



Operations on Type Future

Effective Programming in Scala

Operations on Type Future

In the previous lesson, we have seen that, in Scala, we model asynchronous computations as methods returning Future.

The execution of these computations eventually fails or succeeds.

Orchestrating asynchronous computations is like building a directed graph of computations.

In this lesson, we will show how to write such code in practice.

Transforming a Successful Asynchronous Computation

Similar to what we saw with Try (modeling computations that may fail) and Validated (modeling validated values), we can use map (and flatMap) to transform the result of a successful asynchronous computation.

For instance, assuming we use an in-memory database, we have the following operations:

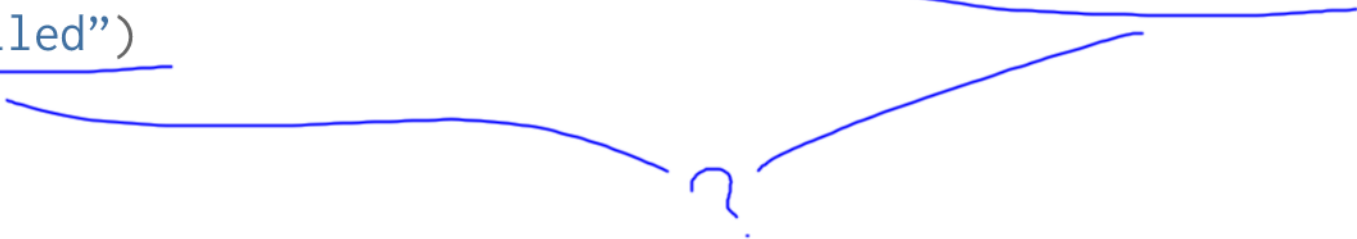
```
def bcrypt(saltRound: Int, password: String): Future[Seq[Byte]]  
def insertUser(login: String, passwordHash: Seq[Byte]): User
```

Computing a password hash and then inserting a user is achieved like this:

```
def hashPasswordAndInsert(name: String, password: String): Future[User] =  
  bcrypt(10, password)  
    .map(passwordHash => insertUser(name, passwordHash))
```

Concurrent Execution Flows

```
println("calling bcrypt")  
bcrypt(10, "asimplepassword").map(hash => println(s"bcrypt result: $hash"))  
println("bcrypt called")
```



A diagram consisting of blue lines. One line starts from the underlined line `println("bcrypt called")` and curves downwards and to the right, ending at a question mark. Another line starts from the `println(s"bcrypt result: $hash")` line in the `map` block and curves upwards and to the left, also ending at the same question mark. This visualizes the uncertainty of the relative execution order of these two print statements.

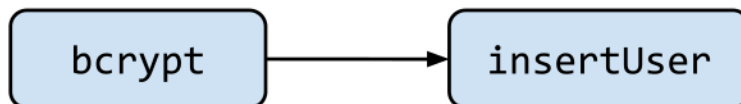
We can say that:

- ▶ the message "calling bcrypt" will be printed *before* the bcrypt method is called,
- ▶ the message "bcrypt result: ..." will be printed *after* the bcrypt method call has terminated.

But we have **no guarantee** that the message "bcrypt called" will be printed *before* or *after* the message "bcrypt result: ..."!

Chaining Asynchronous Computations

In practice, you are likely to use a remote database. In such a case, the operation `insertUser` would also be asynchronous. So, you want to chain two asynchronous computations:

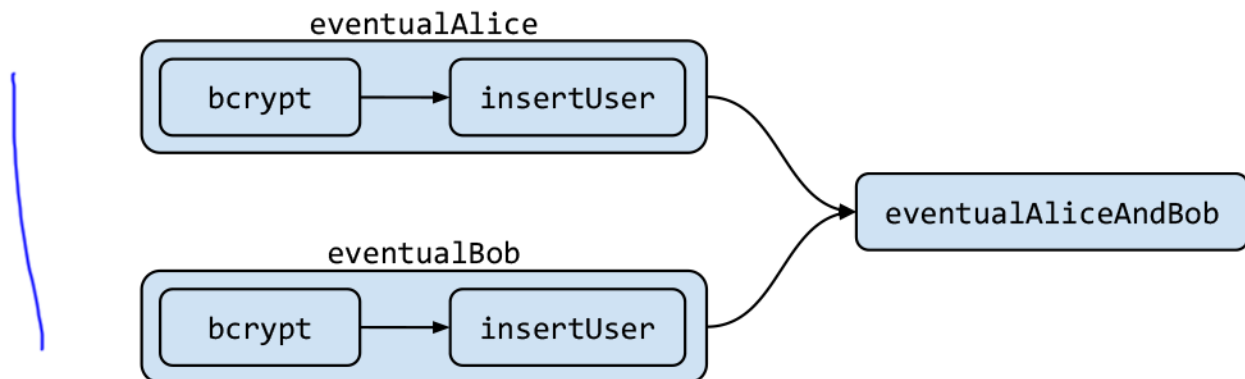


```
def bcrypt(saltRound: Int, password: String): Future[Seq[Byte]]
def insertUser(login: String, passwordHash: Seq[Byte]): Future[User]
```

We achieve this with the operation `flatMap`:

```
def hashPasswordAndInsert(name: String, password: String): Future[User] =
  bcrypt(10, password)
    .flatMap(passwordHash => insertUser(name, passwordHash))
```

Joining Two Future Values



We construct a Future value that completes when two Future values complete with the operation `zip`.

```
val eventualAlice: Future[User] =  
  hashPasswordAndInsert("Alice", "averycomplicatedpassword")  
val eventualBob: Future[User] =  
  hashPasswordAndInsert("Bob", "anothercomplicatedpassword")  
val eventualAliceAndBob: Future[(User, User)] =  
  eventualAlice.zip(eventualBob)
```

Joining an Arbitrary Number of Future Values

```
val userData: Seq[(String, String)] = Seq(  
  "Alice" -> "averycomplicatedpassword",  
  "Bob"    -> "anothercomplicatedpassword"  
)
```

```
val eventualUsers: Future[Seq[User]] =  
  Future.traverse(userData)(hashPasswordAndInsert)
```

The operation `traverse` runs as many asynchronous computations as there are elements in the collection, and returns a `Future` value eventually containing all the results in a collection.

The asynchronous computations are executed independently of each other, in no predefined order.

Failure Handling

So far, we have seen how to transform and chain successful asynchronous computations. What happens if an asynchronous computation fails?

We represent failures with “failed” Future values containing an exception. Like with Try, we can handle these failures with the operation recover:

```
val eventuallyInserted: Future[Boolean] =  
  hashPasswordAndInsert("Alice", "averycomplicatedpassword")  
    .map(_ => true)  
    .recover { case NonFatal(exception) => false }
```


Failure Handling (2)

Similarly to `recover`, another operation for handling failed Future values is `recoverWith`. The difference is that `recoverWith` *tries* to recover from the failure by running another asynchronous computation.

API Summary (Most Common Operations)

```
trait Future[A]:  
  def map[B](f: A => B): Future[B]  
  def zip[B](that: Future[B]): Future[(A, B)]  
  def flatMap[B](f: A => Future[B]): Future[B]  
  def recover(f: Throwable => A): Future[A]  
  def recoverWith(f: Throwable => Future[A]): Future[A]  
  
object Future:  
  def traverse[A, B](as: Seq[A])(f: A => Future[B]): Future[Seq[B]]
```

Aside: Category Theory Primer

We have seen that the operations `map`, `zip`, `flatMap`, and `traverse` are defined on various types (`Seq`, `Try`, `Either`, `Future`). You might encounter them on other types as well.

These operations are indeed very common, they have been well studied, and you may hear about their most general form, named after category theory:

- ▶ `map` is defined by the type class `Functor`
- ▶ `zip` is defined by the type class `Applicative`
- ▶ `flatMap` is defined the type class `Monad`
- ▶ `traverse` is defined the type class `Traverse`

We hope that our course gave you a good intuition of what these operations do!

Summary

map and flatMap let you define what to do *after* a Future value is resolved. They build **sequential** paths of computation.

zip and traverse let you “join” together an arbitrary number of Future values. They build **parallel** paths of computation.

recover and recoverWith let you handle failures.