Exercise 4: Futures

TECHNISCHE UNIVERSITÄT DARMSTADT

Concepts and Technologies for Distributed Systems and Big Data Processing – SS 2017

Solution 1 Using *Future* combinators

The following code reads the content of a file (`IO.read`), parses it as JSON (`JSON.parse`) and computes a result (`someComplexComputation`) based on the JSON dataset if the dataset is not marked to be ignored (`!json.hasProperty(Ignore)`). The respective methods return their result as future for the sake of performance.

```
val content = IO.read("filename.json")
content.onComplete {
  case Success(content) =>
    val json = JSON.parse(content)
    json.onComplete {
      case Success(json) =>
        if (!json.hasProperty(Ignore)) {
          val result = Future {
            someComplexComputation(json)
          }

          result.onComplete {
            case Success(result) =>
              println(s"final result: $result")
            case Failure(_) =>
              println("could not compute result")
          }
        }
        else
          println("could not compute result")

      case Failure(_) =>
        println("could not compute result")
    }

  case Failure(_) =>
    println("could not compute result")
}
```

Rewrite the computation using `Future` combinators such as `flatMap`, `map` and `filter` instead of `onComplete` callbacks.

```
(IO.read("filename.json")
  flatMap JSON.parse
  filter { !_.hasProperty(Ignore) }
  map someComplexComputation
  onComplete {
    case Success(result) =>
      println(s"final result: $result")
    case Failure(_) =>
      println("could not compute result")
  })
```

Solution 2 Implementing *Future* combinators

Implement the `Future` `map` combinator in terms of `async`/`await`, which creates a new future by applying the function given by `mapping` to the result of the given future.

```
def map[T, U](future: Future[T], mapping: T => U): Future[U] =
  async {
    mapping(await(future))
  }
```

Implement the `Future` `filter` combinator in terms of `async`/`await`, which filters the value of the given future with the given `predicate`. If the value of the future satisfies the predicate, the result will hold the same value. Otherwise, the computation will fail.

```
def filter[T](future: Future[T], predicate: T => Boolean): Future[T] =
  async {
    val value = await(future)
    if (!predicate(value))
      throw new NoSuchElementException
    else
      value
  }
```