Оглавление

1 Веб-разработка **9**
1.1 Основы HTTP 10
1.2 HTTP в Clojure 14
1.3 Запросы и ответы 17
1.4 Маршруты 19
1.5 Middleware 28
1.6 Файлы и ресурсы 47
1.7 Стриминг и проксирование 50
1.8 Другие библиотеки 51
1.9 Заключение 52

2 Clojure.spec **55**
2.1 Типы и классы 56
2.2 Основы spec 57
2.3 Исключения 59
2.4 Спеки-коллекции 61
2.5 Вывод значений 65
2.6 Спеки-перечисления 67
2.7 Продвинутые техники 69
2.8 Логические пути 72
2.9 Обратное действие 73
2.10 Анализ ошибок 76
2.11 Понятные ошибки 77
2.12 Парсинг 92
2.13 Разбор Clojure-кода (теория) 104
2.14 Спецификация функций 108
2.15 Повторное использование спек 112
2.16 Дополнения 113
2.17 Заключение 117

3 Исключения **119**
3.1 Основы исключений 120
3.2 Цепочки и контекст 123
3.3 Переходим к Clojure 124
3.4 Подробнее о контексте 128
3.5 Когда бросать исключения 129
3.6 Подробнее о цепочках 131
3.7 Печать исключений 133
3.8 Логирование 135
3.9 Контекст исключений 137
3.10 Сбор исключений 140
3.11 Sentry и Ring 143
3.12 Переходы по коду 145
3.13 Finally и контекстный менеджер 148
3.14 Исключения на предикатах 151
3.15 Приёмы и функции 154
3.16 Заключение 158

4 Изменяемость **159**
4.1 Общие проблемы 159
4.2 Атомы 164
4.3 Volatile 174
4.4 Переходные коллекции 177
4.5 Переменные и alter-var-root 184
4.6 Присваивание с set! 193
4.7 Изменения в контексте 196
4.8 Локальные переменные в контексте 203
4.9 Глобальные изменения в контексте 205
4.10 Заключение 210

5 Конфигурация **213**
5.1 Постановка проблемы 213
5.2 Семантика 214
5.3 Цикл конфигурации 215
5.4 Ошибки конфигурации 217
5.5 Загрузчик конфигурации 217
5.6 Подробнее о переменных среды 224
5.7 Конфигурация в среде 226
5.8 Недостатки среды 228
5.9 Переменные среды в Clojure 232
5.10 Простой менеджер конфигурации 237

5.11 Чтение среды из конфигурации 240
5.12 Короткий обзор форматов 243
5.13 Промышленные решения 248
5.14 Заключение 256

6 Системы **257**
6.1 Подробнее о системе 257
6.2 Подготовка к обзору 259
6.3 Mount 262
6.4 Component 279
6.5 Integrant 309
6.6 Заключение 320

7 Тесты **323**
7.1 Основные понятия 323
7.2 Тесты в Clojure 333
7.3 Полезные практики 340
7.4 Фикстуры 347
7.5 This is fine 355
7.6 Метки и селекторы 357
7.7 Проблема окружения 362
7.8 Тестирование веб-приложений 387
7.9 Тестирование систем 392
7.10 Интеграционные тесты 396
7.11 Другие решения 402
7.12 Заключение 407

Что дальше **411**

Предметный указатель **412**

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`:

```
(context "/content/order/:id" [order-id]
  (GET  "/view" request (order-view request))
  (context "/edit" []
    (GET  "/" request (order-form request))
    (POST "/" request (order-save request))))
```

Правила в `context` наследуют параметры из уровня выше. Это значит, обработчики `order-view`, `order-form` и `order-save` получат параметр `:order-id`.

Статичные ответы

До сих пор в качестве ответа в правилах мы писали что-то вроде `(some-handler request)`. Иногда ответ знают заранее, поэтому нет смысла выносить его в функцию. Рассмотрим подход на примере `healthcheck`-обработчика.

Современные приложения запускают в контейнерах и облачных сервисах. Чтобы узнать, работает приложение или нет, специальная служба опрашивает его. Простой способ сделать это — послать приложению `GET`-запрос по адресу `/health` и проверить статус. Тело и заголовки ответа не играют роли.

Чтобы не создавать лишний обработчик `page-health` с постоянным ответом, поместим ответ в правило:

```
(ANY "/health" _ {:status 200 :body "ok"})
```

Можно сделать проще: в Compojure предусмотрен случай, когда выражение — строка. Она становится телом положительного ответа:

```
(ANY "/health" _ "ok")
```

1.4.4 Роутинг с Bidi


Bidi

Библиотека Bidi[9] строит маршруты по-другому. Она опирается на данные — списки и словари. Метод состоит из нескольких шагов.

На первом этапе объявляют дерево маршрутов. Это комбинация векторов и словарей по особым правилам. В листьях дерева

[9] github.com/juxt/bidi

помещают теги — уникальные метки. Специальная функция принимает дерево и запрос. Она ищет ветку, которой подходят метод и путь запроса. Если ветка нашлась, результатом будут её тег и параметры пути, например:

```
{:route :show-user :route-params: {:id 1}}
```

На втором этапе пишут middleware — промежуточный обработчик запроса. Он принимает запрос, добавляет в него тег и передаёт дальше по цепочке.

На третьем этапе добавляют обработчик запроса. Это уже не функция, а мультиметод. Его функция-диспетчер ищет в запросе тег. Метод с тегом `:default` вернёт ответ `404`, `:show-user` — страницу пользователя и так далее.

На первый взгляд схема кажется сложной. Но однажды настроив, её легко масштабировать. Чтобы сервер подхватил новый путь, в дерево добавляют ветку и расширяют мультиметод.

Перепишем на Bidi пример из начала главы. Добавьте зависимость:

```
[bidi "2.1.5"]                   ;; project.clj
(:require [bidi.bidi :as bidi]) ;; namespace
```

Начнём с дерева маршрутов. Вариант со страницами `page-index`, `page-hello` и `page-404` выглядит так:

```
(def routes
  ["/" {""      :page-index
        "hello" :page-hello
        true    :not-found}])
```

Проверим поиск пути по этому дереву. Функция `match-route` принимает маршруты, путь и возвращает словарь с тегом:

```
(bidi/match-route routes "/hello")
{:handler :page-hello}

(bidi/match-route routes "/test")
{:handler :not-found}
```

Чтобы объединить тег с запросом, используем функцию `match-route*` (со звёздочкой). Это альтернативная версия `match-route`, которая принимает словарь-накопитель. В качестве накопителя укажем запрос. Вызов `match-route*` с запросом добавит в него поле `:handler`.

```
(let [request {:request-method :get
               :uri "/test"}]
  (bidi/match-route* routes (:uri request) request))

{:request-method :get
 :uri "/test"
 :handler :not-found}
```

Теперь обернём код в middleware. Получив запрос, новый обработчик снабдит его тегом и вызовет исходный обработчик.

```
(defn wrap-handler [handler]
  (fn [request]
    (let [{:keys [uri]} request
          request* (bidi/match-route* routes uri request)]
      (handler request*))))
```

Мы ещё не касались техники middleware, но вынуждены применить её сейчас. Ниже мы рассмотрим в деталях, как она работает и почему так важна.

Проверим `wrap-handler` на скорую руку. Для удобства обернём функцию `identity`, которая вернёт переданный в неё аргумент:

```
(def wrapped (wrap-handler identity))

(wrapped {:request-method :get
          :uri "/hello?foo=42"})

{:request-method :get
 :uri "/hello?foo=42"
 :handler :page-hello}
```

Новый обработчик запроса будет мультиметодом. Его функция-диспетчер — просто ключ `:handler`.

```
(defmulti multi-handler
  :handler)

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

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

Обернув `multi-handler` с помощью `wrap-handler`, получим финальное приложение. Запустите сервер и проверьте результат в браузере.

```
(def app (wrap-handler multi-handler))
```

Это был простой роутинг на Bidi. Теперь рассмотрим товары, их просмотр и изменение. Новое дерево выглядит так:

```
(def routes
  [["/content/order/" :id] {"/view" {:get  :page-view}
                            "/edit" {:get  :page-form
                                     :post :page-save}}])
```

В этой версии листья — уже не теги, а словари. Ключ каждого словаря — это метод, а значение — тег. Запрос `GET /content/order/1/edit` разрешается в тег `:page-form`, а POST с таким же адресом — в `:page-save`. Обратите внимание на параметр `:id` в левой части. По аналогии с Compojure, это сигнал о том, что в пути находится параметр. Если был хотя бы один параметр, вместе с `:handler` запрос получит поле `:route-params`. В нашем случае это словарь `{:id "1"}`.

Расширим мультиметод страницами. `Page-view` находит товар по номеру и верстает его HTML-страницу. Если товара нет, вернём ответ 404 «не найдено». Для краткости опустим проверку параметра.

```
(defmethod multi-handler :page-view
  [request]
  (if-let [order (some-> request
                         :route-params
                         :id
                         Integer/parseInt
                         get-order-by-id)]
    {:status 200
     :headers {"content-type" "text/html"}
     :body (render-order-page {:order order})}
    page-404))
```

Страница `:page-form` строит форму редактирования. От просмотра она отличается шаблоном HTML: главное место на странице занимает тег `<form>` с полями ввода и кнопкой `<submit>`. Обновление товара сложнее: нужно выбрать из запроса данные и записать их в базу. По аналогии с примером выше опустим валидацию полей:

```
(defmethod multi-handler :page-save
  [request]
  (let [{:keys [params route-params]} request
        {order-id :id} route-params
        fields [:title :description :price]
        params (select-keys params fields)
        location (str "/content/order/" order-id "/view")]
    (jdbc/update! *db* :orders params ["id = ?" order-id])
    {:status 302
     :headers {"Location" location}}))
```

Обратите внимание: на изменение данных мы отвечаем не страницей, а перенаправлением (редиректом) на неё. Если страница пришла в ответ на POST-запрос, то при её обновлении браузер снова отправит форму. Это чревато странным поведением на сервере. Вариант с редиректом решает проблему: браузер загрузит страницу через GET, и побочных эффектов при обновлении не будет.

1.4.5 Выбор между Compojure и Bidi

Новичку в Clojure будет легче начать с Compojure. У библиотеки достойная документация с примерами. Compojure и Ring написал один автор, поэтому проекты близки и дополняют друг друга.

Маршруты Bidi сложны для понимания: они многословны и не интуитивны. В них легко перепутать вектор и словарь. С другой стороны, у мультиметодов свои преимущества. Код становится линейным, приложение легче расширять.

Если проект небольшой, выбирайте Compojure. Когда проект сложный и маршрутов всё больше, рассмотрите переезд на Bidi.

1.5 Middleware

Выше мы упоминали middleware и даже кинули пробный шар — написали `wrap-handler` (с. 25). Теперь изучим их внимательней. Это самый важный раздел в главе.

В переводе с английского термин middleware означает «промежуточный слой, середина». В программировании так называют код в роли посредника. Он выполняет предварительные шаги перед основным: приводит типы, проверяет права доступа, пишет логи.

Паттерн «декоратор» — частный случай middleware. Декоратор — это функция `A`, которая принимает функцию `B` и возвращает функцию `C`. Говорят, что `A` декорирует `B`. В ходе работы `C` вызывает `B`, но с изменениями. Например, дополняет входные или выходные данные `B`.

Рассмотрим простые декораторы. `Wrap-echo` добавляет к функции побочный эффект: печатает аргументы и результат.

```
(defn wrap-echo [func]
  (fn [& args]
    (apply println "The args are" args)
    (let [result (apply func args)]
      (println "The result is" result)
      result)))
```

Вот как им пользоваться:

```
(def +echo (wrap-echo +))
(+echo 1 2 3)
;; The args are 1 2 3
;; The result is 6
6
```

`Wrap-catch` оборачивает функцию в `try` и `catch`. Если произошло исключение, результатом будет его объект.

```
(defn wrap-catch [func]
  (fn [& args]
    (try
      (apply func args)
      (catch Throwable e
        e))))

(def div-safe (wrap-catch /))

(type (div-safe 0 0))
;; java.lang.ArithmeticException
```

Мы уже видели примеры, как устроен запрос в Ring. Возможно, читатель заметил, что у него нет полей, которые встречаются в других языках. Например, класс `HttpRequest` в Django содержит свойство `.params`. Это словарь параметров из адресной строки или тела запроса.

В Ring запрос несёт только базовую информацию. Почему в нём нет столь важных вещей? Потому что не каждое приложение в них нуждается. Представим, что на каждый запрос Ring парсит адресную строку и тело. Это удобно разработчику, но замедляет код. Возможно, что параметры не нужны в запросе, но сервер потратит на них время.

То же самое с разбором тела: это дорогая операция. Представим, что пришёл запрос с большим JSON-документом. Мы считали его, но затем оказалось, что у пользователя нет прав на запись. Время и ресурсы потрачены зря: нужно было проверить доступ до чтения.

Нужен механизм, чтобы задать предварительные шаги и их порядок. Он называется middleware и широко используется в вебе. В Ring эта техника играет особую роль. Параметры запроса,

сессии, куки, права доступа — всё это функция, которая возвращает функцию. Вам не придётся писать все middleware с нуля: Ring содержит наиболее важные, нужно только подключить их. Рассмотрим основные middleware и принципы их работы.

1.5.1 Параметры запроса

HTTP предусматривает данные в адресной строке. Это пары вида `name=John&city=NY` после знака вопроса. В коде они становятся словарём вида `{:name "John" :city "NY"}`.

В POST-запросах параметры помещают в тело. Так поступают из-за ограничения на длину адреса и безопасности. Длина адреса не превышает 2048 байт, а на тело запроса ограничений нет. Логины и пароли нельзя пересылать в адресе, потому что они остаются в логах и истории браузера.

Функция `wrap-params` из модуля `ring.middleware.params` меняет обработчик следующим образом. Переданный в него запрос обретает поля:

- `:query-params` — словарь параметров из адреса;
- `:form-params` — словарь данных из тела запроса;
- `:params` — их комбинированная версия.

Пусть `app` — ваше веб-приложение. Чтобы получить обёрнутую версию, передайте его в `wrap-params`. Результат функции и будет финальным приложением. На жаргоне разработчиков это называется «врапнуть» (англ. wrap — «обернуть»).

```
(require '[ring.middleware.params :refer [wrap-params]])
(def final-app (wrap-params app))
```

Чтобы не запутаться в именах, придерживайтесь правил: исходное приложение называйте `app-naked` или `app-raw` (голое, сырое), а финальное — просто `app`.

```
(def app (wrap-params app-naked))
```

Доработайте приложение так, чтобы оно учитывало параметры. Например, пусть параметр `who` задаёт имя, кого приветствовать: `/hello?who=John`. Добраться до параметра можно так:

```
(defn page-hello [request]
  (let [who (get-in request [:params "who"])]
    ...))
```

Обратите внимание, что в ключах `:params` находятся строки. Адрес — это строка, и разбор параметров режет её на части. Однако Clojure поощряет нас использовать в словарях кейворды. Исправим это: в поставке Ring идёт middleware, которое приводит `:params` к удобному виду. С ним ключи примут тип `Keyword`.

```
(require '[ring.middleware.keyword-params
           :refer [wrap-keyword-params]])

(def app (wrap-keyword-params
           (wrap-params app-naked)))
```

Поскольку кейворд — это функция, до параметра можно добраться стрелочным оператором как в первой строке. Некоторым этот способ нравится больше, чем `get-in`.

```
(-> request :params :who)
;; or
(get-in request [:params :who])
```

1.5.2 Стек middleware

Типичный проект на Clojure включает около десяти middleware. Как только мы начнём нанизывать их на приложение, появится лесенка:

```
(def app
  (wrap-something-else
    (wrap-current-user
      (wrap-session
        (wrap-keyword-params
          (wrap-params app-naked))))))
```

Если добавить звено посередине, оно каскадом сдвинет элементы ниже. Чтобы победить вложенность, сделаем структуру линейной. Поможет стрелочный оператор:

```
(def app
  (-> app-naked
      wrap-params
      wrap-keyword-params
      wrap-session
      wrap-current-user
      wrap-something-else))
```

При компиляции запись превратится в первый вариант, поэтому логика программы не меняется. Однако мы сделали код удобнее: получился список, который легко читать и поддерживать. Назовём его *стеком* middleware.

Запись в стрелочном виде имеет особенность. Не заглядывая дальше, догадайтесь, в каком порядке выполняются middleware. Правильный ответ: снизу вверх для запроса и сверху вниз для ответа. Это станет ясно при мысленном разборе и аналогии.

Обёртка функциями напоминает упаковку настоящих предметов. Подобно тому, как мы накладываем на приложение middleware, товар покрывают плёнкой, затем заводской коробкой и гофрокартоном для перевозки. Чтобы добраться до изделия, мы снимаем упаковку в обратном порядке: картон, коробка, плёнка. Если представить, что изделие посылает сигналы во внешний мир, сигнал проходит обёртки в прямом порядке: плёнка, коробка, картон. В коде выше `wrap-something-else` применили в последнюю очередь, поэтому оно играет роль картона — внешней обёртки.

Сперва запрос попадёт в `wrap-something-else`. Код внутри него вызовет обработчик, который получен из `wrap-current-user`. Обработчик внутри него — результат `wrap-session` и так далее. Вершиной подъёма станет `app-naked`. В нём сработает основная логика приложения.

Теперь ответ спускается в обратном порядке. Сначала он пройдёт через `wrap-params` и `wrap-keyword-params`. Эти два middleware не меняют ответ и просто вернут его. `Wrap-session` и `wrap-current-user`, возможно, дополнят его заголовки. `Wrap-something-else` сработает последним. Цикл пройден.

Представьте стек middleware как восхождение в гору и спуск с неё. Похоже устроены middleware в Django — промышленном фреймворке на Python. Их роль играют классы, а не функции, но схема обхода такая же.

Порядок middleware порой критичен. Некоторые из них опираются на данные из предыдущих middleware. Рассмотрим уже знакомые `wrap-params` и `wrap-keyword-params`. Последнее ищет в запросе поле `params` и меняет тип ключей. Происходит разделение труда: одно middleware готовит данные, второе улучшает их. Поэтому `wrap-keyword-params` ставят строго после `wrap-params`.

Теперь посмотрим на форму `(def app ...)` выше. В неё закралась **ошибка**. Запрос движется снизу вверх, поэтому `wrap-keyword-params` сработает раньше (строка 4). Он попытается найти `:params` в запросе, но безуспешно. Далее сработает `wrap-params` (строка 3). Он заполнит поле словарём из адресной строки. В результате у `:params` ключи будут строками. Чтобы исправить ошибку, поменяйте `wrap-params` и `wrap-keyword-params` местами.

Неверный порядок middleware стоит часов отладки. Но есть трюк: если два и более middleware зависят друг от друга, их можно «схлопнуть» в одно целое. Функция `comp` принимает функции и возвращает суперфункцию, которая применяет их к аргументу. Напишем умный враппер параметров:

```
(def wrap-params+
  (comp wrap-params wrap-keyword-params))
```

Разберёмся, почему аргументы `comp` идут в таком порядке. Обозначим их `foo` и `bar` и перепишем выражение несколько раз. В каждом столбике одинаковая функция, записанная по-разному.

```
(comp foo bar)   (fn [x]            (fn [x]
                   (foo (bar x)))     (-> x bar foo))
```

Если `x` в третьем столбике — приложение, а `foo` и `bar` — middleware, то всё станет ясно. Во время запроса они работают в обратном порядке, поэтому `foo` запустится раньше `bar`. Значит, на месте `foo` должно быть `wrap-params`, а вместо `bar` — `wrap-keyword-params`. Если подставить их в первый столбик, получим то, что записали вначале.

Плюс на конце означает, что это улучшенная версия `wrap-params`. Заменим в стеке `wrap-params` и `wrap-keyword-params` на версию с плюсом. Цепочка станет короче, а логика параметров поселилась в отдельном месте. Далее мы рассмотрим другие полезные middleware: куки, сессии и JSON.

1.5.3 Cookie

В HTTP куки — это маленькие кусочки информации. Между сервером и браузером соглашение о том, как хранить и передавать их. Если сервер выставил куки, браузер запомнит их для этого сайта. В следующий раз браузер отправит их на сервер автоматически. Так продолжается до тех пор, пока не истечёт их срок жизни или кто-то не удалит их. Удалить куки может как сервер с помощью заголовка, так и пользователь в настройках браузера.

Простейший случай, когда нужны куки, — определить, был ли уже пользователь на сайте. В первый раз приложение ищет в запросе куки с именем `seen`. Если их нет, сервер выставит заголовок:

```
Set-Cookie: seen=true;
```

Получив куки, браузер добавит их ко всем запросам к серверу. Исходящий заголовок выглядит так же, но без частички `Set-` в имени. Приложение проверяет: если поле `seen` истинно, клиент уже был на сайте. Это влияет на показ рекламы, всплывающие окна и так далее.

Технически куки — это длинный заголовок, где поля и значения разделены точками с запятой. Middleware `wrap-cookie` упрощает работу с этим заголовком. В запросе мы получим словарь `:cookies`, в котором два уровня с именами и атрибутами. Чтобы выслать клиенту новые куки, добавьте изменённую копию словаря в ответ. Ring построит из него заголовок `Set-Cookie`.

Ниже страница `page-seen` проверяет, видим ли мы её в первый раз.

```
1  (require '[ring.middleware.cookies
2             :refer [wrap-cookies]])
3
4  (defn page-seen [request]
5    (let [{:keys [cookies]} request
6          seen-path ["seen" :value]
7          seen? (get-in cookies seen-path)
8          cookies* (assoc-in cookies seen-path true)]
9      {:status 200
10      :cookies cookies*
```

```
11      :body (if seen?
12              "Already seen."
13              "The first time you see it!")}))
14
15  (def app (-> page-seen
16               wrap-cookies))
```

Замечание: переменная со звёздочкой на конце означает новую версию исходной переменной (строка 8). Например, такой же словарь, но с новым ключом. Вместо звёздочки иногда ставят штрих. Имя `cookies*` читается как «новые куки на базе старых».

По аналогии с `:params` ключи куки — это строки. Если поменять `"seen"` на `:seen` (строка 6), вы промахнётесь и получите `nil`. В Ring нет аналога `wrap-keyword-cookie`, но такую обёртку легко написать самому. Ниже мы покажем способ для заголовков.

Запустите приложение в браузере. После обновления страницы надпись изменится на Already seen. Она останется даже после перезагрузки сервера, потому что флаг `seen` хранит браузер. Только очистив куки, вы увидите: The first time you see it. Для полноты эксперимента откройте приватную вкладку или другой браузер.

Куки тесно связаны с безопасностью. Убедитесь, что они подписаны ключом, защищены от кражи и не раскрывают секретные данные (пароли, ключи доступа).

Атрибуты `:http-only` и `:secure` существенно снижают риск кражи. Первый означает, что к куки нельзя обратиться из JavaScript. Этим вы защититесь от вредоносных скриптов, которые читают куки и отправляют на чужой сервер. Откройте консоль разработчика в браузере и выполните выражение:

```
> document.cookie
// ring-session=...; seen=true
```

Измените переменную `cookies*` так, чтобы у `seen` был атрибут:

```
(let [cookies* (assoc cookies "seen"
                      {:value true :http-only true})]
  ...)
```

Теперь JavaScript не увидит значение `seen`:

```
> document.cookie
// ring-session=...
```

Атрибут `:secure` означает, что куки передаются только в безопасном соединении по SSL. На время разработки им можно пренебречь, но в боевом запуске `:secure` всегда истина. Приложения без SSL небезопасны, и браузеры указывают на это пользователям.

Мы не будем задерживаться на веб-безопасности: тема слишком обширна и заслуживает отдельной книги.

1.5.4 Сессии

HTTP не предполагает связи между двумя запросами. Серверу неважно, откуда они пришли — с соседних машин или разных континентов. Запросы нельзя связать по времени: кто-то читает страницу час, а другие обновляют каждую минуту.

Разработчики пошли на уловку. Даже если клиенты сидят за одним столом, приложение выдаст им куки с длинной случайной строкой. Поле с этой строкой называют идентификатором. Браузер добавляет его к исходящим запросам, а сервер группирует запросы по идентификатору. Технику назвали *сессией*, или *сеансом*.

Под сессией понимают значения, связанные с текущим пользователем. Например, выбранный язык, состояние виджетов, просмотренные товары. Важно, что пользователь не обязательно авторизован. Это может быть аноним, но сервер отличит его запросы от других анонимов.

Обёртка `wrap-session` выражает процесс в терминах Clojure. Это сложное middleware, которое дополняет запрос словарём `:session`. Его ключи — поля сессии. Чтобы обновить сессию, её новую версию пишут в ответ по аналогии с куки. Middleware различает `nil` и отсутствие сессии в ответе. Если поле `nil`, вся сессия удаляется. Если ключа нет, ничего не происходит.

Различают *бэкенды* сессии, способы хранить её физически. Это может быть память, диск, база данных, системы Memcached и Redis или даже куки. При выборе бэкенда учитывайте, может ли он работать на нескольких машинах одновременно. Что получится, если каждый запрос случайно уходит на одну из десяти машин? Когда сессию хранят памяти, на каждой машине будет её разная

копия. Это чревато странным поведением и отладкой. Аналогично с файлами — машины не делят их между собой. Наоборот, база данных или Redis предлагают централизованный доступ к данным. Они гарантируют целостность сессии для всех клиентов.

Интересно, что сессия в куки тоже работает на многих машинах. В этом случае её хранит браузер. Клиент и сервер обмениваются полной сессией в заголовках. У решения есть минусы: если пользователь очистит куки или запустит другой браузер, сессия будет утеряна. Кроме того, слишком большая сессия засоряет HTTP-запрос и увеличивает трафик.

Из коробки Ring предлагает два бекенда сессии: память и куки. Тип хранилища задают настройками `wrap-session`. Если ничего не указывать, Ring использует память. Чтобы подружить сессию с Redis или другой системой, расширьте протокол `SessionStore`. Работа с протоколами выходит за рамки этой главы, поэтому ограничимся ссылкой на документацию Ring[10] с примером.


Ring Sessions

Рассмотрим пример со счётчиком посещений. Будем считать, сколько раз пользователь зашёл на сайт. Сессию храним в памяти.

```
(require '[ring.middleware.session
           :refer [wrap-session]])

(defn page-counter [request]
  (let [{:keys [session]} request
        session* (update session :counter (fnil inc 0))
        counter (:counter session*)]
    {:status 200
     :session session*
     :body (format "Seen %s time(s)" counter)}))

(def app (-> page-counter
             wrap-session))
```

Запустите приложение и откройте браузер. Обновите страницу, и счётчик в сообщении увеличится с каждым просмотром. Проделайте то же самое в другом браузере. Это будет вторая сессия, которая не зависит от первой. Убедитесь, что просмотр в первом браузере не влияет на второй. Данные лежат в памяти, поэтому они потеряются при новом запуске сервера.

[10] github.com/ring-clojure/ring/wiki/Sessions

Упражнение 1. Выше мы считаем просмотры для всего сайта. Сделайте так, чтобы счётчик работал в разрезе страниц. Например, главную страницу `/` смотрели пять раз, а справку `/help` — три раза. Параметры запроса не влияют на подсчёт.

Упражнение 2. Поскольку сессия хранится в памяти, она потеряется при перезапуске сервера. Изучите документацию Ring и сделайте так, чтобы сессия хранилась в куках. Убедитесь, что перезапуск не влечёт потерю данных.

1.5.5 JSON

Формат JSON служит для передачи данных. Он различает базовые типы: числа, строки, логический тип и коллекции любой вложенности. Это значимый плюс по сравнению с INI или XML, где все значения — строки. JSON совместим с JavaScript: если передать строку в функцию `eval`, она вернёт комбинацию массивов и объектов. Эти и другие причины сделали формат популярным. Сегодня JSON — главный способ передать данные в вебе.

Ring предлагает несколько middleware для JSON. Для удобства их поместили в отдельную библиотеку. Добавьте зависимость:

```
[ring/ring-json "0.4.0"]
```

Middleware `wrap-json-response` упрощает возврат JSON-данных из обработчика. Оно проверяет поле ответа `:body`: если это коллекция, её меняют на кодированную строку и добавляют заголовок `Content-Type` с типом `application/json`.

Рассмотрим API для чтения пользователя. Если нашли его по номеру, вернём словарь полей. Если нет, в ответе структура ошибки.

```
(defn page-data [request]
  (let [user-id (-> request :params :id)]
    (if-let [user (get-user-by-id user-id)]
      {:status 200 :body user}
      {:status 404
       :body {:error_code "MISSING_USER"
              :error_message "No such user"}})))
```

В обоих случаях не нужно кодировать данные вручную — это делает `wrap-json-response`.

```
(require '[ring.middleware.json
           :refer [wrap-json-response]])


(def app (-> page-data
             wrap-params+
             wrap-json-response))
```

Запустите `app` и переключитесь в браузер. Откройте вкладку Network в панели разработчика. Изучите запрос к серверу, в особенности заголовки ответа и его тело. Если это был JSON, браузер распарсит данные и покажет в виде дерева.

Для входящего JSON служат два middleware: `wrap-json-body` и `wrap-json-params`. Оба проверяют, что заголовок `Content-Type` равен `application/json`. Если заголовок верный, они парсят тело с учётом возможных исключений. При ошибке разбора получим статус 400 и текст «JSON body malformed».

Разница между middleware в том, куда они складывают данные. `Wrap-json-body` заменяет поле `:body` запроса на структуру данных. Ниже обработчик `page-body` извлекает имя и город из `:body`. Тело запроса — уже не входящий поток, а словарь. Обратите внимание: middleware принимает дополнительные параметры. Флаг `:keywords? true` означает, что ключи словаря станут кейвордами.

```
(require '[ring.middleware.json
           :refer [wrap-json-body]])

(defn page-body [request]
  (let [{:keys [body]} request
        {:keys [username city]} body]
    (create-user username city)
    {:status 200
     :body {:code "CREATED"
            :message "User created"}}))

(def app (-> page-body
             (wrap-json-body {:keywords? true})))
```


Postman

Чтобы отправить JSON серверу, нужна специальная программа. Подойдёт консольная утилита `cURL` или графическое приложение Postman[11]. Пример с `cURL` (для чистоты отбросим обратные слеши в переносах строк):

```
curl --request POST
     --header "Content-Type: application/json"
     --data '{"username":"John", "city":"NY"}'
     http://localhost:8080/
```

`Wrap-json-params` работает с небольшим отличием — оно пишет данные в поле `:json-params`. Далее, если в теле был словарь, он дополняет `:params`. Объясним, в чём смысл этой логики.

Поле `:params` — это общий аккумулятор параметров. Его наполняют и другие обёртки, например `wrap-params`. Некоторые API не зависят от метода и принимают запросы одновременно через GET или POST. Соответственно, данные находятся в адресной строке или теле запроса.

У гибридного способа преимущество: запросы на чтение выполняют через GET, чтобы применить кэширование на уровне HTTP. Запросы на изменение выполняют через POST, который никогда не кэшируется. Однако в обработчике нас не интересует, какой был метод: мы просто работаем с `:params`.

В других API важно, чтобы параметры не смешивались и не заменяли друг друга. Предположим, наш сервис работает строго по методу POST. Чтобы прочесть только те данные, что пришли в JSON, обратимся к полю `:json-params` или `:body`. Каким враппером пользоваться — `wrap-json-body` или `wrap-json-params`, зависит от конкретного случая.

Заметим, что `:params` — это словарь с ключами-строками. Чтобы слияние прошло правильно, тоже оставляет ключи строками. Исправьте ключи с помощью `wrap-keyword-params` уже после слияния.

Middleware не случайно выделяет поле `:json-params`. Дело в том, что данные в JSON не всегда словарь: это может быть массив, который нельзя объединить с `:params`. На этот случай данные хранят в отдельном поле.

[11] www.postman.com

1.5.6 Свои middleware

До сих пор мы использовали наработки из Ring и смежных библиотек. Рано или поздно вам потребуются свои middleware. Рассмотрим примеры из реальных проектов.

Ключи заголовков

Наш первый враппер обновляет заголовки запроса — меняет тип ключей со строк на кейворды. Заголовкам меняют тип, когда приложение часто к ним обращается. В этом случае удобней переключиться на кейворды, чтобы сохранить синтаксис `:keys` и не ошибиться с типом ключа.

В ответе ключи заголовков тоже должны быть строками. Если указать кейворд, Jetty бросит исключение — его код ожидает строки. Добавим обратное действие: изменим ключи с кейвордов на строки. Обе задачи решают функции `keywordize-keys` и `stringify-keys` из модуля `clojure.walk`. Они рекурсивно обходят коллекцию и меняют тип ключей.

```
(require '[clojure.walk :refer
           [keywordize-keys stringify-keys]])

(defn wrap-headers-kw [handler]
  (fn [request]
    (-> request
        (update :headers keywordize-keys)
        handler
        (update :headers stringify-keys))))
```

Внутри `handler` приложение работает с заголовками как с кейвордами. В примере мы находим заголовок `host` с помощью `:keys`, что не сработало бы для строк. В заголовках ответа тоже кейворд. Браузер покажет сообщение жирным шрифтом, потому что тип содержимого HTML, а не плоский текст.

```
(defn app* [request]
  (let [{:keys [headers]} request
        {:keys [host]} headers]
```

```
    {:status 200
     :headers {:content-type "text/html"}
     :body (format "<h1>Host header: %s</h1>" host)}))

(def app (wrap-headers-kw app*))
```

По аналогии можно сменить тип ключей для словаря куки или сессии.

Идентификатор запроса

По умолчанию запрос и ответ не связаны друг с другом. Порой трудно понять, к какому запросу относится данный ответ и наоборот. Предположим, мы увидели в логах ответ с кодом 500, но какой именно запрос вызвал ошибку? Важно, чтобы система могла их сопоставить.

Проблему решает заголовок `X-Request-Id`. Чаще всего это случайный идентификатор (или UUID), строка из 36 символов. Для краткости его называют «айди» (англ. id). Если клиент не передал id в запросе, ему назначат случайный. Тот же id вернётся в ответе. Идентификатор пишут в лог, чтобы по запросу построить цепочку событий.

Напишем обёртку для id. Считаем, что заголовки — это кейворды, потому что выше по стеку находится наш `wrap-headers-kw`.

```
(import 'java.util.UUID)

(defn wrap-request-id [handler]
  (fn [request]
    (let [path [:headers :x-request-id]
          uuid (or (get-in request path)
                   (str (UUID/randomUUID)))]
      (-> request
          (assoc-in path uuid)
          (assoc :request-id uuid)
          handler
          (assoc :request-id uuid)
          (assoc-in path uuid)))))
```

Обратите внимание: id не только хранят в заголовках, но дублируют в запросе и ответе в поле `:request-id`. К нему часто обращаются в логах, поэтому удобно вынести его в переменную в начале функции:

```
(defn some-handler [request]
  (let [{:keys [params request-id]} request]
    (log/info "Request id: %s" request-id)))
```

Текущий пользователь

Этот враппер добавляет в запрос текущего пользователя. В данном случае мы ищем его номер в сессии, но возможны варианты с куки или параметром запроса. Если номер найден, читаем пользователя из базы и добавляем к запросу. Функция `get-user-by-id` вернёт словарь или `nil`. Её оборачивают в `(when user-id...)`, чтобы не обращаться в базу с пустым номером.

```
(defn wrap-current-user [handler]
  (fn [request]
    (let [user-id (-> request :session :user-id)
          user (when user-id
                 (get-user-by-id user-id))]
      (-> request
          (assoc :user user)
          handler))))
```

Middleware ниже по стеку и обработчик читают пользователя из поля `:user` запроса. В следующем разделе мы рассмотрим подходящий пример.

К вопросу о безопасности: номер пользователя *можно* хранить в сессии. Она подписана секретным ключом, поэтому только сервер знает, как её изменить. Номер пользователя не раскрывает приватные данные. Но не храните в сессии пароли и ключи доступа.

1.5.7 Прерывание стека

До сих пор мы работали с цепочкой middleware, где каждое звено передаёт управление следующему. Цепь не всегда линейна:

иногда её нужно прервать. Предположим, в одном из middleware мы поняли, что у пользователя нет прав. Продолжать не имеет смысла — наоборот, как можно скорее оборвём стек.

Middleware часто содержат условия. Например, `wrap-json-params` читает тело только если заголовок `Content-Type` верного типа. Когда в нём что-то другое, враппер ничего не делает. Разбор JSON бросит исключение, если документ повреждён из-за сбоя в сети. В этом случае `wrap-json-params` не продолжит цепочку. Оно вернёт ответ 400 JSON body malformed, и ни одно middleware ниже по стеку не сработает.

Пусть приложение доступно только по авторизации. С помощью `wrap-current-user` получим текущего пользователя. Это middleware только находит пользователя, но не ограничивает доступ. Добавим ещё одно:

```
(defn wrap-auth-user-only [handler]
  (fn [request]
    (if (:user request)
      (handler request)
      {:status 403
       :headers {"content-type" "text/plain"}
       :body "Please sign in to access this page."})))
```

Переход к следующему middleware находится под условием. Если пользователь не авторизован, звенья ниже `wrap-auth-user-only` будут отброшены.

Мы уже говорили, что цепочка middleware — это подъём в гору и спуск с неё. Если звено терпит неудачу, мы как будто срезаем верхушку: добрались до середины, столкнулись с проблемой и повернули обратно. *Общее правило:* чем раньше мы обнаружим проблему, тем меньше потратим ресурсов. Поэтому более общие проверки ставят выше по стеку (или ниже в операторе $\rightarrow$).

Ещё один вариант развилки — перехват ошибок. Это важный обработчик, но его нет в поставке Ring, потому что реакция на ошибки зависит от многих факторов. Как правило, враппер копируют из проекта в проект с небольшими изменениями.

Что случится, если при обработке запроса возникнет исключение? На этот счёт нет чётких правил: каждый фреймворк ведёт себя по-разному. Один покажет стек-трейс в браузере, другой вернёт HTML с информацией для отладки. Разработчики третьего

посчитали, что показывать эти данные небезопасно. Исключение пишут в лог, а клиенту вернут нейтральную фразу об ошибке.

Полезно, когда мы сами определяем, что делать с исключением. Ниже middleware, которое ловит ошибку, пишет её в лог и возвращает ответ-заглушку:

```
(defn wrap-exception [handler]
  (fn [request]
    (try
      (handler request)
      (catch Throwable e
        (let [{:keys [uri request-method]} request]
          (log/errorf e "Error, method %s, path %s"
                      request-method uri)
          {:status 500
           :headers {"content-type" "text/plain"}
           :body "Sorry, please try later."})))))
```

Выражение `log/errorf` — это макрос для записи ошибки. Он принимает исключение, шаблон сообщения и параметры подстановки. Важно знать, какие были метод и путь запроса, поэтому запишем их тоже. Это облегчит анализ логов в будущем.

Чем выше `wrap-exception` в стеке, тем меньше у исключения шансов дойти до пользователя. В идеале оно стоит на вершине цепочки, чтобы ловить все исключения.

Иногда используют двойной перехват, потому что ошибки в разных частях системы заслуживают разного подхода. Об ошибках в бизнес-логике важно знать всё. Если пользователь не смог купить товар, запишем весь контекст, который был на момент покупки. Но если пришёл повреждённый JSON, это техническая проблема, не связанная с бизнесом. Исключение здесь — это норма, потому его не пишут в лог.

Чтобы разделить бизнес- и технические проблемы, на границах стека расставляют разные `wrap-exception`. Самое нижнее оборачивает `app-naked`: оно ловит исключения в бизнес-логике. Такую ошибку пишут в журнал максимально подробно. На вершине стека другая, облегчённая версия `wrap-exception`. Оно подавляет технические проблемы на ранних этапах запроса. Его задача — вернуть адекватный ответ и не засорять журнал.

1.5.8 Middleware вне стека

Интересен сценарий, когда middleware влияет на запросы по определённому пути. В чём недостаток `wrap-auth-user-only`? Если добавить его в стек, анонимный пользователь не увидит ничего: каждый запрос завершится с кодом 403. Главная страница, контактные данные, форма входа — всё будет недоступно. Кому нужно такое приложение?

Очевидно, проверка должна касаться только части запросов. Например, тех, что начинаются с префикса `/account`, а именно `/account/cart`, `/account/orders` и других. Место `wrap-auth-user-only` не в общем стеке, а ниже — на уровне маршрута.

Реализация зависит от того, как мы строим маршруты. Compojure предлагает middleware под названием `wrap-routes`. Оно принимает правило и *другое* middleware. Последнее сработает только в том случае, если правило подходит запросу. Столь сложная логика нужна, чтобы не вызвать middleware, пока запрос не совпадет с правилом.

Построим отдельные маршруты с личной информацией:

```
(defroutes app-account
  (GET "/cart"    _ "cart")
  (GET "/orders"  _ "orders")
  (GET "/profile" _ "profile"))
```

Смонтируем их на префикс `/account` и обернём проверкой доступа.

```
(defroutes app
  (GET "/"     _ "index")
  (GET "/help" _ "help")
  (context "/account" []
    (wrap-routes app-account wrap-auth-user-only)))
```

Теперь `wrap-auth-user-only` сработает только для путей, которые начинаются с `/account`. На главной и справочной страницах проверки доступа не будет.

Middleware, которое принимает middleware, — довольно крутая абстракция. Если вы поняли, как оно работает, примите поздравления: это серьёзный рубеж.

1.6 Файлы и ресурсы

До сих пор мы возвращали в ответе строки и коллекции. Рассмотрим случай, когда данные находятся в файле. Предположим, мобильное приложение запрашивает адреса банкоматов и отделений. Банк хранит эти данные во внутренней сети, и у веб-приложения нет к ним доступа. Кроме того, новые отделения и банкоматы появляются редко, поэтому обращаться в базу на каждый запрос расточительно. Раз в неделю скрипт выгружает данные в файл и копирует на сервер, где работает веб-приложение.

Наивное решение в том, чтобы прочитать файл функцией `slurp` и вернуть его содержимое:

```
(defn page-terminals [request]
  {:status 200
   :headers {"content-type" "application/json"}
   :body (slurp "terminals.json")})
```

Недостаток в том, что мы читаем файл в память целиком. Из этого следует, что чем больше файл, тем больше ресурсов мы потребляем. Клиенты с плохой связью могут читать ответ медленно, но всё это время файл будет висеть в памяти. Кроме того, `slurp` читает из файла строку, что не подходит для двоичных файлов (картинок, PDF). В примере ниже браузер получит повреждённое содержимое.

```
(defn app [request]
  {:status 200
   :headers {"content-type" "image/png"}
   :body (slurp "/path/to/image.png")})
```

Ring допускает, чтобы телом ответа был файл, экземпляр класса `java.io.File`. Чтобы получить объект файла, путь к нему передают в конструктор класса или функцию `file` из модуля `clojure.java.io`:

```
(require '[clojure.java.io :as io])

(defn page-terminals
  [request]
```

```
  {:status 200
   :headers {"content-type" "application/json"}
   :body (io/file "terminals.json")})
```

Если тело — файл, сервер отдаёт содержимое малыми порциями по мере того, как клиент его читает. Сервер автоматически добавит заголовок `Content-Length` с размером файла.

Когда файлов несколько, плодить правила на каждый из них утомительно:

```
(defroutes app
  (GET "/terminals.json"   req (page-terminals req))
  (GET "/departments.json" req (page-departments req)))
```

Очевидно, можно сопоставить путь к файлу и директорию, где он находится. Для этого служит middleware `wrap-file`. Оно устроено так, что если запрос не подошёл основному приложению, middleware ищет файл в указанной папке.

```
(require '[ring.middleware.file :refer [wrap-file]])

(def app (-> app-naked
             (wrap-file "/var/www/public")))
```

Запрос `/terminals.json` пройдёт мимо `app`, но разрешится в одноимённый файл по пути `/var/www/public/terminals.json`. Позже в эту папку можно добавить другие файлы и ставить на них ссылки по именам: `/manual.pdf`, `/price_2020.xlsx`.

Ссылки на статичные ресурсы отделяют префиксом `/static`. Кроме того, близкие по типу файлы группируют в дочерних директориях: `/static/img` для картинок, `/static/js` для скриптов и так далее. Согласно правилу, физический путь к картинке будет `/var/www/public/static/img/logo.png`.

Кроме файлов, Ring работает с ресурсами JVM. Они похожи на файлы, но после сборки приложения становятся частью uberjar. В этом случае приложение читает ресурсы из самого себя, а не с диска.

Преимущество ресурсов в том, что приложение не зависит от сторонних файлов. Это делает его автономным и упрощает

деплой — не придётся копировать статичные файлы на сервер. С другой стороны, ресурсы нельзя обновить по требованию. Если нужно исправить логотип или стиль CSS, которые входят в ресурсы, вы должны собрать проект заново. Для ресурсов служит похожее middleware `wrap-resource`:

```
(require '[ring.middleware.resource
           :refer [wrap-resource]])

(def app (-> app-naked
             (wrap-resource "public")))
```

По аналогии с `wrap-files` оно принимает путь, относительно которого нужно искать ресурсы. Всё, что мы говорили о префиксах, справедливо и для ресурсов. Чтобы ссылка `/static/img/-logo.png` вела на ресурс, он должен быть по пути `resources/public/static/img/logo.png`. Мы подробно рассмотрим ресурсы в будущих главах (с. 374).

Если открыть браузер и ввести путь со статичным файлом, откроется окно с предложением сохранить файл на диск. Это не всегда то, что мы ожидаем. Современные браузеры умеют показывать почти все известные форматы: картинки, музыку, PDF, JSON, XML и другие. Хотелось бы увидеть файл сразу в браузере.

Окно сохранения возникает потому, что для файлов и ресурсов заголовок `Content-Type` по умолчанию равен `application/octet-stream`, то есть бинарный поток. Чтобы определить тип по расширению файла, добавьте ещё одно middleware `wrap-content-type`:

```
(require '[ring.middleware.content-type
           :refer [wrap-content-type]])

(def app
  (-> app-naked
      (wrap-content-type
       {:mime-types
         {"json" "application/json"
          "png" "image/png"
          "xls" "application/vnd.ms-excel"}})))
```

MIME types

Параметр `:mime-types` принимает словарь MIME-типов и расширений[12]. Добавьте нужные типы и проверьте, что браузер показывает файлы правильно.

1.7 Стриминг и проксирование

Телом ответа может быть в том числе входящий поток, экземпляр класса `InputStream`. Особенность потока в том, что его читают один раз. Потоки широко используются в Java, и некоторые библиотеки возвращают их напрямую. Например, при создании PDF мы получим поток с бинарным содержимым документа. Чтобы не писать поток во временный файл, направим его в ответ HTTP. Потребитель считает поток по мере загрузки в браузере.

Отдача клиенту потока называется стримингом (англ. stream — «поток»). Поток может быть огромных размеров, в том числе бесконечным. Например, поток изображения с камеры потенциально не закончится никогда.

Ещё лучше потоки подходят для проксирования. Под этим словом имеют в виду передачу данных через посредника. Предположим, внутренний сервер компании выдаёт по запросу важные данные. Мы должны предоставить их клиентам. Однако нельзя открывать им прямой доступ к ресурсу. Вместо этого мы пишем промежуточный слой, который проверяет права доступа. Если всё в порядке, мы посылаем HTTP-запрос во внутренний сервис и получаем поток. Он становится телом нашего ответа клиенту. Мы не вмешиваемся в содержимое, а только соединяем поток с нужным потребителем.

Напишем прокси-приложение, которое вернёт главную страницу Яндекса. В запросе мы обращаемся к ней по протоколу GET. В опциях передаём флаг `stream?`, что означает «не читать ответ». В этом случае тело будет потоком, а не строкой HTML. Поля `status` и `body` ответа Яндекса переходят в наш ответ. Это касается и заголовков, но мы возвращаем их не все, а подмножество (в нашем случае только `Content-Type`).

[12] en.wikipedia.org/wiki/MIME

```
(require '[clj-http.client :as client])

(defn app-proxy [request]
  (let [opt {:stream? true}
        response (client/get "https://ya.ru" opt)
        {:keys [status headers body]} response
        headers* (select-keys headers ["Content-Type"])]
    {:status status
     :headers headers*
     :body body}))
```

Запустите `app-proxy` в браузере. Вы увидите главную страницу Яндекса, хотя адрес по-прежнему `localhost`. Код `app-proxy` можно выразить короче с помощью стрелочного оператора:

```
(defn app-proxy [request]
  (-> "https://ya.ru"
      (client/get {:stream? true})
      (select-keys [:status :body :headers])
      (update :headers select-keys ["Content-Type"])))
```

Доработайте прокси, чтобы он работал с другими сайтами. Добавьте поддержку форм методом POST, опробуйте загрузку файлов. Какие ещё заголовки понадобятся вдобавок к `Content-Type`?

Задача для настоящих хакеров. Вы пишете программу для мобильного оператора. Если клиент обратился к HTML-странице по незащищённому соединению, вы должны добавить после тега `<head>` рекламный скрипт. Как это сделать в полёте, не читая страницу целиком в память или на диск?

1.8 Другие библиотеки

Для веб-разработки на Clojure написано много фреймворков и библиотек. Если возможностей Ring не хватает, обратитесь к проектам ниже.


Compojure API

- Compojure API[13] — убернадстройка над обычным Compojure. Набор макросов, чтобы декларативно описать REST API. Библиотека тесно связана с JSON-схемой и Swagger.


Luminus

- Luminus[14] — шаблон веб-приложения. Включает `Compojure API` для маршрутов, модуль базы данных, миграции и многое другое из коробки. У проекта достойная документация и сообщество, куда можно обратиться за помощью.


Pedestal

- Pedestal[15] — фреймворк компании Cognitect. Отличается гибкой системой перехватчиков (англ. interceptors), с помощью которых вложенную логику описывают линейно.


Vase

- Vase[16] — экспериментальная обёртка над Pedestal. Хранит логику приложения в файле EDN. Тесно связана с Datomic, базой данных Cognitect.


Duct

- Duct[17] — новый фреймворк от создателя Ring. Проект на ранней стадии, и документации ещё мало. Делает упор на модульность и систему компонентов (с. 257).


Liberator

- Liberator[18] — аналог проекта Webmachine для Erlang. Запрос и ответ проходят стадии, на каждую из которых можно задать реакцию. Предлагает систему правил на базе мультиметодов.

1.9 Заключение

Современный веб работает по HTTP. Это текстовый протокол на базе стека TCP/IP. Обмен по HTTP проходит в две фазы: запрос и ответ. Оба состоят из первой строки, заголовков и тела, которого может не быть.

Для запроса важны его метод и путь, а для ответа — статус. С развитием веба появились соглашения о том, как строить HTTP

[13] github.com/metosin/compojure-api
[14] luminusweb.com
[15] github.com/pedestal/pedestal
[16] github.com/cognitect-labs/vase
[17] github.com/duct-framework/duct
[18] clojure-liberator.github.io/liberator

API. Самое популярное называется REST. Согласно ему, путь определяет сущность, а метод — действие над ней. Данные передают в формате JSON.

Чтобы писать веб на Clojure, установите Ring. Это набор библиотек, в которых самое нужное: базовые абстракции, middleware и веб-сервер. Обработчик запроса — функция, которая принимает запрос и возвращает ответ. Обе сущности — словари.

В поставке Ring нет маршрутов, для них нужны сторонние библиотеки. Compojure предлагает макросы, чтобы задать маршруты правилами. Bidi строит дерево тегов, которое работает в паре с мультиметодом.

Middleware — это функция, которая оборачивает другую функцию. Их цепочка называется стеком. Для удобства стек описывают стрелочным оператором: это экономит скобки и делает запись наглядней. Middleware нужны для предварительной обработки запроса: прочитать JSON из тела или проверить права доступа. Отдельные middleware прерывают стек, если возникло исключение или запрос нельзя обработать.

Кроме Ring, для Clojure написаны другие фреймворки. Они задают структуру проекта, вводят правила и соглашения. Некоторые из них повторяют аналоги в других языках. Каждый фреймворк в чём-то лучше других, поэтому выбирайте, отталкиваясь от задачи.