App From Scratch Documentation

Mikhail Kuzmin

июл. 15, 2020

Оглавление

Книга о Clean Architecture и Clojure. Автор - Михаил Кузьмин m.kuzmin+afs@darkleaf.ru.

Это некоммерческая книга. Если вы заметили опечатку, пожалуйста, пришлите Pull Request. Если у вас возникли вопросы или недопонимание - напишите мне.

- epub
- pdf
- telegram: @app_from_scratch

Поддержать:

- благодарность за книгу (100)
- большая благодарность (500)

Глава 1

Intro

1.1 Кому?

За плечами несколько крупных проектов на web фреймворках. Столкнулся задачами, где фреймворк начинает мешать. Хочется посмотреть на другие подходы в разработке, получить направление для развития.

Увяз в техническом долге, сложности предметной области. Команда стала медленно поставлять ценность. Тесты выполняются десятки минут. ORM генерирует крайне неэффективные запросы.

Заказчики постоянно изменяют требования, и просят быстрее поставлять ценность.

Хочется получать удовлетворение от работы.

1.2 Что?

Это книга с теорией и практикой. Мы создадим приложение с нуля и без фреймворков, рассмотрим каждый аспект приложения: как моделировать предметную область, как описывать сценарии, как тестировать, как доставить приложение клиенту с помощью web, как работать с базами данных.

Мы познакомимся с языком Clojure(LISP для JVM), принципами SOLID, Clean Architecture, паттернами Data Mapper, Unit of Work, Identity Map. Научимся откладывать принятие технических решений.

Хорошую характеристику языку дал Никита Прокопов в интервью:

> Clojure — это язык для старых, уставших программистов. К нему не приходят в начале карьеры, а он становится интересен уже после 10-15 лет. Во-первых, у тебя абсолютная свобода. Всё, что тебе нужно, ты делаешь сам или берешь из библиотек. В самом языке зашито минимум вещей. Всё остальное ты конструируешь, делаешь ровно так, как тебе нужно. Всё самодельное и можно поменять. Это опасно, когда ты новичок, но полезно, если ты уже эксперт и знаешь то, что тебе нужно.

Глава 2

Clojure

2.1 Введение

Что такое Clojure? Это современный диалект семейства языков LISP. Паразитирует на платформе JVM. Есть реализации для JavaScript и .Net. Язык динамический, функциональный, но не зациклен на чистоте(purity), т.е. допускает побочные эффекты. Т.к. clojure работает на JVM, то нет недостатка в библиотеках.

Почему именно clojure выбран в качестве языка для примеров? Философия языка - доминирование простого(simple) над легким(easy). Simple - простое, работающее, как ожидается. Это объективная характеристика. Она противоположна complex - состоящему из множества частей. Easy - привычное, обыденное. Это субъективная характеристика. Она противоположна difficult - непривычному, требующему усилий для понимания. Подробнее в Simple Made Easy.

Какая цель этой главы? Поверхностное знакомство с языком, его возможностями, которые понадобятся нам в процессе проектирования приложения.

2.2 Базовый синтаксис

Вы можете воспользоваться сервисами https://repl.it и http://clojurescript.io, чтобы поэкспериментировать с языком прямо в браузере без установки окружения на свой компьютер.

Цель этих примеров - научиться читать, но не писать код на clojure.

```
;; комментарий
```

Печать на экран. `prn` - функция печати на экран. `"hello world"` - строка, аргумент функции.

```
(prn "hello world")
```

В lisp используется только префиксная (польская) нотация, т.е. функция стоит всегда на первом месте, а за ней ее аргументы. Вместо функции может быть специальная форма или макрос, но не будем касаться этой темы. Иными словами на первом месте внутри скобок находится нечто, что будет вызвано.

Для примера сравним js и lisp:

```
x.method(y, z) -> (method x y z)
func(x, y, z) -> (func x y z)
x + y -> (+ x y)
```

В Clojure есть поддержка неймспейсов. Мы не будем разбирать их объявление, а сразу перейдем к вызову:

```
(some-ns/some-fn some-arg)
```

Clojure - язык не ленивый, и аргументы функции вычисляются до ее вызова. Рассмотрим на примере:

```
(prn (str "hello" " " "world"))
```

Сначала будет вычислен аргумент функции `prn` - `(str "hello" " " "world")`. `str` - функция конкатенации строк. Таким образом `prn` будет вызван так: `(prn "hello world")`.

Для демонстрации языка воспользуемся утверждениями - assert. Если аргумент ложный, то будет брошено исключение, если истинный, то просто вернется nil.

```
(assert true) ;; #=> nil
;; (assert false) ;; AssertionError Assert failed: false
;; (assert nil)   ;; AssertionError Assert failed: nil
```

Например, я утверждаю, что 1 = 1:

```
(assert (= 1 1))
```

Привычные операторы могут принимать переменное количество аргументов:

```
(assert (= 1)) ;; всегда истинно для одного аргумента
(assert (= 1 1 1)) ;; 1 = 1 = 1

(assert (< 1)) ;; всегда истинно для одного аргумента
(assert (< 1 2 3)) ;; 1 < 2 < 3
```

Примитивные clojure типы - java типы. Проверим это с помощью функции `class`, возвращающей класс своего аргумента

```
(assert (= java.lang.String  (class "some string")))
(assert (= java.lang.Long    (class 1)))
(assert (= java.lang.Boolean (class true)))
```

Мы можем задать название некоторому значению с помощью специальной формы `let`:

```
(let [x 1]
  (assert (= 1 x)))
```

Отмечу, что `x` локальное обозначение, и вне `let` оно не существует.

Новое значение можно связать с тем же названием:

```
(let [x 1
      x 2]
  (assert (= 2 x)))
```

Или переопределить внешнее:

```
(let [+ -]
  (assert (= 0 (+ 1 1))))
```

В рамках текущего пространства имен можно объявить глобальное значение:

```
(def x 1)
(assert (= 1 x))
```

Определим анонимную функцию одного аргумента `x` и прибавляющую к нему единицу:

```
(let [f (fn [x] (+ 1 x))]
  (assert (= 3 (f 2))))
```

Если мы хотим объявить функцию в неймспейсе, то вместо `(def f (fn [x] ...))` удобно воспользоваться макросом `defn`:

```
(defn f [x] (+ 1 x))
(assert (= 3 (f 2)))
```

Clojure поддерживает замыкания:

```
(let [x 1
      f (fn [y] (+ x y))]
  (assert (= 3 (f 2))))
```

Функции это значения:

```
;; `+` - функция, а не оператор
(assert (= 6 (reduce + [0 1 2 3])))
```

Для коротких функций есть краткая форма:

```
(let [x [0 1 2]
      x'  (map #(+ 2 %) x)
      x'' (map (fn [i] (+ 2 i)) x)]
  (assert (= x' x'')))
```

В отличие от императивных языков, в clojure нет присваивания, т.е. `let` создает не переменные, а только именует значения. Если бы это было присваивание, то функция `f` вернула бы 2:

```
(let [x 1
      f (fn [] x) ;; замыкание(closure)
      x 2]
  (assert (= 2 x))
  (assert (= 1 (f))))
```

Имена могут содержать некоторые спецсимволы и их комбинации:

```
(let [x 1
      x' x
      x? x
      ?x x
      x! x
      !x x
      +x+ x
      -x- x
      *x* x
```

(continues on next page)

(продолжение с предыдущей страницы)

```
      =x= x]
  (assert (= x x' x? ?x x! !x +x+ -x- *x* =x=)))
```

Массивы в Clojure называются векторами. Вектора поддерживают доступ по индексу и хранят произвольные типы:

```
(let [x [0 "str" true [1 2 3]]]
  (assert (= "str" (get x 1)))
  (assert (= 0 (first x)))
  (assert (= "str" (second x))))
```

Ассоциативные массивы или мапы хранят ключи и значения произвольных типов. Запятая в clojure - просто пробельный символ, который можно опускать, однако при записи хеша для удобства запятые используют для разделения пар ключ-значение:

```
(let [x {:key 1, "key" 2, 2 3, true "4", [1 2] "5", nil 6}]
  (assert (= 1 (get x :key)))
  (assert (= 6 (get x nil))))
```

`:key` - это специальный тип Keyword. Их частно используют как ключи в ассоциативных массивах. Аналог символов в ruby или атомов в erlang. Реализуют интерфейс функций, т.е. принимает map и возвращает ассоциированное с собой значение:

```
(let [x {:key "value"}]
  (assert (= "value" (:key x))))
```

Все clojure значения - неизменяемые. При изменении возвращается новый объект, старый остается доступен. При этом не происходит полного копирования, т.е. новая структура переиспользует старую.

```
;; pop - возвращает коллекцию без вершины
(let [x  [0 1 2 3]
      x' (pop x)]
  (assert (= [0 1 2 3] x))
  (assert (= [0 1 2] x')))

;; assoc - добавляет значения по их ключам
(let [x {}
      x' (assoc x :k1 1 :k2 2)]
  (assert (= {} x))
  (assert (= {:k1 1, :k2 2} x')))
```

Этот код можно выполнить прямо в браузере. Там доступна консоль, в которой можно выполнять произвольные clojure выражения.

2.3 Зачем такой синтаксис?

Принципиальное отличие LISP от других языков - запись кода в виде данных(списков). В прочих языках, вроде javascript, код записывается текстом. Это наглядно видно при использовании `eval`:

```
// javascript
const code = "1 + 2"
eval(code) //=> 3
```

```
;; clojure
(let [code (quote (+ 1 2))]
  ;; далее будет показано, что содержится в code
  (eval code)) ;;=> 3
```

`quote` - останавливает выполнение, и преобразует код в данные. Без `quote` произошло бы выполнение выражения `(+ 1 2)` и `code` был бы связан со значением 3:

```
(let [code (quote (+ 1 2))]
  (assert (not= 3 code)))
```

Существует краткая форма для записи `quote` - `'`. Далее я буду использовать именно на этот вариант:

```
(assert (= (quote (+ 1 2))
           '(+ 1 2)))
```

`'(+ 1 2)` представляет список из 3-х элементов: символа `+` и 2-х чисел:

```
(let [code '(+ 1 2)
      operator (first code)
      operand  (second code)]
  (assert (= clojure.lang.PersistentList (class code)))
  (assert (= clojure.lang.Symbol (class operator)))
  (assert (= java.lang.Long (class operand))))
```

Символ - специальный тип данных, символизирующий, например, функцию, макрос, значение. Символ `some-name` можно создать следующими способами: `'some-name`, `(symbol "some-name")`. Если рассматривать следующий код как данные, то символами будут: `let`, `x`, `y` и `+`.

```
(let [x 1
      y 2]
  (+ x y))
```

Важно отметить, что `'` останавливает вычисление всех выражений, в том числе вложенных. Если вам нужно просто создать список и вычислить некоторые его элементы, то следует использовать функцию `list`:

```
(let [x 1
      y 2
      code-1 (list '+ x y)
      code-2 '(+ x y)]
    (assert (not= code-1 code-2))
    (assert (= (list '+ 1 2) code-1))
    (assert (= (list '+ 'x 'y) code-2)))
```

2.3.1 Макросы

То обстоятельство, что код записан в виде структур данных, дает нам возможность модифицировать код с помощью этого же языка.

Например, мы можем написать функцию, которая дает возможность записывать арифметические выражения привычным способом.

```
;; (1 + 2) => (+ 1 2)
(defn infix-fn [infixed]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  (let [operand-1 (first infixed)
        operator  (second infixed)
        operand-2 (last infixed)]
      (list operator operand-1 operand-2)))

(let [infixed '(1 + 2)
      code    (infix-fn infixed)
      result  (eval code)]
  (assert (= 3 result)))
```

Но практической ценности от такой функции мало. Каждый раз нужно вызывать `eval`.

В LISP есть функции, выполняющиеся на этапе компиляции. Они используются для модификации кода. Т.е. принимают и возвращают код как данные. Это макросы:

```
;; просто вызываем нашу функцию
;; вместо вызова можно скопировать сюда ее тело
(defmacro infix [infixed]
  (infix-fn infixed))

(assert (= 3 (infix (1 + 2))))
;; после компиляции эта строчка станет такой:
;; (assert (= 3 (+ 1 2)))

;; macroexpand - это аналог eval для макросов
;; т.е. он разворачивает макрос в runtime
(let [code '(infix (1 + 2))
      compiled (macroexpand code)]
    (assert (= '(+ 1 2) compiled)))
```

Т.е. макрос выполняется на этапе компиляции и принимает в качестве аргументов куски кода как данные. Макрос возвращает структуру данных, которая затем будет исполнена при запуске программы.

2.3.2 Разделение ответственности

При таком подходе четко выделяются 2 ответственности:

- то, что код делает
- то, как код выглядит

Например, нам нужно сделать инкремент для каждого элемента, взять нечетные элементы и посчитать сумму элементов. Выглядит это устрашающе:

```
(reduce + (filter odd? (map inc [0 1 2 3])))
```

Можно заметить, что все эти функции принимают коллекцию последним аргументом. Чтобы код проще читался, воспользуемся макросом `->>`:

```
(->> [0 1 2 3]
     (map inc)
     (filter odd?)
     (reduce +))

(macroexpand '(->> [0 1 2 3]
                   (map inc)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                    (filter odd?)
                    (reduce +)))
;;=> (reduce + (filter odd? (map inc [0 1 2 3])))
```

Этот макрос берет свой первый аргумент, подставляет его на последнее место в свой второй аргумент, результат вычисления, подставляет в третий, и так далее.

Важно понять, что этот макрос работает с любыми функциями. Например, в javascript нет макросов и для соединения функций в цепочки каждая библиотека реализует свой способ:

```
// https://lodash.com/docs/4.17.5#chain

var users = [
  { 'user': 'barney',  'age': 36 },
  { 'user': 'fred',    'age': 40 },
  { 'user': 'pebbles', 'age': 1 }
];

var youngest = _
  .chain(users)
  .sortBy('age')
  .map(function(o) {
    return o.user + ' is ' + o.age;
  })
  .head()
  .value();
```

2.3.3 Преимущества

При таком подходе ядро языка реализует минимальный набор специальных форм вроде:

- `(def symbol doc-string? init?)`
- `(if test then else?)`
- `(let [ binding* ] expr*)`
- `(quote form)`
- `(fn name? [params* ] expr*)`

На самом деле их чуть больше, подробнее о них можно прочитать в разделе special_forms.

Все остальное - это функции, макросы или обертки для вызова java кода.

```
(macroexpand '(if-not false 1 2))
;; #=> (if (clojure.core/not false) 1 2)

;; а вот так определяется стандартная функция not
(defn not
  "Returns true if x is logical false, false otherwise."
  {:tag Boolean
   :added "1.0"
   :static true}
  [x] (if x false true))

;; у класса clojure.lang.Util вызывается статический метод identical
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(defn identical?
  "Tests if 2 arguments are the same object"
  {:added "1.0"}
  ([x y] (clojure.lang.Util/identical x y)))
```

2.3.4 Parinfer

Расставлять и выравнивать скобки - неблагодарное занятие. Но есть плагин для множества редакторов, облегчающий редактирование lisp выражений. Это parinfer.

2.3.5 Заключение

LISP языки называют программируемыми языками программирования. Вы можете самостоятельно расширять язык новыми синтаксическими конструкциями. При этом ядро языка остается маленьким и простым.

2.4 Управление состоянием

Традиционно функциональными считают те языки, в которых реализованы функции первого класса. Но под это определение попадают так называемые мультипарадигменные языки вроде javascript и ruby. В нашем случае подойдет более строгое определение. Я буду называть функциональными те языки, в которых отсутствует присваивание. Однако, тот же Haskell допускает присваивание, но под особым контролем. Аналогично и Clojure имеет особый механизм для этого.

Например, в javascript есть присваивание и переменные:

```
let x = 1
let f = () => x
x = 2
f() // => 2
```

В clojure это не так, т.к. используется связывание вместо присваивания:

```
(let [x 1
      f (fn [] x)
      x 2]
  (f)) ;; => 1
```

Философия clojure - большая часть программы должна следовать функциональной парадигме. Для оставшихся частей есть специальный механизм для работы с изменяющимся состоянием.

2.4.1 Atom

```
(let [x (atom 1)
      f (fn [] @x)
      _ (reset! x 2)]
  (f)) ;; => 2
```

Атом можно рассматривать как контейнер, способный заменять свое значение и контролирующий доступ из нескольких потоков.

Атом имеет следующий интерфейс:

- `(atom 1)` - новый атом с начальным значением 1
- `@x` - доступ к значению атома `x`. В один момент времени все потоки прочитают одно и то же значение.
- `(reset! x 2)` - установка значения атома `x` в 2
- `(swap! x inc)` - замена значения с помощью функции `inc`. Т.е. `inc` принимает текущее состояние атома и возвращает будущее.

`swap!` может принимать неограниченное кол-во аргументов, это используется, чтобы уменьшить вложенность:

```
(let [x (atom 0)]
  (swap! x (fn [old-state] (+ old-state 10))))

(let [x (atom 0)]
  (swap! x + 10))
```

Т.е. в функцию `+` первым аргументом подставится старое состояние, а после все дополнительные аргументы `swap!`.

`swap!` использует атомарную операцию *Сравнение с обменом (Compare And Swap)*. Это работает следующим образом:

- Запоминаем текущее состояние атома.
- Применяем переданную функцию к запомненному значению, получаем новое значение.
- Атомарно сравниваем запомненное значение с текущим и если оно не изменилось, устанавливаем новое значение. Если текущее значение изменилось в другом потоке, начинаем сначала.

Под атомарностью тут понимается, что сравнение с обменом - неделимая операция, т.е. другой поток не сможет заменить значение когда сравнение случилось, а обмен еще нет.

Если запустить 5 параллельных потоков(thread), то значение счетчика будет равно 5. Если бы атом не поддерживал *Compare And Swap (CAS)*, то значение счетчика было бы меньше 5, т.к. параллельные потоки затирали бы результаты друг друга.

```
(let [counter (atom 0)]
  (->> (repeatedly #(future (swap! counter inc)))
       (take 5)
       (doall)
       (map deref)
       (doall))
  @counter) ;; => 5
```

Функция `repeatedly` создает ленивую последовательность, где каждый элемент вычисляется с помощью вызова анонимной функции. `#(future ...)` - анонимная функция, создающая `future`. Макрос `future` выполняет свое содержимое в другом потоке и возвращает `future`. Функция `deref` блокирует текущий поток, дожидается исполнения `future` и возвращает результат. `map` и `repeatedly` возвращают ленивую коллекцию, поэтому мы используем `doall`, чтобы вычислить все ее элементы.

2.4.2 Состояние и идентичность

Рассмотрим привычный подход к моделированию. Допустим, наша программа обрабатывает данные людей, и нам важно знать имя, возраст, количество друзей и размер сбережений:

```
//javascript
class Person {
  constructor(name, age, friends, savings) {
    this.name = name
    this.age = age
    this.friends = friends
    this.savings = savings
  }
}

let alice = new Person("Alice", 22, 5, 100)
```

С течением времени Алиса менялась. Каждый день она находится в каком-то определенном состоянии.

Например, сегодня, в ее день рождения, она получила в подарок 300 монет.

```
alice.age++
alice.savings += 300
```

Или, в другой день, она не отдала долг другу и потеряла его:

```
alice.savings += 10
alice.friends -= 1
```

Важно отметить, что она увеличила накопления и потеряла друга одномоментно, но наша модель не соответствует этому. Модель становится противоречивой в момент, когда увеличились накопления, но еще не изменилось количество друзей.

Также, наша модель позволяет записать аттрибуты постороннего человека:

```
alice.name = "Bob"
```

При таком подходе происходит объединение понятий *Идентичность* и *Состояние*. Идентичность это нечто, что однозначно определяет моделируемую сущность. А состояние - это некие данные, описывающие сущность в заданный момент времени.

В нашем случае, идентичность - это сам объект, т.е. та область памяти, которую он занимает. Если бы объекты в javascript имели `object_id`, то он и представлял бы идентичность. Этим же объектом моделируется и состояние сущности.

В Clojure эти понятия разделяются. Состояние моделируется с помощью неизменяемых структур данных, а идентичность - с помощью ссылочных типов, вроде атомов.

```
(def alice (atom {:name "Alice"
                  :age 22
                  :friends 5
                  :savings 100}))

;; устанавливаем валидатор, запрещающий изменять имя
;; перед заменой значения будет запускаться валидатор,
;; если он вернет false, то состояние не будет заменено и будет брошено исключение
(set-validator! alice (fn [new-state]
                        (= (:name new-state)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                            (:name @alice))))

;; атомарно увеличиваем накопления и теряем друга
;; никто не сможет увидеть Алису в несогласованном состоянии
(swap! alice (fn [state]
                 (-> state
                     (update :savings + 100)
                     (update :friends - 1))))

(swap! alice assoc :name "Bob")
;; Boom!
;; IllegalStateException Invalid reference state  clojure.lang.ARef.validate (ARef.java:33)
```

Другим примером разделения идентичности и состояния является философское высказывание: «Нельзя войти в реку дважды». Вода течет, но мы продолжаем ассоциировать воду в разные моменты времени с рекой.

2.5 Оставшиеся ссылочные типы

Мы уже познакомились с атомами. Атом - контейнер, атомарно обновляющий свое содержимое. Вызов `swap!` блокирует текущий поток, т.е. это синхронная операция. При этом операция `swap!` нескоординированная, т.е. `swap!` влияет только на один объект.

Допустим, мы хотим перевести деньги с одного счета на другой, так, чтобы в любой момент времени в системе было постоянное количество денег.

```
(let [a (atom 1000)
      b (atom 0)]
  (future (swap! a - 100)
          (Thread/sleep 100)
          (swap! b + 100))
  (Thread/sleep 50)
  {:total (+ @a @b)
   :a     @a
   :b     @b}) ;;=> {:total 900, :a 900, :b 0}
```

Как видим, атомы не позволяют обеспечить постоянное количество денег в системе. Получилось так, что со счета списалось 100 монет, а на другой еще не записались.

Существуют и другие ссылочные типы, которые могут быть синхронными/асинхронными и скоординированными/нескоординированными.

2.5.1 Ref

С их помощью реализуется механизм Software Transaction Memory (STM).

```
(let [a (ref 1000)
      b (ref 0)]
  (future (dosync
           (alter a - 100)
           (Thread/sleep 100)
           (alter b + 100)))
  (Thread/sleep 50)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  {:total (+ @a @b)
   :a     @a
   :b     @b}) ;;=> {:total 1000, :a 1000, :b 0}

(let [a (ref 1000)
      b (ref 0)]
  (future (dosync
           (alter a - 100)
           (Thread/sleep 100)
           (alter b + 100)))
  (Thread/sleep 150)
  {:total (+ @a @b)
   :a     @a
   :b     @b}) ;;=> {:total 1000, :a 900, :b 100}
```

`dosync` - начинает транзакцию, `alter` - аналог `swap!`, только для `ref`, обязательно вызывается внутри `dosync`.

В примере с ref в любой момент времени состояние счетов непротиворечиво, и ссылки обновляются согласовано.

Если какой-то сторонний поток поменяет ссылку, участвующую в транзакции, то это транзакция перезапустится. Именно поэтому для обновления ссылок нужно использовать чистые функции.

Для изменения значения ссылки есть следующие функции:

- `alter`
- `ref-set`
- `commute`
- `ensure`

Т.е. это скоординированный ссылочный тип, с синхронными операциями.

2.5.2 Var

Мы используем их с начала знакомства с clojure.

Когда мы вызываем `(def x 1)` или `(defn f [arg] 1)`, то на самом деле создаем var(переменную).

Вместо того, чтобы постоянно писать `(@f arg)` для вызова функции, хранящейся в переменной `f`, ввели упрощенный синтаксис: `(f arg)`.

Чтобы получить саму переменную - нужно воспользоваться специальной формой `(var f)` или `#'f`. Наверняка, вы видели `#'user/f`, когда выполняли выражение `(defn f [])`. `user` - это пространство имен, в котором определена переменная.

Таким образом `(f arg)` эквивалентно `(@#'f arg)`.

Переменные позволяют переопределять свое значение для всех потоков:

```
(defn f [] 1)

(f) ;;=> 1

(alter-var-root #'f
                (fn [old-value]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                  (fn [] 2)))

(f) ;;=> 2
```

`alter-var-root` можно использовать для декорирования функций:

```
(defn f [] (prn :ok))
(alter-var-root #'f memoize)
(f) => вернет nil и напечатает :ok
(f) => просто вернет nil
```

`memoize` - стандартная функция, принимающая функцию, и возвращающая мемоизированный вариант.

Повторюсь, что `x` - получение значения переменной, а не самой переменной:

```
(def x 1)

(let [x' x] ;; запомнили содержимое x в x'
  (alter-var-root #'x inc)
  [x' x]) ;;=> [1 2]
```

Если мы используем функции, то значение извлекается на каждый вызов:

```
(def x 1)

(defn f [] x)

(f) ;;=> 1

(alter-var-root #'x inc)

(f) ;;=> 2
```

Если мы хотим ссылаться на саму переменную, то нужно поступить так:

```
(def x 1)

(let [x' #'x] ;; запомнили саму переменную x в x'
  (alter-var-root #'x inc)
  [@x' x]) ;;=> [2 2]
```

Т.е. переменные, как и прочие ссылочные типы, позволяют получить свое значение с помощью `@`.

Переменные реализуют интерфейс функций:

```
(defn f [] 1)

;; вызываем функцию
(f) ;;=> 1

;; получаем переменную, извлекаем ее значение и вызываем это значение
;; аналогично предыдущему
(@#'f) ;;=> 1

;; используем переменную в качестве функции
(#'f) ;;=> 1
```

Это бывает полезно, если вы передаете функцию как значение, например используете функцию высшего порядка, и хотите оставить возможность переопределять значение вашей функции. Это похоже на передачу по значению и по ссылке(указателю):

```
(defn f [] 1)

(defn inspect [f]
  (fn [& args]
    (prn args) ;; печатаем аргументы
    (let [result (apply f args)] ;; вызываем функцию с аргументами в виде коллекции
      (prn result) ;; печатаем результат
      result)))

(let [f1 (inspect f) ;; передача по значению
      f2 (inspect #'f)] ;; похоже на передачу по ссылке
  (alter-var-root #'f (fn [old]
                        (fn [] 2)))
  [(f1) (f2)]) ;;=> [1 2]
```

Переменную можно переопределить для определенной области и вернуть исходное значение:

```
(defn f [] 1)

(with-redefs [f (fn [] 2)]
  (f)) ;;=> 2
```

Нужно учитывать, что все потоки увидят это измененное изменение, что может привести к неожиданным результатам.

Этот способ не работает для inline функций, т.к. они не используют var, т.е. для большинства стандартных:

```
(with-redefs [+ -]
  (+ 2 1)) ;;=> 3
```

Все переменные по умолчанию статические. Бывают еще и динамические. Они позволяют переопределить свое значение только для текущего потока. По соглашению таким переменным надевают наушники: `*some-var*`.

```
(def ^:dynamic *x* 1)

(let [a (future
          (binding [*x* 2]
            (Thread/sleep 100)
            *x*))
      b (future
          (Thread/sleep 50)
          *x*)]
  [@a @b]) ;;=> [2 1]
```

Если бы binding переопределял значение для всех потоков, то `@b` вернул бы 2.

При этом, clojure функции, умеют запоминать контекст, а java tread - нет:

```
(def ^:dynamic *x* 1)

(binding [*x* 2]
  (future (prn "future" *x*))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  (.start (Thread. (fn [] (prn "thread" *x*))))

;; вывод на печать:
;; "future" 2
;; "thread" 1
```

Ленивые коллекции, future и т.п. умеют запоминать контекст треда начиная с версии clojure 1.3. Сторонние библиотеки, вроде core.async, также сохраняют контекст. Есть макрос bound-fn с помощью которого вы можете запомнить контекст, например, при работе с java interop.

В дальнейшем мы будем использовать динамические переменные для внедрения зависимостей. Внедрять зависимость можно и с помощью `alter-var-root`, но как быть, если вам нужен новый инстанс зависимости на каждый запрос, например сессия пользователя. Или вы захотите запуситиь несколько экземпляров приложения с разными зависимостями в одном JVM процессе.

Переменные могут быть и локальными, смотри `with-local-vars`.

Так же переменные можно сделать приватными, т.е. они станут доступны только в пределах своего неймспейса:

```
(def ^:private x 1)
(defn ^:private f [])
(defn- g [])
```

По нашей классификации это нескоординированный ссылочный тип, с синхронными операциями.

2.5.3 Agent

Агент - контейнер для значения с очередью операций над ним:

```
(let [a (agent 0)]
  (send a (fn [old]
            (Thread/sleep 50)
            (inc old)))
  @a) ;;=> 0

(let [a (agent 0)]
  (send a inc)
  (await a) ;; ждем, пока обработается очередь
  @a) ;;=> 1
```

Агент принимает сообщения в виде функций, выстраивает их в очередь и в отдельном потоке заменяет свое значение с помощью этих функций. Т.е. `send` - неблокирующая функция.

При этом агенты встроены в STM, т.е. сообщение будет отправлено только после успешного завершения транзакции:

```
(let [counter     (ref 0)
      calls-atom  (atom 0)
      calls-agent (agent 0)]
  (->> (repeatedly #(future
                      (dosync
                       (swap! calls-atom inc)
                       (send calls-agent inc)
                       (alter counter inc))))
       (take 100)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
      (doall)
      (map deref)
      (doall))
  (await calls-agent) ;; ждем, пока обработается очередь
  {:counter     @counter
   :calls-atom  @calls-atom
   :calls-agent @calls-agent}) ;;=> {:counter 100, :calls-atom 102, :calls-agent 100}
```

Атомы не интегрированы в STM, поэтому при повторении транзакции из-за конфликтов `calls-atom` показывает количество успешных и неуспешных транзакций. Но агент интегрирован в STM и получает сообщения только после успешного завершения транзакции.

Функция `send` использует системный тредпул, и если функция может долго выполняться, то используют `send-off`, который выполняет эту функцию вне системного тредпула.

Это асинхронный несогласованный ссылочный тип.

2.5.4 Валидаторы и наблюдатели

Все ссылочные типы позволяют установить валидатор и добавить наблюдателей:

- `set-validator!`
- `add-watch`

2.5.5 Volatile

Это неполноценный ссылочный тип, но зато очень быстрый. При этом он не имеет поддержки валидаторов и наблюдателей.

```
(let [v (volatile! 0)]
  (vswap! v inc)
  @v) ;;=> 1
```

Он реализован как тривиальный java класс, хранящий состояние в `volatile` переменной. JVM содержит оптимизации, и если один поток изменил переменную, другие потоки могут не увидеть это изменение. Для этого случая в java есть ключевое слово `volatile`, которое показывает, что значение переменной может быть изменено в другом потоке.

Естественно, он не гарантирует атомарности как атом:

```
(let [counter-a (atom 0)
      counter-v (volatile! 0)]
  (->> (repeatedly #(future
                      (swap! counter-a inc)
                      (vswap! counter-v inc)))
       (take 100)
       (doall)
       (map deref)
       (doall))
  {:counter-a  @counter-a
   :counter-v  @counter-v}) ;;=> {:counter-a 100, :counter-v 98}
```

2.6 Полиморфизм

Полиморфизм дает возможность писать **один** код для работы с **многими** типами. Полиморфизм можно грубо разделить на динамический и статический:

- Динамический полиморфизм — это про абстрактные классы, интерфейсы, утиную типизацию, т.е. только в рантайме будет понятно, с каким типом будет работать наш код.
- Статический полиморфизм — это в основном про шаблоны (generics). Когда уже на этапе компиляции из одного шаблонного кода генерируется код специфичный для каждого используемого типа.

Здесь и далее я буду понимать под полиморфизмом только динамический полиморфизм.

2.6.1 Мультиметоды

```
(defmulti foo identity)
;; identity - стандартная функция вида (fn [x] x)

(defmethod foo :a [x]
  [:a x])

(defmethod foo :default [x]
  [:default x])

(assert (= [:a :a] (foo :a)))
(assert (= [:default :b] (foo :b)))
```

`defmulti` - объявляет мультиметод `foo` с функцией диспетчеризации `identity`. Т.к. `identity` принимает один аргумент, то и наш метод будет также принимать один аргумент. Функция диспетчеризации на основе аргументов вычисляет значение диспетчеризации, по которому будет выбираться нужная реализация.

`defmethod` - объявляет реализацию для соответствующего значения диспетчеризации. В нашем случае объявляется метод для `:a` и метод по умолчанию, который будет обрабатывать оставшиеся случаи.

Рассмотрим пример посложнее. Будем моделировать игру в камень-ножницы-бумага:

```
(defmulti winner (fn [x y] (set [x y])))

(defmethod winner #{:rock}     [_ _] :drawn-game)
(defmethod winner #{:paper}    [_ _] :drawn-game)
(defmethod winner #{:scissors} [_ _] :drawn-game)

(defmethod winner #{:rock  :paper}    [_ _] :paper)
(defmethod winner #{:rock  :scissors} [_ _] :rock)
(defmethod winner #{:paper :scissors} [_ _] :scissors)

(assert (= :drawn-game (winner :rock :rock)))
(assert (= :paper (winner :rock :paper)))
(assert (= :paper (winner :paper :rock))) ;; симметричный случай
```

В clojure множества создаются функцией `set`, которая принимает коллекцию. Чтобы объявить множество пользуются конструкцией: `#{1 2 3}` - множество из 1, 2 и 3.

`[_ _]` запись означает, что функция принимает 2 аргумента, но мы их не будем использовать.

Ключевой момент - функция диспетчеризации `(fn [x y] (set [x y]))`. В данном случае значение диспетчеризации - множество, это обстоятельство позволяет не объявлять симметричные случаи, т.к. множество не поддерживает порядок.

Допустим, мы хотим добавить новый тип и получить игру камень-ножницы-бумага-пистолет. Для этого нам не нужно модифицировать предыдущий код. Для этого нам нужно просто объявить соответствующие методы для новых значений диспетчеризации:

```
(defmethod winner #{:gun} [_ _] :drawn-game)
(defmethod winner #{:gun :rock}     [_ _] :gun)
(defmethod winner #{:gun :paper}    [_ _] :gun)
(defmethod winner #{:gun :scissors} [_ _] :gun)
```

Причем таким образом можно расширять мультиметоды, объявленные в другом пространстве имен или даже в другой библиотеке.

Мультиметоды поддерживают иерархии, которые позволяют реализовывать наследование, в том числе множественное.

```
;; keyword могут иметь пространство имен
;; ::a - краткое объявление кейворда в текущем пространстве имен
;; текущее пространство - user
(assert ::a :user/a)

(defmulti foo identity)
(defmethod foo ::a [_] "implementation for ::a")

(defmulti bar identity)
(defmethod bar ::b [_] "implementation for ::b")

;; множественное наследование
;; x - производная a и b
(derive ::x ::a)
(derive ::x ::b)

(assert (= "implementation for ::a" (foo ::x)))
(assert (= "implementation for ::b" (bar ::x)))
```

Наследование работает и в случае, если значение диспетчеризации - вектор:

```
(defmulti foo (fn [x y] [x y]))
(defmethod foo [::a ::b] [_ _] true)

(derive ::x ::a)
(derive ::x ::b)

(assert (foo ::a ::b))
(assert (foo ::x ::x))
(assert (foo ::a ::x))
(assert (foo ::x ::b))
```

Т.к. мультиметоды используют функцию диспетчеризации и поддерживают значение по умолчанию, то это ООП, реализованное на принципах отправки сообщений (Alan Kay).

2.6.2 Протоколы

В большинстве случаев достаточно диспетчеризации по классу первого аргумента:

```
(defmulti foo (fn [this x y z] (class this)))
```

Для подобных случаев в clojure появились протоколы. Но прежде, нужно познакомиться с записями:

```
(defrecord User [id name])

(let [user (->User 1 "Alice")]
  (assert (= 1 (:id user)))
  (assert (= "Alice" (:name user)))
  (assert (= User (class user))))
```

Запись - это java класс, реализующий интерфейсы ассоциативных массивов(map). Атрибуты записи реализованы как соответствующие поля java класса. Кроме заранее указанных полей, запись может хранить произвольные:

```
(let [user (->User 1 "Alice")
      user (assoc user :additional "some value")]
  (assert (= "some value" (:additional user)))
  (assert (= User (class user))))
```

Запись - это надстройка над Типом. Тип - простой java класс не реализующий каких-либо интерфейсов. Как правило, пользуются Записями, а Типы используют, когда нужны «чистые» объекты.

```
(deftype T [attr])

(.-attr (->T 1)) ;;=> 1
```

Допустим, кроме записи `User`, у нас есть еще запись `Admin` и мы хотим проверять может ли кто-то создавать пользователей:

```
(defrecord User [id name])

(defrecord Admin [id name])

(defprotocol CreateUserAbility
  (can-create-user? [this]))

(extend User
  CreateUserAbility
  {:can-create-user? (fn [_] false)})

(extend Admin
  CreateUserAbility
  {:can-create-user? (fn [_] true)})

(let [user (->User 1 "Alice")]
  (assert (not (can-create-user? user))))

(let [admin (->Admin 1 "Bob")]
  (assert (can-create-user? admin)))
```

Протоколы могут содержать любое количество методов, как обычные java интерфейсы, но не поддерживают наследование. Протоколы могут расширять любую запись и любой java класс.

Кроме функции `extend` есть макросы `extend-type` и `extend-protocol`, которые делают запись более удобной:

```
(extend-type User
  CreateUserAbility
  (can-create-user? [_] false)
  OtherProtocol
  (some-method [_] :ok))

(extend-protocol CreateUserAbility
  User
  (can-create-user? [_] false)
  Admin
  (can-create-user? [_] true))
```

Если вы указываете реализацию протокола сразу при объявлении записи, то запись будет реализовывать java интерфейс, что повысит производительность. Эта же форма позволяет Записи реализовывать не протокол, а просто java интерфейс.

```
(defprotocol CreateUserAbility
  (can-create-user? [this]))

(defrecord User [id name]
  CreateUserAbility
  (can-create-user? [_] false))

(defrecord Admin [id name]
  CreateUserAbility
  (can-create-user? [_] true))

(let [user (->User 1 "Alice")]
  (assert (not (can-create-user? user))))

(let [admin (->Admin 1 "Bob")]
  (assert (can-create-user? admin)))
```

Стоит отметить, что если вы хотите добавить метод для работы с конкретной записью, то вам не нужен полиморфизм, и достаточно воспользоваться обычной функцией:

```
(defrecord User [id name])

(defn present [this]
  (str (:id this) " - " (:name this)))

(let [user (->User 1 "Alice")]
  (assert (= "1 - Alice" (present user))))
```

Записи и протоколы не поддерживают наследование, но при расширении типа протоколом можно воспользоваться функцией `extend` которая оперирует обычными ассоциативными массивами и функциями. Таким образом может быть реализовано наследование, примеси и т.п.:

```
(defrecord A [])
(defrecord B [])

(defprotocol Proto
  (method [this]))

(let [impl {:method (fn [_] :some-body)}]
  (extend A Proto impl)
  (extend B Proto impl))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(assert (= (method (->A))
           (method (->B))))
```

Кроме всего этого, есть возможность создать анонимную реализацию протокола или java интерфейса с помощью `reify`. Это удобно для тестирования или взаимодействия с java кодом. Для этого `reify` поддерживает замыкания:

```
(defprotocol Proto
  (method [this]))

(let [val      :val
      instance (reify Proto
                 (method [_] val))]
  (assert (= val (method instance))))
```

2.6.3 Функции и методы

Мультиметоды и протоколы позволяют расширять существующий «тип». Это позволяют делать и другие языки, например, в js можно добавить метод экземплярам:

```
String.prototype.foo = function() { return "foo" };
"any string".foo() //=> "foo"
```

Но что, если 2 библиотеки добавят метод с одним названием? Победит последний.

Благодаря префиксной записи вызов (мульти)метода не отличается от вызова функции:

```
(ns definitions)

(defn example-fn [x] :fn)

(defmulti example-multimethod identity)

(defprotocol P
  (example-method [this]))

;; ~~~~~~~~~~~~~~~~
(ns usage
  (:require [definitions]))

(deftype T [])

(extend-type T
  definitions/P
  (example-method [this] :method))

(defmethod definitions/example-multimethod :default [_] :multimethod)

(def instance (T.))
(definitions/example-fn instance)
(definitions/example-multimethod instance)
(definitions/example-method instance)
```

Таким образом расширения «типа» доступны через неймспейс и не конфликтуют между собой. Однако,

это не относится к реализации протокола напрямую через `deftype` или `defrecord`, т.к. методы java класса не имеют неймспейса.

2.6.4 Benchmark

Бенчмарк с помощью criterium. Исходники в виде проекта можно получить тут.

```
(ns bench.bench
  (:require
   [criterium.core :as criterium]
   [clojure.template :as template]))

(defprotocol Proto
  (proto-method [this]))

(deftype A []
  Proto
  (proto-method [_] :ok))

(deftype B [])

(extend-type B
  Proto
  (proto-method [_] :ok))

(def c (reify
         Proto
         (proto-method [_] :ok)))

(deftype D [])
(defmulti multi-method class)
(defmethod multi-method D [_] :ok)

(defn bench []
  (template/do-template [method obj-expr]
                        (do
                          (prn '(method obj-expr))
                          (let [obj obj-expr]
                            (criterium/quick-bench (method obj)))
                          (print "\n\n\n"))
                        proto-method (->A)
                        proto-method (->B)
                        proto-method c
                        multi-method (->D)))
```

```
(proto-method (->A))
Evaluation count : 132892038 in 6 samples of 22148673 calls.
             Execution time mean : 2.863123 ns
    Execution time std-deviation : 0.019320 ns
   Execution time lower quantile : 2.838807 ns ( 2.5%)
   Execution time upper quantile : 2.879423 ns (97.5%)
                   Overhead used : 1.666364 ns



(proto-method (->B))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Evaluation count : 97196952 in 6 samples of 16199492 calls.
             Execution time mean : 4.596519 ns
    Execution time std-deviation : 0.064386 ns
   Execution time lower quantile : 4.548777 ns ( 2.5%)
   Execution time upper quantile : 4.701984 ns (97.5%)
                   Overhead used : 1.666364 ns

Found 1 outliers in 6 samples (16.6667 %)
    low-severe   1 (16.6667 %)
 Variance from outliers : 13.8889 % Variance is moderately inflated by outliers



(proto-method c)
Evaluation count : 131449896 in 6 samples of 21908316 calls.
             Execution time mean : 2.857191 ns
    Execution time std-deviation : 0.018021 ns
   Execution time lower quantile : 2.843163 ns ( 2.5%)
   Execution time upper quantile : 2.885828 ns (97.5%)
                   Overhead used : 1.666364 ns



(multi-method (->D))
Evaluation count : 15134856 in 6 samples of 2522476 calls.
             Execution time mean : 38.633649 ns
    Execution time std-deviation : 0.717109 ns
   Execution time lower quantile : 37.971764 ns ( 2.5%)
   Execution time upper quantile : 39.594540 ns (97.5%)
                   Overhead used : 1.666364 ns
```

Как видно, протоколы на порядок быстрее мультиметодов. Реализация протокола в `deftype`, `defrecord` или `reify` в 2 раза быстрее `extend`.

2.6.5 Expression problem

Этот пункт необязателен, и для интересующихся я оставлю ссылку на соответствующую статью

2.7 Самостоятельное чтение

Книга не преследует цели научить clojure. Это нужно сделать самостоятельно. Вот некоторые материалы для этого:

- clojure.org
- расшифровки выступлений автора языка - Rich Hickey
- brave clojure - отличная книга о clojure
- clojurecourse.by - материалы русскоязычных курсов
- Clojure Koans

2.8 Окружение для разработки

Ранее мы использовали сервис repl.it и теперь настал момент для установки полноценного окружения.

2.8.1 tools.deps

Для управления зависимостями воспользуемся утилитой tools.deps от разработчиков clojure.

Изначально clojure не имела утилит и распространялась в виде одного jar файла. И зависимостями управляли с помощью maven. Позднее появились сторонние системы сборки написанные на clojure: lein и boot. Кроме того есть диалект clojurescript, компилирующийся в javascript, но для него библиотеки тоже распространяются в виде jar файлов, а не npm пакетов. С релизом 1.9 clojure распространяется в виде 3-х jar файлов и встал вопрос об официальной утилите и собственном формате пакетов.

Итак, tools.deps позволяет:

- подключать зависимости из:
  - maven репозиториев
  - git репозиториев
  - локальных jar файлов
  - локальных подпроектов
- строить java class path на их основе
- запускать repl
- задавать различные entry point, подобно разделу `scripts` в `package.json`

Он хранит конфигурацию проекта в файле `deps.edn`, размещенном в корне проекта. Также можно создать файл `~/.clojure/deps.edn`, который будет использоваться для всех проектов. В нем стоит указывать конфигурацию, специфичную для вас - версии repl и т.п.

EDN расшифровывается как extensible data notation. Он использует clojure синтаксис и поддерживает все структуры данных clojure. Можно провести аналогию форматом JSON, который использует javascript синтаксис.

Прежде чем двигаться дальше стоит изучить документацию:

- https://clojure.org/guides/getting_started
- https://clojure.org/guides/deps_and_cli
- https://clojure.org/reference/deps_and_cli

2.8.2 Docker

Есть готовые образы: https://hub.docker.com/_/clojure/

```
# alpine
run --rm -it clojure:tools-deps-alpine clojure
```

```
# debian
run --rm -it clojure:tools-deps clojure
```

2.8.3 Repl

Т.к. стандартный repl малофункционален, то мы воспользуемся rebel-readline вместо него.

Этот repl легко расширяется, и я сделал собственный вариант, который позволяет:

- перезагружать код в измененных файлах
- запускать тесты

Подробности - https://github.com/darkleaf/repl-tools-deps.

Если вы не работали с Emacs, и не планируете его изучение - это ваш выбор.

Но если вы никогда не использовали редактор с интегрированным repl, то вы живете неправильно.

2.8.4 Emacs + Cider

Наверняка есть и другие редакторы с поддержкой интеграции с repl, но Emacs - by design ориентирован на интерактивную разработку и lisp подобные языки.

cider - пакет для Emacs, превращающий его в полноценную clojure IDE.

Способ подключения cider-nrepl через tools.deeps, описанный в readme, не работает и к тому же не позволяет задать порт и хост на котором запустится сервер nrepl.

Я написал простую обертку - https://github.com/darkleaf/cider-tools-deps

2.8.5 Parinfer

Расставлять и выравнивать скобки - неблагодарное занятие. Но есть плагин для множества редакторов, облегчающий редактирование lisp выражений:

https://shaunlebron.github.io/parinfer/

2.8.6 Code reloading

Для clojure есть библиотека для перезагрузки кода без перезагрузки jvm процесса. Это tools.namespace. Ее использует и cider и repl-tools-deps.

Пока мы будем работать с stateless кодом, а он перезагружается тривиально. В дальнейшем мы столкнемся с stateful кодом и я покажу как с ним работать.

Для перезагрузки кода используйте:

- `:repl/reload` для repl-tools-deps
- `C-c C-x` для cider.

2.8.7 Примеры кода

По ходу изложения будут даваться примеры кода. Все они доступны в директории sources. В readme даны инструкции по запуску repl/cider.

2.9 Either

В дальнейшем нам потребуется моделировать вычисления, которые могут завершиться неудачей. В мире функционального программирования для этого используют монаду Either. Не бойтесь слова «монада» и просто примите как данность, что Either это монада. В конце будет материал для любознательных.

Рассмотрим программу на javascript. Это сценарий входа в систему. Детали функций заменены заглушкой `realLogic()`.

```
function checkLoggedOut() {
  if ( realLogic() ) { return { type: "already-logged-in" } }
  return
}

function findUser(params) {
  if ( realLogic() ) { return { type: "authentication-failed" } }
  return { type: "user", id: 1}
}

function checkAuthentication(user, params) {
  if ( realLogic() ) { return { type: "authentication-failed" } }
  return
}

function checkParams(params) {
  if ( realLogic() ) { return { type: "invalid-params", explain: "some data" } }
  return
}

function logIn(user) {
    realLogic()
    right
}

function process(params) {
  var err

  err = checkLoggedOut()
  if err { return err }

  err = checkParams(params)
  if err { return err }

  let user = findUser(params)

  err = checkAuthentication(user, params)
  if err { return err }

  logIn(user)

  return { type: "processed", user: user }
}
```

Т.е. функция `process` всегда возвращает или успешный ответ или ошибку. Вызывающая сторона обработает результат и соответствующим образом сообщит пользователю системы. В случае с web приложением, это может быть редирект или отображение ошибки.

Для обработки ошибок можно воспользоваться исключениями. Но:

- мы всегда обрабатываем эти ошибки, т.е. это уже не исключительная ситуация
- мы хотим передавать дополнительные данные, например ошибки валидации
- исключения содержат stacktrace, и из-за его формирования снижается производительность

Вместо исключений в предыдущем примере используется ранний возврат из функции. Привет, Golang! Но теперь мы постоянно думаем об ошибках и это засоряет функцию.

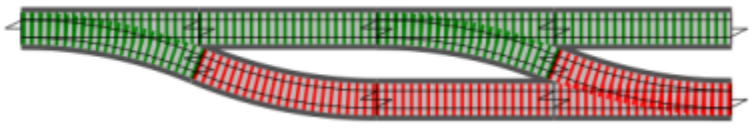
Пример на `javascript` можно переписать на `clojure`:

```
(defn check-logged-out []
  (if (real-logic)
    {:type ::already-logged-in}))

(defn find-user [params]
  (if (real-logic)
    {:type ::authentication-failed}
    {:type :user, :id 1}))

(defn check-authentication [user params]
  (if (real-logic)
    {:type ::authentication-failed}))

(defn check-params [params]
  (if (real-logic)
    {:type ::invalid-params, :explain "some-data"}))

(defn log-in! [user]
  (real-logic))

(defn process [params]
  (or (check-logged-out)
      (check-params params)
      (let [user (find-user params)]
        (or (check-authentication user params)
            (do (log-in! user)
                {:type ::processed :user user})))))
```

Здесь используется `or`, т.к. он вернет первый истинный результат, т.е. не `false` или не `nil`. Функции `check-*` в случае ошибки вернут ассоциативный массив, который считается истинным.

Из-за того, что в `clojure` нет раннего возврата, сильно увеличивается вложенность.

Но есть способ лучше. Мы можем воспользоваться Either. Вводится 2 типа-обертки: `Left` и `Right`. Если в вычислении встречается значение `Left`, то вычисление прерывается и сразу возвращается это значение. Можно провести аналогию с железной дорогой. Если в процессе встречается Left, то движение идет по красной ветке:



railway composition

```
(defn check-logged-out= []
  (if (real-logic)
```

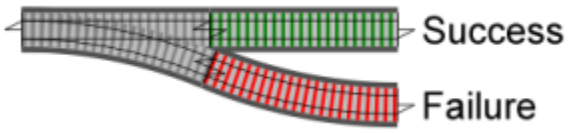
(continues on next page)

(продолжение с предыдущей страницы)

```
    (left {:type ::already-logged-in})
    (right)))

(defn find-user= [params]
  (if (real-logic)
    (left {:type ::authentication-failed, :explain "some-data"})
    (right {:type :user, :id 1})))

(defn check-authentication= [user params]
  (if (real-logic)
    (left {:type ::authentication-failed})
    (right)))

(defn check-params= [params]
  (if (real-logic)
    (left {:type ::invalid-params})
    (right)))

(defn log-in! [user]
  (real-logic))

(defn process= [params]
  (let= [ok   (check-logged-out=)
         ok   (check-params= params)
         user (find-user= params)
         ok   (check-authentication= user params)]
    (log-in! user)
    (right {:type ::processed :user user})))
```

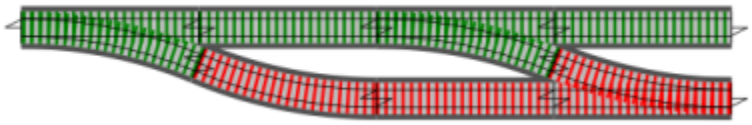
Т.е. если `check-authentication=` вернет `(left {:type ::authentication-failed})`, то и функция `process=` вернет то же самое.

Это напоминает железную дорогу. Функцию `check-logged-out=` можно представить так:



railway fn

А `let=` комбинирует подобные функции следующим образом:



railway composition

Из-за аналогии с рельсами, наши функции, возвращающие Either будут заканчиваться на `=`.

Подробности можно узнать из статьи Railway oriented programming

Таким образом, мы не используем ранний возврат и исключения.

В случае с javascript можно провести аналогию с механизмом `Promise`. Однако, он позволит строить только цепочки, в отличие от `let=`, который позволяет строить сложные зависимости.

2.9.1 Соглашения именования

- `f=` - функция, возвращающая `either`
- `fs=` - коллекция функций, возвращающих `either`
- `mv` - значение, завернутое в `either`
- `mf` - функция, завернутая в `either`

2.9.2 Интерфейс

Есть 2 функции: `left` и `right`. Они принимают в качестве аргумента значение и возвращают контейнер с этим значением. Они могут не принимать значение, тогда в контейнере должен быть `nil`.

T.к. clojure - динамический язык, удобно принять за `Right` любое значение кроме `Left`.

Доступ к значению в контейнере осуществляется с помощью функции `extract`.

```
(t/testing "with value"
      (let [val 42
            l   (sut/left val)
            r   (sut/right val)]
        (t/is (= val
                 (sut/extract l)
                 (sut/extract r)))))

(t/testing "without value"
  (let [l (sut/left)
        r (sut/right)]
    (t/is (= nil
             (sut/extract l)
             (sut/extract r)))))

(t/testing "default right"
  (t/is (sut/right? 1))
  (t/is (sut/right? "str"))
  (t/is (sut/right? []))
  (t/is (sut/right? nil)))
```

Есть предикаты: `(left? x)`, `(right? x)`.

```
(t/testing "left?"
  (t/is (sut/left? (sut/left)))
  (t/is (not (sut/left? (sut/right)))))
(t/testing "right?"
  (t/is (sut/right? (sut/right)))
  (t/is (not (sut/right? (sut/left)))))
```

Полезно иметь функцию, которая меняет тип с `Left` на `Right` и наоборот:

```
(t/testing "invert"
  (let [val 42]
    (t/is (= (sut/left val)
             (sut/invert (sut/right val))))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    (t/is (= (sut/right val)
             (sut/invert (sut/left val))))))
```

Для изменения содержимого контейнеров доступны функции:

- `(bimap left-fn right-fn either)`
- `(map-left left-fn either)`
- `(map-right right-fn either)`

Если в `bimap` передаем `Left`, то к его значению применится первая функция, если `Rigth` - вторая. `map-left` и `map-right` - частные случаи `bimap`.

```
(t/testing "bimap"
  (t/is (= (sut/left 1)
           (->> 0 sut/left (sut/bimap inc identity))))
  (t/is (= (sut/right 1)
           (->> 0 sut/right (sut/bimap identity inc)))))

(t/testing "map-left"
  (t/is (= (sut/left 1)
           (->> 0 sut/left (sut/map-left inc))))
  (t/is (= (sut/right 0)
           (->> 0 sut/right (sut/map-left inc)))))

(t/testing "map-right"
  (t/is (= (sut/left 0)
           (->> 0 sut/left (sut/map-right inc))))
  (t/is (= (sut/right 1)
           (->> 0 sut/right (sut/map-right inc)))))
```

Напомню, что макрос `->>` преобразует `(->> 0 left (map-right inc))` в `(map-right inc (left 0))`.

Макрос `let=` позволяет использовать вместе выражения и прерывать исполнение, если одно из них вернуло `Left`.

```
(t/testing "right"
  (let [ret (sut/let= [x (sut/right 1)
                       y 2]
              (+ x y))]
    (t/is (= (sut/right 3)
             ret))))

(t/testing "left"
  (let [ret (sut/let= [x (sut/left 1)
                       y (sut/right 2)]
              (sut/right (+ x y)))]
    (t/is (= (sut/left 1)
             ret))))
```

Привязки `x` и `y` - соответствуют значениям контейнеров:

```
(let= [x (right 1)
       y (right 2)]
  (prn x) ;; => 1
  (prn y) ;; => 2
  (right (+ x y)))
```

Проверка прерывания исполнения:

```
(t/testing "computation"
  (t/testing "right"
    (let [effect-spy   (promise)
          side-effect! (fn [] (deliver effect-spy :ok))]
      (sut/let= [x (sut/right 1)
                 y (sut/right 2)]
        (side-effect!)
        (sut/right (+ x y)))
      (t/is (realized? effect-spy))))

  (t/testing "left"
    (let [y-spy        (promise)
          effect-spy   (promise)
          side-effect! (fn [] (deliver effect-spy :ok))]
      (sut/let= [x (sut/left 1)
                 _ (deliver y-spy :ok)]
        (side-effect!))
      (t/is (not (realized? y-spy)))
      (t/is (not (realized? effect-spy))))))
```

Для проверки прерывания исполнения используются «шпионы». Шпион, это промис, и мы можем проверить с помощью предиката `realized?` было ли доставлено ему какое-либо значение или нет. Таким образом можно понять, вызывался ли тот или иной кусок кода.

Полезно иметь поддержку распаковки:

```
(t/testing "destructuring"
  (let [ret (sut/let= [[x y] (sut/right [1 2])]
              (+ x y))]
    (t/is (= (sut/right 3)
             ret))))
```

Функция `>>=` позволяет строить цепочки следующего вида `(>>= either-value some-fn= another-fn=)`. Т.е. ее первый аргумент - `Either`, а последующие - функции, принимающие обычные значения и возвращающие `Either`. При этом если первый аргумент `Left` или любая функция вернула `Left`, то выполнение прерывается.

```
(t/testing "right rights"
  (let [mv   (sut/right 0)
        inc= (comp sut/right inc)
        str= (comp sut/right str)
        ret  (sut/>>= mv inc= str=)]
    (t/is (= (sut/right "1")
             ret))))

(t/testing "left right"
  (let [mv   (sut/left 0)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        inc= (comp sut/right inc)
        ret  (sut/>>= mv inc=)]
    (t/is (= (sut/left 0)
             ret))))

(t/testing "right lefts"
  (let [mv   (sut/right 0)
        fail= (fn [_] (sut/left :error))
        ret  (sut/>>= mv fail=)]
    (t/is (= (sut/left :error)
             ret)))))
```

Макрос >> тоже строит цепочки, но в отличие от >>= цепочки значений, а не функций. Он полезен для последовательного вызова независимых функций. При этом, если в его аргументах оказался `Left`, то он прерывает цепочку.

```
(>> (check-attrs= attrs)
    (update-post= post attrs))
```

Если за `Left` принять `false`, а за Right - `true`, то >> будет подобен `and`, т.е. будет вычислять выражения до первого ложного:

```
(and
 (do (prn 1) true)
 (do (prn 2) false)
 (do (prn 3) true)) ;; 3 не будет напечатано
```

```
(t/testing "rights"
  (let [ret (sut/>> (sut/right 1)
                    2)]
    (t/is (= (sut/right 2)
             ret))))
(t/testing "lefts"
  (let [spy (promise)
        ret (sut/>> (sut/left 1)
                    (deliver spy :ok))]
    (t/is (= (sut/left 1)
             ret))
    (t/is (not (realized? spy)))))
```

Оригинальный `let` неявно заворачивает свое тело в `do`:

```
(let [x 1]
  (prn x)
  x)

(let [x 1]
  (do
    (prn x)
    x))
```

И это используется только для побочных эффектов, т.к. значением формы `(let ...)` станет последнее выражение внутри этой формы. Т.е. результат `(prn x)` игнорируется.

Не будем менять эту семантику для `let=`:

```
(let= [x (right 1)]
  (prn x) ;; => напечатает 1
  (right x))
```

Возможно вы заходите сделать так:

```
(let= [x (right 1)]
  (some-fn=)
  (right x))
```

В этом случае результат `some-fn=` будет проигнорирован, даже если это будет `Left`, и результатом будет `(right 1)`.

Явно используйте `>>`:

```
(let= [x (right 1)]
  (>> (some-fn=)
      (right x)))
```

2.9.3 Задание

Проект содержит заготовку неймспейса `either.core` и рассмотренные тесты.

Склонируйте этот репозиторий, запустите окружение и проверьте, что все тесты падают.

Задание разбито на 3 этапа:

1. реализация типов и функций над ними
2. реализация `let=`
3. реализация `>>=` и `>>`

При выполнении внимательно смотрите на тесты. Прочитайте шпаргалку. В конце этого параграфа будет ссылка для самостоятельной проверки.

2.9.4 Шпаргалка

https://clojure.org/api/cheatsheet - ваш главный справочный материал. Ищите здесь информацию по любой стандартной функции/макросу.

Вам потребуются Типы. Типы - это просто java классы, они не реализуют никаких дополнительных протоколов и интерфейсов.

```
(deftype T [val])

(->T 1) ;; конструктор.
```

`->` - просто часть имени автоматически сгенерированной функции-конструктора.

Типы также как и Записи поддерживают протоколы:

```
(defprotocol Proto
  (method [this]))

(deftype T []
  Proto
  (method [this] :ok))
```

Вместо ветвления (if, case) используйте полиморфизм.

```
(defrecord Either [val kind])

(defn left? [either]
  (= (:kind either) :left))

(defn right? [either]
  (= (:kind either) :right))
```

```
(defprotocol Either
  (left? [this])
  (right? [this]))

(deftype Left [val]
  Either
  (left? [_] true)
  (right? [_] false))

(deftype Right [val]
  Either
  (left? [_] false)
  (right? [_] true))
```

Любой существующий тип, или все типы сразу можно расширить протоколом. Однако `(= nil (class nil)`, т.е. `nil` не наследует от `Object`, поэтому `nil` требует объявления отдельной реализации протокола.

```
(extend-protocol Either
  Object
  (left? [this] false)
  (right? [this] true)

  nil
  (left? [this] false)
  (right? [this] true)
```

Т.к. Типы - по умолчанию не реализуют ничего, то вам нужно реализовать печать их экземпляров:

```
(deftype T [val])

(defmethod print-method T [v ^java.io.Writer w]
  (doto w
    (.write "#<T ")
    (.write (pr-str (.val v)))
    (.write ">")))
```

При использовании протокола первый аргумент всегда - экземпляр класса, типа или записи. Бывают ситуации, когда нужно поменять порядок аргументов.

Делайте это с помощью функции-обертки:

```
(defprotocol P
  (m1 [this])
  (-m2 [this x y])

(defn m2 [x y this]
  (-m2 this x y))
```

Используйте паттерн Null-object. В частности функцию identity.

Например:

```
(map identity some-collection)
```

Будет возвращена новая коллекция из тех же элементов.

Функции могут иметь различные определения в зависимости от количества аргументов:

```
(defn foo
  ([] (foo nil))
  ([x] :some-body))
```

Если вы хотите сделать функцию с произвольным количеством аргументов, то переменный вариант должен принимать столько же или больше аргументов:

```
(defn foo
  ([x y] :do-something)
  ([x y & ys] :do-another))
```

Добавляйте в функцию поддержку переменного количества аргументов с помощью `cons` и `reduce`:

```
(defn foo
  ([x y] :do-something)
  ([x y & ys] (reduce foo x (cons y ys))))
```

Иногда нужно использовать функции, которые еще не объявлены:

```
(declare x)

(defn y []
  (x))

(defn x [] :ok)
```

Макросы могут быть рекурсивными.

Отлаживайте макросы:

```
(-> '(let= [x (left 1)
            y (right 2)]
       (right (+ x y)))
    macroexpand-1
    clojure.pprint/pprint)
```

Для cider используйте `M-x cider-macroexpand-1` или `C-c RET`.

Не забывайте о шаблонизации для макросов:

```
(defmacro foo [x y]
  `(+ ~x ~y))

(defmacro bar [& body]
  `(do ~@body))
```

Если вам нужно объявить какой-то символ внутри макроса, используйте генератор символов:

```
(defmacro foo [x y]
  `(let [z# 1]
     (+ ~x ~y z#)))


(let [z 3]
  (foo z z)) ;; => 7

;; (foo z z) преобразуется в
;; (let [z__14213__auto__ 1]
;;   (+ z z z__14213__auto__))
```

Бывают ситуации, когда такой способ не работает. Например, вы вручную собираете форму:

```
(defmacro foo [y]
  (let [val (gensym "val")]
    `(let [~val ~y]
       ~(list `+ val 2))))

(foo 1)

;; (let [val15558 1] (+ val15558 2))
```

Используйте утверждения:

```
(assert (-> bindings count even?))
```

Используйте те функции при работе с коллекциями, которые выражают ваши намерения:

```
;; добавить элемент эффективным способом
(let [l (list 1 2 3)]
  (conj l 0)) ;; => (0 1 2 3)

(let [v [1 2 3]]
  (conj v 0)) ;; => [1 2 3 0]

;; добавить элемент в начало коллекции и получить последовательность
(let [l (list 1 2 3)]
  (cons 0 l)) ;; => (0 1 2 3)

(let [v [1 2 3]]
  (cons 0 v)) ;; => (0 1 2 3)
```

2.9.5 Ответ

https://github.com/darkleaf/either

2.9.6 Для любознательных

Для тех, кто знает Haskell, фактически мы реализуем вместо `Either` нечто вроде монадного трансформера `EitherT a (IO b)`, т.к. функции в Clojure могут иметь побочные эффекты.

`bimap`, `>>=`, `>>` взяты из Haskell. Последние 2 адаптированы для использования с переменным количеством аргументов.

Clojure не Haskell. Haskell имеет мощную систему типов. Также он ленивый, т.е. не вычисляет аргументы функции до ее вызова и не гарантирует порядок вычислений. Поэтому `>>` - макрос, а не функция, чтобы отложить вычисления. Он подобен макросу `or`, который вычисляет аргументы до первого истинного.

В Haskell есть так называемая `do` нотация, фактически синтаксический сахар:

```
do
  x <- Left "error"
  y <- Right 2
  right(x + y)
-- #> Left "error"

do
  x <- Right 1
  y <- Right 2
  right(x + y)
-- #> Right 3
```

Это эквивалентно:

```
-- \x -> x - лямбда

Left "error" >>= (\x -> Right 2 >>= (\y -> right (x + y)))
-- #> Left "error"

Right 1 >>= (\x -> Right 2 >>= (\y -> right (x + y)))
-- #> Right 3
```

Возможно вы заметили, что у нашего `let=` и `do` нотации есть много общего. Сравните:

```
(let= [x (right 1)
       y (right 2)]
   (right (+ x y)))
```

```
do
  x <- Right 1
  y <- Right 2
  right(x + y)
```

В отличие от Haskell, для Clojure, нет нужды обеспечивать порядок вычислений и реализовывать поддержку прочих монад. К тому же создание множества анонимных функций и множественные вызовы `>>=` существенно уменьшат производительность. Поэтому `let=` реализован как макрос, а с его помощью `>>=` и `>>`.

2.10 Spec

Начиная с версии 1.9 clojure поставляется с библиотекой clojure.spec. Она добавляет возможность создания спецификаций данных и функций. Благодаря спецификациям можно

- валидировать данные
- генерировать тестовые данные
- разбирать данные на составные части (destructuring)
- проверять входные и выходные параметры функций
- автоматически тестировать функции (generative tests)

Ознакомьтесь с официальными материалами:

- rationale and overview
- spec guide

Поэкспериментируйте с библиотекой в тестовом проекте.

Обратите внимание на комментарии ниже.

2.10.1 Комментарии

Спецификации напоминают статическую типизацию, только проверки выполняются в рантайме. Однако, есть экспериментальный проект spectrum запускающий проверки спецификаций в compile time.

При использовании `st/instrument` проверяются только аргументы функции, но не `:ret` и `:fn`. Возможно, это поведение изменится, а пока можно воспользоваться библиотекой orchestra, которая позиционируется как замена `clojure.spec.test.alpha`.

Нужно быть внимательным при создании спецификаций функций высшего порядка, т.к. instrument будет проверять соответствие спецификации принимаемой или возвращаемой функции. Это допустимо для чистых функций, но неприемлемо для функций с побочным эффектом:

```
(s/fdef f
   :args (s/cat :g (s/fspec
                    :args (s/cat :x int?)
                    :ret int?)))

(defn f [g]
  (g 42))

(require '[clojure.spec.test.alpha :as st])
(st/instrument `f)

(f str)
;;=> ExceptionInfo Call to #'user/f did not conform to spec:
;;=> In: [0] val: "0" fails at: [:args :g :ret] predicate: int?

(f inc)
;;=> 43

(f (fn [x] (prn x) x))
-1
0
-1
0
-4
-1
-2
-11
-4
0
5
81
-3
196
12
-1853
83
1399
-3
-11
-57026
42
;;=> 42
```

Выход - просто проверять, что аргумент любая функция:

```
(s/fdef f :args (s/cat :g fn?))
```

Instrument не работает для протоколов. Используйте обертки:

```
(defprotocol P
  (-foo [x y z]))

(s/fdef foo ...)

(defn foo [x y z]
  (-foo x y z))
```

Хочется использовать spec для валидации форм, но сгенерировать понятные пользователю сообщения об ошибках из структуры `explain-data` - нетривиальная задача. В этом поможет библиотека phrase.

Генератор не всемогущ и использует перебор:

```
(s/def ::login (s/and string? #(re-matches #"\w{3,255}" %)))
(-> ::login s/gen sgen/generate) ;;=> "wbW8"

;; UTF symbols
(s/def ::smile (s/and string? #(re-matches #"[]" %)))
(-> ::smile s/gen sgen/generate)
;;=> ExceptionInfo Couldn't satisfy such-that predicate after 100 tries.
```

Существуют библиотеки генераторов, которые, например, могут по регулярному выражению сгенерировать требуемую строку: test.chuck, strgen.

Глава 3

Design

3.1 Объектно-ориентированное проектирование

Перед тем как двигаться дальше, важно понять как Clojure соотносится с наработками в области объектно-ориентированного проектирования.

В книге «Совершенный код» есть мысль:

> Как сказал Дэвид Грайс, подход к программированию не должен определяться используемыми инструментами. В связи с этим он проводит различие между программированием *на* языке (programming *in* language) и программированием *с использованием* языка (programming *into* language). Разработчики, программирующие «на» языке, ограничивают свое мышление конструкциями, непосредственно поддерживаемыми языком. Если предоставляемые языком средства примитивны, мысли программистов будут столь же примитивными.
>
> Разработчики, программирующие «с использованием» языка, сначала решают, какие мысли они хотят выразить, после чего определяют, как выразить их при помощи конкретного языка.

В книге «Приемы объектно-ориентированного проектирования. Паттерны проектирования» есть абзац, показывающий связь между парадигмой языка и способностью реализовывать паттерны:

> Выбор языка программирования безусловно важен. В наших паттернах подразумевается использование возможностей Smalltalk и C++, и от этого зависит, что реализовать легко, а что - трудно. Если бы мы ориентировались на процедурные языки, то включили бы паттерны **наследование**, **инкапсуляция** и **полиморфизм**. Некоторые из наших паттернов напрямую поддерживаются менее распространенными языками.

В книге Clean Architecture дается определение объектно-ориентированному проектированию(OO):

> What is OO? There are many opinions and many answers to this question. To the software architect, however, the answer is clear: OO is the ability, through the use of polymorphism, to gain absolute control over every source code dependency in the system. It allows the architect to create a plugin architecture, in which modules that contain high-level policies are independent of

> modules that contain low-level details. The low-level details are relegated to plugin modules that can be deployed and developed independently from the modules that contain high-level policies.

Clojure имеет поддержку динамического полиморфизма по определению, т.к. Clojure - динамический язык. Кроме этого он имеет концепции мультиметодов и протоколов, с которыми мы познакомились ранее. И тот факт, что Clojure функциональный язык, не должен мешать применять подходы объектно-ориентированного проектирования.

3.2 SOLID

Полнее всего SOLID раскрывается в материалах Роберта Мартина:

- Clean Architecture
- SOLID Principles

SOLID - набор принципов, следование которым сделает систему поддерживаемой и расширяемой. Они перекликаются с идеями Agile, Роберт Мартин один из авторов Agile manifesto.

3.2.1 The Single Responsibility Principle (SRP)

Формулировка: *A module should be responsible to one, and only one, actor.*

Под модулем тут понимается модуль/класс/неймспейс/файл. Под ответственностью понимается, что создание этого модуля, его редактирование и удаление контролирует одно действующее лицо. Это действующее лицо - не программист, а бизнес.

Допустим, мы делаем карточную игру. На проекте есть действующее лицо, например аналитик, которое формулирует правила игры. Есть дизайнер, который отвечает за внешний вид стола, карт, игроков. Есть администратор баз данных, который проектирует схему базы данных и запросы к ней. Согласно этому принципу не должно быть модуля, который изменяется из-за требований разных действующих лиц. Все изменения в этом модуле может запрашивать только одна роль.

Очевидно, что ролей может быть больше. Например, есть редактор-переводчик, отвечающий за текстовые сообщения в игре, в том числе на разных языках. В этом случае UI не может содержать текст, и обязательно использование движков локализации, хранящих переводы отдельно от кода UI.

Исходя из своего опыта вы можете представлять одного человека как несколько действующих лиц. Например, один и тот же человек в команде отвечает за правила в игре и дизайн. Но вы понимаете, что проект будет разрабатываться долго и вероятно, что дизайном займется отдельный человек, по этому вы трактуете требования одного аналитика как требования 2х действующих лиц. Или вы уверены, что локализация приложения не потребуется, а ваша команда не имеет опыта локализации, тогда вы принимаете решение что код UI будет содержать текст сообщений.

3.2.2 The Open Closed Principle (OCP)

Формулировка: *A software artifact should be open for extension but closed for modification.*

Под software artifact тут понимается файл с исходным текстом, бинарный файл вроде jar или dll, пакет вроде gem или npm.

Речь идет от том, что без модификации артефакта мы можем менять его поведение. Например, артефакт содержит код рассылки писем клиентам. Но объект, рассылающий письма, ничего не знает о способе доставки. И артефакт не содержит кода доставки. Будет ли это SMTP сервер или сервис, или SMS мы решим когда воспользуемся этим артефактом.

Фактически речь идет о плагинах.

Опять-таки, это не правило, а принцип. Чтобы сделать действительно расширяемую систему мы должны знать будущие требования. Исходя из своего опыта мы можем предвидеть будущие требования и сделать систему гибкой в нужных местах. При этом мы должны учитывать, что обеспечение гибкости имеет свою цену.

3.2.3 The Liskov Substitution Principle (LSP)

Имеет сложное математическое определение, которое можно заменить на: *Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.*

Компилятор не может автоматически проверить то, что кто-то нарушит контракт, поэтому эта проверка возлагается на разработчика.

Например, есть класс Stack, реализующий следующий интерфейс: length, push, pop. И вы ожидаете, что любой стек увеличивает длину на единицу, если в него что-то положили. Некто создает потомка DoubleStack, который дублирует добавляемые элементы. С точки зрения компилятора, функции, работающие со Stack, могут работать и с DoubleStack, но определенно, что эти функции не будут работать корректно с этим потомком.

Это относиться не только к наследованию. Производный тип можно получить, например с помощью паттерна декоратор.

3.2.4 The Interface Segregation Principle (ISP)

Формулировка: *Make fine grained interfaces that are client specific.*

Можно перевести как: *Клиенты не должны зависеть от методов, которые они не используют.*

Этот принцип относится только к тем языкам, которые содержат концепции подобные абстрактным классам и интерфейсам. Например, в Clojure это протоколы.

3.2.5 DIP: The Dependency Inversion Principle

Формулировка: *Depend on abstractions, not on concretions.*

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Что такое модули верхних уровней? Как определить этот уровень? Чем ближе модуль к вводу/выводу, тем ниже уровень модуля. Т.е. модули низкого уровня работают с базой данных, интерфейсом пользователя, сокетами и т.п. А модули, реализующие бизнес-логику — высокого уровня.

Зависимость модулей - это ссылка на модуль в исходном коде, т.е. import, require и т.п. Но как бизнес-логика не будет зависеть от модулей работы с базой данных? Важно разделять compile-time и runtime зависимости. Да, в runtime бизнес-логика обязательно использует модули взаимодействия с базой, но исходный код, содержащий бизнес-логику, не ссылается напрямую на модуль низкого уровня. Такой трюк возможен благодаря динамическому полиморфизму.

Есть модуль Logic, реализующий логику, который должен отсылать уведомления. В этом же пакете объявляется интерфейс ISender, который используется Logic. Уровнем ниже, в другом пакете объявляется ConcreteSender, реализующий ISender. Получается, что в момент компиляции Logic не зависит

от ConcreteSender. В runtime, мы можем настроить Logic так, чтобы он работал с ConcreteSender. Как правило, конструктор класса Logic позволяет передать экземпляр ConcreteSender.

Такое направление зависимостей позволяет начать разработку с самого главного - с высокоуровневых правил. При этом мы должны представлять, какими будут модули нижних уровней, чтобы правильно выбрать абстракции. Мы должны решить, будет ли наш проект использовать key-value хранилище или это будет реляционная база, будет ли оно поддерживать ACID транзакции, способно ли оно выполнять полнотекстовые запросы, будет ли использоваться шардирование.

Начиная с модулей верхнего уровня мы можем корректно подсчитать расходы. Допустим, мы не следуем этому принципу и на проект затрачено 100 монет и мы не можем сказать сколько из них потрачено на бизнес-правила, а сколько на интерфейс. Но если мы следуем этому правилу, то после реализации бизнес-правил можно подсчитать затраты, допустим, потрачено 10 монет. Соответственно, бизнес может решить не тратить в следующий раз 90 монет на интерфейс.

3.2.6 Резюме

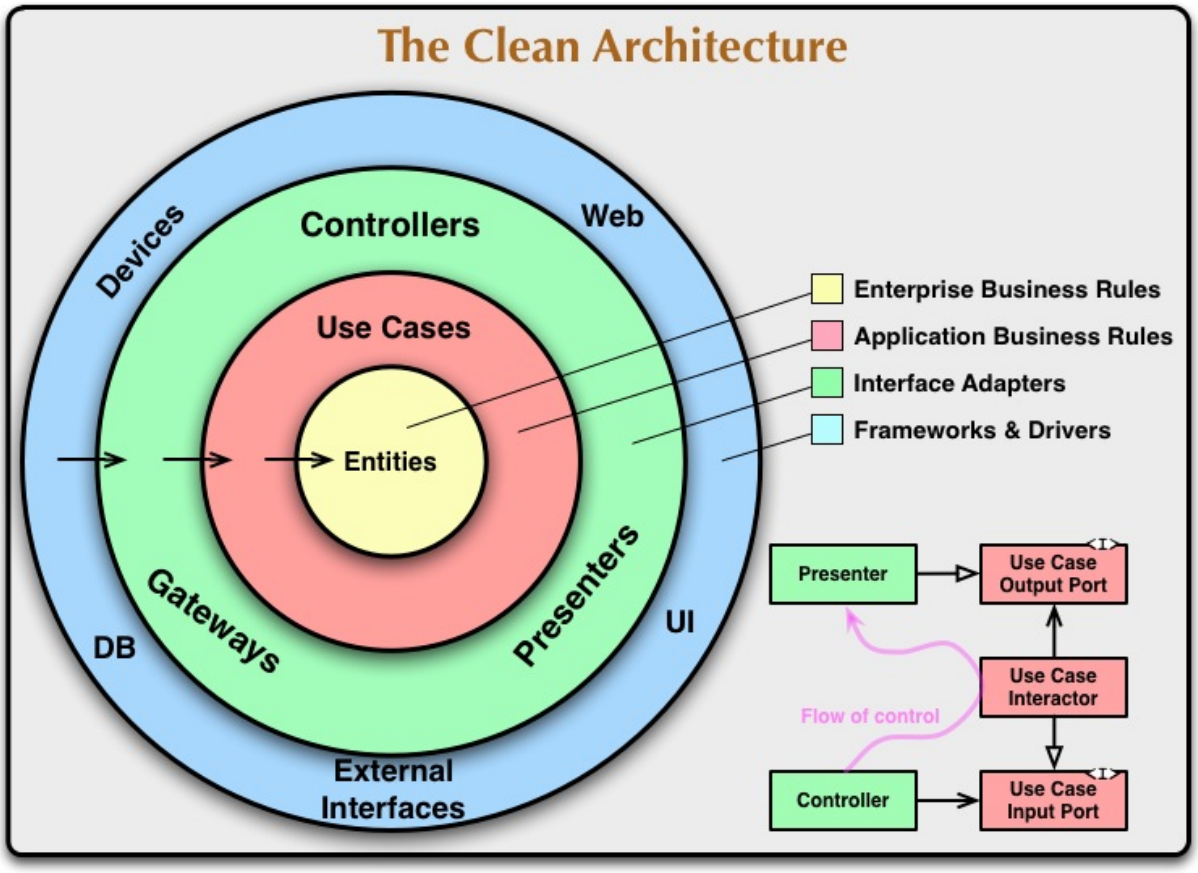
SOLID - набор принципов, а не правил. Нужно осознано идти на нарушение этих принципов, ведь абстракции имеют свою цену. Взять тот же OCP, не стоит все делать расширяемым, жизнь все равно поставит задачу, для которой система будет не расширяемой. Нет серебряной пули, именно поэтому нам платят деньги за поиск баланса и принятие технических решений.

3.3 Clean Architecture

Базовое описание дается в статье The Clean Architecture.

Подробности можно узнать в этих материалах:

- Clean-Architecture
- Clean Code Episode 7



Clean Architecture

3.3.1 Комментарии

Clean Architecture базируется на DIP принципе SOLID.

Может быть больше или меньше слоев, но Dependency Rule должно соблюдаться.

Понять как разделить логику между слоями Entities и Use Cases можно по следующему правилу. Если бизнес-правило существует без автоматизации, то оно относится к слою Entities. Если бизнес-правило содержит элементы автоматизации, вроде рассылки писем, подразумевает наличие пользователя системы, то оно относится к слою Use Cases.

Представьте, что компьютеры еще не изобретены и вы владелец банка. Естественно, что есть некие бизнес-процессы, например, перевод денег со счета на счет. В помещении располагается картотека, а клерк ищет карточки клиентов, проверяет баланс и проводит операцию перевода. Таким образом правило перевода денег относится к Entities.

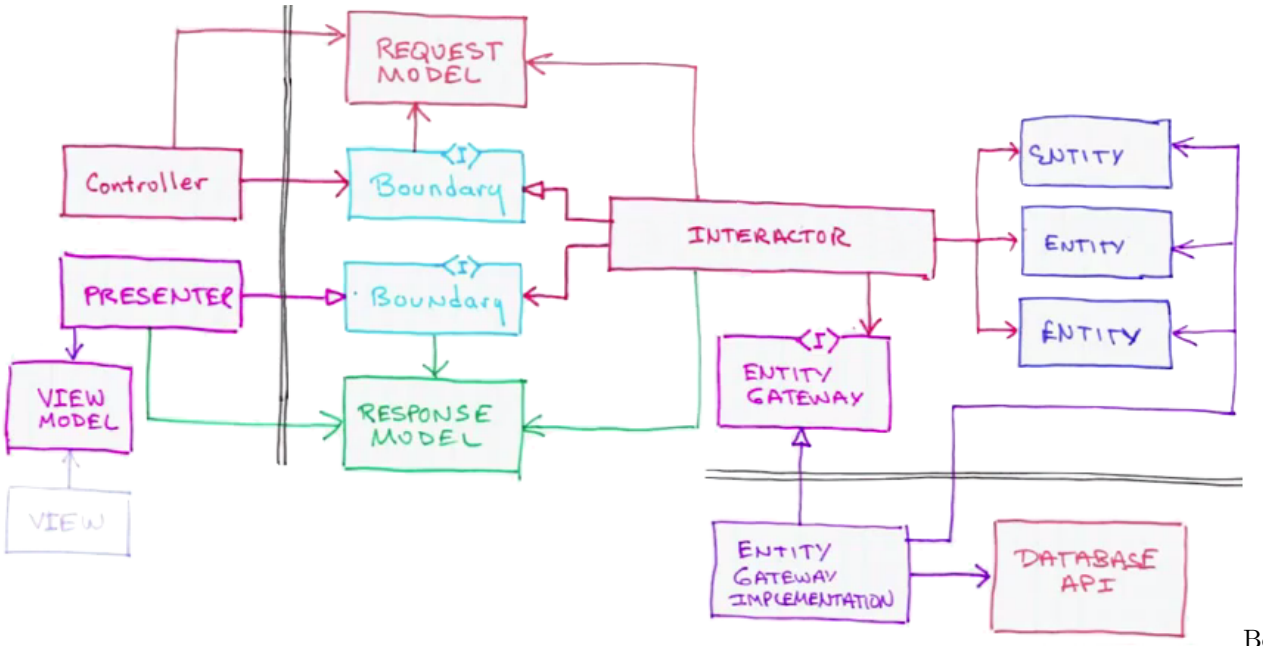
Но может быть и Use Case, использующий правила уровня Entities. Например, текущий пользователь системы переводит деньги другому пользователю, при этом получатель должен получить уведомление.

Под сценарием(use case) тут понимается один запрос к системе(backend).

T.e. это не сценарий:

- пользователь переходит на страницу входа в систему
- пользователь вводит логин и пароль, нажимает на кнопку войти
- переходит на страницу перевода денег
- выбирает получателя
- ...

Use Case не могут вызывать друг-друга. При взгляде на код сценария мы должны видеть все его шаги. Никто не запрещает выносить общий код в службы(service).



Boundaries

Эту диаграмму сложно понять без примера. Вот пример на ruby:

```ruby
class Interactor
  def initialize(presenter)
    @presenter = presenter
  end

  def call(request_model)
    response_model = do_smth(request_model)
    presenter.present response_model
    nil
  end
end


presenter = Presenter.new
interactor = Interactor.new presenter
controller = Controller.new interactor
request = Request.new
```

(continues on next page)

(продолжение с предыдущей страницы)

```
controller.process request
view_model = presenter.view_model
view = View.new
view.deliver view_model
```

При таком подходе получается, что сценарий устанавливает презентеру `response_model`. Т.е. презентер имеет состояние.

Позднее, мы *разберем* как это соотносится с web.

3.4 Domain-driven design

Мы будем использовать некоторые идеи из методологии Domain-driven design. Впервые она описана в одноименной книге. В основном мы будем использовать информацию из глав 5 и 6.

3.4.1 Объект-значение

Языки программирования содержат различные примитивные типы, вроде чисел, строк, символов. Их еще называют значениями. Они неизменяемы. Значения могут быть равны, даже если хранятся в разных областях памяти.

Объект-значение подобен примитивным типам, но моделирует понятие предметной области. Примерами могут быть цвет или деньги. Цвета равны, если равны их RGB компоненты. Мы не можем поменять компоненту R зеленому цвету, т.к. это будет уже другой цвет.

Т.е. объект-значение неизменен и полностью идентифицируется своими атрибутами.

3.4.2 Сущность

Напротив, сущность определяется только своим идентификатором и может изменяться.

Например, есть публикация в интернете. Ее URL есть ее идентификатор. При этом в разные моменты времени ее содержимое может быть разным.

Мы уже касались этой темы, когда разбирали управление состоянием в clojure и разбирали ссылочные типы вроде атомов.

3.4.3 Ссылки / Ассоциации

Возьмем публикации в интернете или в журналах. Публикации могут ссылаться друг на друга. Но связь эта однонаправленная. Например, публикации А и Б ссылаются на публикацию Г, при этом Г не знает, кто на нее ссылается. Очевидно, что и Г может ссылаться на А и Б, но А и Б не будут знать этого. Конечно, можно построить некий индекс и определить кто ссылается на публикацию, но это внешний по отношению к публикациям механизм.

Поддержка двунаправленных связей дорога, по этому при моделировании выбирают наиболее важное направление. Например, есть Автор и Публикация. Автор может создать несколько публикаций. Представите, что у вас есть карточки в картотеке. Можно указать автора на карточке публикации. Но тогда узнать о всех публикациях автора можно будет только по стороннему индексу. Практичнее записывать ссылки на публикации в карточке автора. Имея карточку автора можно понять были ли у него публикации, сколько их.

3.4.4 Агрегаты

Большое количество связей между сущностями затрудняет их понимание. Поэтому стремятся сократить количество ассоциаций.

Агрегаты - подход к организации сущностей, чтобы сократить количество связей между ними.

Каждый агрегат имеет корневую сущность и вложенные сущности. Любая внешняя сущность может ссылаться только на корень агрегата. Корневая сущность может выдавать только временные ссылки на вложенные сущности. Все изменения вложенных сущностей проходят только через корень. Корень отслеживает целостность всего агрегата. Транзакция не может пересекать границу агрегата, иными словами в транзакции участвует агрегат целиком.

Подробнее на сайте Мартина Фаулера.

3.4.5 Службы(сервисы)

Представим, что есть автомобиль и авто-мойка. Автомобиль моется в авто-мойке? Или авто-мойка моет машину?

Служба не содержит состояния и просто моделирует некое действие.

3.4.6 Комментарии

Entity, Value object, Service - ортогональные понятия относительно Clean Architecture. Да, верхний слой в Clean Architecture называется Entities, и это вносит путаницу. Слои Entities и Use Cases могут содержать свои Сущности, Объекты-значения и Службы.

Отмечу, что никто кроме корневой сущности агрегата снаружи не может иметь постоянную ссылку на внутренние сущности этого агрегата. В добавок, агрегат должен извлекаться целиком.

Допустим, есть агрегат с корнем Пост и вложенными Комментариями. Эти сущности имеют простые, не составные идентификаторы. В этом случае http приложение не может иметь такой ресурс `/posts/:post-id/comments/:comment-id`, т.к. ни у кого не должно быть возможности получить постоянную сслыку на внутреннюю сущность. Но приложение вполне может иметь ресурс `/posts/:post-id/comments`. В случае с html страницами можно использовать якоря для ссылки на внутреннюю сущность: `/posts/:post-id/#comment-id`.

Однако, комментарий, как корень другого агрегата, может иметь составной идентификатор `[post-id, post-comment-id]`. При этом `post-comment-id` должен быть уникальным только в контексте поста. И в этом случае ресурс `/posts/:post-id/comments/:comment-id` корректен.

Разумеется, URL - это всего лишь деталь способа доставки приложения пользователю, и он должен зависеть от типа идентификатора, а не наоборот.

Из-за отсутствия постоянных ссылок на внутренние сущности их удобно хранить в векторах, где идентификатором сущности выступает ее индекс в векторе.

3.5 Dependency injection

Есть принцип Dependency Inversion (DIP), но он не говорит как именно объект получает свои зависимости.

Есть несколько способов внедрить в объект зависимость, например:

- через конструктор / сеттер
- с помощью Service Locator

Рассмотрим их на примере http приложения, создающего пользователей. Оно состоит из:

- роутера
- экшена контроллера
- юзкейса
- сущности пользователя

При создании, пользователь-объект должен получить идентификатор. Также приложение должно послать уведомление о создании пользователя.

Именно для IdGenerator и Notifier будет применяться принцип инверсии зависимости, т.е. приложение будет знать только об интерфейсе, а не реализации зависимостей.

3.5.1 Внедрение через конструктор

Сначала разберем пример на js, а потом перейдем к clojure.

Сущность User - простая структура, просто принимающая id и логин через конструктор:

```
class User {
  constructor(id, login) {
    this.id = id;
    this.login = login;
  }
}
```

Чтобы при создании пользователя ему устанавливался сгенерированный идентификатор, нужно создавать пользователя с помощью фабрики:

```
class User {
  constructor(id, login) {
    this.id = id;
    this.login = login;
  }
}

class UserFactory {
  constructor(idGenerator) {
    this.idGenerator = idGenerator;
  }

  build(login) {
    const id = this.idGenerator.generate();
    return User.new(id, login);
  }
}
```

Как раз в фабрику через конструктор внедряется idGenerator. И фабрика знает только о его интерфейсе, т.е. о методе `generate`, но не о его реализации.

В clojure нет привычных конструкторов. Воспользуемся функциями. Названия функций-конструкторов будут начинаться с `->`, чтобы отличить их от обычных функций.

```
(ns app.constructor)

(defrecord User [id login])

(defn ->user-factory [id-generator]
  (fn [login]
    (->User (id-generator) login)))

(defn ->create-user-use-case [user-factory notifier]
  (fn [login]
    (let [user (user-factory login)]
      (notifier user)
      user)))

(defn ->create-user-action [create-user-use-case]
  (fn [req]
    (let [login (-> req :params :login)
          user  (create-user-use-case login)
          id    (:id user)]
      {:status  302
       :headers {"Location" (str "/users/" id)}})))

(defn ->routing [create-user-action]
  (fn [req]
    (cond
      (= (:url req) "/users") (create-user-action req)
      :else {:status 404})))

(defn ->id-generator []
 (let [counter (atom 0)]
   (fn []
     (swap! counter inc))))

(defn ->notifier []
  (fn [user]
    (prn user)))

(defn main []
  (let [id-generator         (->id-generator)
        notifier             (->notifier)
        user-factory         (->user-factory id-generator)
        create-user-use-case (->create-user-use-case user-factory notifier)
        create-user-action   (->create-user-action create-user-use-case)
        routing              (->routing create-user-action)]
    (routing {:url "/users", :params {:login "Admin"}})))

(main)
```

Т.е. каждый компонент не зависит напрямую ни от чего, все зависимости он получает через конструктор с помощью замыкания. При этом нужно явно связывать компоненты в функции main.

При этом в одном рантайме мы можем иметь сколько угодно копий приложения с разными зависимостями, т.к. мы не используем глобальное состояние.

В примитивном случае такой подход требует явного конфигурирования зависимостей, как это сделано в функции `main`.

Представьте себе приложение из пары десятков юзкейсов. Сколько зависимостей придется явно сконфигурировать?

Существуют целые фреймворки(IoC container) для автоматического управления зависимостями. Некоторые из них вместо явного конфигурирования самостоятельно ищут реализации абстракций (Convention over Configuration).

3.5.2 Service locator

Локатор сервисов - глобальный объект, разрешающий зависимость. В мире статически типизированных языков считается антипаттерном.

```
(ns app.service-locator)

(def service-locator (atom {}))

(defrecord User [id login])

(defn user-factory [login]
  (let [{:keys [id-generator]} @service-locator]
    (->User (id-generator) login)))

(defn create-user-use-case [login]
  (let [{:keys [notifier]} @service-locator
        user (user-factory login)]
    (notifier user)
    user))

(defn create-user-action [req]
  (let [login (-> req :params :login)
        user  (create-user-use-case login)
        id    (:id user)]
    {:status  302
     :headers {"Location" (str "/users/" id)}}))

(defn routing [req]
  (cond
    (= (:url req) "/users") (create-user-action req)
    :else {:status 404}))

(defn ->id-generator []
 (let [counter (atom 0)]
   (fn []
     (swap! counter inc))))

(defn ->notifier []
  (fn [user]
    (prn user)))

(defn main []
  (swap! service-locator assoc
         :id-generator (->id-generator)
         :notifier     (->notifier))
  (routing {:url "/users", :params {:login "Admin"}}))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(main)
```

В этом случае между компонентами есть явные зависимости на уровне исходного кода, и внедряются только `id-generator` и `notifier`.

В случае clojure локатор сервисов напоминает Var. Только Var хранит одну зависимость и менее нагляден. Мы всегда можем заменить корневое значение переменной с помощью `with-redefs`.

При использовании service locator или `with-redefs` в одном рантайме может быть только одна копия приложения, т.к. используется глобальное стояние.

3.5.3 Dynamic binding

Мы можем использовать динамические переменные, которые позволяют устанавливать их значение для текущего потока исполнения. При этом clojure функции умеют сохранять этот контекст и предавать его во вновь созданные потоки.

```
(ns app.binding)

(declare ^:dynamic *id-generator*)
(declare ^:dynamic *notifier*)

(defrecord User [id login])

(defn user-factory [login]
  (->User (*id-generator*) login))

(defn create-user-use-case [login]
  (let [user (user-factory login)]
    (*notifier* user)
    user))

(defn create-user-action [req]
  (let [login (-> req :params :login)
        user  (create-user-use-case login)
        id    (:id user)]
    {:status  302
     :headers {"Location" (str "/users/" id)}}))

(defn routing [req]
  (cond
    (= (:url req) "/users") (create-user-action req)
    :else {:status 404}))

(defn ->id-generator []
 (let [counter (atom 0)]
   (fn []
     (swap! counter inc))))

(defn ->notifier []
  (fn [user]
    (prn user)))

(defn main []
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  (binding [*id-generator* (->id-generator)
            *notifier*     (->notifier)]
    (routing {:url "/users", :params {:login "Admin"}})))

(main)
```

Таким образом мы убрали конфигурирование зависимостей и получили возможность запускать несколько копий приложения в одном рантайме. Но теперь нельзя подменить любоую зависимость только внутри одного треда.

Однако существующую переменную можно сделать динамической:

```
(def foo 1)

(binding [foo 2]
  foo) ;; Can't dynamically bind non-dynamic var: app.person/foo

(.setDynamic #'foo true)
(.isDynamic #'foo) ;; => true
(binding [foo 2]
  foo) ;; => 2
```

Это может пригодиться как замена `alter-var-root` при распараллеливании тестов.

3.5.4 Sources

Исходники

3.5.5 Ссылки

- Dependency Injection in .Net
- Фаулер о injection

3.6 Persistence

Есть разные способы работы с базой данных.

3.6.1 Active Record

Примеры для ruby и ActiveRecord из Ruby on Rails.

Есть проблема с отслеживанием изменений.

```
user = User.first
user.skills << "codding"
user.save
```

В ruby массивы мутабельны, соответственно ORM не может отследить добавление нового навыка и не сохранит это изменение. Можно конечно сразу после загрузки делать deep copy, и при сохранении сравнивать текущее состояние с изначальным, но не всегда это возможно и приемлемо.

Доступна загрузка ассоциаций по требованию:

```
user.posts
```

Однако вполне возможна рассинхронизация состояния базы данных и программы:

```
user.posts.length #=> 2
Post.create user: user, other_attr: ""
user.posts.length #=> 2
```

Вы можете загрузить одну и ту же сущность в разные объекты:

```
o1 = User.first
o2 = User.find(o1.id)

o1.object_id != o2.object_id
```

Ваши сущности зависят от фреймворка (см. Dependency Inversion Principle)

```
class User < ActiveRecord::Base
  has_many :posts
end
```

Разумеется есть и другие особенности, но нам достаточно приведенных.

В целом, для своей ниши это отличная ORM, но в сложных проектах она начинает откровенно вредить.

3.6.2 Datomic / Datascript

Если бы все наши сущности хранились в одном атоме(world), то можно было бы использовать чисто функциональный подход и обходиться только состояниями сущностей:

```
(swap! world #(-> (update % save-person (build-person {:name "Alice"}))
                  (update % save-person (build-person {:name "Bob"}))
                  (update % delete-last-person)))
```

Очевидно, что загружать все содержимое базы данных в память для любой операции это плохая идея при больших объемах. Проект Datascript - in-memory база, и проектировался для использования в браузере. Datomic использует ленивую загрузку данных.

https://docs.datomic.com/cloud/whatis/data-model.html

3.6.3 Commands & Queries

Наиболее простой механизм. Объявляются функции, которые только извлекают данные и только изменяют данные. Задавая вопрос, не меняй ответ.

```
(defn perform [params]
  ...
  (let [user (queries/get-user-by-id some-id)
        post (post/build params)
        post (assoc post :author-id (:id user))]
    (commands/put-post post)
    ...))
```

Тут уже нет изменяемых объектов, user и post - просто структуры данных вроде map или record. Таким образом бизнес логка не зависит от деталей реализации команд или запросов, а сами они могут быть легко подменены в тестах.

Естественно, не получится ходить про связям `user.posts`.

Вы по прежнему можете отобразить одну сущность в несколько объектов в памяти:

```
(let [user (queries/get-user-by-id 1)
      user (update user :achievements conj :fishing)
      ...
      author (queries/get-user-by-id 1)
      author (update author :achievements conj :writing)]
  (commands/put-user user)
  ...
  (commands/put-user author))
```

В данном примере мы теряем часть изменений, а именно изменения «автора» перетрут изменения пользователя.

Если используются транзакции, и эти транзакции занимают некоторое время, то при большом потоке изменений будут возникать дедлоки и придется вручную расставлять блокировки.

Этот подход хорошо работает в функциональных языках и просто языках без развитой инфраструктуры ORM.

3.6.4 Data Mapper & Identity map & Unit of Work

Мы хотим:

- отделить логику сохранения сущности от бизнес-логики. Data Mapper
- получать тот же объект сущности при повторном извлечении из хранилища. Identity Map
- автоматически отслеживать изменения и сохранять только разницу. Unit of Work

Clojure разделяет неизменяемое состояние и идентичность. Это дает тривиальную реализацию указанных паттернов.

Мы можем моделировать наши сущности используя Record и Ref. Record отвечает за состояние, а Ref - за идентичность.

```
(defrecord User [id login friends])

(let [alice (ref (->User 1 "alice" []))
      bob   (ref (->User 2 "bob" []))]
  (dosync
    (alter alice update :friends conj (:id bob))
    (alter bob   update :friends conj (:id alice))))
```

Можно было бы вместо Ref использовать Atom, но атом поддерживает только нескоординированное изменение. В примере выше установка отношения между Alice и Bob семантически атомарна, поэтому даже если мы в принципе не будем работать с `alice` и `bob` в несколько потоков, оправдано использование Ref, а не Atom.

При этом становится тривиальной реализация Identity map:

```
(storage/with-tx t
  (let [e  (storage/get-one t 1)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        e' (storage/get-one t 1)]
    (identical? e e')))
```

`get-one` принимает транзакцию `t` и идентификатор сущности и возвращает Ref. `t` внутри себя хранит отображение идентификаторов сущностей на идентичность (объекты ref в памяти). При повторном запросе будет возвращен тот же объект(идентичность), что и в первый раз.

Теперь мы хотим создавать и изменять сущности, а так же зафиксировать изменения в хранилище:

```
(storage/with-tx t
  (let [alice (storage/get-one t 1)
        bob   (storage/create (user/->User 2 "bob"))]
    (dosync
     (alter alice update :friends conj (:id bob))
     (alter bob   update :friends conj (:id alice)))))
```

`create` - создает Ref с переданным состоянием и регистрирует ее в Identity Map. Не имеет значения была ли фиксация, `(get-one t 2)` вернет `bob`.

Мы можем использовать 2 стратегии фиксации изменений: оптимистическую и пессимистическую.

В случае оптимистической стратегии мы можем воспользоваться паттерном Единица работы(Unit of Work). Идентичность создается с помощью функций `create` или `get-one`. При этом сохраняется отображение идентификатора на начальное состояние сущности. При первом извлечении с помощью `get-one` мы так же запоминаем версию сущности. При фиксации происходит сравнение начального состояния идентичностей с их текущим состоянием. Открывается транзакция и если версии сущностей не изменились, то происходит фиксация в хранилище. Отмечу, что при таком подходе мы на очень короткое время забираем соединение из пула.

В случае пессимистической стратегии мы можем при каждом изменении идентичности, а для Ref можно задать наблюдателя, производить соответствующие изменения в хранилище. При этом транзакция хранилища длится столько же, сколько и бизнес-транзакция.

Если мы используем оптимистическую стратегию, то в рамках бизнес-транзакции мы не можем делать выборки по произвольному ключу. Допустим, мы выбираем пользователя по его идентификатору . Оказалось, что ему 42 года. Установим его возраст равным 43. В этой же бизнес-транзакции выберем всех пользователей с возрастом равным 42. Очевидно, что перед выборкой необходимо сохранить «грязные» сущности в БД, чтобы запрос вернул то, что мы ожидаем. Но у нас нет открытой транзакции уровня базы данных и мы не можем обеспечить изоляцию.

Мы можем воспользоваться Запросами(Query) и извлекать любые данные(состояние) вне транзакции и перечитать данные находясь в транзакции:

```
(let [moderators-ids (queries/fetch-moderators)]
  (storage/with-tx t
    (let [moderators (storage/get-many moderators-ids)]
      ...)))
```

При этом происходит разделение ответственности. Для запросов мы можем использовать поисковые движки, масштабировать чтение с помощью реплик. А API storage всегда работает с основным хранилищем(мастер). Естественно, что реплики будут содержать устаревшие данные, перечитывание данных в транзакции решает эту проблему.

Также очевидно, что для задач массового изменения данных этот подход не подходит. Для этих редких случаев мы можем использовать Command, которая выполнит необходимый запрос.

Отмечу, что подобным образом можно работать не только с SQL базами данных. Например, Redis также поддерживает транзакции и оптимистические блокировки. Но придется вручную поддерживать

вторичные индексы для произвольных выборок.

Мы не коснулись многих вопросов, и разберем их, когда будем проектировать абстракции.

Глава 4

Core

4.1 Проект

Будем разработать приложение для публикации текстов. Приложение реализует следующие сценарии(use-cases):

- регистрация пользователя
- вход в систему
- выход из системы
- создание поста
- редактирование поста
- отображение поста
- отображение списка всех постов

Информация:

- Исходники
- Demo

Проект структурно разбит на подпроекты. Каждый подпроект - clojure проект с собственными зависимостями, управляемыми с помощью tools.deps.

В текущем разделе мы реализуем подпроект `core`. Он будет содержать всю бизнес-логику нашего приложения. Остальные подпроекты будут зависеть от `core`, например, `web` - доставит приложение пользователю с помощью web технологий, а `persistence` - реализует работу с базой данных.

Реализуется это с помощью абстракций и инверсии засисимости. `core` объявляет абстракцию и примитивную реализацию для отладки и тестирования. Остальные подпроекты содержат промышленную реализацию этих абстракций.

`core` должен быть написан максимально качественно, ведь это и есть ваше приложение. Для остальных подпроектов планку качества можно опустить.

Согласно терминологии Clean Architecture логика приложения разделена между слоями Entities и UseCases. Для удобства разработки `core` содержит оба этих слоя. В зависимости от размера команды, наличия общей предметной области в разных проектах, может быть будет иметь смысл каждый слой оформить в виде подпроекта.

Термины DDD и Clean Architecture имеют пересечение. DDD оперирует понятием Entity - единица предметной области, а в Clean Architecture Entity - основной слой приложения. Поэтому основной слой приложения будет называться `domain` и содержать `entities`, `services`, `value-objects`. Следующий слой будет называться `use-cases`.

Исходники подпроекта

4.2 Domain

4.2.1 Состояние сущностей

Сделаем одну сущность Person в отдельном проекте.

Есть несколько способов моделировать состояние сущности в clojure

1. Использовать мапы:

   ```
   {:id   1
    :type :person
    :name "Alise"}
   ```

   Его минус в том, что реализовать полиморфизм для мап можно только с помощью мультиметодов. Также нам явно нужно указывать тип.

2. Использовать записи:

   ```
   (defrecord Person [id name])
   ```

   При этом мы можем использовать как протоколы, так и мультиметоды. Каждая сущность имеет свой тип(класс). При этом записи поддерживают интерфейс мап. И их объявление является документацией того, какие поля они имеют.

3. Модель datomic. Не рассматриваем.

Мы будем использовать Записи. Для начала напишем тест на конструктор. Конструктор - это обычная функция, возвращающая экземпляр `Person`. Назовем наш конструктор `build`:

```
(ns app.person
  (:require
   [clojure.test :as t]))

(declare build)
(declare person?)

(t/deftest build-test
  (let [params {:name "Alice"}
        person (build params)]
    (t/is (person? person))))
```

Добавим реализацию:

```
(defrecord Person [id name])

(defn build [{:keys [name]}]
  (map->Person {:name name}))

(defn person? [x] (instance? Person x))
```

Отмечу, что наш конструктор не устанавливает идентификатор. И наши сущности получаются неполноценными.

Напишем спецификацию на наш конструктор, чтобы проверять корректность возвращаемого значения.

```
(ns app.person
  (:require
   [clojure.test :as t]
   [clojure.spec.alpha :as s]
   [orchestra.spec.test :as st]))

(s/def ::id pos-int?)
(s/def ::name string?)
(s/def ::person (s/keys :req-un [::id ::name]))

(defrecord Person [id name])

(s/fdef build
        :args (s/cat :params (s/keys :req-un [::name]))
        :ret ::person)

(defn build [{:keys [name]}]
  (map->Person {:name name}))

(defn person? [x] (instance? Person x))

;; подменяем функции на вариант, проверяющий спецификацию
(st/instrument)

(t/deftest build-test
  (let [params {:name "Alice"}
        person (build params)]
    (t/is (person? person))))
```

Ожидаемо наш тест не прошел, т.к. `build` не устанавливает обязательное для персоны(`::person`) поле `id`.

Мы пока не знаем, как мы будем сохранять наши сущности, но уже сейчас нам нужно генерировать идентификаторы. Отложим принятие решения о конкретной реализации генератора и объявим абстракцию генератора:

```
(defprotocol IdGenerator
  (-generate-id [this]))

(declare ^:dynamic *id-generator*)

(s/fdef generate-id
        :ret ::id)

(defn generate-id []
  (-generate-id *id-generator*))
```

Т.е. наш генератор должен реализовывать протокол `IdGenerator` и его экземпляр должен храниться в динамической переменной `*id-generator`.

Теперь мы можем использовать наш генератор в конструкторе:

```
(defn build [{:keys [name]}]
  (map->Person {:id (generate-id)
                :name name}))
```

Для тестов напишем фейковую реализацию, хранящую данные в памяти. Подробнее про фейки, моки и стабы можно посмотреть тут.

```
(deftype FakeIdGenerator [counter]
  IdGenerator

  (-generate-id [_]
    (swap! counter inc)))

(defn build-fake-id-generator []
  (FakeIdGenerator. (atom 0)))
```

Теперь перед каждым тестом нужно создавать экземпляр генератора и устанавливать его в динамическую переменную. Для этого воспользуемся фикстурами:

```
(t/use-fixtures :each (fn [test]
                        (binding [*id-generator* (build-fake-id-generator)]
                          (test))))
```

И теперь тест проходит.

Весь пример полностью.

Очевидно, что мешать весь этот функционал в одном файле - плохая идея. Не пугайтесь, я дам в дальнейшем пример структуры.

Задание

1. Возьмите за основу пример.
2. Реализуйте Пользователя(User) с набором полей: `id`, `login`, `full-name`, `password-digest`, `created-at`. Параметры конструктора: `login`, `full-name`, `password`.
3. Функцию `(defn authenticated? [user password])` для проверки пароля.

Вам понадобится абстрактный `PasswordHasher` для получения `password-digest` и сверки пароля. По аналогии нужно предусмотреть возможность задавать текущее время в тестах.

Проверьте себя:

- absctractions.password-hasher
- fakes.password-hasher
- abstractions.instant

4.2.2 Agregate & Identity

Агрегат

Ранее мы уже знакомились с *Агрегатами*. Теперь поговорим об их реализации.

Возьмем сущности пост и комментарий. В большинстве случаев они образуют агрегат, где корнем будет пост, а внутренними сущностями будут комментарии. Пост будет моделироваться Записью, а комментарии, например вектором хешей. Если комментарии могут быть иерархическими, стоит воспользоваться специализированными структурами вроде datascript.

Агрегат должен иметь идентификатор и проверять свою целостность. Идентификатор может быть глобально уникальным либо уникальным в рамках контекста. Этим контекстом может быть класс базовой сущности. Для удобства будем использовать глобально уникальные идентификаторы. Для проверки целостности будем использовать `clojure.spec`.

Смоделируем это с помощью протокола:

```
(ns app.aggregate
  (:require
   [clojure.spec.alpha :as s]))

(defprotocol Aggregate
  (id [this])
  (spec [this]))

(s/def ::aggregate #(satisfies? Aggregate %))
```

Пост, как корень агрегата, должен реализовать этот протокол:

```
(ns app.post
  (:require
   [app.aggregate :as aggregate]
   [app.post.comment :as comment]
   [clojure.spec.alpha :as s]))

(s/def ::id pos-int?)
(s/def ::title string?)
(s/def ::content string?)
(s/def ::comments (s/coll-of ::comment/comment :kind vector?))
(s/def ::post (s/keys :req-un [::id ::title ::content ::comments]))

(defrecord Post [id title content comments]
  aggregate/Aggregate
  (id [_] id)
  (spec [_] ::post))
```

Комментарии моделируются простыми ассоциативными массивами. Если будет нужно можно будет легко перейти на записи(record).

```
(ns app.post.comment
  (:require
   [clojure.spec.alpha :as s]))

(s/def ::content string?)
(s/def ::author string?)
(s/def ::comment (s/keys :req-un [::content ::author]))
```

Как вы уже заметили, комментарии не хранят идентификатор. Комментарии хранятся в виде вектора, и идентификатором будет индекс комментария в этом векторе:

```
;; map->Post генерируется при объявлении записи Post
(map->Post {:id       1
            :title    "Lorem ipsum"
            :content  "Some text"
            :comments [{:content "Awesome post!"
                        :author  "anonymous"}]})
```

Identity

Мы смоделировали состояние агрегата. Но нам еще нужна идентичность, чтобы работать с изменениями.

Есть 2 альтернативы: атомы и ссылки. Атомы используются для независимого изменения состояния, а ссылки для скоординированного. Вряд ли приложение будет изменять одни и те же сущности из нескольких потоков, однако важно правильно выразить намерение:

```
;; версия с атомами
(swap! alice-acount dec 100)
(swap! bob-acount inc 90)
(swap! bank inc 10)

;; версия с ссылками
(dosync
 (alter alice-acount dec 100)
 (alter bob-acount inc 90)
 (alter bank inc 10))
```

Т.е. мы показываем, что эти изменения часть транзакции, а не сами по себе. Таким образом мы будем использовать ссылки для моделирования идентичности.

Если забыли, то прочитайте параграфы про управление состоянием: *1*, *2*.

Агрегат изменяется как одно целое, поэтому ссылка будет хранить весь агрегат целиком.

Ссылки могут иметь валидаторы. Воспользуемся ими, чтобы проверять изменения:

- нельзя менять идентификатор корня агрегата
- нельзя менять класс корня агрегата (может быть неактуально для некоторых приложений)
- нельзя записывать невалидный агрегат

Задание

В исходниках к этому параграфу есть вышеперечисленные листинги плюс неймспейс для идентичности и падающий тест валидатора. Вам нужно реализовать валидатор.

Ознакомьтесь с:

- `ref`
- `set-validator!`, либо используйте опцию `:validator`
- `class`
- `ex-info`
- `ex-data`
- clojure.spec

Проверьте себя

4.2.3 Services

Сервис - это действие не имеющее состояния, которое моделируется с помощью функции. Сервисы слоя Domain (или Entities в терминологии Clean Architecture) представляют только те действия, которые могут выполняться сотрудниками без компьютеров, например с помощью картотеки. Т.е. сервисы этого слоя не могут делать рассылки, выборки в БД и т.п.

В нашем приложении нет сервисов.

Сервисом будет, например перевод денег:

```
(defn money-transfer [ifrom ito amount]
  (dosync
   (alter ifrom update :account - amount)
   (alter ito update :account + amount)))
```

При этом, в слое Use-cases будет свой сервис перевода денег, использующий `money-transfer` и, например, отправляющий уведомление получателю.

Сервисы могут работать как с идентичностями, как в примере выше, так и с просто состоянием.

4.3 Use cases

4.3.1 Interactor

Интерактор - реализация сценария взаимодействия пользователя с системой. Еще их называют Use Case. Термин интерактор пришел из книги Clean Architecture. В нашем приложении слой, содержащий интеракторы и вспомогательные неймспейсы, будет называться use-cases.

Мы разрабатываем веб-приложение, поэтому взаимодействие пользователя с системой стоится по модели запрос-ответ. Приложения с интерфейсом командной строки или десктопные приложения строятся по этой же модели.

Напомню список сценариев, которые нам нужно реализовать:

- регистрация пользователя
- вход в систему
- выход из системы
- создание поста
- редактирование поста
- отображение поста
- отображение списка всех постов

Интерактор представляет собой набор функций. Как правило, это 3 функции.

authorize - заранее узнать, разрешена ли эта операция или почему запрещена. В случае регистрации пользователя здесь будет проверка того, что пользователь не зашел в систему. При редактировании поста нужно проверить, что пользователь вошел в систему и является автором поста.

initial-data - получить начальные данные формы. Мы должны на основе наших данных сформировать представление, удобное для редактирования пользователем. С помощью функции `process` мы обработаем правки пользователя.

В случае редактирования поста в форму передаются только разрешенные атрибуты. В зависимости от уровня доступа пользователь может не увидеть некоторые атрибуты.

При регистрации можно определить страну, язык, часовой пояс и предзаполнить форму.

process - обработать запрос пользователя.

Не всегда интерактор содержит 3 эти функции, например для отображения списка постов нужен только `process`.

Уже сейчас понятно, что приложение будет хранить пользователей и посты в базе данных, и использовать сессию. Пока мы не знаем детали их реализации и, аналогично предыдущему разделу, спрячем их за абстракциями. Прежде чем переходить к коду интерактора разберемся с ними.

4.3.2 Session

Сессия - сеанс работы с системой. Сохраняет данные между запросами. Наши сценарии используют сессию, чтобы отслеживать вошел ли пользователь в систему.

Мы не собираемся делать сложные выборки по данным в сессии. Для наших сценариев достаточно интерфейса ключ-значение. Мы пока не знаем, как оно будет реализовано, но уже знаем его интерфейс:

```
(ns publicator.use-cases.abstractions.session
  (:refer-clojure :exclude [get set!]))

(defprotocol Session
  (-get [this k])
  (-set! [this k v]))

(declare ^:dynamic *session*)

(defn get [k]
  (-get *session* k))

(defn set! [k v]
  (-set! *session* k v))
```

Для тестирования будем использовать тривиальную реализацию, хранящую состояние в атоме:

```
(ns publicator.use-cases.test.fakes.session
  (:require
   [publicator.use-cases.abstractions.session :as session]))

(deftype FakeSession [storage]
  session/Session
  (-get [_ k] (get @storage k))
  (-set! [_ k v] (swap! storage assoc k v)))

(defn binding-map []
  {#'session/*session* (FakeSession. (atom {}))})
```

Сама сессия дает только низкоуровневый интерфейс. Поэтому сделаем службу для работы с сессией пользователя:

```
(ns publicator.use-cases.services.user-session
  (:require
   [publicator.use-cases.abstractions.session :as session]
   [publicator.use-cases.abstractions.storage :as storage]))

(defn user-id []
  (session/get ::id))

(defn logged-in? []
  (boolean (user-id)))

(defn logged-out? []
  (not (logged-in?)))

(defn log-in! [user]
  (session/set! ::id (:id user)))

(defn log-out! []
  (session/set! ::id nil))

(defn user []
  (when-let [id (user-id)]
    (storage/tx-get-one id)))

(defn iuser [t]
  (when-let [id (user-id)]
    (storage/get-one t id)))
```

Абстракцию `storage` рассмотрим в следующем параграфе.

4.3.3 Storage

Мы уже познакомились с *способами работы с БД* и разобрались *как моделировать идентичности*. Теперь рассмотрим основную абстракцию хранилища подробнее.

Абстрактное хранилище для нашего приложения должно удовлетворять следующим требованиям:

- поддержка транзакций
- получение идентичности по идентификатору
- реализация Identity Map
- создание идентичности из ее состояния

Не во всех приложениях безвозвратное удаление имеет смысл. Чвсто удаление заменяют архивированием. Опустим этот функционал.

Выборки по различным условиям рассмотрим в следующем параграфе.

Основной код абстракции:

```
(ns publicator.use-cases.abstractions.storage
  (:require
   [clojure.spec.alpha :as s]
   [publicator.domain.abstractions.id-generator :as id-generator]
   [publicator.domain.abstractions.aggregate :as aggregate]
   [publicator.domain.identity :as identity]
   [publicator.utils.ext :as ext]))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(defprotocol Storage
  (-wrap-tx [this body]))

(defprotocol Transaction
  (-get-many [t ids])
  (-create [t state]))

(s/fdef get-many
  :args (s/cat :tx any?
               :ids (s/coll-of ::id-generator/id :distinct true))
  :ret (s/map-of ::id-generator/id ::identity/identity))

(s/fdef create
  :args (s/cat :tx any?
               :state ::aggregate/aggregate)
  :ret ::identity/identity)

(defn get-many [t ids] (-get-many t ids))
(defn create   [t state] (-create t state))

(declare ^:dynamic *storage*)

(defmacro with-tx
  "Note that body forms may be called multiple times,
   and thus should be free of side effects."
  [tx-name & body-forms-free-of-side-effects]
  `(-wrap-tx *storage*
             (fn [~tx-name]
               ~@body-forms-free-of-side-effects)))
```

Можно попытаться написать спецификацию для `get-many`, которая будет проверять поддержку Identity Map, но эта спецификация будет очень сложной, поэтому проверка ложится на программиста и тесты.

С помощью макроса `with-tx` мы можем удобно объявлять транзакцию:

```
(storage/with-tx t
  (storage/create t user-1-state)
  (storage/create t user-2-state))
```

Для оптимизации запросов, протокол транзакции поддерживает только метод `get-many`, а метод `get-one` выражается через него :

```
(s/fdef get-one
  :args (s/cat :tx any?
               :id ::id-generator/id)
  :ret (s/nilable ::identity/identity))

(defn get-one [t id]
  (let [res (get-many t [id])]
    (get res id)))
```

Часто мы будем выполнять только одно действие в транзакции, для этого объявим вспомогательные методы:

```
(s/fdef tx-get-one
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  :args (s/cat :id ::id-generator/id)
  :ret (s/nilable ::aggregate/aggregate))

(defn tx-get-one [id]
  (with-tx t
    (when-let [x (get-one t id)]
      @x)))


(s/fdef tx-get-many
  :args (s/cat :ids (s/coll-of ::id-generator/id :distinct true))
  :ret (s/map-of ::id-generator/id ::aggregate/aggregate))

(defn tx-get-many [ids]
  (with-tx t
    (->> ids
         (get-many t)
         (ext/map-vals deref))))


(s/fdef tx-create
  :args (s/cat :state ::aggregate/aggregate)
  :ret ::aggregate/aggregate
  :fn #(= (-> % :args :state)
          (-> % :ret)))

(defn tx-create [state]
  (with-tx t
    @(create t state)))


(s/fdef tx-alter
  :args (s/cat :state ::aggregate/aggregate
               :f fn?
               :args (s/* any?))
  :ret (s/nilable ::aggregate/aggregate))

(defn tx-alter [state f & args]
  (with-tx t
    (when-let [x (get-one t (aggregate/id state))]
      (dosync
       (apply alter x f args)))))
```

Что бы лучше понять, как это использовать, разберитесь в тестах фейковой реализации этой абстракции:

```
(ns publicator.use-cases.test.fakes.storage-test
  (:require
   [publicator.use-cases.test.fakes.storage :as sut]
   [publicator.use-cases.abstractions.storage :as storage]
   [publicator.domain.abstractions.aggregate :as aggregate]
   [publicator.domain.identity :as identity]
   [publicator.utils.test.instrument :as instrument]
   [clojure.test :as t]))

(t/use-fixtures :once instrument/fixture)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(t/use-fixtures
  :each
  (fn [f]
    (with-bindings (sut/binding-map (sut/build-db))
      (f))))

(defrecord Test [counter]
  aggregate/Aggregate
  (id [_] 42)
  (spec [_] any?))

(t/deftest create
  (let [test (storage/tx-create (->Test 0))
        id (aggregate/id test)]
    (t/is (some? test))
    (t/is (some? (storage/tx-get-one id)))))

(t/deftest change
  (let [test (storage/tx-create (->Test 0))
        id   (aggregate/id test)
        _    (storage/tx-alter test update :counter inc)
        test (storage/tx-get-one id)]
    (t/is (= 1 (:counter test)))))

(t/deftest identity-map-persisted
  (let [test (storage/tx-create (->Test 0))
        id   (aggregate/id test)]
    (storage/with-tx t
      (let [x (storage/get-one t id)
            y (storage/get-one t id)]
        (t/is (identical? x y))))))

(t/deftest identity-map-in-memory
  (storage/with-tx t
    (let [x (storage/create t (->Test 0))
          y (storage/get-one t (aggregate/id @x))]
      (t/is (identical? x y)))))

(t/deftest identity-map-swap
  (storage/with-tx t
    (let [x (storage/create t (->Test 0))
          y (storage/get-one t (aggregate/id @x))
          _ (dosync (alter x update :counter inc))]
      (t/is (= 1 (:counter @x) (:counter @y))))))

(t/deftest concurrency
  (let [test (storage/tx-create (->Test 0))
        id   (aggregate/id test)
        n    10
        _    (->> (repeatedly #(future (storage/tx-alter test update :counter inc)))
                  (take n)
                  (doall)
                  (map deref)
                  (doall))
        test (storage/tx-get-one id)]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    (t/is (= n (:counter test)))))

(t/deftest inner-concurrency
  (let [test (storage/tx-create (->Test 0))
        id   (aggregate/id test)
        n    10
        _    (storage/with-tx t
               (->> (repeatedly #(future (as-> id <>
                                           (storage/get-one t <>)
                                           (dosync (alter <> update :counter inc)))))
                    (take n)
                    (doall)
                    (map deref)
                    (doall)))
        test (storage/tx-get-one id)]
    (t/is (= n (:counter test)))))
```

Наконец, сама фейковая реализация:

```
(ns publicator.use-cases.test.fakes.storage
  (:require
   [publicator.domain.identity :as identity]
   [publicator.domain.abstractions.aggregate :as aggregate]
   [publicator.use-cases.abstractions.storage :as storage]
   [publicator.utils.ext :as ext]))

(deftype Transaction [data identity-map]
  storage/Transaction
  (-get-many [_ ids]
    (let [ids-for-select (remove #(contains? @identity-map %) ids)
          selected       (->> ids-for-select
                              (select-keys data)
                              (ext/map-vals identity/build))]
      ;; Здесь принципиально использование reverse-merge,
      ;; т.к. другой поток может успеть извлечь данные из базы,
      ;; создать объект-идентичность, записать его в identity map
      ;; и сделать в нем изменения.
      ;; Если использовать merge, то этот поток затрет идентичность
      ;; другим объектом-идентичностью с начальным состоянием.
      ;; Фактически это нарушает саму идею identity-map -
      ;; сопоставление ссылки на объект с его идентификатором
      (-> identity-map
          (swap! ext/reverse-merge selected)
          (select-keys ids))))

  (-create [_ state]
    (let [id     (aggregate/id state)
          istate (identity/build state)]
      (swap! identity-map (fn [map]
                            {:pre [(not (contains? map id))]}
                            (assoc map id istate)))
      istate)))

(deftype Storage [db]
  storage/Storage
  (-wrap-tx [_ body]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    (loop []
      (let [data         @db
            identity-map (atom {})
            t            (Transaction. data identity-map)
            res          (body t)
            changed      (ext/map-vals deref @identity-map)
            new-data     (merge data changed)]
        (if (compare-and-set! db data new-data)
          res
          (recur))))))

(defn build-db []
  (atom {}))

(defn binding-map [db]
  {#'storage/*storage* (->Storage db)})
```

Эта фейковая реализация хранит все данные в атоме `db`. Этот атом содержит отображение идентификаторов на состояние сущностей:

```
{1 (->User 1 ...)
 2 (->Post 2 ...)
 3 (->Post 3 ...)}
```

`identity-map` - тоже атом, но содержит отображение идентификаторов на идентичности сущностей:

```
{1 (ref (->User 1 ...))
 2 (ref (->Post 2 ...))}
```

При этом `identity-map` будет содержать не все сущности, что есть в `db`, а только те, которые участвуют в транзакции.

`-wrap-tx` в бесконечном цикле пытается выполнить транзакцию. Если с начала транзакции никто не успел поменять данные, то транзакция проходит. Тут используется оптимистическая блокировка. А помогает в этом низкоуровневая атомарная операция атома `compare-and-set!`.

4.3.4 Queries

Абстракция Storage не позволяет делать произвольные выборки, а выбирает весь агрегат целиком по его id.

Часто этого недостаточно. Например, нужно найти пользователя по его логину, т.е. выбрать весь агрегат по условию. Или выбрать таблицу постов с именами их авторов, т.е. выбрать агрегаты с дополнительными полями. Или собрать аналитику и вернуть простые структуры данных.

Т.е. запрос выбирает данные. Эти данные могут быть в том числе состоянием агрегата (записью).

Рассмотрим абстракцию запросов постов. В этом неймспейсе описаны 2 запроса: Получить список постов и получить пост по id.

```
(ns publicator.use-cases.abstractions.post-queries
  (:require
   [publicator.domain.aggregates.user :as user]
   [publicator.domain.aggregates.post :as post]
   [clojure.spec.alpha :as s]))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(defprotocol GetList
  (-get-list [this]))

(declare ^:dynamic *get-list*)

(s/def ::post (s/merge ::post/post
                       (s/keys :req [::user/id ::user/full-name])))

(s/fdef get-list
        :ret (s/coll-of ::post))

(defn get-list []
  (-get-list *get-list*))


(defprotocol GetById
  (-get-by-id [this id]))

(declare ^:dynamic *get-by-id*)

(s/fdef get-by-id
  :args (s/cat :id ::post/id)
  :ret (s/nilable ::post))

(defn get-by-id [id]
  (-get-by-id *get-by-id* id))
```

На самом деле тут выбирается не совсем пост. Этот неймспейс объявляет свой «тип» `::post`, который кроме атрибутов поста содержит еще и некоторые атрибуты своего автора. Это работает благодаря тому, что записи в clojure открыты к добавлению новых атрибутов.

Чтобы избежать конфликтов имен, атрибуты пользователя будут содержать неймспейс. Это видно по объявлению спецификаций. Обратите внимание на `req-un` и `req`:

```
(ns publicator.domain.aggregates.post
  ...)
(s/def ::post (s/keys :req-un [::id ::title ::content ::created-at]))

(ns publicator.use-cases.abstractions.post-queries
  (:require
   [publicator.domain.aggregates.post :as post]
   ...))
(s/def ::post (s/merge ::post/post
                       (s/keys :req [::user/id ::user/full-name])))
```

Теперь разберем фейковую реализацию. В предыдущем параграфе я показывал, что данные хранятся в атоме, содержащем следующую структуру:

```
{1 (->User 1 ...)
 2 (->Post 2 ...)
 3 (->Post 3 ...)}
```

Мы можем сделать так, чтобы фейк делал выборки из этого атома. В нем нет индексов, кроме первичного ключа, и большинство выборок будут выполняться полным сканированием, но это не критично для использования в тестах. Если в вашем приложени ожидаются сложные выборки, то можно использовать специальные структуры для хранения данных вроде datascript.

```
(ns publicator.use-cases.test.fakes.post-queries
  (:require
   [publicator.use-cases.abstractions.post-queries :as post-q]
   [publicator.domain.aggregates.post :as post]
   [publicator.domain.aggregates.user :as user]
   [publicator.domain.services.user-posts :as user-posts]))

(defn- author-for-post [db post]
  (->> @db
       (vals)
       (filter user/user?)
       (filter #(user-posts/author? % post))
       (first)))

(defn- assoc-user-fields [post user]
  (assoc post
         ::user/id (:id user)
         ::user/full-name (:full-name user)))

(deftype GetList [db]
  post-q/GetList
  (-get-list [_]
    (->> @db
         (vals)
         (filter post/post?)
         (map #(when-some [author (author-for-post db %)]
                 (assoc-user-fields % author)))
         (remove nil?))))

(deftype GetById [db]
  post-q/GetById
  (-get-by-id [_ id]
    (when-some [post (get @db id)]
      (when-some [author (author-for-post db post)]
        (assoc-user-fields post author)))))

(defn binding-map [db]
  {#'post-q/*get-list* (->GetList db)
   #'post-q/*get-by-id* (->GetById db)})
```

Самостоятельно ознакомьтесь с выборками пользователей:

- abstractions.user-queries
- fakes.user-queries

4.3.5 Реализация интеракторов

Рассмотрим 2 интерактора. Остальные рассмотрите самостоятельно.

Отображение поста

Помимо аттрибутов поста ответ должен содержать идентификатор и имя автора. *Ранее* мы рассматривали устройство `abstractions.post-queries`.

Интерактор содержит только метод `process`, т.к. нам не нужна форма и все пользователи системы могут смотреть все посты. Результатом может быть или успех или неудача из-за того, что поста нет в

хранилище.

```
(ns publicator.use-cases.interactors.post.show
  (:require
   [publicator.use-cases.services.user-session :as user-session]
   [publicator.use-cases.abstractions.post-queries :as post-q]
   [publicator.domain.aggregates.post :as post]
   [darkleaf.either :as e]
   [clojure.spec.alpha :as s]))

(defn- get-by-id= [id]
  (if-let [post (post-q/get-by-id id)]
    (e/right post)
    (e/left [::not-found])))

(defn process [id]
  (e/extract
   (e/let= [user (user-session/user)
            post (get-by-id= id)]
     [::processed post])))

(s/def ::not-found (s/tuple #{::not-found}))
(s/def ::processed (s/tuple #{::processed} ::post-q/post))

(s/fdef process
  :args (s/cat :id ::post/id)
  :ret (s/or :ok  ::processed
             :err ::not-found))
```

Для моделирования вычислений, могут окончиться неудачей воспользуемся монадой either, которую мы реализовывали *ранее*.

Спецификация `process` описывает все возможные ответы.

Тест:

```
(ns publicator.use-cases.interactors.post.show-test
  (:require
   [publicator.use-cases.interactors.post.show :as sut]
   [publicator.use-cases.test.fakes :as fakes]
   [publicator.utils.test.instrument :as instrument]
   [publicator.use-cases.test.factories :as factories]
   [clojure.test :as t]))

(t/use-fixtures :each fakes/fixture)
(t/use-fixtures :once instrument/fixture)

(t/deftest process
  (let [post       (factories/create-post)
        post-id    (:id post)
        user       (factories/create-user {:posts-ids #{post-id}})
        [tag post] (sut/process (:id post))]
    (t/is (= ::sut/processed tag))
    (t/is (some? post))))
```

Желательно, чтобы тесты покрывали все возможные ответы. Вы даже можете на основе спецификаций автоматически проверять наличие тестов для каждого типа ответа, но не будем отвлекаться.

Редактирование поста

Прежде всего мы должны проверить, что пользователь залогинен и является автором этого поста. Затем мы проверяем новые атрибуты поста, при этом мы не должны записать лишние поля, которые может передать злоумышленник. Далее мы устанавливаем измененные атрибуты.

```
(ns publicator.use-cases.interactors.post.update
  (:require
   [publicator.domain.aggregates.post :as post]
   [publicator.domain.identity :as identity]
   [publicator.use-cases.services.user-session :as user-session]
   [publicator.use-cases.abstractions.storage :as storage]
   [publicator.utils.spec :as utils.spec]
   [darkleaf.either :as e]
   [clojure.spec.alpha :as s]))

(s/def ::params (utils.spec/only-keys :req-un [::post/title ::post/content]))

(defn- check-authorization= [t id]
  (let [iuser (user-session/iuser t)]
    (cond
      (nil? iuser)                              (e/left [::logged-out])
      (not (contains? (:posts-ids @iuser) id)) (e/left [::not-authorized])
      :else                                     (e/right [::authorized]))))

(defn- find-post= [t id]
  (if-some [ipost (storage/get-one t id)]
    (e/right ipost)
    (e/left [::not-found])))

(defn- check-params= [params]
  (if-some [ed (s/explain-data ::params params)]
    (e/left [::invalid-params ed])))

(defn- update-post [ipost params]
  (dosync (alter ipost merge params)))

(defn- post->params [post]
  (select-keys post [:title :content]))

(defn initial-params [id]
  (storage/with-tx t
    (e/extract
     (e/let= [ok     (check-authorization= t id)
              ipost  (find-post= t id)
              params (post->params @ipost)]
       [::initial-params @ipost params]))))

(defn process [id params]
  (storage/with-tx t
    (e/extract
     (e/let= [ok    (check-authorization= t id)
              ok    (check-params= params)
              ipost (find-post= t id)]
       (update-post ipost params)
       [::processed @ipost]))))

(defn authorize [ids]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  (storage/with-tx t
    (->> ids
         (map #(check-authorization= t %))
         (map e/extract))))

(s/def ::logged-out (s/tuple #{::logged-out}))
(s/def ::invalid-params (s/tuple #{::invalid-params} map?))
(s/def ::not-found (s/tuple #{::not-found}))
(s/def ::not-authorized (s/tuple #{::not-authorized}))
(s/def ::initial-params (s/tuple #{::initial-params} ::post/post map?))
(s/def ::processed (s/tuple #{::processed} ::post/post))
(s/def ::authorized (s/tuple #{::authorized}))

(s/fdef initial-params
  :args (s/cat :id ::post/id)
  :ret (s/or :ok  ::initial-params
             :err ::logged-out
             :err ::not-authorized
             :err ::not-found))

(s/fdef process
  :args (s/cat :id ::post/id
               :params any?)
  :ret (s/or :ok  ::processed
             :err ::logged-out
             :err ::not-authorized
             :err ::not-found
             :err ::invalid-params))

(s/fdef authorize
  :args (s/cat :ids (s/coll-of ::post/id))
  :ret (s/coll-of (s/or :ok  ::authorized
                        :err ::logged-out
                        :err ::not-found
                        :err ::not-authorized)))
```

`clojure.spec` из коробки не поддерживает строгую валидацию ключей, поэтому воспользуемся собственным макросом `utils.spec/only-keys`.

Пост - множественный ресурс, в отличие от, например, регистрации. Скажем, при отображении списка постов нужно показать пользователю, какие посты он может редактировать, а какие нет. По этой причине `authorize` должен принимать множество идентификаторов, чтобы избежать проблемы N+1.

В нашем случае `check-authorization=` оперирует только идентификатором поста и не нужно выбирать из хранилища все посты для переданных `ids` в `authorize`. Но если бы нам нужно было быть в `check-authorization=` использовать сам пост, то можно воспользоваться identity-map:

```
(defn- check-authorization= [t id]
  (let [iuser (user-session/iuser t)
        ipost (storage/get-one t id)] ;; <1>
    (some-logic iuser ipost)))

(defn authorize [ids]
  (storage/with-tx t
    (storage/preload ids) ;; <2>
    (->> ids
         (map #(check-authorization= t %))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        (map e/extract))))
```

Как видно, `check-authorization=` принимает объект транзакции `t`, который хранит кэш выбранных сущностей в рамках этой транзакции. Поэтому в `<1>` будет выборка из кэша, т.к. в `<2>` мы предварительно загрузили все сущности одним запросом.

Задание

Самостоятельно посмотрите оставшиеся интеракторы и их тесты. Тесты покрывают не все случаи, допишите их.

4.3.6 Итог

Мы познакомились с интеракторами и их абстракциями.

Самостоятельно ознакомьтесь с оставшимися интеракторами и их тестами:

- интеракторы
- тесты

В качестве практики вы можете реализовать архивирование постов.

4.4 Итог

Мы проанализировали и записали в виде кода предметную область приложения. Мы не делали ничего лишнего, только реализовали и протестировали бизнес-логику.

В реальном проекте, в процессе написания кода появилось бы много уточняющих вопросов заказчику. Команда и заказчик стали лучше понимать предметную область. Заказчик внес правки, увеличивающие бизнес-ценность. Переделки дешевы, т.к. затрагивают только логику.

Команда поняла как эффективно хранить данные приложения.

Тесты бизнес-логики занимают доли секунды, т.к. все данные находятся в памяти и не используются сторонние сервисы.

Глава 5

Web

5.1 Введение

В этой главе рассмотрим доставку приложения пользователю с помощью веб.

Исходники подпроекта web.

5.2 Ring

Ring - самая распространенная абстракция web сервера в clojure. Вдохновлен ruby rack, python wsgi.

Обработчик запроса - простая функция, принимающая запрос, и возвращающая ответ. И запрос и ответ - простые clojure структуры:

```
(defn handler [req]
  {:status 200
   :headers {}
   :body (:uri req)})
```

В этом примере всегда отвечаем 200, а в теле ответа будет Uri.

Для добавления различного функционала мы можем использовать функции обертки - middleware:

```
(def app (-> handler
             (wrap-content-type "text/html")
             (wrap-keyword-params)
             (wrap-params)))
```

Middleware устроена следующим образом:

```
(defn wrap-example [handler some-args]
  (fn [req]
    (let [req  (change-req req]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
          resp (handler req)
          resp (change-resp resp)]
      resp)))
```

T.e. middleware должна принять обработчик, некоторые параметры и вернуть новый обработчик.

Подробнее про структуру запросов и ответов, обработчики и middleware можно почитать в wiki проекта.

5.2.1 Server

Ring - абстракция над веб-сервером, а их может быть много, например:

- ring-jetty-adapter - адаптер для java сервера jetty
- http-kit
- aleph

Мы будем использовать jetty. Для этого нужно подключить 2 зависимости:

- `ring/ring-core {:mvn/version "1.6.2"}`
- `ring/ring-jetty-adapter {:mvn/version "1.6.2"}`

```
(ns app.app
  (:require
   [clojure.pprint :as pp]
   [ring.adapter.jetty :as jetty]))

(defn handler [req]
  {:status 200
   :headers {"Content-Type" "text/plain"}
   :body (with-out-str (pp/pprint req))})

(jetty/run-jetty handler {:port 4445})
```

Этот пример запускает http сервер на 4445 порту. В ответ на любой запрос тело ответа будет содержать красиво распечатанный запрос.

5.2.2 Другие проекты

Есть и другие, несовместимые с ring, проекты:

- pedestal
- catacumba

5.3 Управление stateful компонентами

Если не использовать интерактивную разработку в repl и на каждое изменение кода перезапускать jvm, то проблем не возникает, т.к. все ресурсы освобождаются при завершении jvm процесса.

Теперь мы хотим перезагружать наши неймспейсы не перезапуская jvm. Для этого воспользуемся tools.namespace.

Допустим, у нас есть неймспейс:

```
(ns app.app
  (:require
   [clojure.pprint :as pp]
   [ring.adapter.jetty :as jetty]))

(defn handler [req]
  {:status 200
   :headers {"Content-Type" "text/plain"}
   :body (with-out-str (pp/pprint req))})

(jetty/run-jetty handler {:port 4445})
```

При перезагрузке он очищается, и его код выполняется заново. При этом будет неудачная попытка запуска сервера с новым обработчиком, т.к. старый сервер не остановлен и занимает порт.

Для stateless кода перезагрузка работает тривиально, а для stateful кода нужно освобождать ресурсы. Поэтому stateful код изолируют с помощью компонентов.

Компонент - нечто, что можно запустить и остановить. Компоненты зависят от других компонентов и задача фреймворка в правильном порядке запускать и останавливать компоненты.

В clojure есть 2 популярных проекта для управления stateful компонентами.

- component
- mount

Component внутренне проще, функциональнее, но несколько сложнее в использовании, а mount - наоборот.

Наше приложение состоит из ядра и различных плагинов. Например, приложение может работать с фейковым хранилищем в памяти и персистентным хранилищем в postgresql. В случае с component становится тривиальным запуск 2-х копий приложения на разных портах с разными хранилищами, в случае с mount - это не так.

Фреймворк компонентов - вспомогательная библиотека для нашего приложения. Ни логика, ни реализации абстракций не зависят от этого фреймворка. Поэтому для своего приложения вы можете выбрать любую библиотеку или написать свою. Сейчас же я буду использовать component.

5.3.1 Component

Component использует записи, реализующие протокол `Lifecycle` с 2 методами: `start` и `stop`. Используются записи, а не типы, т.к. зависимости компонента устанавливаются через `assoc`. Таким образом компоненты объединяются в Систему. Таким образом можно запускать и останавливать компоненты в порядке их зависимости друг от друга.

Так же Component позиционируется как фреймворк для внедрения зависимостей. Но в этом случае наше приложение жестко зависит от этой библиотеки, что не приемлемо для нас. К тому же мы уже используем dynamic var для внедрения зависимостей.

Подробнее узнать про компонент:

- https://github.com/stuartsierra/component
- https://www.youtube.com/watch?v=13cmHf_kt-Q
- https://github.com/matthiasn/talk-transcripts/blob/master/Sierra_Stuart/Components.md

С component наше приложение будет выглядеть так:

```
(ns app.app
  (:require
   [clojure.pprint :as pp]
   [com.stuartsierra.component :as component]
   [ring.adapter.jetty :as jetty]))

(defn handler [req]
  {:status 200
   :headers {"Content-Type" "text/plain"}
   :body (with-out-str (pp/pprint req))})

(defrecord Jetty [val]
  component/Lifecycle
  (start [this]
    (if val
      this
      (assoc this :val
             (jetty/run-jetty #'handler
                              {:port 4445
                               :join? false}))))
  (stop [this]
    (if-not val
      this
      (do
        (.stop val)
        (assoc this :val nil)))))

(defn build-jetty []
  (->Jetty nil))
```

5.3.2 Перезагрузка

Перед перезагрузкой нужно остановить приложение, а после - запустить. Подробнее про этот механизм можно прочитать в readme tools.namespace.

Плюс приложение нужно как-то запустить в начале работы.

Когда мы открываем repl, то начинаем в пространстве имен `user`. Удобно добавить в него функции `start` и `stop`. Добавим файл `dev/user.clj`, и добавим директорию `dev` в пути поиска неймспейсов:

```
;; dev/user.clj
(ns user
  (:require
   [com.stuartsierra.component :as component]
   [app.app :as app]))

(def component (app/build-jetty))

(defn start []
  (alter-var-root #'component component/start))

(defn stop []
  (alter-var-root #'component component/stop))
```

```
;; deps.edn
{:deps {ring/ring-core             {:mvn/version "1.6.2"}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
       ring/ring-jetty-adapter    {:mvn/version "1.6.2"}
       com.stuartsierra/component {:mvn/version "0.3.2"}}

 :paths ["dev" "src"]} ;; добавляем директорию dev
```

Наш компонент хранится в переменной `component`. Она инициализируется при загрузке неймспейса незапущенным компонентом. А функции `start` и `stop` заменяют значение этой переменной.

Для разработки мы можем использовать либо просто repl, либо Emacs + cider:

- clojure -Arepl
- clojure -Acider

И нужно как-то указать им какие функции вызывать перед и после перезагрузки.

Для Emacs воспользуемся файлом `.dir-locals.el` в корне директории с примерами:

```
((nil
  (eval .
        (setq cider-refresh-before-fn "user/stop"
              cider-refresh-after-fn "user/start"))))
```

А в repl можно передать их как аргумент. deps.edn:

```
{:aliases {:repl  {:extra-deps {darkleaf/repl-tools-deps
                                {:git/url "https://github.com/darkleaf/repl-tools-deps.git"
                                 :sha     "04e128ca67785e4eb7ccaecfdaffa3054442358c"}}
                   :main-opts  ["-m" "darkleaf.repl-tools-deps"
                                "reload-before-fn" "user/stop"
                                "reload-after-fn" "user/start"]}}}
```

Теперь в Emacs можно делать перезагрузку с помощью `C-c C-x`, а в repl - `:repl/reload`.

5.3.3 Перезагрузка без потери состояния

Важно отметить, что перезагрузка убирает любое состояние. Например, мы запустили приложение с веб-сервером и фейковым хранилищем. Внесли какие-то данные, например создали пост, внесли правки в код, перезагрузили. А т.к. данные хранятся в памяти, то наш пост будет утерян.

Веб-сервер запускается следующим образом: `(jetty/run-jetty handler {:port 4445})`. Т.е. мы передаем обработчик. Но в этом случае мы передаем значение обработчика, а не переменную, содержащую обработчик. Если мы зададим новое значение этой переменной, веб-сервер будет использовать старое значение.

Благодаря тому, что сама переменная поддерживает интерфейс функции, можно передавать саму переменную: `(jetty/run-jetty #'handler {:port 4445})`. В этом случае при каждом запросе будет использовать обработчик, содержащийся в переменной.

Это дает возможность менять код, видеть изменения, но не перезагружать все приложение.

5.4 Routing

Роутинг сопоставляет определенный запрос с обработчиком этого запроса.

Для Ring есть множество библиотек. Из популярных можно отметить:

- compojure
- liberator
- yada
- bidi

Какую выберете вы, не очень принципиально. Однако, удобно, когда библиотека поддерживает обратный роутинг, т.е. зная имя роута можно получить url запроса. Иначе придется хардкодить url строками, и легко оставить битую ссылку.

Воспользуемся библиотекой sibiro. Это простая библиотека, не имеющая лишних для нашего приложения функций.

```
(def routes
  #{[:get "/" #'root :root]
    [:get "/:page" #'page :page]})
```

Каждый маршрут моделируется вектором со следующими элементами:

1. http метод
2. шаблон url, с поддержкой параметров и регулярных выражений
3. функция-обработчик
4. название маршрута для обратного роутинга

Маршруты объявляются как множество, т.е. маршруты не имеют порядка. Нет вложенности, невозможно задать middleware для группы роутов, например для `/admin`. Подробнее ознакомьтесь с readme библиотеки.

Напомню, что ранее мы обсуждали перезагрузку кода. Если бы мы не использовали `#'root`, а просто передавали значение `root`, то при изменении кода пришлось бы перезагружать все приложение целиком.

Для примера напишем сайт из нескольких страниц. Главная страница содержит заголовок и список ссылок на прочие страницы. Страницы содержат заголовок и ссылаются на главную.

```
(ns app.app
  (:require
   [clojure.string :as str]
   [ring.util.response :as ring.response]
   [sibiro.core]
   [sibiro.extras]))

(declare routes)

(defn path-for [& args]
  (let [ret (apply sibiro.core/path-for routes args)]
    (assert (some? ret) (str "route not found for " args))
    ret))

(defn page-link [slug]
  (let [url (path-for :page {:page slug})]
    (str "<div><a href=\"" url "\">" slug "</a></div>")))

(defn root-link []
  (let [url (path-for :root)]
    (str "<div><a href=\"" url "\">root</a></div>")))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(defn root [req]
  (let [body (str "<h1>Root page</h1>"
                  (->> ["about" "contacts" "resources"]
                       (map page-link)
                       str/join))]
    (-> (ring.response/response body)
        (ring.response/header "Content-Type" "text/html"))))

(defn page [req]
  (let [slug (-> req :route-params :page)
        body (str
              "<h1>" slug "</h1>"
              (root-link))]
    (-> (ring.response/response body)
        (ring.response/header "Content-Type" "text/html"))))

(def routes
  (sibiro.core/compile-routes
   #{[:get "/" #'root :root]
     [:get "/:page" #'page :page]}))

(def handler (sibiro.extras/make-handler routes))
```

Библиотечная функция `sibiro.core/path-for` возвращает `nil`, если не находит подходящий маршрут, что приводит к коварным ошибкам. Будем использовать обертку `path-for`, которая бросит исключение, если маршрут не найден.

Запустите этот пример, добавьте парочку маршрутов. Исходники примера.

5.5 HTTP методы и HTML

HTTP поддерживает множество методов:

- get используется только для получения данных, может быть закэширован
- post - для создания новой сущности
- patch - для частичного обновления сущности
- put - для создания или полной замены сущности, upsert
- delete - для удаления
- и т.д.

Однако HTML умеет работать только с get и post. Get используется при переходе по ссылкам, а post для отправки форм.

Если вы хотите создать кнопку, отправляющую некоторую команду серверу, то придется реализовать ее в виде html формы, т.к. команда будет отправлена с помощью post и не будет закэширована.

При проектировании маршрутов не стоит ограничиваться только get и post, используйте семантику протокола. В этом поможет js. В экосистеме Ruby on Rails есть проект rails-ujs. И он доступен отдельно от rails в виде npm пакета. Благодаря unpkg.com его очень просто добавить на страницу:

```
<!-- content -->
    <script src="https://unpkg.com/rails-ujs@5.2.0/lib/assets/compiled/rails-ujs.js"></script>
  </body>
</html>
```

rails-ujs сканирует документ на предмет ссылок с data-атрибутом method:

```
<a class="nav-item nav-link" data-method="delete" href="/some-url">
  some text
</a>
```

При клике на такую ссылку rails-ujs создает невидимую форму и отправляет ее post запросом. Метод устанавливается в поле формы `_method`. Если вы используете отличные от get и post методы, то приложение должно отслеживать параметр `_method` и подменять метод в запросе.

5.6 Сессия

Ранее мы рассматривали *абстракцию сессии* теперь займемся её реализацией.

Ring добавляет поддержку http сессии с помощью middleware `ring.middleware.session/wrap-session`. В запросе появляется ключ `:session`, который и хранит данные сессии. По умолчанию сессия хранится в памяти процесса, есть возможность хранить ее в cookie или написать свою реализацию.

Воспользуемся http сессией и реализуем нашу абстракцию:

```
(ns publicator.web.middlewares.session
  (:require
   [publicator.use-cases.abstractions.session :as session]))

(deftype Session [storage]
  session/Session
  (-get [_ k] (get @storage k))
  (-set! [_ k v] (swap! storage assoc k v)))

(defn wrap-session [handler]
  (fn [req]
    (let [storage (atom (get-in req [:session ::storage]))
          resp    (binding [session/*session* (Session. storage)]
                    (handler req))]
      (-> resp
          (assoc :session/key (:session/key req))
          (assoc :session (:session req))
          (assoc-in [:session ::storage] @storage)))))
```

5.7 Адаптер

Весь подпроект web является адаптером между слоем сценариев и веб-сервером. Это можно показать так:

```
(defn handler [req]
  (let [args   (req->args req)
        result (apply interactor args)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        resp   (result->resp result)]
    resp))
```

Т.е. обработчик имеет 2 обязанности:

- преобразование ring запроса в аргументы интерактора, показана как функция `req->args`
- преобразование результата интерактора к ring ответу, показана как функция `result->resp`

Разделим обработчик на составные части. Назовем `req->args` контроллером, а `result->resp` респондером.

```
(defn controller [req]
  (let [arg1 (req->arg1 req)
        arg2 (req->arg2 req)]
    [interactor arg1 arg2])) ;; <1>

(defn responder [result [arg1 arg2]]
  (let [viewmodel (presenter result arg1 arg2)
        html      (template viewmodel)]
    {:status  200
     :headers {}
     :body    html}))

(defn middleware [handler]
  (fn [req]
    (let [[interactor & args] (handler req)
          result              (apply interactor args)]
      (responder result))))

(def handler
  (-> controller
      middleware))
```

Теперь у нас 4 слабосвязанных компонента.

Middleware и handler существуют в единственном экземпляре, тривиальны и не требуют модульного тестирования.

Контроллеры и респондеры не зависят друг от друга. *Вспомним* диаграмму связи интерактора, контроллера и презентера. В этом случае респондер - это презентер в терминах диаграммы.

Обратите внимание на <1>, контроллер не вызывает интерактор, а просто возвращает функцию и ее аргументы как данные. Таким образом при тестировании не нужно подменять интерактор, чтобы проверить корректность формирования его аргументов.

Все это делает модульное тестирование очень простым.

В следующих параграфах мы разберем эти компоненты подробнее.

5.8 Controller

Контроллер - адаптер для интерактора, который конвертирует ring запрос в данные, понятные интерактору. Также контроллер содержит объявление маршрутов.

Контроллер - название довольно условное, и в своем проекте вы можете использовать другое.

Рассмотрим контроллер для сценария обновления поста. Напомню спецификации функций интерактора:

```
(s/fdef initial-params
  :args (s/cat :id ::post/id)
  :ret (s/or :ok  ::initial-params
             :err ::logged-out
             :err ::not-authorized
             :err ::not-found))

(s/fdef process
  :args (s/cat :id ::post/id
               :params any?)
  :ret (s/or :ok  ::processed
             :err ::logged-out
             :err ::not-authorized
             :err ::not-found
             :err ::invalid-params))
```

Мы должны показать пользователю форму и обработать данные этой формы.

Назовем экшены контроллера аналогично функциям интерактора: `initial-params` и `process`.

```
(ns publicator.web.controllers.post.update
  (:require
   [publicator.use-cases.interactors.post.update :as interactor]))

(defn- req->id [req]
  (-> req
      :route-params
      :id
      Integer.))

(defn initial-params [req]
  (let [id (req->id req)]
    [interactor/initial-params id]))

(defn process [{:keys [transit-params] :as req}]
  (let [id (req->id req)]
    [interactor/process id transit-params]))

(def routes
  #{[:get "/posts/:id{\\d+}/edit" #'initial-params :post.update/initial-params]
    [:post "/posts/:id{\\d+}/edit" #'process :post.update/process]})
```

Ключ `:route-params` добавляет библиотека роутинга, он содержит url параметры, в нашем случае - `id`.

Форма отправляет данные в формате transit. Соответствующая middleware добавляет ключ `:transit-params` с расшифрованными данными формы.

Тот факт, что контроллер не вызывает интерактор позволяет не подменять интерактор заглушкой и не реализовывать повторно сценарии тестирования интерактора:

```
(ns publicator.web.controllers.post.update-test
  (:require
   [publicator.utils.test.instrument :as instrument]
   [publicator.web.controllers.post.update :as sut]
   [publicator.use-cases.interactors.post.update :as interactor]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   [publicator.use-cases.test.factories :as factories]
   [ring.mock.request :as mock.request]
   [clojure.test :as t]
   [clojure.spec.alpha :as s]
   [sibiro.core]
   [sibiro.extras]))

(t/use-fixtures :once instrument/fixture)

(def handler
  (-> sut/routes
      sibiro.core/compile-routes
      sibiro.extras/make-handler))

(t/deftest initial-params
  (let [req             (mock.request/request :get "/posts/1/edit")
        [action & args] (handler req)
        args-spec       (-> `interactor/initial-params s/get-spec :args)]
    (t/is (= interactor/initial-params action))
    (t/is (nil? (s/explain-data args-spec args)))))

(t/deftest process
  (let [params          (factories/gen ::interactor/params)
        req             (-> (mock.request/request :post "/posts/1/edit")
                            (assoc :transit-params params))
        [action & args] (handler req)
        args-spec       (-> `interactor/process s/get-spec :args)]
    (t/is (= interactor/process action))
    (t/is (nil? (s/explain-data args-spec args)))))
```

Проверяем роутинг. Проверяем правильность возвращаемой функции интерактора, а также соответствие полученных аргументов интерактора их спецификации.

Кроме контроллеров - адаптеров интеракторов, в веб приложении есть потребность в обычных страницах. Экшены таких контроллеров возвращают не вектор, как описывалось ранее, а обычный ring ответ:

```
(ns publicator.web.controllers.pages.root
  (:require
   [publicator.web.responses :as responses]))

(defn show [_]
  (responses/render-page "pages/root" {}))

(def routes
  #{[:get "/" #'show :pages/root]})
```

В предыдущем параграфе мы видели middleware, оборачивающую экшены контроллеров:

```
(defn middleware [handler]
  (fn [req]
    (let [[interactor & args] (handler req)
          result              (apply interactor args)]
      (responder result args))))
```

Для того, чтобы обрабатывать обычные ring ответы добавим соответствующее условие:

```
(ns publicator.web.middlewares.responder
  (:require
   [publicator.web.responders.base :as responders.base]
   ;; ..
   ))

(defn wrap-reponder [handler]
  (fn [req]
    (let [resp (handler req)]
      (if (vector? resp)
        (let [[interactor & args] resp
              result              (apply interactor args)]
          (responders.base/result->resp result))
        resp))))
```

Самостоятельно просмотрите все контролеры.

5.9 Респондер

Респондер отвечает за преобразование ответа интерактора к ring ответу. Для этого он может использовать презентеры, шаблоны и формы, эти детали мы рассмотрим в следующих параграфах.

Рассмотрим респондер для сценария обновления поста. Напомню спецификации функций интерактора:

```
(s/fdef initial-params
  :args (s/cat :id ::post/id)
  :ret (s/or :ok  ::initial-params
             :err ::logged-out
             :err ::not-authorized
             :err ::not-found))

(s/fdef process
  :args (s/cat :id ::post/id
               :params any?)
  :ret (s/or :ok  ::processed
             :err ::logged-out
             :err ::not-authorized
             :err ::not-found
             :err ::invalid-params))
```

Как видно интерактор имеет 2 успешных и 4 провальных типа ответа. Очевидно, реакция на случаи `::logged-out`, `::not-authorized` и `::not-found` будет повторяться и для ответов других интеракторов.

Для обработки множества типов ответов удобно использовать мультиметод. При этом мультиметоды поддерживают наследование, что позволяет задать общие обработчики конкретным типам ответов.

```
(ns publicator.web.responders.base
  (:require
   [publicator.web.responses :as responses]
   [publicator.web.presenters.explain-data :as explain-data]
   [publicator.web.routing :as routing]))

(defmulti result->resp first)

(defmethod result->resp ::forbidden [_]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  {:status 403
   :headers {}
   :body "forbidden"})

(defmethod result->resp ::not-found [_]
  {:status 404
   :headers {}
   :body "not-found"})

(defmethod result->resp ::invalid-params [[_ explain-data]]
  (-> explain-data
      explain-data/->errors
      responses/render-errors))

(defmethod result->resp ::redirect-to-root [_]
  (responses/redirect-for-form (routing/path-for :pages/root)))
```

Здесь мы объявляем мультиметод, принимающий 2 аргумента: ответ интерактора и вектор аргументов. Также объявляются общие реализации для последующего связывания с конкретными ответами:

```
(ns publicator.web.responders.post.update
  (:require
   [publicator.use-cases.interactors.post.update :as interactor]
   [publicator.web.responders.base :as responders.base]
   [publicator.web.responses :as responses]
   [publicator.web.forms.post.params :as form]))

(defmethod responders.base/result->resp ::interactor/initial-params [[_ post params]]
  (let [form (form/build-update (:id post) params)]
    (responses/render-form form)))

(derive ::interactor/processed ::responders.base/redirect-to-root)
(derive ::interactor/invalid-params ::responders.base/invalid-params)
(derive ::interactor/logged-out ::responders.base/forbidden)
(derive ::interactor/not-authorized ::responders.base/forbidden)
(derive ::interactor/not-found ::responders.base/not-found)
```

Отмечу, что ответ с типом `::interactor/initial-params` не содержит идентификатора поста, этот идентификатор извлекается из аргументов с которыми был вызван интерактор.

```
(ns publicator.web.responders.post.update-test
  (:require
   [publicator.utils.test.instrument :as instrument]
   [publicator.web.responders.post.update :as sut]
   [publicator.web.responders.base :as responders.base]
   [publicator.use-cases.test.factories :as factories]
   [publicator.use-cases.interactors.post.update :as interactor]
   [publicator.web.responders.shared-testing :as shared-testing]
   [ring.util.http-predicates :as http-predicates]
   [clojure.spec.alpha :as s]
   [clojure.test :as t]))

(t/use-fixtures :once instrument/fixture)

(t/deftest all-implemented
  (shared-testing/all-responders-are-implemented `interactor/initial-params)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  (shared-testing/all-responders-are-implemented `interactor/process))

(t/deftest initial-params
  (let [result (factories/gen ::interactor/initial-params)
        resp   (responders.base/result->resp result)]
    (t/is (http-predicates/ok? resp))))
```

```
(ns publicator.web.responders.shared-testing
  (:require
   [publicator.web.responders.base :as responders.base]
   [clojure.spec.alpha :as s]
   [clojure.test :as t]))

(defn all-responders-are-implemented [sym]
  (t/testing sym
    (let [[_ & pairs] (-> sym s/get-spec :ret s/describe)
          specs       (keep-indexed
                       (fn [idx item] (if (odd? idx) item))
                       pairs)
          implemented (-> responders.base/result->resp methods keys)]
      (doseq [spec specs]
        (t/testing spec
          (t/is (some #(isa? spec %) implemented) "not implemented"))))))
```

5.10 Презентер

Презентер возвращает view model, которая передается в шаблон. View model содержит все данные, всю логику, так, чтобы шаблон был максимально простым и не требовал модульного тестирования. Все ссылки, активность кнопок устанавливаются тут. Таким образом при необходимости легко написать модульный тест для презентера.

```
(ns publicator.web.presenters.layout
  (:require
   [publicator.use-cases.services.user-session :as user-session]
   [publicator.web.routing :as routing]
   [ring.middleware.anti-forgery :as anti-forgery]))

(defn present [req]
  (cond-> {:csrf anti-forgery/*anti-forgery-token*}
    (user-session/logged-in?)
    (assoc :log-out {:text   "Log out"
                     :url    (routing/path-for :user.log-out/process)})

    (user-session/logged-out?)
    (assoc :register {:text "Register"
                      :url  (routing/path-for :user.register/initial-params)})

    (user-session/logged-out?)
    (assoc :log-in {:text "Log in"
                    :url  (routing/path-for :user.log-in/initial-params)})))
```

```
(ns publicator.web.presenters.post.list
  (:require
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   [publicator.use-cases.interactors.post.list :as interactor]
   [publicator.use-cases.interactors.post.create :as interactors.post.create]
   [publicator.use-cases.interactors.post.update :as interactors.post.update]
   [publicator.domain.aggregates.user :as user]
   [publicator.web.routing :as routing]))

(defn- post->model [post authorization]
  {:id             (:id post)
   :url            (routing/path-for :post.show/process {:id (-> post :id str)})
   :update-url     (routing/path-for :post.update/initial-params {:id (-> post :id str)})
   :title          (:title post)
   :can-update?    (= [::interactors.post.update/authorized] authorization)
   :user-full-name (::user/full-name post)})

(defn processed [posts]
  (let [authorizations (interactors.post.update/authorize (map :id posts))
        view-models    (map post->model posts authorizations)
        can-create?    (= [::interactors.post.create/authorized]
                          (interactors.post.create/authorize))]
    (cond-> {}
      :always     (assoc :posts view-models)
      can-create? (assoc :new {:text "New"
                               :url  (routing/path-for :post.create/initial-params)}))))
```

5.11 Шаблон

Можно использовать различные шаблонизаторы, в том числе и для java:

- hiccup
- selmer
- cljstache
- jade4j (java)
- thymeleaf (java)

Java шаблонизаторы принимают `Map<String, Object>` в качестве модели. Generics - это особенность Java, но не JVM, а clojure map поддерживают интерфейс Map, поэтому мы можем передавать просто хеши со строковыми ключами.

Из всех перечисленных шаблонизаторов самым простым является cljstache, с ним и будем работать.

Для рендеринга воспользуемся простой оберткой:

```
(ns publicator.web.template
  (:require
   [cljstache.core :as mustache]))

(defn render [template-name data]
  (let [path (str "publicator/web/templates/" template-name ".mustache")]
    (mustache/render-resource path data)))
```

Вот примеры шаблонов, для презентеров, рассмотренных ранее:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

        <meta name="csrf-token" content="{{csrf}}">
        <meta name="csrf-param" content="__anti-forgery-token">

        <link rel="stylesheet"
              href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
              integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
              crossorigin="anonymous">
    </head>
    <body>
        <nav class="navbar navbar-expand-lg navbar-light bg-light">
            <div class="container">
                <a class="navbar-brand" href="/">Publicator</a>
                <div class="navbar-nav mr-auto">
                    <a class="nav-item nav-link" href="/posts">Posts</a>
                </div>
                <div class="navbar-nav">
                    {{#log-in}}
                        <a class="nav-item nav-link"
                           href="{{url}}">
                            {{text}}
                        </a>
                    {{/log-in}}

                    {{#register}}
                        <a class="nav-item nav-link"
                           href="{{url}}">
                            {{text}}
                        </a>
                    {{/register}}

                    {{#log-out}}
                        <a class="nav-item nav-link"
                           data-method="post"
                           href="{{url}}">
                            {{text}}
                        </a>
                    {{/log-out}}
                </div>
            </div>
        </nav>
        <div class="container">
            {{&content}}
        </div>

        <script src="https://unpkg.com/rails-ujs@5.2.0/lib/assets/compiled/rails-ujs.js"></script>
        <script src="https://unpkg.com/form-ujs@0.0.2/dist/form-ujs.js"></script>
    </body>
</html>
```

```
{{#new}}
    <a href="{{url}}" class="btn btn-primary my-3">
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        {{text}}
    </a>
{{/new}}

<table class="table">
    <thead>
        <tr>
            <th scope="col">#</th>
            <th scope="col">Title</th>
            <th scope="col">Author</th>
            <th scope="col">Actions</th>
        </tr>
    </thead>
    <tbody>
        {{#posts}}
            <tr>
                <th scope="row">{{id}}</th>
                <td>
                    <a href="{{url}}">{{title}}</a>
                </td>
                <td>{{user-full-name}}</td>
                <td>
                    {{#can-update?}}
                        <a href="{{update-url}}">edit</a>
                    {{/can-update?}}
                </td>
            </tr>
        {{/posts}}
    </tbody>
</table>
```

5.12 Формы

Есть несколько способов работать с формами в web.

1. HTML формы, формируемые на backend. Подходит для простых случаев. Никаких автокомплитов, date-picker, вложенных форм и т.п.
2. HTML формы, формируемые на backend + js. Значительно лучше. Но в проект добавляется новая компонента - frontend. Приходится работать с другой технологией, управлять npm пакетами, использовать системы сборки. Появляются проблемы сериализации данных, например json не умеет сериализовывать даты. Сложно тестировать, т.к. это только интеграционные тесты с selenium и т.п.
3. Формы на js. Больше возможностей, сложнее формы. Но логика еще сильнее расползается между backend и frontend. Для специализированных или сложных форм это единственное решение.

Выделю следующие проблемы:

1. Кроме backend появляется еще и frontend. Разработчик должен овладеть новыми инструментами или команда пополняется frontend разработчиком.
2. Возникают проблемы с передачей данных. Скажем, некоторые поля формы имеют тип Date, UUID или множество Keyword. Приходится явно прописывать правила сериализации/десериализации.

3. Расползается логика, скажем в html добавили поле, а js - забыли.
4. Сложно тестировать.

Для типовых форм, которые, например, используются в админках можно решить эти проблемы.

5.12.1 Transit format

Для безболезненной передачи данных с бэкенда на фронтенд и обратно воспользуется форматом transit. transit-clj - библиотека для бэкенда, поддерживает все стандартные clojure типы и позволяет добавить собственные. transit-js - библиотека для фронтенда, добавляет свои типы для работы с transit типами.

В качестве транспорта используется json, а браузер имеет встроенную оптимизированную поддержку json, поэтому сериализация/десериализация происходит очень быстро. Транзит поддерживает замену повторяющихся частей короткими идентификаторами, поэтому, например массивы хешей занимают меньше места, чем json:

```
(def some-ids [{:very-long-id 1} {:very-long-id 2} {:very-long-id 3}])
(t/write w some-ids)
```

Результат:

```
[["^ ","~:very-long-id",1],["^ ","^0",2],["^ ","^0",3]]
```

Т.е. последующие упоминания `:very-long-id` заменяются на `^0`.

Подробнее вы можете прочитать в статьях:

- Transit format: An interactive tutorial - better than JSON (part 1)
- Transit format: An interactive tutorial - custom types and caching (part 2)

Transit-js добавляет свои типы:

```
import t from 'transit-js';
const kw = t.keyword;

t.map([
  kw('widget'), kw('input'),
  kw('type'), 'password',
  kw('label'), 'Password',
]),
```

в clojure это эквивалентно:

```
{:widget :input
 :type   "password"
 :label  "Password"}
```

5.12.2 Form-ujs

В мире Ruby on Rails популярен подход «Ненавязчивый javascript (Unobtrusive javascript)». Ненавязчивость подразумевает, что js на странице есть, но мы его не пишем. Ранее мы *знакомились* с проектом rails-ujs, который следует этой парадигме.

По аналогии я написал прототип библиотеки form-ujs, которая находит на странице описание формы и рендерит ее.

В код страницы нужно добавить один js тэг:

```
<script src="https://unpkg.com/form-ujs@0.0.2/dist/form-ujs.js"></script>
```

Бэкенд описывает форму в терминах стандартных виджетов:

```
(ns publicator.web.forms.user.register
  (:require
   [publicator.web.routing :as routing]))

(defn description []
  {:widget :submit, :name "Зарегистрироваться"
   :url (routing/path-for :user.register/process), :method :post, :nested
   {:widget :group, :nested
    [:login {:widget :input, :label "Логин"}
     :full-name {:widget :input, :label "Полное имя"}
     :password {:widget :input, :label "Пароль", :type "password"}]}})

(defn build [initial-params]
  {:initial-data initial-params
   :errors       {}
   :description  (description)})
```

Которое добавляется на страницу:

```
<div data-form-ujs='["^ ","~:initial-data",["^ "],"~:errors",["^ "],"~:description",["^ ","~:widget
↪","~:submit","~:name","Зарегистрироваться","~:url","/register","~:method","~:post","~:nested",["^
↪ ","^3","~:group","^9",["~:login",["^ ","^3","~:input","~:label","Логин"],"~:full-name",["^ ","^3
↪","^<","^=","Полное имя"],"~:password",["^ ","^3","^<","^=","Пароль","~:type","password"]]]]]' />
```

Результат можно посмотреть на демо-сайте.

5.12.3 Ошибки

Виджет `submit` по клику на кнопку отправляет данные на сервер. В случае успеха сервер может прислать редирект, а в случае ошибок - структуру с ошибками.

Для валидации используется `clojure.spec` и нужно привести эту структуру к человекопонятному виду:

```
(ns publicator.web.presenters.explain-data
  (:require
   [clojure.spec.alpha :as s]
   [phrase.alpha :as phrase]))

;; todo: использовать локализацию, например: https://github.com/tonsky/tongue

(phrase/defphraser :default
  [ctx {:keys [in]}]
  [in "Неизвестная ошибка"])

(phrase/defphraser #(contains? % k)
  [ctx {:keys [in]} k]
  [(conj in k) "Обязательное"])

(phrase/defphraser string?
  [ctx {:keys [in]}]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  [in "Должно быть строкой"])

(phrase/defphraser #(re-matches re %)
  [ctx {:keys [in]} re]
  (or
   (when-some [[_ r-min r-max] (re-matches #"\\w\{(\d+),(\d+)\}" (str re))]
     [in (str "Кол-во латинских букв и цифр от " r-min " до " r-max)])
   (when-some [[_ r-min r-max] (re-matches #"\.\{(\d+),(\d+)\}" (str re))]
     [in (str "Кол-во символов от " r-min " до " r-max)])
   [in "Неизвестная ошибка"]))

(defn ->errors [explain-data]
  (let [problems (::s/problems explain-data)
        pairs    (map #(phrase/phrase :ctx %) problems)]
    (reduce
     (fn [acc [in message]]
       (assoc-in acc (conj in :form-ujs/error) message))
     {}
     pairs)))
```

```
(ns publicator.web.presenters.explain-data-test
  (:require
   [clojure.test :as t]
   [clojure.spec.alpha :as s]
   [publicator.web.presenters.explain-data :as sut]))

(s/def ::for-required (s/keys :req-un [::required-1 ::required-2]))

(t/deftest required
  (let [ed     (s/explain-data ::for-required {})
        errors (sut/->errors ed)]
    (t/is (= {:required-1    {:form-ujs/error "Обязательное"}
              :required-2 {:form-ujs/error "Обязательное"}}
             errors))))

(s/def ::login (s/and string? #(re-matches #"\w{3,255}" %)))
(s/def ::password (s/and string? #(re-matches #".{8,255}" %)))

(s/def ::for-regexp-w (s/keys :req-un [::login]))
(s/def ::for-regexp-. (s/keys :req-un [::password]))

(t/deftest regexp
  (t/testing "\\w"
    (let [ed     (s/explain-data ::for-regexp-w {:login ""})
          errors (sut/->errors ed)]
      (t/is (= {:login {:form-ujs/error "Кол-во латинских букв и цифр от 3 до 255"}}
               errors))))
  (t/testing "."
    (let [ed     (s/explain-data ::for-regexp-. {:password ""})
          errors (sut/->errors ed)]
      (t/is (= {:password {:form-ujs/error "Кол-во символов от 8 до 255"}}
               errors)))))
```

5.13 Безопасность

Приложение не использует готовые фреймворки и за безопасность полностью отвечает разработчик.

С web связаны различные уязвимости рассмотрим некоторые из них.

5.13.1 Атака на идентификаторы

Мы используем последовательные идентификаторы, таким образом можно легко просмотреть все страницы.

Кроме этого идентификаторы глобально уникальны, и может возникнуть ситуация, когда в интерактор удаления поста злоумышленник передал идентификатор пользователя, а интерактор не проверяет права или тип агрегата.

Для защиты можно шифровать идентификаторы с помощью hashids или добавлять в query параметр подпись URL: `/posts/1?sign=some-sign`, а при обработке поверять эту подпись.

5.13.2 Cross Site Request Forgery

На backend используется библиотека ring-anti-forgery. Она состоит из middleware, проверяющая CSRF токен, и динамической переменной, содержащей этот токен.

```
(ns publicator.web.handler
  (:require
   [ring.middleware.anti-forgery :as ring.anti-forgery]
   [publicator.web.routing :as routing]
   ;;...
   ))

(defn build [binding-map]
  (-> routing/handler
      ;; ...
      ring.anti-forgery/wrap-anti-forgery
      ;; ...
      ))
```

Презентер получает токен, и добавляет его в ViewModel:

```
(ns publicator.web.presenters.layout
  (:require
   [publicator.use-cases.services.user-session :as user-session]
   [publicator.web.routing :as routing]
   [ring.middleware.anti-forgery :as anti-forgery]))

(defn present [req]
  (cond-> {:csrf anti-forgery/*anti-forgery-token*}
  ;; ...
  ))
```

В `html > head` добавляются мета-теги

```
<meta name="csrf-token" content="{{csrf}}">
<meta name="csrf-param" content="__anti-forgery-token">
```

Мы используем rails-ujs для реализации ссылок, отправляющих post запросы. Она автоматически подхватывает этот токен и вставляет его в форму при отправке.

form-ujs с помощью которой отправляются формы так же подхватывает токен.

5.14 System

Ранее мы познакомились к *компонентами*. Я упоминал, что компоненты объединяются в систему. Подробнее о системах.

Мы уже сделали достаточно, чтобы собрать полноценную систему и провести демонстрацию.

```
(defn build []
  (component/system-map
   :binding-map (->BindingMap nil)
   :seed        (component/using (->Seed nil)               [:binding-map])
   :handler     (component/using (handler/build)            [:binding-map])
   :jetty       (component/using (jetty/build {:port 4445}) [:binding-map :handler])))
```

Наша система состоит из 4-х компонентов:

- `binding-map` - содержит реализации абстракций вместе с их состоянием
- `seed` - компонент без состояния, который добавляет в хранилище начальные данные
- `handler` - компонент без состояния, оборачивающий ring handler
- `jetty` - веб-сервер

Эта система используется для разработки, но вы можете развертывать ее на тестовых серверах, для демонстрации промежуточного результата. Эта система использует фейки, хранящие данные в памяти, поэтому вы можете легко поднимать тестовое окружение на каждую фичу.

Файлы для разработки хранятся в директории web/dev .

```
(ns system
  (:require
   [com.stuartsierra.component :as component]
   [publicator.web.components.jetty :as jetty]
   [publicator.web.components.handler :as handler]
   [publicator.use-cases.test.factories :as factories]
   [publicator.use-cases.test.fakes.storage :as storage]
   [publicator.use-cases.test.fakes.user-queries :as user-q]
   [publicator.use-cases.test.fakes.post-queries :as post-q]
   [publicator.domain.test.fakes.id-generator :as id-generator]
   [publicator.domain.test.fakes.password-hasher :as password-hasher]))

(defrecord BindingMap [val]
  component/Lifecycle
  (start [this]
    (let [db (storage/build-db)]
      (assoc this :val
             (merge (storage/binding-map db)
                    (user-q/binding-map db)
                    (post-q/binding-map db)
                    (id-generator/binding-map)
                    (password-hasher/binding-map)))))
  (stop [this] this))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(defrecord Seed [binding-map]
  component/Lifecycle
  (start [this]
    (with-bindings (:val binding-map)
      (let [post1 (factories/create-post)
            user1 (factories/create-user {:login "user1"
                                          :password "12345678"
                                          :full-name "User1"
                                          :posts-ids #{(:id post1)}})
            post2 (factories/create-post)
            user2 (factories/create-user {:login "user2"
                                          :password "12345678"
                                          :full-name "User2"
                                          :posts-ids #{(:id post2)}})]))
    this)
  (stop [this]
    this))

(defn build []
  (component/system-map
   :binding-map (->BindingMap nil)
   :seed (component/using (->Seed nil)
                          [:binding-map])
   :handler (component/using (handler/build)
                             [:binding-map])
   :jetty (component/using (jetty/build {:port 4445})
                           [:binding-map :handler])))
```

```
(ns user
  (:require
   [com.stuartsierra.component :as component]
   [system]))

(def system (system/build))

(defn start []
  (alter-var-root #'system component/start))

(defn stop []
  (alter-var-root #'system component/stop))
```

Таким образом набрав в repl `(start)` вы запустите приложение.

Глава 6

Persistence

6.1 Введение

В этой главе рассмотрим реализацию абстракций работы с хранилищем.

Будем использовать PostgreSQL.

Исходники.

6.2 Инструменты

Здесь перечисляются базовые инструменты, которыми мы будем пользоваться.

6.2.1 clojure.jdbc

Для работы с БД воспользуемся clojure.jdbc. Кроме нее есть «стандартная» библиотека clojure/java.jdbc. Первая показалась мне удобнее. Об их различиях можно почитать тут.

Пример использования:

```
(require '[jdbc.core :as jdbc])

(with-open [conn (jdbc/connection dbspec)]
  (jdbc/execute conn "CREATE TABLE foo (id serial, name text);"))
```

Есть возможность прописать правила преобразования sql типов в clojure типы и обратно, пример.

В качестве `dbspec` можно передавать connection pool.

6.2.2 Connection pool

clojure.jdbc работает с различными пулами соединений. Я выбрал c3p0.

Оформим пул в компонент:

```
(ns publicator.persistence.components.data-source
  (:require
   [com.stuartsierra.component :as component])
  (:import
   [com.mchange.v2.c3p0 ComboPooledDataSource]))

(defrecord DataSource [config val]
  component/Lifecycle
  (start [this]
    (assoc this :val
           (doto (ComboPooledDataSource.)
             (.setJdbcUrl (:jdbc-url config))
             (.setUser (:user config))
             (.setPassword (:password config)))))
  (stop [this]
    (.close val)
    (assoc this :val nil)))

(defn build [config]
  (DataSource. config nil))
```

6.2.3 Query builder

Для clojure есть разные SQL query builder. Одни используют data DSL, как honeysql, другие - sql, как hugsql.

Я предпочитаю работать с sql, и не переводить мысленно код из DSL в sql. Также это позволяет без боли использовать расширения синтаксиса postgresql.

Но никто не запрещает использовать оба подхода в одном приложении. Data DSL отлично подходит для построения сложного запроса по большому количеству условий.

6.2.4 Test db

В следующих параграфах будут приводиться тесты и нужно разобраться как готовить базу для тестов.

Предполагается, что база данных заранее создана и ее настройки передаются через переменную окружения `TEST_DATABASE_URL`.

Хорошо это или плохо, но тесты в clojure независимы. Нет способа запустить произвольный код перед запуском тестов. Можно только для каждого неймспейса с тестами прописать фикстуры. Иными словами, для каждого неймспейса с тестами нужно явно готовить окружение. В нашем случае это запуск миграций.

Заведем неймспейс с фикстурами, подготавливающими тестовую бд. `publicator.persistence.test.db` экспортирует 2 функции `once-fixture` и `each-fixture`, которые мы должны использовать в своих тестах, работающих с БД.

Мы объявляем систему из 2х компонентов: `data-source` и `migration`.

```
(ns publicator.persistence.test.db
  (:require
   [publicator.persistence.components.data-source :as data-source]
   [publicator.persistence.components.migration :as migration]
   [publicator.persistence.utils.env :as env]
   [com.stuartsierra.component :as component]
   [jdbc.core :as jdbc]
   [hugsql.core :as hugsql]
   [hugsql.adapter.clojure-jdbc :as cj-adapter]))

(hugsql/def-db-fns "publicator/persistence/test/db.sql"
  {:adapter (cj-adapter/hugsql-adapter-clojure-jdbc)
   :quoting :ansi})

(defn- build-system []
  (component/system-map
   :data-source (data-source/build (env/data-source-opts "TEST_DATABASE_URL"))
   :migration (component/using (migration/build)
                               [:data-source])))

(defn- with-system [f]
  (let [system (atom (build-system))]
    (try
      (swap! system component/start)
      (f @system)
      (finally
        (swap! system component/stop)))))

(declare ^:dynamic *data-source*)

(defn once-fixture [t]
  (with-system
    (fn [system]
      (let [data-source (-> system :data-source :val)]
        (binding [*data-source* data-source]
          (t))))))

(defn each-fixture [t]
  (try
    (t)
    (finally
      (with-open [conn (jdbc/connection *data-source*)]
        (truncate-all conn)))))
```

Т.е. все тесты неймспейса выполняются в рамках одной системы, а после каждого теста происходит очистка. Тесты получают доступ к пулу коннектов с помощью динамической переменной `*data-source*`.

```
-- db.sql

-- :name- truncate-all :!
DO $$
DECLARE
  statements CURSOR FOR
  SELECT tablename FROM pg_tables
  WHERE schemaname = 'public'
    AND tablename != 'flyway_schema_history';
```

(continues on next page)

(продолжение с предыдущей страницы)

```
BEGIN
    FOR stmt IN statements LOOP
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(stmt.tablename);
    END LOOP;
END $$
```

6.3 Миграции

Думаю, не стоит рассказывать зачем нужны миграции БД и почему их стоит хранить в системе контроля версий.

Отмечу только, что миграции откатывать нельзя. Как вы откатите удаление колонки? Чтобы иметь возможность откатить деплой, нельзя ломать обратную совместимость схемы. Т.е. вы должны иметь возможность запускать новые миграции без выкатки нового кода. Подробнее в видео.

Для миграций я выбрал библиотеку flyway. Вы можете использовать что-то другое, что лучше подойдет вам.

Оформим запуск миграций в виде компонента, чтобы они автоматически запускались при старте системы:

```
(ns publicator.persistence.components.migration
  (:require
   [com.stuartsierra.component :as component])
  (:import
   [org.flywaydb.core Flyway]))

(defrecord Migration [data-source]
  component/Lifecycle
  (start [this]
    (doto (Flyway.)
      (.setDataSource (:val data-source))
      (.migrate))
    this)
  (stop [this]
    this))

(defn build []
  (Migration. nil))
```

Flyway автоматически загружает файлы миграций из classpath, поэтому добавим `recources` в файл deps.edn:

```
{:paths   ["src" "resources"]}
```

Миграции хранятся в директории `resources/db/migration` и должны иметь определенные имена, вроде `V1__id_sequence.sql`.

Самостоятельно ознакомьтесь с миграциями проекта.

6.4 Id generator

Напомню, что `id-generator` имеет один метод `generate`, который возвращает положительное целое число:

```
(ns publicator.domain.abstractions.id-generator
  (:require
   [clojure.spec.alpha :as s]))

(defprotocol IdGenerator
  (-generate [this]))

(declare ^:dynamic *id-generator*)

(s/def ::id pos-int?)

(s/fdef generate
  :ret ::id)

(defn generate []
  (-generate *id-generator*))
```

В PostgreSQL для генерации идентификаторов используют sequence.

Создадим ее первой миграцией:

```
-- persistence/resources/db/migration/V1__id_sequence.sql
CREATE SEQUENCE "id-generator";
```

Генерируется новый идентификатор следующим запросом:

```
SELECT nextval('id-generator') AS id
```

Добавим реализацию протокола `IdGenerator`:

```
(ns publicator.persistence.id-generator
  (:require
   [jdbc.core :as jdbc]
   [publicator.domain.abstractions.id-generator :as id-generator]))

(deftype IdGenerator [data-source]
  id-generator/IdGenerator
  (-generate [_]
    (with-open [conn (jdbc/connection data-source)]
      (let [stmt (jdbc/prepared-statement conn "SELECT nextval('id-generator') AS id")
            resp (jdbc/fetch-one conn stmt)]
        (:id resp)))))

(defn binding-map [datasource]
  {#'id-generator/*id-generator* (->IdGenerator datasource)})
```

И его тест:

```
(ns publicator.persistence.id-generator-test
  (:require
   [clojure.test :as t]
   [publicator.domain.abstractions.id-generator :as id-generator]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   [publicator.utils.test.instrument :as instrument]
   [publicator.persistence.test.db :as db]
   [publicator.persistence.id-generator :as sut]))

(defn- setup [t]
  (with-bindings (sut/binding-map db/*data-source*)
    (t)))

(t/use-fixtures :once
  instrument/fixture
  db/once-fixture)

(t/use-fixtures :each
  db/each-fixture
  setup)

(t/deftest generate
  (t/is (pos-int? (id-generator/generate))))
```

6.4.1 Оптимизация

Если поштучная генерация идентификаторов станет проблемой, ее можно оптимизировать. Можно увеличивать счетчик не на 1, а сразу на 1000 и держать в памяти диапазон идентификаторов и при необходимости генерировать новый. Однако, вы потеряете хронологическую сортировку идентификаторов, если запущено более одного инстанса приложения.

6.5 Storage

Напомню абстракцию хранилища:

```
(ns publicator.use-cases.abstractions.storage
  (:require
   [clojure.spec.alpha :as s]
   [publicator.domain.abstractions.id-generator :as id-generator]
   [publicator.domain.abstractions.aggregate :as aggregate]
   [publicator.domain.identity :as identity]
   [publicator.utils.ext :as ext]))

(defprotocol Storage
  (-wrap-tx [this body]))

(defprotocol Transaction
  (-get-many [t ids])
  (-create [t state]))

(s/fdef get-many
  :args (s/cat :tx any?
               :ids (s/coll-of ::id-generator/id :distinct true))
  :ret (s/map-of ::id-generator/id ::identity/identity))

(s/fdef create
  :args (s/cat :tx any?
```

(continues on next page)

(продолжение с предыдущей страницы)

```
               :state ::aggregate/aggregate)
  :ret ::identity/identity)

(defn get-many [t ids] (-get-many t ids))
(defn create   [t state] (-create t state))

(declare ^:dynamic *storage*)

(defmacro with-tx
  "Note that body forms may be called multiple times,
   and thus should be free of side effects."
  [tx-name & body-forms-free-of-side-effects]
  `(-wrap-tx *storage*
             (fn [~tx-name]
               ~@body-forms-free-of-side-effects)))

(s/fdef get-one
  :args (s/cat :tx any?
               :id ::id-generator/id)
  :ret (s/nilable ::identity/identity))

(defn get-one [t id]
  (let [res (get-many t [id])]
    (get res id)))

;; ...
```

Ранее я рассказывал, что есть 2 стратегии выполнения бизнес транзакций: оптимистический и пессимистический. Эта реализация будет основываться на оптимистической стратегии.

Исходя из специфики бизнес-транзакций можно реализовать и пессимистическую стратегию. Отмечу, что для этого не нужно переписывать сами бизнес-транзакции.

При использовании оптимистических блокировок мы свободно читаем любые агрегаты, но запоминаем их версии, а при фиксации проверяем, что версии не изменились. Если версии изменились, то повторяем бизнес-транзакцию.

Каждому агрегату будет соответствовать свой маппер, который реализует специфичную для этого агрегата persistence логику. Каждый маппер должен:

- извлекать (select) агрегаты по списку их идентификаторов
- вставлять (insert) агрегаты в базу
- удалять (delete) агрегаты из базы
- блокировать агрегат и извлекать его версию

Отмечу, что маппер поддерживает только вставку и удаление, но не изменение. Для надежности и упрощения кода изменение сведено к удалению и вставке. Да, это не оптимально с точки зрения производительности, зато просто и надежно. Если начнутся проблемы с производительностью, можно применить описанную далее оптимизацию.

В конце бизнес-транзакции нужно начать sql транзакцию и вычитать версии изменившихся агрегатов. Т.к. транзакции идут параллельно, то запрос должен содержать блокировку `FOR UPDATE`, чтобы другие транзакции дождались изменений в текущей транзакции.

Протокол маппера:

```
(defprotocol Mapper
  (-lock   [this conn ids])
  (-select [this conn ids])
  (-insert [this conn aggregates])
  (-delete [this conn ids]))

(s/def ::mapper #(satisfies? Mapper %))

(s/fdef lock
  :args (s/cat :this ::mapper, :conn any?, :ids (s/coll-of ::id-generator/id))
  :ret (s/coll-of ::versioned-id))

(s/fdef select
  :args (s/cat :this ::mapper, :conn any?, :ids (s/coll-of ::id-generator/id))
  :ret (s/coll-of ::versioned-aggregate))

(s/fdef insert
  :args (s/cat :this ::mapper, :conn any?, :aggregates (s/coll-of ::aggregate/aggregate))
  :ret any?)

(s/fdef delete
  :args (s/cat :this ::mapper, :conn any?, :ids (s/coll-of ::id-generator/id))
  :ret any?)
```

Для тестирования мы будем использовать тестовый агрегат, содержащий одно поле - счетчик:

```
(defrecord TestEntity [id counter]
  aggregate/Aggregate
  (id [_] id)
  (spec [_] any?))
```

Эта сущность будет храниться в таблице:

```
CREATE TABLE "test-entity" (
  "id" bigint PRIMARY KEY,
  "counter" integer
);
```

Отмечу, что эта таблица не имеет поля «версия». В PostgreSQL каждая таблица содержит служебную колонку `xmin`, будем использовать ее для отслеживания версии, т.к. нам достаточно определить совпадают версии или нет.

`xmin` - Идентификатор (код) транзакции, добавившей строку этой версии. (Версия строки — это её индивидуальное состояние; при каждом изменении создаётся новая версия одной и той же логической строки.) Подробнее.

Обратите внимание, `xmin` может переполняться и в этом случае начинает считать сначала. Иными словами, есть гипотетическая вероятность ложно-положительного сравнения версий, когда выполняется очень долгая бизнес-транзакция, а другая транзакция БД с тем же `xmin` изменила нашу запись.

Тестовый маппер использует следующие запросы:

```
-- :name- drop-test-entity-table :! :raw
DROP TABLE "test-entity"

-- :name- test-entity-insert :!
INSERT INTO "test-entity" VALUES :tuple*:vals;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
-- :name- test-entity-select :? :*
SELECT *, xmin AS version FROM "test-entity" WHERE id IN (:v*:ids)

-- :name- test-entity-delete :!
DELETE FROM "test-entity" WHERE id IN (:v*:ids)

-- :name- test-entity-locks :? :*
SELECT id, xmin AS version FROM "test-entity" WHERE id IN (:v*:ids) FOR UPDATE
```

В процессе исполнения бизнес-транзакции мы отслеживаем какие сущности извлекались, создавались или изменялись, так же как отслеживали в *фейковой реализации*.

В конце бизнес-транзакции мы выбираем с блокировкой версии измененных агрегатов, если версии не изменились, то группируем агрегаты по типу и производим удаление и вставку с помощью мапперов.

```
(ns publicator.persistence.storage
  (:require
   [jdbc.core :as jdbc]
   [publicator.use-cases.abstractions.storage :as storage]
   [publicator.domain.abstractions.aggregate :as aggregate]
   [publicator.domain.abstractions.id-generator :as id-generator]
   [publicator.domain.identity :as identity]
   [publicator.utils.ext :as ext]
   [clojure.spec.alpha :as s])
  (:import
   [java.util.concurrent TimeoutException]
   [java.time Instant]))

(s/def ::version some?)
(s/def ::versioned-id (s/keys :req-un [::id-generator/id ::version]))
(s/def ::versioned-aggregate (s/keys :req-un [::aggregate/aggregate ::version]))

;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defprotocol Mapper
  (-lock   [this conn ids])
  (-select [this conn ids])
  (-insert [this conn aggregates])
  (-delete [this conn ids]))

(s/def ::mapper #(satisfies? Mapper %))

(s/fdef lock
  :args (s/cat :this ::mapper, :conn any?, :ids (s/coll-of ::id-generator/id))
  :ret (s/coll-of ::versioned-id))

(s/fdef select
  :args (s/cat :this ::mapper, :conn any?, :ids (s/coll-of ::id-generator/id))
  :ret (s/coll-of ::versioned-aggregate))

(s/fdef insert
  :args (s/cat :this ::mapper, :conn any?, :aggregates (s/coll-of ::aggregate/aggregate))
  :ret any?)

(s/fdef delete
  :args (s/cat :this ::mapper, :conn any?, :ids (s/coll-of ::id-generator/id))
  :ret any?)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(defn- default-for-empty [f default]
  (fn [this conn coll]
    (if (empty? coll)
      default
      (f this conn coll))))

(def lock   (default-for-empty -lock   []))
(def select (default-for-empty -select []))
(def insert (default-for-empty -insert nil))
(def delete (default-for-empty -delete nil))

;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defrecord Transaction [data-source mappers identity-map]
  storage/Transaction
  (-get-many [this ids]
    (with-open [conn (jdbc/connection data-source)]
      (let [ids-for-select (remove #(contains? @identity-map %) ids)
            selected       (->> mappers
                                (vals)
                                (mapcat #(select % conn ids-for-select)) ;; <1>
                                (map (fn [{:keys [aggregate version]}]
                                       (let [iaggregate (identity/build aggregate)]
                                         (alter-meta! iaggregate assoc
                                                      ::version version
                                                      ::initial aggregate)
                                         iaggregate)))
                                (group-by #(-> % deref aggregate/id))
                                (ext/map-vals first))]
        ;; Здесь принципиально использование reverse-merge,
        ;; т.к. другой поток может успеть извлечь данные из базы,
        ;; создать объект-идентичность, записать его в identity map
        ;; и сделать в нем изменения.
        ;; Если использовать merge, то этот поток затрет идентичность
        ;; другим объектом-идентичностью с начальным состоянием.
        ;; Фактически это нарушает саму идею identity-map -
        ;; сопоставление ссылки на объект с его идентификатором
        (-> identity-map
            (swap! ext/reverse-merge selected)
            (select-keys ids)))))

  (-create [this state]
    (let [id     (aggregate/id state)
          istate (identity/build state)]
      (swap! identity-map (fn [map]
                            {:pre [(not (contains? map id))]}
                            (assoc map id istate)))
      istate)))

(defn- build-tx [data-source mappers]
  (Transaction. data-source mappers (atom {})))

(defn- need-insert? [identity]
  (not= @identity
        (-> identity meta ::initial)))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(defn- need-delete? [identity]
  (let [initial (-> identity meta ::initial)]
    (and (some? initial)
         (not= @identity initial))))

(defn- lock-all [conn mappers identities]
  (let [ids             (->> identities
                             (vals)
                             (filter need-delete?)
                             (map deref)
                             (map aggregate/id))
        db-versions     (->> mappers
                             (vals)
                             (mapcat #(lock % conn ids))
                             (group-by :id)
                             (ext/map-vals #(-> % first :version)))
        memory-versions (->> (select-keys identities ids)
                             (ext/map-vals #(-> % meta ::version)))]
    (= db-versions memory-versions)))

(defn- delete-all [conn mappers identities]
  (let [groups (->> identities
                    (vals)
                    (filter need-delete?)
                    (map deref)
                    (group-by class)
                    (ext/map-keys #(get mappers %))
                    (ext/map-vals #(map aggregate/id %)))]
    (doseq [[manager ids] groups]
      (delete manager conn ids))))

(defn- insert-all [conn mappers identities]
  (let [groups (->> identities
                    (vals)
                    (filter need-insert?)
                    (map deref)
                    (group-by class)
                    (ext/map-keys #(get mappers %)))]
    (doseq [[manager aggregates] groups]
      (insert manager conn aggregates))))

(defn- commit [tx mappers]
  (let [data-source (:data-source tx)
        identities  @(:identity-map tx)]
    (with-open [conn (jdbc/connection data-source)]
      (jdbc/atomic conn
                   (when (lock-all conn mappers identities)
                     (delete-all conn mappers identities)
                     (insert-all conn mappers identities)
                     true)))))

(defn- timestamp []
  (inst-ms (Instant/now)))

(deftype Storage [data-source mappers opts]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  storage/Storage
  (-wrap-tx [this body]
    (let [soft-timeout (get opts :soft-timeout-ms 500)
          stop-after   (+ (timestamp) soft-timeout)]
      (loop [attempt 0]
        (let [tx       (build-tx data-source mappers)
              res      (body tx)
              success? (commit tx mappers)]
          (cond
            success?                    res
            (< (timestamp) stop-after) (recur (inc attempt))
            :else                       (throw (TimeoutException.
                                                (str "Can't run transaction after "
                                                     attempt " attempts")))))))))



(s/fdef binding-map
  :args (s/cat :data-source any?
               :mappers (s/map-of class? ::mapper)
               :opts (s/? map?))
  :ret map?)

(defn binding-map
  ([data-source mappers]
   (binding-map data-source mappers {}))
  ([data-source mappers opts]
   {#'storage/*storage* (Storage. data-source mappers opts)}))
```

Тест повторяет тест *фейковой реализации*:

```
(ns publicator.persistence.storage-test
  (:require
   [publicator.utils.test.instrument :as instrument]
   [clojure.test :as t]
   [hugsql.core :as hugsql]
   [jdbc.core :as jdbc]
   [publicator.domain.abstractions.aggregate :as aggregate]
   [publicator.use-cases.abstractions.storage :as storage]
   [publicator.persistence.test.db :as db]
   [publicator.persistence.storage :as sut]))

(defrecord TestEntity [id counter]
  aggregate/Aggregate
  (id [_] id)
  (spec [_] any?))

(defn build-test-entity []
  (TestEntity. 42 0))

;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(hugsql/def-db-fns "publicator/persistence/storage_test.sql")

(defn- aggregate->sql [aggregate]
  (vals aggregate))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(defn- row->versioned-aggregate [row]
  {:aggregate (-> row (dissoc :version) map->TestEntity)
   :version   (-> row (get :version))})

(def mapper (reify sut/Mapper
              (-lock [_ conn ids]
                (test-entity-locks conn {:ids ids}))
              (-select [_ conn ids]
                (map row->versioned-aggregate (test-entity-select conn {:ids ids})))
              (-insert [_ conn states]
                (test-entity-insert conn {:vals (map aggregate->sql states)}))
              (-delete [_ conn ids]
                (test-entity-delete conn {:ids ids}))))

;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defn- setup [t]
  (with-bindings (sut/binding-map db/*data-source* {TestEntity mapper})
    (t)))

(defn- test-table [t]
  (with-open [conn (jdbc/connection db/*data-source*)]
    (drop-test-entity-table conn)
    (create-test-entity-table conn))
  (t))

(t/use-fixtures :once
  instrument/fixture
  db/once-fixture)

(t/use-fixtures :each
  db/each-fixture
  test-table
  setup)

(t/deftest create
  (let [entity (storage/tx-create (build-test-entity))]
    (t/is (some? (storage/tx-get-one (aggregate/id entity))))))

(t/deftest change
  (let [entity (storage/tx-create (build-test-entity))
        _      (storage/tx-alter entity update :counter inc)
        entity (storage/tx-get-one (:id entity))]
    (t/is (= 1 (:counter entity)))))

(t/deftest identity-map-persisted
  (let [id (:id (storage/tx-create (build-test-entity)))]
    (storage/with-tx t
      (let [x (storage/get-one t id)
            y (storage/get-one t id)]
        (t/is (identical? x y))))))

(t/deftest identity-map-in-memory
  (storage/with-tx t
    (let [x (storage/create t (build-test-entity))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
          y (storage/get-one t (aggregate/id @x))]
      (t/is (identical? x y)))))

(t/deftest identity-map-swap
  (storage/with-tx t
    (let [x (storage/create t (build-test-entity))
          y (storage/get-one t (aggregate/id @x))]
      (dosync (alter x update :counter inc))
      (t/is (= 1 (:counter @x) (:counter @y))))))

(t/deftest concurrency
  (let [test (storage/tx-create (build-test-entity))
        id   (aggregate/id test)
        n    10
        _    (->> (repeatedly #(future (storage/tx-alter test update :counter inc)))
                  (take n)
                  (doall)
                  (map deref)
                  (doall))
        test (storage/tx-get-one id)]
    (t/is (= n (:counter test)))))

(t/deftest inner-concurrency
  (let [test (storage/tx-create (build-test-entity))
        id   (aggregate/id test)
        n    10
        _    (storage/with-tx t
               (->> (repeatedly #(future (as-> id <>
                                           (storage/get-one t <>)
                                           (dosync (alter <> update :counter inc)))))
                    (take n)
                    (doall)
                    (map deref)
                    (doall)))
        test (storage/tx-get-one id)]
    (t/is (= n (:counter test)))))
```

6.5.1 Оптимизация

Например, у нас есть агрегат Пост, содержащий вложенные комментарии. Пост и Комментарий сохраняются в отдельных таблицах. Для начальной и новой версии агрегата нужно сгенерировать списки операций вставки:

```
;; initial
[[:post {:id 1, :title "123", :content "123"}]
 [:comment {:id 1, :title "awesome!"}]]

;; current
[[:post {:id 1, :title "123", :content "123 - additional text"}]
 [:comment {:id 1, :title "awesome!"}]]
```

Сравнивая эти списки получаем набор sql операций. В данном случае нужно только удалить и вставить строку с постом, т.к. комментарии не изменились.

Возможно, вы обратили внимание на <1>. Чтобы найти одну запись, нужно выполнить select для *всех* мапперов. Такой подход сильно упрощает логику, но ухудшает производительность. Если ожидается, что в вашем приложении будет большое кол-во агрегатов, то стоит добавить в абстракции поддержку пространств для идентификаторов:

```
;; было
(id-generator/generate)
(storage/get-many t some-ids)

;; стало
(id-generator/generate :user)
(storage/get-many t :user some-ids)
(aggregate/space user) ;; => :user
```

6.5.2 Mappers

Самостоятельно разберите мапперы Пользователя и Поста:

- миграции
- преобразование типов
- реализация
- тесты

6.6 Запросы

Для примера рассмотрим запросы для агрегата Пост, а именно получение списка постов и поста по идентификатору. При этом пост должен содержать дополнительные аттрибуты: идентификатор и полное имя автора.

Вот абстракция:

```
(ns publicator.use-cases.abstractions.post-queries
  (:require
   [publicator.domain.aggregates.user :as user]
   [publicator.domain.aggregates.post :as post]
   [clojure.spec.alpha :as s]))

(defprotocol GetList
  (-get-list [this]))

(declare ^:dynamic *get-list*)

(s/def ::post (s/merge ::post/post
                       (s/keys :req [::user/id ::user/full-name])))
(s/def ::posts (s/coll-of ::post))

(s/fdef get-list
  :args nil?
  :ret ::posts)

(defn get-list []
  (-get-list *get-list*))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(defprotocol GetById
  (-get-by-id [this id]))

(declare ^:dynamic *get-by-id*)

(s/fdef get-by-id
  :args (s/cat :id ::post/id)
  :ret (s/nilable ::post))

(defn get-by-id [id]
  (-get-by-id *get-by-id* id))
```

Напомню миграции создающие таблицы для постов и пользователей:

```
-- persistence/resources/db/migration/V2__create_post.sql

CREATE TABLE "post" (
  "id" bigint PRIMARY KEY,
  "title" varchar(255),
  "content" text,
  "created-at" timestamp
);
```

```
-- persistence/resources/db/migration/V3__create_user.sql

CREATE TABLE "user" (
  "id" bigint PRIMARY KEY,
  "login" varchar(255) UNIQUE,
  "full-name" varchar(255),
  "password-digest" text,
  "posts-ids" bigint[],
  "created-at" timestamp
);
```

Обратите внимание, что пользователь хранит идентификаторы постов с помощью postgresql массивов. При этом добавляются операции над массивами, например `@>` - «содержит».

Вот sql реализация запросов:

```
-- :name- post-get-list :? :n
SELECT "post".*,
       "user"."id"        AS "publicator.domain.aggregates.user/id",
       "user"."full-name" AS "publicator.domain.aggregates.user/full-name"
FROM "post"
JOIN "user" ON "user"."posts-ids" @> ARRAY["post"."id"]

-- :name- post-get-by-id :? :1
SELECT "post".*,
       "user"."id"        AS "publicator.domain.aggregates.user/id",
       "user"."full-name" AS "publicator.domain.aggregates.user/full-name"
FROM "post"
JOIN "user" ON "user"."posts-ids" @> ARRAY["post"."id"]
WHERE "post"."id" = :id
```

Отмечу, что БД не содержит индекса для `posts-ids`, но если вы будете хранить много данных, то

можете его добавить.

Нам осталось использовать эти запросы и выполнить некоторые преобразования типов:

```
(ns publicator.persistence.post-queries
  (:require
   [hugsql.core :as hugsql]
   [jdbc.core :as jdbc]
   [publicator.use-cases.abstractions.post-queries :as post-q]
   [publicator.domain.aggregates.post :as post]))

(hugsql/def-db-fns "publicator/persistence/post_queries.sql")

(defn- sql->post [row]
  (post/map->Post row))

(deftype GetList [data-source]
  post-q/GetList
  (-get-list [this]
    (with-open [conn (jdbc/connection data-source)]
      (map sql->post (post-get-list conn)))))

(deftype GetById [data-source]
  post-q/GetById
  (-get-by-id [this id]
    (with-open [conn (jdbc/connection data-source)]
      (when-let [row (post-get-by-id conn {:id id})]
        (sql->post row)))))

(defn binding-map [data-source]
  {#'post-q/*get-list*  (GetList. data-source)
   #'post-q/*get-by-id* (GetById. data-source)})
```

```
(ns publicator.persistence.post-queries-test
  (:require
   [clojure.test :as t]
   [publicator.utils.test.instrument :as instrument]
   [publicator.use-cases.test.factories :as factories]
   [publicator.domain.test.fakes.password-hasher :as fakes.password-hasher]
   [publicator.domain.test.fakes.id-generator :as fakes.id-generator]
   [publicator.persistence.storage :as persistence.storage]
   [publicator.persistence.storage.user-mapper :as user-mapper]
   [publicator.persistence.storage.post-mapper :as post-mapper]
   [publicator.persistence.test.db :as db]
   [publicator.use-cases.abstractions.post-queries :as post-q]
   [publicator.persistence.post-queries :as sut]
   [publicator.domain.aggregates.user :as user]))

(defn setup [t]
  (with-bindings (merge
                  (fakes.password-hasher/binding-map)
                  (fakes.id-generator/binding-map)
                  (persistence.storage/binding-map db/*data-source*
                                                   (merge
                                                    (user-mapper/mapper)
                                                    (post-mapper/mapper)))
                  (sut/binding-map db/*data-source*))
    (t)))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(t/use-fixtures :once
  instrument/fixture
  db/once-fixture)

(t/use-fixtures :each
  db/each-fixture
  setup)

(defn post-with-user [post user]
  (assoc post
         ::user/id (:id user)
         ::user/full-name (:full-name user)))

(t/deftest get-list-found
  (let [post (factories/create-post)
        user (factories/create-user {:posts-ids #{(:id post)}})
        res  (post-q/get-list)
        item (first res)]
    (t/is (= 1 (count res)))
    (t/is (= (post-with-user post user)
             item))))

(t/deftest get-list-empty
  (let [res (post-q/get-list)]
    (t/is (empty? res))))

(t/deftest get-by-id
  (let [post (factories/create-post)
        id   (:id post)
        user (factories/create-user {:posts-ids #{id}})
        item (post-q/get-by-id id)]
    (t/is (= (post-with-user post user)
             item))))

(t/deftest get-by-id-not-found
  (let [item (post-q/get-by-id 42)]
    (t/is (nil? item))))
```

Глава 7

Crypto

7.1 Password hasher

Для шифрования паролей и их проверки воспользуемся библиотекой buddy-hashers.

Напомню абстракцию:

```
(ns publicator.domain.abstractions.password-hasher
  (:refer-clojure :exclude [derive])
  (:require [clojure.spec.alpha :as s]))

;; check нужет, т.к. derive для одного и того же пароля может давать разные результаты,
;; т.к. результат может содержать случайную соль

(defprotocol PasswordHasher
  (-derive [this password])
  (-check [this attempt encrypted]))

(declare ^:dynamic *password-hasher*)

(s/def ::password string?)
(s/def ::encrypted string?)

(s/fdef derive
  :args (s/cat :password ::password)
  :ret ::encrypted
  :fn #(not= (-> % :args :password)
             (-> % :ret)))

(defn derive [password]
  (-derive *password-hasher* password))


(s/fdef check
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  :args (s/cat :attempt ::password
               :encrypted ::encrypted)
  :ret boolean?)

(defn check [attempt encrypted]
  (-check *password-hasher* attempt encrypted))
```

Вот ее реализация:

```
(ns publicator.crypto.password-hasher
  (:require
   [buddy.hashers]
   [publicator.domain.abstractions.password-hasher :as password-hasher]))

(deftype PasswordHasher []
  password-hasher/PasswordHasher
  (-derive [_ password]
    (buddy.hashers/derive password))
  (-check [_ attempt encrypted]
    (buddy.hashers/check attempt encrypted)))

(defn binding-map []
  {#'password-hasher/*password-hasher* (PasswordHasher.)})
```

И тест:

```
(ns publicator.crypto.password-hasher-test
  (:require
   [clojure.test :as t]
   [publicator.utils.test.instrument :as instrument]
   [publicator.crypto.password-hasher :as sut]
   [publicator.domain.abstractions.password-hasher :as password-hasher]))

(defn- setup [t]
  (with-bindings (sut/binding-map)
    (t)))

(t/use-fixtures :once
  instrument/fixture)

(t/use-fixtures :each
  setup)

(t/deftest ok
  (let [pass   "strong password"
        digest (password-hasher/derive pass)]
    (t/is (password-hasher/check pass digest))))
```

Глава 8

Main

8.1 Введение

Подпроект связывает все компоненты вместе и содержит `main` функцию.

Исходники.

8.2 System

Ранее, мы уже собирали систему. Но она состояла из фейковых реализаций. Настало время для промышленных компонентов.

```
(ns publicator.main.core
  (:require
   [com.stuartsierra.component :as component]
   [signal.handler :as signal]
   [publicator.web.components.jetty :as jetty]
   [publicator.web.components.handler :as handler]
   [publicator.persistence.components.data-source :as data-source]
   [publicator.persistence.components.migration :as migration]
   [publicator.persistence.utils.env :as env]
   [publicator.main.binding-map :as binding-map]))

(defn http-opts []
  {:host "0.0.0.0"
   :port (bigint (System/getenv "PORT"))})

(defn -main [& _]
  (let [system (component/system-map
                :data-source (data-source/build (env/data-source-opts "DATABASE_URL"))
                :migration (component/using (migration/build) [:data-source])
                :binding-map (component/using (binding-map/build) [:data-source])
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                :handler (component/using (handler/build) [:binding-map])
                :jetty (component/using (jetty/build (http-opts)) [:binding-map :handler]))
        system (component/start system)]
    (signal/with-handler :term
      (prn "caught SIGTERM, quitting.")
      (component/stop system)
      (System/exit 0))))
```

`binding-map` - компонент, содержащий все реализации, кроме `session`, т.к. ее реализация устанавливается в handler.

```
(ns publicator.main.binding-map
  (:require
   [com.stuartsierra.component :as component]
   [publicator.persistence.storage :as storage]
   [publicator.persistence.storage.user-mapper :as user-mapper]
   [publicator.persistence.storage.post-mapper :as post-mapper]
   [publicator.persistence.user-queries :as user-q]
   [publicator.persistence.post-queries :as post-q]
   [publicator.persistence.id-generator :as id-generator]
   [publicator.crypto.password-hasher :as password-hasher]))

(defrecord BindingMap [data-source val]
  component/Lifecycle
  (start [this]
    (let [data-source (:val data-source)
          mappers     (merge
                       (post-mapper/mapper)
                       (user-mapper/mapper))
          binding-map (merge
                       (storage/binding-map data-source mappers)
                       (user-q/binding-map data-source)
                       (post-q/binding-map data-source)
                       (password-hasher/binding-map)
                       (id-generator/binding-map data-source))]
      (assoc this :val binding-map)))
  (stop [this]
    (assoc this :val nil)))

(defn build []
  (BindingMap. nil nil))
```

8.3 Логирование

Если вы добавляете отладочную печать в код модуля, то, вероятно, вы нарушаете Single Responsibility Principle. Рассмотрим 2 случая.

Для отладки модуля в продакшене вам, как программисту, нужно собрать некоторые логи. И вы решаете добавить отладочную печать прямо в этот модуль. Т.е. инициатором изменения в коде являетесь вы, а не «владелец» модуля (например аналитик). Таким образом вы нарушаете SRP. Что, например, выльется в конфликты в системе контроля версий, когда другой разработчик внесет запрошенные аналитиком изменения.

Аналитику для принятия некоего решения потребовалось собрать статистику. В этом случае стоит ввести абстракцию логгера, как мы это делали, например для генератора идентификаторов.

В первом случае логирование нужно добавлять в подпроекте main. Т.е. воспользоваться Open-Close Principle, расширить поведение без модификации артефакта.

Можно использовать паттерн декоратор.

Например, мы хотим знать, какую длину паролей используют наши пользователи. Для этого нам нужно обернуть реализацию PasswordHasher в соответствующий декоратор.

Если мы хотим залогировать вызов функции, то можем применить декоратор к переменной, содержащей эту функцию. Воспользуемся `alter-var-root` или robert-hooke.