Ruby on Rails

Урок 8

Деплой

Git. Heroku. Ssh. Capistrano. Заключение.

Система контроля версий Git

С начала курса мы пользовались Git, чтобы сохранять этапы разработки. Это не единственное, для чего используются системы контроля версий. Основная цель их введения − организация совместной разработки приложения несколькими программистами.

Git − распределенная система контроля версий. Это значит, что центральный сервер в ней возможен, но не обязателен. Копия истории всех изменений хранится у каждого разработчика локально. Тем не менее для разработки проекта с использованием распределенной системы контроля версий типично поддержание одного «общего» репозитория. Он может быть на сервере в локальной корпоративной сети или в системе в интернете, предоставляющей такую услугу. Такими системами являются GitHub или BitBucket.

Ключевой особенностью Git является система ветвления. Ветка − это набор коммитов. Допустим, один из программистов хочет разрабатывать backend-часть, а другой предпочитает сосредоточиться на клиентской части приложения. Тогда один из них может создать ветку с названием server, а другой − client, в которых они могут работать независимо. В тот момент, когда одна из задач выполнена, ее ветка сливается в общую ветку проекта. Обычно она называется master. После этого изменения становятся доступны всем остальным участникам проекта.

На официальном сайте Git есть отличная книга GitPro.

GitHub

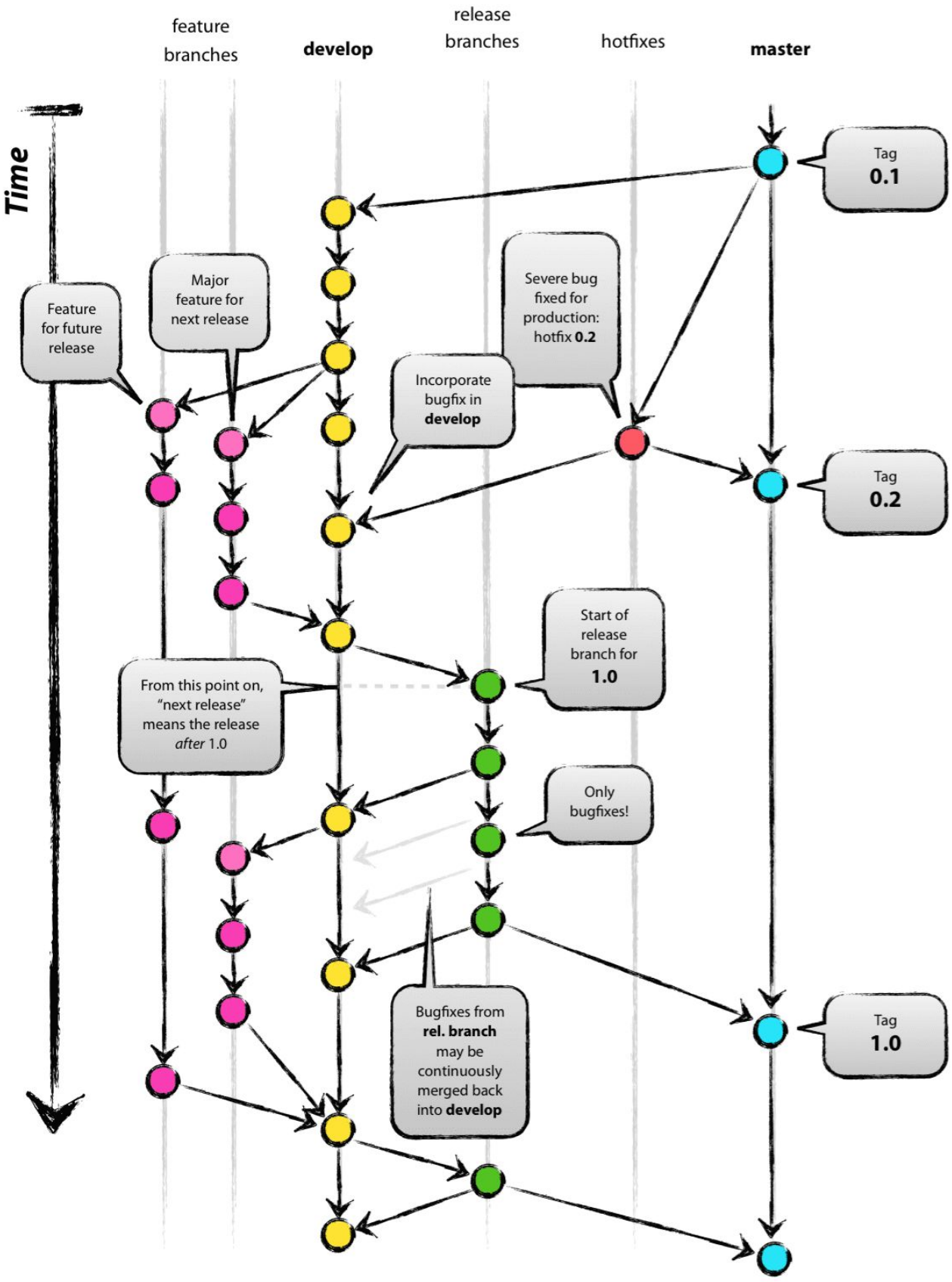
GitHub − это веб-хостинг для проектов, использующих в качестве системы контроля версий Git. Он выполняет роль центрального репозитория. Кроме того, он дает возможность добавить к каждому репозиторию wiki-описание, встроенный трекер для приема замечаний от пользователей и статичный веб-сайт, а также предоставляет удобные инструменты для проведения Code Review. На данный момент Github − это крупнейший сервис из оказывающих подобные услуги.

Для opensource-проектов Github предоставляет услуги бесплатно. Если нужны приватные репозитории, необходимо приобрести соответствующий пакет. Но есть альтернативные бесплатные git-хостинги, допускающие создание закрытых репозиториев, например bitbucket.org.

Часто готовые решения проблем можно найти в уже существующих проектах. Для этого GitHub предоставляет гибкий поиск по проектам.

На GitHub любой человек может внести вклад в развитие open source проекта. Это возможно благодаря простому и удобному созданию форков (копий) проекта и системе пулл реквестов (запросов на слияние). Если вы заметили ошибку в приложении, вы можете создать issue на его github-странице либо решить эту ошибку и создать пулл реквест на принятие вашего исправления.

GitFlow


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Система контроля версий Git часто используется для организации непрерывной доставки изменений на серверы. Такие процессы могут быть организованы по-разному. На схеме представлен один из возможных gitflow-процессов.

Разработка проекта ведется в develop-ветке (желтая), от которой разработчики под каждую из задач отводят отдельную ветку (красная). По мере готовности ветки с задачами вливаются в develop-ветку.

Если необходимо выкатить релиз, создается отдельная ветка (зеленая) для его тестирования и редактирования. После того, как релиз стабилизирован, ветка сливается в develop- и master-ветки.

Master-ветка (синяя) предназначена для выкатки в продакшн. Если нужно срочное исправление, от master-ветки отводится ветка hotfix, результаты работы в которой сливаются и в master-, и в develop-ветки.

Часто под каждую из веток в Ruby on Rails-приложении создаются отдельные площадки и окружения. Например, для develop-ветки может использоваться staging-окружение и площадка. Под отдельные задачи – дополнительные стейджинги. Под релиз и хотфиксы – preprod-окружение (площадка). Master-ветка выкатывается в продакшн на «рабочие» серверы.

Деплой на Heroku

Процедура доставки и развертывания приложения на серверы называется *деплоем.* Популярным решением для деплоя готовых Ruby on Rails-приложений является сервис heroku.com, который предоставляет минимальный контейнер (dino) бесплатно. Чтобы им воспользоваться, необходимо зарегистрироваться в сервисе и установить утилиту heroku для удаленного управления аккаунтом из командной строки.

Установка

Установим инструменты для Heroku. Для этого выполним следующие команды:

```
add-apt-repository "deb https://cli-assets.heroku.com/branches/stable/apt ./"
curl -L https://cli-assets.heroku.com/apt/release.key | sudo apt-key add -
sudo apt-get update
sudo apt-get install heroku
```

Инструкции по установке для других платформ можно найти на официальном сайте.

Регистрация приложения

Чтобы воспользоваться возможностями heroku, следует выполнить команду **heroku login**. Она запросит логин и пароль пользователя, которые были указаны при регистрации в сервисе.

```
heroku login
Enter your Heroku credentials:
Email: igorsimdyanov@gmail.com
Password: ********
Logged in as igorsimdyanov@gmail.com
```

Далее, находясь в папке проекта, необходимо создать приложение на стороне heroku при помощи команды **heroku create**:

```
heroku create
Creating app... done, ⬢ polar-spire-33894
https://polar-spire-33894.herokuapp.com/ |
https://git.heroku.com/polar-spire-33894.git
```

Команда создает на стороне heroku проект со случайным именем (polar-spire-33894), а также добавляет в git-репозиторий удаленный сервер heroku. Убедиться в этом можно, выполнив команду **git remote -v**:

```
git remote -v
heroku    https://git.heroku.com/polar-spire-33894.git (fetch)
heroku    https://git.heroku.com/polar-spire-33894.git (push)
origin    git@github.com:igorsimdyanov/blog.git (fetch)
origin    git@github.com:igorsimdyanov/blog.git (push)
```

В примере выше видно, что текущий репозиторий поддерживает два удаленных сервера:

- origin – для публикации и загрузки изменений с сервером githum.com;
- heroku – для обмена изменениями с хостингом Heroku.

Каждая команда git push в удаленный репозиторий heroku вызывает процесс развертывания приложения.

```
git push heroku master
Counting objects: 144, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (126/126), done.
Writing objects: 100% (144/144), 84.05 KiB | 4.42 MiB/s, done.
Total 144 (delta 22), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
...
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/polar-spire-33894.git
 * [new branch]    master -> master
```

После развертывания приложение доступно на домене третьего уровня https://polar-spire-33894.herokuapp.com/, где polar-spire-33894 – имя, назначенное приложению при выполнении команды heroku create.

Открыть развернутое приложение можно прямо из консоли. Для этого используется команда **heroku open**, которая открывает указанный выше адрес в странице браузера:

```
heroku open
```

Доступ к логам

Так как Web-сервер запущен и выполняется на удаленном хосте, сложно понять, что происходит в случае возникновения ошибок. Поможет команда heroku logs --tail, которая выводит сообщения об ошибках в режиме реального времени.

```
heroku logs --tail
```

Чтобы прекратить вывод логов в консоль, достаточно нажать комбинацию клавиш Ctrl+C.

Запуск консоли на удаленном сервере

Для запуска консоли на удаленном сервере предназначена команда **heroku run rails c**.

```
heroku run rails c
Running rails c on ⬢ polar-spire-33894... up, run.5900 (Free)
Loading production environment (Rails 5.1.4)
>>
>> 2 + 2
=> 4
```

Причем выполнить можно любую команду – не обязательно rails c. Например, можно запустить командную оболочку bash, в которой выполняется команда ls -la. Эта команда позволяет просмотреть содержимое текущей директории удаленного сервера.

```
heroku run bash
Running bash on ⬢ polar-spire-33894... up, run.2470 (Free)
~ $ ls -la
total 332
drwx------  14 u52593 dyno   4096 Jan 20 14:20 .
drwxr-xr-x  15 root   root   4096 Jan 16 10:45 ..
drwx------  11 u52593 dyno   4096 Jan 20 14:00 app
-rw-------   1 u52593 dyno    312 Jan 20 14:00 .babelrc
drwx------   2 u52593 dyno   4096 Jan 20 14:00 bin
drwx------   2 u52593 dyno   4096 Jan 20 14:00 .bundle
drwx------   6 u52593 dyno   4096 Jan 20 14:00 config
-rw-------   1 u52593 dyno    130 Jan 20 14:00 config.ru
drwx------   3 u52593 dyno   4096 Jan 20 14:00 db
-rw-------   1 u52593 dyno   1667 Jan 20 14:00 Gemfile
-rw-------   1 u52593 dyno   4371 Jan 20 14:00 Gemfile.lock
-rw-------   1 u52593 dyno    524 Jan 20 14:00 .gitignore
drwx------   4 u52593 dyno   4096 Jan 20 14:00 lib
drwx------   2 u52593 dyno   4096 Jan 20 14:00 log
drwx------ 787 u52593 dyno  32768 Jan 20 14:01 node_modules
-rw-------   1 u52593 dyno    192 Jan 20 14:00 package.json
-rw-------   1 u52593 dyno     52 Jan 20 14:00 .postcssrc.yml
drwx------   2 u52593 dyno   4096 Jan 20 14:00 .profile.d
drwx------   4 u52593 dyno   4096 Jan 20 14:01 public
-rw-------   1 u52593 dyno    227 Jan 20 14:00 Rakefile
-rw-------   1 u52593 dyno    374 Jan 20 14:00 README.md
drwx------   3 u52593 dyno   4096 Jan 20 14:01 tmp
drwx------   6 u52593 dyno   4096 Jan 20 14:00 vendor
-rw-------   1 u52593 dyno 209693 Jan 20 14:00 yarn.lock
```

Установка переменных окружения

Для установки переменных окружения на удаленном сервере предназначена команда **heroku config:set**. Например, при помощи нее можно установить переменную окружения SECRET_KEY_BASE, которая используется в файле config/secrets.yml.

```
$ heroku config:set SECRET_KEY_BASE=secret_value
Setting SECRET_KEY_BASE and restarting ⬢ polar-spire-33894... done, v6
SECRET_KEY_BASE: secret_value
```

Получить список всех установленных значений можно при помощи команды **heroku config**.

```
$ heroku config
=== polar-spire-33894 Config Vars
DATABASE_URL:
postgres://ujxjbzhleopgog:cfba7d4ba7f5dfde9c998d1d8f39d8add40ba7503ab6bbf6ef57e1
bb300757c8@ec2-54-243-193-227.compute-1.amazonaws.com:5432/dvg2akejptrcg
LANG:                    en_US.UTF-8
RACK_ENV:                production
RAILS_ENV:               production
RAILS_LOG_TO_STDOUT:     enabled
RAILS_SERVE_STATIC_FILES: enabled
SECRET_KEY_BASE:         secret_value
```

База данных

Heroku поддерживает базы данных Redis, MongoDB, MySQL и PostgreSQL. При выполнении команды heroku config мы видим параметр DATABASE_URL. Он задает логин, пароль, адрес сервера и базу данных, которые были автоматически сформированы heroku для нашего приложения. Дополнительную информацию о базе данных можно получить из отчета команды **heroku pg**.

```
heroku pg
=== DATABASE_URL
Plan:                    Hobby-dev
Status:                  Available
Connections:             0/20
PG Version:              10.1
Created:                 2018-01-20 14:01 UTC
Data Size:               7.6 MB
Tables:                  0
Rows:                    0/10000 (In compliance)
Fork/Follow:             Unsupported
Rollback:                Unsupported
Continuous Protection: Off
Add-on:                  postgresql-shallow-78670
```

Для обслуживания базы данных, например для выполнения миграций и заполнения данными, можно воспользоваться командой **heroku run**.

```
heroku run rails db:migrate db:seed
```

Протокол SSH

Для установки на удаленном сервере программного обеспечения и работы на нем, настройки конфигурационных файлов и мониторинга запущенных процессов, нужно выполнение команд на сервере. Для связи с сервером, выполнения команд и обмена файлами администраторы и Web-разработчики используют протокол SSH (Secure Shell), обеспечивающий полное шифрование трафика.

OpenSSH-сервер

Как правило, сервер OpenSSH устанавливается одновременно с дистрибутивом. Если этого не произошло, установить его можно при помощи менеджера пакетов apt-get. Командой sudo менеджер запускается с привилегиями суперпользователя root. В качестве устанавливаемого пакета можно указать openssh-server:

```
sudo apt-get install openssh-server
```

Либо можно указать общий пакет ssh, который установит openssh-server как зависимый пакет:

```
sudo apt-get install ssh
```

После ввода пароля и подтверждения происходит загрузка и установка пакета.

Как правило, на коммерческих хостингах или виртуальных машинах под управлением Vagrant ssh-сервер уже запущен и настроен.

Клиент для доступа по SSH

Чтобы обратиться к удаленному серверу, необходимо выполнить команду ssh, передав ей адрес или домен удаленного хоста:

```
ssh remote.server.com
```

При этом утилита ssh использует в качестве логина имя текущего пользователя, которое редко совпадает с именем учетной записи. Поэтому имя учетной записи часто указывают явно, отделяя его от адреса символом @:

```
ssh igor@remote.server.com
```

При первом соединении с удаленным сервером он может попросить подтверждение от пользователя. После его получения он сохранит адрес удаленного хоста в локальном файле ~/.ssh/known_hosts.

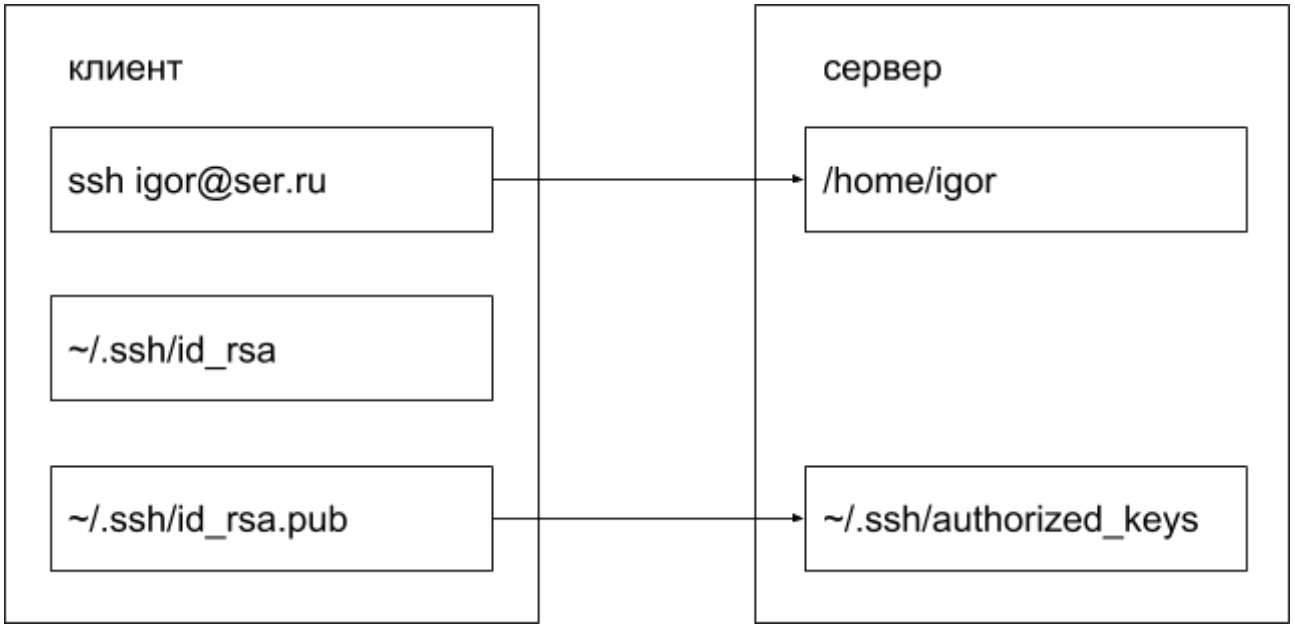
```
ssh igor@remote.server.com
```

```
The authenticity of host '192.168.0.1 (192.168.0.1)' can't be established.
RSA key fingerprint is 32:f5:a3:e6:50:14:86:00:04:d4:1d:3e:a8:3e:af:4c.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.1' (RSA) to the list of known hosts.
```

В следующий раз, обнаружив IP-адрес в этом файле, ssh-клиент установит соединение с сервером без дополнительных вопросов.

Доступ по SSH-ключу

Чтобы не вводить каждый раз пароль, для организации доступа на сервер используют ключи. Клиент заводит пару файлов-ключей: открытый и закрытый. Закрытый ключ помещается в домашнем каталоге локального компьютера ~/.ssh/id_rsa. Открытый (public) ключ id_rsa.pub размещается на сервере в конфигурационном файле ~/.ssh/authorized_keys в домашнем каталоге того пользователя, под учетной записью которого производится вход на сервер.



Для генерации ключей служит команда ssh-keygen, которую следует выполнить на клиентской машине:

```
ssh-keygen -t rsa
```

Во время выполнения команда задаст несколько вопросов. В частности, она попросит указать путь, куда будут сохранены ключи. По умолчанию это папка .ssh домашнего каталога пользователя. Далее будет предложено ввести пароль для закрытого ключа. Лучше его указать. Тогда при похищении ключом не смогут воспользоваться без пароля.

В результате выполнения команда создаст в домашнем каталоге скрытый подкаталог .ssh, в котором разместит закрытый id_rsa и открытый id_rsa.pub ключи. Закрытый ключ никогда не должен попадать в чужие руки или передаваться через незащищенные сетевые каналы. В идеале он вообще не

должен покидать компьютер, на котором был создан. Открытый ключ может свободно распространяться. Его можно регистрировать на всех хостах, к которым нужно получить доступ, включая площадки хост-провайдеров и удаленные репозитории, вроде GitHub.

Если у вас уже есть открытый и закрытый ключи, следует самостоятельно создать каталог .ssh, например при помощи команды **mkdir**. Нужно скопировать в него ключи и установить закрытому ключу UNIX-доступ только для владельца файла. Для последней операции можно воспользоваться утилитой chmod.

```
mkdir .ssh
cp /path/to/keys/id_rsa ~/.ssh/id_rsa
cp /path/to/keys/id_rsa.pub ~/.ssh/id_rsa.pub
chmod 0600 ~/.ssh/id_rsa
```

После того, как ключи созданы, открытый ключ id_rsa.pub переправляется на сервер и дописывается в конец файла authorized_keys:

```
cat id_rsa.pub >> ~/.ssh/authorized_keys
```

RSA не единственно возможный формат. Параметр -t в команде ssh-keygen позволяет задать несколько типов шифрования: rsa, dsa или edcsa. Но последний не поддерживается старыми версиями SSH-клиентов.

После регистрации открытого ключа в файле authorized_keys вход по протоколу SSH будет проходить без запроса пароля.

Копирование файлов по SSH-протоколу

По SSH-протоколу можно как загружать файлы на сервер, так и скачивать их с сервера. Для этой процедуры предназначена утилита scp, которая повторяет синтаксис традиционной файловой утилиты копирования cp:

```
scp /path/to/source path/to/destination
```

Первый параметр задает источник копирования, второй − пункт назначения. Каждый из параметров задается как путь к удаленному хосту, который строится по следующим правилам: имя_пользователя@хост:путь_к_файлу. В примере ниже локальный файл id_rsa.pub загружается на сервер по адресу /home/igor/.ssh/id_rsa.pub

```
scp id_rsa.pub igor@remote.server.com:/home/user/.ssh/id_rsa.pub
```

Если папка /home/igor − это домашним каталог пользователя igor на хосте remote.server.com, вместо абсолютного пути можно указать путь относительно домашнего каталога:

```
scp id_rsa.pub igor@remote.server.com:.ssh/id_rsa.pub
```

Подставив сетевой путь в качестве первого параметра-источника, можно скачать файл с удаленной машины на локальную:

```
scp igor@remote.server.com:.ssh/config config
```

Кроме того, утилита scp допускает передачу файла от одного удаленного хоста другому без загрузки копии на хост управления. One и two – псевдонимы удаленных серверов, заданных в конфигурационном файле config:

```
scp one:.ssh/config two:.ssh/config
```

Для передачи папки вместе со всеми вложенными подпапками можно воспользоваться ключом -r (рекурсивное копирование):

```
scp -r one:.ssh/dir two:.ssh/dirs
```

Однако следует помнить об особенности команды scp – она не копирует скрытые UNIX-файлы, т. е. файлы, которые начинаются с точки.

Гем Capistrano

Capistrano – это библиотека, написанная на Ruby. Она предоставляет основу для автоматизации задач, которые относятся к деплою приложений на удаленный сервер: получение кода из git-репозитория, выполнение компиляции ассетов, миграций, рестарта сервера и т.д.

В облачном хостинге Heroku деплой автоматизирован. Capistrano позволяет деплоить на неподготовленные контейнеры и серверы. Например, на серверы Amazon, где тоже можно бесплатно получить минимальный контейнер сроком на год.

Capflow

Capistrano – это универсальный инструмент, с помощью которого можно деплоить любые приложения, а не только Ruby on Rails. Изначально процесс деплоя описывается четырьмя стадиями, каждая из которых генерирует два события: начало процесса и завершение.

```
deploy:starting    - start a deployment, make sure everything is ready
deploy:started     - started hook (for custom tasks)
deploy:updating    - update server(s) with a new release
deploy:updated     - updated hook
deploy:publishing  - publish the new release
deploy:published   - published hook
deploy:finishing   - finish the deployment, clean up everything
deploy:finished    - finished hook
```

К каждой из этих стадий привязывается rake-задача. Так образуется последовательность задач, которые необходимо выполнить, чтобы выкатить приложение. Например, в случае RoR-приложения последовательность задач может выглядеть следующим образом:

```
deploy
  deploy:starting
    [before]
      deploy:ensure_stage
      deploy:set_shared_assets
    deploy:check
  deploy:started
  deploy:updating
    git:create_release
    deploy:symlink:shared
  deploy:updated
    [before]
      deploy:bundle
    [after]
      deploy:migrate
      deploy:compile_assets
      deploy:normalize_assets
  deploy:publishing
    deploy:symlink:release
  deploy:published
  deploy:finishing
    deploy:cleanup
  deploy:finished
    deploy:log_revision
```

В этот процесс можно добавлять собственные стадии, оформляя их в виде rake-задач. Более того, существующие в экосистеме Ruby гемы самостоятельно встраиваются в процесс деплоя.

Подключение к проекту

Для подключения гема Capistrano к проекту следует подключить гем capistrano-rails в группе development. В других окружениях гем использоваться не будет. Также следует подключить capistrano-bundler и capistrano-rvm. Они позволяют автоматически устанавливать гем bundler на сервере и поддерживать менеджер версий rvm. В случае применения менеджера версий rbenv следует воспользоваться capistrano-rbenv.

Gemfile

```
...
group :development do
  gem 'capistrano-bundler', require: false
   gem 'capistrano-rails', require: false # gem 'capistrano-rbenv', require:
'false'
  gem 'capistrano-rvm', require: false
end
...
```

После внесения изменений в файл Gemfile, следует выполнить команду bundle install. Гем capistrano предоставляет генератор, запустить который можно при помощи команды cap install.

```
cap install
mkdir -p config/deploy
create config/deploy.rb
create config/deploy/staging.rb
```

```
create config/deploy/production.rb
mkdir -p lib/capistrano/tasks
create Capfile
Capified
```

Команда создает файл config/deploy.rb, который определяет конфигурационные параметры процедуры деплоя. Для каждого из окружений, для которых предусмотрены удаленные площадки, создаются файлы в каталоге config/deploy – staging.rb и production.rb. Они задают индивидуальные настройки для текущего окружения. Помимо этого создается директория lib/capistrano/tasks, в которой можно размещать rake-задачи, предназначенные для встраивания в capflow-процесс. Также создается файл Capfile, который служит точкой входа для вызовов команд capistrano.

Деплой

Чтобы задеплоить проект, следует воспользоваться командой **cap deploy.** По умолчанию она выполняет деплой production-окружения. Уточнить, какое из окружений подвергается деплою, можно при помощи команды **cap staging deploy**.

```
cap staging deploy
```

Роли и хосты

Деплой проекта делают, как правило, сразу на несколько хостов. Допускается деплой на выбранный хост или хосты, которые задаются переменной окружения HOSTS.

```
HOSTS=p1.remote.server.com,p2.remote.server.com cap staging deploy
```

Группам серверов может быть назначена роль. App или front – это серверы front-части, на которые поступают запросы от пользователей. Backend – серверы, несущие ответственность за функционирование системы администрирования. Queue – серверы, на которых выполняются фоновые задачи и т.д. Задать роль позволяет переменная окружения ROLES. Это дает возможность задеплоить проект только на серверы, принадлежащие данной роли.

```
ROLES=front cap production deploy
```

Настройка деплоя

Чтобы команда cap deploy корректно заработала, необходимо настроить проект. Для этого в файле Capfile следует раскомментировать следующие require-конструкции:

Capfile

```
require "capistrano/rvm"
# require "capistrano/rbenv"
# require "capistrano/chruby"
require "capistrano/bundler"
require "capistrano/rails/assets"
```

```
require "capistrano/rails/migrations"
# require "capistrano/passenger"
```

Содержимое конфигурационного файла config/deploy.rb следует поправить так:

config/deploy.rb

```
lock "~> 3.10.0"

set :application, "my_bolojek"
set :repo_url, "git@github.com:igorsimdyanov/my_blojek.git"

server 'proftime.edu.ru', roles: %w(app db)
set :ssh_options, user: 'igor'

set(:branch, proc { `git rev-parse --abbrev-ref HEAD`.chomp }.call)
set(:branch, ENV['BRANCH']) if ENV['BRANCH']

set :deploy_to, '/var/www/my_blojek' # => production

set :bundle_jobs, 4

set :format, :pretty

set :log_level, :debug

set :pty, true

append :linked_dirs, 'log', 'tmp/pids', 'sockets', 'public/static'

set :rvm_type, :user
set :rvm_ruby_version, '2.3.1'

set :default_env, rails_env: fetch(:stage)
```

Гем dotenv

Для передачи Rails-приложению переменных окружения в различных конфигурационных файлах пригодится гем dotenv-rails. Подключить его можно в конфигурационном файле Gemfile.

Gemfile

```
...
gem 'dotenv-rails'
...
```

В корне проекта нужно завести файл .env.local, который содержит пары ключ-значение, задающие переменные окружения.

.env.local

```
RESQUE=true
SECRET_KEY_BASE=dflsdjfldsjfldsjfldsjfldsfjdsl
```

Файл следует поместить в исключения .gitignore, чтобы для каждого из хостов файл был уникальный. Так задаются переменные окружения, которые будут расположены на сервере и никогда не будут покидать его пределов.

Запуск rake-задач на удаленном сервере

Для запуска команд на сервере необходимо запускать их через bundle exec. В переменной окружения RAILS_ENV нужно указать актуальное окружение, чтобы команда имела возможность соединиться с корректным сервером базы данных.

```
RAILS_ENV=production bundle exec rails db:seed
```

Запуск консоли на удаленном на сервере

Схожим образом запускается консоль на удаленном сервере. Для ее запуска следует перейти в корень проекта и запустить команду bundle exec rails c, передав ей в качестве аргумента название окружения.

```
bundle exec rails c production
```

Запуск puma на удаленном сервере

Для запуска Web-сервера puma на удаленном хосте необходимо подготовить конфигурационный файл. Для этого в папке config следует создать файл puma_prod.rb следующего содержания:

```
require 'dotenv'
Dotenv.load('.env.local')

workers Integer(ENV.fetch("WEB_CONCURRENCY", 8))
threads_count = Integer(ENV.fetch("MAX_THREADS", 1))
threads(threads_count, threads_count)

preload_app!

rackup DefaultRackup
port 8082
environment 'production'

on_worker_boot do
  ActiveRecord::Base.establish_connection
```

```
end
```

После этого запустить сервер можно при помощи команды:

```
RAILS_ENV=production bundle exec puma --config config/puma_prod.rb
```

Web-сервер nginx

Серверы на Ruby – thin, unicorn, puma – имеют ограничения. Они связаны со скоростью обработки запросов. Программы, созданные на интерпретируемом языке, выполняются медленнее, чем промышленный Web-сервер nginx. Он спроектирован для обслуживания гигантского количества соединений. Ряд запросов вообще не требуют выполнения логики, например отдача статики (изображений и других ассетов). Те запросы, которые можно обслужить без привлечения Ruby-сервера, стараются выполнить посредством nginx. Он работает быстрее и потребляет меньше памяти.

Для установки nginx следует выполнить команду:

```
sudo apt-get install nginx
```

Управляют сервером при помощи команды service, после которой указывается одно из управляющих ключевых слов.

```
sudo service nginx start
sudo service nginx stop
sudo service nginx restart
sudo service nginx reload
```

Команда **start** запускает сервер, **stop** – останавливает, **restart** – перезапускает, а **reload** – перечитывает конфигурационные файлы.

Чтобы убедиться, что процессы nginx запущены, можно обратиться к утилите ps, показывающей список процессов, и отфильтровать результаты по ключевому слову nginx:

```
ps aux | grep nginx
root       559  0.0  0.0  86792  2412 ?  Ss     2015   0:00 nginx: master
process
www-data   570  0.0  0.1  88168  4592 ?  S 2015   9:26 nginx: worker process
www-data   571  0.0  0.1  88160  4540 ?  S 2015   9:18 nginx: worker process
www-data   572  0.0  0.1  88160  4540 ?  S 2015   9:17 nginx: worker process
www-data   573  0.0  0.1  87840  4212 ?  S 2015   9:35 nginx: worker process
```

Конфигурационные файлы nginx сосредоточены в папке /etc/nginx. Все они состоят из секций, внутри которых размещаются директивы:

```
<секция> {
  <директива> <значение>;
}
```

Обратите внимание, что директивы внутри секции завершаются обязательной точкой с запятой.

Секции задают разный уровень действия директив: весь сервер, виртуальный хост (отдельный сайт), папка, файл с определенным расширением и т. п.

Один сервер может обслуживать несколько сайтов. Клиенты отправляют HTTP-заголовок Host с доменным именем сайта, чтобы сервер мог определить, какому из сайтов адресован HTTP-запрос. Набор директив, которые обслуживают такой отдельный сайт, называется виртуальным хостом. В nginx для организации виртуальных хостов предназначена специальная секция server. Внутри нее при помощи директивы listen указывается порт, прослушивающий nginx, а при помощи директивы server_name − доменные имена, относящиеся к текущему виртуальному хосту.

Файл виртуального хоста размещается в папке /etc/nginx/sites-available/. Чтобы активировать конфигурационный файл, в папке /etc/nginx/sites-enabled/ необходимо создать символическую ссылку. Сделать это можно при помощи команды **ln с** параметром -s:

```
sudo ln -s /etc/nginx/sites-available/example.com
/etc/nginx/sites-enabled/example.com
```

Первый путь определяет файл, на который указывает ссылка, второй − местоположение ссылки.

Типичное содержимое виртуального хоста для Ruby on Rails-приложения выглядит так:

```
server {
  listen 80;
  server_name blog.domain.ru;
  server_tokens off;
  root /var/www/blog/current/public;

  access_log  /var/log/nginx/blog_access.log;
  error_log   /var/log/nginx/blog_error.log;

  location / {
    try_files $uri @blog;
  }

  location @blog {
    proxy_read_timeout 300;
    proxy_connect_timeout 300;
    proxy_redirect      off;

    proxy_set_header   X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header   X-Forwarded-Proto $scheme;
    proxy_set_header   Host                $http_host;
    proxy_set_header   X-Real-IP           $remote_addr;
    proxy_set_header   CLIENT_IP           $remote_addr;
```

```
  proxy_pass http://127.0.0.1:8082;
  }
}
```

Заключение

В этом курсе мы изучили базовые подходы к разработке приложений на Rails, концепции MVC и рендеринга html-страниц на стороне сервера.

Мы не углублялись в тему frontend намеренно. В современной разработке есть тенденция отказа от рендеринга визуального представления на стороне сервера и переноса логики контроллеров и представлений в клиентское JavaScript-приложение. Тем не менее Rails остается актуальным и мощным инструментов для реализации backend, общающегося с клиентом посредством Rest API.

У такого подхода есть свои плюсы и минусы. Классический рендеринг на стороне backend позволяет сделать клиента «легковесным». Это благоприятно сказывается на времени ожидания пользователем загрузки приложения. Для небольших и простых по своей структуре frontend'ов такой способ вполне удачен и приемлем.

Если же клиентская часть приложения имеет очень сложную логику, возможно, стоит воспользоваться иным подходом к ее разработке. В частности, можно присмотреться к таким технологиям, как webpack, react (redux, react-router и т.д.), postcss.

В рамках курса мы прошлись по основным понятиям и примерам использования ORM, ActiveRecord. Для небольших приложений функционала и скорости работы ORM может быть достаточно. Но в больших приложениях часто необходимо использовать особенности баз данных: хранимые процедуры, оконные запросы, сложные джоины, представления и т.д. Поэтому лучше пройти специальный курс, посвященный базам данных, или изучить эту тему самостоятельно.

Обратите внимание на инструменты разработки, которые помогают следить за чистотой кода. Их называют линтерами. Самыми популярными линтерами в мире Ruby являются rubocop и reek. Я рекомендую использовать оба. Советы линтеров помогают не только оставлять код читабельным для других разработчиков, но и правильно его структурировать. Тогда в будущем его проще модифицировать и расширять. Для всех современных популярных IDE или редакторов кода есть удобные плагины для работы с линтерами.

Домашнее задание

1. Разверните ваше приложение на Heroku.

Дополнительные материалы

1. Книга про GIT.
2. Документация по использованию Heroku.
3. Документация по Capistrano.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Книга про GIT.
2. Getting Started on Heroku with Ruby.
3. capistranorb.com.
4. Ben Dixon. Reliably Deploying Rails Applications.
5. Майкл Хартл. Ruby on Rails для начинающих. Изучаем разработку веб-приложений на основе Rails.
6. Ruby on Rails Guides.
7. Obie Fernandez. The Rails 5 Way.