W tym dokumencie chciałbym wyjaśnić i opisać różne decyzje które podjąłem, a które zostały powzięte w celach często bardziej demonstracji wiedzy na temat scali niż powodów praktycznych. W projekcie używam wszystkich podstawowych charakterystycznych dla scali typów abstrakcji (trait, abstract type, self type), a także projektuje i implementuje strukturę do zarządzania rezerwacjami wykorzystując standardowe (choć uproszczone względem implementacji w bibliotekach) techniki łączenia paradygmatu funkcyjnego zewnętrznego interfejsu z imperatywnym stylem w rozumieniu mutowalnego stanu w celu osiągnięcia sprawności implementacji. (Celowo pozwalam sobie na wykorzystanie miejscami pętli while). A także innych narzędzi w scali które to umożliwiają jak ograniczanie dostępności pola klasy/traitu do innych metod pakietu. Zadanie jest w scali 2 z użyciem play.

Zacznijmy od pliku app/models/Types.scala. Jest tu dużo aliasów dla czytelności. Celowo ze względu na rozmiar zdecydowałem się na najprostsze a jednocześnie zachowujące maksymalną sprawność typów prymitywnych rozwiązanie. W większym projekcie wykorzystał bym zaletę statycznego typowania scali i stworzyłbym dla przykładowego prostego typu String lub innego zwłaszcza dla prymitywnego liczbowego jak Int osobne bardzo proste klasy je opakowujące dające w ten sposób gwarancję że np w żadnym miejscu gdzie używam funkcji biorącej dwa typy o innym znaczeniu ale tym samym typie podstawowym nie zostaną przez przypadek zamienione miejscami. W scali 2 możny dla typów prymitywnych dziedziczyć po AnyVal i przy zachowaniu pewnych ograniczeń mieć nadzieje na sprawniejsze zachowanie względem typu referencyjnego. Natomiast nawet AnyVal czasami jest “boxed” przez JVM, w pewnych przypadkach nawet kilkakrotnie. Rozwiązanie oferuje tutaj scala 3 która pozwala na tworzenie takich prostych typów dając gwarancję że w runtime zastąpiony zostanie on typem prymitywnym. Natomiast moje proste rozwiązanie pozwala na zdefiniowanie czytelnych extension method dla rozpakowywania tupli.

Przejdźmy teraz do app/models/Row.scala. Jest tu implementacja specjalnej struktury. Listy która oznacza rząd w kinie i pozwala na łatwe nim zarządzanie. Ma ona być funkcyjna dla docelowego użytkownika, a jednocześnie możliwie sprawnie implementować swoje metody, zwłaszcza budowanie nowej listy. Jest ona w dużym stopniu przepisaniem wielu elementów z artykułu Independently Extensible Solutions to the Expression Problem. Możliwości tworzenia rozszerzalnego i wielokrotnego użytku oprogramowania w scali są o rząd wielkości większe niż w Javie gdzie często oznaczają sparametryzowaną skomplikowaną abstrakcyjną i mało czytelną klasę. Tutaj tworzy się strukturę bez wiedzy o możliwych przyszłych rozszerzeniach oraz bez modyfikacji istniejącego już kodu. Jednym z podstawowych problemów którego już bardziej nie adresowałem przy braku kowariantności jest brak możliwości zrobienia jednego elementu który byłby odpowiednikiem Nil dla zwykłych list w scali. Nie możemy już wykorzystać typu Nothing i mieć własnego Nila który byłby podtypem wszystkich list i musimy ich stworzyć co najmniej jeden dla danego typu.

Fundamentem jest oczywiście BaseList

```
trait BaseList {
  type list[T] <: ListBase[T]

  trait ListBase[T] {
    def +:(h: T): list[T]
    def ++:(r: list[T]): list[T]
    def isNil: Boolean
    def get: list[T]

    private[row] var next: Option[list[T]] = None
    final def tail: list[T] = next.get
  }

  def elem[E](h: E, t: list[E]): list[E]
  def nil[E](): list[E]

  trait ElemBase[T] extends ListBase[T] {
    val head: T
    final def isNil = false
  }

  trait NilBase[T] extends ListBase[T] { def isNil = true }
}
```

Nie będę tutaj tłumaczył na czym polega ten wzorzec natomiast opowiem o pewnym usprawnieniu. Zacznijmy od tego że niestety struktura nie jest kowariantna nad czym bardzo ubolewam ale jest to następstwo posiadania pola var i wynikających z tego tytułu standardowych problemów z typami gdyby struktura była kowariantna.

Zwróćmy uwagę na pole private[row] var next: Option[list[T]]. Będzie ono dostępne nawet dla funkcji i metod z innych klas pod warunkiem że są w tym samym pakiecie. Umożliwi to nam sprawne konstruowanie nowych rzędów poprzez dodanie na koniec w czasie stałym i stworzenie uproszczonego odpowiednika ListBuffer do budowanie naszej struktury.

Nasz RowBuffer jest krótki i jest w istocie uproszczonym odpowiednikiem ListBuffer z bardzo podobną implementacją. Dopóki nasza lista nie została zwrócona pozwalamy na

dodawanie elementów na końcu w czasie stałym. Możliwość użycia konstrukcji imperatywnych moim zdaniem daje nam przewagę nad najmocniejszym językiem funkcyjnym jakim jest haskell, gdzie od kolegi wiem że można mieć dodawanie w listach w czasie stałym na końcu jeśli użyjemy list różnicowych. Właśnie ta konieczność sięgania po wysokie i często trudne abstrakcję w haskellu by zrobić coś prostego jest jednym z powodów że pozostał językiem bardziej akademickim, a przecież sprawne budowanie list to tylko jeden przykład.

```
class RowBuffer[T] {
  import Row._
  private var first: list[T] = nil[T]()
  private var last0: list[T] = null
  private var aliased: Boolean = false

  def addOne(element: T): this.type = {
    if (aliased) throw new UnsupportedOperationException("buffer already turned to list")
    val last1 = elem[T](element, nil[T]())
    if (first.isNil) first = last1 else last0.next = Some(last1)
    last0 = last1
    this
  }

  def toRow: list[T] = {
    aliased = first.isNil
    first
  }
}
```

Możemy teraz wykorzystać RowBuffer do konkatenacji list:

```
def ++:(r: list[T]): list[T] = {
  val l1 = toList
  val l2 = r.toList
  val buff = new RowBuffer[T]()
  l1.foreach(buff.addOne(_))
  l2.foreach(buff.addOne(_))
  buff.toRow
}
```

Wracając do naszej struktury to w rozwiązaniu expression problem możliwe jest stworzenie metod map, flatMap i withFilter co pozwala na wykorzystanie wyrażenia jakim jest for w scali, a który w całości jest do tych metod tłumaczony w czasie kompilacji. Do ich sprawnej implementacji używam pętli while. Osobiście spotkałem się w kodzie optymalizatora zapytań neo4j z użyciem w kodzie scali while z komentarzem aby tego nie zmieniać ponieważ przeszedł przez profiler.

```
val l = 1 +: 2 +: 3 +: 4 +: 5 +: nil[Int]()

val l2 = for { i ← l
               if i ≠ 5
             } yield (i*2)
```

Niestety jako że mam in-memory bazę to dobrym pomysłem okazały się tablice więc klasy Cinema i ScreenRoom nie są w pełni funkcyjne natomiast scala pozwala na zastosowanie uniform access principle w przypadku funkcyjnych metod bezparametrowych ukrywając szczegóły implementacji danego pola. Klasy app/model/Cinema.scala i app/model/ScreenRoom.scala są już mniej pasjonujące. Zasadniczo wykorzystują wiele podstawowych metod na listach. Warto jeszcze wspomnieć o app/controller/TicketBookingController.scala. Java wykorzystuje unicode więc Polskie znaki nie są problemem. Jedynie sprawdzam czy String oznaczający imię i nazwisko matchuje się z regexem:

```
val name, surname = "([a-zA-ZąĄęĘóÓćĆńŃłŁśŚźŹżŻ]{3,20})"
val optionalSurname = "(-[a-zA-ZąĄęĘóÓćĆńŃłŁśŚźŹżŻ]{3,20}){0,1}"
val nameSurnameRegex = (name + " " + surname + optionalSurname).r

def isNameSurnameValid(nameSurname: String): Boolean = nameSurnameRegex.pattern.matcher(nameSurname).matches
```

Ostatnią fajną mini techniką w scali która jest podpatrzona z leniwego haskella jest wykorzystanie lazy var aby swobodnie zdefiniować rożne stałe nawet jeśli część z nich nie mogłaby się w danym przypadku obliczyć. Jeszcze w ScreenRoom.scala metoda (chyba metody jako jedyne nie są obiektami w scali) filtrująca pokazy mogłaby zostać rozwinięta do bardziej uniwersalnej która bierze funkcję filtrującą. Umiejętne użycie abstrakcji funkcyjnych jak funkcje częściowe, czy funkcje wyższego rzędu może znacznie uprościć i skrócić kod z czym w javie jest problem. Jeśli pisze w pythonie to też używam tych abstrakcji. Używam uproszczonej reprezentacji czasu jako Inta ze względu że używanie dat jest proste ale może niepotrzebnie miejscami żmudne. Na koniec zwracam tylko cenę bez terminu opłacenia jeśli dobrze rozumiem ten use case bo jest to łatwe i skupiam się na innych istotniejszych rzeczach i pomyślałem że już wystarczy.

GET /getScreenings/:day/:movieStart/:movieEnd
controllers.TicketBookingController.getScreenings(day: Int, movieStart: Int, movieEnd: Int)

GET /chooseScreening/:userId/:roomId/:screeningID
controllers.TicketBookingController.chooseScreening(userId: Int, roomId: Int, screeningID: Int)

PUT /makeReservation/:userId
controllers.TicketBookingController.makeReservation(userId: Int)

API jest raczej proste. Użytkownik po zdobyciu listy filmów dostaje również id. Dałem elementarną synchronizację więc teoretycznie wielu użytkowników może składać rezerwację. Również finalnie byłby jakiś frontend i użytkownik nie miałby dostępu do pola id.

Załączam demo.sh z use casem i kilkoma błędnymi zapytaniami.