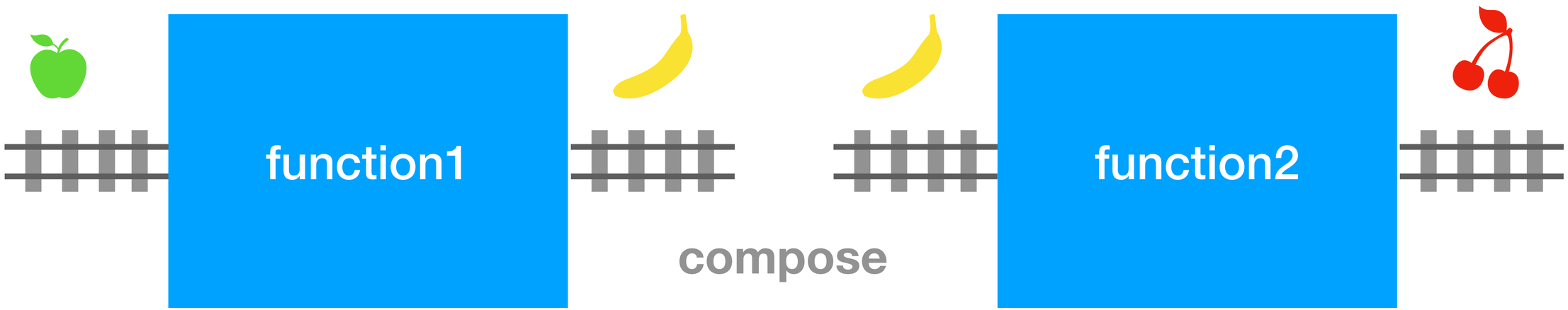
FP

Peter van Rijn

Function

Apple => Banana

Function Composition



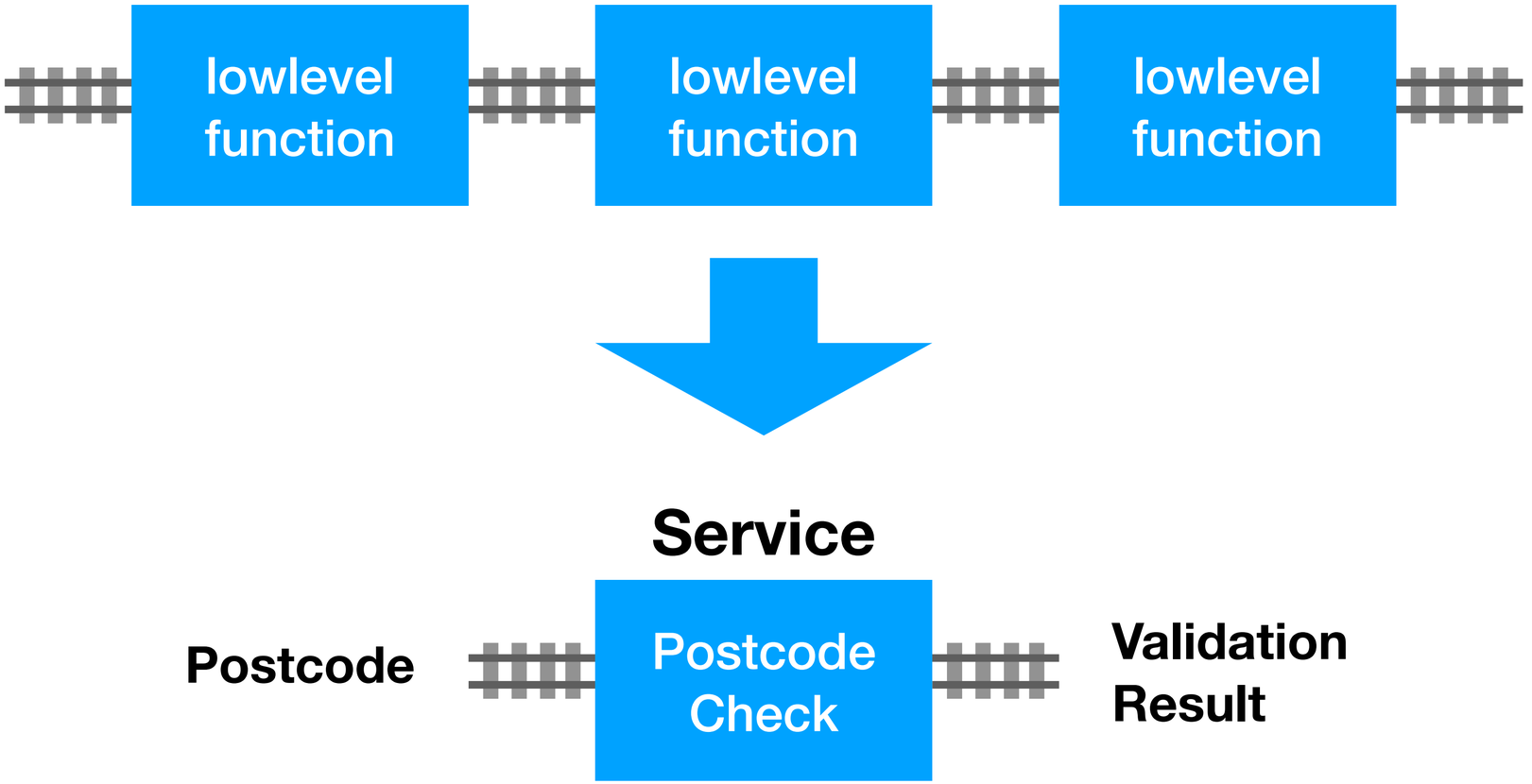
Apple => Banana

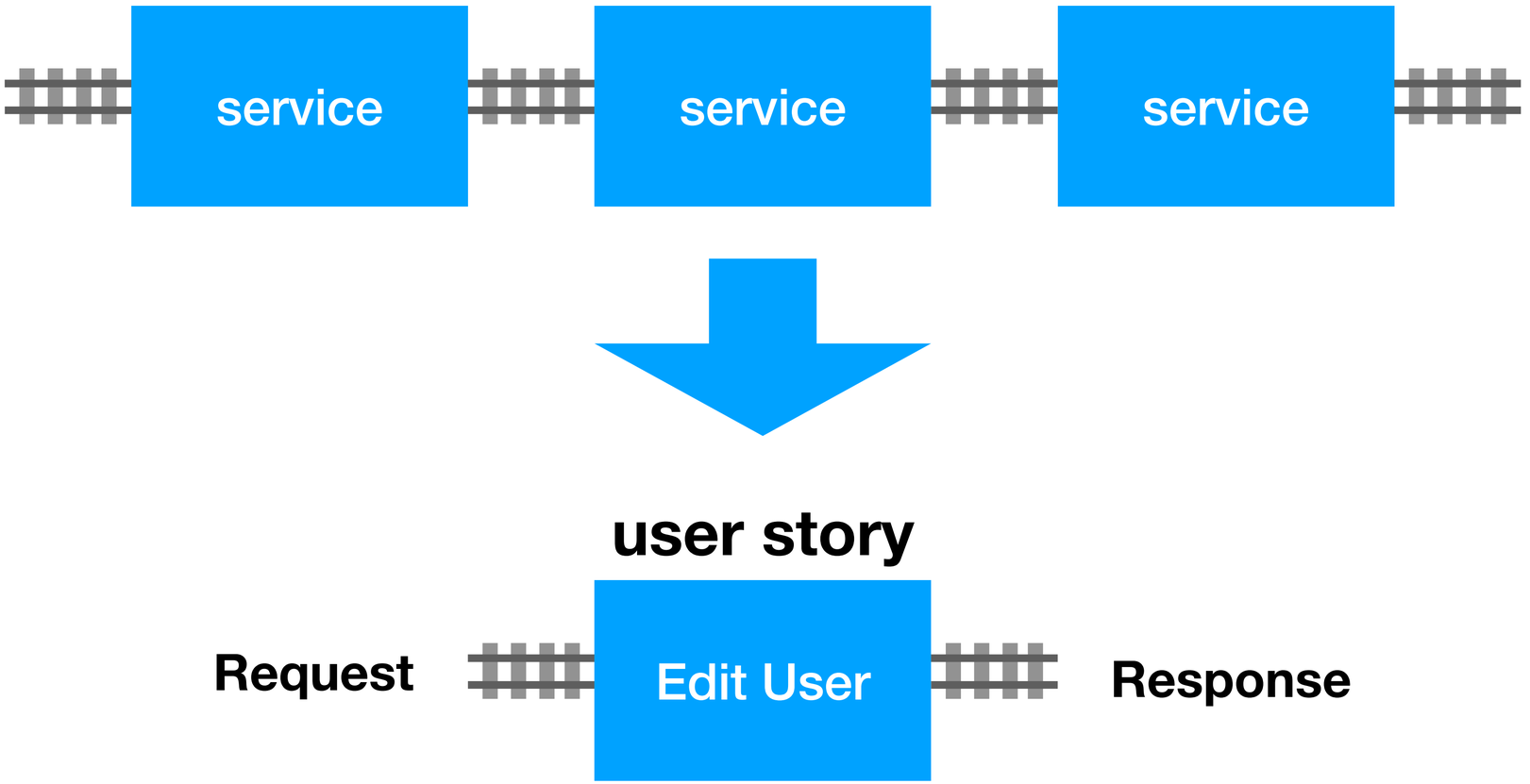
Banana => Cherie

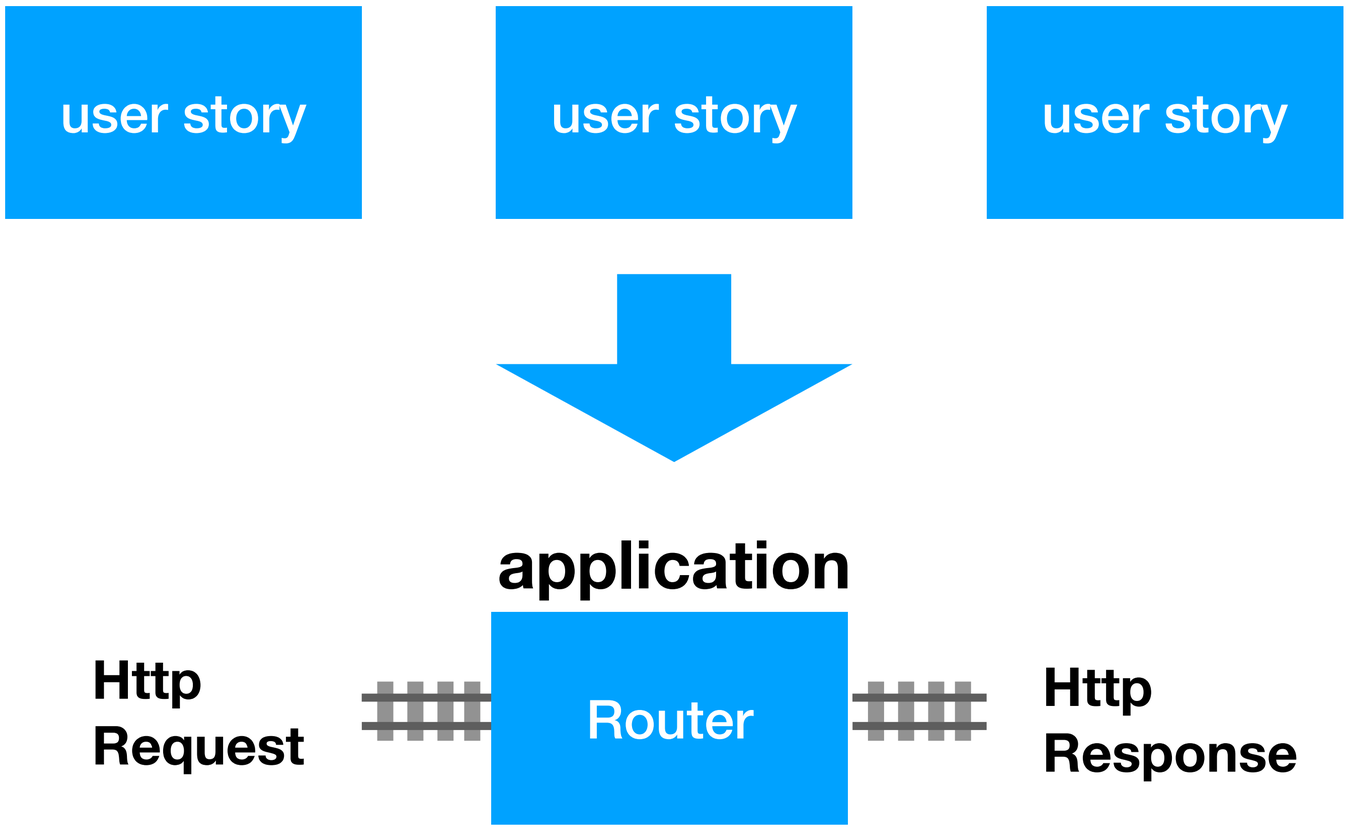
Function Composition

Apple => Cherie

Function Composition




service
service
service
user story
Request
Edit User
Response


user story
user story
user story
application
Http
Request
Router
Http
Response

Functions as Values

```
val add = (a: Int, b: Int) => a + b

add(3, 4)
```

Functions are Things and can be stored in a variable, returned as a value and passed as a parameter.

Function as Parameter

```
val add = (a: Int, b: Int) => a + b


def calculate (f: (Int, Int) => Int, x: Int, y: Int) =
  f(x, y)


calculate(add, 3, 4)
```

Function as Return Value

```
val add = (a: Int, b: Int) => a + b


def calculation(): (Int, Int) => Int = {
  return add
}

calculation()(3, 4)
```

Partial Applied Function

```
def sum = (a: Int, b: Int) => a + b
sum(3,4)

val a = sum(_, _)
a(3,4)

val b = sum(3, _)
b(4)
```

When you invoke a function, passing in the arguments, you *apply* that function *to* the arguments

With partial applied function you invoke the function with some or none of the arguments

Currying

```
def sum(a: Int)(b: Int) => a + b
sum(3)(4)


val b = sum(3)
b(4)

val add = (a: Int) => (b: Int) => a + b
add(3)(4)
```

When you invoke a function, passing in the arguments, you *apply* that function *to* the arguments

With partial applied function you invoke the function with some or none of the arguments

Strategy OO

```
trait Strategy {
  def apply(a: Int, b: Int): Int
}

class MyClass(strategy: Strategy) {
  def calculate(a: Int, b: Int): Int = {
    strategy.apply(a, b)
  }
}

val myClass = MyClass((a: Int, b: Int) => a + b)
val result = myClass.calculate(3, 4)
```

Strategy FP

```
def calculation(strategy: (Int, Int) => Int)(a: Int, b: Int) =
  strategy.apply(a,b)


val result = calculation((a, b) => a + b)(3, 4)
println(result)
```

Decorator OO

```
trait Decorator:
  def decorate(): String

class Start(str: String) extends Decorator:
  override def decorate() = str

class Capitalize(str: Decorator) extends Decorator:
  override def decorate() = str.decorate().capitalize

class Exclamation(str: Decorator) extends Decorator:
  override def decorate() = str.decorate() + "!"


val result = Exclamation(Capitalize(Start("hello"))).decorate()
```

Decorator FP

```
def start(str: String) = str

def capitalize(str: String) = str.capitalize

def exclamation(str: String) = str + "!"
```

```
val all = start andThen capitalize andThen exclamation
val result = all("hello")
```

DI OO

```
class Service:
  val users = List("John", "Rita", "Ali")

  def findById(id: Int) =
    users(id)


class Controller(service: Service):
  def findById(id: Int) =
    service.findById(id)


val controller = Controller(Service())
val user = controller.findById(2)
```

DI FP

```scala
def findByIdService(id: Int) =
  val users = List("John", "Rita", "Ali")
  users(id)


def findById(f: Int => String)(id: Int) =
  f(id)


// partial applied
val findWithId = findById(findByIdService)

val user2 = findWithId(2)
```

DI FP Givens

```
def findByIdService(id: Int) =
  val users = List("John", "Rita", "Ali")
  users(id)


type Service = Int => String

def findById(using f: Service)(id: Int) =
  f(id)

given Service = findByIdService

val user = findById(2)
```