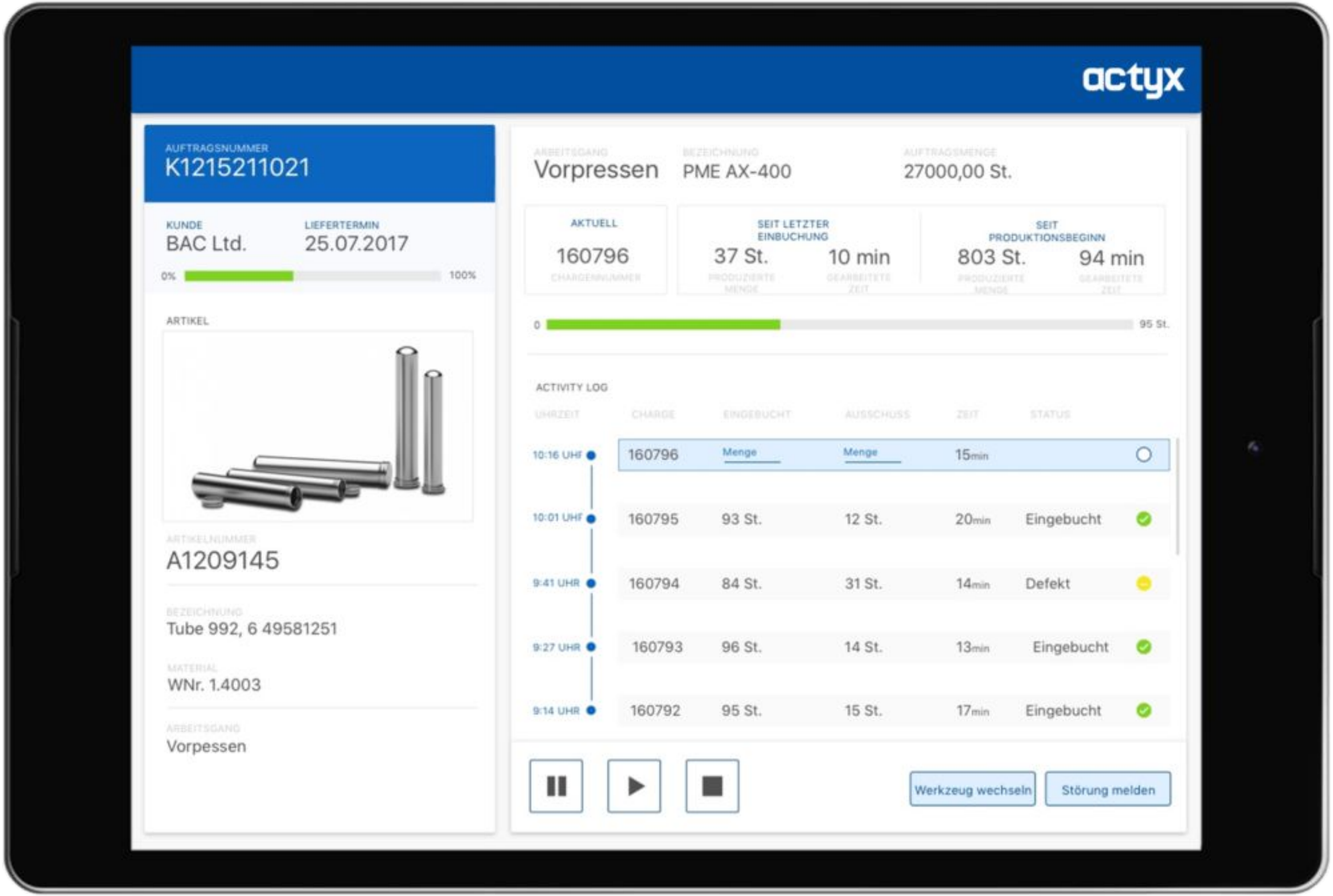
Real serverless with Akka CRDTs and IPFS

@gosubpl

actyx



AUFTRAGSNUMMER
K1215211021

KUNDE
BAC Ltd.

LIEFERTERMIN
25.07.2017

0% 100%

ARTIKEL

ARTIKELNUMMER
A1209145

BEZEICHNUNG
Tube 992, 6 49581251

MATERIAL
WNr. 1.4003

ARBEITSGANG
Vorpessen

ARBEITSGANG
Vorpressen

BEZEICHNUNG
PME AX-400

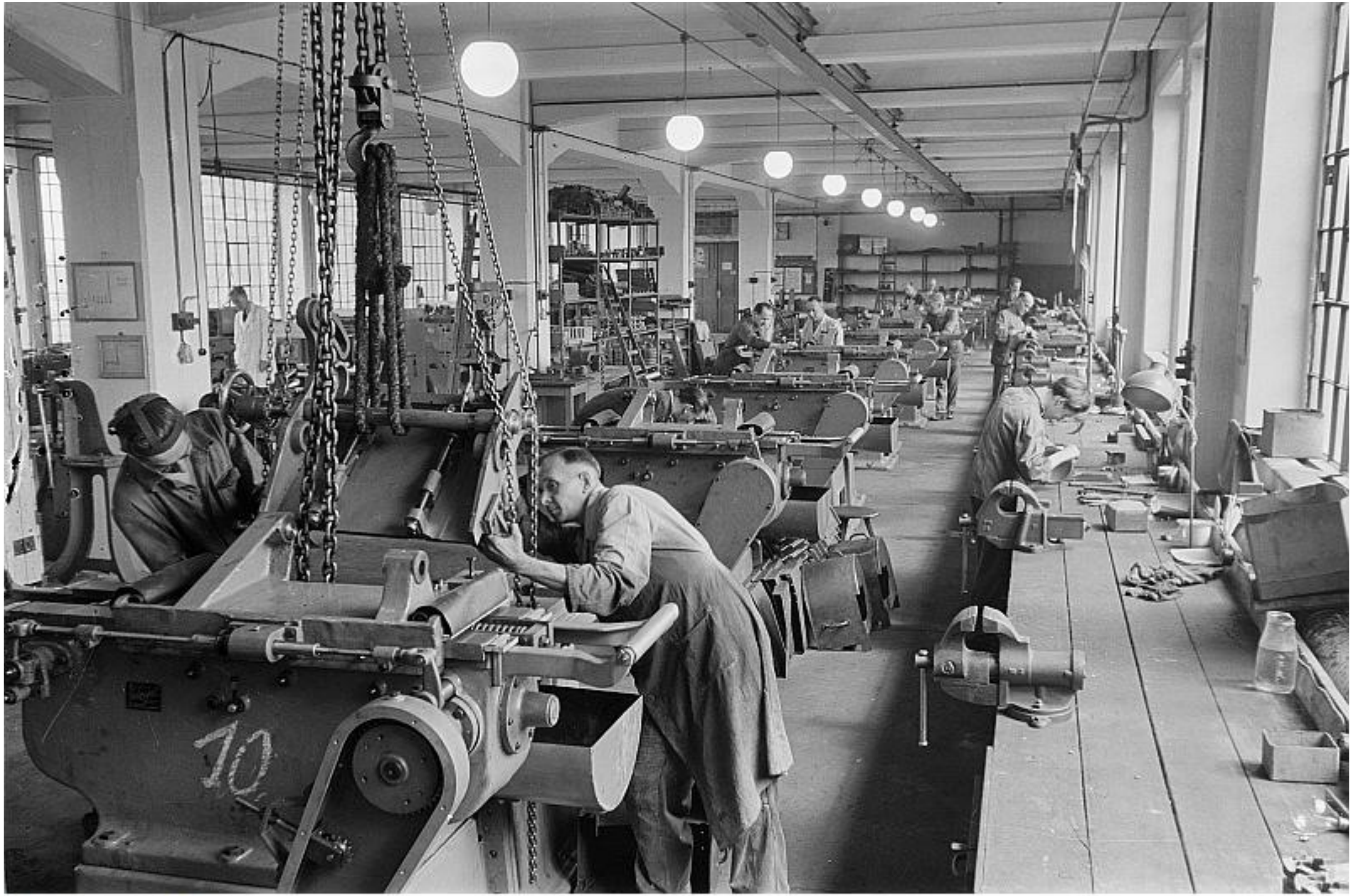
AUFTRAGSMENGE
27000,00 St.

AKTUELL	SEIT LETZTER EINBUCHUNG		SEIT PRODUKTIONSBEGINN	
160796	37 St.	10 min	803 St.	94 min
CHARGENNUMMER	PRODUZIERTE MENGE	GEARBEITETE ZEIT	PRODUZIERTE MENGE	GEARBEITETE ZEIT

0 95 St.

ACTIVITY LOG

UHRZEIT	CHARGE	EINGEBUCHT	AUSSCHUSS	ZEIT	STATUS	
10:16 UHR	160796	Menge	Menge	15min		○
10:01 UHR	160795	93 St.	12 St.	20min	Eingebucht	✓
9:41 UHR	160794	84 St.	31 St.	14min	Defekt	●
9:27 UHR	160793	96 St.	14 St.	13min	Eingebucht	✓
9:14 UHR	160792	95 St.	15 St.	17min	Eingebucht	✓



Quelle: Deutsche Fotothek

serverless

server · less

job · less

job · less

From *Collins Cobuild Dictionary*

“Someone who is jobless does not have a job, although they would like one. “

sth·**less**

Without that something...

server · less

By Dmitry Nosachev - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=46899884

client only, without a designated central “server” peer-to-peer

INTERNET PROTOCOL

PROTOCOL SPECIFICATION

September 1981

Peer-to-peer technologies

why ?

CC-BY-SA 2.0 Brad Gilette

#1

ability to operate despite failure

(maybe in degraded mode)

#2
ability to operate in a hostile environment (flaky wifi)

Let’s write an app to help factory workers complete supply of raw materials necessary to produce some product

Order picking

Raw material information:

```
{ “article”: “glue”,
  “id”: “GX123”,
  “quantity”: 50,
  “complete”: false }
```

Raw material information:

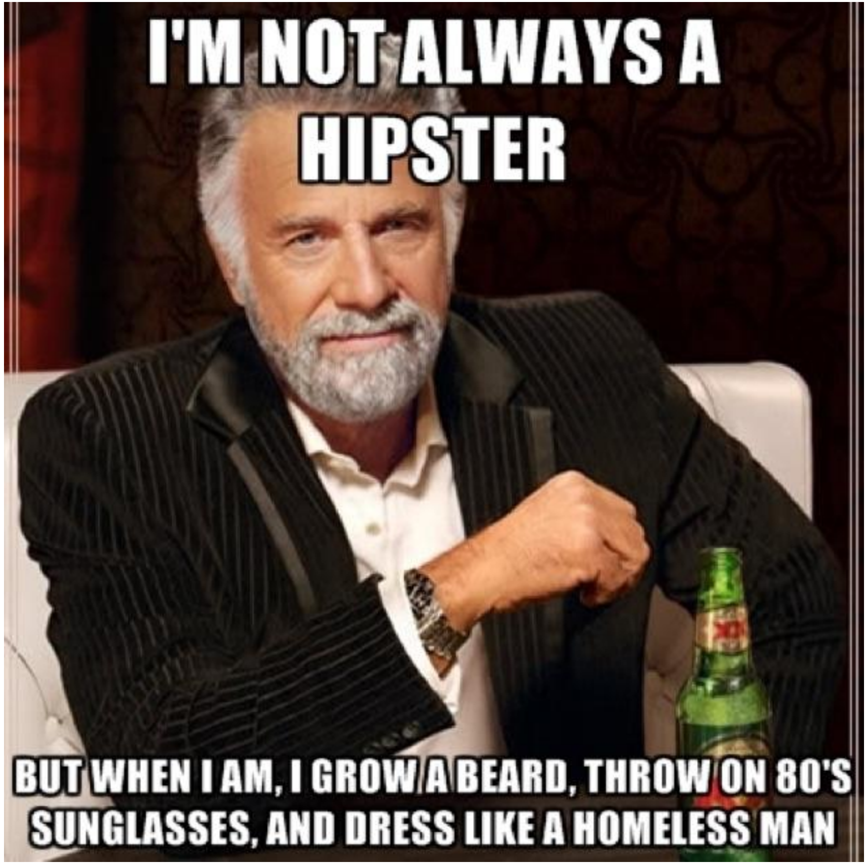
```
{ “article”: “glue”,
  “id”: “GX123”,
  “quantity”: 50,
  “complete”: true }
```

Raw materials for order:

```
/orders/ABC123/materials/GX123
              /materials/QZ125
              /materials/VF675
```

So we want to store JSON in a tree...

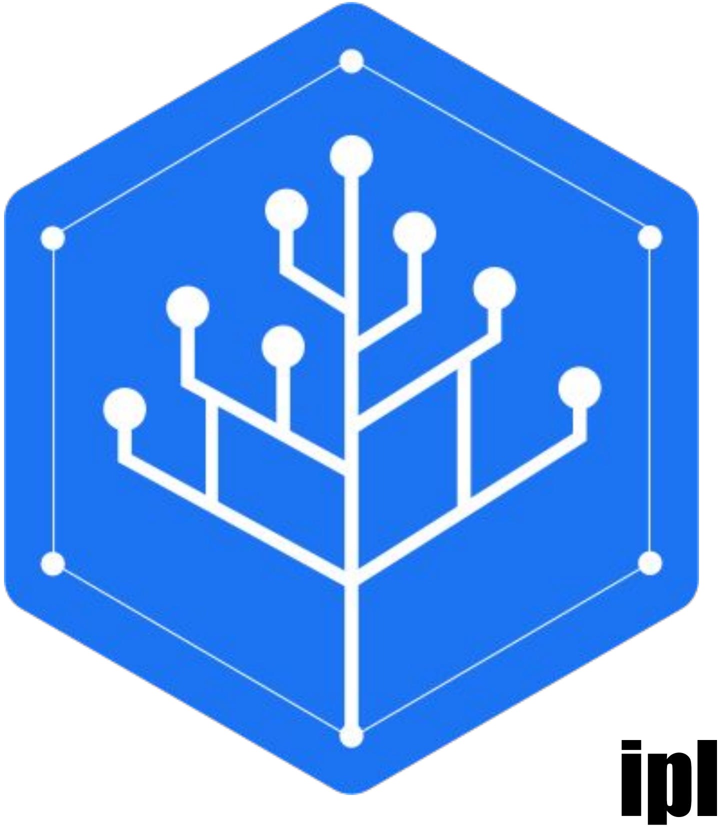
Distributing data


I'M NOT ALWAYS A HIPSTER
BUT WHEN I AM, I GROW A BEARD, THROW ON 80'S SUNGLASSES, AND DRESS LIKE A HOMELESS MAN

http://www.vitamin-ha.com/funniest-hipster-memes/funny-hispters20/#prettyPhoto bittorrent + bitcoin + blockchain + dht + kademlia + json + hypermedia + git + content-addressed block storage + distributed file system + dag = ?

IPFS is a peer-to-peer distributed file system that seeks to connect all computing devices with the same system of files. In some ways, IPFS is similar to the World Wide Web, but IPFS could be seen as a single BitTorrent swarm, exchanging objects within one Git repository. In other words, IPFS provides a high-throughput, content-addressed block storage model, with content-addressed hyperlinks.[11] This forms a generalized Merkle directed acyclic graph (DAG)

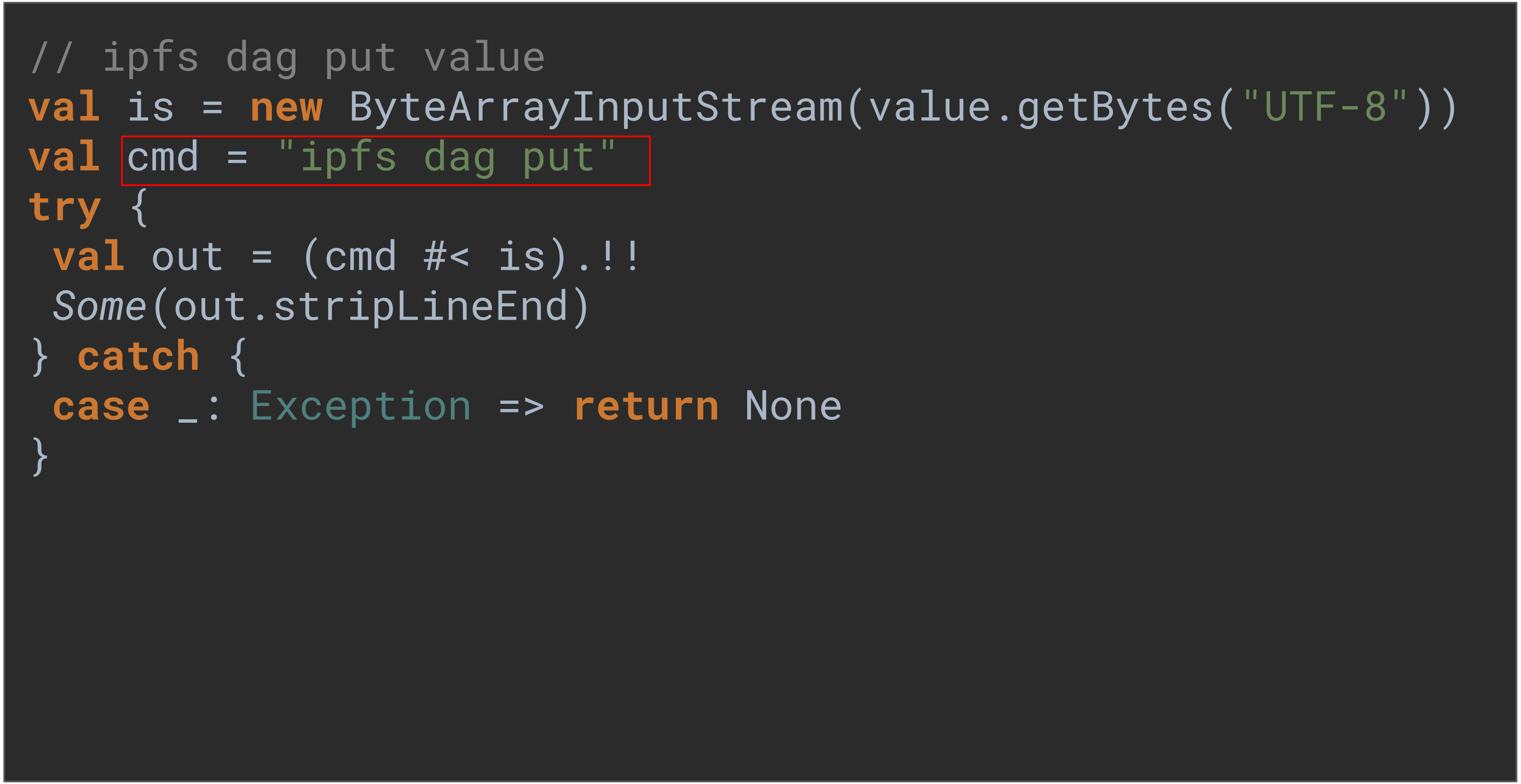
https://en.wikipedia.org/wiki/InterPlanetary_File_System



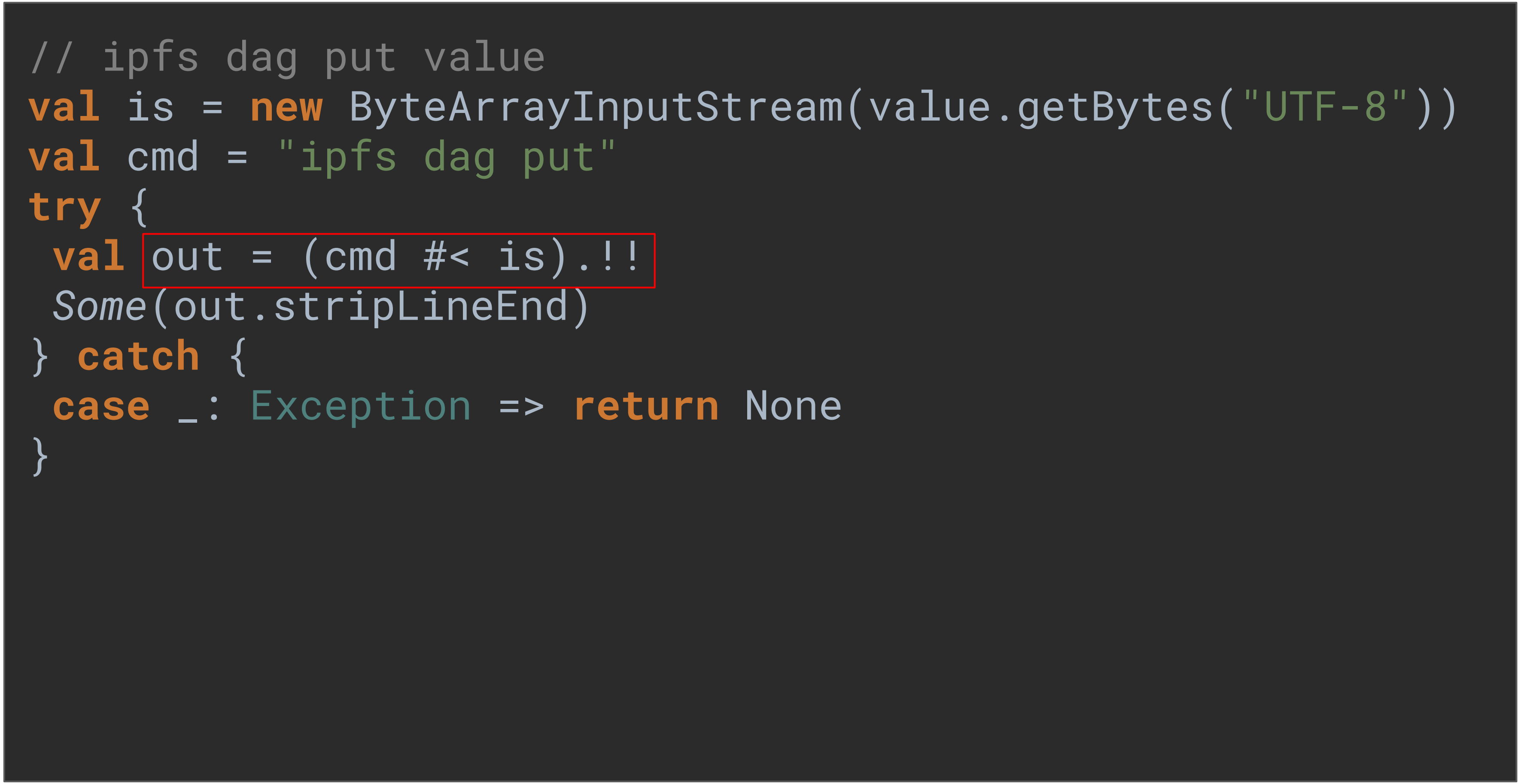
ipld

```
$ echo '{"article": "glue", "id": "GX123",
"quantity": 50, "complete": false}' | ipfs dag put

zdpuAoCUWWeWkRjGzxV26g1MgDdbRnspRmfxQWZGsv2epEoXF

$ ipfs dag get
zdpuAoCUWWeWkRjGzxV26g1MgDdbRnspRmfxQWZGsv2epEoXF

{"article":"glue","complete":false,"id":"GX123","quan
tity":50}
```

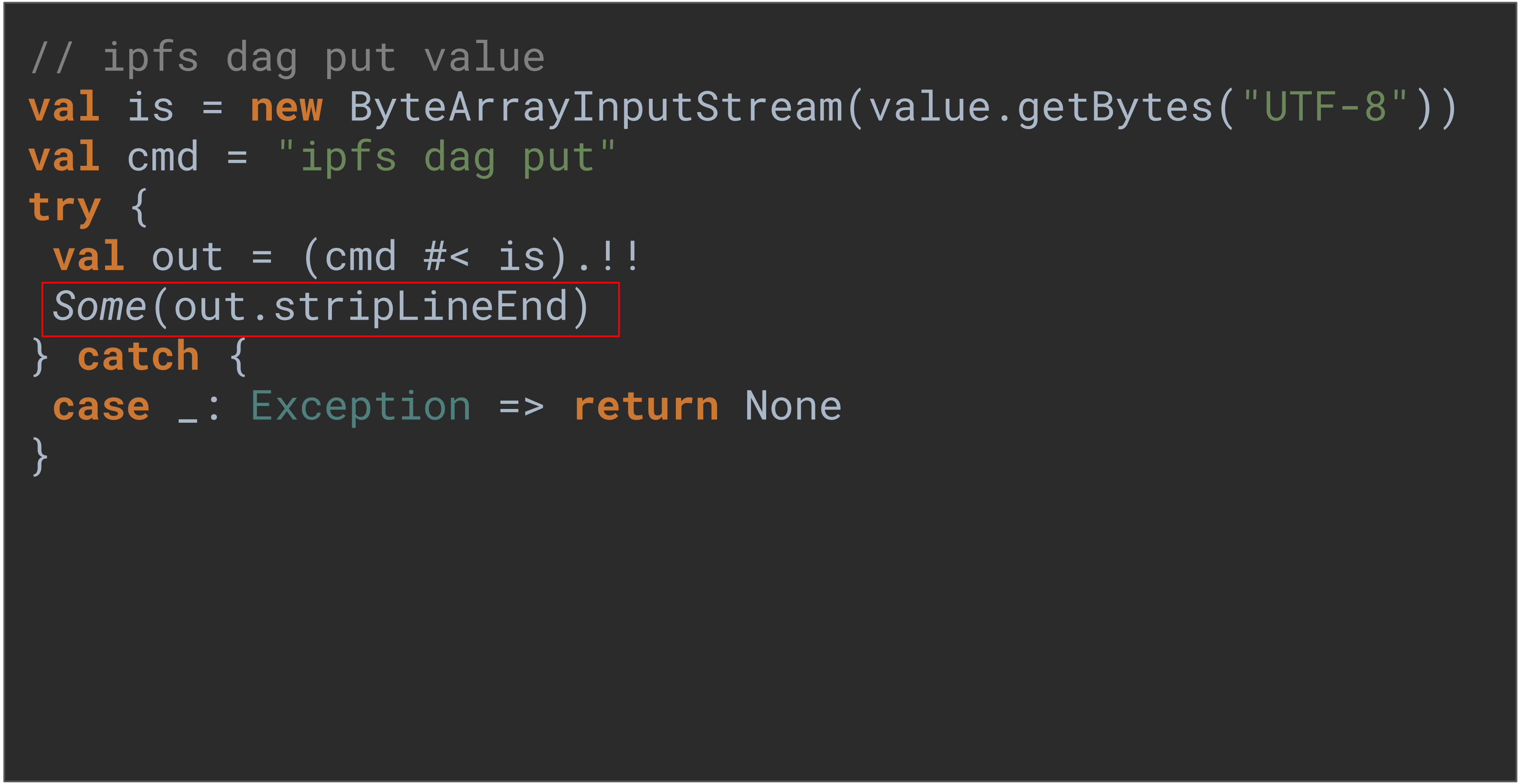
```scala
// ipfs dag put value
val is = new ByteArrayInputStream(value.getBytes("UTF-8"))
val cmd = "ipfs dag put"
try {
 val out = (cmd #< is).!!
 Some(out.stripLineEnd)
} catch {
 case _: Exception => return None
}
```



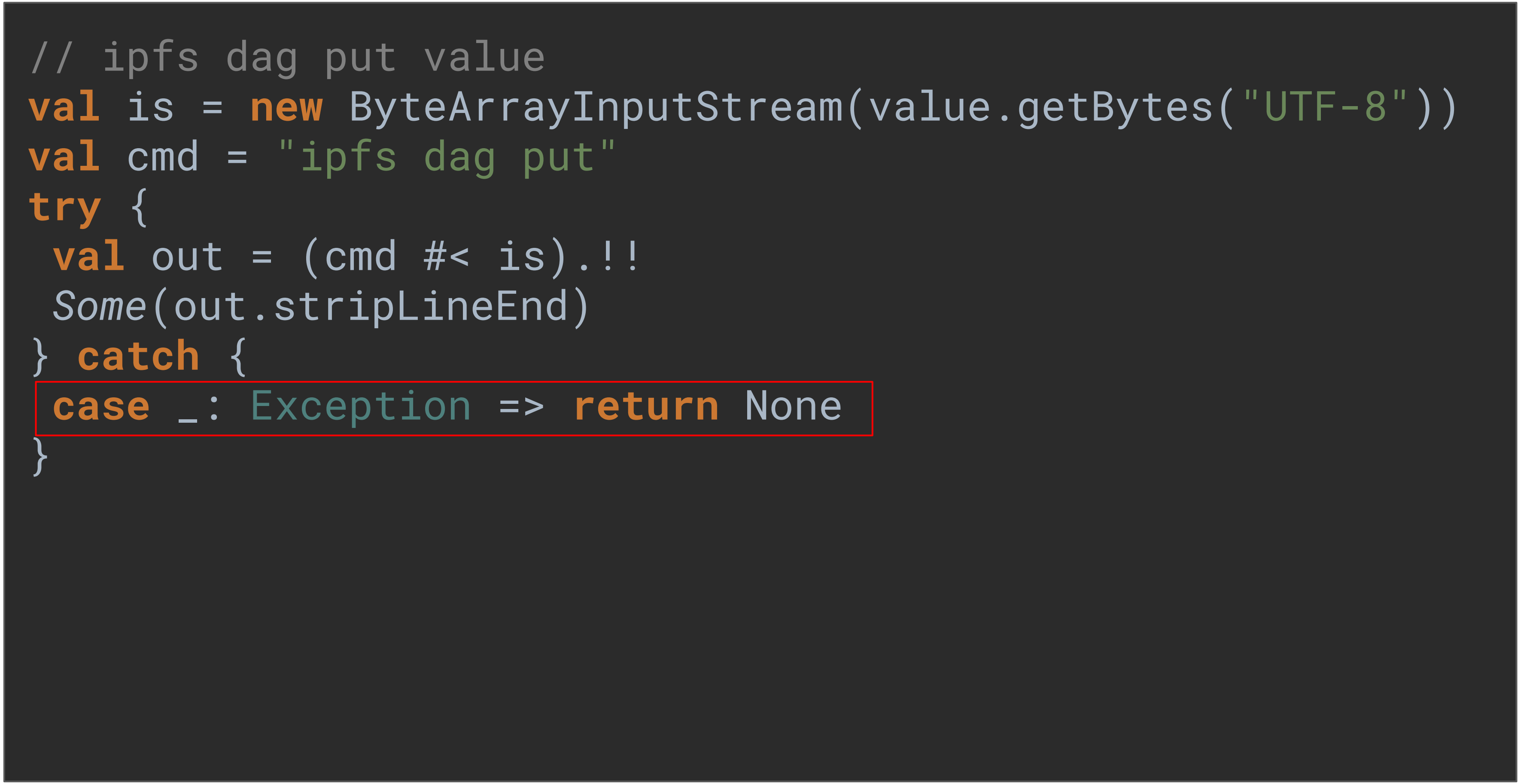
```
// ipfs dag put value
val is = new ByteArrayInputStream(value.getBytes("UTF-8"))
val cmd = "ipfs dag put"
try {
 val out = (cmd #< is).!!
 Some(out.stripLineEnd)
} catch {
 case _: Exception => return None
}
```



```
// ipfs dag put value
val is = new ByteArrayInputStream(value.getBytes("UTF-8"))
val cmd = "ipfs dag put"
try {
 val out = (cmd #< is).!!
 Some(out.stripLineEnd)
} catch {
 case _: Exception => return None
}
```



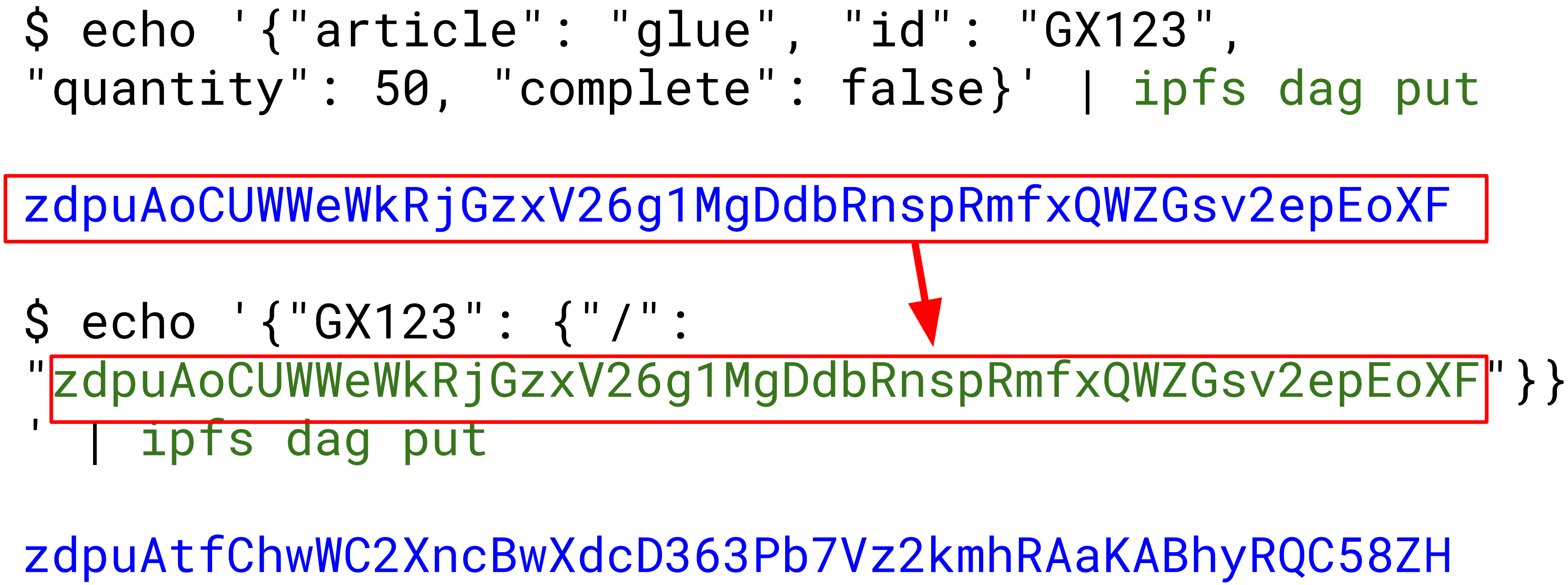
```
// ipfs dag put value
val is = new ByteArrayInputStream(value.getBytes("UTF-8"))
val cmd = "ipfs dag put"
try {
 val out = (cmd #< is).!!
 Some(out.stripLineEnd)
} catch {
 case _: Exception => return None
}
```



```
// ipfs dag put value
val is = new ByteArrayInputStream(value.getBytes("UTF-8"))
val cmd = "ipfs dag put"
try {
 val out = (cmd #< is).!!
 Some(out.stripLineEnd)
} catch {
 case _: Exception => return None
}
```

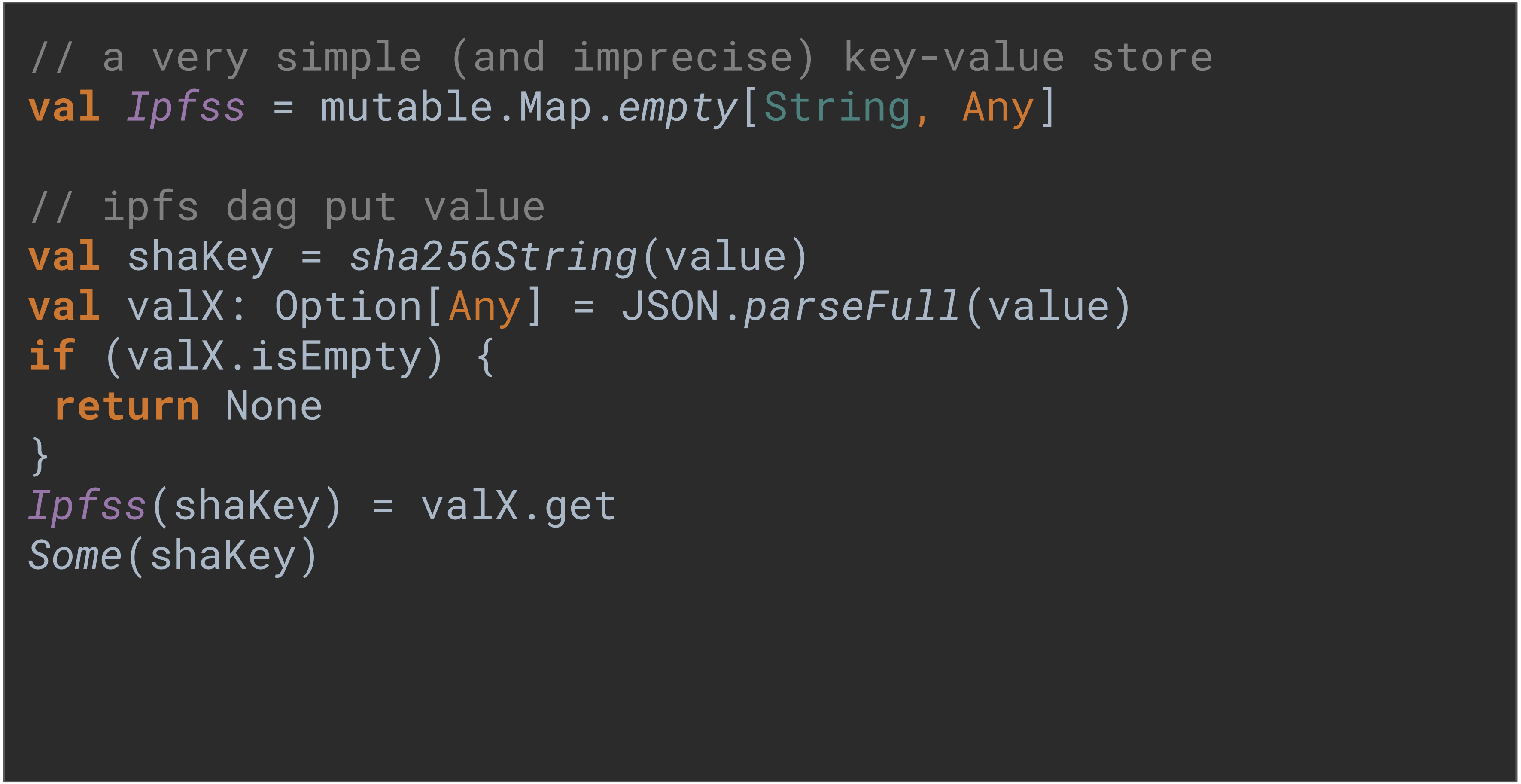
```scala
// ipfs dag get path
val cmd = s"ipfs dag get $path"
try {
 val out = cmd.!!
 Some(out.stripLineEnd)
} catch {
 case _: Exception => return None
}
```

```
$ ipfs dag get someRootHash/orders/ABC123/materials/GX123

{"article":"glue","complete":false,"id":"GX123","quantity":50}
```

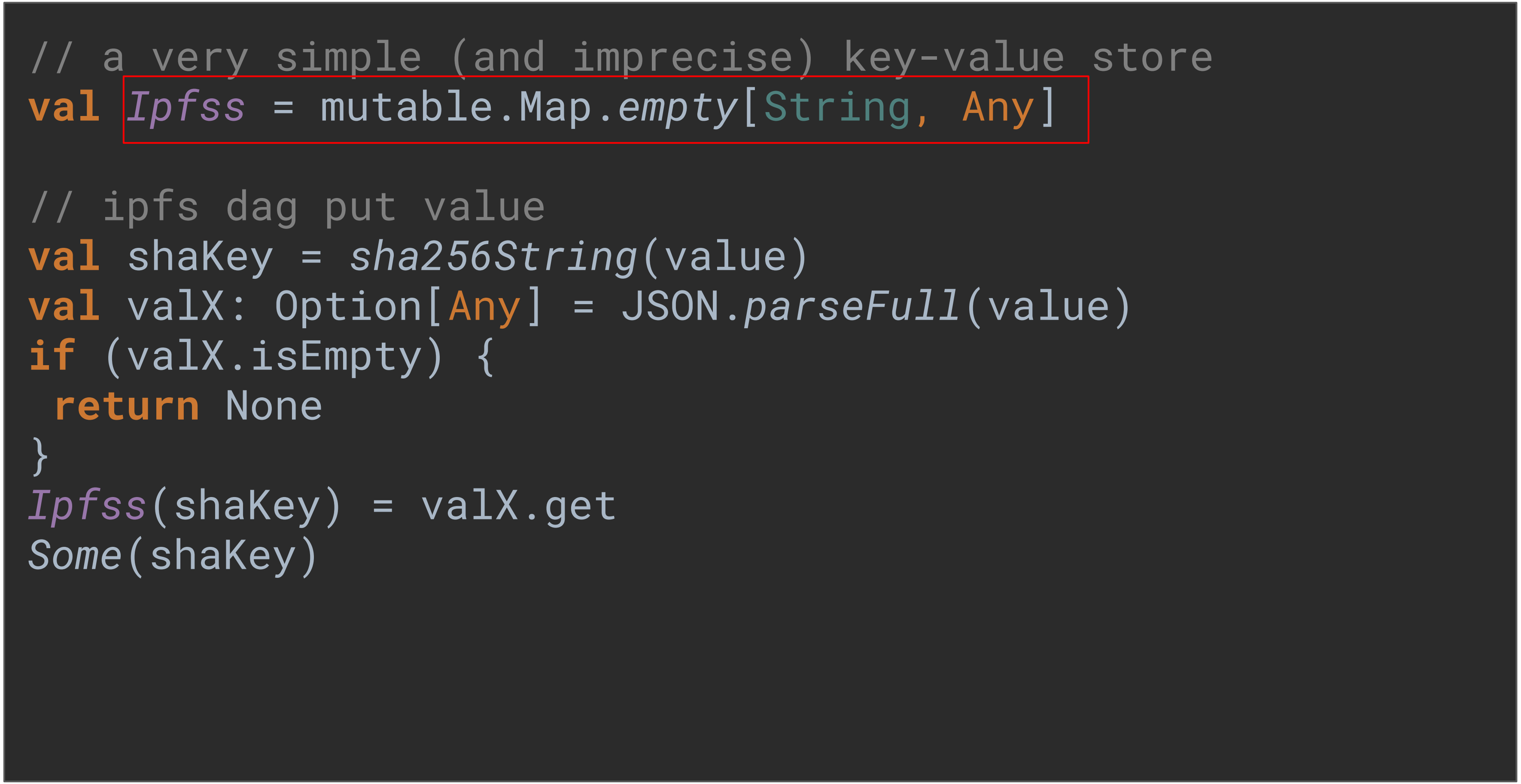


```
$ echo '{"article": "glue", "id": "GX123",
"quantity": 50, "complete": false}' | ipfs dag put

zdpuAoCUWWeWkRjGzxV26g1MgDdbRnspRmfxQWZGsv2epEoXF

$ echo '{"GX123": {"/":
"zdpuAoCUWWeWkRjGzxV26g1MgDdbRnspRmfxQWZGsv2epEoXF"}}
' | ipfs dag put

zdpuAtfChwWC2XncBwXdcD363Pb7Vz2kmhRAaKABhyRQC58ZH
```

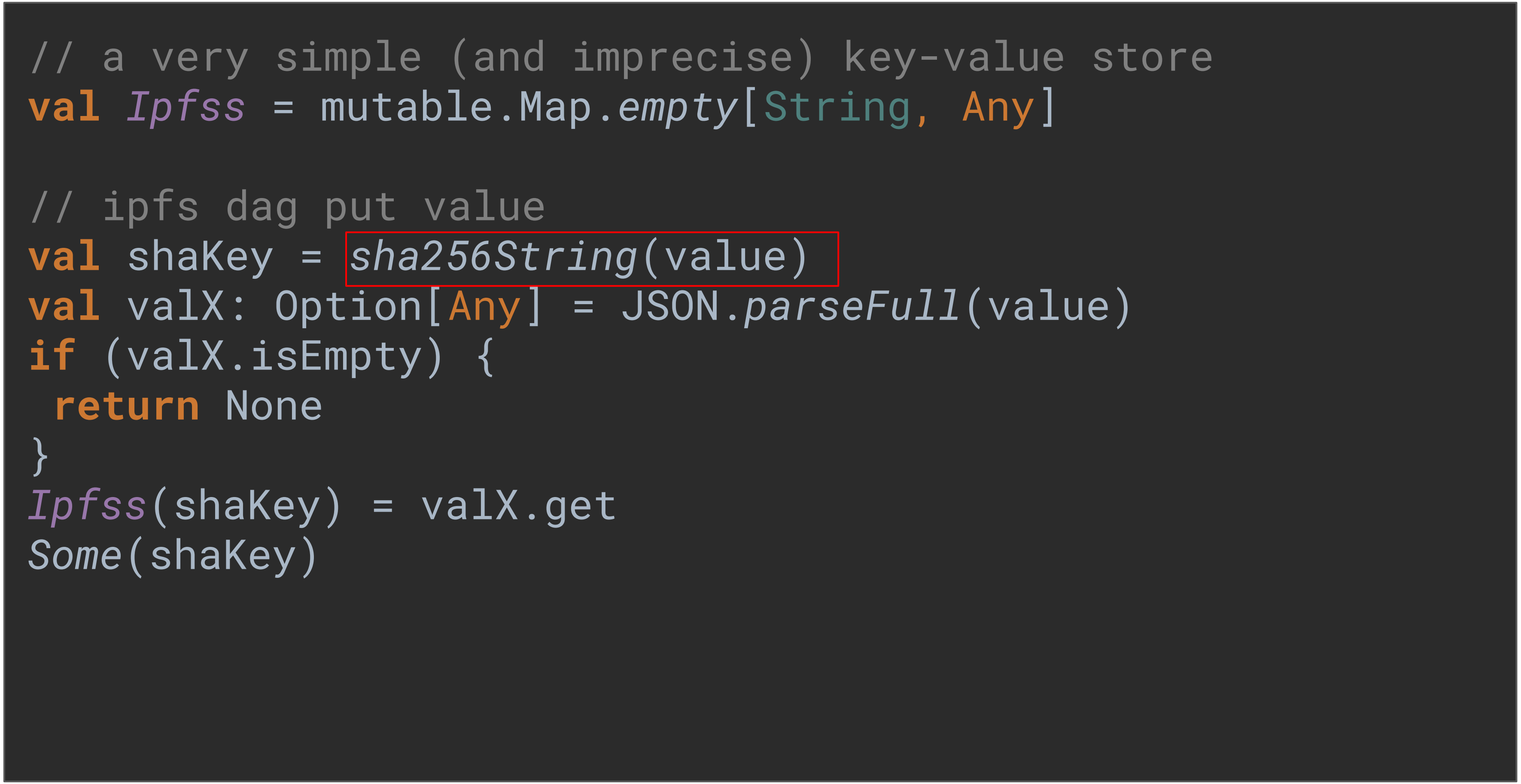
```
$ ipfs dag get
zdpuAtfChwWC2XncBwXdcD363Pb7Vz2kmhRAaKABhyRQC58ZH

{"GX123":{"/":"zdpuAoCUWWeWkRjGzxV26g1MgDdbRnspRmfxQW
ZGsv2epEoXF"}}

$ ipfs dag get
zdpuAtfChwWC2XncBwXdcD363Pb7Vz2kmhRAaKABhyRQC58ZH/GX1
23

{"article":"glue","complete":false,"id":"GX123","quan
tity":50}
```



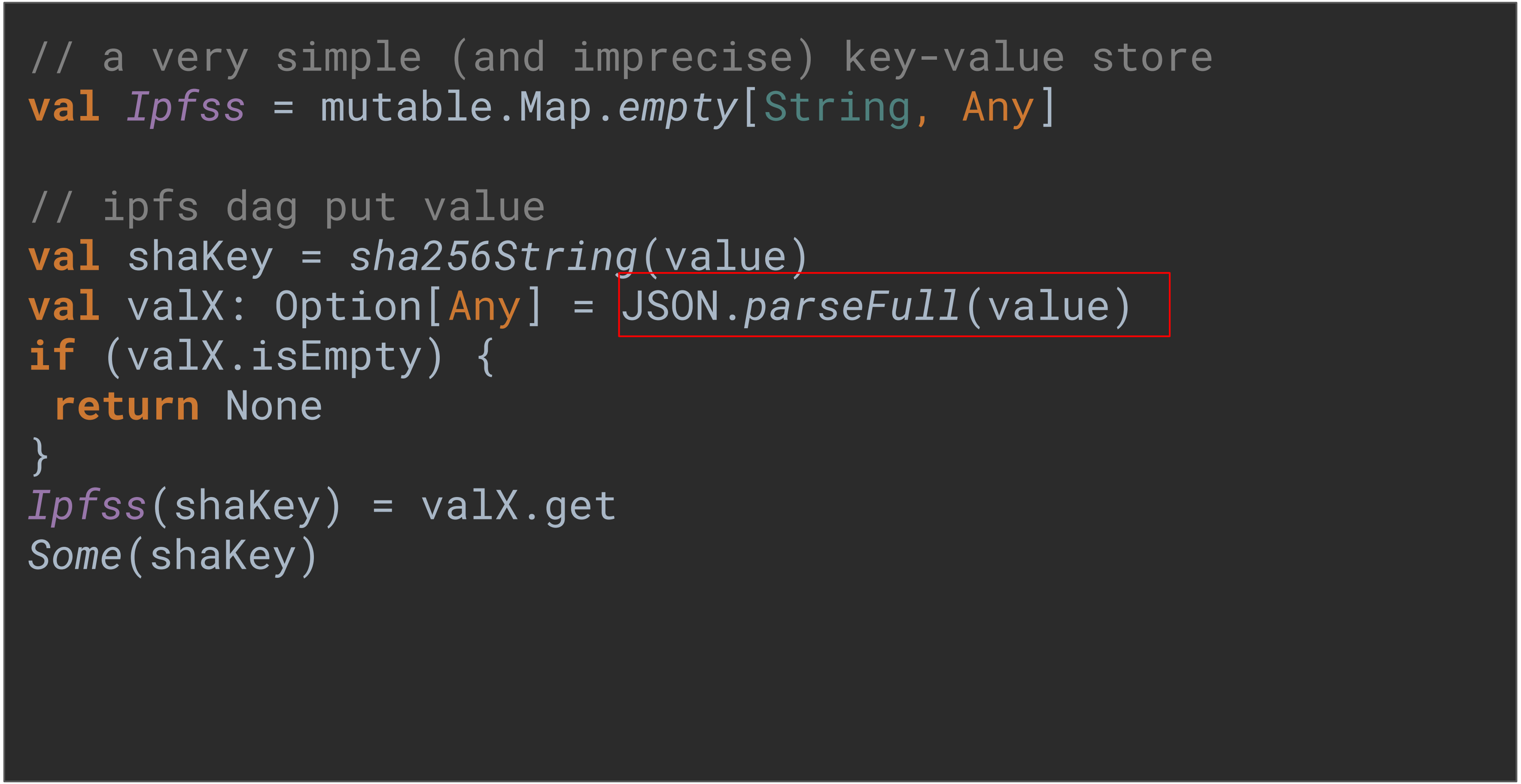
```
// a very simple (and imprecise) key-value store
val Ipfss = mutable.Map.empty[String, Any]

// ipfs dag put value
val shaKey = sha256String(value)
val valX: Option[Any] = JSON.parseFull(value)
if (valX.isEmpty) {
 return None
}
Ipfss(shaKey) = valX.get
Some(shaKey)
```



```
// a very simple (and imprecise) key-value store
val Ipfss = mutable.Map.empty[String, Any]

// ipfs dag put value
val shaKey = sha256String(value)
val valX: Option[Any] = JSON.parseFull(value)
if (valX.isEmpty) {
 return None
}
Ipfss(shaKey) = valX.get
Some(shaKey)
```

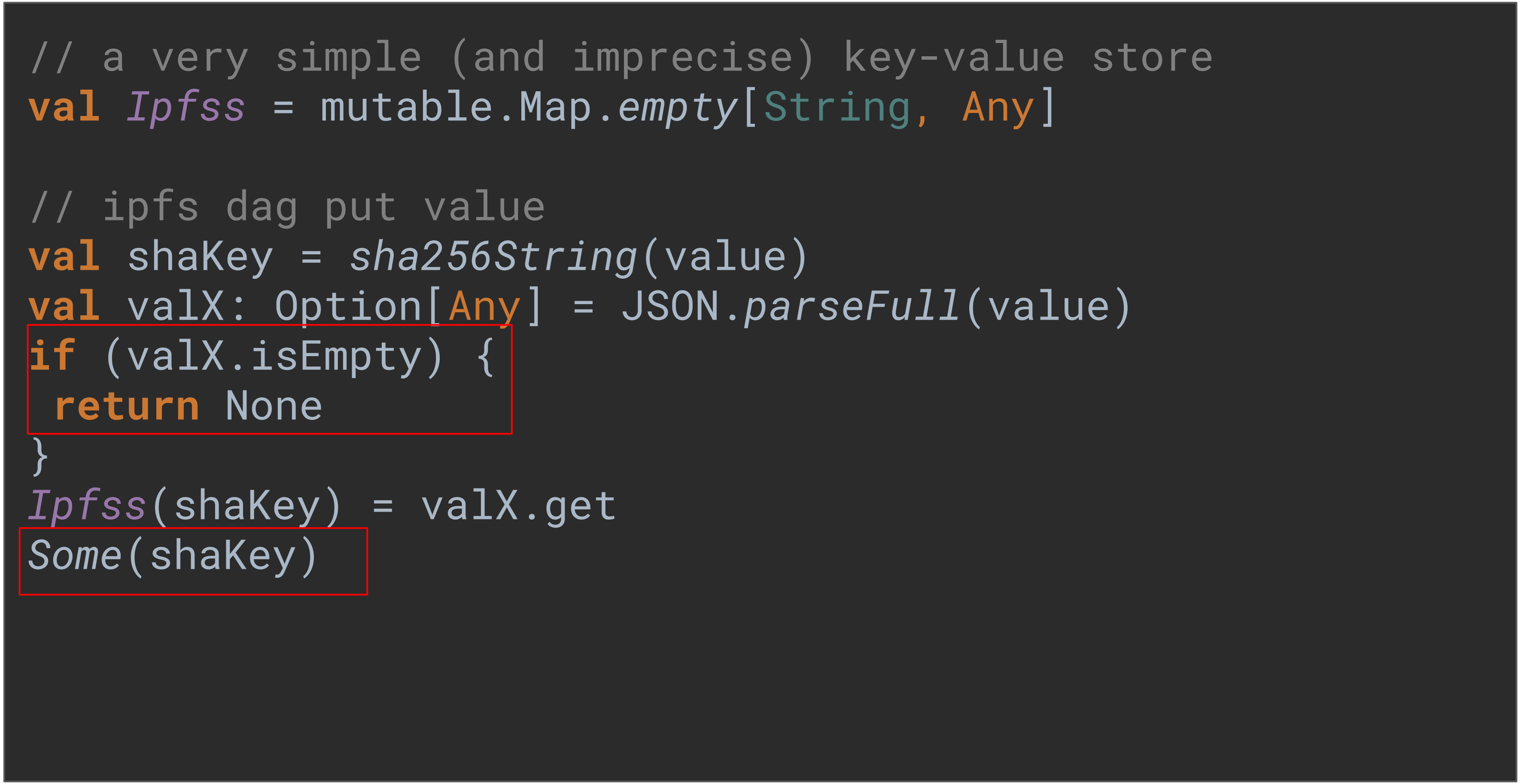


```
// a very simple (and imprecise) key-value store
val Ipfss = mutable.Map.empty[String, Any]

// ipfs dag put value
val shaKey = sha256String(value)
val valX: Option[Any] = JSON.parseFull(value)
if (valX.isEmpty) {
 return None
}
Ipfss(shaKey) = valX.get
Some(shaKey)
```

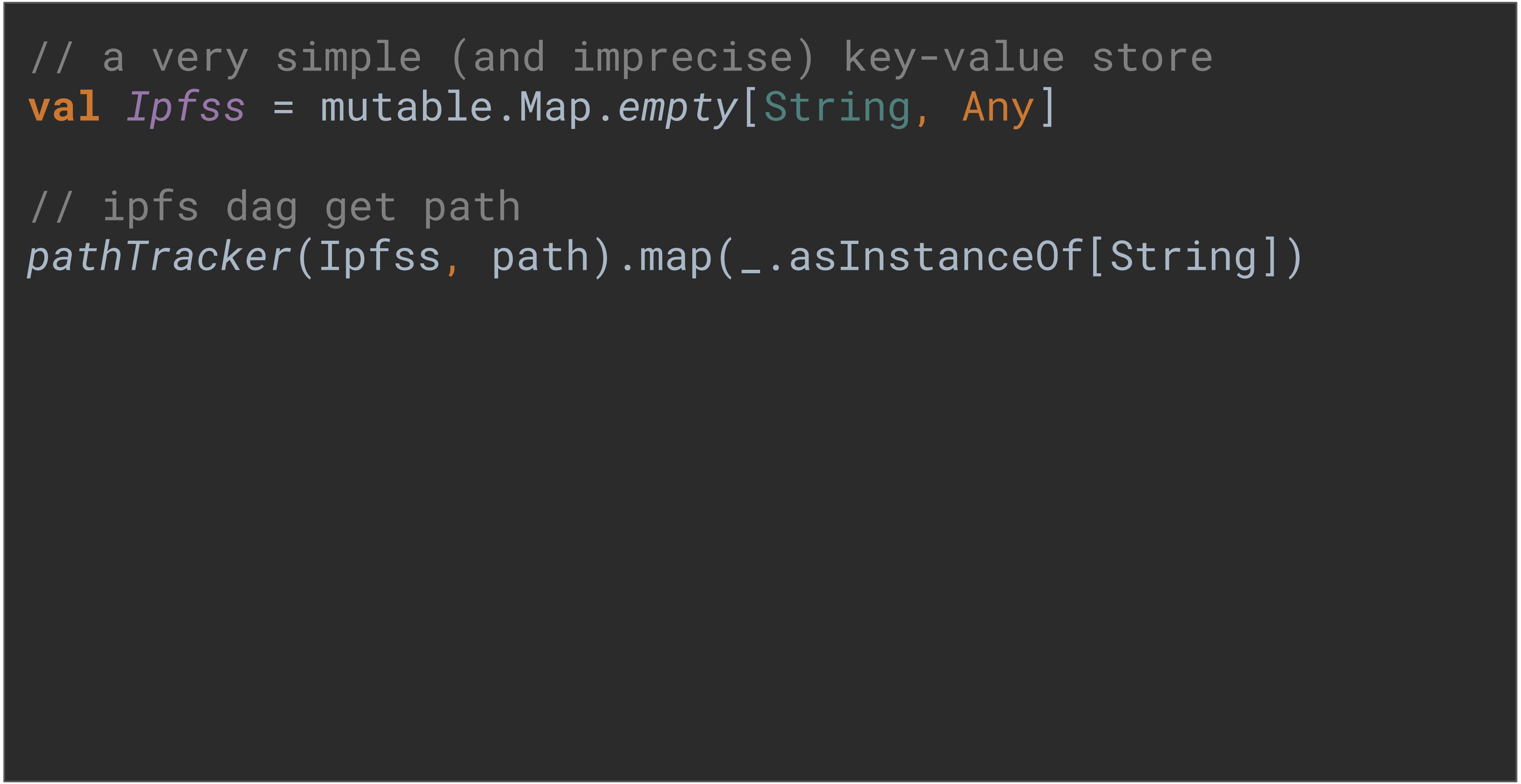


```
// a very simple (and imprecise) key-value store
val Ipfss = mutable.Map.empty[String, Any]

// ipfs dag put value
val shaKey = sha256String(value)
val valX: Option[Any] = JSON.parseFull(value)
if (valX.isEmpty) {
 return None
}
Ipfss(shaKey) = valX.get
Some(shaKey)
```

```
// a very simple (and imprecise) key-value store
val Ipfss = mutable.Map.empty[String, Any]

// ipfs dag put value
val shaKey = sha256String(value)
val valX: Option[Any] = JSON.parseFull(value)
if (valX.isEmpty) {
 return None
}
Ipfss(shaKey) = valX.get
Some(shaKey)
```



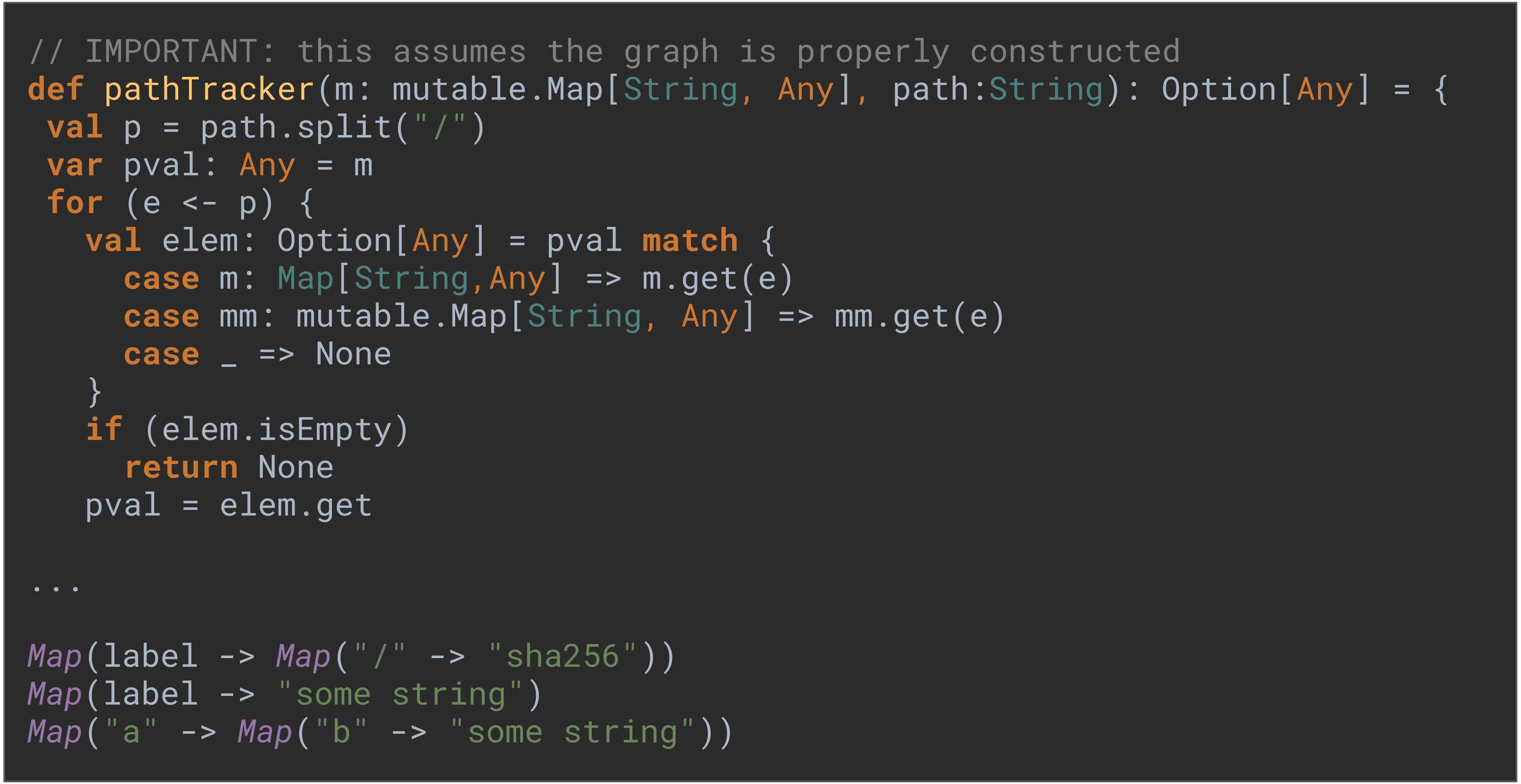
```scala
// a very simple (and imprecise) key-value store
val Ipfss = mutable.Map.empty[String, Any]

// ipfs dag put value
val shaKey = sha256String(value)
val valX: Option[Any] = JSON.parseFull(value)
if (valX.isEmpty) {
 return None
}
Ipfss(shaKey) = valX.get
Some(shaKey)
```



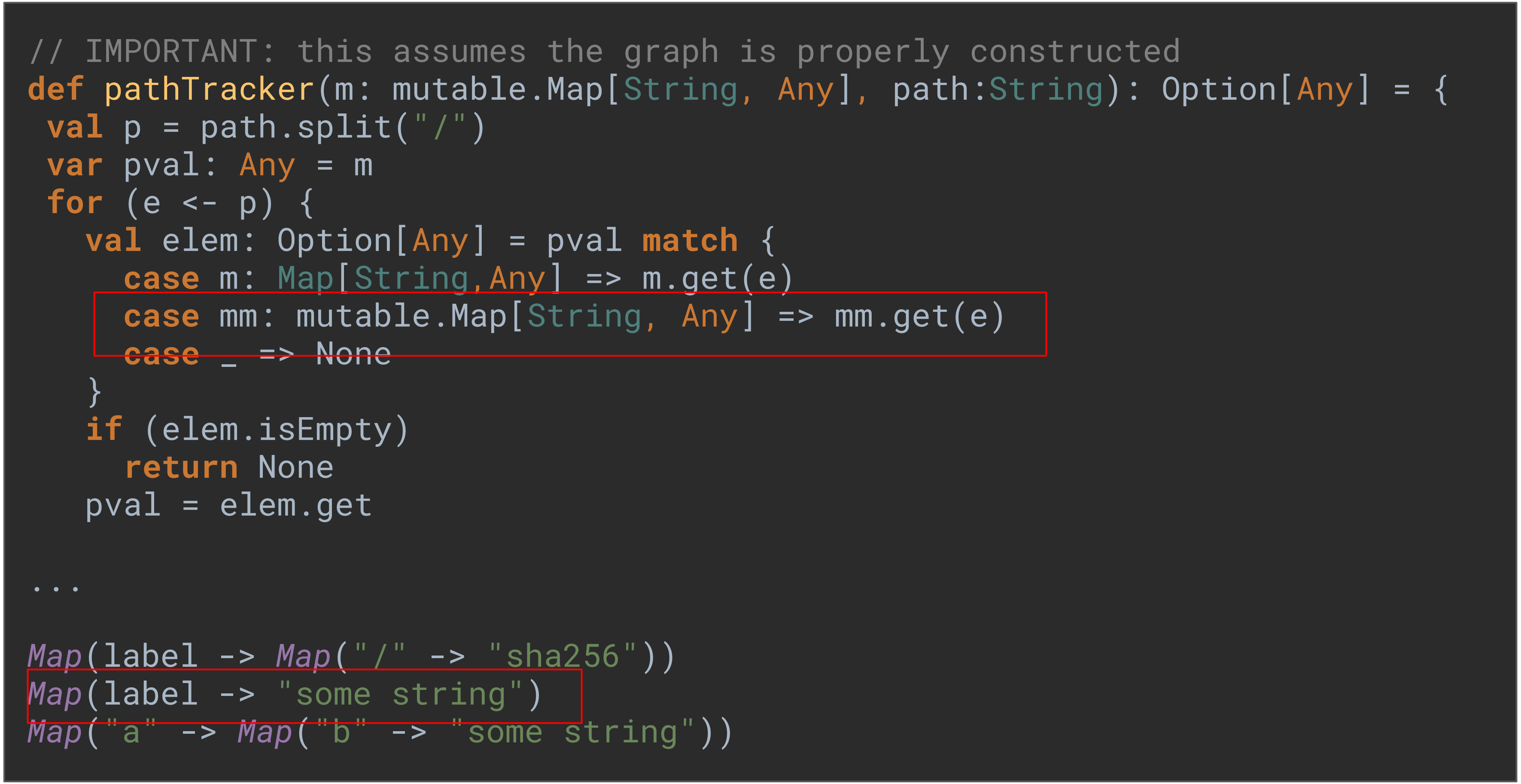
```scala
// a very simple (and imprecise) key-value store
val Ipfss = mutable.Map.empty[String, Any]

// ipfs dag get path
pathTracker(Ipfss, path).map(_.asInstanceOf[String])
```

```scala
// a very simple (and imprecise) key-value store
val Ipfss = mutable.Map.empty[String, Any]

// ipfs dag get path
pathTracker(Ipfss, path).map(_.asInstanceOf[String])






Map(label -> Map("/" -> "sha256"))
Map(label -> "some string")
```

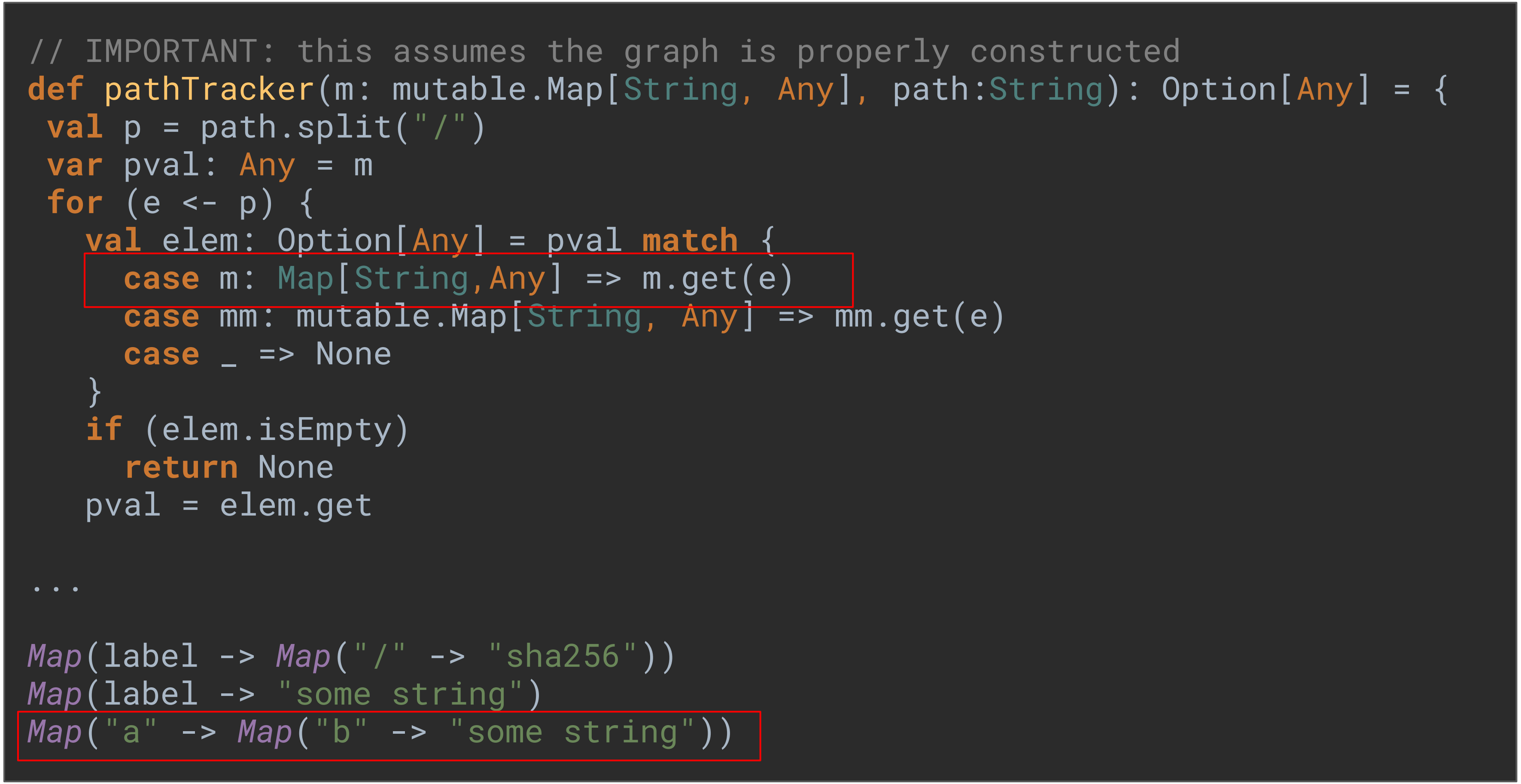
```
// IMPORTANT: this assumes the graph is properly constructed
def pathTracker(m: mutable.Map[String, Any], path:String): Option[Any] = {
 val p = path.split("/")
 var pval: Any = m
 for (e <- p) {
   val elem: Option[Any] = pval match {
     case m: Map[String,Any] => m.get(e)
     case mm: mutable.Map[String, Any] => mm.get(e)
     case _ => None
   }
   if (elem.isEmpty)
     return None
   pval = elem.get

...
```



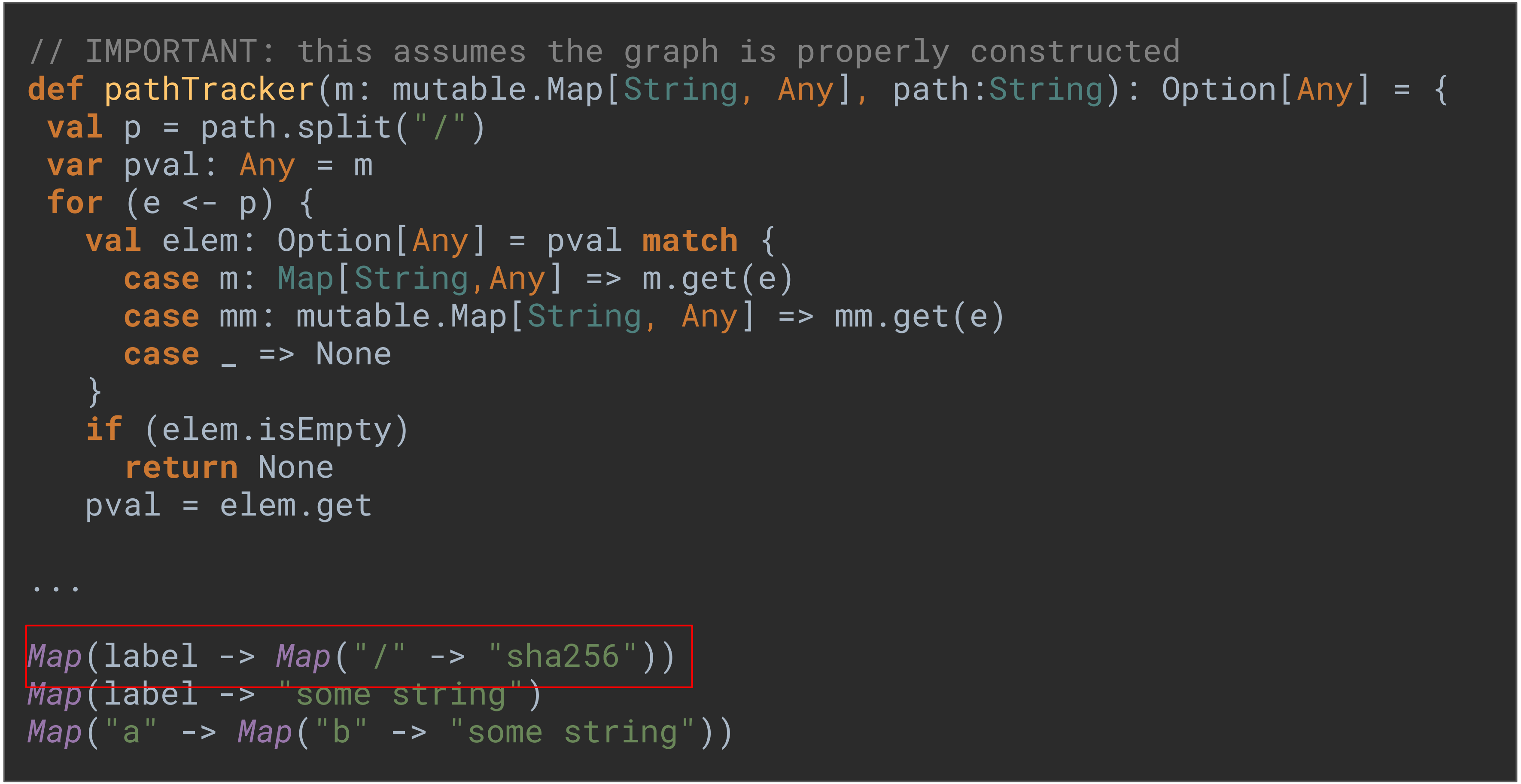
```scala
// IMPORTANT: this assumes the graph is properly constructed
def pathTracker(m: mutable.Map[String, Any], path:String): Option[Any] = {
 val p = path.split("/")
 var pval: Any = m
 for (e <- p) {
   val elem: Option[Any] = pval match {
     case m: Map[String,Any] => m.get(e)
     case mm: mutable.Map[String, Any] => mm.get(e)
     case _ => None
   }
   if (elem.isEmpty)
     return None
   pval = elem.get

...

Map(label -> Map("/" -> "sha256"))
Map(label -> "some string")
Map("a" -> Map("b" -> "some string"))
```



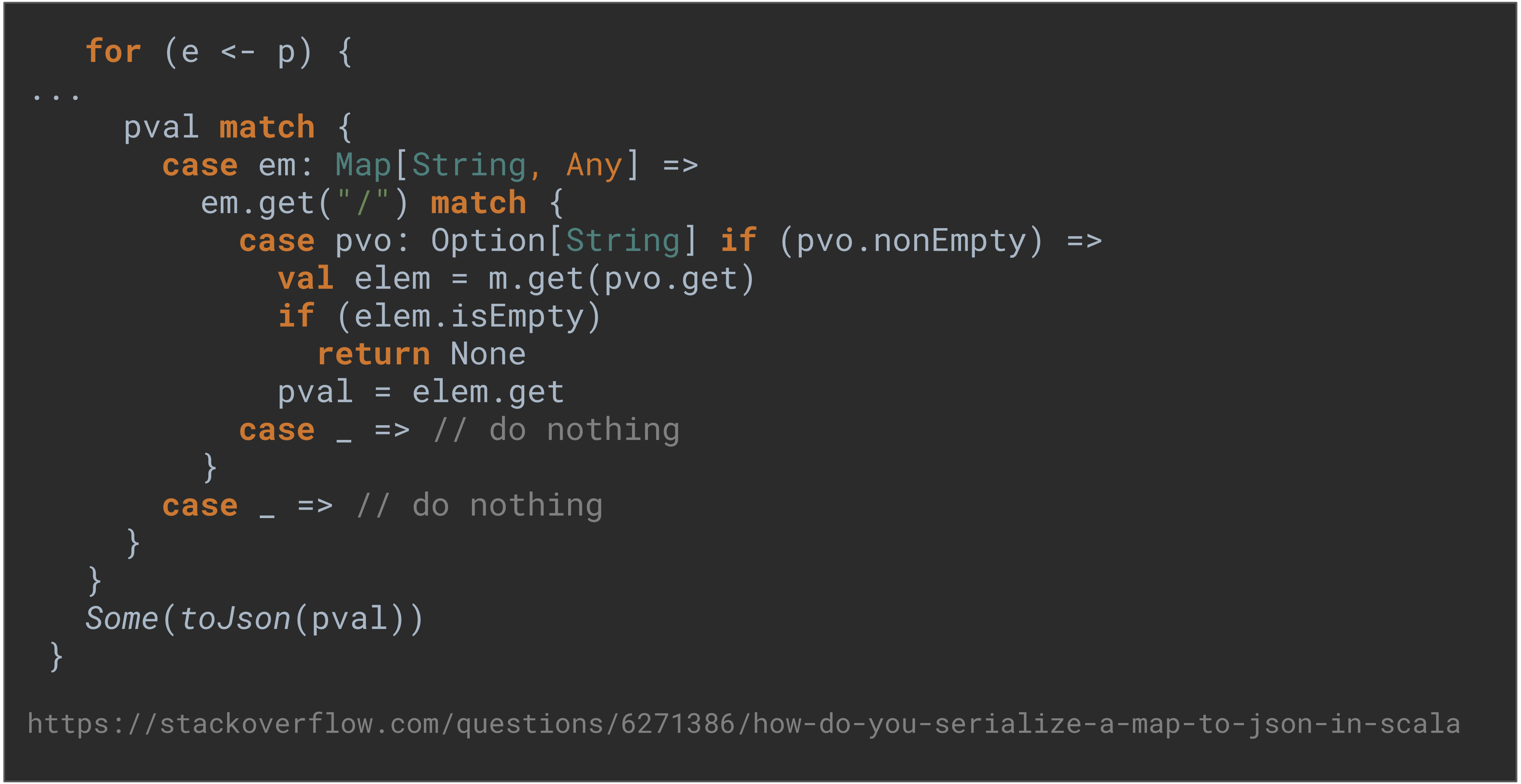
```
// IMPORTANT: this assumes the graph is properly constructed
def pathTracker(m: mutable.Map[String, Any], path:String): Option[Any] = {
 val p = path.split("/")
 var pval: Any = m
 for (e <- p) {
   val elem: Option[Any] = pval match {
     case m: Map[String,Any] => m.get(e)
     case mm: mutable.Map[String, Any] => mm.get(e)
     case _ => None
   }
   if (elem.isEmpty)
     return None
   pval = elem.get

...

Map(label -> Map("/" -> "sha256"))
Map(label -> "some string")
Map("a" -> Map("b" -> "some string"))
```



```
// IMPORTANT: this assumes the graph is properly constructed
def pathTracker(m: mutable.Map[String, Any], path:String): Option[Any] = {
 val p = path.split("/")
 var pval: Any = m
 for (e <- p) {
   val elem: Option[Any] = pval match {
     case m: Map[String,Any] => m.get(e)
     case mm: mutable.Map[String, Any] => mm.get(e)
     case _ => None
   }
   if (elem.isEmpty)
     return None
   pval = elem.get

...

Map(label -> Map("/" -> "sha256"))
Map(label -> "some string")
Map("a" -> Map("b" -> "some string"))
```



```
// IMPORTANT: this assumes the graph is properly constructed
def pathTracker(m: mutable.Map[String, Any], path:String): Option[Any] = {
 val p = path.split("/")
 var pval: Any = m
 for (e <- p) {
   val elem: Option[Any] = pval match {
     case m: Map[String,Any] => m.get(e)
     case mm: mutable.Map[String, Any] => mm.get(e)
     case _ => None
   }
   if (elem.isEmpty)
     return None
   pval = elem.get

...

Map(label -> Map("/" -> "sha256"))
Map(label -> "some string")
Map("a" -> Map("b" -> "some string"))
```



```
   for (e <- p) {
...
     pval match {
       case em: Map[String, Any] =>
         em.get("/") match {
           case pvo: Option[String] if (pvo.nonEmpty) =>
             val elem = m.get(pvo.get)
             if (elem.isEmpty)
               return None
             pval = elem.get
           case _ => // do nothing
         }
       case _ => // do nothing
     }
   }
   Some(toJson(pval))
 }
```

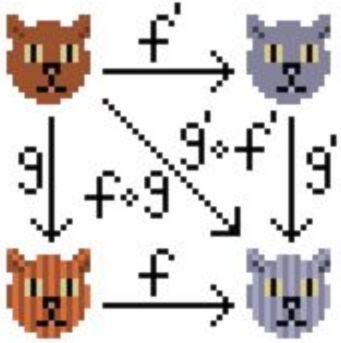
https://stackoverflow.com/questions/6271386/how-do-you-serialize-a-map-to-json-in-scala

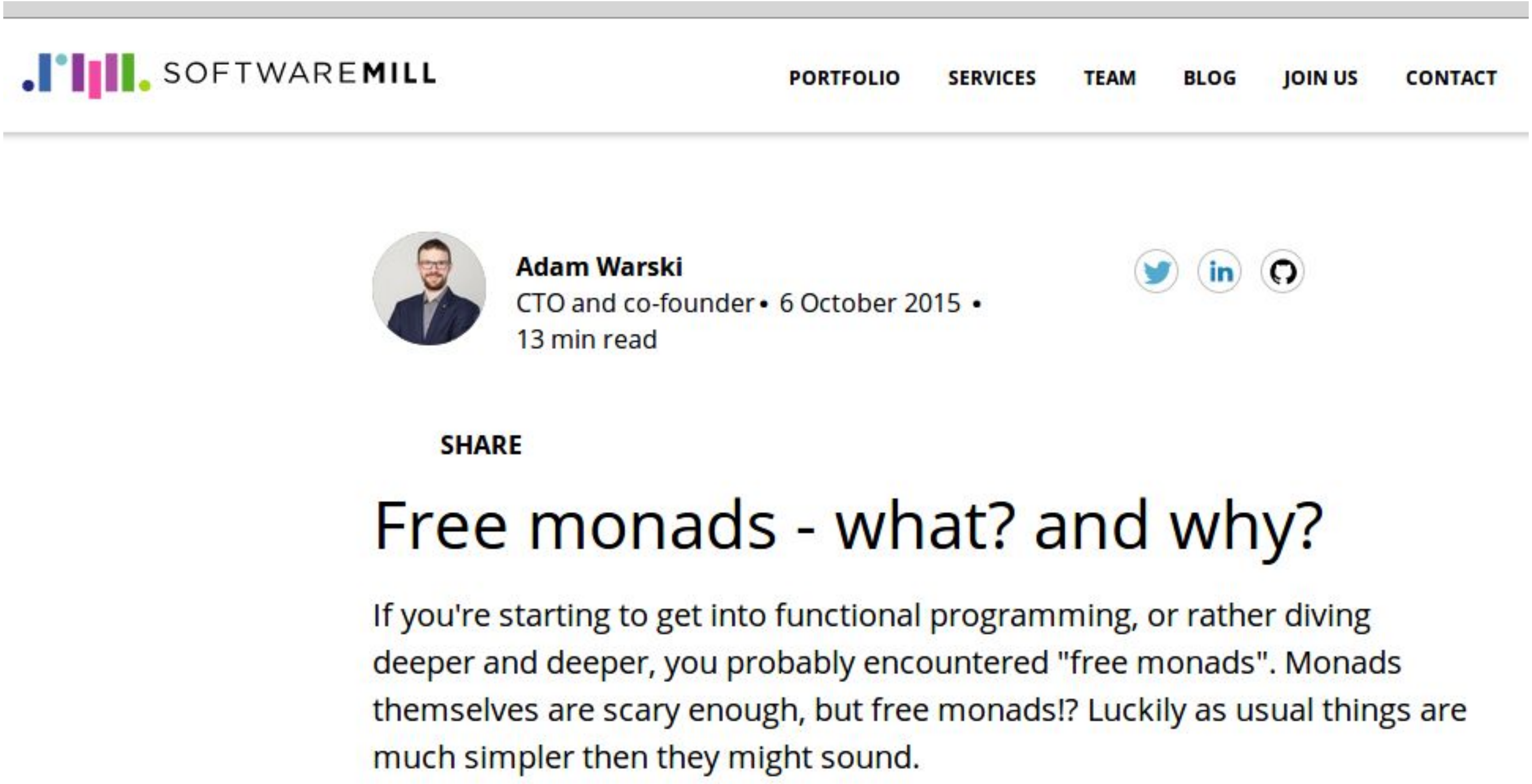
```
$ ipfs dag get someRootHash/orders/ABC123/materials/GX123

{"article":"glue","complete":false,"id":"GX123","quantity":50}
```

Ability to create such paths... and we are done!

Free

Free Monad


f'
g
f∘g
g'∘f'
g'
f



Adam Warski
CTO and co-founder • 6 October 2015 •
13 min read

Free monads - what? and why?

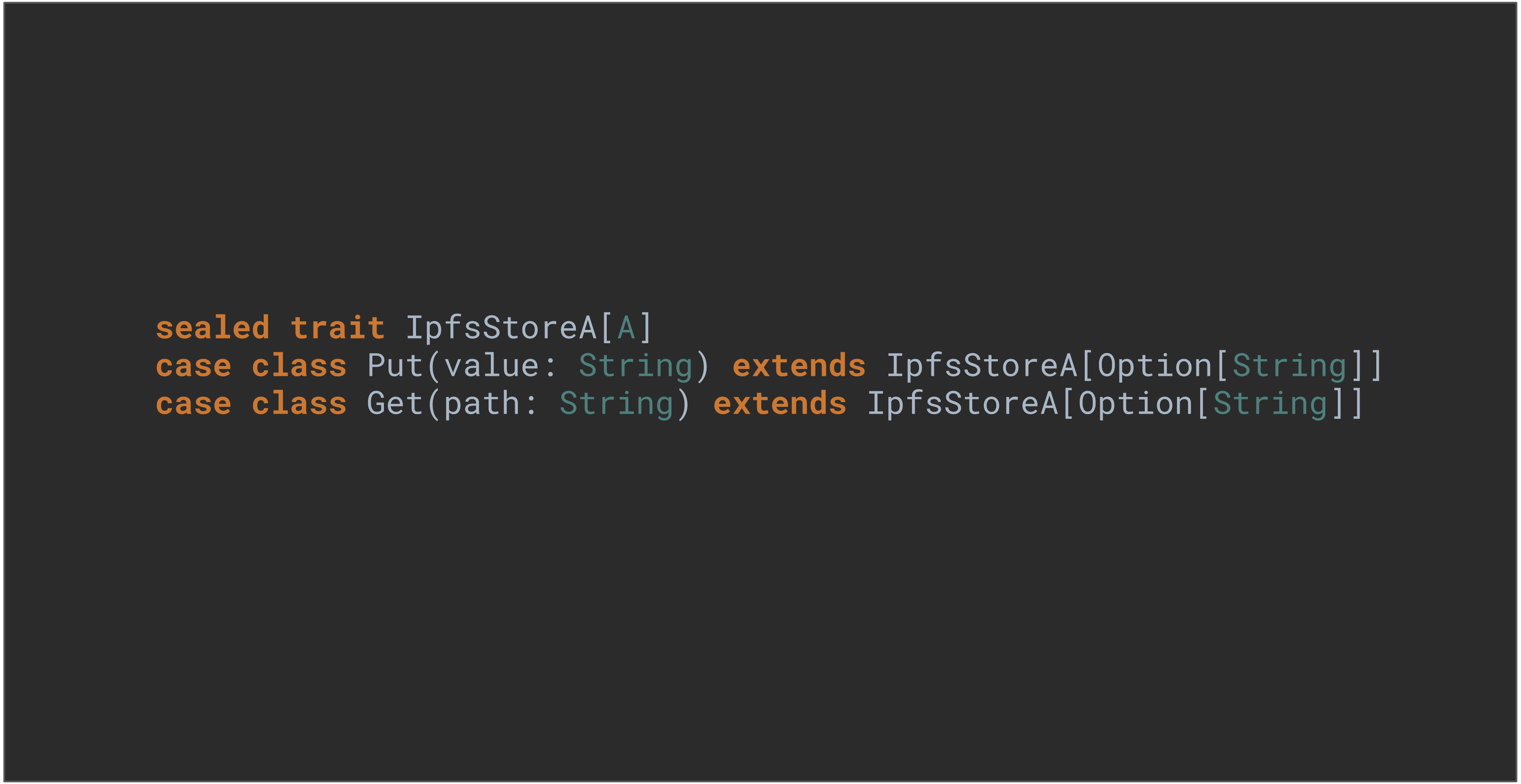
If you're starting to get into functional programming, or rather diving deeper and deeper, you probably encountered "free monads". Monads themselves are scary enough, but free monads!? Luckily as usual things are much simpler then they might sound.

Make your programs free / Free Monads - Paweł Szulc

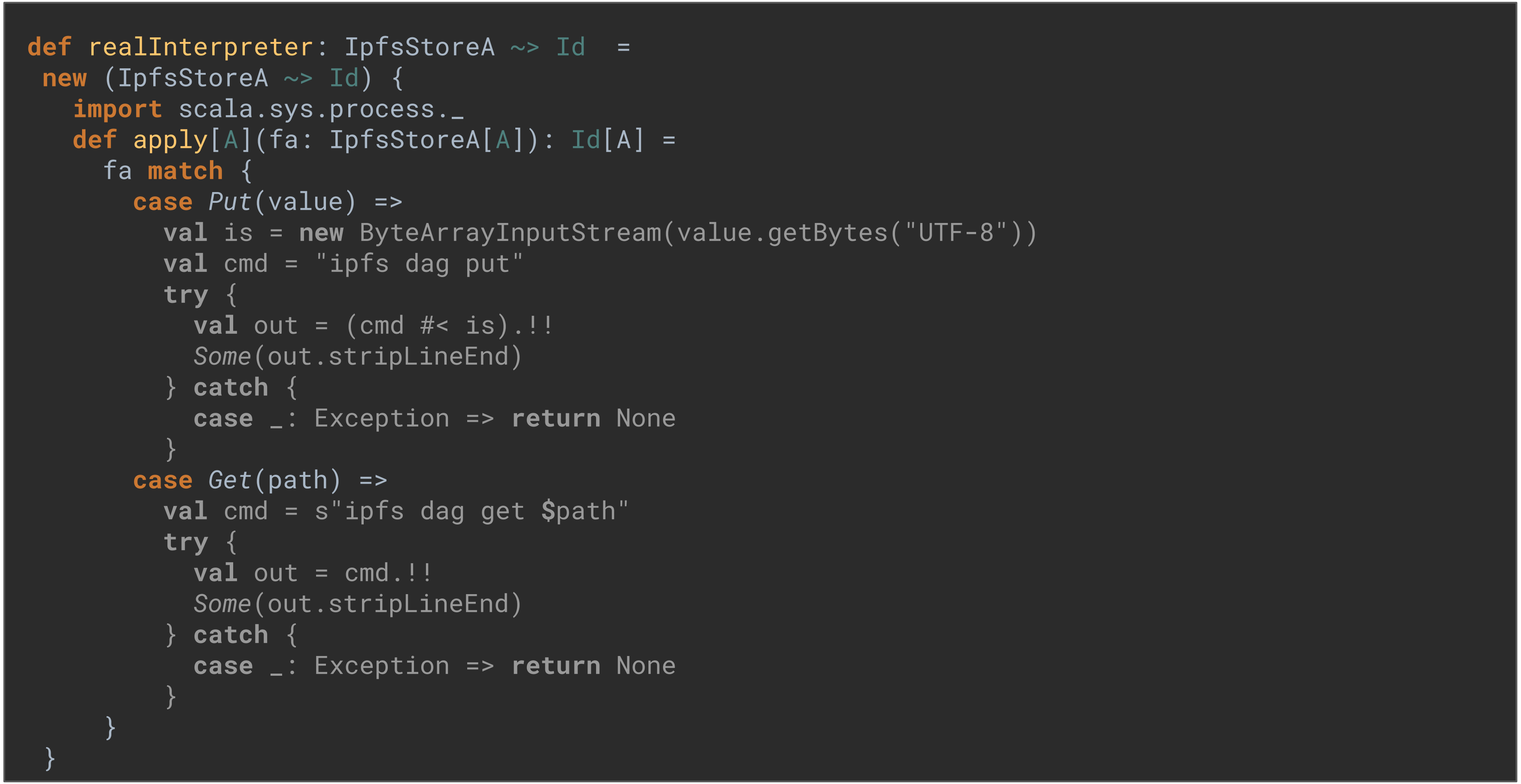
Free as in Monads - Daniel Spiewak

A Year living Freely – Chris Myers

John DeGoes: Beyond Free Monads - λC Winter Retreat 2017

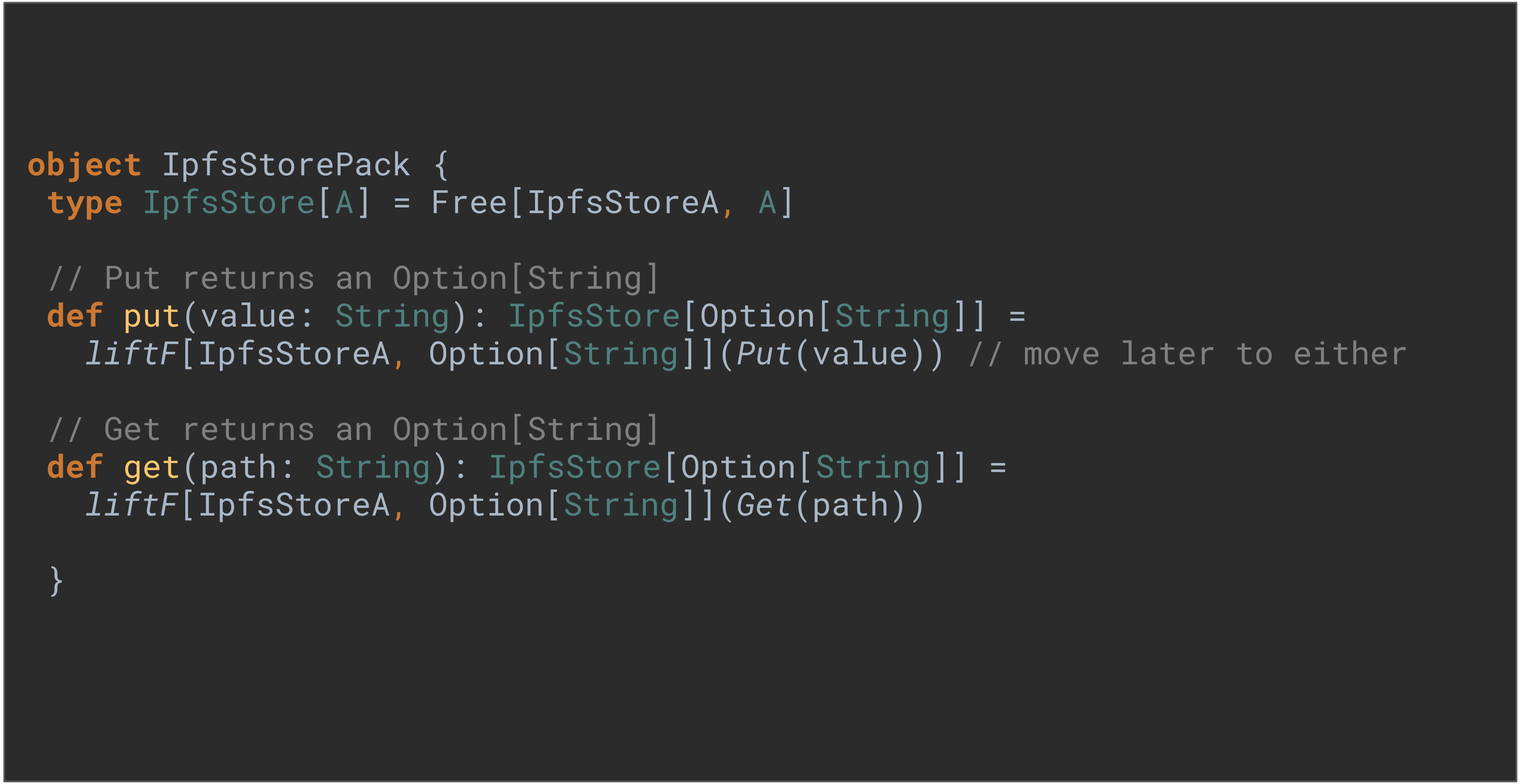


```
sealed trait IpfsStoreA[A]
case class Put(value: String) extends IpfsStoreA[Option[String]]
case class Get(path: String) extends IpfsStoreA[Option[String]]
```

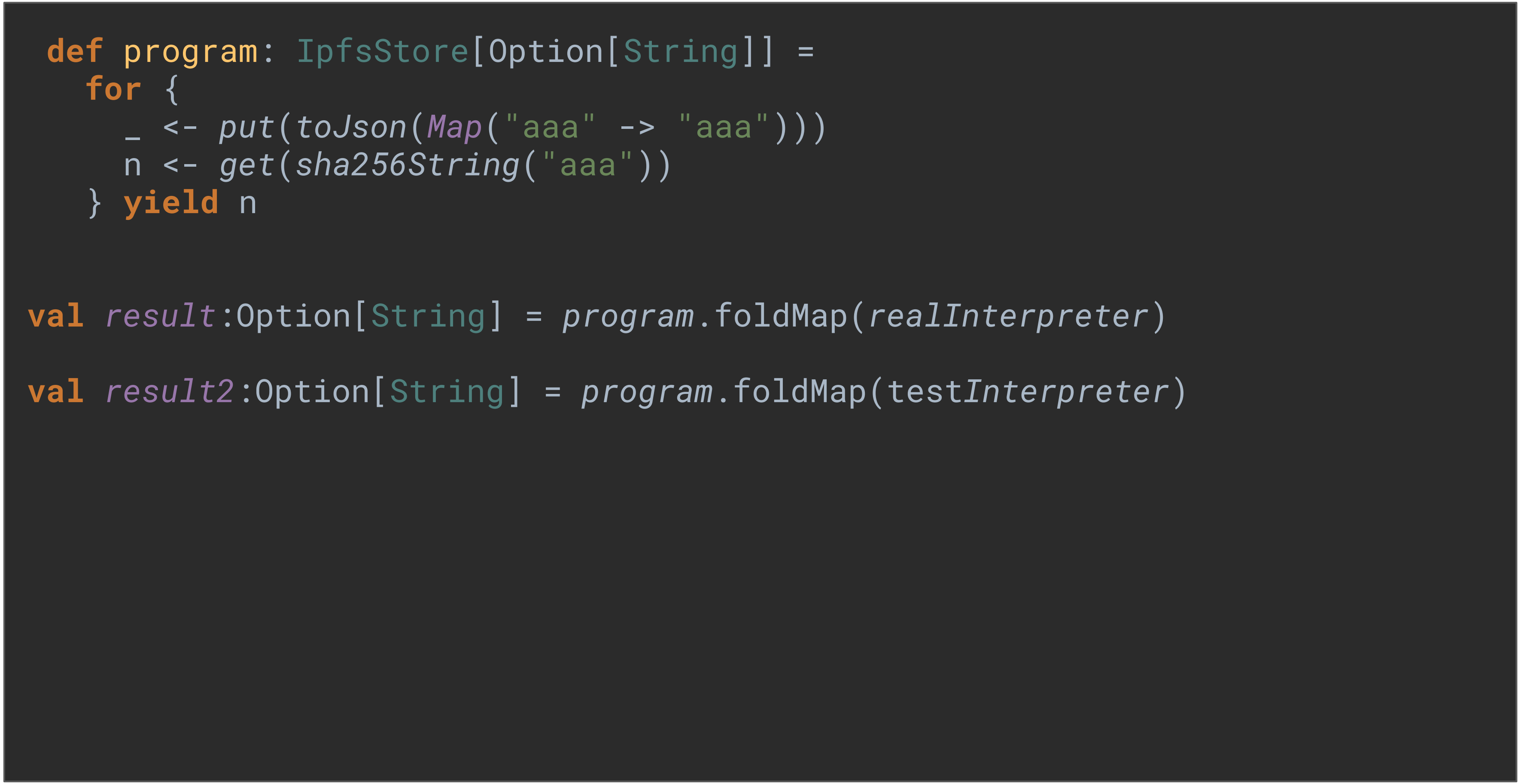


```scala
def realInterpreter: IpfsStoreA ~> Id  =
 new (IpfsStoreA ~> Id) {
   import scala.sys.process._
   def apply[A](fa: IpfsStoreA[A]): Id[A] =
     fa match {
       case Put(value) =>
         val is = new ByteArrayInputStream(value.getBytes("UTF-8"))
         val cmd = "ipfs dag put"
         try {
           val out = (cmd #< is).!!
           Some(out.stripLineEnd)
         } catch {
           case _: Exception => return None
         }
       case Get(path) =>
         val cmd = s"ipfs dag get $path"
         try {
           val out = cmd.!!
           Some(out.stripLineEnd)
         } catch {
           case _: Exception => return None
         }
     }
 }
```

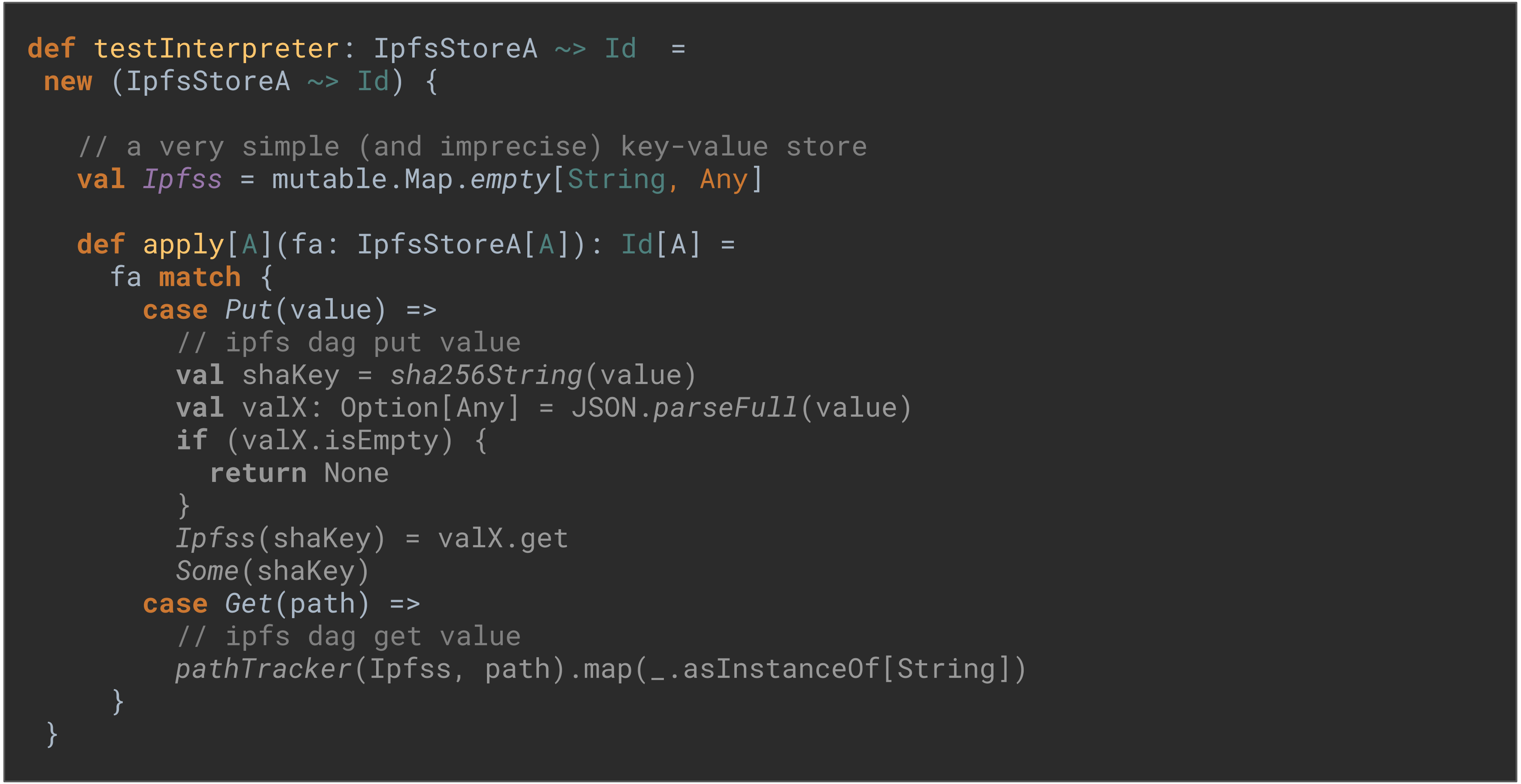
Handling errors logic... TBD



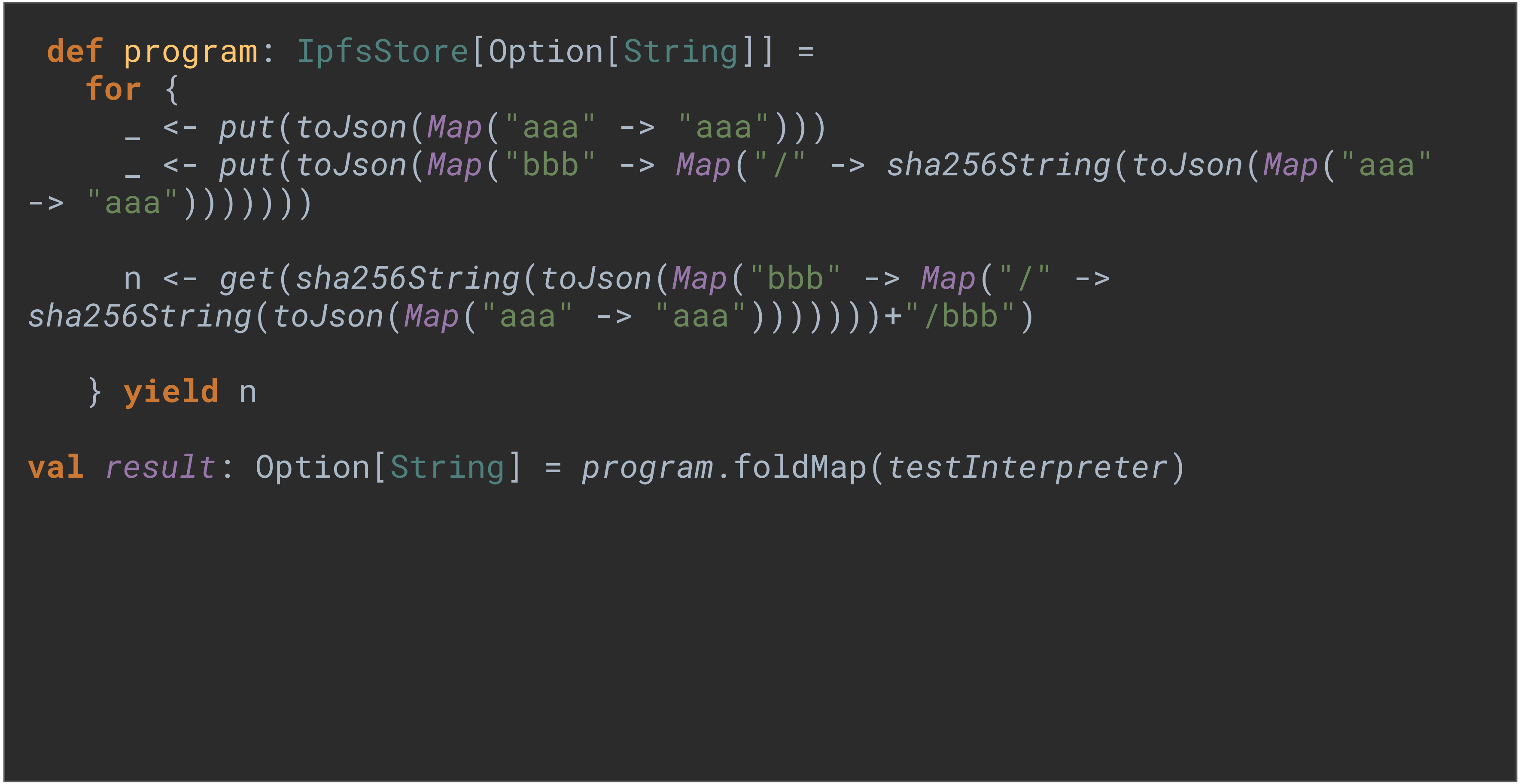
```scala
object IpfsStorePack {
 type IpfsStore[A] = Free[IpfsStoreA, A]

 // Put returns an Option[String]
 def put(value: String): IpfsStore[Option[String]] =
   liftF[IpfsStoreA, Option[String]](Put(value)) // move later to either

 // Get returns an Option[String]
 def get(path: String): IpfsStore[Option[String]] =
   liftF[IpfsStoreA, Option[String]](Get(path))

 }
```



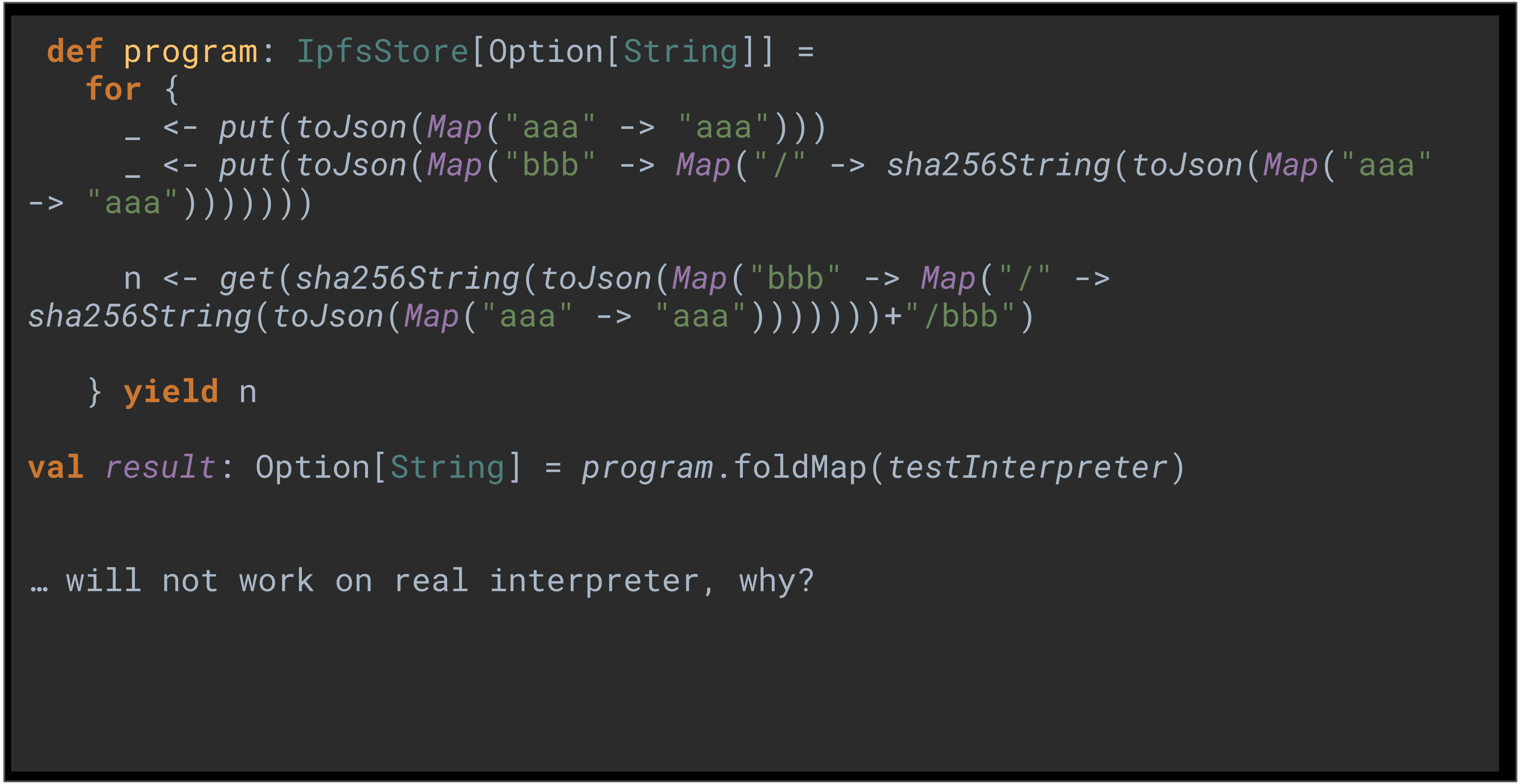
```
 def program: IpfsStore[Option[String]] =
   for {
     _ <- put(toJson(Map("aaa" -> "aaa")))
     n <- get(sha256String("aaa"))
   } yield n


val result:Option[String] = program.foldMap(realInterpreter)

val result2:Option[String] = program.foldMap(testInterpreter)
```



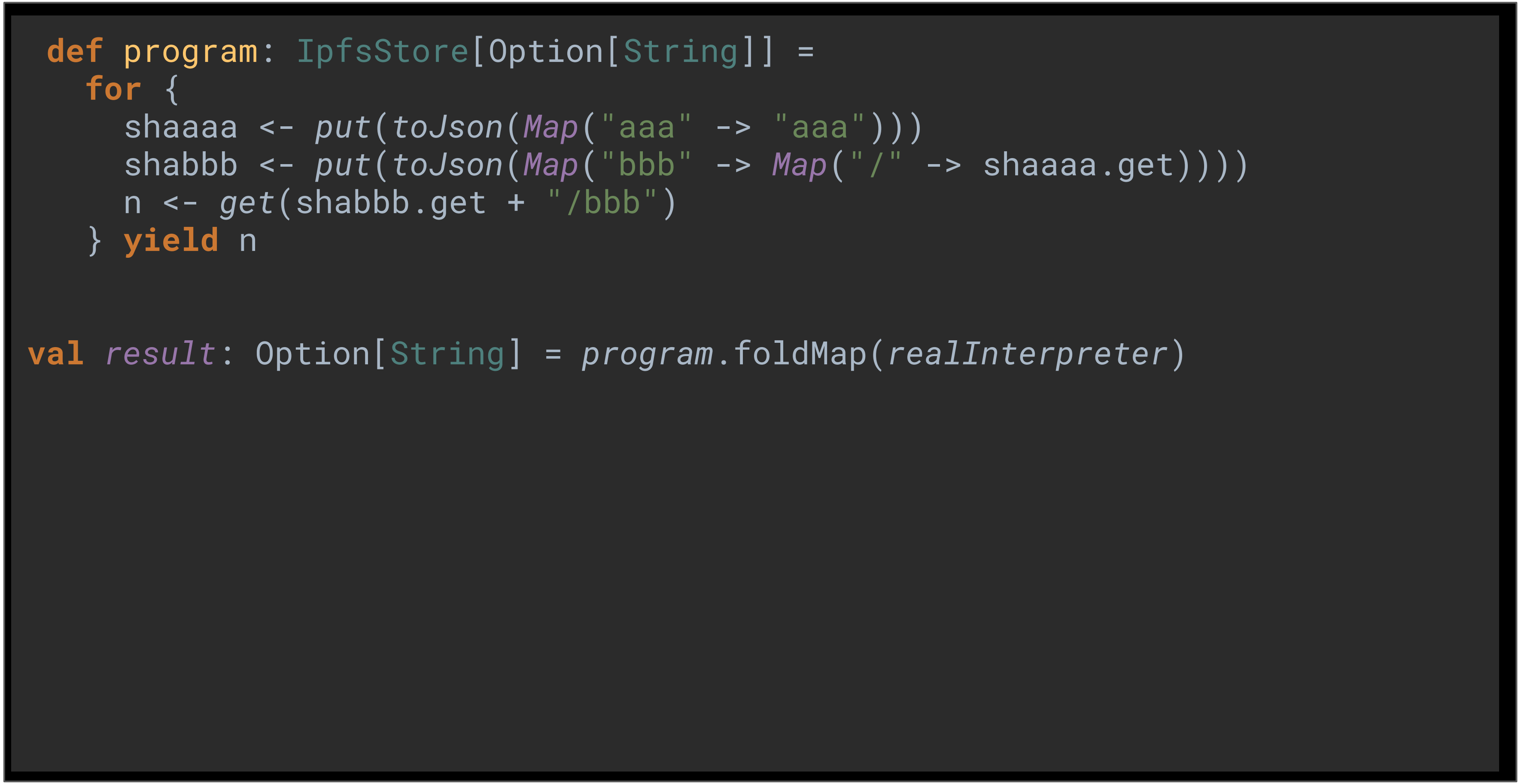
```
def testInterpreter: IpfsStoreA ~> Id  =
 new (IpfsStoreA ~> Id) {

   // a very simple (and imprecise) key-value store
   val Ipfss = mutable.Map.empty[String, Any]

   def apply[A](fa: IpfsStoreA[A]): Id[A] =
     fa match {
       case Put(value) =>
         // ipfs dag put value
         val shaKey = sha256String(value)
         val valX: Option[Any] = JSON.parseFull(value)
         if (valX.isEmpty) {
           return None
         }
         Ipfss(shaKey) = valX.get
         Some(shaKey)
       case Get(path) =>
         // ipfs dag get value
         pathTracker(Ipfss, path).map(_.asInstanceOf[String])
     }
 }
```



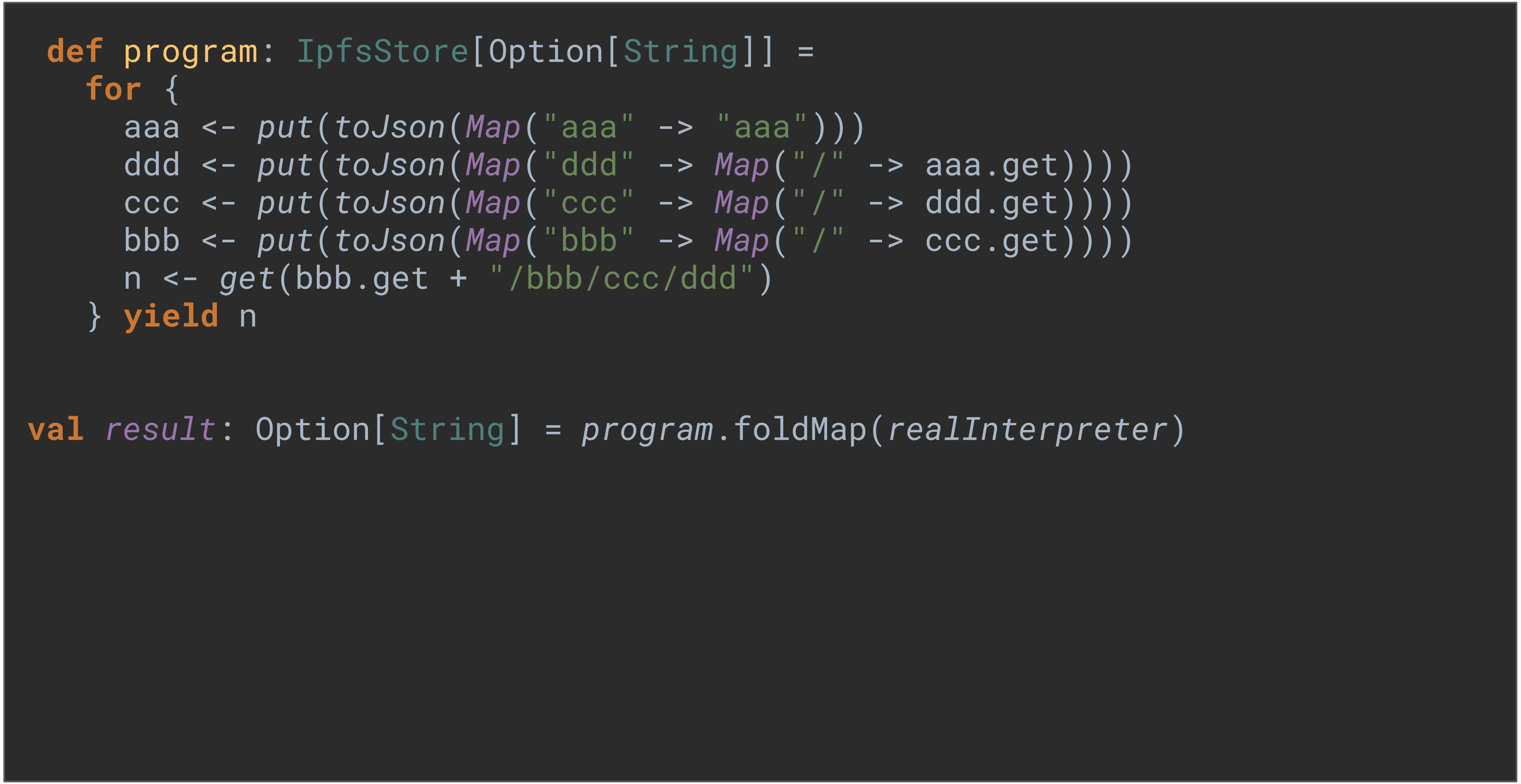
```
 def program: IpfsStore[Option[String]] =
   for {
     _ <- put(toJson(Map("aaa" -> "aaa")))
     _ <- put(toJson(Map("bbb" -> Map("/" -> sha256String(toJson(Map("aaa"
-> "aaa")))))))

     n <- get(sha256String(toJson(Map("bbb" -> Map("/" ->
sha256String(toJson(Map("aaa" -> "aaa"))))))))+"/bbb")

   } yield n

val result: Option[String] = program.foldMap(testInterpreter)
```



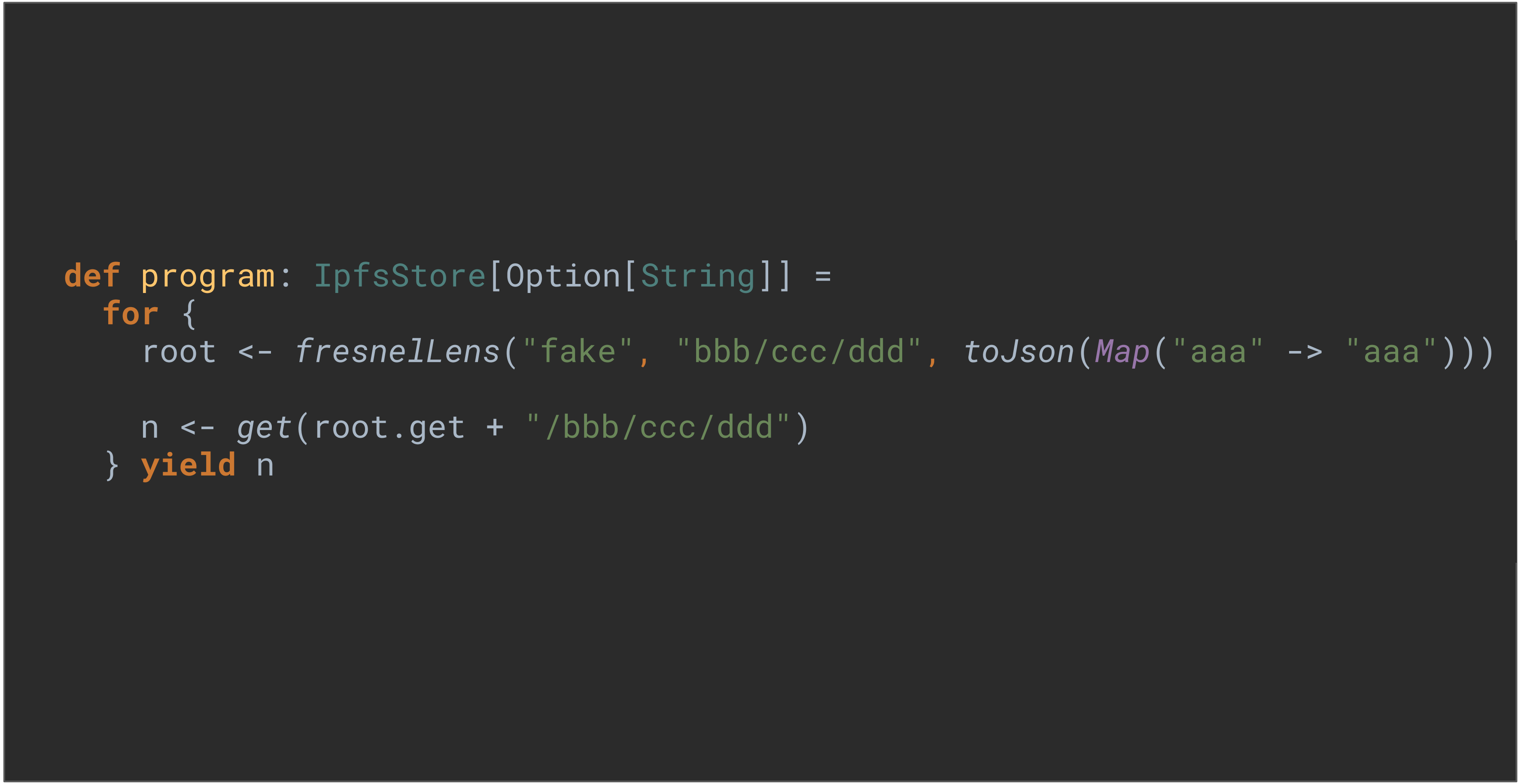
```
 def program: IpfsStore[Option[String]] =
   for {
     _ <- put(toJson(Map("aaa" -> "aaa")))
     _ <- put(toJson(Map("bbb" -> Map("/" -> sha256String(toJson(Map("aaa"
-> "aaa")))))))

     n <- get(sha256String(toJson(Map("bbb" -> Map("/" ->
sha256String(toJson(Map("aaa" -> "aaa")))))))+"/bbb")

   } yield n

val result: Option[String] = program.foldMap(testInterpreter)


… will not work on real interpreter, why?
```



```scala
 def program: IpfsStore[Option[String]] =
   for {
     shaaaa <- put(toJson(Map("aaa" -> "aaa")))
     shabbb <- put(toJson(Map("bbb" -> Map("/" -> shaaaa.get))))
     n <- get(shabbb.get + "/bbb")
   } yield n


val result: Option[String] = program.foldMap(realInterpreter)
```

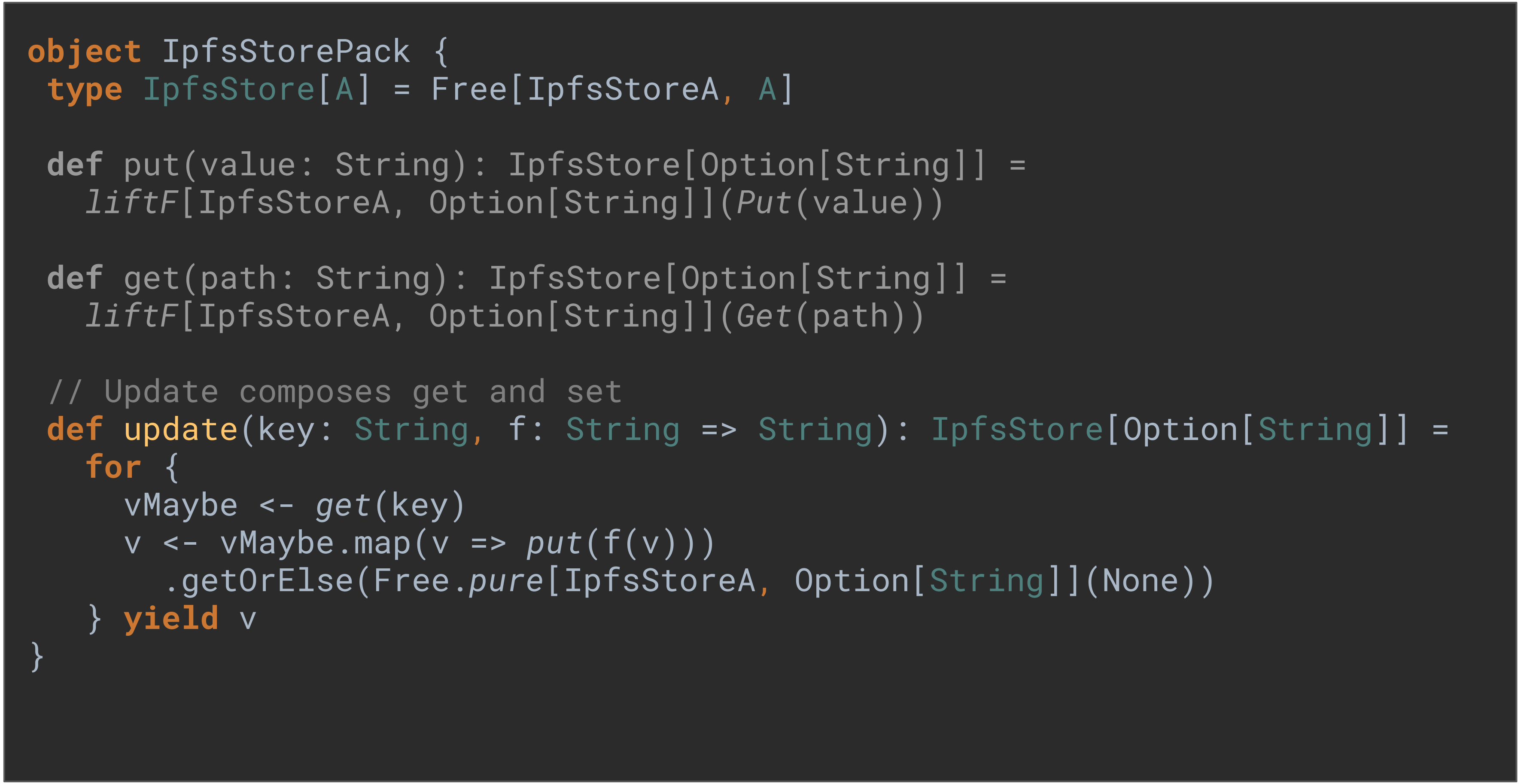


```
 def program: IpfsStore[Option[String]] =
   for {
     aaa <- put(toJson(Map("aaa" -> "aaa")))
     ddd <- put(toJson(Map("ddd" -> Map("/" -> aaa.get))))
     ccc <- put(toJson(Map("ccc" -> Map("/" -> ddd.get))))
     bbb <- put(toJson(Map("bbb" -> Map("/" -> ccc.get))))
     n <- get(bbb.get + "/bbb/ccc/ddd")
   } yield n


val result: Option[String] = program.foldMap(realInterpreter)
```

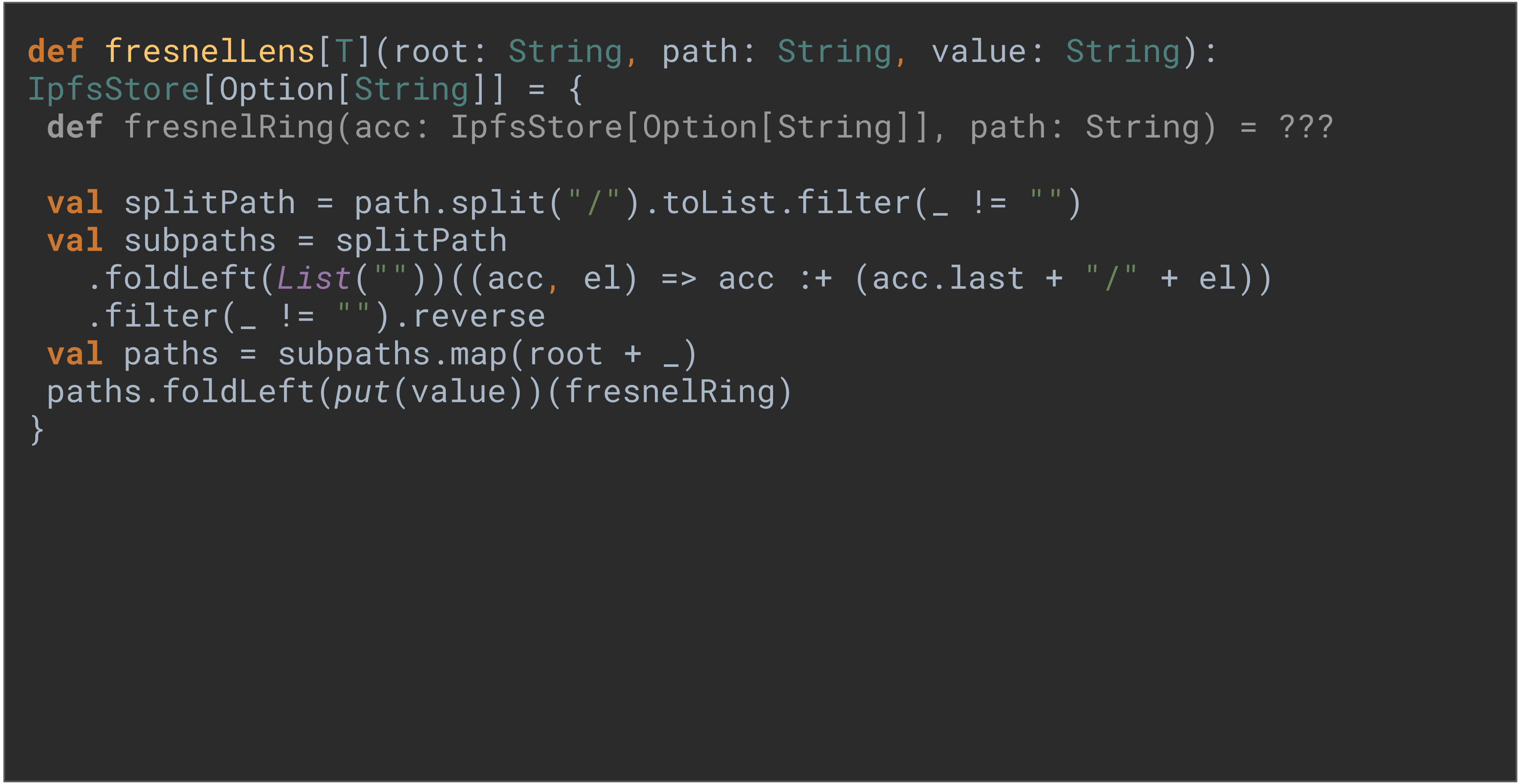


```
def program: IpfsStore[Option[String]] =
  for {
    root <- fresnelLens("fake", "bbb/ccc/ddd", toJson(Map("aaa" -> "aaa")))

    n <- get(root.get + "/bbb/ccc/ddd")
  } yield n
```

Fresnel Lens

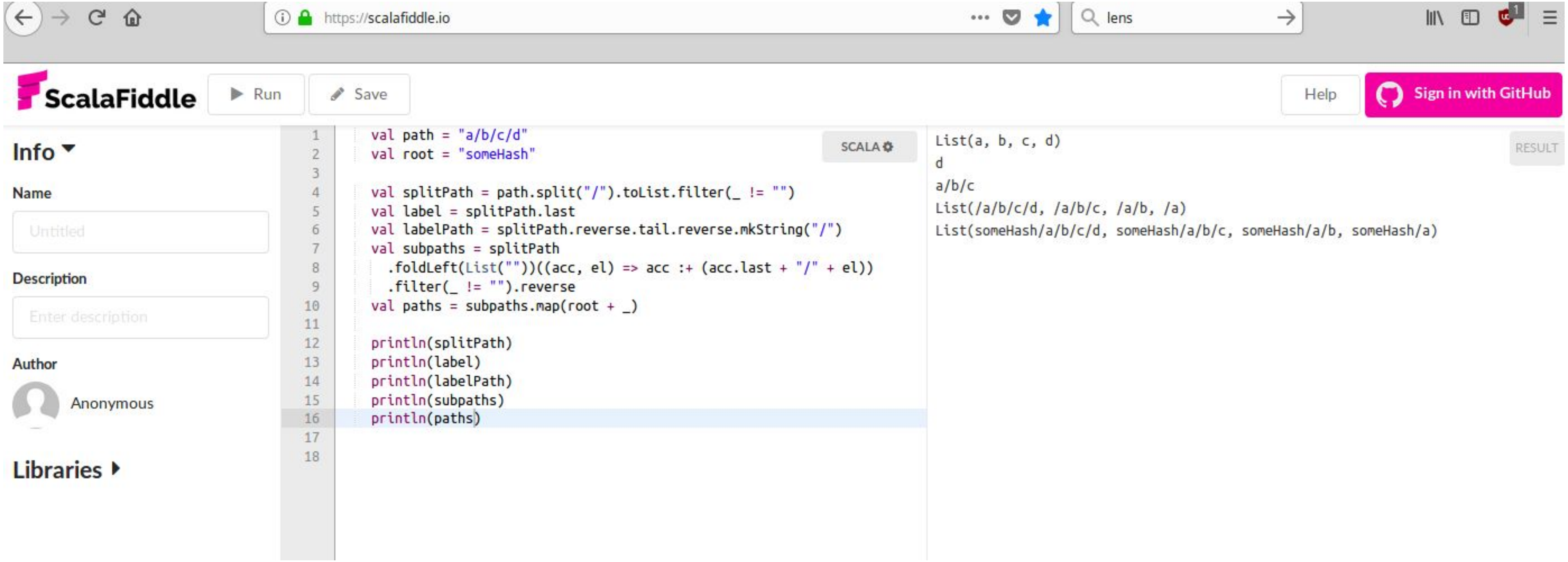


```scala
object IpfsStorePack {
 type IpfsStore[A] = Free[IpfsStoreA, A]

 def put(value: String): IpfsStore[Option[String]] =
   liftF[IpfsStoreA, Option[String]](Put(value))

 def get(path: String): IpfsStore[Option[String]] =
   liftF[IpfsStoreA, Option[String]](Get(path))

 // Update composes get and set
 def update(key: String, f: String => String): IpfsStore[Option[String]] =
   for {
     vMaybe <- get(key)
     v <- vMaybe.map(v => put(f(v)))
       .getOrElse(Free.pure[IpfsStoreA, Option[String]](None))
   } yield v
}
```



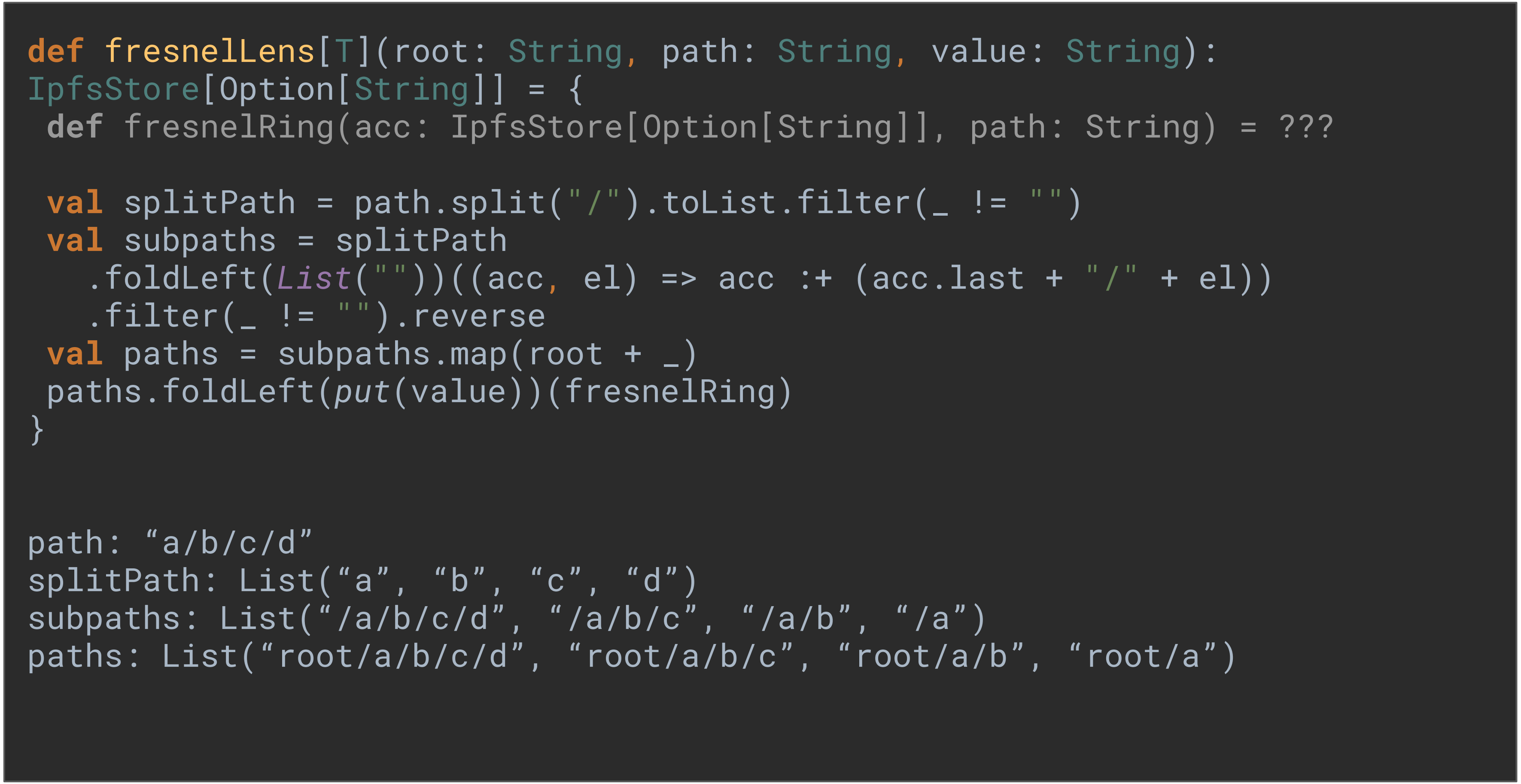
```
def fresnelLens[T](root: String, path: String, value: String):
IpfsStore[Option[String]] = {
 def fresnelRing(acc: IpfsStore[Option[String]], path: String) = ???

 val splitPath = path.split("/").toList.filter(_ != "")
 val subpaths = splitPath
   .foldLeft(List(""))((acc, el) => acc :+ (acc.last + "/" + el))
   .filter(_ != "").reverse
 val paths = subpaths.map(root + _)
 paths.foldLeft(put(value))(fresnelRing)
}
```

```scala
def fresnelLens[T](root: String, path: String, value: String):
IpfsStore[Option[String]] = {
 def fresnelRing(acc: IpfsStore[Option[String]], path: String) = ???

 val splitPath = path.split("/").toList.filter(_ != "")
 val subpaths = splitPath
    .foldLeft(List(""))((acc, el) => acc :+ (acc.last + "/" + el))
    .filter(_ != "").reverse
 val paths = subpaths.map(root + _)
 paths.foldLeft(put(value))(fresnelRing)
}
```

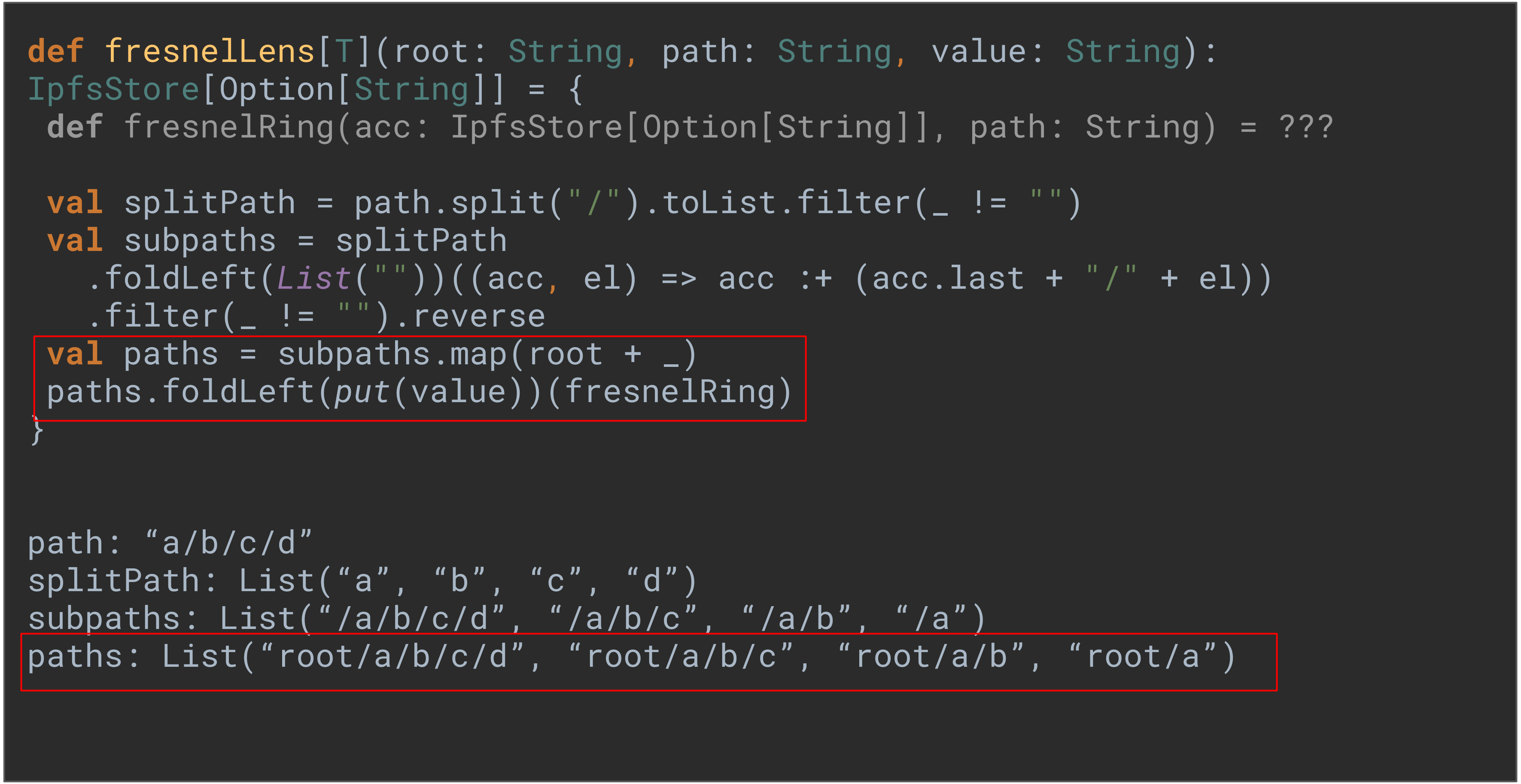


```scala
val path = "a/b/c/d"
val root = "someHash"

val splitPath = path.split("/").toList.filter(_ != "")
val label = splitPath.last
val labelPath = splitPath.reverse.tail.reverse.mkString("/")
val subpaths = splitPath
  .foldLeft(List(""))((acc, el) => acc :+ (acc.last + "/" + el))
  .filter(_ != "").reverse
val paths = subpaths.map(root + _)

println(splitPath)
println(label)
println(labelPath)
println(subpaths)
println(paths)
```

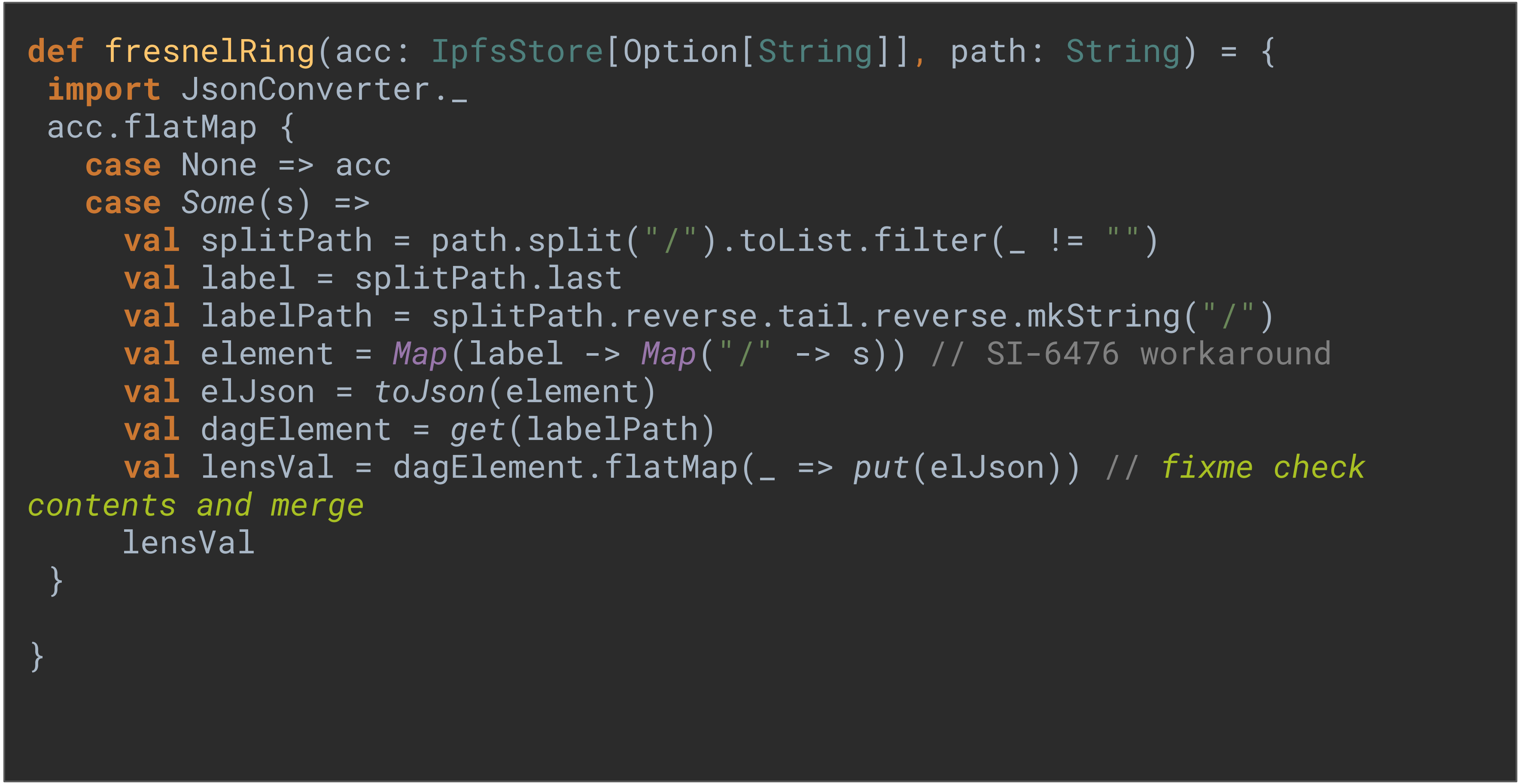
```
List(a, b, c, d)
d
a/b/c
List(/a/b/c/d, /a/b/c, /a/b, /a)
List(someHash/a/b/c/d, someHash/a/b/c, someHash/a/b, someHash/a)
```



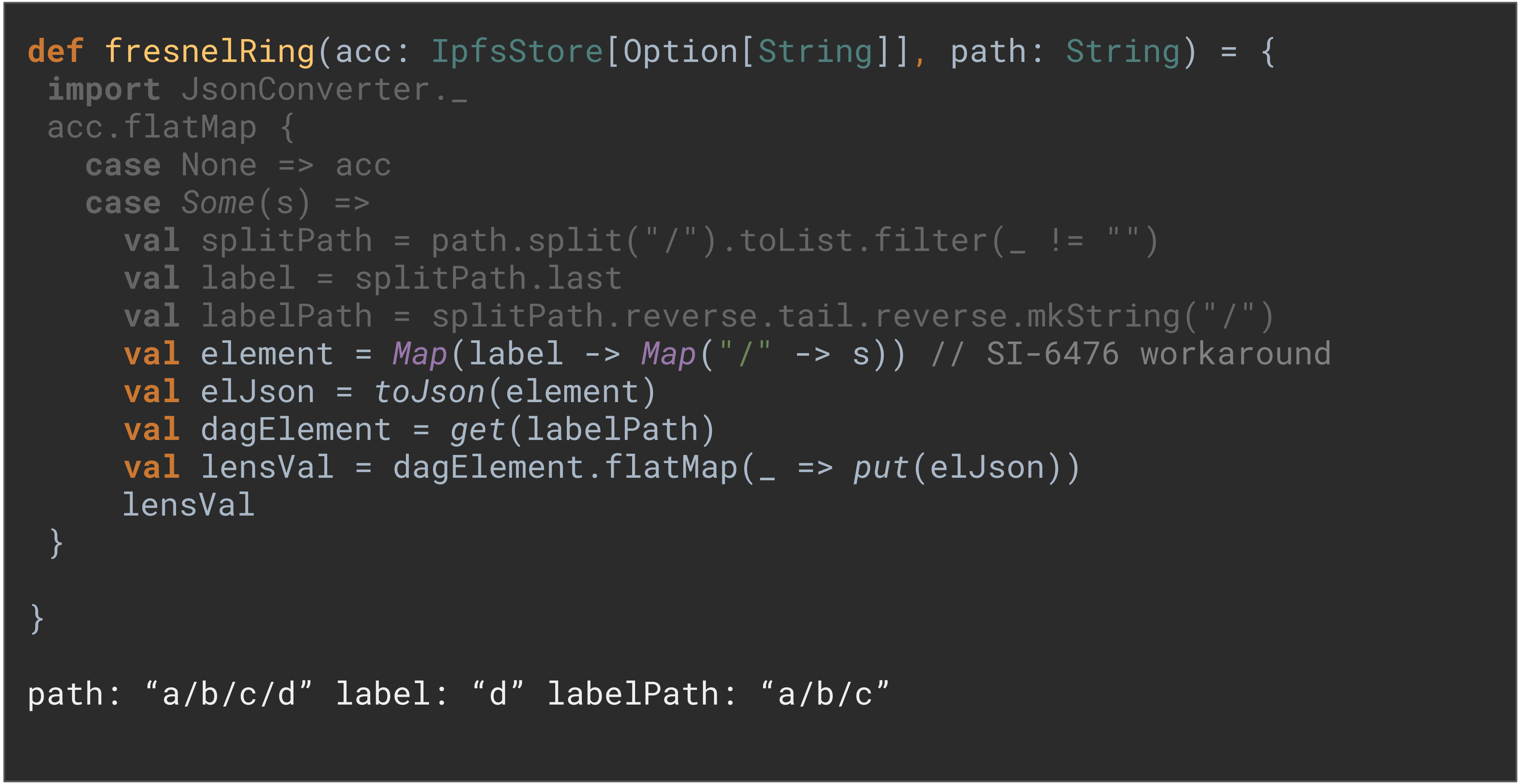
```
def fresnelLens[T](root: String, path: String, value: String):
IpfsStore[Option[String]] = {
 def fresnelRing(acc: IpfsStore[Option[String]], path: String) = ???

 val splitPath = path.split("/").toList.filter(_ != "")
 val subpaths = splitPath
   .foldLeft(List(""))((acc, el) => acc :+ (acc.last + "/" + el))
   .filter(_ != "").reverse
 val paths = subpaths.map(root + _)
 paths.foldLeft(put(value))(fresnelRing)
}


path: “a/b/c/d”
splitPath: List(“a”, “b”, “c”, “d”)
subpaths: List(“/a/b/c/d”, “/a/b/c”, “/a/b”, “/a”)
paths: List(“root/a/b/c/d”, “root/a/b/c”, “root/a/b”, “root/a”)
```



```
def fresnelLens[T](root: String, path: String, value: String):
IpfsStore[Option[String]] = {
 def fresnelRing(acc: IpfsStore[Option[String]], path: String) = ???

 val splitPath = path.split("/").toList.filter(_ != "")
 val subpaths = splitPath
   .foldLeft(List(""))((acc, el) => acc :+ (acc.last + "/" + el))
   .filter(_ != "").reverse
 val paths = subpaths.map(root + _)
 paths.foldLeft(put(value))(fresnelRing)
}



path: “a/b/c/d”
splitPath: List(“a”, “b”, “c”, “d”)
subpaths: List(“/a/b/c/d”, “/a/b/c”, “/a/b”, “/a”)
paths: List(“root/a/b/c/d”, “root/a/b/c”, “root/a/b”, “root/a”)
```

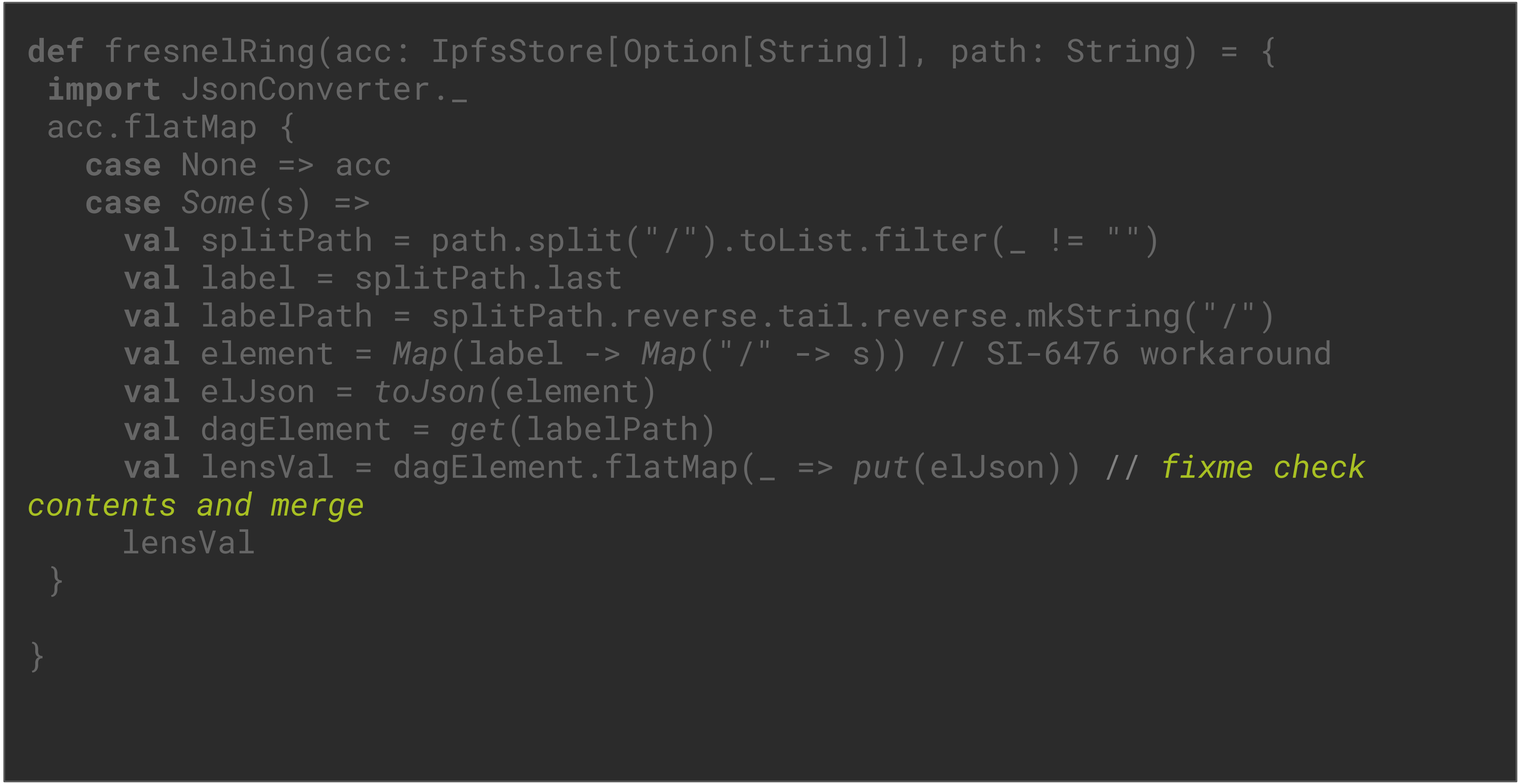


```
def fresnelRing(acc: IpfsStore[Option[String]], path: String) = {
 import JsonConverter._
 acc.flatMap {
   case None => acc
   case Some(s) =>
     val splitPath = path.split("/").toList.filter(_ != "")
     val label = splitPath.last
     val labelPath = splitPath.reverse.tail.reverse.mkString("/")
     val element = Map(label -> Map("/" -> s)) // SI-6476 workaround
     val elJson = toJson(element)
     val dagElement = get(labelPath)
     val lensVal = dagElement.flatMap(_ => put(elJson)) // fixme check
contents and merge
     lensVal
 }

}
```



```scala
def fresnelRing(acc: IpfsStore[Option[String]], path: String) = {
 import JsonConverter._
 acc.flatMap {
   case None => acc
   case Some(s) =>
     val splitPath = path.split("/").toList.filter(_ != "")
     val label = splitPath.last
     val labelPath = splitPath.reverse.tail.reverse.mkString("/")
     val element = Map(label -> Map("/" -> s)) // SI-6476 workaround
     val elJson = toJson(element)
     val dagElement = get(labelPath)
     val lensVal = dagElement.flatMap(_ => put(elJson))
     lensVal
 }

}
```

path: “a/b/c/d” label: “d” labelPath: “a/b/c”



```
def fresnelRing(acc: IpfsStore[Option[String]], path: String) = {
 import JsonConverter._
 acc.flatMap {
   case None => acc
   case Some(s) =>
     val splitPath = path.split("/").toList.filter(_ != "")
     val label = splitPath.last
     val labelPath = splitPath.reverse.tail.reverse.mkString("/")
     val element = Map(label -> Map("/" -> s)) // SI-6476 workaround
     val elJson = toJson(element)
     val dagElement = get(labelPath)
     val lensVal = dagElement.flatMap(_ => put(elJson)) // fixme check
contents and merge
     lensVal
 }

}
```

```scala
def program: IpfsStore[Option[String]] =
  for {
    r1 <- fresnelLens("fake", "bbb/zzz", toJson(Map("zzz" -> "zzz")))
    root <- fresnelLens(r1.get, "bbb/ccc/ddd", toJson(Map("aaa" -> "aaa")))
    n <- get(root.get + "/bbb")
  } yield n
```

```
{"ccc":{"/":"zdpuAq2Wq1mgwTw9WGDeZg6BtuwgRGgrvXfzr9fxXnr68VpDM"},"zzz":{"/"
:"zdpuAr1fEERJntqtLojcLTdhoHnsDBW16qWbWb5CAkLH7EoZD"}}
```

```scala
val lensVal = dagElement.flatMap(v => v.fold(put(elJson))(existingJson => {
 val oldJson = JSON.parseFull(existingJson).get
 val elementVal: Any = Map("/" -> s)
 val newJson = oldJson match {
   case m: Map[String, Any] =>
     m + (label -> elementVal)
   case _ => label -> elementVal
 }
 put(toJson(newJson))
}))
```

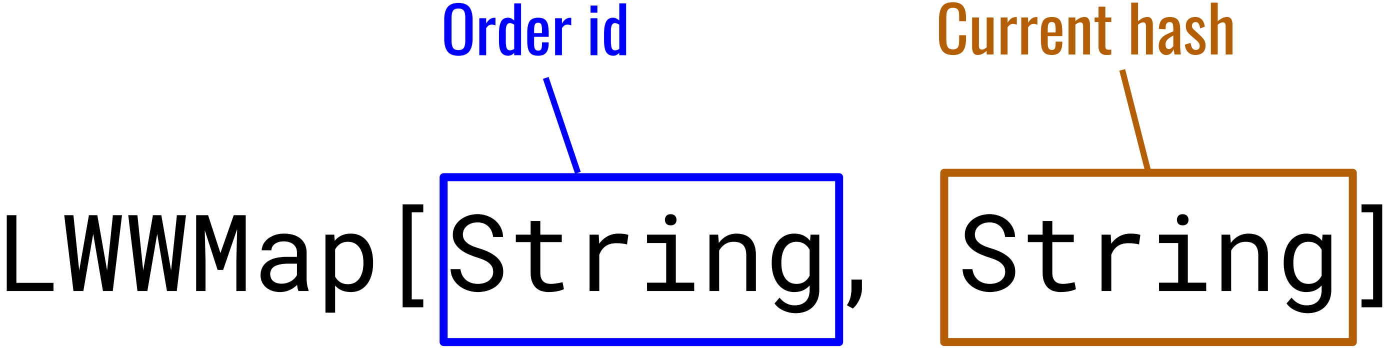
Now we only need to map the stream of hashes to the order id

akka

```
LWWMap[String, String]
```

Order id

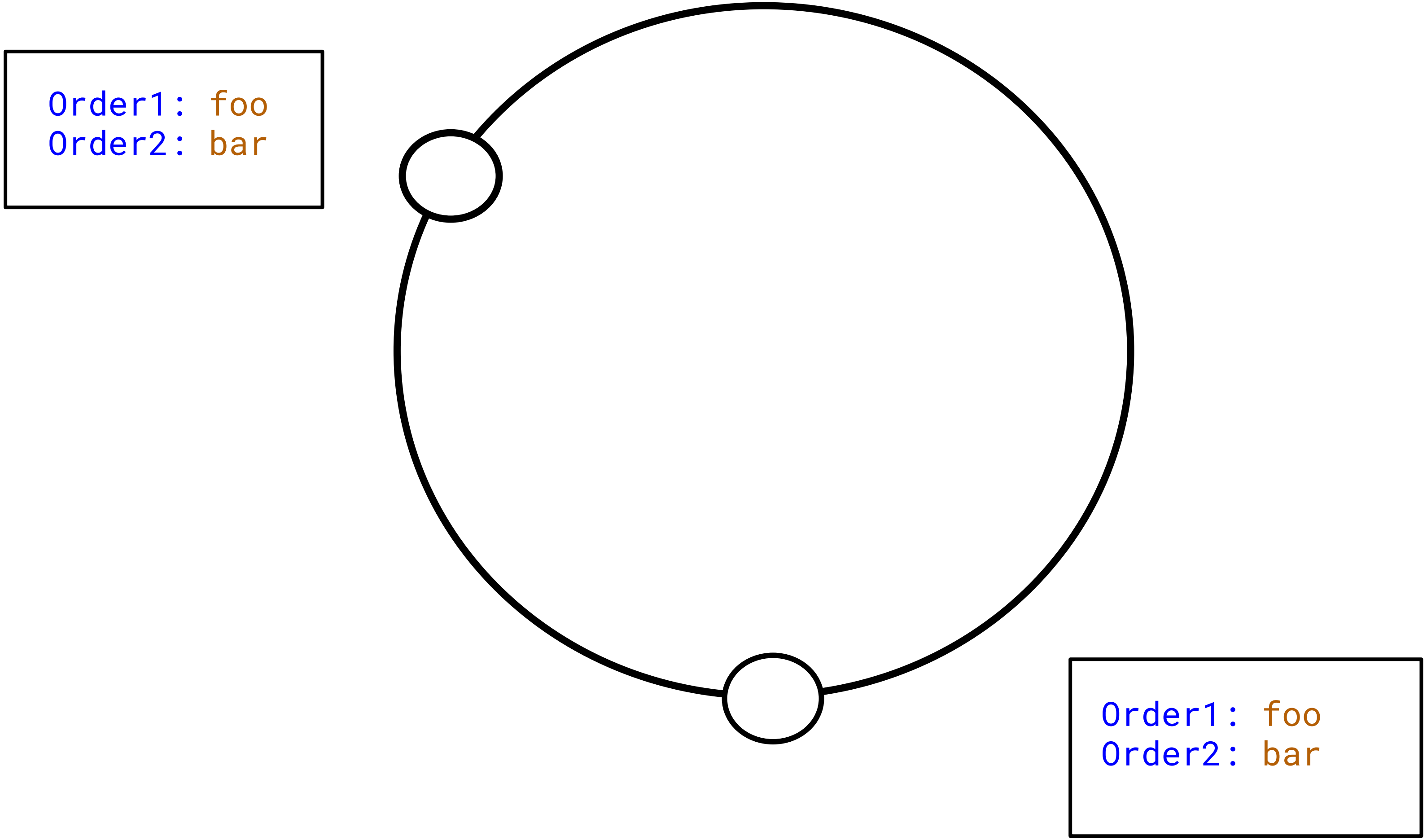
```
LWWMap[String, String]
```



Order id

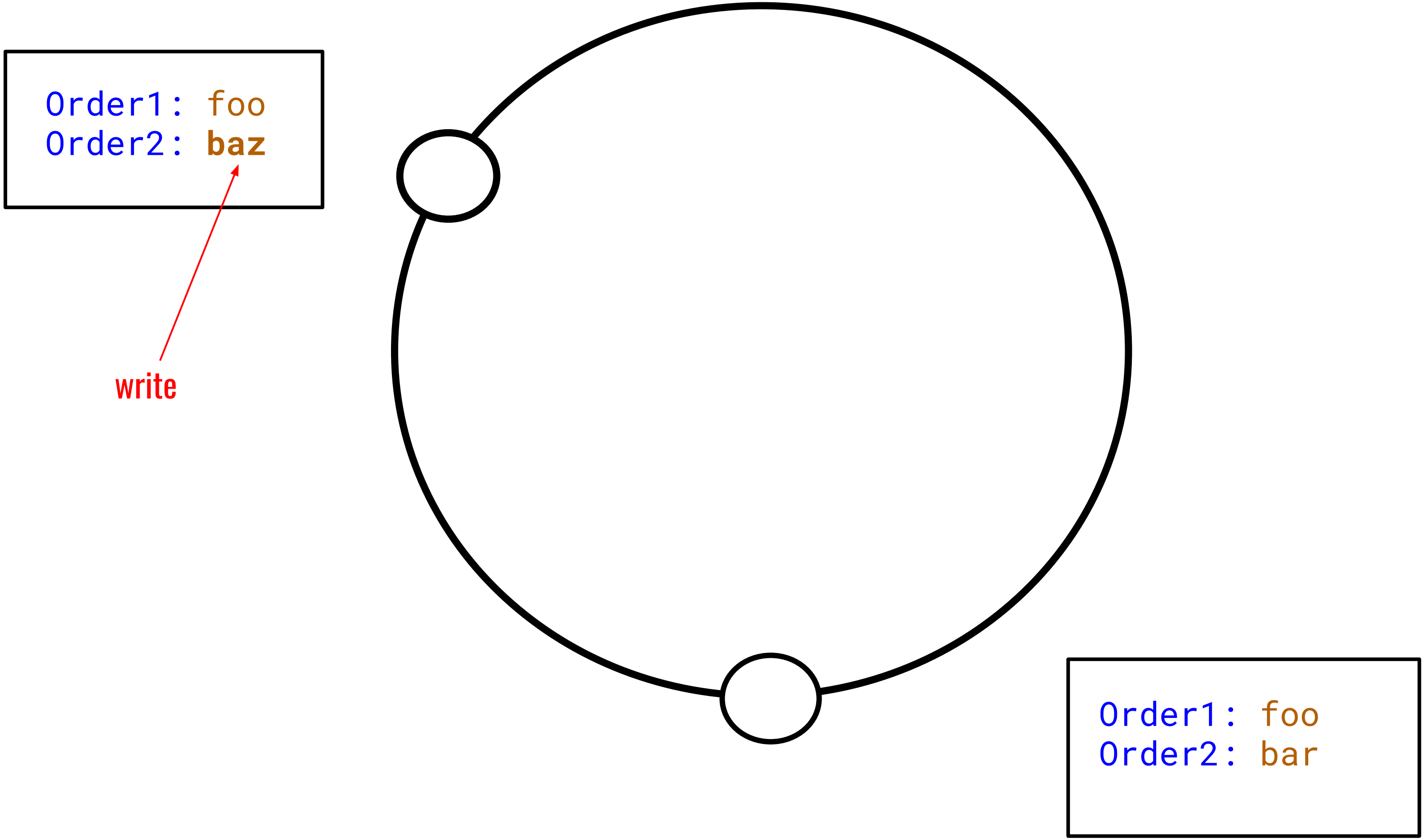
Current hash

```
LWWMap[String, String]
```



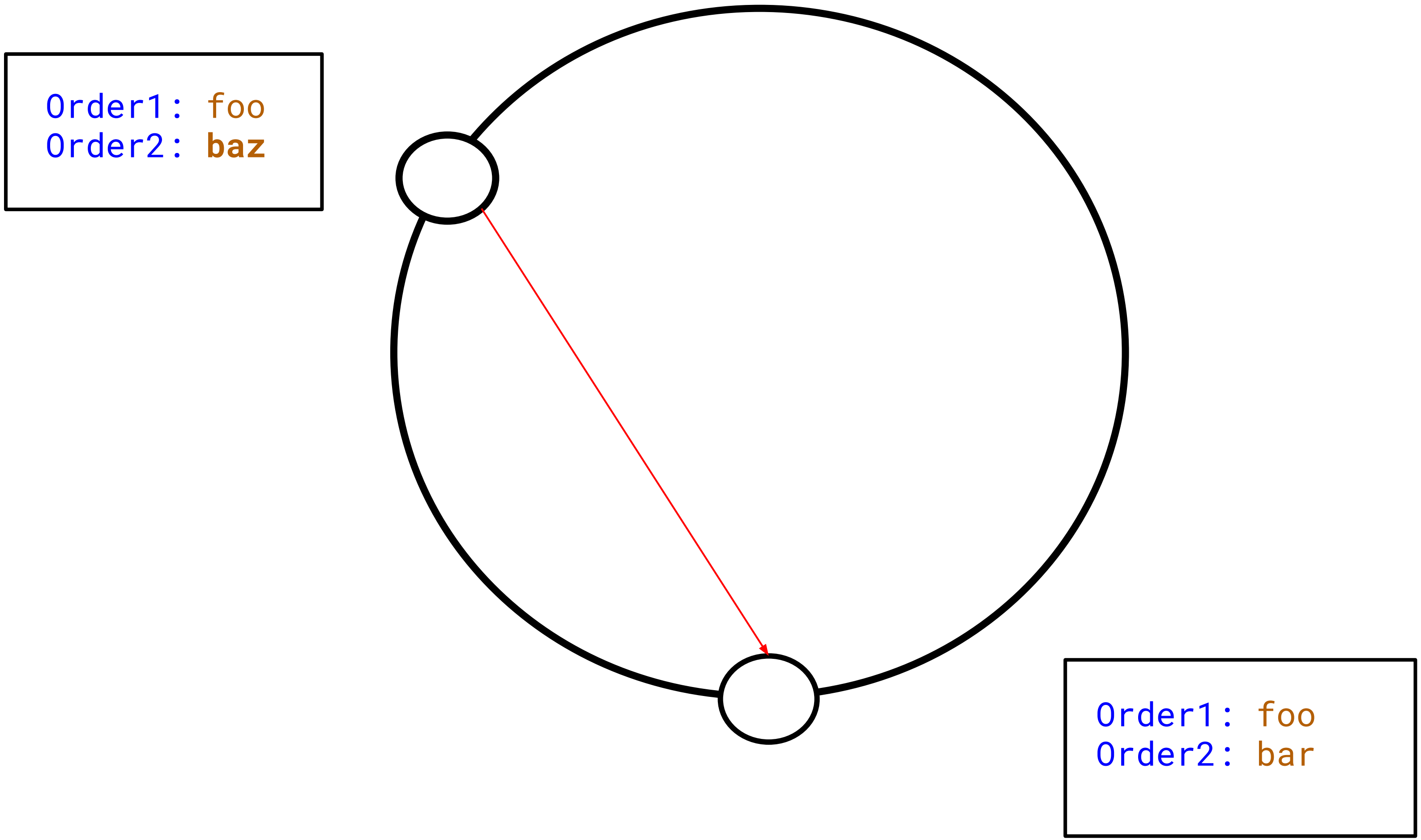
Order1: foo
Order2: bar

Order1: foo
Order2: bar



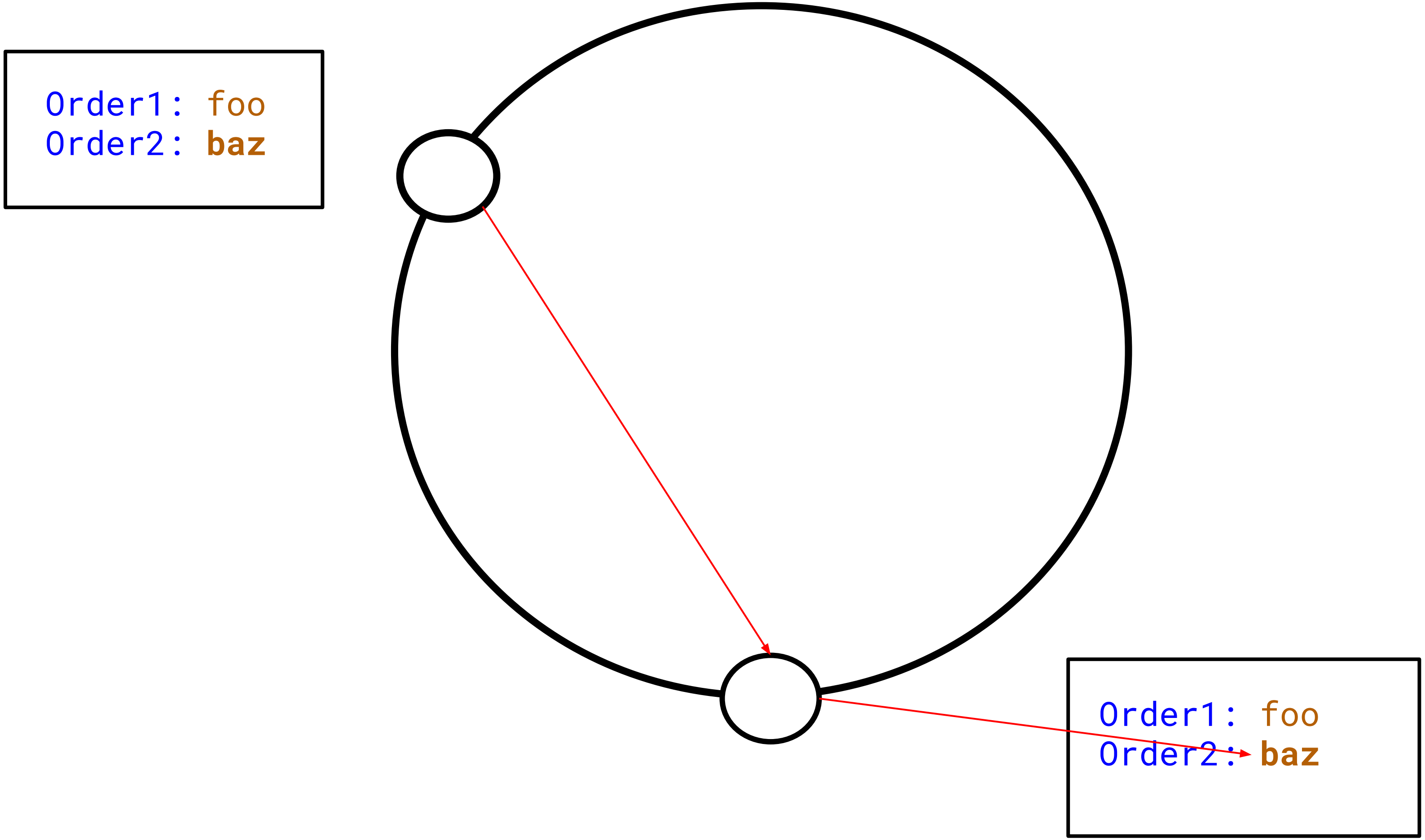
Order1: foo
Order2: **baz**

Order1: foo
Order2: bar



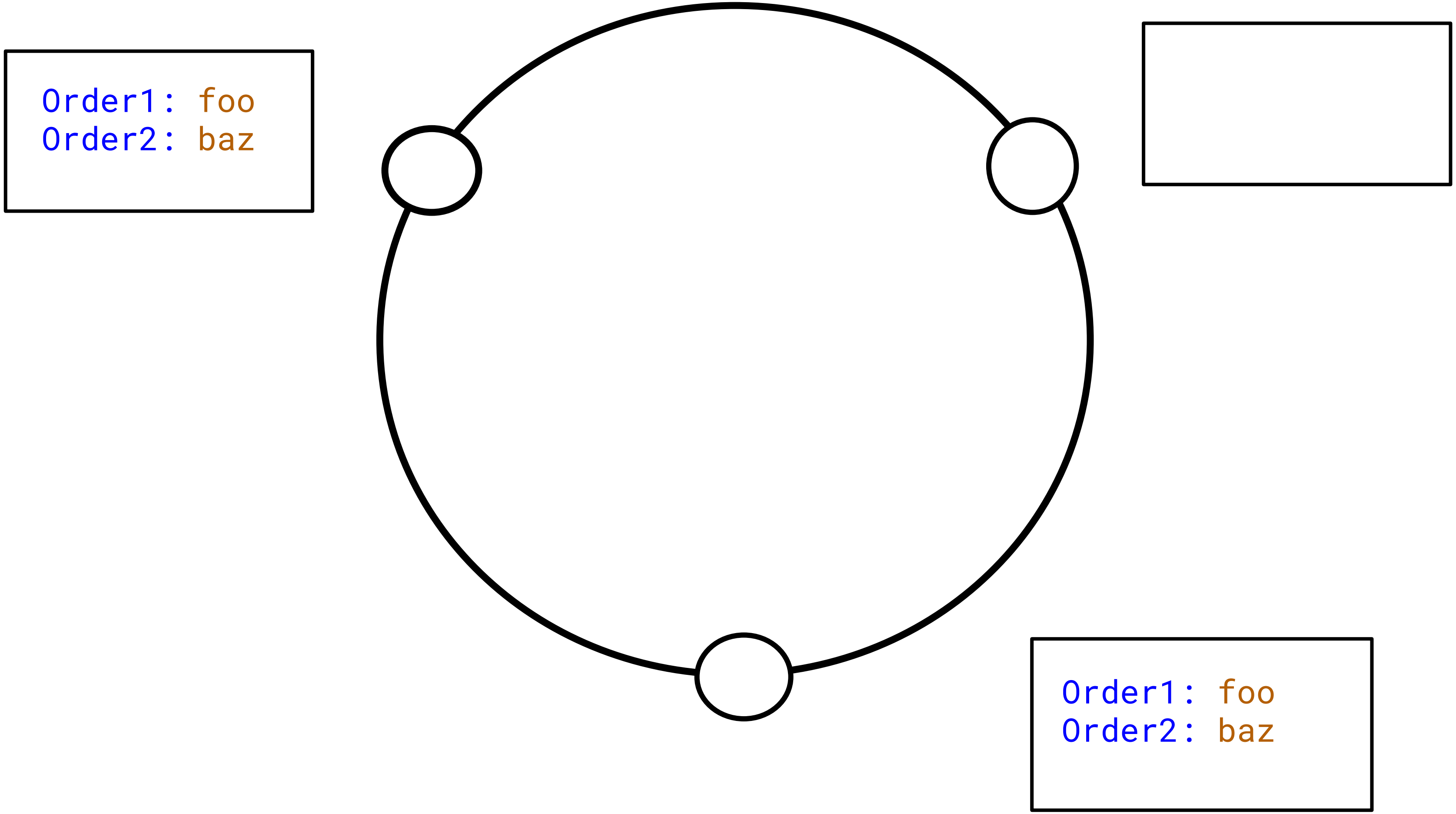
```
Order1: foo
Order2: **baz**
```

```
Order1: foo
Order2: bar
```



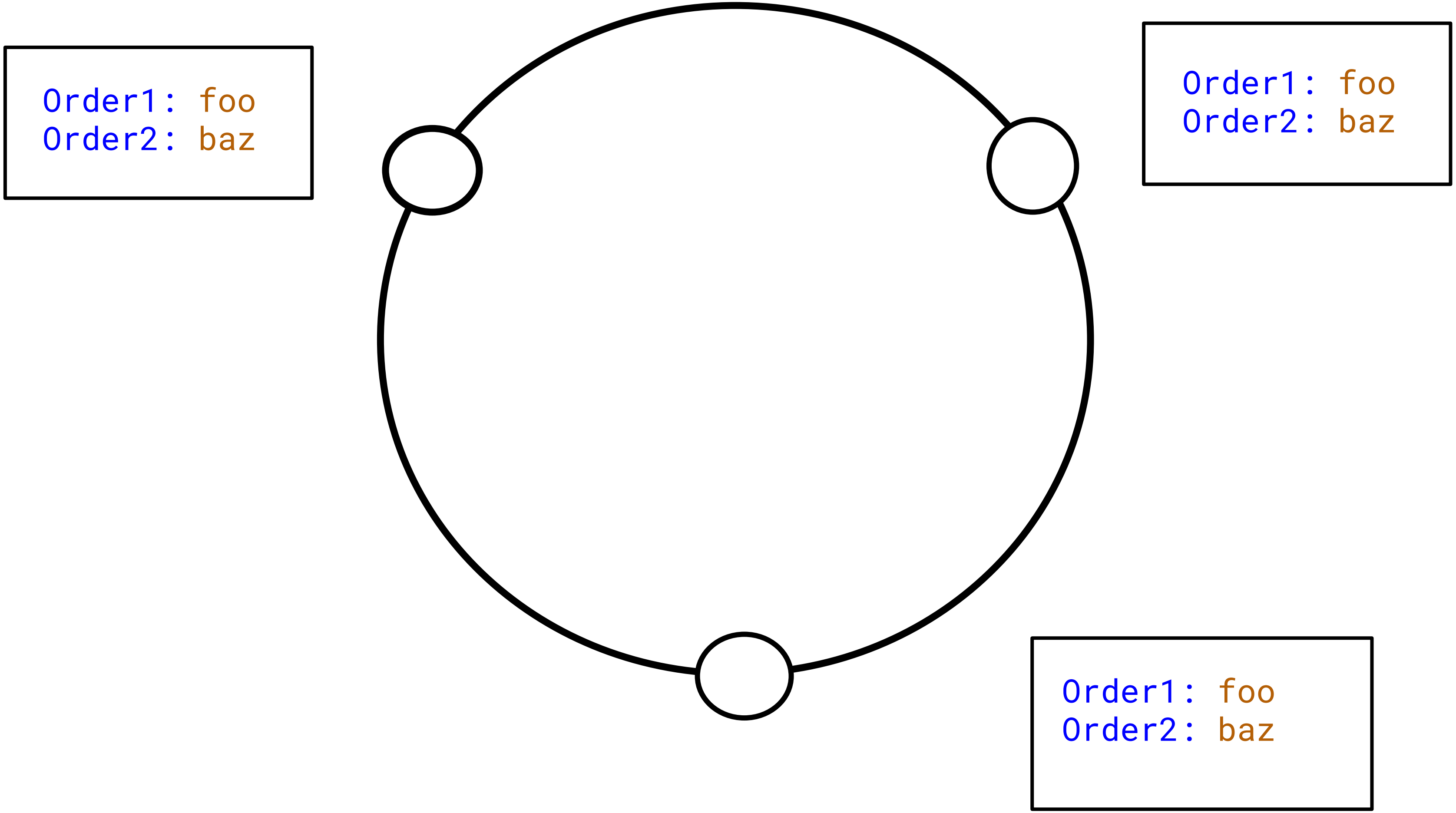
Order1: foo
Order2: **baz**

Order1: foo
Order2: **baz**



Order1: foo
Order2: baz

Order1: foo
Order2: baz



Order1: foo
Order2: baz

Order1: foo
Order2: baz

Order1: foo
Order2: baz

Thank you!