Оглавление

Об этой книге

У вас в руках книга о языке программирования Clojure. Это современный диалект Лиспа на платформе JVM. От устаревших диалектов он отличается тем, что делает ставку на функциональный подход и неизменяемость данных. Язык устроен так, чтобы решать сложные задачи простым способом.

Эта книга — не перевод, она изначально написана на русском языке. Вы не найдёте тяжёлых предложений, в которых слышна английская речь. Вам не придётся читать «маркер» вместо «токен» и другую нелепицу. Термины написаны в том виде, чтобы быть понятными программисту.

В книге нет вводной части, где написано, что скачать и установить. Также мы не рассматриваем азы вроде чисел и строк. На тему введения в Clojure уже написаны статьи и посты в блогах. Будет нечестно предлагать материал, где половина повторяет сказанное ранее. Эта книга от начала и до конца — то, о чём ещё никто не писал.

Другое её достоинство — упор на практику. Примеры кода взяты из реальных проектов. Все техники и приёмы автор опробовал лично. В описании проблем мы отталкиваемся от того, что вас ждёт на производстве. Покажем, где теория расходится с практикой и что предпочесть в таком случае.

Коротко о том, что вас ждёт. Начнём с веб-разработки — вспомним протокол HTTP и как с ним работать в Clojure. Затем рассмотрим Clojure.spec — библиотеку для проверки данных. Третья глава расскажет про исключения, четвёртая — про изменяемые данные. Далее переходим к конфигурации. В шестой главе знакомимся с системами. В последней научимся писать тесты.

Даже если вы не любите Лисп и книга попала к вам случайно, не спешите её откладывать. Clojure — это новые правила и другой мир, а книга — шанс туда попасть. Может быть, Clojure изменит ваше мнение о программировании. Обнаружит вопросы там, где, казалось бы, всё решено.

Во втором издании исправлены опечатки, ошибки и смысловые неточности. В некоторых местах текст сокращён, в других, наоборот, стал подробней. По просьбе читателей добавлены примеры к сложным разделам.

Желаем читателю терпения, чтобы прочесть книгу до конца.

Благодарности

Спасибо стартапу Flyerbee, моей первой работе на Clojure. Именно там я закрепил скромные знания языка.

Я счастлив работать в компании Exoscale в окружении талантливых инженеров. Многие вещи, не только технические, я узнал в этом коллективе.

Спасибо Петру Маслову и Евгению Климову за крупные партии найденных опечаток. Досбол Жантолин внёс важные замечания к последней главе. Молодцы все, кто указал на ошибки в комментариях в блоге.

Алексей Шипилов адаптировал книгу под мобильные устройства и выполнил много рутинных задач по вёрстке.

Вместе с Евгением Бартовым мы перевели книгу на английский язык. Во время перевода Евгений нашёл неточности в русской версии, которые мы тоже исправили.

Алексей Иванцов нашёл ошибки во втором издании книги и исправил их в репозитории на GitHub.

Благодарю коллектив издательства «ДМК Пресс» за то, что взяли рукопись в работу. Их усилиями вы читаете эту книгу сейчас.

Обратная связь

Автор будет признателен за указанные опечатки и неточности. Присылайте их по адресу `ivan@grishaev.me`. Возможно, в промежутках между тиражами получится обновить макет, и следующий читатель не увидит ошибки, о которой вы сообщили. Ваши замечания попадут и в английскую версию книги.

Код


Clojure book

Исходный код книги в виде файлов LaTeX находится на GitHub в репозитории `igrishaev/clj-book`[1]. Если вы нашли опечатку, откройте pull request или issue с описанием проблемы.

[1] github.com/igrishaev/clj-book

Book sessions

Все фрагменты кода из этой книги записаны в репозитории `igrishaev/book-sessions`[2]. Вы можете использовать код в любых целях, в том числе коммерческих.

Ресурсы

Следующие ресурсы помогут вам освоить язык и найти на нём работу.

Clojure

- Официальный сайт Clojure[3]. Его разделы «Getting Started», «Reference» и «Guides» подробно описывают язык и экосистему в целом. Прочтите их, даже если уверены в своих знаниях.

Slack Clojurians

- Сообщество в Slack под названием Clojurians[4]. Включает сотни каналов на разные темы, в том числе для отдельных библиотек и проектов. Каналы с кодами стран объединяют пользователей по языку. Есть канал `#ru` для русскоговорящих пользователей.

Telegram cloju-re_ru

- Чат в Телеграме[5] на русском языке. Основные темы: решение проблем, советы по оформлению кода, вакансии и поиск работы, анонсы мероприятий.

Ask Clojure

- Ask Clojure[6] — сервис вопросов и ответов по языку и его окружению, аналог StackOverflow.

[2] github.com/igrishaev/book-sessions
[3] clojure.org
[4] clojurians.slack.com
[5] t.me/clojure_ru
[6] ask.clojure.org

Глава 1

Веб-разработка

В первой главе мы рассмотрим, как писать веб-приложения на Clojure. Поговорим о передаче данных по протоколу HTTP. Какие абстракции над ним возводят и что предлагает Clojure. Чем хорош функциональный подход и почему разработка на нём удобнее.


Cognitect 2018

Каждый год компания Cognitect опрашивает[1] разработчиков на Clojure. Один из вопросов уточняет, в какой области вы работаете. В 2010 году под веб писала половина опрошенных. К 2018 году эта цифра выросла до 80%, что уже четыре человека из пяти. Похожую динамику показывают опросы StackOverflow[2]. Согласно им, всё больше инженеров переходят в веб из смежных областей.


Stack-Overflow 2018

Если вы найдёте работу на Clojure, скорее всего это будет веб-приложение. Мы специально не говорим «сайт», потому что термин уходит в прошлое. Сегодня веб-приложение — это не только текст с картинками. В широком плане это сложный обмен данными по HTTP.

Протокол служит для передачи разметки HTML, но со временем подошёл и для данных. Его дизайн оказался настолько гибким, что не пришлось менять стандарт. Прежде чем перейти к Clojure, освежим в памяти устройство протокола: из каких частей он состоит и как с ним работает сервер. Это важно, потому что языки и фреймворки меняются, а протокол нет.

[1] cognitect.com/blog/2017/1/31/clojure-2018-results
[2] insights.stackoverflow.com/survey/2018

1.1 Основы HTTP


RFC 2616

Протокол HTTP работает поверх стека TCP/IP. В широком смысле протоколы — это соглашения о том, как обмениваться данными. Они записаны в официальных документах. Документ HTTP называется RFC 2616[3]. С ним сверяются разработчики фреймворков и браузеров, чтобы код работал на разных языках и платформах.

HTTP удобен тем, что это текст. Не нужно парсить байты, чтобы понять, что происходит. Протокол работает и с бинарными данными, но главные его части остаются текстом. В HTTP различают запрос и ответ. Оба состоят из трёх частей: первая строка, заголовки и тело.

Первая (стартовая) строка несёт самую важную информацию. Её формат отличается для запроса и ответа. Для запроса это метод, путь и версия, для ответа — статус, сообщение и версия.

Заголовки — это пары ключей и значений. В коде их описывают словарём. Заголовки несут дополнительные сведения о запросе или ответе. Например, `Content-Type` сообщает, как читать тело. Был ли это XML- или JSON-документ? Программа сверяет заголовок и читает тело должным образом.

После заголовков следует тело. Им может быть что угодно — текст, пары полей и значений, JSON, картинка. Стандарт допускает смешанный тип, `multipart-encoding`. Тело такого запроса состоит из ячеек, в каждой из которых своё содержимое: текст, картинка, снова текст, архив.

Рассмотрим примеры трафика HTTP. Именно в таком виде его передают по сети. Ниже запрос к главной странице Google по слову clojure:

```
GET /search?q=clojure HTTP/1.1
Host: google.com
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE)
```

А это POST-запрос с JSON:

[3] tools.ietf.org/html/rfc2616

```
POST /api/users/ HTTP/1.1
Host: example.com
Content-Type: application/json

{"username": "John", "city": "NY"}
```

Обратите внимание на пустую строку выше: она отделяет тело от заголовков. Ответ на этот запрос:

```
HTTP/1.1 200 OK
Date: Tue, 19 Mar 2019 15:57:11 GMT
Server: Nginx
Connection: close
Content-Type: application/json

{
  "code": "CREATED",
  "message": "User has been created."
}
```

Видно, как изящно устроен протокол: данные идут по убыванию важности. Прочитав только первую строку, клиент и сервер готовы принять решение о том, что делать дальше.

Рассмотрим случай, когда метод и путь запроса равны `GET /about`, но такой страницы не существует. Сервер проверит путь по таблице маршрутов. Если его нет, получим ответ со статусом 404. Статус идёт раньше тела, что открывает путь для оптимизации. Логика клиента может быть такова, что, получив негативный статус, он пропустит ответ и потому справится быстрее. Подход выгоден и серверу, потому что ему не придётся пересылать тело.

Чтение и разбор всего содержимого занимает много времени. Современные фреймворки не делают этого зря. По заголовку `Content-Type` они определяют, стоит ли читать тело. Если приложение работает только с JSON, то для `text/xml` получим ошибку. Аналогично поступают с заголовком `Content-Length`, где указана длина тела в байтах. Если значение больше лимита, сервер отклонит запрос до чтения.

Главные части запроса — это *метод* и *путь*. Путь указывает на определённый ресурс на сервере. Иногда он означает файл относительно заданной папки. Например, `/images/map.jpg` вернёт

одноимённый файл из `/var/www/static`. Раздача файлов — это частный случай пути, и у него много других сценариев. В пути может быть номер сущности: `/users/9677/profile`. Сервер можно настроить так, что запросы с префиксом `/internal` и `/public` уходят на разные машины.

Метод запроса означает действие, которое мы намерены выполнить над ресурсом. Основные методы — это `GET`, `POST`, `PUT` и `DELETE`, что значит прочитать, создать, обновить и удалить ресурс. Запрос `POST /users/` означает создать пользователя, а `GET /users/` — получить список пользователей.

Главный параметр ответа — это статус, целое положительное число. Статусы группируют по старшей цифре. Значения с 200 до 299 считают положительными. Они означают, что сервер обработал запрос без ошибки. Для краткости интервал обозначают `2xx`.

Значения из группы `3xx` связаны с направлением на другую страницу. В заголовке `Location` указан адрес, куда нужно отправить новый запрос. Современные браузеры и клиенты делают это автоматически. По адресу `http://yandex.ru` получим пустой документ с заголовком `Location: https://yandex.ru`. Разница в схеме протокола: сервер обязывает перейти на безопасное соединение. Мы даже не заметим этого, потому что браузер сделает это сам.

Статусы `4xx` означают ошибку на стороне клиента. Чаще других встречается 404 — страница не найдена. Если прислать ошибочные данные, сервер ответит: 400 Bad request. Когда нет прав доступа, получим код 403.

Значения из группы `5xx` говорят о проблеме на стороне сервера. В основном это ошибки в коде: отказ базы данных, нехватка места на диске. Если сервер на техобслуживании, он вернёт код 503. В редких случаях сервер выключен и не отвечает на запросы.

Принято считать, что ответ со статусом, отличным от `2xx` означает ошибку. Многие HTTP-клиенты бросают исключение на ответ с негативным статусом. Это верно только на прикладном уровне, когда мы пишем код. С точки зрения протокола ответ 404 такой же правильный, как и 200.

Когда действий с ресурсом много, применяют другие, более редкие методы. Например, HEAD — получить краткие сведения о сущности. Сервис Amazon S3 в ответ на HEAD вернёт только

статус и заголовки с пустым телом. В них указаны тип файла и его размер, контрольная сумма, дата изменения. HEAD-запрос предпочтительней GET. Обычно метаданные хранят отдельно от файла, поэтому доступ к ним быстрее, чем к диску.


REST

Подход «метод и ресурс» вырос в то, что сегодня называется REST[4]. Сторонники REST выделяют сущности и CRUD-операции над ними (**C**reate, **R**ead, **U**pdate, **D**elete). Считается верным подход, когда сущность задают через путь, например `/users/1`, а операцию — методом. Если это запрос на изменение, данные читают из тела с JSON.

REST — не идеальный и не единственный подход к веб-разработке. Он конкурирует с JSON-RPC, gRPC и другими аналогами. В этой книге мы не будем задерживаться на конкретной парадигме. Протокол не заставляет следовать REST и другим правилам. Работайте с HTTP так, как это удобно проекту. Идеальная архитектура не обещает успех, и наоборот: успех не значит, что в коде всё идеально.

1.1.1 Фреймворк

Фреймворк — это абстракция над HTTP. Разработчик не читает запрос по байтам вручную — задачу берёт на себя чужой код. Взамен нам дают классы, чтобы описать логику приложения. Типичный проект на Python или Java состоит из следующих классов.

`Application` — это главная сущность проекта: она группирует классы рангом ниже. `Router` определяет, на какой обработчик подать входящий запрос — `Request`. Обработчик — это класс `Handler` с методами `.onGet`, `.onPost` и другими. Они вернут экземпляр класса `Response`. Так устроены промышленные фреймворки вроде Django и Rails. Имена и состав классов отличаются, но смысл прежний: приложение, роутер, обработчик, запрос и ответ.

Большие проекты делят на слои. Слой транспорта отвечает за обмен данными, слой логики исполняет код, ничего не зная об источнике данных. С таким подходом логика не зависит от транспорта, и последний можно сменить в любой момент. Например, направить долгий запрос в очередь задач или ввести данные через

[4] restapitutorial.com

CLI-интерфейс. На практике это работает не всегда: по разным причинам, в том числе из-за спешки, слои перемешиваются.

Проекты на Clojure опираются на фреймворки. Принципы, о которых мы говорили выше, справедливы и для этого языка.

1.2 HTTP в Clojure

James Reeves

Разработчик Джеймс Ривз[5] (James Reeves) известен вкладом в экосистему Clojure. Нет проекта, который бы не использовал его библиотеки. Джеймс ввёл стандарт веб-разработки для Clojure на заре языка. Стандарт опирается на несколько простых идей.

Приложения бывают сколь угодно сложными: они полагаются на сторонние сервисы, машинное обучение, учитывают сотню фактов о клиенте. Но даже самое сложное приложение принимает запрос и возвращает ответ, и поэтому это функция. Скептики заметят, что мысль не нова. В Django обработчик тоже бывает не классом, а функцией. Разница в том, что обработчик — это ещё не приложение. Ему не хватает роутера, middleware и других абстракций. Функция-обработчик в других языках — всего лишь приятная возможность.

В Clojure приложение остаётся функцией *на всех* уровнях. Маршрут — это функция, которая принимает запрос, ищет обработчик и передаёт ему управление. Middleware — тоже функция, которая дополняет приложение логикой. Каждую тяжёлую абстракцию (классы `Application`, `Router`, `Handler`) в Clojure заменяют функцией. Это удобно, потому что в отличие от классов функции компонуются между собой.

Другая идея в том, чтобы зафиксировать структуру запроса и ответа. Должны быть документы (не код, а именно документы), где описаны поля и их семантика. Это напоминает протокол HTTP: спецификация упрощает код и делает его переносимым. Удобно, когда разные проекты на Clojure работают с одними структурами. Если фреймворк соблюдает стандарт, к нему легче привлечь сообщество. Следовать ему в интересах разработчика.

[5] www.booleanknot.com

1.2.1 Ring


Ring

Идеи воплотились в проекте Ring[6]. Сегодня это стандарт разработки на Clojure под веб. Репозиторий содержит описание запроса и ответа и базовый код для работы с ними. Прилагаются основные middleware, обёртка для сервера Jetty и документация.

Со временем появился термин «Ring-совместимость». Ему следуют все фреймворки на Clojure. Ring-приложение работает на платформах Jetty, Immutant и других без изменений в коде.

Библиотека Ring разбита на отдельные части, чтобы можно было установить только нужные. Перечислим компоненты, которые понадобятся по ходу главы:

- `ring-core` — базовый набор: параметры, куки, сессии;
- `ring-jetty-adapter` — запуск сервера из функции;
- `ring/ring-json` — поддержка JSON.

Первое приложение мы напишем даже без библиотеки. Вот оно:

```
(defn app [request]
  (let [{:keys [uri request-method]} request]
    {:status 200
     :headers {"Content-Type" "text/plain"}
     :body (format "You requested %s %s"
                   (-> request-method name .toUpperCase)
                   uri)}))
```

Приложение читает путь и метод запроса и строит ответ. Его статус положительный — 200. Мы выставили заголовок с типом «простой текст». Поле `:body` содержит строку, которую получим функцией `format`. Поскольку `app` — функция, вызовем её с разными запросами:

```
(app {:request-method :get :uri "/index.html"})

{:status 200
 :headers {"Content-Type" "text/plain"}
 :body "You requested GET /index.html"}
```

[6] github.com/ring-clojure/ring

```
(app {:request-method :post :uri "/users"})

{:status 200
 :headers {"Content-Type" "text/plain"}
 :body "You requested POST /users"}
```

Кругом словари, и не ясно, что будет в браузере. Запустим приложение в виде сервера. Он принимает приложение, параметры и включает сложный процесс. Сервер слушает указанный порт и читает байты. Из бинарных данных он строит словарь запроса. В отдельном потоке сервер вызывает функцию приложения с этим словарём. Получим словарь ответа. Сервер переводит ответ в байты и пишет в удаленный порт клиента. Цикл повторяется для каждого запроса.

Добавим в проект зависимости:

```
[ring/ring-core "1.7.1"]
[ring/ring-jetty-adapter "1.7.1"]
```

Запустим сервер:

```
(require '[ring.adapter.jetty :refer [run-jetty]])
(run-jetty app {:port 8080 :join? true})
```

Происходит следующее. Мы добавили в текущий модуль функцию `run-jetty`. Она принимает приложение и словарь опций. Ключ `join?` определяет, будет ли заблокирован текущий поток до конца работы сервера. Если передать `false`, сервер запустится в фоне. Чтобы остановить его, нужно поместить результат `run-jetty` в переменную и позже вызвать у неё метод `.stop`:

```
(def server
  (run-jetty app {:port 8080 :join? false}))

;; after a while
(.stop server)
```

Если флаг — истина (как в первом случае), главный поток будет ждать до тех пор, пока сервер не выключат. Чтобы это сделать, нажмите `Ctrl+C`. Пока сервер работает, откройте браузер по

адресу `http://localhost:8080`. Вы увидите строку «You requested GET /». Теперь измените путь на `/hello` или `/some/file.txt` — сообщение изменится.

1.3 Запросы и ответы

Мы написали приложение, которое на все запросы печатает метод и путь. Кроме этих полей, запрос содержит порт и адрес сервера, строку параметров, тип протокола, заголовки и тело. Всё вместе — это неизменяемый словарь с ключами типа `keyword`. Полное описание запроса и ответа смотрите в репозитории на GitHub[7].


Ring spec

Здесь и далее будем писать слово `keyword` по-русски: «кейворд». В других языках тип называют токеном или тегом.

Обратим внимание на поля запросов `:headers` и `:body`. Заголовки — это неизменяемый словарь, но его ключи — не кейворды, а строки. Такой словарь не работает с разбиением по `:keys`. Ниже переменная `host` окажется равна `nil`:

```
(defn some-handler
  [request]
  (let [{:keys [headers]} request
        {:keys [host]} headers]
    ...))
```

Чтобы извлечь заголовки правильно, используйте `:strs`:

```
(defn some-handler
  [{:keys [headers]}]
  (let [{:strs [host user-agent]} headers]
    ...))
```

или функцию `get` со строкой:

```
(get headers "host") ;; "127.0.0.1"
```

Имя заголовка всегда в нижнем регистре. В протоколе HTTP варианты `Content-Type` и `content-type` одинаковы, но в Java

[7] github.com/ring-clojure/ring/blob/master/SPEC

(и поэтому в Clojure) регистр имеет значение. Ring приводит заголовки к нижнему регистру, чтобы избежать недоразумений.

Значения заголовков — тоже строки. Стандарт задаёт типы некоторых заголовков, но Ring не выводит их. Например, `Content-Length` передаёт длину тела в байтах. Современные фреймворки приводят его к числу и помещают в отдельное поле. По умолчанию Ring не делает ничего подобного, но это легко исправить. Ниже мы рассмотрим как именно.

Новички забывают, что ключи заголовков — это строки. Так появляется ошибка, когда вместо правильного значения приходит `nil`:

```
(get headers :host) ;; nil
```

Можно обработать заголовки, сменив тип ключей. Для одного случая это нормально, но если это делает каждый обработчик, получается лишняя работа. Приложение меняют так, чтобы в функцию приходили уже исправленные заголовки. Техника называется middleware, и мы рассмотрим её по ходу главы (с. 41).

Поле запроса `:body` опционально. Согласно HTTP, тела может не быть. Обратите внимание на его тип: это не строка, а входящий поток — `java.io.InputStream`. Поток — это источник данных, который читают только раз. По умолчанию Ring не читает поток. Делать это или нет, остаётся на ваше усмотрение.

Чтение и разбор тела — это сложная операция. По заголовкам определяют тип документа, его длину и читают нужное число байт. Из них восстанавливают данные (JSON, XML). Технически возможно послать JSON-документ, но указать тип `text/xml`. Чтение такого запроса обернётся ошибкой. Сервер должен быть готов к подобным сценариям.

Легче всего прочитать тело в строку функцией `slurp`:

```
(defn handler [request]
 (let [content (-> request :body slurp)]
  {:status 200
   :headers {"content-type" "text/plain"}
   :body (format "Content: %s" content)}))
```

В современном вебе всё меньше работают с текстом: на его место приходят данные в виде JSON. Позже мы рассмотрим, как подружить Ring с этим форматом.

Ответ Ring — это словарь с полями `:status`, `:headers` и `:body`:

- `:status` — целое число, признак успеха или неудачи. Мы рассмотрели семантику статуса в начале главы;
- `:headers` — заголовки ответа с ключами-строками, например:

```
{:status 302
 :headers {"content-length" 0
           "location" "/new/page.html"}}
```

- `:body` — тело ответа. Как и в запросе, тела может не быть. Обычно это строка, но может быть и файл, ресурс или поток.

1.4 Маршруты

Мы запустили приложение и проверили в браузере. На любой запрос оно выдаёт текст с небольшими отличиями. На практике приложение строят из обработчиков, каждый из которых решает узкую задачу. Входящие запросы распределяют по ним согласно правилам. Процесс называют маршрутизацией или роутингом.

В Clojure и Ring нет класса роутера. Его роль играет функция, которая принимает обработчики и возвращает приложение, тоже функцию. Приложение принимает запрос и по методу и пути подбирает нужный обработчик. Затем вызывает его с запросом и возвращает ответ.

Вообразим, что по адресу `/` мы бы хотели видеть название сайта, а по `/hello` — приветствие. Другие адреса вернут `404 Page not found`. Напишем обработчики:

```
(defn page-index [request]
  {:status 200
   :headers {"content-type" "text/plain"}
   :body "Learning Clojure"})
```

```
(defn page-hello [request]
  {:status 200
   :headers {"content-type" "text/plain"}
   :body "Hi there! Keep trying!"})

(defn page-404 [request]
  {:status 404
   :headers {"content-type" "text/plain"}
   :body "Page not found."})
```

Каждый обработчик можно запустить в виде сервера и открыть в браузере. Осталось связать их в единое целое.

1.4.1 Наивный подход

Сделаем самое простое, что приходит в голову. Напишем обработчик, который находит маршрут вручную. Для этого проверим путь оператором `case`:

```
(defn app [request]
  (let [{:keys [uri]} request]
    (case uri
      "/"      (page-index request)
      "/hello" (page-hello request)
      (page-404 request))))
```

Ответ функции зависит от поля запроса `:uri`. Запустите приложение в браузере и проверьте разные адреса. Это наивный подход, но он работает.

Недостатки функции очевидны. Мы не учитываем метод запроса: `GET /users` и `POST /users` отличаются по смыслу. Мы сравниваем пути в лоб без учёта параметров. В правильных маршрутах пути `/users/1` и `/users/99` сходятся в один обработчик с разным параметром `id`. Код получился шумный. Хотелось бы задать маршруты правилами, декларативно.

Эти и другие проблемы решают библиотеки. Мы рассмотрим Compojure и Bidi. Обе строят маршруты, но их подходы ортогональны.

1.4.2 Compojure


Compojure

Библиотека Compojure[8] предлагает макросы для описания маршрутов. Макросы похожи на таблицу правил. Добавим зависимость в проект:

```
[compojure "1.6.1"]
```

Так выглядит приложение на Compojure. Оно чище и короче того, что мы написали вначале.

```
(require '[compojure.core :refer [GET defroutes]])

(defroutes app
  (GET "/"      request (page-index request))
  (GET "/hello" request (page-hello request))
  page-404)
```

Разберёмся, что получилось на выходе. Переменная `app` — функция, которая принимает запрос. Мы задали её не через `def` или `defn`, а макросом `defroutes`. Он строит функцию-роутер и связывает её с переменной `app`. С макросом получается меньше кода.

После имени следуют правила. Правило — это форма ⟨ метод, путь, переменная, выражение ⟩. Её читают так: если метод и путь подходят текущему запросу, связать его с переменной и выполнить выражение. Compojure строит из правила функцию с той же логикой. Согласно первому правилу, для метода и пути `GET /` получим ответ `(page-index request)`. Результат станет ответом сервера.

Макрос `defroutes` оборачивает несколько правил в перебор. На каждом шаге он берёт очередное правило и применяет к нему запрос. Первое значение, отличное от `nil`, станет ответом к текущему запросу.

Что будет, если не подошло ни одно правило? Результат `nil` вызовет ошибку сервера. Чтобы избежать `nil`, к правилам добавляют ещё одно, которое сработает всегда. Это функция `page-404`: её результат не зависит от запроса. Так мы гарантируем, что даже

[8] github.com/weavejester/compojure

если запрос не подошёл двум первым правилам, получим ответ 404, а не ошибку типов.

Так устроен роутинг на Compojure. Приложение состоит из отдельных обработчиков. С помощью макросов `GET` и `POST` их оборачивают в правила. Правило строит функцию, которая проверяет, что метод и путь подходят. Если да, получим вызов обработчика с запросом.

1.4.3 Продвинутые возможности

Выше мы обозначили проблему: правила `GET /users/1` и `GET /users/99` — это один обработчик с параметром. Его записывают так:

```
(GET "/users/:id" [id :as request] (page-user request))
```

Обратите внимание на двоеточие перед `id` и квадратные скобки в середине. Часть пути с двоеточием означает параметр. На время запроса Compojure поместит его в словарь `params`.

Предположим, страница `page-user` выводит имя и фамилию пользователя по номеру из пути. Условная функция `get-user-by-id` читает пользователя из базы и возвращает словарь. Находим в словаре имя и фамилию и составляем строку.

```
(defn page-user [request]
  (let [user (-> request :params :id get-user-by-id)
        {:keys [fname lname]} user]
    {:status 200
     :body (format "User is %s %s" fname lname)}))
```

Compojure решает проблему вложенных адресов. Рассмотрим приложение для учёта товаров. По пути `/content/order/1/view` открывается карточка товара. Страница `/content/order/1/edit` показывает форму редактирования этого товара. Чтобы его обновить, нужно отправить форму по тому же адресу, но методом POST.

Очевидно, правила пересекаются. Чтобы избежать повторов, используем макрос `context`: