この物語は、Railsプログラマの
平凡な開発風景を淡々と描くものです。
過度に技術的な期待はしないでください。

第0話　ドックフードを食べる


律先輩、ドックフードを食べてください


えッ!!


さあ口を開けるです!


や、やめろおおおおおお モゴっ


まあ冗談ですけど


はぁ・・・はぁ・・・


自分で作ったプログラムを自分で試すことを『ドックフードを食べる』といいます


へぇー


なので律先輩にはできあがったばかりのこのシステムを使ってSSを書いてもらいます


なるほど これを試せばいいんだな!


じゃあこれまでの開発の経緯をストーリにしてください


お、覚えてるかな・・・


しっかりしてくださいよ・・・


あ、なにか使いにくいところとか気づいたことがあったらすぐ教えてください
そのためのドックフードですから


ムギも使ってみたのか~?


カチャカチャカチャカチャカチャカチャ(ハァハァ)
カチャカチャカチャカチャカチャカチャ(ハァハァ)
カチャカチャカチャカチャカチャカチャ(ハァハァ)
カチャカチャカチャッ ターーーンッ!!!(ハァハァ)


こいつッ・・・ すでに大長編百合小説を書き始めている・・・だとッ

解説
このシステムは、まさにこの原稿を書くために作りました。
その原稿の最初の稿が「このシステムを使って原稿を書け」という内容なわけで、
まあなんかよくわからない世界に迷い込んだ感じがします。

2次元への扉を垣間見た瞬間です。

第1話　ムギちゃんの野望


ねぇねぇ、梓ちゃん


(珍しいな ムギ先輩が話しかけてくるなんて)
はい、なんですか?


このページを見てほしいの!


どれどれ・・・ 「けいおんSSまとめ」ですか・・・
SSってなんでしょう


こ、こ、こ、こ、これはっ
み、み、み、澪先輩と・・・り、り、り、律先輩が・・・
なんかえっちです!!


あら 気にってくれた?


か、からかわないでください!!


こんなのもあるのよ~?


あたしが唯先輩にメチャクチャにされてるぅぅぅぅぅぅぅぅぅ
あっ ダメっ そんなところダメっ


(ハァハァハァハァハァハァハァハァハァハァ)


(ハッ!!!!)
もう! なんなんですか ムギ先輩!


SSってステキでしょ 私も書いてみたいの


か、書けばいいじゃないですか・・・


htmlとかわからないの・・・
それでね、梓ちゃんプログラム書けるだってね?
SSを簡単に書けるプログラムを書いてほしいの


あたしはPerl,PHP,Python,Rubyなら使えます
http://twitter.com/#!/nakano_azusa/status/3241649273


いいですよ、作ります
じゃあまず報酬の話からしましょうか


(梓ちゃんってもしかして結構ガメツイのかな・・・)

解説
お金は大事です。

第2話　チーム


ではプロジェクトチームを組みましょう
あたしが開発者でムギ先輩はプロダクトオーナーです


プロダクトオーナーって何をするのかしら


プロダクトオーナーはアプリケーションの機能を決める人です
今回は顧客とユーザもムギ先輩ですね


顧客もチームの一員なのね!


あとプロジェクトマネージャを決めましょう
そうですね、あそこで暇そうにしている
律先輩!!


えっ! 私!?


律先輩にはプロジェクトマネージャとしてプロジェクト管理とちょっとした開発と主に雑用をしてもらいます


や、やだよ 管理なんて! もっと人の上に立つ素質のあるやつにやらせろよ!!
例えば軽音部の部長とか!!


あら、軽音部の部長はりっちゃんよ


しまった! ハメられた!!


とりあえずプロジェクトマネージャは律先輩で決定ですね
それではこのチームでSSエディタ(仮)をつくって行きます
よろしくおねがいします

解説
アジャイル臭がプンプンするぜーーーッ
客はいつも無理を言ってくる敵じゃない、開発のメンバーなんだ、
全員で協力して最高のプロダクトつくろうぜ!!!ってことです。

第3話　環境構築


開発はRuby on Railsでつくろうと思います
お二人はどんなコンピュータを使ってますか?


私は普通のWindowsパソコンだ!


私はmacよ


ムギ先輩はそのままでいいです
律先輩はUbuntu Linuxをインストールしてください


Windowsじゃダメなのか!?


ダメです
Windowsはプログラミングに向きません
VirtualBoxを使って仮想環境(VM)を作ってそこにインストールで大丈夫です


検索してみたらデュアルブートって方法もあるらしいぜ


デュアルブートはダメです
いろいろ事故が起きやすいですし面倒になって片方のOSを起動しなくなります
だからVMが一番です


VirtualBoxとUbuntuのインストールは簡単だな
あっさり終わったぜ!


次は必要なソフトウェアのインストールですね
openssh openssl emacs ruby-mode git gitk libxml2 zlib sqlite3
あたりを入れておけば十分でしょう
OSのパッケージシステムを使って入れましょう


私は開発できないけど入れるのかしら?


ムギ先輩もプロダクトオーナーとして実際に開発中のアプリケーションに触ってみる必要があります


ん? かんじんのrubyを入れてないぞ?


私のmacには入ってるわ


rubyはrvmを使って入れます https://rvm.beginrescueend.com/
書かれているコマンドをコピペして実行すればインストールできます
ムギ先輩も、macに最初から入っているrubyは使いません


おっ インストールできたぞ
あとは rvm install ruby-head で最新のrubyがインストールできるんだな...
できた!


では次はいろいろなサービスに登録していきましょう
まず最初に先ほどインストールしたgitを使うために
githubに登録します http://github.com


gitというのはソースコードのバージョン管理をするソフトウェアです
このプロジェクトはすべてgitでコードを管理します
githubはgitで管理したコードを共有するためのサービスです
ss_editor という名前でこのプロジェクトのリポジトリを作ってみました
http://github.com/np-complete/ss_editor


なにかキーを登録するって出てきたわ


ssh-keygen でキーを作ってください
その後 ~/.ssh/id_rsa.pub の中身をコピペするだけです
ちなみにWindowsがプログラミングに向かないのは、sshとgitがまともに使えないからです


キーの登録もできたぜ!
次は何するんだ?


次はPivotal Trackerに登録しましょう https://www.pivotaltracker.com
これはプロジェクトでつくる予定の機能を、
「ストーリー」という形で管理するためのツールです
これも ss_editor という名前でプロジェクトを登録しました
https://www.pivotaltracker.com/projects/267615


なんだか面白そうね!
タスクが可視化されてとても便利だわ


これで環境の準備は終わりました

あっ、大切なことを忘れてました!
キーボードをHappyHackingKeyboardにしましょう!
もちろん無刻印です http://amzn.to/iW8Y2m


(た、高い・・・)

解説
道具は全ての入り口です。一番使いやすい道具を揃えましょう。
それはLinuxであり、gitであり、HHKであり、2面以上のモニタです。
作業効率は楽しさを感じる重要な要素の一つです。

第4話　ストーリー


なあ梓ー 開発はまだ始まらないのかー?


いよいよ今回から開発が始まります!
と言ってもまだプログラムは書きません
前回でてきた「ストーリー」をみんなでつくっていきます


「ストーリー」・・・
たしかプロジェクトで作る機能の事よね?


(よく覚えてるな・・・ ムギがプロジェクトマネージャやればいいのに・・・)


そうです
ストーリーは一般的に「誰々が何々できる」という形式で書いていきます
例えば「ユーザはSSを閲覧できる」みたいにです
みんなでストーリーを出していきましょう!


そうだなー
「管理者はSSを書ける」とかどうだー?


いいですね! そんな感じです


「管理者は顔アイコンが登録できる」なんてどうかなー


なるほど~ 文章だけじゃなくアイコンも付けたかったですね
ストーリーの洗い出しはプロダクトオーナーが一番がんばるチャンスです!


そういえば、出た意見はどうするんだ?


最初は付箋紙に一つづつ書いていきましょう
あとでまとめてPivotal Trackerに登録です
ツールを使わずに付箋紙でアナログにやっていくのも楽しいポイントですよ
みんなが参加している感はとても重要です!


.....


よし、だいたい出尽くしたぞ!
これだけでどんなアプリケーションになるか想像できるわ!


次はストーリーひとつひとつに「ストーリーポイント」を振っていきます
ストーリーがどれくらい難しそうかを基準にポイントをつけていきます


1ポイントはどれくらい難しいんだ?


1ポイントがどれくらい難しいかは気にしません
大事なのは2ポイントは1ポイントより2倍くらい難しいということです
開発を進めてチームが安定してくると、1ポイントがどれくらいの難しさで、
どれくらい時間がかかるかがだんだん分かってきます


さっそくやっていきましょうか~
でも私、開発者じゃないからどれくらい難しいかわからないわ


仕方ないので今回は私ひとりでストーリーポイントを振っていきます
でも、なにか気になるところがあったら指摘してください

例えば「管理者は顔アイコンが登録できる」に小さい数字を振ったらどうしますか?


そうね、「正方形にリサイズ」を忘れてないかしら?
もっと難しくなりそうよ!!


良い指摘ですね!
もしその指摘がなかったらのちのち大変なことになっていたかもしれません
そんな感じでストーリーポイントを振っていきます


...

さて、全部のストーリーポイントを振り終わったぞ
これをPivotal Trackerに登録すればいいんだな


そうですね
登録は雑用係の律先輩がやってくださいね


(えっ・・・めんどくさい・・・ イタズラしちゃえ)

えっと、 「唯は梓をペロペロできる」っと


何してるんですか、律先輩


どんなストーリーが作られたかは、
Pivotal Tracker の「DONE」のタブで一覧できますよ
https://www.pivotaltracker.com/projects/267615#

解説

ストーリーポイント、モノは試しでやってみてください。
だんだん見積もりの辻褄が合ってくるのが実感できると思います。

アジャイル全体は導入できなくても、ストーリーポイントの「相対的に見積もる」という考え方はすぐに導入でき、非常に効果的です。
また、ストーリーを洗い出すミーティングは、想像できない楽しさがあります。
開発で一番面白いところと言っても過言じゃないかも。
「見積もりポーカー」というトランプも売ってたりします。よりゲームっぽく楽しめます。

第5話　イテレーション


準備の最後に開発の流れを説明しておきます
これが終われば次から開発ですよ


キタコレ!!


開発は1週間単位でおこないます
これを「イテレーション」といいます
1回で消化できるストーリーポイントの合計を「ベロシティー」といいます


Pivotal Trackerの右上に書いてある数字がそれのことか


Pivotal Trackerは直近3回分の平均値をとっています
イテレーションを繰り返すとだいたい数値が安定していきます
これがチームの開発スピードの目安になります


だからストーリーポイントの1ポイントの重みは気にしなくて良いのね!


繰り返すうちにストーリーポイントを付ける癖も固まってきますから
見積もりがだんだん正確になっていきますね


なるほど便利だなぁ
だんだん正確になっていくってところがポイントだな


あと毎日の日例ミーティングと
イテレーション終了時に振り返りミーティングを行います


日例ミーティングでは
・昨日やったこと　・今日やること　・今起きている問題
を全員が共有します
特に重要なのは 今起きている問題 です


問題って・・・例えばどんな?


・~~の仕様に漏れがありそう とか　・~~得意な人がいたら手伝って欲しい とか
・誰かセキュリティのチェックして とか
なにか不安を感じていたら全て言ったほうがいいです
・今日ちょっと体調が悪い なんてのも結構重要です


で、デコが・・・心配・・・


次は振り返りミーティングです
ここではKPT(ケプト)と呼ばれる反省会をします


反省会・・・


KPTとか Keep Problem Try の頭文字で
・よかったので続けたいこと（Keep）
・解決できてない問題（Problem）
・次のイテレーションでやってみたいこと（Try）
を共有します


毎日ミーティングとか KPTとか めんどくさくないか?


それが意外に楽しいんですよ
ロールプレイングゲームみたいな感じです


あと大事なことをひとつ
githubのリポジトリのコードは常に動く状態をキープします


じゃあ私はいつでも試せるのね!


次からいよいよRailsで開発ですよ

解説
Web開発だとイテレーションを意識することはあまり無いかもしれません。
開発とアップデートを短期間で(場合によっては1日に何度も!)繰り返すので、
イテレーションという単位は大きすぎることがあります。
毎日のミーティングと週次の振り返りのための概念になっています。

第6話　ｒａｉｌｓ　ｎｅｗ


今回から開発していきますよ!
まずはrailsをインストールします
$gem install railsとコマンドを打つだけで rails がインストールされます


ちょっと時間がかかるけど... いろいろインストールされたみたいだ!


railsのインストールが終わったらさっそくrailsプロジェクトを作りましょう
$rails new ss_editorで ss_editor
というディレクトリにrailsアプリケーションが作られます

create ~~ ってたくさん表示されたぞ
とりあえず ss_editor に移動して...
app とか config とかいろいろファイルがあるな

さっそくサーバを動かしてみましょう
$rails serverとコマンドを打ってください

えっ 何も触ってないのにもう動くのか?
あれ? なんかエラーが出たぞ?
Could not find gem 'sqlite3 (>= 0)' in any of the gem sources listed in your Gemfile. だって

ああ、必要なgemが入ってないみたいですね

さっきみたいに
$gem install sqlite3ってやればいいのかしら?

いえ、bundlerという仕組みを使います
$bundle installと入力すると Gemfile
というファイルに書かれたgemを自動でインストールしてくれます
こうすることで他の人の環境でも簡単に整合性を保つことができます

sqlite3 をインストールしたらサーバが起動するようになったぞ

それではブラウザで http://localhost:3000/ にアクセスしてみてください
Welcom aboard というページが見えますか?

Railsアイコンのページが見えたわ!

あとは開発に必要なおまじないをいくつかします
rspec-rails という gem を使うので律先輩、追加してみてください
ただし development と test の環境でのみ使います

んーと、Gemfile に書くんだな・・・
下の方に group :development, :test do ~~ end というのがあるな
この中に書けばいいんだな!

それで正解です!
あっ 行頭に # があるとその行はコメントなので # も消しておいてください
保存したら bundle install です

rspec-rails のインストールが終わったら rspec を使う準備をします
$rails generate rspec:installとコマンドを打ちます

ところで rspec ってなんだ?

テストのフレームワークです


テスト・・・・ッ!!!!!!!!


別に学校のテストじゃないんですからそんなに怖がらなくても・・・
とりあえずrailsの準備はこれで終わりですね

解説
数回コマンドを打つだけでサーバが起動して準備完了。素晴らしいですよね。

第7話　gitを使う


railsの準備ができたのでgit管理を始めましょう


確かgithubを使うんだよな!


その前にgitで管理する宣言をしないといけません
ss_editorのディレクトリで
$git init .と打ってください


Initialized empty Git repository in ss_editor/.git
って表示されたぞ!


それでgit管理が始まりました
$git statusと打ってみてください


Untracked files ってところに赤い文字でファイル名がいっぱい出てるな


まだgit管理下に置かれていないファイルの事ですね
すべてgitに追加しましょう
$git add .です


Changes to be commited に変わったぞ
今度はファイル名が緑色だ
. は全部のファイルって言う意味なのか


そうしたらコミットします
コミットはリポジトリに変更を反映する操作のことですね
$git commit -m 'プロジェクト開始'でコミットします
-m でコミットメッセージを書きましょう


[master xxxxxxx] プロジェクト開始
って書かれたぞ
これでコミットできたのか?


statusを見てください
nothing to commit と書かれていますね
commitは成功して最新から変更されている部分はないという意味です

やりましたね律先輩! すごいですよ!


おおぅ


さて次は変更をgithubに反映させます
まずはgithubを共有先として登録しましょう
$git remote add origin
(githubのurl)ですgithubのurlはプロジェクトページに書かれています


SSH | HTTPS | Git Read-only の3つがあるな
これのSSHを使えばいいのかな


登録が終わったら
$git push origin masterでローカルの内容をgithubに共有します


pushしてみたぞ!
githubにも「プロジェクト開始」って書かれてるな!


これでgithubとコード共有ができました
次からいよいよコードを書いていきます

解説

もうgitがないと生きていけない体になりました。手元にリポジトリがあるので無限にブランチできます。
どんな無謀な思いつきもブランチ切ればコミットし放題。無謀すぎたら消せば良い。
subversionには戻れません。もちろんバージョン管理無しの世界なんか論外です。

第8話　scaffold


さて、Pivotal Trackerのストーリーからひとつ選んで機能を作っていきましょう
まず「ユーザはSSを読める」とかにしましょうか
Startボタンを押してストーリーを開始してください


ど、どこから手をつけていいんだ・・・


railsアプリケーションは現実世界の「モノ」を対象に作っていきます
さあ「ユーザはSSを読める」で対象になる「モノ」はなんだと思いますか?


んーーー 「ユーザ」か「SS」のどちらか・・・
たぶん「SS」かな?


そうです! 正解です!
SSだと分かりにくいのでstoryと呼びましょう
storyを作るには次のようにします
$rails generate scaffold story


おおっ 色々ファイルが作られたぞ
story とか stories とか書いてある


まずデータベースのテーブルを決めます
db/migrations/**_create_stories.rb
というファイルがあると思います


なんか create_table :stories do |t| ~ end ってのがるな
この中になにか書けそうだな


そこにテーブル定義を書くとテーブルを作ってくれます


で、テーブルってなんだ!?


あ、そっか
テーブルというのはいくつかの情報を持った入れ物ですね
ストーリーはどんな情報を持っていると思いますか?


んーと とりあえずタイトルだろ
あとタイトルと・・・ タイトル?


一緒じゃないですか!
ムギ先輩は何が必要だと思います?


そうね、公開するかしないかの設定が欲しいわ
下書きがないと困るの
それ以外には浮かばないわね


じゃあとりあえずタイトルと公開フラグですね
t.string :title, :null => false
t.boolean :published, :default => false
と書いてみてください


なんとなく何がしたいのか分かるな
title と published っていう情報を持ったテーブルができるんだな
published ってなんで過去形なんだ?


先輩・・・これは過去形じゃなくて受動態ですよ・・・
This story is published. だから受動態です
なるべく英語の文法として正しくしたほうがいいです


え、英語・・・


じゃあテーブルを作っちゃいましょう
$rake db:migrateと打ってみてください


create_table(:stories)
って出てきたぞ!
多分これで作れたんだな~


じゃあもう一回migrateしてみてください

えっ もうテーブルできてるのにまたやったらおかしなことにならないのか?

大丈夫ですよ
一回実行されたmigrateはもう実行されないんです
だから多人数で開発してもテーブルの定義に一貫性を持たせることができるんですね

へぇーー便利だなぁ

これでもうSSを作ることができます
http://localhost:3000/stories
にアクセスしてみてください

new story ってリンクがあるわ
リンク先はcreate storyってボタンがあるだけのページね
押したら・・・エラーになってしまったわ

タイトルを入れてないからですね
とりあえずこの段階でいったんgitにコミットしておきましょう
でも今はmasterブランチにいます
新しいブランチを切ったほうがいいですね

ブランチってなんだ?

ブランチとは枝分かれのことです
ストーリー機能の開発用に別の歴史に分岐するんです
そうすることで本流の歴史に影響なく開発をすすめることができます
もちろん開発が成功した機能を本流の歴史に合流することもできます

そういえばgithubのコードは常に正しく動く状態って言ってたわ!
ブランチがあるから開発中のコードが混ざらないのね!

そうなんです!
$git checkout -b create_storyで create_story
というブランチを作ってそのブランチに切り替えます
ブランチ名はつくる機能を表したものにしましょう!

新しいブランチに移動したらそこでコミットすればいいのか?
$git add .$git commit -m 'storyのテーブルを作る'っと

gitk という履歴をみるプログラムがあるのでそれで見てみましょう
[master]プロジェクト開始 より [create_story] storyのテーブルを作る
の方が先に進んでいるのが解ると思います

こうやって絵で表示されると分かりやすいわ!

今回はこのくらいにして
次回はデータを保存できるようにしましょう!

解説
scaffoldは良コードの宝庫です。Railsがよくわからないという人はscaffoldが生成したコードを穴が空くほど見てみましょう。
何か迷ったときはだいたいレールを外れようとしている時です。scaffoldを参考にレールに戻りましょう。

第9話　storyを完成させる


実際にコードを書いていく前に
まず今の状態でテストしてみましょう
$rakeと打ってください


ひッ テストッ!!!


あら ほとんど落ちてますね
ほら赤い点がいくつも・・・


赤点ッッッッッッ!!!!!

追試ッ・・・
ぎゃあアアアアアアアアアアアアアアアアアアアア


テストは怖くありません!
それに
追 試 は 何 千 回 も し ま す よ ？


・・・・・・・・・


とりあえず今の状態でテストが通るようにしましょう
ログを見たところデータベースのNULL制約で怒られているようなので
spec/controllers/stories_controller_spec.rb を開いて
valid_attributes に {:title => 'title'} 書いてください
ほら通りました


赤点なくなったあああああああああ!!!


とりあえずこの段階でgitにコミットします
そしたら次はテストを書いていきます
spec/models/story_spec.rb を開いてください


最初にテスト書くのか


ちゃんと保存できるかどうかのテストから書きましょう
保存できるかのテストは (object).should be_valid や .should_not be_valid
といった書き方をします

~~はvalidであるべきである ってことなのか
validっていうのは・・・ 「妥当な」という意味か

例えば問題ない場合のテストは
it ”タイトルがあれば妥当” do
Stroy.new(:title => 'title').should be_valid
end
と書けますね

タイトルがない場合
it ”タイトルがない場合は妥当じゃない” do
Story.new.should_not be_valid
end
でいいのか?

それでいいと思います
今回は単体のテストで十分なので
$rspec spec/models/story_spec.rb -cfsでrspecを実行してください

タイトルがない場合のテストが赤い・・・
赤点・・・ ガクガクブルブル

ちゃんとテストが落ちましたね!
じゃあこのテストが通るようにStoryのモデルにコードを書きましょう

この・・・テストが・・・通るように?
赤点取ってからテスト勉強するようななんか変な感じだな

い つ も の こ と じ ゃ な い で す か
妥当性チェック(validation)を付け加えるのはすごく簡単です
app/models/story.rb に1行
validates_presence_of :title
とくわえるだけです

おおっ たったそれだけでテスト通った!
validates_ なんとかってのが他にもたくさんありそうだな

一意であることを表す validates_uniqueness_of なんかもよく使います
自分でメソッドを定義して使うこともできます

さて、全体のテストをして通ったらコミットしてしまいましょう

$rakeだな・・・よし通ってるぞ
$git add . ; git commit -m 'Storyのバリデーション設定'こんな感じでいいか

次はコントローラのテストですが今回は特に触る必要はないと思います
じゃあビューを作っていきましょうか
app/views/stories/_form.html.erb を開いてください

面倒なことはすべて form_for do |f| ... end がやってくれます
ブロックの中に %= f.text_field :title %> と書くだけで
titleのためのテキストボックスが表示されるようになります

おおっ 表示された
適当に入力してボタン押すと・・・ 登録された!
空白でボタン押すと・・・ エラーでた!
編集もできてるなぁ

あとは app/views/stories/index.html.erb
でタイトル表示だけしておきましょうか
@storiesのループの中で %= story.title %> と書いておけば大丈夫でしょう
あとはコミットしてmasterブランチにマージしましょう

そうか 今は開発ブランチだったな!
masterブランチにマージするにはどうすればいいんだ?

$git checkout masterでmasterブランチに移ってから
$git merge create_storyで create_story のブランチをマージします
もし競合(コンフリクト)が起こったら手でマージしてください

問題なくマージできたみたいだ
これで完成か?

gitは賢いので大体のマージは上手くいきます
あとは
$git push origin masterでgithubにpushして終わりです!

あと Pivotal Tracker のストーリーを完了にしないとな

解説

テスト以上にプログラミングの嫌な部分を軽減してくれるものはありません。
テストを書くことでプログラミングの楽しい部分を存分に味わうことができるのです。
グリーン、テスト追加、レッド、コード追加、グリーンの繰り返しです。まずテストを書くのです。
これがテスト駆動開発です。テストを書いておけば、自分のコードが正しいかどうかは一目瞭然です。

テストは慣れです。最初は何をどうテストすれば良いのか分からないと思います。
そういう時はscaffoldで作られたテストを見ましょう。他の人のコードを読みましょう。
慣れればテストを書くのはとても簡単です。

テストは常に繰り返し実行することが重要です。
自動でテストを繰り返すauto_testや、CI(継続的インテグレーション)ツールも色々あります。
導入しておいたほうがいいでしょう。

第10話　リレーション

次はどのストーリーをやりましょうか
「SSに会話を登録できる」なんてどうでしょうか

storyと同じようにやればいいんだな!
scaffoldで会話のファイルを生成すればよさそうだな
会話って英語でなんて言うんだろう・・・

会話はdialogがいいんじゃないでしょうか
どんなカラムが必要だと思いますか?

会話の内容だろ・・・ 喋る人だろ・・・ 他には?

どのストーリーの会話かって言うデータも持たないとダメですよね

なるほど! そういう目に見えないデータも必要なのか!
どうすればいいんだ?

こういう他のテーブルとの関係をリレーショナルといいます
関連性を定義するカラムは story_id というように 対象モデル名_id
の形式のカラム名をつけます

別の名前にするとどうなるんだ?

ルールに従うとRailsがすべて自動で処理してくれるんですが、
ルールから外れると面倒な設定を自分で書かないといけなくなります

それは面倒だな・・・
できるだけルールに従ったほうがいいんだな

ん? じゃあこの「喋る人」もストーリーと一緒でリレーションになるのか!?

よく気づきましたね!
今はまだ作りませんが、後々 character という名前で作ることにしましょう
その場合カラム名は何になりますか?

character_id だな!

そのとおりです!
先に必要なカラムがわかればこんなscaffoldの書き方もできます
$rails g scaffold dialogs message:string story_id:integer
character_id:integer line_num:integer

最初からmigrationのファイルに定義が書かれてるのか!
_form.html.erb にもinputタグが書かれてるな

. . . な ぜ 最 初 に 教 え な か っ た

次はモデルに関連性を定義しましょう
「DialogはStoryに属するので」 Dialog には belongs_to :story と書きます
dialog.story でStoryにアクセスできるようになります

belongs_to :character_id を付けると喋る人が取得できるのか!

逆の立場だと「StoryはDialogを持っている」って言い換えることもできるぞ?

逆の立場は has_one :dialog か has_many :dialogs と書きます
この場合、会話は複数になるので has_many :dialogs ですね
story.dialogs で会話の一覧にアクセスできます

ルールに従うだけでこんなにやってくれるなんて・・・

次は /stories/1/dialogs というアドレスでアクセスできるようにしましょう

id=1のストーリーの会話っていうのがなんとなくわかるぞ

RESTという考え方に従っています
簡単に言うとリソースを表すURLとHTTPメソッドの組み合わせで、
何が起こるか決定しようという考え方です
例えば /stories/1/dialogs/12
にDELETEでアクセスした場合どうなると思いますか?

id=1のストーリーのid=12の会話文が削除される?

そうです 何も悩まなくても一瞬で分かりますよね!

config/routes.rb を見てください
resources :dialogs ってありますよね

:dialogs と :stories もあるな

```
resources :stories do
resources :dialogs
end
```
という方法で親子関係を定義できます

/dialogs というアクセスができなくなるのでテストが落ちまくることになります
テストを修正してその後コードを修正していきましょう

urlを指定している部分が dialogs_path みたいになってるのはどうするんだ?

そこは story_dialogs_path(@story) のように書き換えることができます
または [@story, :dialogs] のように配列で渡してもいいです

@storyはどこからくるんだ?


コントローラで params[:story_id] でストーリーidを取得することができます
コントローラ内で @story = Story.find(params[:story_id])
と言う感じに取得しておけばいいでしょう


なるほど、けっこう機械的に修正するだけで大丈夫そうだな
よーし全部テスト通ったぞ


じゃあ「会話を登録できる」のストーリーは完了ですね

解説
レールに乗っかると最も楽になるのがActiveRecord、特にリレーション関係です。
全てまるっとRailsが面倒事を引き受けてくれます。
そのためにはテーブル定義から正しくRails的な設計をする必要があります。変なこだわりは捨てましょう。
レールに乗っていれば幸せになれるのです。

第11話　ファイルアップロード


キャラクターと顔の登録にとりかかりましょう
ファイルアップロードの処理をする必要がありますね


なんかスッゲーめんどくさそうだな・・・


そうです とてつもなくめんどくさいです
なのでプラグインを使って全部やってもらいましょう
めんどくさいと思ったらプラグインを探す 鉄則です


で、どんなプラグインを使うんだ?


プラグインを探すのがまた難しいんです
たくさんありすぎてどれを選んでいいのか見極めるのが難しいです


どんな基準で選べばいいんだ?


今一番新しくて勢いがあるものです
Railsの進歩が速いのでそれに合わせてプラグインの流行り廃りも速いです
検索は過去に流行った物が表示されるので向きません
新しめのブログ記事なんかを見極める必要があります


それ欲しい機能を実際に書くより難しいんじゃないのか・・・


それを考えてはダメです・・・
実際もっと上手い探し方を知りたいですよ・・・


で結局どれ使えばいいんだ?


paperclipというgemを使いましょう
最近これが流行ってるみたいです
プラグインプラグイン言ってましたが正解はgemでしたね


paperclipのgithubにどうやって使うか書いてありますね
ファイルの属性を保存するためのカラムをいくつか追加すればいいようです
リサイズも自動でやってくれるみたいですよ


インストールに失敗するぞ・・・


ImageMagickという画像編集ライブラリが必要みたいです
パッケージ管理ソフトでlibmagickを検索して出てきたものを片っ端からインストールしてみてください


おおインストールできた!


じゃああとはドキュメントに従ってやってみましょうか


おお・・・簡単にできてしまった・・・

テストはどうするんだ?


わからないです
わからないですよ
stubでごまかしましょう
それでいいです それでいいです


なんで怒ってるんだ・・・


テストしやすいようにメソッドを切るのも重要テクニックですね
例えばアップロードした画像は @face.avatar.url(:thumb)
でアクセスするんですが
これじゃテストしづらいです
@face.thumbnail_url というメソッドをつくるとまるごとstubにできますね

解説
高度に発達したプログラミング言語やフレームワークで一番問題になるのは、
自分の欲しい機能を実現ために、どのライブラリを使うべきということです。
ライブラリ自体はググればいくつも見つかります。問題は果たしてそれを使っていいのか。

そのライブラリが良いか悪いかを決めるのは、コードの実装ではありません。
処理速度でもありません。将来どのように保守されるかが一番重要です。
ポイントは
・とにかく新しいこと ・作ってる人がマトモなこと ・今一番流行しているらしいこと

・最近アップデートされていること ・ドキュメントが英語であること
このあたりをチェックすれば、そのプラグインに将来性があると見ていいと思います。
そして、いずれそのライブラリも廃れる時代が来ることも覚悟しておくべきです。
その時に、ライブラリに依存しすぎていて、どうにもならないということにならないように気を付けるべきです。

プログラミングで一番難しいのはライブラリ選びです。
Rubyもperlもpythonも、ここで一番悩みます。
phpはそれほど悩みません。だってpearマトモに動かないし。

第12話　KPT


イテレーションが終わったのでKPT(ケプト)しましょう!


KPTってなんだっけ???


Keep Problem Try の略だったはずよ


さすがですね ムギ先輩!

イテレーションを振り返って
良かったこと 悪かったこと 次挑戦したいこと を考える時間です


良かったこと~~
railsは簡単でイイな!
あと梓がヒント出してくれるのは良かったな


自分のコンピュータでいつも試せるのが良かったわ
あとgitを使うとコードの共有が簡単ね


私は Pivotal Tracker のバーンダウンチャートが楽しいと思いましたね

なにか問題はありましたか?


メタな話をすると平日深夜と土日しか開発時間が取れないから進捗が遅いな


メタな話をするとリアル仕事も忙しくて平日の睡眠時間も削ってかなり体調が悪いわ


メタな話をすると開発と文章書きが両立しないですね
締切りがだいぶヤバそうです


あと文章って書くの難しいな
梓の特徴がつかめないぞ


いつになったら私のアイコンは用意されるのかしら・・・


問題山積みみたいですね・・・

挑戦したいことはありますか?


リアル仕事から早く帰る


絶賛デスマ中・・・
無理そうです・・・


原稿が間に合わないときのためにグッズを作りたいわ!


もう完全にメタな内容ですね・・・

まあそんな感じで次イテレーションもよろしくお願いします

解説
ホワイトボードを十字で4つに区切り、それぞれの枠に K P T と書きます。
あとはメンバー全員でジョークを交えつつ意見を出していきます。
KPTに関係ない雑感という枠を作っているチームもあるようです。

リアル仕事のあるイテレーションのKPTの例です
K: イカ おやつ
P: 進捗ヤバい メモリリーク直らん イカ臭い
T: Flasher足りないから人さらい

酷いチームだと思います

第13話　仕様変更


梓ちゃん! 梓ちゃん!
大変なことに気づいたの・・・


ど、どうしたんですか!?


梓ちゃん、私、アイコン描けないの・・・


仕様の前提が間違ってるレベルの大問題じゃないですか・・・


それでね、他の人がアップロードしたアイコンが使えればなぁー
って思うの


なるほど
今までは自サイトに組み込むような感じで作っていましたが
それだと完全にWebサービス型になりますね


今からそんな変更できるのか!?


できますよー


で、でも これっていわゆる仕様変更だろ!?
良いのか気軽に受けちゃって


問題ないです
そもそも仕様は常に変化するものなので
最初から仕様変更に対応しやすいように作っていくものですよ


助かるわ


まずどんな機能が増えるかストーリーを出しましょうか
「ユーザは登録できる」「ユーザはログインできる」
「ユーザは自分のSS一覧が見える」「ユーザは自分がアップロードしたアイコンが見える」
ぐらいでどうでしょう


それにそれぞれポイントを振るんだな


とりあえず stories と facies に誰が作ったかを表す user_id
ってカラムをつけましょう
本当なら
$rails generate migration
add_user_id_to_story_and_faceみたいに新しいマイグレーションファイルを作らないといけないんですが・・・


本当なら??


まだ本番環境にdeployしていないので create_xxxx.rb
のマイグレーションをいじってしまいましょう
編集したら db/development.sqlite3 を削除してもう一度
$rake db:migrations で作り直しましょう


新しいマイグレーションを作るよりそっちのほうがいいのか?


追加だとカラムに not_null とか
uniqueインデックスの制約を付けづらかったりしますね
既存のデータが邪魔してしまうことがよくあります


とりあえず追加したぞ
Userも作ったほうがいいのかな?


その前にどうやって認証するか考えましょう
パスワードとか持ちたくないのでOAuthかOpenIDがいいです


たしかにパスワードとか持つと変更の仕組みもつくらないとダメだろ
それはめんどくさそうだなー


OAuthするほどのことじゃないのでOpenIDにしましょうか
最近はdeviseというgemが流行りみたいですが・・・複雑すぎですね
ちょっと古いですけど open_id_authentication
というプラグインがあるみたいですよ


サンプルを見た感じこれで十分だな!


openid_urlというカラムをつけてUserを作りましょう
とりあえず今日はここまでです

解説
仕様変更は必ず起こります。起きないようにするなんてのは無理な話です。
仕様変更を極力起こさないようにするより、いつでも変更しやすい開発をすることに注力しましょう。
なるべく変更が小さく済むように、常に最新の状態のプロダクトを触って、問題点をその場で洗い出すことが重要です。
全て完成させてから客に触らせるのではなく、小さな機能ができるたびに触らせるのです。
そのために、リポジトリは常に動作する状態のプログラムになっていないといけないのです。

第14話　ログイン


ログイン機能を作りましょう
まずプラグインをインストールします
$rails plugin install
git://github.com/rails/open_id_authentication.gitgemも必要なのでrack-openidもインストールしておきましょう


サンプルだとコントローラで authenticate_with_open_id
を使えって書いてあるな
どのコントローラに書くんだ?


ここは少しレールから外れましょう
ユーザ個別の処理はdashboardというコントローラにまとめましょう
ログイン・ログアウト・ユーザページはこのdashboardで処理します


scaffoldだったところをcontrollerにすればいいのか
$rails generate controller dashboardでいいのか?
空っぽのファイルができたぞ

loginとlogoutのメソッドを作ってきましょう
まずログインしている状態がどういう状態なのかはっきりさせましょう
そうすればログアウトは簡単に作れます

ログインしている状態・・・
なんか変数が定義されているとか?

セッションに特定のデータが有るか無いかでいいと思います
OpenId の identity_url を session に入れましょう
session[:identity_url] でアクセスできます
さてログアウトはどうしたらいいと思いますか?

identity_urlがあればログイン状態なんだな・・・
session[:identity_url] = nil
で消してやればいいんじゃないか?

それで問題ないですね
ログインの方も authenticate_with_open_id の結果の identity_url を
sessionに入れればOKでしょう
そのあと新規ユーザか既存ユーザか判定します

identity_url
が一致するユーザがいなかったらユーザ登録画面に飛ばせばいいのか
ログインしてるユーザの情報が欲しいんだがどうすればいいんだ?

すべてのアクションが実行される前にログインチェックの処理を追加しましょう
application_controller の before_filter に追加すると
すべてのアクションの前に実行されます
その処理の中で @auth にユーザ情報を入れればいいでしょう

session[:indentity_url] でユーザを検索して @auth
に突っ込むメソッドを作ればいいんだな

これでログインユーザ情報が使えるようになりましたよ!

ところでログインページにどうやってアクセスすればいいんだ?

そういえばルートの登録がまだでしたね
config/routes に
match "/login", :to => "dashboard#login", :as => :login
のように書かないといけませんでした

おお、これで login_path とかも使えるようになったぞ
/login にアクセスしたらログイン処理もしてくれるみたいだ

ちょっとレールを外れるだけでだいぶ面倒ですね
deviseならこの辺りもうまく処理してくれるんでしょうか?

そういえばテストは?


authenticate_with_open_idの部分はどうやってテストすればいいか分かりません・・・
他の部分はいつも通りテストできると思いますよ

解説
もう個人がちんまりやってるWebサービスに、自前の認証なんて持ってはいけません。
どれだけ気をつけてもバグは無くせないし、セキュリティホールも無くせません。
そもそもOSにもセキュリティホールは存在するので、どれだけアプリが気をつけても意味がない部分もあります。
OauthやOpenIdなど、信頼できる外部認証プロバイダを利用しましょう。
不要な個人情報も取ってはいけません。個人情報を持つ事自体が大きなリスクです。
人間の悪意が向いた瞬間は恐ろしい。P*x*vみたいになっちゃうぞ!

第15話　アクセス制御


ログインできるようになったら次はアクセス制御ですね


アクセス制御ってなんだ?


他人の作ったデータを編集できないようにしたりすることです
例えば stories#edit は他の人がアクセスできたら困りますよね


たしかに困るな


他にもログインしてない人が stories#new にアクセスできても困りますよね


それも困るな
どうすればいいんだ?


他の人のデータを触れなくするのは簡単な方法があります
Story.find(1) のようになっている部分を @auth.stories.find(1)
に変えるんです
そうすると自分のstory以外は検索に引っかからなくなります


なんかズルイな


ズルイですよ
とりあえずアクセス制限が必要なところは@authから検索するようにしましょう


そういえばコントローラのrspecでどうやってログイン状態を設定すればいいんだ?


方法は色々あります
sessionに存在するidentity_urlを入れるとか
ユーザを探すコードをstubにするとか

コントローラの@authにユーザを設定しておくとか

今回はコントローラの@authにユーザを設定しましょう
controller.instance_variable_set(:@auth, User.find(1))
のようにインスタンス変数を外から設定できます

そうすれば@authにid=1のユーザが入るんだな
これならテストできそうだな

ログインしてない場合のアクセス制限はどうするんだ?

before_filter でアクションの前にログインのチェックをしましょう
before_filter :check_login, :except => [:index, :show]
のようにアクションを限定することもできます

index と show のメソッド以外はログインチェックするってことだな
一覧ページと単体ページは見せるけど作成画面とかは見せないってことだな

check_login の処理はどう書くんだ?

全体で使えるように application_controller にメソッドを作りましょう
処理の内容は
@auth をチェックして設定されていなかったらトップページにリダイレクト
でいいと思います

意外に簡単なんだな
これをログインが必要なコントローラに設定していけばいいんだな

解説

ログイン処理を作るときに重要なのは、余計なものをセッションに残さないことです。
余計な情報を入れすぎると、ログアウト時に関連データを削除し忘れたりします。
この場合identity_urlだけを保存して、アクセス毎にUserテーブルを引いています。
Userの情報をセッションに直接入れるのも、DBとセッションの両方にデータがあることになり、
整合性が取れないバグを潜在的に生むことになるので、やめたほうがいいと思います。
毎回Userテーブルを引くことでクエリ数は増えますが、安全性と保守性の面からこちらの方がいいでしょう。

第16話　リリース

とりあえず最低限のログイン機能が完成したです

もう試せるのかしら!?

そろそろサーバにデプロイしましょうか

デプロイってなんだ?


デプロイはサーバにアプリケーションを配置することですね
ファイルをアップロードしたりデータベースをマイグレーションしたりして
最新の状態にします


手作業なのか!? 嫌だぞめんどくさそうだ!!!


もちろん手作業なんかじゃないです
capistranoというツールを使います
これもgemなのでインストールしてください
あとついでにcapistrano_colorsもいれたほうがいいですね


Gemfileに書くんだな
そのあと bundle install っと


終わったら
$capify .とコマンドを打つと初期設定ファイルが作られます


config/deploy.rb ができたぞ
テンプレートを書き換えるだけでいいのか?


それだけではちょっとダメですね
サーバ上でもrvmを使うのでその設定が必要になります
詳しくはrvmのサイトにあるcapistranoの項目を見てください


ほかにもここは面倒なところで
rvmを使うために色々設定しないといけません
rvmのサイトにあるpassengerの項目なども見てください
サーバ側にrvmのインストールとrubyのインストールも必要です


ところでサーバはどこにあるんだ?


・・・考えてませんでした
どうしましょう Railsが使えてある程度パフォーマンスが確保できるサーバは
そこそこ高い気がします


安心して、梓ちゃん!
こんなこともあろうかとデータセンタを買収しておいたわ!


やりすぎですよ・・・
まあこれでサーバは用意できました
律先輩、復習です
サーバにログインしてrvmのセットアップをしてください


・・・・


忘れたんですか?


まあサーバのセットアップは私がやります
apacheやpassengerのセットアップも面倒ですし


終わったらどうすればいいんだ?


$cap deploy:migrationsで全自動でデプロイしてくれます
セットアップ終わったのでやってみてください


な、なんか色々メッセージが表示されたぞ・・・
成功したのか?


実際にアクセスしてみましょう
ほら成功してますね


見える、見えるぞ・・・ッ!!!


じゃあ
律先輩、ドッグフードを食べてください

解説
デプロイが全自動。こんなに素晴しいことはありません。
「ファイルを置くだけで動く!簡単!!」なんてのは馬鹿の思考です。
実際にデプロイ作業というのはかなり問題が起きやすいタイミングです。
多くのサービスが、アップグレード時にわざわざ停止メンテナンスを入れるのは、そのリスクを減らす理由も大きいと思います。
誰でも同じ手順で全自動でデプロイできる、テストサーバにデプロイできたら本番サーバでも同様にデプロイできる。
デプロイ作業に再現性があると想像以上に楽になります。

最後のセリフでピンときましたか? 時間軸的にはこの後0話に続きます。

第17話　バグ報告


梓ちゃん!
あちこちでなんか変な動きをしているわ!


ど、どこですか!?


ここと、ここことか、
あとここも思ってたのと違うわ!
あとここは間違っていないけど使いづらいわ!


ちょっ 一気に言うな!
分からなくなる


BTS(バグトラッキングシステム)を導入しましょうか・・・
RedmineというBTSがあるんですがサーバの問題で導入しづらかったんですが


いいサービスを見つけました
Webサービス型のはないのか?
Pivotal Trackerじゃダメなのか?


Pivotal Trackerはちょっと役割が違いますね
あれはストーリーの単位で管理するもので
バグ管理やタスク管理はもっと細かい単位ですね


ふーむ・・・
それで、どんなサービスを見つけたんだ?


FluxFlex（http://fluxflex.com）というサービスです
RailsなどのWebアプリをホスティングしてくれるサービスなんですが、
何より凄いのがワンクリックでredmineがインストールできるんです


でもお高いんでしょう?


それがなんとアプリ3つまで無料です!


!!!!!!


これもそこで動かせばよかったんじゃないかしら・・・


http://redmine.np-complete-doj.in にRedmineを立てました
基本的な使い方はチケットをどんどん追加していってください
バグ報告の場合はBugを 新機能の場合はFeatureを選んでください


改良の提案はどうすればいいのかしら?


そうですね
Featureはちょっと意味が違いそうな気がします
カスタマイズできるので Request というのを追加してみましょうか


す、すごい勢いでムギがチケットを登録してるぞ・・・


早速バグを潰していきましょう!!
開発を始めるときはチケットを更新してステータスを「In Progress」にします
着手中という意味ですね
早速バグを潰していきましょう!!


なんでそんなに嬉しそうなんだ・・・

解説
Railsのホスティングで一番有名なのはHerokuというサービスです。
Herokuはrake一発でデプロイできたりなかなかカッコいいサービスです。
でもDBがPostgreSQLという大問題があるので今回は見送りました。

FluxFlexは日本人が作っているんですが、サービスは完全に英語で最初から海外を狙っています。
かなりガチなベンチャーなのでがんばって欲しいですね。

第18話　眠れない奴隷


さー これで開発も終わったかぁ?


そんなわけないじゃないですか・・・
ね? ムギ先輩


そうね!こんな機能やあんな機能が欲しいわ!
ここもちょっと使いづらいし!画面のカスタマイズもしたいわ!


ほら、開発はまだまだ続きますよ?
次のリリースに向かって今からストーリーを洗い出しをしましょう


おい・・・ここはアイデアのバーゲンセールか!?
いつになったら開発は終わるんだ・・・・


終わりなんてあるわけないじゃないですか


なん・・・だと・・・ッ!!!


終わりのないのが『終わり』 それが『Webサービス開発』

解説
Webサービスの開発ってホントに終わりがないんですよね。
開発が終わったときはすなわちサービスが終わるときなので・・・
18話タイトルはジョジョ5部の「眠れる奴隷」から取ったんですが、
今リアル仕事の納期と原稿の締切りで完全に眠れない奴隷になってます。
これがWeb系エンジニアのデフォルトです。

ちなみにエンジニアのジョジョ好きもデフォルトです。
ジョジョに出てくる『覚悟』という単語を『テスト』に変えると、スゴい名言がいくつも生まれます。

・ 『テスト』とは!!暗闇の荒野に!!進むべき道を切り開くことだッ!!
しっくりきますね

・ テストはいいか? オレはできている
理想の幹部ですね

・明日「死ぬ」とわかっていても、「テスト」があるから幸福なんだ!
残されたプロジェクトメンバーも安心して開発を続けられます。社畜の鑑ですね。

あずにゃんとペアプロしてる気分になれる薄い本
- 完 -

しかし開発はまだまだ続く
<= TO BE CONTINUED.

あとがき

Rails本だと思った? 残念、アジャイル本でした!!

このサークルでは初参加です。初めてエロ漫画じゃない真面目な本を作りました。
もともとけいおんメンバーがWebサービスを開発して、その開発風景をまとめて本にするという設定で始めました。
最初の案が「楽譜共有サイト」だったんですが、途中まで開発して、さて楽譜のテーブルを作るぞといったところで、自分が全然楽譜が読めないことに気づき断念しました。
楽譜共有サイトだったらキャラ設定とかストーリーとか違和感がなかったのに・・・

そこでネタを変更してSSを作るサービスの開発にとりかかりました。
なぜSSにしたのかというと、SSサイトなら使いながら原稿を書けると思ったから。
0話を見て気づいた人もいると思いますが、まさに自分が作ったサービスを自分でテストとして使って書いた0話です。
サーバにデプロイした次の瞬間から今までのことを思い出して原稿を書き始めました。
りっちゃんと一緒で思い出すのが大変でした。

本当はもっとコードを示してRailsプログラミングの解説をしたかったんですが、いかんせんSSとコードがとても相性が悪く、
見返してみるとRails本ではなく、アジャイル開発の雰囲気を伝える本になってしまいました。
コードは全てgithubに置いてあるのでそちらを見てください。
http://github.com/np-complete/ss_editor
まあ中途半端にコードの解説をするより、読み物としてアジャイル開発が表現できてればいいかなと。
次の本はきちんとコードを書くような本にしたいですね。
今回はhtmlとpdfだったので次回はきちんとTeXで・・・

当サークルは普段、ニコ生でプログラミング風景を生放送するという形で活動しています。
サービスの開発、原稿、アイコンお絵かきの全ての時間をニコ生で放送していました。
今後も開発風景をダダ流ししていくので興味があれば覗いてみてください。
http://com.nicovideo.jp/community/co600306

Enjoy Programming!
まさらっき

奥付

「あずにゃんとペアプロしてる気分になれる薄い本」

発行

NP-complete

http://np-complete-doj.in/

著者

まさらっき

http://twitter.com/masarakki

発行日

2011/08/13