Bachelor project

Adding support for Scala-CLI on Scastie

Marwan Azuz, marwan.azuz@epfl.ch
Supervised by Julien Richard-Foy (Scala Center) and Jędrzej Rochala (VirtusLab)

Summer 2023

This is a draft. Not meant for grading or reading purposes.

This document contains a lot of clickable links, that can be used to jump on specific code or documentation. Feel free to read it on a PDF viewer instead of printing it. You are saving some trees!

Contents

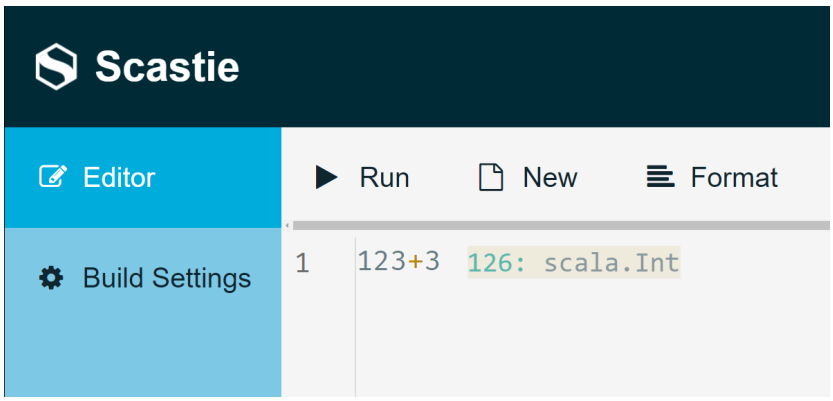
1 Introduction

1.1 Presentation of Scastie

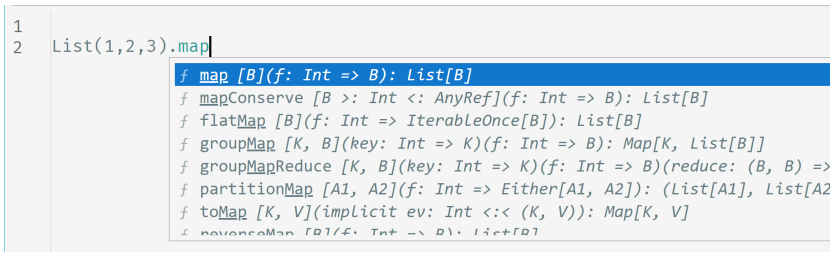
Scastie is an interactive playground for Scala in the browser, available at scastie.scala-lang.org. It offers a quick and easy way to write Scala code in an interactive manner, similar to worksheets.

Figure 1: Using Scastie to run code for demonstration purposes on the CS-206 Parallelism & Concurrency course at EPFL.

In the same manner as worksheets, it offers a preview of intermediate values :



and also features autocompletions using Metals :



1.2 Usage of Scastie ?

Scastie is great to have a worksheet right in the browser, accessible from any device. When teaching the Scala language, it can be a quite handy tool as the following tweet demonstrates :

It supports adding libraries in a user-friendly interface and also sharing snippets in a hassle-free way : perfect to demo the usage of a library or Scala feature !

1.3 The issue with Scastie

Scastie has a little flaw in how it runs snippets : when using a configuration that is not the default configuration (i.e. specific Scala version and/or libraries), it can have a lot of delay when running a snippet. To explain more in detail this problem, let's go with a quick overview of how Scastie works. Scastie is a distributed system and has different components. Here a quick visual representation of the architecture :

When an user submits a snippet to be ran, it passes through the dispatcher that sends it to the load balancer that forwards it to one of the available runners.

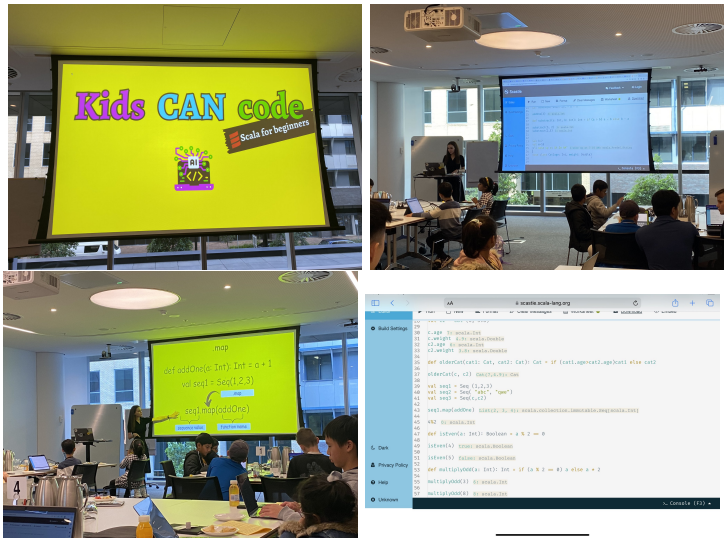


Figure 2: Tweet from @BessieIFunction learning kids Scala using Scastie

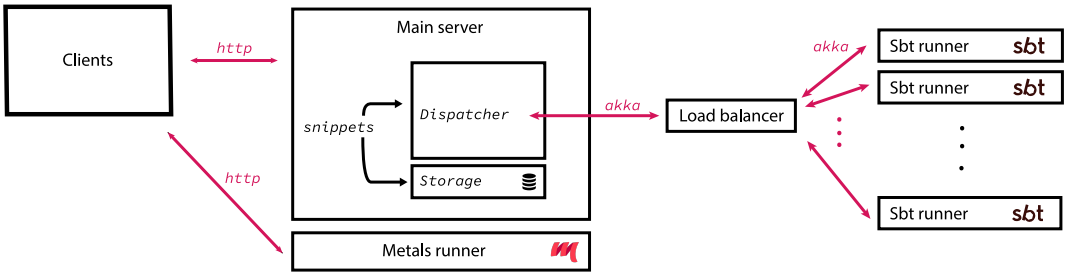


Figure 3: Quick visual overview of Scastie's architecture

One of the current issues with Scastie is how it handles libraries and is linked with how SBT handles it. On a local SBT project, one should add the library in the configuration file `build.sbt` and then reload SBT. Keep in mind that reloading SBT is an operation that takes a lot of time and makes the specific runner busy. If one user wants to use – for instance `com.lihaoyi::os-lib` –, the load balancer will try to find a runner that has the library already loaded.

More generally, one can define 'configuration': the set containing the Scala version and libraries. The load balancer will try to find a runner that has the same configuration as the required one to run a specific snippet – to avoid reloading unnecessarily a runner. If it manages to find such a runner, it forwards the execution. Otherwise, it will ask for a runner to reload with the required configuration. This creates a lot of overhead – hence, a lot of latency to the user – to execute a snippet.

The goal of this project is to replace SBT by Scala-CLI : a faster Scala runner. We can safely assume that the utterly complex Scala Build Tool represents a too much convoluted tool to be used on Scastie, which main role is to execute quick user snippets. Therefore, Scala-CLI seems to be a suitable replacement for SBT.

1.4 Scala-CLI directives

Adding Scala-CLI offers for a quick and easy way to specify how Scala should be ran. The two most important ones are :

```
//> using scala "2"

object HelloWorld extends App {
   println("Hello world!")
}
```

which defines the Scala version (here, the latest Scala 2 version.) and

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

@main def main = println("Hello world")
```

which adds the dependency `com.lihaoyi::os-lib:0.9.1`.
Let's note that Scastie must support those directives (including Metals and the Scala-CLI runner.)

1.5 Timeline of the project

The project has been decomposed into the following milestones :

1. The project works on the student's computer.
2. The Scala-CLI runner can execute Scala code.
3. The runner is correctly implemented.
4. Metals is working correctly along with dependencies.
5. The runner is connected to the website with the UI to support it.

The report will follow an almost linear approach of all the milestones achieved during this project, that has been done in the context of a mandatory semester project at EPFL.

2 The birth of the Scala-CLI runner

Multiple solutions about how to interact with Scala-CLI have been thought. Here is a list with explanations on every of these approach.
From now on, we will refer to "script" as a general Scala file: it could be a worksheet or a standalone Scala file.

2.1 The most basic runner – ever

A first way of tackling this problem would be to let Scala-CLI in charge of everything: compiling and running the script. Not even bothering to save the script to a file: pipe the script to the standard input of Scala-CLI and get the result of it. This enables for a single runner to execute multiple scripts at the same time.

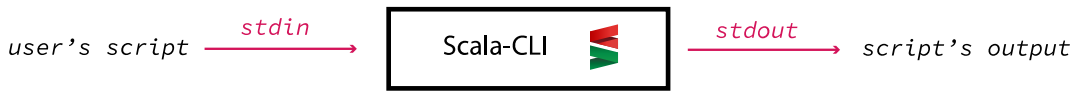


Figure 4: The most basic runner

Spawning a process, piping content to the input and listening the output can be easily done in Scala using the scala.sys.process package. Such a prototype can be found on commit `a74627d`, file `ScliActor.scala`
This solution is easy and quick. However, it is not resilient to compilation errors without extensive effort : since the output is made to be shown in a terminal for a human, managing through such a solution requires proper parsing (which will be very likely unnecessarily complicated and archaic).
Here is an example :

```
maeeen@Marwan-PC:~/test$ cat test.scala
@main def main = unknownfunction("hello")

maeeen@Marwan-PC:~/test$ scala-cli - < test.scala
Starting compilation server
Compiling project (Scala 3.2.2, JVM)
```

```
[error] stdin:1:30
[error] Not found: unknownfunction
Error compiling project (Scala 3.2.2, JVM)
Compilation failed
```

This is how Scastie handles SBT by default and it would be preferable to go with a more reliable option that offers interaction in a machine-understandable format.

A side remark : my fellow student colleague working on Scastie implements multiple files support. Choosing such a solution and stopping here would require extra modifications to the runner to add such a feature.

2.2 The Build Server Protocol at the rescue

Many builds tools offer a protocol to interact with them. One of these protocols is the BSP (short for Build Server Protocol, see the specification on build-server-protocol.github.io/docs/specification). Scala-CLI does offer a BSP server with the command line argument `bsp`.

Interacting with the build server protocol is still done with the input and output streams of the child process, but using JSON with an extra-header. There exists multiple solutions to handle the BSP protocol but the most straightforward way is to use the ch.epfl.scala::bsp4j package on Maven. A similar artefact exists for Scala (`ch.epfl.scala::bsp4s`) : however, it has not been updated after 2021 according to the Scaladex and had almost no documentation on its usage on the Build Server Protocol specification (by the time writing this report.)

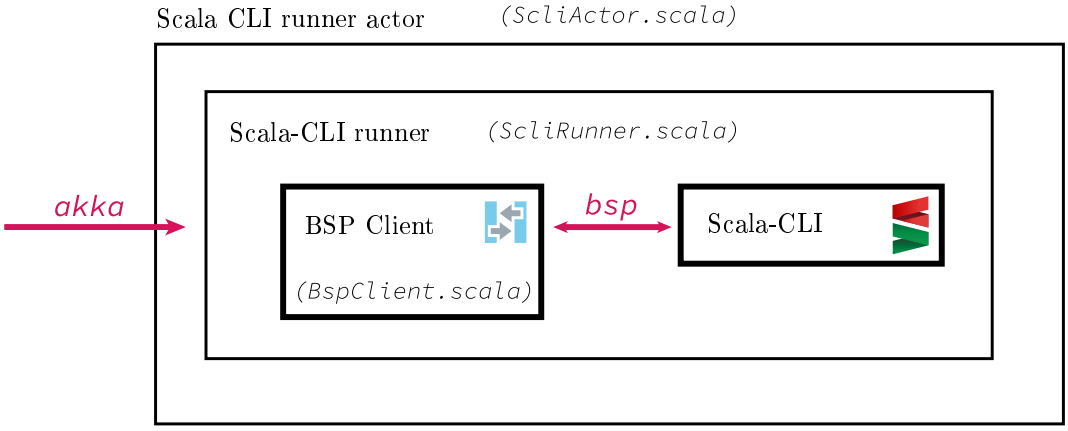


Figure 5: Architecture of the Scala-CLI runner

In this configuration, the runner spawns a Scala-CLI server in BSP mode inside a temporary folder. Afterwards, the BSP client initializes the workspace to make it ready for executions.

Then, on every script that needs to be ran, the content of the user script is written to a `main.scala` file, at the root of the project. Afterwards, the BSP client does the following tasks :

1. Reloading the workspace `workspace/reload` : to be sure that any dependencies changes is taken into consideration.
2. Get build targets' id `workspace/buildTargets` : since Scala-CLI setups a new workspace, it has a specific build id. This id has to be specified for every request afterwards. Let's note that we take the first build target which is not a test build target.
3. Compilation `buildTarget/compile` : Triggers the compilation of the workspace.

2.2.1 Running snippets: let Scala-CLI run the snippet

4. Run `buildTarget/run` : let Scala-CLI runs by itself the script

Keep in mind that Scastie has to stay resilient against annoying users to not overload unnecessarily the runners. For instance, the following script should be canceled after a timeout (e.g. 30 seconds) :

```
@main def main =
   while (true) do println("oopsie")
```

However, Scala-CLI has an issue : when scripts are run by Scala-CLI, they can not be canceled despite submitting a cancelation request. This is an issue in Scala-CLI's codebase. So another solution has to be found.

2.2.2 Running snippets: let us execute Java on our own

The main idea would be to delegate the "executing" part of the code snippet to the Scala CLI runner, and not to the Scala CLI instance.

Using the buildTarget/jvmRunEnvironment, one can get the `classpath` for the compiled user code. Launching Java is then straightforward : `java <main-class> <additional-run-settings>` where `<additional-run-settings>` contains the additional run settings given by the BSP request. Specific environment variables have to be specified according to that same BSP request, including setting the `CLASSPATH` environment variable.

However, using such a solution discards the possibility of choosing a specific JVM to run a snippet, using the `//> using jvm "<jvm>"` directive. It is however preferable to not let users chose their specific JVM instead of letting them submitting scripts that do not halt.

2.2.3 Known issues: Infinite compilation

An issue that is hard to tackle and has not been handled yet is infinite compilations. With Scala's metaprogramming, it is possible to create Scala code that never finishes to compile. Scala-CLI relies on Bloop to compile Scala code. In case of such "never-ending compilation" scripts, Bloop can not cancel the compilation : this has been forwarded as an issue on Bloop's Github, issue #2019. The selected temporary solution is to make the runner exit abruptly using `sys.exit(-1)` if the compilation did not finish in a given period. The Docker container will be aware of the fact that the runner has killed itself and needs to be restarted. Also, the problematic snippet will be logged to enable further investigations.

2.3 Instrumentation and directives

When users request a snippet to be run in **worksheet mode**, the snippet is properly instrumented before being executed. Let's talk about how it works because it created some issues.

2.3.1 Instrumentation walk through

First of all, the whole code is appended with the following :

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

It is the main class that should be executed. But, what is the `Playground` object one might ask ?

An example is better than a long paragraph explaining how it works. Suppose we have the following, rather simple, input :

```
1
```

It patches the input to this :

```
import _root_.com.olegych.scastie.api.runtime._
```

```
object Playground extends ScastieApp { private val instrumentationMap$ =
    _root_.scala.collection.mutable.Map.empty[_root_.com.olegych.scastie.api.Position,
    _root_.com.olegych.scastie.api.Render];def instrumentations$ =
    instrumentationMap$.toList.map{ case (pos, r) =>
    _root_.com.olegych.scastie.api.Instrumentation(pos, r) };
scala.Predef.locally {val $t = 1;
    instrumentationMap$(_root_.com.olegych.scastie.api.Position(0, 1)) =
    _root_.com.olegych.scastie.api.runtime.Runtime.render($t);$t}}
```

Okay, this looks complex. Let's beautify this a bit!

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

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
   }
}
```

As this is out of scope of this project, the details of implementation and design choices will not be discussed, but the main idea is clear. Every line is going to be wrapped with a `locally` definition and its result is going to be saved inside a mutable map that is going to be printed by the `Main` object, alongside with the corresponding positions before any patches.

A first issue to tackle was the hard-coded dependency : for the instrumentation to work, the library `org.scastie::runtime-scala` is needed. So, the Scala CLI runner has to prepend the instrumented code with the line `//> using lib "org.scastie::runtime-scala"`.

However, instrumenting code requires to be aware in which version of Scala the snippet is. One can not use the request `buildTarget/scalacOptions` or similar because the worksheet code is not compilable at all ! So, the Scala CLI runner has to be aware of the Scala version used by the snippet before doing any BSP request.

This has been done by checking if the directive `//> using scala "<scala-version>"` is present in the user directives, and in which case, retrieve the Scala version. If it does not find the directive or finds an invalid Scala version, the Scala CLI runner will use the default Scala version (i.e. the latest Scala 3 version).

2.4 Handling directives

Before any execution, the code given by the user is split in two. Namely user directives and user code :

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

. Directives can only appear at the top of a Scala file according to the documentation.

However, the syntax to specify directives have gone through a lot of changes since its introduction. The support of the directives will be discussed for each part of Scastie on which modifications have been made.

On the runner end, only directives specified with the most recent syntax (i.e. using these `//>` comments) will be supported. Support for the `//using` and `@using` directive could be added but since it is now deprecated, no need to handle them.

To get back on the instrumentation, the code will be instrumented without these directives and prepended before written to the `main.scala` file.

Note, attention to the reader : This comes at a surprise to no-one, but we have to take into consideration that the code sent by the user is not going to be the code that is going to be executed!

2.5 Java is confused despite knowing its classpath: which class is the Main class?

Because of instrumentation, we have now multiple classes that are now candidates for an execution!

We could either hard-code it, or make it properly.

BSP comes again to the rescue : BSP can be extended for specific build servers. If one runs a BSP server for C++, it can be extended to support additional requests (such as defining specific compile flags). In the same manner, an extension for Scala exists. The one that is interesting to us is the `buildTarget/scalaMainClasses`.

In the current implementation of the BSP client, such a request is done and the first class that is runnable is selected to be executed. If a user submits code, without worksheet mode, that does not contain a runnable class : a `"No main class found"` error is thrown and forwarded to the user.

2.6 Handling BSP notifications

Scala-CLI notifies the BSP client on multiple events. Those in which we are interested are :

- `build/logMessage` : which is sent by Scala-CLI to "ask the client to log a particular message."
- `build/publishDiagnostics` : which is sent by Scala-CLI to report compilation warnings, errors and informations, generally called diagnostics.

A first implementation of the client kept track - for a possible update in the future - for multiple compilations : however, this created some issue. To keep track of all compilations, two buffers of the types `Map[SnippetId, List[LogMessage]]` and `Map[SnippetId, List[Diagnostics]]` have been created. To keep track which snippet triggered any event, an `originId` can be specified :

Suppose that the snippet with the `snippetId =2` is sent to be executed. Every request to the BSP will be linked to the `snippetId` using `originId` which takes the form of `<snippetId>-<req. type>` (e.g. `2-compile`, `2-main-classes`). However, some diagnostics that are sent by Scala-CLI itself (and not bloop, e.g diagnostics for using a deprecated syntax of directives) did not contain this specified `originId`.

The solution that has been taken is to avoid non-necessary over complications and consider only a single compilation at a time, which is the currently implemented solution.

When the execution is finished, all of these log messages and diagnostics will be forwarded to the Scala-CLI runner, which will then forward them to the user.

2.7 Extra-feature: deprecated calls

While testing, it has been noticed that calling deprecated methods did not trigger any diagnostic about deprecation.

For example, running the following script :

```
@deprecated def hello = println("hello world! but deprecated... :'(")

@main def main = hello
```

would not trigger any diagnostic, whether the worksheet mode was enabled or disabled. But it would log the following message :

```
there was 1 deprecation warning; re-run with -deprecation for details
```

A simple solution to this was to add the flag `-deprecation` when launching Scala-CLI in BSP mode.

2.8 Summary of the Scala-CLI runner

After this very long discussion, we can summarize the Scala-CLI runner as follows :

The runner initializes a temporary directory and launches `scala-cli . bsp -deprecated` inside. It then creates the BSP client and waits for the connection to be made. Once made, the runner is ready.

On each run request, the runner will :

1. Find the Scala version of the snippet (if specified with directives) or use the default Scala version
2. Split the code in two distincts parts : the directives and the user code
3. If in worksheet mode: instrument the user code
4. Write the directives appended with the (maybe instrumented) user code to the `main.scala` file
5. Ask the BSP client to compile it :
   (a) `workspace/reload` : to reload the workspace and be sure that any dependencies changes is taken into account
   (b) `workspace/buildTargets` : to get the builds' target id.
   (c) `buildTarget/compile` : to compile the snippet.
   (d) If the compilation succeeded, `buildTarget/scalaMainClasses` : to get the main class of the snippet.
   (e) If a main class has been found, `buildTarget/jvmRunEnvironment` : to get the `classpath` to execute the snippet.
   (f) If a JVM environment has been found, return a `ProcessBuilder` that contains the process to spawn to execute the snippet.
   (g) During all the previous operations, log messages and diagnostics are buffered then returned with the result of the previous operations.
6. Forward all log messages and diagnostics
7. If the BSP runner succeeded, run the process to execute the snippet, with a timeout of 30 seconds and forward its output

At this point, we have a fully working Scala-CLI runner!
But that's not everything. We still have to make it work with Metals.

3 Integration of Scala-CLI in the architecture : UI

First of all, let us define what a Scala Target is : it is a "flavor" of Scala. For example, Scala 3 is a Scala Target and has its own definition `Scala3(scalaVersion: String)`, Scala 2 has a similar one, Scala.JS and Scala Native also have their own Scala Target.

To make a transition possible, a new Scala Target has been defined: `ScalaCli` which is a specific Scala Target that contains the Scala version (let's note that it can be undefined, especially in the case where it is not linked to code).

With that in mind, some extra-settings have been added on the UI, including the new Scala Target :

Figure 6: The new target

Since time allowed for extra work, a button has been set-up to allow the user to convert almost seamlessly (under the condition that no extra SBT configuration has been added) from SBT to Scala-CLI :

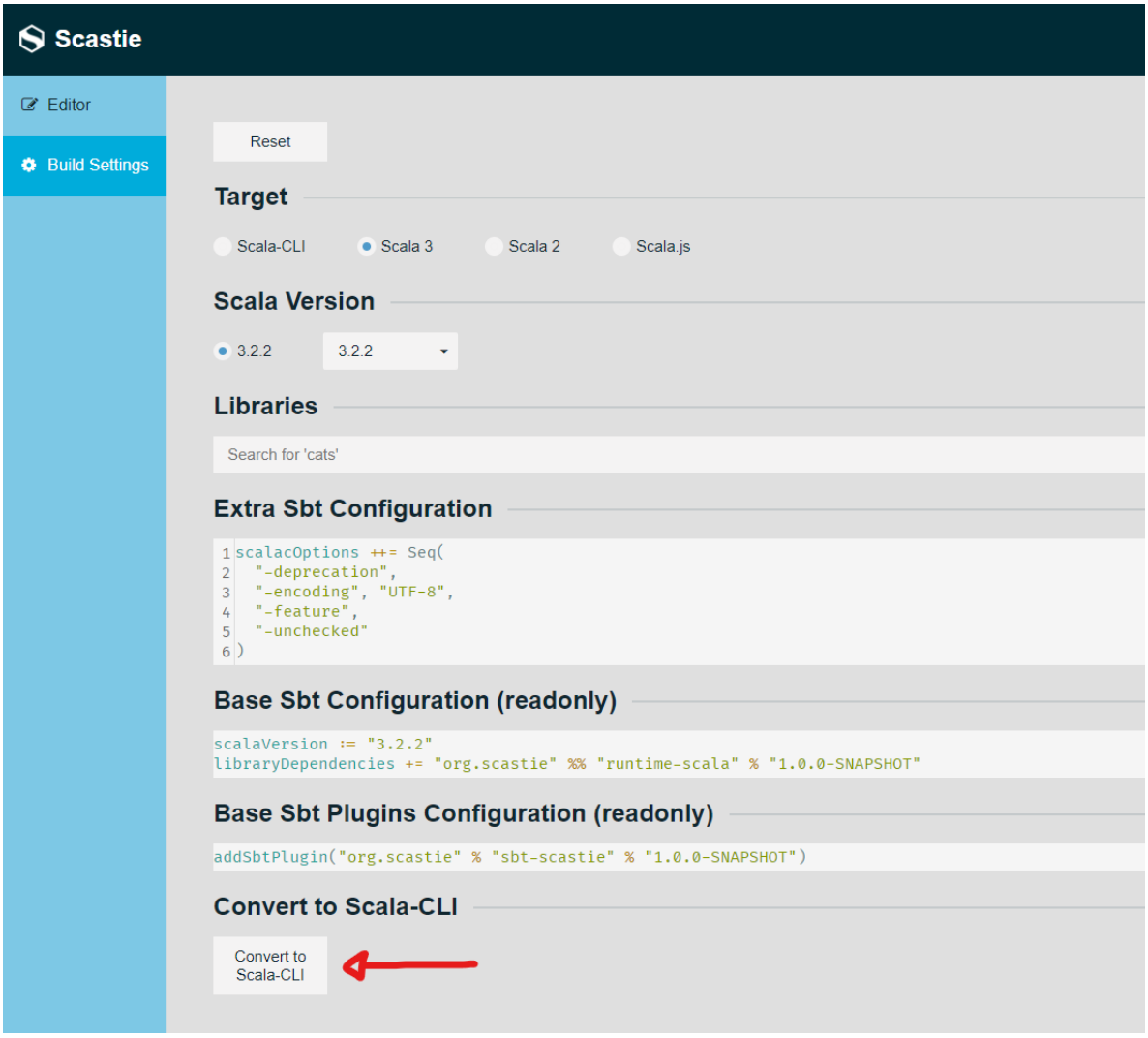


Figure 7: The convert button

Its behavior is not really surprising :

- Checks whether any SBT extra-configuration has been added. If yes, abort.
- Change the Scala target to the Scala CLI target.
- Prepend the code with the following :
  - `//> using scala "<scala-version>"` with `<scala-version>` being the Scala version of the snippet.
  - For each dependency :
    `//> using dep "<dependency.groupId>::<dependency.artifact>::<dependency.version>"`
    with `<dependency.groupId>`, `<dependency.artifact>` and `<dependency.artifact>` being respectively the dependency's `groupId`, `artifact`'s name and `version` the version.
  - `"========"` to separate the directives from the user code.

4 Metals support

As shown in the figure 1.3, the Metals server is totally independent from the Scastie server and the runners. Therefore, we need to make Metals understand the Scala-CLI directives.

Let note that the auto-completion of the `//> using dep` is already there by default on Metals when worksheet mode is disabled. To understand why, we need to look at the Metals architecture.

4.1 Introduction to the Metals server

The Metals runner uses ScalaMeta's presentation compiler. Since instantiating a presentation compiler is a costly operation, Metals keeps a single instance of it and reuses it for every request : there is no idea of a session, every request does not contain session details and the only thing that matters is the configuration of the snippet.

Here is a quick overview of how the Scastie client and the Metals server interact, hence a quick list of the endpoints that matter :

- `/isConfigurationSupported` : sent out by the client containing a `ScastieMetalsOption(dependencies: Set[ScalaDependency], scalaTarget: ScalaTarget)`
- `/complete` : sent out by the client containing a `LSPRequestDTO(options: ScastieMetalsOption, offsetParams: ScastieOffsetParams)`
- and other similar requests to get documentation where every of them contains a `ScastieMetalsOption` with a `ScastieOffsetParams` to give the cursor's position.

When a client wants to make any of these requests, it should first of all probe the Metals runner to know whether the current configuration of the user is supported ; then it can make the `/complete` request or similar. Note that in fact the `/isConfigurationSupported` is not mandatory, but it is a good practice to actually show to the user that the configuration is not supported.

4.2 Worksheet mode?

Let's recall that this auto-completion provided by ScalaMeta's presentation compiler is similar to the one that one may have in an IDE. With this supposition, the presentation compiler expects a totally valid Scala project's code.

Hence, in worksheet mode, it needs proper instrumentation. In this case, it is easier than before. It consists of only adding the following (and doing proper modifiations to the offset which is adding the indentation offset and the length of `object worksheet {\n`) :

```
object worksheet {
   <user code>
}
```

With this in mind, we can understand why the `//> using dep` was working without worksheet mode. When worksheet mode is disabled, the said instrumentation is then disabled and auto-completes as desired.

The chosen solution to split the user code as done in the Scala-CLI runner's instrumentation part.

```
<user directives>
object worksheet {
   <user code>
}
```

When the user is making such a request, we need to appropriately take into consideration the given offset of the cursor. If the cursor is located in the directives, we do nothing. Otherwise, we need to appropriately offset it by `object worksheet {\n`.

One may read the code that handles this on the following file metals-runner/src/.../DTOExtensions.scala

4.3 Adding auto-completion of dependencies

To enable such an auto-completion, dependencies should be parsed and resolved at some point. Multiple solutions have been thought :

1. Parse the directives and resolve dependencies client-side

2. Parse the directives and resolve dependencies on the Metals runner's end and make the client aware of it

3. Parse the directives and resolve dependencies on the Scala-CLI's runner end and make the client aware of it, then make appropriate requests to the Metals runner with these information.

The first choice is something that costs too much to the client to do. We want it to be lightweight so that it can run on any device. Keep in mind that we have to make it "refreshable" (i.e. when adding a dependency, either client or servers should be aware of such changes and give proper auto-completions).

The last choice is too complex and involves creating new transmissions channels amongst totally independent components.

One can try to make it simple and the second solution has been chosen : parse the directives server side, check whether all the dependencies are compatible and return a `ScastieMetalsOption` with the Scala version and the dependencies. The client will have to send the `ScastieMetalsOption` in every request so that every call that worked using traditional targets will work with this specific `ScastieMetalsOption`. This will be discussed further, in the next subsection :).

4.4 Parsing the directives

Handling directives is simply about converting a `ScastieMetalsOption` that has Scala-CLI as a target to a `ScastieMetalsOption` that does not have Scala-CLI as a target. The code that does this is located inside the metals-runner/. . . /ScalaCliParser.scala file.

VirtusLab provides a library `using_directives` to parse these directives, available on Github and Maven. This library has been used to quickly parse the directives and create a proper `ScastieMetalsOption` from the directives !

Let note that in the case where no directives are provided, the latest Scala 3 version is used and no dependencies are added.

4.5 Updating the client with the new working of ScastieMetalsOption

As well, proper modifications on the so-called `InteractiveProvider` (which is the component that handles Metals) have been done to make it aware of the new `ScastieMetalsOption` and the modified file can be consulted at client/. . . /InteractiveProvider.scala.

The details do not really matter and neither are interesting.

4.6 And what about a refresh?

If the user changes directives, we should be able to either :

1. Refresh the auto-completion automatically

2. Show to the user that auto-completions requires a refresh with the changed directives

The first case requires to have a way to know that the directives have changed. This can be done by comparing the directives with the previous ones. If they are different, we can refresh the auto-completion. This is not a good solution because it requires constant communication with the Metals runner and we would like to keep used resources on the Metals runner's end to a minimum (remember that only a single instance of the Metals runner is running for all the users !).

Hence, the selected solution is the second one. When the user changes the directives, we show to the user a button to refresh directives. The user can then click on the said button to refresh the auto-completion. It simply restarts the directives.

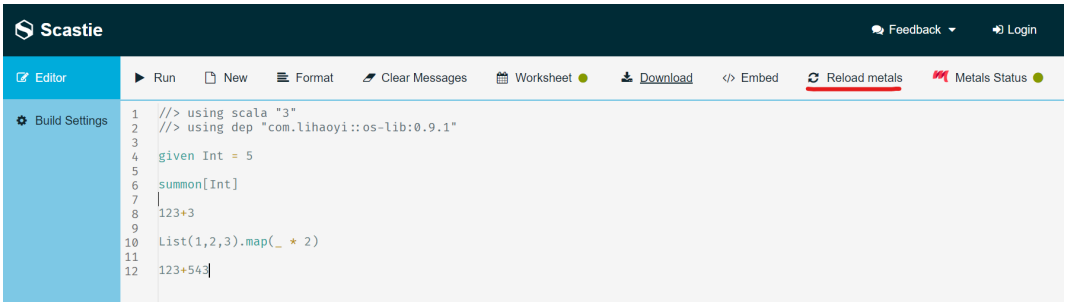


Figure 8: Refresh directives button

Note : details of implementation are not discussed because it is 1. not relevant in the context of this report, 2. talks about how ReactJS works and handles components which is not the purpose of this report and also is a bit of hell. The focus of the author is to explain the given user experience and the technical details that are relevant to the project. If any question occur about the implementation of this specific part, the author (me) will be happy to address them.

5 Conclusion and results?

With all of these minimal modifications, Scastie is now able to run Scala-CLI snippets on Scala-CLI which includes auto-completion of dependencies and Scala versions. The user experience is not perfect but it is a good start, it is a good proof of concept and showcases the potential of Scala-CLI.

TODO: Add some benchmarks

6 Special thanks

A real and grateful thanks to :

- Jędrzej Rochala for all the support, either technical or emotional and the nice talks we had about the Scala language that will mark my life forever
- Julien Richard-Foy and also again Jędrzej Rochala for the trust they gave me to work on this project (despite this being originally a master project !)
- VirtusLab for Scala-CLI and the using_directives library

It has been a pleasure to explore the Scala language in this manner and I hope that this project will be useful to the community !