Action Controller это C в аббревиатуре MVC. После того, как роутинг определит, какой контроллер использовать для обработки запроса, ваш контроллер ответственен за осмысление запроса и генерацию подходящего ответа. К счастью, Action Controller делает за вас большую часть грязной работы и использует элегантные соглашения, чтобы сделать это по возможности максимально просто.

Для большинства приложений, основанных на RESTful, контроллер получает запрос (это невидимо для вас, как для разработчика), извлекает или сохраняет данные в модели и использует вьюху для создания результирующего HTML. Если контроллеру необходимо работать немного по другому, не проблема, это всего лишь обычный способ работы контроллера.

Таким образом, можно представить себе контроллер как посредника между моделями и вьюхами. Он делает данные модели доступными вьюхе, так что она может отображать эти данные пользователю, и он сохраняет или обновляет данные от пользователя в модель.

Более детально о процессе маршрутизации смотрите Роутинг в Rails.

**Соглашение по именованию контроллеров**

Соглашение по именованию контроллеров в Rails устанавливает предпочтение множественного числа в последнем слове имени контроллера, хотя строго это не требуется (например, ApplicationController). К примеру, ClientsController более предпочтителен, чем ClientController, SiteAdminsController более предпочтителен, чем SiteAdminController или SitesAdminsController, и так далее.

Следование этому соглашению позволяет вам использовать генераторы маршрутов по умолчанию (например, resources и т.п.) без необходимости определять каждый :path или :controller, и сохраняет последовательным использование хелперов URL и путей во всем вашем приложении. Соглашение по именованию контроллеров отличается от соглашения по именованию моделей, которое ожидается в единственном числе.

**Методы и экшны**

Контроллер - это класс Ruby, унаследованный от ApplicationController и имеющий методы, как и любой другой класс. Когда ваше приложение получает запрос, роутинг определит, какой контроллер и экшн запустить, затем Rails создаст экземпляр этого контроллера и запустит метод с именем, как у экшна.

|  |
| --- |
| class ClientsController < ApplicationController    def new    end  end |

В качестве примера, если пользователь перейдет в /clients/new в вашем приложении, чтобы добавить нового пользователя, Rails создаст экземпляр ClientsController и запустит метод new. Отметьте, что пустой метод из вышеописанного примера будет прекрасно работать, так как Rails по умолчанию отрендерит вьюху new.html.erb, если экшн не сообщит иное. Метод new может сделать доступной для вьюхи переменную экземпляра @client для создания нового Client:

|  |
| --- |
| def new    @client = Client.new  end |

ApplicationController наследуется от ActionController::Base, который определяет несколько полезных методов. Это руководство раскроет часть из них, но если вы любопытны, можете увидеть их все сами в документации по API.

Только public методы могут быть вызваны как экшны. Хорошей практикой является уменьшение области видимости методов, не предназначенных быть экшнами, таких как вспомогательные методы и фильтры.

**Параметры**

Возможно, вы хотите получить доступ к данным, посланным пользователем, или к другим параметрам в экшнах вашего контроллера. Имеются два типа параметров, возможных в веб приложениях. Первый - это параметры, посланные как часть URL, называемые параметрами строки запроса. Строка запроса всегда следует после "?" в URL. Второй тип параметров обычно упоминаются как данные POST. Эта информация обычно приходит из формы HTML, заполняемой пользователем. Они называются данными POST, так как могут быть посланы только как часть HTTP запроса POST. Rails не делает каких-либо различий между строковыми параметрами и параметрами POST, и они оба доступны в хэше params в вашем контроллере:

|  |
| --- |
| class ClientsController < ApplicationController    # Этот экшн использует параметры строки запроса, потому, что он    # запускается HTTP запросом GET, но это не делает каких-либо    # различий в способе, с помощью которого можно получить доступ к    # ним. URL для этого экшна выглядит как этот, запрашивающий список    # активированных клиентов: /clients?status=activated    def index      if params[:status] == "activated"        @clients = Client.activated      else        @clients = Client.inactivated      end    end      # Этот экшн использует параметры POST. Они, скорее всего, пришли от    # формы HTML, которую подтвердил пользователь. URL для этого    # RESTful запроса будет /clients, и данные будут посланы    # как часть тела запроса.    def create      @client = Client.new(params[:client])      if @client.save        redirect\_to @client      else        # Эта строчка переопределяет поведение рендеринга по умолчанию, который        # отрендерил бы вьюху "create".        render "new"      end    end  end |

**Параметры в хэше и в массиве**

Хэш params не ограничен одномерными ключами и значениями. Он может содержать массивы и (вложенные) хэши. Чтобы послать массив значений, добавьте пустую пару квадратных скобок "[]" к имени ключа:

|  |
| --- |
| GET /clients?ids[]=1&ids[]=2&ids[]=3 |

Фактический URL в этом примере будет перекодирован как "/clients?ids%5b%5d=1&ids%5b%5d=2&ids%5b%5b=3", так как "[" и "]" не допустимы в URL. В основном, вам не придется беспокоиться об этом, так как браузер позаботится об этом за вас, а Rails декодирует это обратно, когда получит, но если вы когда-нибудь будете отправлять эти запросы вручную, имейте это в виду.

Значение params[:ids] теперь будет ["1", "2", "3"]. Отметьте, что значения параметра всегда строчное; Rails не делает попыток угадать или предсказать тип.

Чтобы послать хэш, следует заключить имя ключа в скобки:

|  |
| --- |
| <form accept-charset="UTF-8" action="/clients" method="post">    <input type="text" name="client[name]" value="Acme" />    <input type="text" name="client[phone]" value="12345" />    <input type="text" name="client[address][postcode]" value="12345" />    <input type="text" name="client[address][city]" value="Carrot City" />  </form> |

Когда эта форма будет подтверждена, значение params[:client] будет { "name" => "Acme", "phone" => "12345", "address" => { "postcode" => "12345", "city" => "Carrot City" } }. Обратите внимание на вложенный хэш в params[:client][:address].

Отметьте, что хэш params фактически является экземпляром ActiveSupport::HashWithIndifferentAccess, который ведет себя как хэш, но позволяет взаимозаменяемо использовать символы и строки как ключи.

**Параметры JSON**

Если вы пишете приложение веб-сервиса, возможно вам более комфортно принимать параметры в формате JSON. Если заголовок "Content-Type" вашего запроса установлен в "application/json", Rails автоматически преобразует ваши параметры в хэш params, к которому можно получить доступ обычным образом.

Так, к примеру, если вы пошлете такое содержимое JSON:

|  |
| --- |
| { "company": { "name": "acme", "address": "123 Carrot Street" } } |

То получите params[:company] как { "name" => "acme", "address" => "123 Carrot Street" }.

Также, если включите config.wrap\_parameters в своем инициализаторе или вызовете wrap\_parameters в своем контроллере, можно безопасно опустить корневой элемент в параметре JSON. Параметры будут клонированы и обернуты в ключ, соответствующий по умолчанию имени вашего контроллера. Таким образом, вышеупомянутый параметр может быть записан как:

|  |
| --- |
| { "name": "acme", "address": "123 Carrot Street" } |

И предположим, что мы посылаем данные в CompaniesController, тогда он будет обернут в ключ :company следующим образом:

|  |
| --- |
| { name: "acme", address: "123 Carrot Street", company: { name: "acme", address: "123 Carrot Street" } } |

После обращения к документации API: вы сможете настроить имя ключа или определенные параметры, которые вы хотите обернуть.

Поддержка парсинга параметров XML была извлечена в гем actionpack-xml\_parser

**Параметры роутинга**

Хэш params будет всегда содержать ключи :controller и :action, но следует использовать методы controller\_name и action\_name вместо них для доступа к этим значениям. Любой другой параметр, определенный роутингом, такой как :id, также будет доступен. Как пример рассмотрим перечень клиентов, где список может быть показан либо для активных, либо для неактивных клиентов. Мы можем добавить маршрут, который перехватывает параметр :status в "красивом" URL:

|  |
| --- |
| get '/clients/:status' => 'clients#index', foo: 'bar' |

В этом случае, когда пользователь откроет URL /clients/active, params[:status] будет установлен в "active". Когда использован этот маршрут, params[:foo] также будет установлен в "bar", как будто он был передан в строке запроса. Аналогично params[:action] будет содержать "index".

**default\_url\_options**

Можно установить глобальные параметры по умолчанию для создания URL, определив в контроллере метод по имени default\_url\_options. Этот метод должен возвращать хэш с желаемыми значениями по умолчанию, ключи которого должны быть символами:

|  |
| --- |
| class ApplicationController < ActionController::Base    def default\_url\_options      { locale: I18n.locale }    end  end |

Эти опции будут использованы как начальная точка при генерации URL, поэтому, они могут быть переопределены опциями, переданными в url\_for.

Если определить default\_url\_options в ApplicationController, как это показано в примере, они будут использованы для создания всех URL. Метод также может быть определен в одном отдельном контроллере, в этом случае он только влияет на URL, создаваемые в нем.

**Strong Parameters**

С помощью strong parameters параметры Action Controller запрещены к использованию в массовом назначении Active Model, кроме случая, когда они есть в белом листе. Это означает, что вы должны сознательно выбрать, какие атрибуты допустимы для массового назначения, и, таким образом, предотвратить случайную уязвимость, когда ее не должно быть.

Кроме того, параметры могут быть помечены как обязательные, отсутствие которых приведет к 400 Bad Request с помощью предопределенных raise/rescue.

|  |
| --- |
| class PeopleController < ActionController::Base    # Это вызовет исключение ActiveModel::ForbiddenAttributes так как используется    # массовое назначение без явного шага permit.    def create      Person.create(params[:person])    end      # Это будет проходить, пока в параметрах есть ключ person, иначе будет    # вызвано исключение ActionController::ParameterMissing, которое будет    # поймано в ActionController::Base и превращено в отклик 400 Bad Request.    def update      person = current\_account.people.find(params[:id])      person.update\_attributes!(person\_params)      redirect\_to person    end      private      # Использование приватного метода для инкапсуляции разрешенных параметров -      # это всего лишь хороший паттерн, с помощью которого можно повторно      # использовать тот же самый список разрешений при создании и обновлении      # Этот метод также можно адаптировать к проверке разрешенных атрибутов для      # определенных пользователей.      def person\_params        params.require(:person).permit(:name, :age)      end  end |

**Разрешенные скалярные величины**

Для данного

|  |
| --- |
| params.permit(:id) |

ключ :id пройдет проверку белым списком, если он появится в params и будет иметь разрешенную скалярную величину. В ином случае ключ будет отфильтрован, таким образом, массивы, хэши и любые другие объекты не смогут быть переданы.

Разрешенные скалярные типы следующие String, Symbol, NilClass, Numeric, TrueClass, FalseClass, Date, Time, DateTime, StringIO, IO, ActionDispatch::Http::UploadedFile и Rack::Test::UploadedFile.

Чтобы объявить, что значение в params должно быть массивом разрешенных скалярных величин, свяжите ключ с пустым массивом:

|  |
| --- |
| params.permit(id: []) |

Чтобы добавить в белый список полный хэш параметров, может быть использован метод permit!

|  |
| --- |
| params.require(:log\_entry).permit! |

Это пометит хэш параметров :log\_entry и любые вложенные хэши как разрешенные. Следует соблюдать предельную осторожность при использовании permit!, так как он позволит массовое назначение всех текущих и будущих атрибутов модели.

**Вложенные параметры**

Также можно позволять вложенные параметры, следующим образом:

|  |
| --- |
| params.permit(:name, { emails: [] },                friends: [ :name,                           { family: [ :name ], hobbies: [] }]) |

Это объявление поместит в белый список атрибуты name, emails и friends. Ожидается, что emails будет массивом разрешенных скалярных величин, и что friends будет массивом ресурсов с определенными атрибутами: у них будет атрибут name (допустимо любое скалярное значение), атрибут hobbies как массив разрешенных скалярных величин, и атрибут family, который может иметь только name (также допустимо любое скалярное значение).

**Дополнительные примеры**

Иногда хочется использовать разрешенные атрибуты в экшне new. Это вызывает проблему, что нельзя использовать require на корневом ключе, так как обычно он не существует при вызове new:

|  |
| --- |
| # используя `fetch`, можно предоставить значение по умолчанию и использовать  # далее Strong Parameters API.  params.fetch(:blog, {}).permit(:title, :author) |

accepts\_nested\_attributes\_for позволяет обновлять и удалять связанные записи. Это основывается на параметрах id и \_destroy:

|  |
| --- |
| # permit :id and :\_destroy  params.require(:author).permit(:name, books\_attributes: [:title, :id, :\_destroy]) |

Хэши с числовыми ключами трактуются по другому, можно объявить атрибуты так, как будто они являются прямыми детьми. Такой тип параметров вы получите при использовании accepts\_nested\_attributes\_for с комбинацией со связью has\_many:

|  |
| --- |
| # Чтобы пропустить следующие данные:  # {"book" => {"title" => "Some Book",  #             "chapters\_attributes" => { "1" => {"title" => "First Chapter"},  #                                        "2" => {"title" => "Second Chapter"}}}}    params.require(:book).permit(:title, chapters\_attributes: [:title]) |

**За пределами Strong Parameters**

Strong parameter API был разработан для наиболее общих вариантов использования. Это не панацея от всех ваших проблем белого списка. Однако можно легко смешивать API с вашим собственным кодом для адаптации к вашей ситуации.

Представим сценарий, когда вы хотите поместить в белый список атрибут, содержащий хэш с любыми ключами. Используя strong parameters нельзя разрешить эш с любыми ключами, но вы можете использовать простое присвоение для выполнения задачи:

|  |
| --- |
| def product\_params    params.require(:product).permit(:name).tap do |whitelisted|      whitelisted[:data] = params[:product][:data]    end  end |

**Сессия**

У вашего приложения есть сессия для каждого пользователя в которой можно хранить небольшие порции данных, которые будут сохранены между запросами. Сессия доступна только в контроллере и во вьюхе, и может использовать один из нескольких механизмов хранения:

* ActionDispatch::Session::CookieStore - Хранит все на клиенте.
* ActionDispatch::Session::CacheStore - Хранит данные в кэше Rails.
* ActionDispatch::Session::ActiveRecordStore - Хранит данные в базе данных с использованием Active Record. (требует гем activerecord-session\_store).
* ActionDispatch::Session::MemCacheStore - Хранит данные в кластере memcached (эта реализация - наследие старых версий, вместо нее рассмотрите использование CacheStore).

Все сессии используют куки для хранения уникального ID каждой сессии (вы обязаны использовать куки, Rails не позволяет передавать ID сессии в URL, так как это не безопасно).

Для большинства способов хранения этот ID используется для поиска данных сессии на сервере, в т.ч. в таблице базы данных. Имеется одно исключение, это дефолтное и рекомендуемое хранение сессии - CookieStore - которое хранит все данные сессии в куки (ID остается доступным, если он вам нужен). Преимущества этого в легкости, отсутствии настройки для нового приложения в порядке использования сессий. Данные в куки криптографически подписаны, что делает их защищенными от взлома. И они также зашифрованы, таким образом любой получивший к ним доступ, не сможет прочитать их содержимое (Rails не примет их, если они были отредактированы).

CookieStore могут хранить около 4kB данных - намного меньше, чем остальные - но этого обычно хватает. Хранение большего количества данных в сессии не рекомендуется, вне зависимости от того, как хранятся они в приложении. Следует специально избегать хранения в сессии сложных объектов (ничего, кроме простых объектов Ruby, например экземпляры моделей), так как сервер может не собрать их между запросами, что приведет к ошибке.

Если пользовательские сессии не хранят критичные данные или нет необходимости в ее сохранности на долгий период (скажем, если вы используете ее для сообщений во flash), можете рассмотреть использование ActionDispatch::Session::CacheStore. Он сохранит сессии с использованием реализации кэша, которую вы настроили для своего приложения. Преимущество этого в том, что для хранения сессий можно использовать существующую инфраструктуру кэширования без необходимости дополнительных настроек или администрирования. Минус, разумеется, в том, что сессии будут недолговечными и время от времени исчезать.

Если вы нуждаетесь в другом механизме хранения сессий, измените его в файле config/initializers/session\_store.rb:

|  |
| --- |
| # Use the database for sessions instead of the cookie-based default,  # which shouldn't be used to store highly confidential information  # (create the session table with "rails g active\_record:session\_migration")  # YourApp::Application.config.session\_store :active\_record\_store |

Rails настраивает ключ сессии (имя куки) при подписании данных сессии. Он также может быть изменен в config/initializers/session\_store.rb:

|  |
| --- |
| # Be sure to restart your server when you modify this file.    YourApp::Application.config.session\_store :cookie\_store, key: '\_your\_app\_session' |

Можете также передать ключ :domain и определить имя домена для куки:

|  |
| --- |
| # Be sure to restart your server when you modify this file.    YourApp::Application.config.session\_store :cookie\_store, key: '\_your\_app\_session', domain: ".example.com" |

Rails настраивает (для CookieStore) секретный ключ, используемый для подписания данных сессии. Он может быть изменен в config/initializers/secret\_token.rb:

|  |
| --- |
| # Be sure to restart your server when you modify this file.    # Your secret key for verifying the integrity of signed cookies.  # If you change this key, all old signed cookies will become invalid!  # Make sure the secret is at least 30 characters and all random,  # no regular words or you'll be exposed to dictionary attacks.  YourApp::Application.config.secret\_key\_base = '49d3f3de9ed86c74b94ad6bd0...' |

Изменение секретного ключа при использовании CookieStore делает все предыдущие сессии невалидными.

**Доступ к сессии**

В контроллере можно получить доступ к сессии с помощью метода экземпляра session.

Сессии лениво загружаются. Если вы не получаете доступ к сессиям в коде экшна, они не будут загружаться. Следовательно, никогда не нужно отключать сессии, просто не обращайтесь к ним, чтобы они не работали.

Значение сессии хранится, используя пары ключ/значение, подобно хэшу:

|  |
| --- |
| class ApplicationController < ActionController::Base      private      # Находим пользователя с ID, хранящимся в сессии с ключем    # :current\_user\_id Это обычный способ обрабатывать вход пользователя    # в приложении на Rails; вход устанавливает значение сессии, а    # выход убирает его.    def current\_user      @\_current\_user ||= session[:current\_user\_id] &&        User.find\_by(id: session[:current\_user\_id])    end  end |

Чтобы что-то хранить в сессии, просто присвойте это ключу, как в хэше:

|  |
| --- |
| class LoginsController < ApplicationController    # "Создаем" логин (при входе пользователя)    def create      if user = User.authenticate(params[:username], params[:password])        # Сохраняем ID пользователя в сессии, так что он может быть использован        # в последующих запросах        session[:current\_user\_id] = user.id        redirect\_to root\_url      end    end  end |

Чтобы убрать что-то из сессии, присвойте этому ключу nil:

|  |
| --- |
| class LoginsController < ApplicationController    # "Удаляем" логин (при выходе пользователя)    def destroy      # Убираем id пользователя из сессии      @\_current\_user = session[:current\_user\_id] = nil      redirect\_to root\_url    end  end |

Для сброса существующей сессии, используйте reset\_session.

**Flash**

Flash - это специальная часть сессии, которая очищается с каждым запросом. Это означает, что значения хранятся там доступными только до следующего запроса, что полезно для хранения сообщений об ошибке и т.п.

Доступ к нему можно получить так же, как к сессии, подобно хэшу (это экземпляр FlashHash).

Давайте посмотрим действие логаута как пример. Контроллер может послать сообщение, которое будет отображено пользователю при следующем запросе:

|  |
| --- |
| class LoginsController < ApplicationController    def destroy      session[:current\_user\_id] = nil      flash[:notice] = "You have successfully logged out."      redirect\_to root\_url    end  end |

Отметьте, что также возможно назначить сообщение флэш как часть перенаправления. Можно назначить :notice, :alert или общего назначения :flash:

|  |
| --- |
| redirect\_to root\_url, notice: "You have successfully logged out."  redirect\_to root\_url, alert: "You're stuck here!"  redirect\_to root\_url, flash: { referral\_code: 1234 } |

Экшн destroy перенаправляет на root\_url приложения, где будет отображено сообщение. Отметьте, что от следующего экшна полностью зависит решение, будет ли он или не будет что-то делать с тем, что предыдущий экшн вложил во flash. Принято отображать любые сообщения об ошибке или уведомления из flash в макете приложения:

|  |
| --- |
| <html>    <!-- <head/> -->    <body>      <% flash.each do |name, msg| -%>        <%= content\_tag :div, msg, class: name %>      <% end -%>        <!-- дальнейшее содержимое -->    </body>  </html> |

В этом случае, если экшн установил сообщения уведомления или предупреждения, макет отобразит их автоматически.

Можно передать все, что только сессия может хранить; вы не ограничены уведомлениями или предупреждениями:

|  |
| --- |
| <% if flash[:just\_signed\_up] %>    <p class="welcome">Welcome to our site!</p>  <% end %> |

Если хотите, чтобы значение flash было перенесено для другого запроса, используйте метод keep:

|  |
| --- |
| class MainController < ApplicationController    # Давайте скажем этому экшну, соответствующему root\_url, что хотим    # все запросы сюда перенаправить на UsersController#index. Если    # экшн установил flash и направил сюда, значения в нормальной ситуации    # будут потеряны, когда произойдет другой редирект, но Вы можете    # использовать 'keep', чтобы сделать его доступным для другого запроса.    def index      # Сохранит все значения flash.      flash.keep        # Можете также использовать ключ для сохранения определенных значений.      # flash.keep(:notice)      redirect\_to users\_url    end  end |

**flash.now**

По умолчанию, добавление значений во flash делает их доступными для следующего запроса, но иногда хочется иметь доступ к этим значениям в том же запросе. Например, если экшн create проваливается в сохранении ресурса, и вы рендерите макет new непосредственно тут, то не собираетесь передавать результат в новый запрос, но хотите отобразить сообщение, используя flash. Чтобы это сделать, используйте flash.now так же, как использовали нормальный flash:

|  |
| --- |
| class ClientsController < ApplicationController    def create      @client = Client.new(params[:client])      if @client.save        # ...      else        flash.now[:error] = "Could not save client"        render action: "new"      end    end  end |

**Куки**

Ваше приложение может хранить небольшое количество данных у клиента - в так называемых куки - которое будет сохранено между запросами и даже сессиями. Rails обеспечивает простой доступ к куки посредством метода cookies, который - очень похоже на session - работает как хэш:

|  |
| --- |
| class CommentsController < ApplicationController    def new      # Автозаполнение имени комментатора, если оно хранится в куки.      @comment = Comment.new(author: cookies[:commenter\_name])    end      def create      @comment = Comment.new(params[:comment])      if @comment.save        flash[:notice] = "Thanks for your comment!"        if params[:remember\_name]          # Запоминаем имя комментатора.          cookies[:commenter\_name] = @comment.author        else          # Удаляем из куки имя комментатора, если оно есть.          cookies.delete(:commenter\_name)        end        redirect\_to @comment.article      else        render action: "new"      end    end  end |

Отметьте, что если для удаления сессии устанавливался ключ в nil, то для удаления значения куки следует использовать cookies.delete(:key).

**Рендеринг данных XML и JSON**

ActionController позволяет очень просто рендерить данные XML или JSON. Если создадите контроллер с помощью скаффолдинга, то он будет выглядеть следующим образом.

|  |
| --- |
| class UsersController < ApplicationController    def index      @users = User.all      respond\_to do |format|        format.html # index.html.erb        format.xml  { render xml: @users}        format.json { render json: @users}      end    end  end |

Отметьте, что в вышеописанном коде использован render xml: @users, а не render xml: @users.to\_xml. Если объект не String, то Rails автоматически вызовет to\_xml.

**Фильтры**

Фильтры - это методы, которые запускаются до, после или "вокруг" экшна контроллера.

Фильтры наследуются, поэтому, если вы установите фильтр в ApplicationController, он будет запущен в каждом контроллере вашего приложения.

Фильтры "before" могут прерывать цикл запроса. Обычный фильтр "before" это, например, тот, который требует, чтобы пользователь был авторизован для запуска экшна. Метод фильтра можно определить следующим образом:

|  |
| --- |
| class ApplicationController < ActionController::Base    before\_action :require\_login      private      def require\_login      unless logged\_in?        flash[:error] = "You must be logged in to access this section"        redirect\_to new\_login\_url # прерывает цикл запроса      end    end  end |

Метод просто записывает сообщение об ошибке во flash и перенаправляет на форму авторизации, если пользователь не авторизовался. Если фильтр "before" рендерит или перенаправляет, экшн не запустится. Если есть дополнительные фильтры в очереди, они также прекращаются.

В этом примере фильтр добавлен в ApplicationController, и поэтому все контроллеры в приложении наследуют его. Это приводит к тому, что всё в приложении требует, чтобы пользователь был авторизован, чтобы пользоваться им. По понятным причинам (пользователь не сможет зарегистрироваться в первую очередь!), не все контроллеры или экшны должны требовать его. Вы можете не допустить запуск этого фильтра перед определенными экшнами с помощью skip\_before\_action:

|  |
| --- |
| class LoginsController < ApplicationController    skip\_before\_action :require\_login, only: [:new, :create]  end |

Теперь, экшны LoginsController new и create будут работать как раньше, без требования к пользователю быть зарегистрированным. Опция :only используется для пропуска фильтра только для этих экшнов, также есть опция :except, которая работает наоборот. Эти опции могут также использоваться при добавлении фильтров, поэтому можете добавить фильтр, который запускается только для выбранных экшнов в первую очередь.

**Последующие фильтры и охватывающие фильтры**

В дополнение к фильтрам "before", можете запустить фильтры после того, как экшн был запущен, или и до, и после.

Фильтр "after" подобен "before", но, поскольку экшн уже был запущен, у него есть доступ к данным отклика, которые будут отосланы клиенту. Очевидно, фильтр "after" не сможет остановить экшн от запуска.

Фильтры "around" ответственны за запуск экшна с помощью yield, подобно тому, как работают промежуточные программы Rack.

|  |
| --- |
| class ChangesController < ApplicationController    around\_action :wrap\_in\_transaction, only: :show      private      def wrap\_in\_transaction      ActiveRecord::Base.transaction do        begin          yield        ensure          raise ActiveRecord::Rollback        end      end    end  end |

Отметьте, что фильтры "around" также оборачивают рендеринг. В частности, если в вышеуказанном примере вьюха сама начнет считывать из базы данных (например через скоуп), она это осуществит внутри транзакции, предоставив, таким образом, данные для предварительного просмотра.

Можно не вызывать yield и создать отклик самостоятельно, в этом случае экшн не будет запущен.

**Другие способы использования фильтров**

Хотя наиболее распространенный способ использование фильтров - это создание private методов и использование \*\_action для их добавления, есть два других способа делать то же самое.

Первый - это использовать блок прямо в методах \*\_action. Блок получает контроллер как аргумент, и вышеупомянутый фильтр require\_login может быть переписан с использованием блока:

|  |
| --- |
| class ApplicationController < ActionController::Base    before\_action do |controller|      redirect\_to new\_login\_url unless controller.send(:logged\_in?)    end  end |

Отметьте, что фильтр в этом случае использует метод send, так как logged\_in? является private, и фильтр не запустится в области видимости контроллера. Это не рекомендуемый способ применения такого особого фильтра, но в простых задачах он может быть полезен.

Второй способ - это использовать класс (фактически, подойдет любой объект, реагирующий правильными методами) для управления фильтрацией. Это полезно для более сложных задач, которые не могут быть осуществлены предыдущими двумя способами по причине трудности читаемости и повторного использования. Как пример, можете переписать фильтр авторизации снова, использовав класс:

|  |
| --- |
| class ApplicationController < ActionController::Base    before\_action LoginFilter  end    class LoginFilter    def self.filter(controller)      unless controller.send(:logged\_in?)        controller.flash[:error] = "You must be logged in"        controller.redirect\_to controller.new\_login\_url      end    end  end |

Опять же, это - не идеальный пример для этого фильтра, поскольку он не запускается в области видимости контроллера, а получает контроллер как аргумент. Класс фильтра имеет метод класса filter, который запускается до или после эшна, в зависимости от того, определен ли он предварительным или последующим фильтром. Классы, используемые как охватывающие фильтры, могут также использовать тот же метод filter, и они будет запущены тем же образом. Метод должен иметь yield для исполнения экшна. Альтернативно, он может иметь методы и before, и after, которые запускаются до и после экшна.

**Защита от подделки запросов**

Подделка межсайтовых запросов - это тип атаки, в которой сайт обманом заставляет пользователя сделать запрос на другой сайт, возможно добавляя, изменяя или удаляя данные на этом сайте без ведома или разрешения пользователя.

Первый шаг в избегании этого - убедиться, что все "разрушительные" экшны (создание, обновление и уничтожение) могут быть доступны только не-GET запросам. Если вы следуете соглашениям RESTful, то уже делаете это. Однако, сайт злоумышленника может также легко послать не-GET запрос на ваш сайт, поэтому и необходима защита от подделки запросов. Как следует из ее имени, она защищает от подделанных запросов.

Это можно сделать, добавив неугадываемый токен, известный только вашему серверу, в каждый запрос. При этом способе, если запрос приходит без подходящего токена, ему будет отказано в доступе.

Если вы генерируете подобную форму:

|  |
| --- |
| <%= form\_for @user do |f| %>    <%= f.text\_field :username %>    <%= f.text\_field :password %>  <% end %> |

то увидите, как токен будет добавлен в скрытое поле:

|  |
| --- |
| <form accept-charset="UTF-8" action="/users/1" method="post">  <input type="hidden"         value="67250ab105eb5ad10851c00a5621854a23af5489"         name="authenticity\_token"/>  <!-- fields -->  </form> |

Rails добавит этот токен в каждую форму, генерируемую с помощью хелперов форм, таким образом, большую часть времени можете об этом не беспокоиться. Если вы пишете формы вручную или хотите добавить токен по другой причине, это можно сделать, используя метод form\_authenticity\_token:

form\_authenticity\_token создает валидный аутентификационный токен. Его полезно помещать в места, куда Rails не добавляет его автоматически, например в произвольные вызовы Ajax.

**Объекты Request и Response**

В каждом контроллере есть два accessor-метода, указывающих на объекты запроса и отклика, связанные с циклом запроса, находящегося в текущее время на исполнении. Метод request содержит экземпляр AbstractRequest, а метод response возвращает объект отклика, представляющий то, что собирается быть отправлено обратно на клиента.

**Объект request**

Объект request содержит множество полезной информации о запросе, полученном с клиента. Чтобы получить полный перечень доступных методов, обратитесь к документации по API. В числе свойств, доступных для этого объекта, следующие:

|  |  |
| --- | --- |
| **Свойство request** | **Назначение** |
| host | Имя хоста, используемого для этого запроса. |
| domain(n=2) | Первые n сегментов имени хоста, начиная справа (домен верхнего уровня). |
| format | Тип содержимого, запрошенного с клиента. |
| method | Метод HTTP, использованного для запроса. |
| get?, post?, patch?, put?, delete?, head? | Возвращает true, если метод HTTP - это GET/POST/PATCH/PUT/DELETE/HEAD. |
| headers | Возвращает хэш, содержащий заголовки, связанные с запросом. |
| port | Номер порта (число), использованного для запроса. |
| protocol | Возвращает строку, содержащую использованный протокол плюс "://", например "http://". |
| query\_string | Часть URL со строкой запроса, т.е. все после "?". |
| remote\_ip | Адрес IP клиента. |
| url | Полный URL, использованный для запроса. |

**path\_parameters, query\_parameters и request\_parameters**

Rails собирает все параметры, посланные вместе с запросом, в хэше params, были ли они посланы как часть строки запроса, либо в теле запроса post. У объекта request имеется три метода доступа, которые дают доступ к этим параметрам в зависимости от того, откуда они пришли. Хэш query\_parameters содержит параметры, посланные как часть строки запроса, а хэш request\_parameters содержит параметры, посланные как часть тела post. Хэш path\_parameters содержит параметры, распознанные роутингом как часть пути, ведущего к определенному контроллеру и экшну.

**Объект response**

Объект response обычно не используется напрямую, но он создается в течение исполнения экшна и рендеринга данных, которые посылаются обратно пользователю, но иногда - например, в последующем фильтре - бывает полезно иметь доступ к отклику напрямую. Некоторые из этих методов доступа имеют настройки, позволяющие изменять их значения.

|  |  |
| --- | --- |
| **Свойство response** | **Назначение** |
| body | Это строка данных, которая будет возвращена клиенту. Часто это HTML. |
| status | Код статуса HTTP для отклика, например 200 для успешного отклика или 404 для ненайденного файла. |
| location | URL, по которому клиент будет перенаправлен, если указан. |
| content\_type | Тип содержимого отклика. |
| charset | Кодировка, используемая для отклика. По умолчанию это "utf-8". |
| headers | Заголовки, используемые для отклика. |

**Установка пользовательских заголовков**

Если хотите установить произвольные заголовки для отклика, то response.headers - как раз то место, что нужно. Атрибут заголовков - это хэш, который связывает имена заголовков с их значениями, и Rails устанавливает некоторые из них автоматически. Если хотите добавить или изменить заголовок, назначьте его response.headers следующим образом:

|  |
| --- |
| response.headers["Content-Type"] = "application/pdf" |

Note: в вышеприведенном случае более очевидным было бы использование сеттера content\_type.

**Аутентификации HTTP**

Rails поставляется с двумя встроенными механизмами аутентификации HTTP:

* Простая аутентификация
* Digest аутентификация

**Простая аутентификация HTTP**

Простая аутентификация HTTP - это аутентификационная схема, поддерживаемая большинством браузеров и других клиентов HTTP. Как пример, рассмотрим административный раздел, который доступен только при вводе имени пользователя и пароля в основном диалоговом окне браузера. Использование встроенной аутентификации достаточно простое, и требует использования одного метода http\_basic\_authenticate\_with.

|  |
| --- |
| class AdminsController < ApplicationController    http\_basic\_authenticate\_with name: "humbaba", password: "5baa61e4"  end |

Разместив это, можете создавать именнованные контроллеры, наследуемые от AdminController. Таким образом, предварительный фильтр будет запущен для всех экшнов в этих контроллерах, защищая их с помощью основной аутентификации HTTP.

**Digest аутентификация HTTP**

Digest аутентификация HTTP превосходит простую аутентификацию, так как она не требует от клиента посылать незашифрованный пароль по сети (хотя простая аутентификация HTTP безопасна через HTTPS). Использовать digest аутентификацию с Rails просто, и это потребует только один метод authenticate\_or\_request\_with\_http\_digest.

|  |
| --- |
| class AdminsController < ApplicationController    USERS = { "lifo" => "world" }      before\_action :authenticate      private      def authenticate      authenticate\_or\_request\_with\_http\_digest do |username|        USERS[username]      end    end  end |

Как мы видим из примера, блок authenticate\_or\_request\_with\_http\_digest принимает только один аргумент - имя пользователя. И блок возвращает пароль. Возврат false или nil из authenticate\_or\_request\_with\_http\_digest вызовет провал аутентификации.

**Потоки и загрузка файлов**

Иногда хочется послать пользователю файл вместо рендеринга страницы HTML. Все контроллеры в Rails имеют методы send\_data и send\_file, которые направляют данные на клиента. send\_file - это удобный метод, который позволяет указать имя файла на диске, а он направит содержимое этого файла вам.

Чтобы направить данные на клиента, используйте send\_data:

|  |
| --- |
| require "prawn"  class ClientsController < ApplicationController    # Создает документ PDF с информацией на клиента и возвращает    # его. Пользователь получает PDF как загрузку файла.    def download\_pdf      client = Client.find(params[:id])      send\_data generate\_pdf(client),                filename: "#{client.name}.pdf",                type: "application/pdf"    end      private      def generate\_pdf(client)      Prawn::Document.new do        text client.name, align: :center        text "Address: #{client.address}"        text "Email: #{client.email}"      end.render    end  end |

Экшн download\_pdf в примере вызовет private метод, который фактически создаст документ PDF и возвратит его как строку. Эта строка будет направлена клиенту как загрузка файла, и пользователю будет предложено имя файла. Иногда при передаче файлов пользователю, вы можете не захотеть, чтобы их скачивали. Принятие изображений, к примеру, которое может быть внедрено в страницы HTML. Чтобы сказать браузеру, что файл не подразумевает быть скачанным, нужно установить опцию :disposition как "inline". Противоположное и дефолтное значение этой опции - "attachment.

**Отправка файлов**

Если хотите отправить файл, уже существующий на диске, используйте метод send\_file.

|  |
| --- |
| class ClientsController < ApplicationController    # Передача файла, который уже был создан и сохранен на диск.    def download\_pdf      client = Client.find(params[:id])      send\_file("#{Rails.root}/files/clients/#{client.id}.pdf",                filename: "#{client.name}.pdf",                type: "application/pdf")    end  end |

Это прочтет и передаст файл блоками в 4kB за раз, что избегает загрузки целого файла в память единовременно. Можете отключить потоковость с помощью опции :stream или отрегулировать размер блока с помощью опции :buffer\_size.

Если не указан :type, он будет угадан по расширению файла, указанного в :filename. Если для расширения не зарегистрирован тип содержимого, будет использован application/octet-stream.

Будьте осторожны, когда используете данные, пришедшие с клиента (params, куки и т.д.), для обнаружения файла на диске, так как есть риск безопасности в том, что кто-то может получить доступ к файлам, к которым они не должны.

Не рекомендуется передавать статичные файлы через Rails, если можно вместо этого разместить их в папке public на Вашем вебсервере. Более эффективно разрешить пользователям скачивать файлы напрямую, используя Apache или другой вебсервер, сохраняя запрос от ненужного прогона через весь стек Rails.

**Загрузка RESTful**

Хотя send\_data работает прекрасно, если вы создаете приложение на принципах RESTful, наличие отдельных экшнов для загрузок файла обычно не рекомендовано. В терминологии REST, файл PDF из примера выше можно считать еще одним представлением ресурса client. Rails предоставляет простой и наглядный способ осуществления загрузок в стиле RESTful. Вот как можно переписать пример так, что загрузка PDF является частью экшна show, без какой-либо потоковости:

|  |
| --- |
| class ClientsController < ApplicationController    # Пользователь может запросить получение этого ресурса как HTML или PDF.    def show      @client = Client.find(params[:id])        respond\_to do |format|        format.html        format.pdf { render pdf: generate\_pdf(@client) }      end    end  end |

Для того, чтобы этот пример заработал, нужно добавить PDF тип MIME в Rails. Это выполняется добавлением следующей строки в файл config/initializers/mime\_types.rb:

|  |
| --- |
| Mime::Type.register "application/pdf", :pdf |

Конфигурационные файлы не перезагружаются с каждым запросом, поэтому необходимо перезапустить сервер для того, чтобы изменения вступили в силу.

Теперь пользователь может запрашивать получение версии в PDF, просто добавив ".pdf" в URL:

|  |
| --- |
| GET /clients/1.pdf |

**Live Streaming произвольных данных**

Rails позволяет отдавать в потоке не только файлы. Фактически в объъекте отклика можно отдать все, что хотите. Модуль ActionController::Live позволяет создать постоянное соединение с браузером. Используя этот модуль, можно послать в браузер произвольные данные в определенные моменты времени.

**Подключение Live Streaming**

Включение ActionController::Live в класс вашего контроллера представит всем экшнам контроллера возможность отдавать данные в потоке. Этот модуль можно включить следующим образом:

|  |
| --- |
| class MyController < ActionController::Base    include ActionController::Live      def stream      response.headers['Content-Type'] = 'text/event-stream'      100.times {        response.stream.write "hello world\n"        sleep 1      }    ensure      response.stream.close    end  end |

Вышеприведенный код будет поддерживать постоянное соединение с браузером, и пошлет 100 сообщений "hello world\n", раз в секунду каждое.

В вышеприведенном примере нужно обратить внимание на ряд вещей. Необходимо убедиться, что потоковый отклик будет закрыт. Если забыть закрыть, поток оставит навсегда открытым сокет. Также необходимо установить тип содержимого text/event-stream до записи в поток отклика. Это так, потому что заголовки не могут быть записаны после того, как отклик был подтвержден (когда response.committed возвращает истинное значение), что случается, когда вы вызывает write или commit для потокового отклика.

**Пример использования**

Предположим, мы создаем машину караоке, и пользователь хочет получить слова для определенной песни. В каждом Song имеется определенное количество строчек, и у каждой строчки есть время num\_beats для завершения пения.

Если мы хотим возвращать слова в манере караоке (посылая строчку, только когда певец закончил предыдущую), можно использовать ActionController::Live следующим образом:

|  |
| --- |
| class LyricsController < ActionController::Base    include ActionController::Live      def show      response.headers['Content-Type'] = 'text/event-stream'      song = Song.find(params[:id])        song.each do |line|        response.stream.write line.lyrics        sleep line.num\_beats      end    ensure      response.stream.close    end  end |

Вышеприведенный код посылает следующую строчку только после того, как певец завершил предыдующую строчку.

**Обсуждение Streaming**

Streaming произвольных данных – чрезвычайно мощный инструмент. Как показано в предыдущих примерах, вы можете выбирать, когда и что посылать в потоковом отклике. Однако, также необходимо отметить следующие вещи:

* Каждый потоковый отклик создает новый тред и копирует локальные переменные из текущего треда. Наличие большого количество тредовых локальных переменных может отрицательно сказаться на производительности. Большое количество тредов также препятствует производительности.
* Незакрытие потокового отклика оставит соответствующий сокет открытым навсегда. Убедитесь, что вызываете close при использовании потокового отклика.
* Сервера WEBrick буферизируют все отклики, поэтому включение ActionController::Live не будет работать. Необходимо использовать веб-сервер, не буферизирующий отклики автоматически.

**Фильтрация лога**

Rails ведет лог-файл для каждой среды в папке log. Это чрезвычайно полезно при отладке, что мы уже фактически наблюдали в вашем приложении, но в реальной жизни вам может быть не нужно хранение каждого бита информации в лог-файле.

**Фильтрация параметров**

Можно фильтровать определенные параметры запросов в файлах лога, присоединив их к config.filter\_parameters в настройках приложения. Эти параметры будут помечены в логе как [FILTERED].

|  |
| --- |
| config.filter\_parameters << :password |

**Фильтрация редиректов**

Иногда нужно фильтровать из файлов лога некоторые деликатные ссылки, на которые редиректит ваше приложение. Это можно осуществить с использованием конфигурационной опции config.filter\_redirect:

|  |
| --- |
| config.filter\_redirect << 's3.amazonaws.com' |

Ей можно передать строку, регулярное выражение или массив из них.

|  |
| --- |
| config.filter\_redirect.concat ['s3.amazonaws.com', /private\_path/] |

Соответствующие URL будут помечены как '[FILTERED]'.

**Обработка ошибок**

Скорее всего, ваше приложение будет содержать ошибки или, другими словами, вызывать исключения, которые нужно обработать. Например, если пользователь проходит по ссылке на ресурс, который больше не существует в базе данных, Active Record вызовет исключение ActiveRecord::RecordNotFound.

Дефолтный обработчик исключений Rails отображает сообщение "500 Server Error" для всех исключений. Если запрос сделан локально, отображается прекрасная трассировка и некоторая дополнительная информация, так что вы можете выяснить, что пошло не так, и разобраться с этим. Если запрос был удаленным, Rails отобразит пользователю лишь простое сообщение "500 Server Error", или "404 Not Found", если была проблема с роутингом или запись не была найдена. Иногда вы захотите настроить, как эти ошибки будут перехвачены, и как они будут отображены пользователю. В приложении на Rails доступны несколько уровней обработки исключений:

**Дефолтные шаблоны 500 и 404**

По умолчанию приложение в среде production будет рендерить или 404, или 500 сообщение об ошибке. Эти сообщения содержатся в статичных файлах HTML в папке public, в 404.html и 500.html соответственно. Можете настроить эти файлы, добавив дополнительную информацию и разметку, но помните, что они статичны; т.е. нельзя использовать RHTML или макеты в них, только чистый HTML.

**rescue\_from**

Если хотите сделать нечто более сложное при перехвате ошибок, можете использовать rescue\_from, которая управляет исключениями определенного типа (или нескольких типов) во всем контроллере и его субклассах.

Когда происходит исключение, которое перехватывается директивой rescue\_from, сбойный объект передается в обработчик. Обработчик может быть методом или объектом Proc, переданным опции :with. Также можно использовать блок вместо объекта Proc.

Вот как можно использовать rescue\_from для перехвата всех ошибок ActiveRecord::RecordNotFound и что-то с ними делать.

|  |
| --- |
| class ApplicationController < ActionController::Base    rescue\_from ActiveRecord::RecordNotFound, with: :record\_not\_found      private      def record\_not\_found      render text: "404 Not Found", status: 404    end  end |

Конечно, этот пример далеко не доработан, и ничуть не улучшает обработку исключений по умолчанию, но раз вы уже перехватили все эти исключения, то вольны делать с ними все, что хотите. Например, можете создать свои классы исключений, которые будут вызваны, когда у пользователя нет доступа в определенные разделы вашего приложения:

|  |
| --- |
| class ApplicationController < ActionController::Base    rescue\_from User::NotAuthorized, with: :user\_not\_authorized      private      def user\_not\_authorized      flash[:error] = "You don't have access to this section."      redirect\_to :back    end  end    class ClientsController < ApplicationController    # Проверим, что пользователь имеет права доступа к клиентам.    before\_action :check\_authorization      # Отметим как экшны не беспокоятся об авторизационных делах.    def edit      @client = Client.find(params[:id])    end      private      # Если пользователь не авторизован, просто вызываем исключение.    def check\_authorization      raise User::NotAuthorized unless current\_user.admin?    end  end |

Некоторые исключения перехватываемы только из класса ApplicationController, так как они вызываются до того, как контроллер будет инициализирован, и экшны будут выполнены. Смотрите статью Pratik Naik по этой теме.

**Навязывание протокола HTTPS**

Иногда хочется навязать определенному контроллеру быть доступным только через протокол HTTPS по причинам безопасности. Можно использовать в контроллере метод force\_ssl, для принуждения к этому:

|  |
| --- |
| class DinnerController    force\_ssl  end |

подобно фильтру, можно также передать :only и :except для принуждения безопасного соединения только определенным экшнам.

|  |
| --- |
| class DinnerController    force\_ssl only: :cheeseburger    # или    force\_ssl except: :cheeseburger  end |

Пожалуйста, отметьте, что если вы добавили force\_ssl во многие контроллеры, то, возможно, вместо этого хотите навязать всему приложению использование HTTPS. В этом случае можно установить config.force\_ssl в файле окружения.