
Scala Full Stack
With ZIO, Tapir && Laminar
(https://github.com/cheleb/zio-scalajs-laminar.g8)

Scala Full Stack

LLVM

↑

Scala Native

|

Scala

Scala JVM

<-- Scalac ---

--- ScalaJS -->

SJIR --->

↑

|

Ono @ ledger.fr

Olivier NOUGUIER / software engineer

- @cheleb on github
- oNouguier on twitter

Cryptocurrency hardware wallet leader worldwide.

- Founded in 2014
- Headquarters in Paris
- Offices in London, Grenoble && Montpellier

What is Ledger?

We secure both infrastructure, assets and solutions for cryptocurrencies and blockchain applications.

The creator of the world's most popular hardware wallet familly.

- Flex
- Stax
- Nano S
- Nano X

Are all products of Ledger.

Ledger is hiring, in almost all positions/stacks, check out our job offers (https://www.ledger.fr/jobs/).

Scala Full Stack the easy way

Why this talk?

- I ❤️ Web apps for 25+ years aka cgi-bin era
- I ❤️ Scala for 12+ years
- I ❤️ ScalaJs since v0.6
- I ❤️ Effect system 5+ years

Yet in another dimension…

- Angular#, React, vus, ember …
- npm, grunt, yarn …

ZIO Rite of Passage

From Rock the JVM.

- Scala Backend
- ScalaJs
- ZIO, Tapir, Laminar
- Postgres, ChatGPT, Stripe

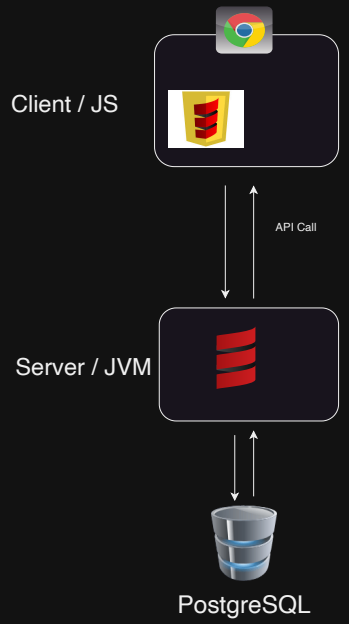
❤️ Scala definitively Rocks the JVM

😴 Setup new project

😔 Starting dev environment

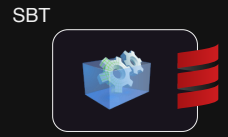
Agenda

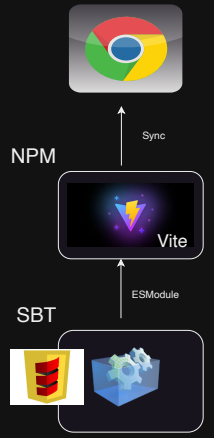
```
object ScalaFullStack:

  object SetUp

  object Projects

  object Deployement
```



Setup

- Build tools
  - NPM
    - Node Package Manager.
    - Vite: Hot reload of UI.
  - SBT
    - Scala JS
    - Scala JVM
- Docker
- Kubernetes





Setup / Demo

Dozen of tools to setup ...

let's automate it with a template

```
sbt new cheleb/zio-scalajs-laminar.g8 --name=scalaZio-fullstack-demo
```

```
code scalaZio-fullstack-demo
```

VSCode / Metals 🤘

Task automation with `.vscode/tasks.json` and `launch.json`

```
{
  "version": "2.0.0",
  "tasks": [
    {
       "label": "demo",
        "runOptions": {
            "runOn": "folderOpen"
        },
        "dependsOrder": "sequence",
        "dependsOn": [
            "setup",
            "runDemo"
        ],
        "problemMatcher": [],
        "group": {
            "kind": "build"
        }
    }
  ]
}
```

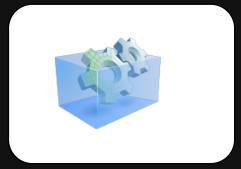
One project to rule them all

```
// Cross project support, to spread project resources between js and jvm world
addSbtPlugin("org.portable-scala" % "sbt-scalajs-crossproject" % "1.3.2")
// Scala.js support
addSbtPlugin("org.scala-js" % "sbt-scalajs"        % "1.17.0")
addSbtPlugin("org.scala-js" % "sbt-jsdependencies" % "1.0.2")
// Scala.js bundler
addSbtPlugin("ch.epfl.scala" % "sbt-scalajs-bundler"     % "0.21.1")
addSbtPlugin("ch.epfl.scala" % "sbt-web-scalajs-bundler" % "0.21.1")
// TypeScript support
addSbtPlugin("org.scalablytyped.converter" % "sbt-converter" % "1.0.0-beta44")
// Static file generator
addSbtPlugin("org.playframework.twirl" % "sbt-twirl" % "2.0.5")
// Will reStart server on code modification.
addSbtPlugin("io.spray" % "sbt-revolver" % "0.10.0")
```

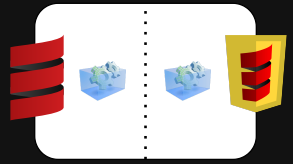
Project Structure / SBT

CrossProject shared

SBT



server



client

SBT Cross Project

- plugins.sbt

```
addSbtPlugin("org.portable-scala" % "sbt-scalajs-crossproject" % "1.3.2")
```

- build.sbt

```
lazy val shared: CrossProject = crossProject(JSPlatform, JVMPlatform)
  .crossType(CrossType.Pure)
  .disablePlugins(RevolverPlugin)
  .in(file("modules/shared"))
  /* [...] */
  .settings(publish / skip := true)

lazy val sharedJvm: Project = shared.jvm
lazy val sharedJs: Project  = shared.js
```

SBT Cross Project

Shared

```
lazy val shared: CrossProject = crossProject(JSPlatform, JVMPlatform)
// [...]
lazy val sharedJvm: Project = shared.jvm
lazy val sharedJs: Project  = shared.js
```

Server

```
lazy val server = project
  .settings( /* [...] */ )
  .dependsOn(sharedJvm)
```

Client

```
lazy val client = project
  .enablePlugins(ScalaJSPlugin)
  .settings( /* [...] */ )
  .dependsOn(sharedJs)
```

Setup / Backend

Setup / Frontend / NPM / package.json

```json
{
  "name": "scalaZio-fullstack-demo",
  "private": true,
  "version": "0.0.1",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "license": "MIT",
  "dependencies": {
    "@ui5/webcomponents": "2.1.0",
    "@ui5/webcomponents-fiori": "2.1.0",
    "@ui5/webcomponents-icons": "2.1.0",
    "chart.js": "2.9.4"
  },
  "devDependencies": {
    "@scala-js/vite-plugin-scalajs": "^1.0.0",
    "vite": "^5.4.9",
    "typescript": "5.6.3",
    "@types/chart.js": "2.9.29"
  }
}
```

Setup / Frontend / NPM / vite.config.js

```
import { defineConfig } from "vite";
import scalaJSPlugin from "@scala-js/vite-plugin-scalajs";

export default defineConfig({
    plugins: [scalaJSPlugin({
        // path to the directory containing the sbt build
        // default: '.'
        cwd: '../..',

        // sbt project ID from within the sbt build to get fast/fullLinkJS from
        // default: the root project of the sbt build
        projectID: 'client',

        // URI prefix of imports that this plugin catches (without the trailing ':')
        // default: 'scalajs' (so the plugin recognizes URIs starting with 'scalajs:')
        uriPrefix: 'scalajs',
    })],
    build: {
        sourcemap: true,
    }
});
```

Setup / Frontend

```scala
object ScalaFullStack:

  object Frontend

  object Backend
```

Shared

What can be shared between the client and the server?

- Payloads
- Business logic
- Validation
- Error handling

And …

REST API definition

Tapir

Tapir stands between Http transport and effect or direct style.

Http server

- ZIO-HTTP
- HTTP4S
- Netty
- Play
- ...

Effect or direct style

- ZIO
- Cat Effects
- Future
- Direct style
- ...

ZIO 101

ZIO is a library for asynchronous and concurrent programming in Scala.

Simplified, ZIO is a data type that represents a computation:

```
trait ZIO[-R, +E, +A]
```

- that may require an environment of type `R`
- that may fail with an error of type `E`
- that may succeed with a value of type `A`

ZIO 101

A simple mental model is to think of ZIO as a function:

```
type ZIO[R,E,A] = R => Either[E, A]
```

Many aliases are provided for common use cases:

```
type IO[+E, +A]   = ZIO[Any, E, A]         // Succeed with an `A`, may fail with `E`         , no requirements.
type Task[+A]     = ZIO[Any, Throwable, A] // Succeed with an `A`, may fail with `Throwable`, no requirements.
type RIO[-R, +A]  = ZIO[R, Throwable, A]   // Succeed with an `A`, may fail with `Throwable`, requires an `R`.
type UIO[+A]      = ZIO[Any, Nothing, A]   // Succeed with an `A`, cannot fail               , no requirements.
type URIO[-R, +A] = ZIO[R, Nothing, A]     // Succeed with an `A`, cannot fail               , requires an `R`.
```

Tapir 101 by SoftwareMill

First step is to define the API endpoints as values.

```
//                                  In      Error     Out
//                                   ☟        ☟        ☟
val createEndpoint: PublicEndpoint[Person, Throwable, User, Any] = ???
```

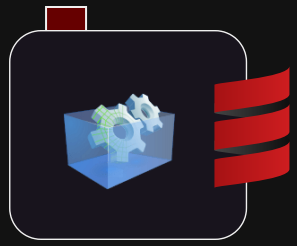
From this definition, Tapir can generate:

- OpenAPI documentation
- Http server squeleton
- Sttp client

```
POST /person HTTP/1.1
{
    "email": "john.doe@foo.bar",
    "password": "notsecured",
    [ ... ]
}
```

Tapir / HTTP Server

After a bunch of imports, we can implement the server side of the API.



Given:

```
trait PersonService:
  def register(person: Person): Task[User]
```

Controller implementation:

```
class PersonController private (personService: PersonService, jwtService: JWTService)
```

Tapir / HTTP Server

```scala
class PersonController private (personService: PersonService, jwtService: JWTService)
    extends BaseController
    with SecuredBaseController[String, UserID](jwtService.verifyToken) {

  val create: ServerEndpoint[Any, Task] = PersonEndpoint.create
    .zServerLogic:
      personService.register

  val login: ServerEndpoint[Any, Task] = PersonEndpoint.login.zServerLogic { lp =>
    for {
      user  <- personService.login(lp.login, lp.password)
      token <- jwtService.createToken(user)
    } yield token
  }

  val profile: ServerEndpoint[Any, Task] = PersonEndpoint.profile.securedServerLogic { userId => withPet =>
    personService.getProfile(userId, withPet)
  }

  val routes: List[ServerEndpoint[Any, Task]] =
    List(create, login, profile)
}
```

Tapir / HTTP Server

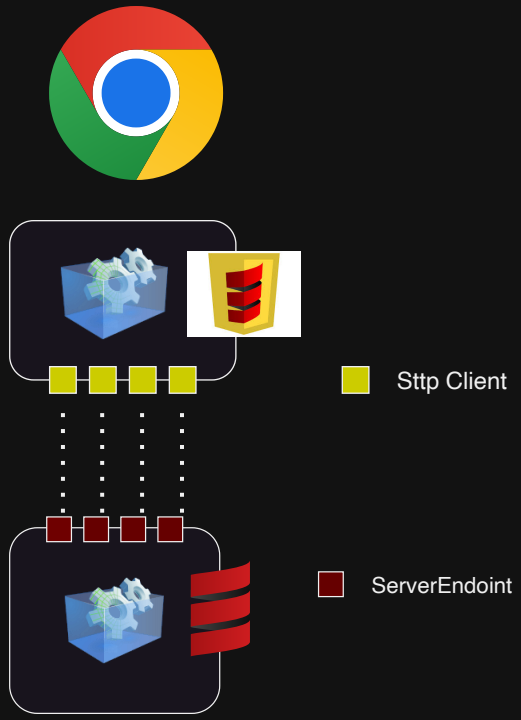
```
object PersonController:
  def makeZIO: URIO[PersonService & JWTService, PersonController] =
    for
      jwtService    <- ZIO.service[JWTService]
      personService <- ZIO.service[PersonService]
    yield new PersonController(personService, jwtService)
```

Tapir / HTTP Server

```
private val program: RIO[FlywayService & PersonService & JWTService & Server, Unit] =
    for {
      _ <- runMigrations
      _ <- server
    } yield ()
```

Mermaid

Tapir / Client - Server



Tapir / Sttp Client

In the same way, we can implement the client side of the API.

And we want to process the result in the UI.

- stay in the Scala world
- use a type-safe API
- use reactive stuff both http client and UI side
- with a minimal of boilerplate

Tapir Client Side

With another bunch of imports, we can implement the client side of the API, in one liner:

Let say we have a Person instance:

```
val person = Person("john.doe@foo.bar", "notsecured")
```

- `emitTo` is another extension method, we will detail shortly
- Question: there is another extention in action here, where ?

Tapir 101

```
//                                  In      Error     Out
//                                   ☟        ☟        ☟
val create: PublicEndpoint[Person, Throwable, User, Any] = baseEndpoint
    .name("person")
    .post                 // POST method
    .in("person")         // Endpoint path is /person
    .in(
      jsonBody[Person]    // Request body is JSON-encoded Person
    )
    .out(jsonBody[User])  // Response is JSON-encoded User
```

Under the hood - Http Request

Inspired from Rock The JVM courses/blog/videos ZIO rite of passage.

```
import dev.cheleb.ziolaminartapir.*
```

Et voilà, you can now use Tapir in a reactive way, in your ScalaJS project.

```
// This import brings bunch of extension to Endpoint and ZIO.
```

```
extension [I, E <: Throwable, O](endpoint: Endpoint[Unit, I, E, O, Any])
```

Under the hood - Http Request

Endpoint —> ZIO.

```
private[ziolaminartapir] abstract class BackendClient(
    backend: SttpBackend[Task, ZioStreamsWithWebSockets],
    interpreter: SttpClientInterpreter
):

  private[ziolaminartapir] def endpointRequestZIO[I, E <: Throwable, O](
        baseUri: Uri,
        endpoint: Endpoint[Unit, I, E, O, Any]
    )(
        payload: I
    ): ZIO[Any, Throwable, O] =
      backend
        .send(endpointRequest(baseUri, endpoint)(payload))
        .map(_.body)
        .absolve
```

Under the hood - Http Request

Endpoint --> Request

```
private[ziolaminartapir] abstract class BackendClient(
    backend: SttpBackend[Task, ZioStreamsWithWebSockets],
    interpreter: SttpClientInterpreter
):

  private[ziolaminartapir] def endpointRequest[I, E, O](
      baseUri: Uri,
      endpoint: Endpoint[Unit, I, E, O, Any]
  ): I => Request[Either[E, O], Any] =
    interpreter.toRequestThrowDecodeFailures(endpoint, Some(baseUri))
```

Under the hood - Http Request

```
Endpoint --> I --> RIO[SameOriginBackendClient, O]
```

We need antoher extension method to run the ZIO, and process the result in the UI.

```
extension [E <: Throwable, A](zio: ZIO[SameOriginBackendClient, E, A])
```

Under the hood - Http Request

There are a few more extension methods:

- Different origin client
- Bearer token support (local storage)

```
PersonEndpoint.profile(false).emitTo(userBus)
```

For the curious, the `profile` endpoint is defined as:

```
val profile: Endpoint[String, Boolean, Throwable, (User, Option[Pet]), Any]
```

Under the hood - Http Request

```scala
extension [I, E <: Throwable, O](endpoint: Endpoint[String, I, E, O, Any])
  // Call the secured endpoint with a payload, and get a ZIO back.
  @targetName("securedApply")
  def apply[UserToken <: WithToken](
      payload: I
  )(using session: Session[UserToken]): RIO[SameOriginBackendClient, O] =
    ZIO
      .service[SameOriginBackendClient]
      .flatMap(_.securedEndpointRequestZIO(endpoint)(payload))

  // Call the secured endpoint with a payload on a different backend than the
  def on[UserToken <: WithToken](baseUri: Uri)(
      payload: I
  )(using session: Session[UserToken]): RIO[DifferentOriginBackendClient, O] =
    ZIO
      .service[DifferentOriginBackendClient]
      .flatMap(_.securedEndpointRequestZIO(baseUri, endpoint)(payload))
```

Under the hood - Http Request

No boilerplate, no magic, just Scala.

ZIO Laminar Tapir

- Endpoint enhancement
- Marshalling/unmarshalling
- Error handling
- Reactive
- Type-safe
- Minimal boilerplate

```
PersonEndpoint //
          .create(personVar.now())
          .emitTo(userBus, errorBus)
```

Tapir Sttp

Both server and client side are implemented with Tapir and Sttp.

- Tapir: API definition
- Sttp as HTTP client: One line to run the request
- Sttp as backend ZIO: One line handle the request

In the mean time:

- Json marshalling/unmarshalling
- Error handling
- Security (Bearer token)
- Reactive

Laminar: 100% Scala, Reactive UI

```
import com.raquo.laminar.api.L._

val app = {
  val name = Var("World")
  div(
    input(
      placeholder := "What's your name?",
      onInput.mapToValue --> name.writer
    ),
    child.text <-- name.signal
                      .map(name => s"Hello, $name!")
  )
}
```

Basicaly Laminar provides reactive components:

- `Var` - a mutable value
- `Signal` - a read-only value that updates when its dependencies change
- `EventStream` - a stream of events
- `EventBus` - a stream of events that can be emitted to

And 2 reactive operators:

- `-->` Write to a Laminar reactive element
- `<--` Read from a Laminar reactive element

In action
https://demo.laminar.dev/app/basic/hello
Demo
Demo signup page

```
val personVar = Var(Person("John", "john.does@foo.bar", Password("notsecured") ))
val userBus  = EventBus[User]()
val errorBus = EventBus[Throwable]()
div(
  h1("Signup"),
  div(
    styleAttr := "float: left;",
    // The form is generated from the case class
    personVar.asForm,
  ),
  div(
    styleAttr := "max-width: fit-content; margin:1em auto",
    Button(
      "Create",
      disabled <-- personVar.signal.map(_.errorMessages.nonEmpty),
      onClick --> { _ =>
        PersonEndpoint
          .create(personVar.now())
          .emitTo(userBus, errorBus)
      }
    )
  )
)
```

Under the hood - Laminar Form Derivation

No boilerplate, no magic, just Scala.

Laminar Form Derivation with Magnolia

- Case class to form
- Databinding
- Validation
- Error handling

```
import import dev.cheleb.scalamigen.*

case class Person(name: String, email: String, password: Password)

val personVar = Var(Person("John", "john.does@foo.bar", Password("notsecured") ))

personVar.asForm
```

Scalablytyped

Scalablytyped (scalablytyped.dev) is a tool that generates Scala.js facade for TypeScript definitions.

Then this facade can be used in ScalaJS code.

In the template, Scalablytyped is used to generate facade for Chart.js.

Add result is used in the Scalablytyped demo.

Production deployement

In this setup, the frontend is served by the backend, webassets are deployed as webjars.

```
MOD=prod sbt -mem 4096 server/run
```

- Standalone Jar
- Docker
- Kubernetes

```
sbt new cheleb/zio-scalajs-laminar.g8  --name=zio-laminar-demo-k8s --githubUser=cheleb --with-argocd=true --version=0.0.

code zio-laminar-demo-k8s
```

Take over / What next?

In the box

Oneliners

- scafolding
- Form derivation
- Scalablytyped
- Http client
- Docker deployement
- K8s deployement
  - CD ArgoCD
  - Image updater

Next

- Testing: Testcontainers, ZIO Test
- Observability: ZIO Telemetry, ZIO Tracing
- GraphQL: Caliban
- WebSockets
- Security: OTP, OAuth, JWT
- WASM
- µServices: ZIO gRPC, ZIO-Pravega
- Resiliency(?): ZIO Circuit Breaker, ZIO Rate Limiter
- Service Mesh: Istio
- CI: GitHub Actions

Resources

- ScalaJS by ScalaCenter / Sebastien Doeraene
- SBT
- Scalablytyped

Depencencies:

- ZIO by Ziverge
- Laminar by raquo
- Tapir by SoftwareMill

Tools:

- Vite
- Docker
- VSCode
- Metals

Mentors:

- RTJVM by Daniel Ciociarlan
- Incredible Kit Langton