

Welcome!

Implementing Scala-CLI on Scastie

Semester bachelor project at the Scala-Center

Please try to avoid printing this document, use the online version for links and saving trees 🌳

Before anything

- I hope you are having a great day.
- Take a bottle of water, it is really hot outside 🥵
- This oral presentation is a condensed and oral version of my report. Some details had to be removed but for completeness, you may refer to it.
 - Every slide that has a link to the report will be mentionned in the footer.

What is Scastie?

- A demo is worth a thousand words...

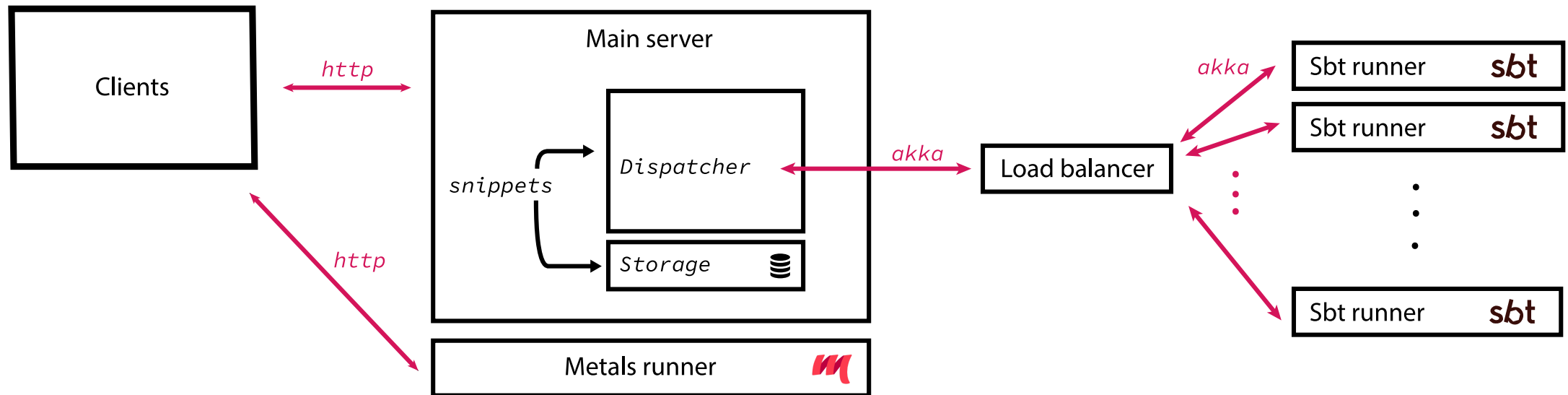
What is Scala-CLI?

- A Scala runner with extra features.
- We focus on directives.

```
//> using scala "3.2.2"  
//> using dep "com.lihaoyi::os-lib:0.9.1"
```

- Better than `build.sbt` sometimes...

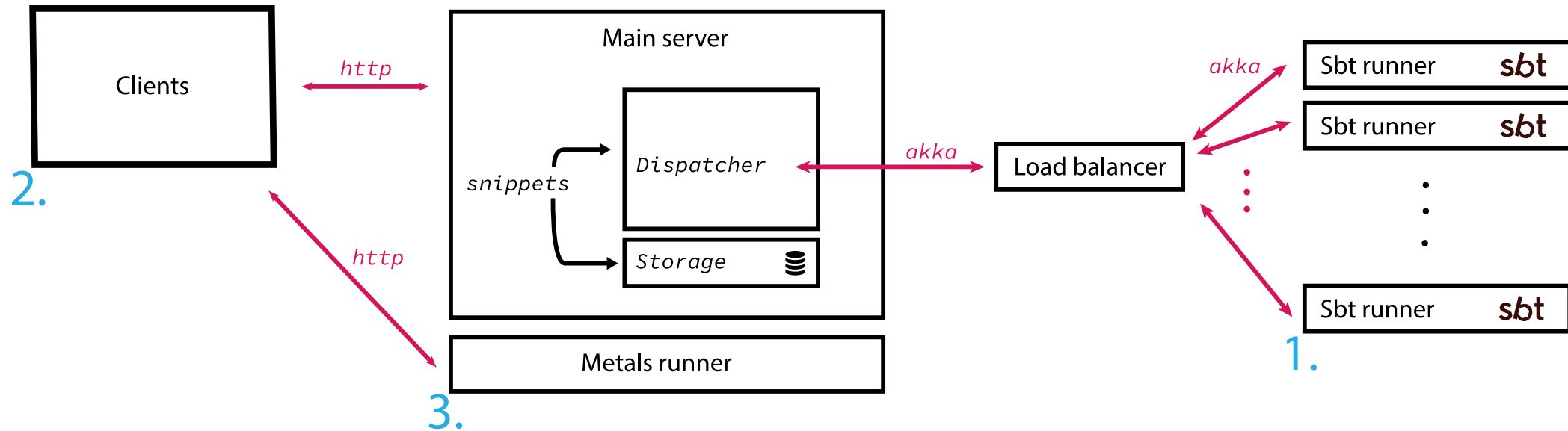
How Scastie works?



→ The issue: reloading takes too much time on runners

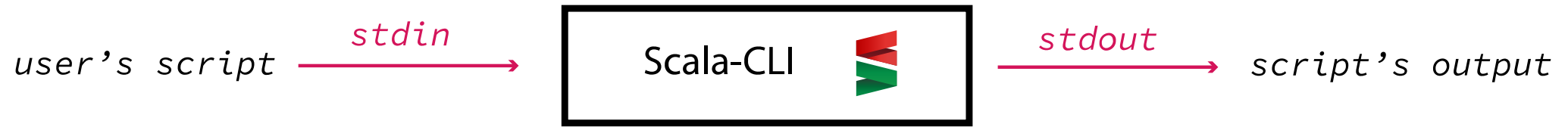
Steps

1. Create the Scala-CLI runner
2. Create UI components for Scala-CLI
3. Make directives work with Metals



1. How the Scala-CLI runner was born...

a. A first ~~stupid~~ idea, the prototype



- The issue?
 - Compilation errors are not machine readable. Hard to handle them and forward them nicely to the users.
 - Might create some obvious issues with my colleague you've seen previously...

Previous implementation with SBT

⚠️ The runner was **parsing the process' output!** Crazy people... I'm lazy.

👍 Lazy is a good point despite what you might think, I like to write the fewest lines of code possible so it is easy to maintain 😊

b. Let's start from scratch and do it properly

Idea: How do IDEs talk to compilers?

Solution: A protocol?

Is there something like this already? Someone must have thought of it.

Open-source software people are crazy.

The [Build Server Protocol](#) comes to the rescue!

The Build Server Protocol (BSP) provides endpoints for IDEs and build tools to communicate about directory layouts, external dependencies, compile, test and more.

Or to be short:

Talk to compilers.



Let's do it properly with the Build Server Protocol

It's nice!

BSP

A protocol to talk to compilers. Inspired by the Language Server Protocol.

- Did someone also made a library for this? Of course!

Why BSP?

- Because Jędrzej (❤️) told me so...

Please stop joking, it's a serious presentation.

Why BSP?

- Trigger compilations (`buildTarget/compile`)
- Trigger runs (`buildTarget/run`)
- Get diagnostics such as compilation errors and warnings (notifications on `build/publishDiagnostics`)

→ Looks perfect!

What's more? Extensions!

- If the compiler has specific capabilities, some endpoints are defined. In the case of Scala and this project, the endpoint `buildTarget/scalaMainClasses` was useful.
- Why? Wait for a bit... Everything in its time!

Side note: Futures are great in Scala.

How does the actual code looks like? 1/2

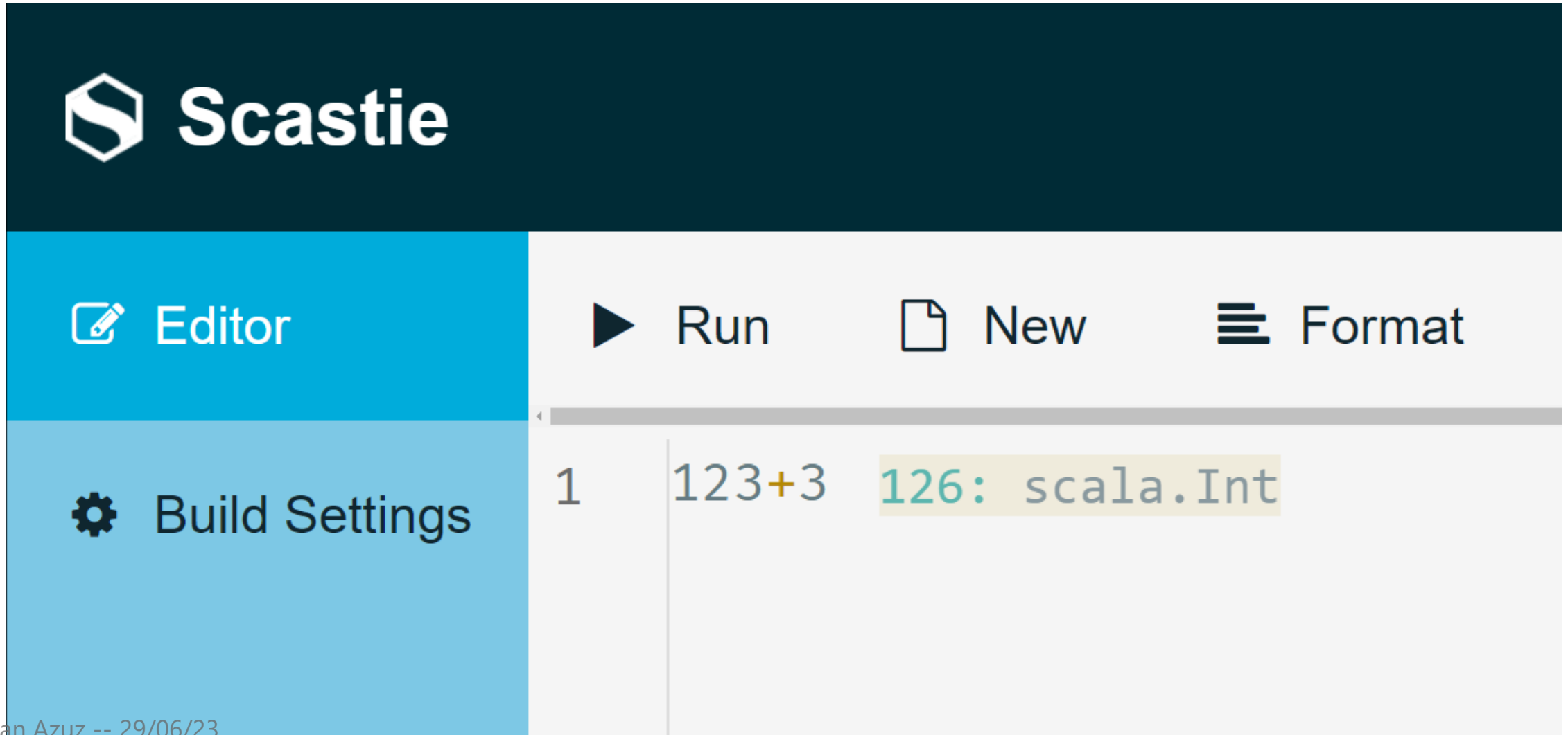
```
for (  
  r <- reloadWorkspace;  
  buildTarget <- getBuildTargetId;  
  
  // Compile  
  compilationResult <- withShortCircuit(buildTarget, target => compile(id, target));  
  
  // Get main class  
  // Note: it is combined to compilationResult so if compilationResult fails,  
  // then we do not continue  
  mainClass <- withShortCircuit[(BuildTargetIdentifier, CompileResult), ScalaMainClass](  
    combineEither(buildTarget, compilationResult),  
    {  
      case ((tId: BuildTargetIdentifier, _)) => getMainClass(tId)  
    }  
  );  
  
  // ...
```

How does the actual code looks like? 2/2

```
// Get JvmRunEnv
jvmRunEnv <- withShortCircuit[(BuildTargetIdentifier, ScalaMainClass), JvmEnvironmentItem](
  combineEither(buildTarget, mainClass),
  {
    case ((tId: BuildTargetIdentifier, _)) => getJvmRunEnvironment(tId)
  }
)
```

Why?! Everything in its time... 😊 Trust me.

Instrumentation? With which magic we end-up with this?



Instrumentation walk-through 1/3

Suppose that we execute

1

We are minimalist

Instrumentation walk-through 2/3

🧙 Some magic... Scala Meta-programming's documentation was not for me...

⚠ The shown result will be a beautified one. Refer to the report for details.

Instrumentation walk-through 3/3

Result:

```
import com.olegrych.scastie.api.runtime._

object Main {
  def suppressUnusedWarnsScastie = Html
  val playground = Playground
  def main(args: Array[String]): Unit = {
    playground.main(Array())
    scala.Predef.println(<instrumentations result>)
  }
}

object Playground extends ScastieApp {
  private val instrumentationMap$ = Map[Position, Render].empty
  def instrumentations$ = instrumentationMap$.toList.map {
    case (pos, r) => Instrumentation(pos, r)
  }

  locally {
    val $t = 1
    instrumentationMap$(Position(0, 1)) = render($t)
    $t
  }
}
```

Remarks on the instrumentation

- We have new objects
- Comments will end-up in the `object Playground`.
 - if comments ends up here, directives too! We have to put them at the top of the file.

How to do it?

Fairly easy thanks to the Scala standard library.

```
val (userDirectives, userCode) = task.inputs.code.split("\n")  
    .span(line => line.startsWith("//>"))
```



Now, what is the runner doing?

Let's backtrack to all my tries, including the one where I took my laptop out in the replacement bus from Genève Cornavin to Chêne-Bourg, looking like a nerd in front of normal people.

What's doing the runner?

1. Starting Scala-CLI in BSP

Spawns a `scala-cli bsp .` in a temporary empty folder. Scala-CLI initializes a new workspace.

I lied...

It spawns `scala-cli bsp . -deprecation` to enable warnings on deprecations: 🤖.

2. Inits a BSP connection to the server

build/initialize :

```
export interface InitializeBuildParams {  
  /** The rootUri of the workspace */  
  rootUri: URI = "<folder>";  
  
  /** Name of the client */  
  displayName: String = "BspClient";  
  
  /** The version of the client */  
  version: String = "1.0.0";  
  
  /** The BSP version that the client speaks */  
  bspVersion: String = "2.1.0-M4";  
  
  /** The capabilities of the client */  
  capabilities: BuildClientCapabilities = BuildClientCapabilities(listOf("scala"));  
  
  /** ... */  
}
```

Once finished,

The runner is ready for run requests!

Overall idea:

1. Instrument code, and write it on disk.
2. Compile the code `buildTarget/compile` . Forward any error to the user.
3. Run the code using `buildTarget/run` . Forward any result or timeout issue.

You have seen the code before and it has way many more steps, what happened?!

1. Instrumenting the code

We have to take into consideration the directives. This is done.

Write everything to a file `main.scala` .

⚠ Instrumentation needs to know the Scala version

The runner tries to find the directive `//> using scala "<version>"`. Finds the target from `<version>` and forward it to the instrumenter!

BSP world is here.

2. Compile the code

Compilation was fairly easy to handle. But how does it work?

How does it work?

- Client (Scala-CLI runner) sends a `buildTarget/compile` request.
- Server (Scala-CLI) will notify the progress and compilations errors in a notification `build/publishDiagnostics` .
- We need to keep track of these diagnostics.

But before compiling, we need to refresh the workspace (to be extra-sure) that dependencies are taken into consideration!

So the steps of compiling are:

1. `workspace/reload`
2. `workspace/buildTargets` to get the builds' target `id`.
3. `buildTarget/compile` to compile. While keeping track of the diagnostics.

Compilation is working! Except something...

Infinite compilations

With Scala's meta-programming, it is possible to make code that never compiles. Despite sending cancellation requests, the cancellation never ends for the current file (that may contain this issue).

The chosen solution is to log the problematic script, `sys.exit(-1)` and let Docker restart the container.




Example of non-compiling code (please do not compile it):

```
//> using scala "3.2.2"  
//> using lib "co.fs2::fs2-io:3.6.1"  
  
import cats.effect.*  
import fs2.*  
  
object Main extends IOApp.Simple:  
  def run: IO[Unit] = Stream.duration
```

On my pretty good PC, the compilation took 11GB of RAM, constant 50% usage of my CPU (i9-13900k), 2 minutes to fail.

The runner is limited to 1.5GB of RAM.

Where are we?

1. Instrumentation 
2. Compiling 
3. Running 

How to run the code?

Use `buildTarget/run` and voilà!

It did work as intended! However...

Non-finishing programs such as:

```
while(true) do println("hello to the ones reading the slides!")
```

or if you like for-comprehensions

```
for (_ <- Stream.from(0)) do println("hello again...")
```

create an issue.

We can send a cancellation request to Scala-CLI to stop the program.

But, it did not **work** for a reason that is obscure and related to Scala-CLI's codebase.
How to make it work then?

As I said, BSP is amazing. They have thought of everything.

We are grown-ups. I have enough knowledge (🙄) to run it by myself.

⚠️ Scala cool kid area 😎

We can control the running process as we want to 😊, including terminating it.

Running Java by ourselves

Extension of BSP for JVM:

- `buildTarget/jvmRunEnvironment`
 - Gives the classpath, specific Java CLI arguments, working directory, environment variables...

Do not fall in the same trap as me.

In UNIX-like, classpath directories are separated by `:`. On Windows, it is `;`...

Took me a while to find this.

Note on running time exceptions

After talking with Jędrzej, we decided to print out running time exceptions directly in the console.

Multiple solutions have been thought before including:

- Using a Java agent (issue: need to pack a .jar and configure the project for this, I prefer `Makefile` s than `build.sbt` files. Probably worked only in Java and needed some libraries)
- Modifying instrumentation to add a `Try` (issue: need to modify the good tests, without doing any mistakes)

Before (with SBT): It was still parsing the output!

 **The runner is working** 

The boring/funny part (depends on you)

UI

[Editor](#)[Build Settings](#)[Reset](#)

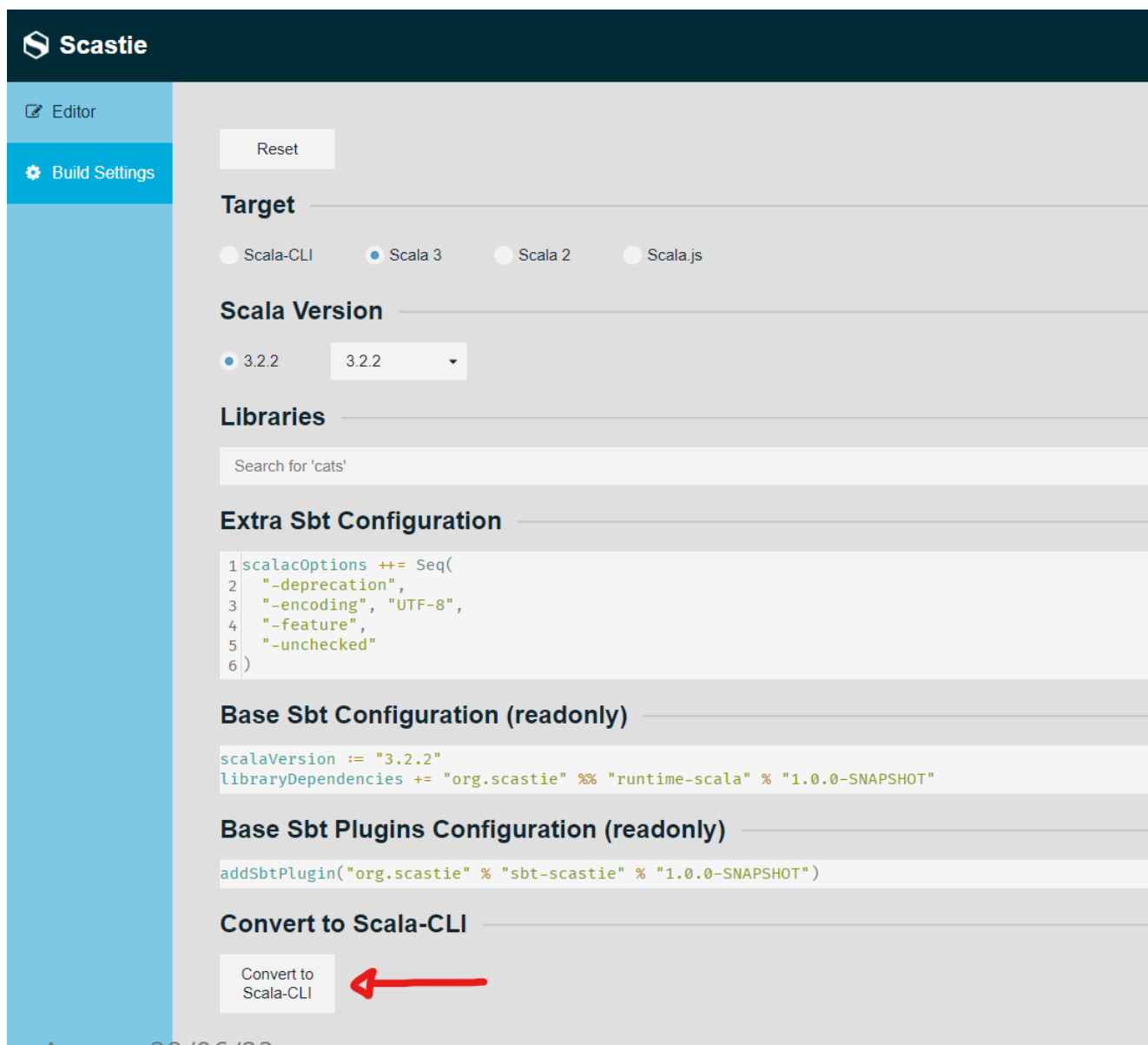
Target

[Scala-CLI](#)[Scala 3](#)[Scala 2](#)[Scala.js](#)

Scala Version

[3.2.2](#)[3.2.2](#)

Also, users can convert snippets from SBT to Scala-CLI



The screenshot shows the Scastie web application interface. On the left, there is a sidebar with two tabs: 'Editor' and 'Build Settings'. The 'Build Settings' tab is active. The main content area is divided into several sections:

- Reset**: A button to reset the settings.
- Target**: A section with four radio buttons: 'Scala-CLI', 'Scala 3' (selected), 'Scala 2', and 'Scala.js'.
- Scala Version**: A section with a radio button for '3.2.2' and a dropdown menu showing '3.2.2'.
- Libraries**: A section with a search bar containing the text 'Search for \'cats\''.
- Extra Sbt Configuration**: A section with a code editor containing the following code:

```
1 scalacOptions += Seq(  
2   "-deprecation",  
3   "-encoding", "UTF-8",  
4   "-feature",  
5   "-unchecked"  
6 )
```
- Base Sbt Configuration (readonly)**: A section with a code editor containing the following code:

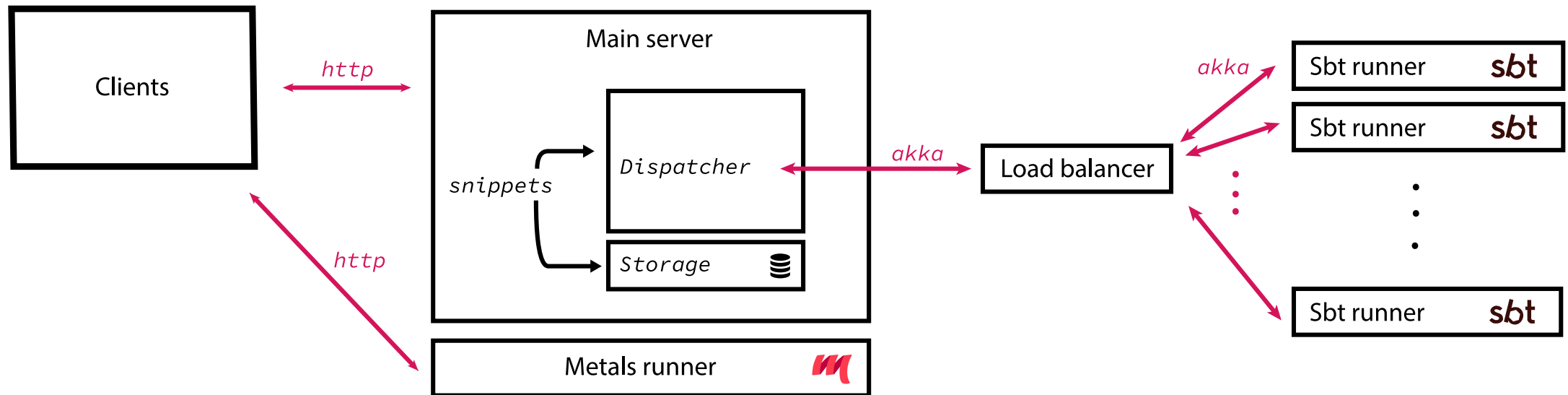
```
scalaVersion := "3.2.2"  
libraryDependencies += "org.scastie" %% "runtime-scala" % "1.0.0-SNAPSHOT"
```
- Base Sbt Plugins Configuration (readonly)**: A section with a code editor containing the following code:

```
addSbtPlugin("org.scastie" % "sbt-scastie" % "1.0.0-SNAPSHOT")
```
- Convert to Scala-CLI**: A section with a button labeled 'Convert to Scala-CLI'. A red arrow points to this button.

Interesting part again: Metals

How does it work?! How do I make it work?! *panic* :face_with_peeking_eye: 🙈

How do the client get auto-completions?



Focus on the Metals runner. It is totally independant.

How do the client get auto-completions?

- Client sends a `/isConfigurationSupported` to know whether the configuration (i.e. set of dependencies and Scala version) are compatible and if the auto-completion is possible.
- Client sends a multiple requests when hovering, etc. with the configuration.

How can we modify things so that it handles directives?

The selected solution:

- Bundle the snippet's content in `/isConfigurationSupported`
- Let the Metals runner parse the directives, do its job and return a specific `ScastieMetalsOption` that contains all the directives :)
- The client will have to give that `ScastieMetalsOption` for next requests.
- Easy life?

Metals now works!

Next steps

- Refactor the code to make it more understandable for the maintainers.
- Solve issues with BSP library that sometimes crashes with an "internal error" issue.
- Merge it if it is approved :)

But nice things do not last in life...

...like my presentation.

Thank you for listening and your attention

Thanks for Julien Richard-Foy and Jędrzej Rochala for letting me work on this project
It was a freaking good atmosphere to work on such a project. I will miss those weekly meetings on Monday with Jędrzej 🤝.

Thanks to Eugène Flesselle and Hamza Remmal for giving some tips, gave some ideas, talking about the project.

All of this made me fond of the Scala ecosystem. Maybe count me as a future Dotty contributor?

I hope the project suited your expectations, because it did suit mines.

You are now well prepared for "hell" codebases – Jędrzej, 2023