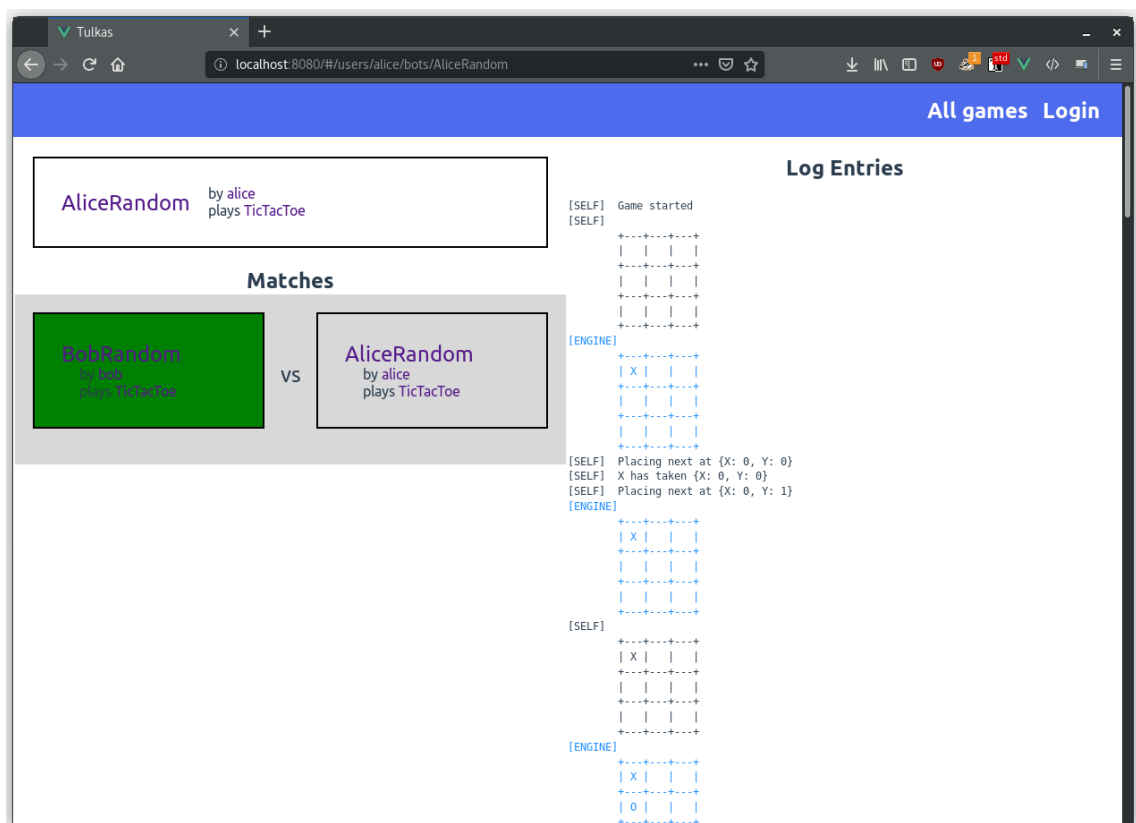
Web interface bot screen
Played matches and log entries for selected match

Glossary

Game

- A playable entity for which playing bots can be written. It consists of a game API, a game engine, and other meta data including a unique name and player number information.

- An actual event of a game being played by a bot or bots, and guided by the game's engine.

Challenge   A single player game, whose result (if not an error) is a whole number representing the score received by the playing bot, and an optional maximum points limit.

Match   A two player, competitive game, which can be played by two different bots, and results in a three way outcome or an error.

Game Runtime API   A java library containing type definitions that allows games and bots to be created. It is to be used as a provided (non included) dependency for the game engine, game api, and bot implementations.

Engine Runtime API   A java library containing type definitions and abstract base implementations that allow game engines to be written. It is to be used as a provided (non included) dependency for the game engine.

(Game) Engine   An executable library that provides the necessary logic and implementation for a game to work according to its rules.

Game API   A java library that contains type definitions and utility code that enable users to create bots for the specific game.

Bot interface   An interface that extends the `BotInterface` marker interface defined in the game runtime api. It defines the game specific methods, through which the game engine can interact with the bot.

Bot   A virtual player built to play a specific game — alone if that is a challenge or against another bot if that is a match. It is implemented as a class realizing the game's bot interface.

Actor   A collective term referring to a game engine and the bot or bots playing its game in a given challenge or match.

Actor client   A separate process which handles and interacts with a single actor throughout a game.

Bibliography

[1] Insert-koin. *KOIN*. `https://insert-koin.io`. (Visited on 11/28/2018).

[2] Kodein. *Kodein DI*. `https://kodein.org/di/`. (Visited on 11/28/2018).

[3] Márkus Krisztián. *Kreator*. `https://github.com/mktiti/Kreator`. (Visited on 11/28/2018).

[4] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. `https://www.martinfowler.com/articles/injection.html#FormsOfDependencyInjection`. (Visited on 11/28/2018).

[5] Oracle. *Access Control Mechanisms and Algorithms*. `https://docs.oracle.com/javase/8/docs/technotes/guides/security/spec/security-spec.doc4.html#a20389`. (Visited on 11/28/2018).

[6] Oracle. *Beans as Injectable Objects*. `https://docs.oracle.com/javaee/6/tutorial/doc/gizks.html`. (Visited on 11/28/2018).

[7] Oracle. *Contexts and Dependency Injection for the Java EE Platform*. `https://docs.oracle.com/javaee/6/tutorial/doc/gjbnr.html`. (Visited on 11/28/2018).

[8] Oracle. *Evolving the Sandbox Model*. `https://docs.oracle.com/javase/8/docs/technotes/guides/security/spec/security-spec.doc1.html#a18314`. (Visited on 11/28/2018).

[9] Oracle. *Permissions and Security Policy*. `https://docs.oracle.com/javase/8/docs/technotes/guides/security/spec/security-spec.doc3.html#a19802`. (Visited on 11/28/2018).

[10] Wikia. *Battleship (game)*. `http://battleship.wikia.com/wiki/Battleship_(game)`. (Visited on 11/28/2018).