MVC によるウェブアプリケーション開発 (Ridge を使った開発)

講義の中での位置づけ

- ORマッパ
- Webアプリケーションフレームワーク ← イマココ!
- JavaScript

今日の内容

1. HTTPとURI
2. Webアプリケーション概説
3. MVCフレームワーク
4. Ridge
    - この節だけボリューム多い
5. Perl WAF界隈の最新動向

6. 課題

7. 以下、Web Application FrameworkはWAFと表記します

1.HTTPとURI

- Webアプリに入る前のウォーミングアップです
- 知ってる人は聞き流してください
- Webの基本になる２つの技術
    - HTTP
    - URI

HTTP

- HTTP (Hypertext Transfer Protocol)
- 中身はテキストで書かれたヘッダと（あれば）ボディ
- リクエストとレスポンス

リクエストとレスポンスの例

curl -v を使うと中身が見られます

```
curl -v 'http://d.hatena.ne.jp/'
```

リクエスト

```
 GET / HTTP/1.1
User-Agent: curl/7.20.0 (i386-apple-darwin10.3.0) libcurl/7.20.0 OpenSSL/0.9.8
o zlib/1.2.5 libidn/1.19
Host: d.hatena.ne.jp
Accept: */*
```

レスポンス

```
 HTTP/1.1 200 OK
Date: Tue, 03 Aug 2010 22:53:06 GMT
Server: Apache
X-runtime: 84ms
X-pagemaker: IndexTop
Content-Type: text/html; charset=euc-jp
Age: 277
Content-Length: 52827
X-Cache: HIT from squid.hatena.ne.jp
X-Cache-Lookup: HIT from squid.hatena.ne.jp:80
Via: 1.1 diarysquid02.hatena.ne.jp:80 (squid/2.7.STABLE5)
Set-Cookie: b=$1$qckjv7GB$3ryJj4jkhAA5zGJ/jUzWx/; path=/; expires=Mon, 29-Jul-
30 22:53:06 GMT; domain=.hatena.ne.jp
Vary: Accept-Encoding
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=euc-jp">
...以下略
```

- **ステートレス**
  - 基本的にサーバは情報を保存しない
- メソッドが8つしかないシンプルなプロトコル
  - シンプル故に実装が簡単
  - 故に広く普及
- メソッド GET, HEAD, PUT, POST, DELETE, OPTIONS, TRACE, CONNECT
- Webアプリに必要なのはだいたい GET, HEAD, PUT, POST, DELETEくらい

その中でも

- 日常的に使うのは GET, POSTのみ
- GET
  - リソースの取得
  - パラメータはURIに入れる
- POST
  - リソースの作成、変更、削除
  - 変更、削除は本来ならPUT, DELETEメソッドでやるべきだが、HTMLのformがGET/POSTしかサポートしないためPOSTで代替するのが一般的
  - パラメータはURIとは別
  - URI長の制限を受けない

ステータスコード

- HTTPレスポンスではステータスコードを返さなくてはならない
- リダイレクト、エラーハンドリング等を行うため、正しいステータスコードを返そう
- Webエンジニアは大抵暗記してる
  - 僕はしてないです

代表的なステータスコード

- 200 OK
- 301 Moved Permanently
  - 恒久的なリダイレクト
- 302 Found
  - 一時的なリダイレクト
- 400 Bad Request
  - リクエストが間違い
  - クライアント側の問題
- 404 Not Found
  - リソースがない
- 500 Internal Server Error
  - アプリケーションのエラー
  - たぶん今日よく見ることになります
- 503 Service Unavailable
  - 落ちていると出る
  - よく見る

URI

- URI (Uniform Resource Identifier)
- 統一的なリソースを指し示すもの
- **URIは名詞である**
- OK: http://example.com/bookmark?id=1
- NG: http://example.com/bookmark?action=update&id=1
  - メソッド名がURIに入ると、リファクタリングなどでURIが変わってしまう

クールなURIの恩恵

- 検索、ソーシャルブックマークなどでURIが分散しない
  - ずっと変わらず統一的なリソースを指し示す
  - PV、収益的にもGood!
- ユーザビリティを向上させる。
  - サイトの構造を意識させることができる

HTTPとの関係

- URIは名詞、HTTPメソッドが動詞

```
GET http://example.com/bookmark?id=1
```

```
POST http://example.com/bookmark.edit
```

ここまでのまとめ

- HTTP
  - テキストベースのシンプルなプロトコル
  - GETでリソースの取得
  - POSTでリソースの作成･削除･更新
- URI
  - リソースを指し示すもの
  - クールなURIは変わらない
- URIは名詞、HTTPは動詞

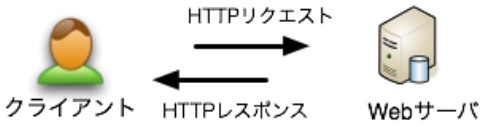
2. Webアプリケーション概説

Webアプリケーションの基本

- モチベーション
  - 動的なWebページをつくりたい
- 基本的な動作
  - リクエストから何らかの表現（HTML等）を動的に作ってレスポンスを返す

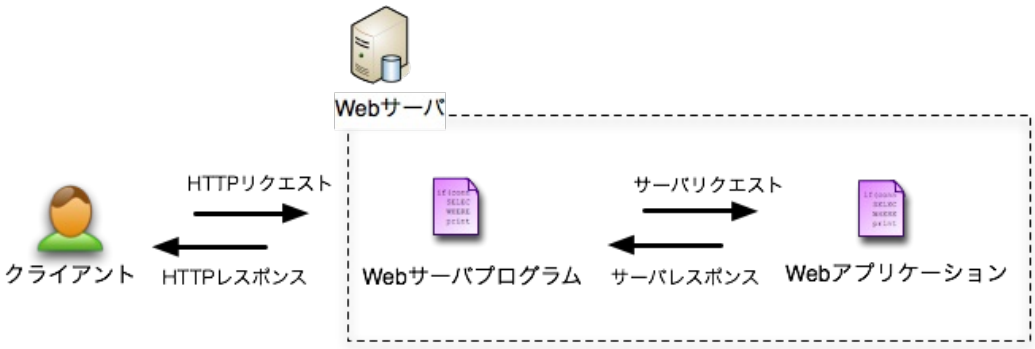
Webアプリの動作

最もシンプルな図



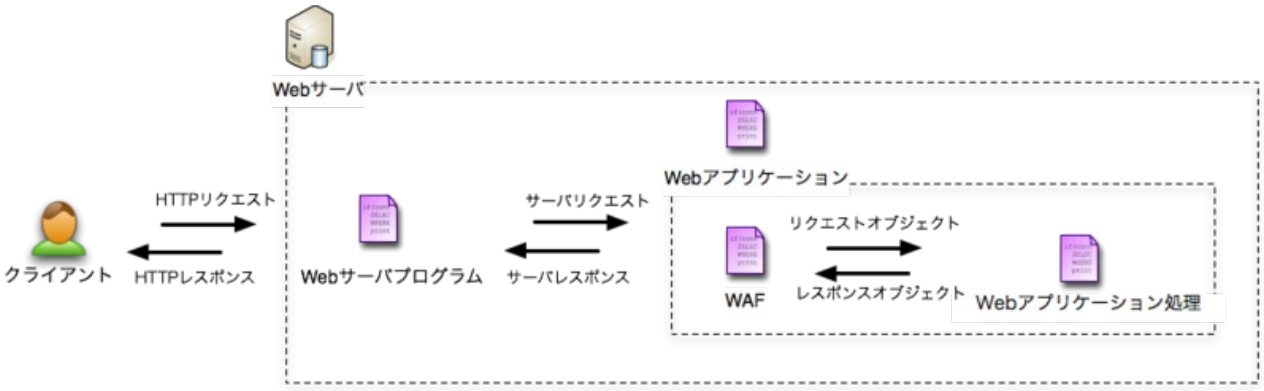
- 動作
  - サーバが *クライアントから* HTTPリクエストを受けとる
  - サーバが *クライアントに* HTTPレスポンスを返す

サーバとアプリケーションを分離した図



- 追加された動作
  - アプリケーションが *サーバから* サーバリクエストを受けとる
  - アプリ-ションが *サーバに* サーバレスポンスを返す
- Webサーバプログラム
  - Apache, lighttpd, Tomcat...
- サーバリクエスト、サーバレスポンスはサーバのインターフェイス依存
  - mod_perl, FastCGI

WAFとWebアプリケーション処理を分離した図



- 追加された動作
  - WAFが *サーバから* サーバリクエストを受けとる
  - Webアプリケーション処理がWAFからリクエストオブジェクトを受けとる
  - Webアプリケーション処理がWAFにレスポンスオブジェクトを返す
  - WAFが *サーバに* サーバレスポンスを返す
- WAF
  - サーバとの対話を仲介、抽象化する
- Webアプリケーション処理
  - ビジネスロジック、DBアクセス、HTML生成など...

ここまでのまとめ

- WebアプリケーションはHTTPリクエストに対し、動的にHTTPレスポンスを返す
- サーバ側はWebサーバプログラム、WAF、Webアプリケーション処理に分けられる
- WAFを使えばWebアプリケーション処理の実装に集注できる

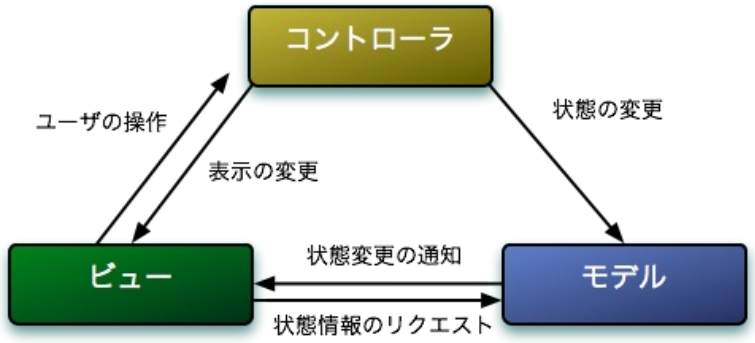
3. MVC

- 先ほどのWebアプリケーション処理の中身を解説します

MVC

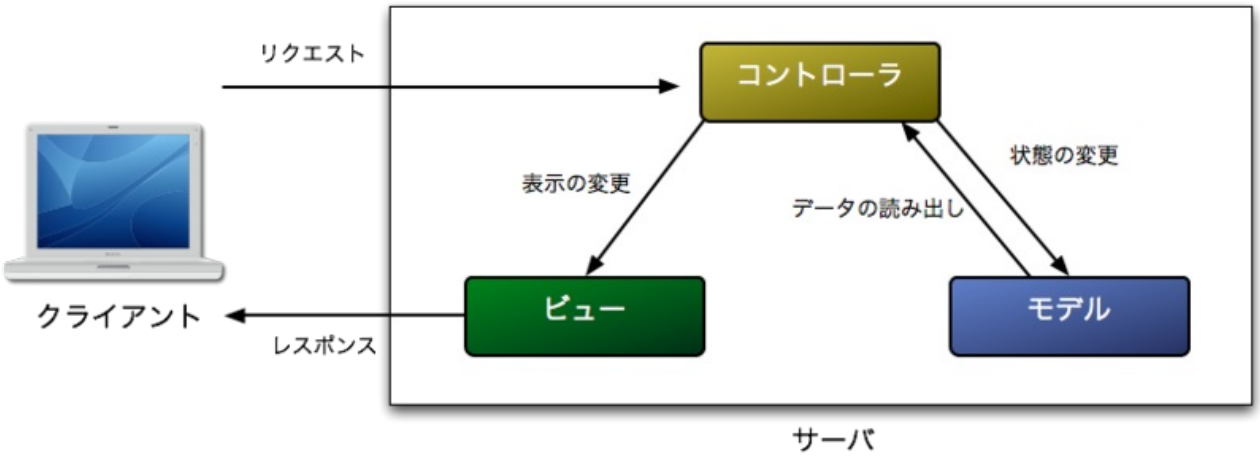
- Model, View, Controller
  - 表現とロジックを分離
  - デザイナーとエンジニアの作業分担を促進
  - テストがしやすくなる
- GUIプログラムのデザインパターンのひとつ
  - Smalltalkに由来
  - Webアプリケーション以前より存在

古典的なMVC



- GUIのMVCフレームワーク
  - Cocoa
  - .NET Framework (たぶん)

WebアプリケーションのMVC



- Webアプリケーション用に再定義された
  - MVC Model 2と呼ばれることも

MVCの中身

- Model
  - 定義では: 抽象化されたデータとそのデータに関するロジック
  - Webでは: ORマッパ、ビジネスロジック
  - はてなでは: DBIx::MoCo
- View
  - 定義では: リソースの表現
  - Webでは:　HTML, JSON, XML, 画像等を生成するもの
  - はてなでは: Template, JSON::XS
- Controller
  - 定義では: ユーザの入力によって処理の流れを決定し、Model の API を呼び、View に必要なデータを渡す
  - Webでは: Webアプリケーションフレームワーク
  - はてなでは: **Ridge**

ここまでのまとめ

- MVCとはModel, View, Controllerにより表現とロジックを分離したもの
- 表現とロジックの分離により、デザイナーとエンジニアで作業が分担できる

4. Ridge

- この節長いです
- 4.1 Ridgeとは
- 4.2 bookmark.plをWebアプリにする
  - 4.2.1 URI設計
  - 4.2.2 Ridgeプロジェクトを作る
  - 4.2.3 URIに対応したコードを書く
  - 4.2.3.1 RidgeのAPI
  - 4.2.3.2 （必要最低限の）HTML入門
  - 4.2.3.2 Template-Toolkit入門
- 4.3 その他 Ridgeの便利機能たち

4.1 Ridgeとは

- はてな社内WAF
- id:naoya ら
- Hatena → Hatena2 → Ridge

現代的なWAFの特徴多数

- MVC
- URIからクラス、メソッドへの動的なディスパッチ
- プラグインサポート
- スケルトン生成
- ローカル開発用のサーバが付属

薄いフレームワーク

- ORマッパーやテンプレートは付属しない
- 好きなものを組み合わせて使うスタンス
  - 実際に社内で標準化されているので何でも使えるわけではない
- **読んでわかるフレームワーク**
- 足りないものはCPAN or 自分で作る

では、いよいよ

- Ridgeを使ってWebアプリを作ります

4.2 bookmark.plをWebアプリにする

- 機能
  - 一覧 (list)
  - 表示
  - 作成 (add)
  - 削除 (del)
- 課題では前回の diary.pl を Web アプリにしてもらいます

4.2.1 URI設計

はてなでは

- スキーマ設計
- URI設計

の順に行うので、まずURIを考えます

Bookmarkアプリでの要件

パス	動作
/	ブックマーク一覧
/bookmark?id=id	ブックマークの permalink
/bookmark.add?url=url&comment=comment (POST)	ブックマークの追加
/bookmark.delete?id=id (POST)	ブックマークの削除

既に出来上がったものがこちらに

ここでデモ

今回の遷移図

- これから作り方を見ていきます
- お手元にコードが欲しい方はこちらに
- motemenさんの Intern-Bookmark の続きとして作ってあります

前回のIntern-Bookmark-2011のリポジトリで

```
 git checkout -t origin/ridge
git submodule update --init
```

または再度下のようにして別の場所にclone してもよいです。

```
 git clone https://github.com/hatena/Intern-Bookmark-2011.git
cd Intern-Bookmark-2011
git checkout -t origin/ridge
git submodule update --init
```

4.2.2 Ridge プロジェクトを作る

```
 % ridge.pl Bookmark
created directory "Bookmark/lib"
created "Bookmark/lib/Bookmark.pm"
created directory "Bookmark/lib/Bookmark"
created "Bookmark/lib/Bookmark/Engine.pm"
created directory "Bookmark/script"
created "Bookmark/script/server.pl"
```

```
created "Bookmark/script/create.pl"
created directory "Bookmark/lib/Bookmark/Engine"
created "Bookmark/lib/Bookmark/Engine/Index.pm" ...
```

* Bookmark ディレクトリと以下にいろいろなファイル/ディレクトリが作られる * アプリケーションのモジュール名は Bookmark::Hoge になる

ディレクトリ構成

- lib/ : Perl モジュール
- script/ : Ridgeのヘルパースクリプト
- static/ : 静的なファイル(画像、CSS、 JavaScript)
- t/ : テスト
- templates/ : テンプレート

デフォルトで配置されるモジュール

- lib/Bookmark.pm
  - アプリケーションのクラス、いろんな所についてまわる
  - use base qw/Ridge/;
- lib/Bookmark/Config.pm
  - アプリケーション全体の設定はここに
- lib/Bookmark/Engine.pm
  - すべての Engine のベースクラス
- lib/Bookmark/Engine/Index.pm
  - / に対応するエンジン、中にアクションを書く

エンジン、アクションってなんぞ

- 特定のURIに対して呼び出されるハンドラ
  - URIに対応するメソッドが動的に決定されます
- コントローラ処理はここに書く

```
 # lib/Bookmark/Engine/Index.pm
# エンジン: 特定の URI に対するハンドラ
package Bookmark::Engine::Index;
use strict;
use warnings;
use Bookmark::Engine -Base;

#アクション: 特定の URI に対するアクション
sub default : Public {
   my ($self, $r) = @_;
   $r->res->content_type('text/plain');
   $r->res->content('Welcome to the Ridge world!');
} 1;
```

リクエストパスと エンジン、テンプレートの対応

- パスによって エンジン名、テンプレートファイル名が決まる
- 階層 → エンジン名
- "." 以降 → アクション

パス	Engine+Action	テンプレート
/	Engine::Index::default()	templates/index.html
/index.hoge	Engine::Index::hoge()	templates/index.hoge.html
/foo	Engine::Foo::default()	templates/foo.html
/foo.hoge	Engine::Foo::hoge()	templates/foo.hoge.html
/foo/	Engine::Foo::Index::default()	templates/foo/index.html
/foo/bar	Engine::Foo::Index::Bar::default()	templates/foo/bar/index.html

- ※ "hoge.json" というパスではアクションは json にならず Engine::Hoge::default() が呼ばれ、ビューが JSON::Syckになる
- (BK) JSON::Syckはobsoluteなので JSON::XSを使って自前で書こう

URI と Engine のマッピング をもう少し

- / は暗黙的に /index の省略とみなして Index.pm
- アクション(. で区切られた以降の指示子)が明示的に指定さ れていない場合は、default()
- /hello は Engine/Hello.pm
- /hello/ は Engine/Hello/Index.pm
- /index.json は Engine/Index.pm の default() が呼ばれる

ビュー

- ふつう Template (というモジュール) のテンプレートファイルで、生成する HTML を記述
- HTML 以外も出力できる (JSON, YAML)

テストサーバを起動

テストサーバを起動

```
 % cd Bookmark
% perl script/server.pl
Loading entire application modules from Bookmark ... Loaded 3 modules.
Server is now launched as debug mode. Server bind to port 3000, ready to accep
t a new connection
→ http://localhost:3000/
```

- 以降はファイルを更新すると自動的に再読み込みしてくれます
- ファイルを追加したときは再起動してね

4.2.3 URIに対応したコードを書く

URI設計(再掲)

パス	動作
/	ブックマーク一覧
/bookmark?id=id	ブックマークの permalink
/bookmark.add?url=url&comment=comment (POST)	ブックマークの追加
/bookmark.delete?id=id (POST)	ブックマークの削除

一覧ページ (/) を作る

- / に対応するのは Bookmark::Engine::Index
- / にアクセスすると "Welcome to the Ridge world!" と表示される

lib/Bookmark/Engine/Index.pm

```
 package Bookmark::Engine::Index;
use strict;
use warnings;
use Bookmark::Engine -Base;

sub default : Public {
    my ($self, $r) = @_;
    $r->res->content_type('text/plain');
    $r->res->content('Welcome to the Ridge world!');
}

1;
```

* デフォルトのアクションは default メソッド * $self: エンジンのインスタンス (Bookmark::Engine::Index) * $r: Ridge オブジェクト (Bookmark) * default : Public * このメソッドをアクションとして公開するという意 * ": Public" を削除すると / にアクセスしても 404 * [debug] Action "default" is not Public at /usr/local/share/perl/5.8.8/Ridge/Engine.pm line 66 * $r->res->content で出力を設定 * ここで指定しなければテンプレート (templates/index.html) を使用

Engine::Index::default を書く

- bookmark.pl の list_bookmarks() に対応

bookmark.pl

```
 sub list_bookmarks {
```

```
    my ($user) = @_;

    printf " *** %s's bookmarks ***\n", $user->name;

    my $bookmarks = $user->bookmarks;
    foreach my $bookmark (@$bookmarks) {
        print $bookmark->as_string, "\n";
    }
}
```

- エンジンがやるべきこと
  - ユーザのブックマーク一覧を取得
  - 取得したブックマーク一覧を出力 (ビューに渡す)

lib/Bookmark/Engine/Index.pm

```
sub default : Public {
    my ($self, $r) = @_;
    # とりあえずユーザは ninjinkun 決め打ち
    my $user = moco('User')->retrieve_by_name('ninjinkun');
    # ブックマーク一覧を取得
    my $bookmarks = $user->bookmarks;
    # ブックマーク一覧をビューに渡す
    $r->stash->param(
        bookmarks => $bookmarks,
    );
}
```

* ユーザからブックマークを取得するところ (モデルへのアクセス) は一緒 * $user->bookmarks * 出力するところ * $bookmarks->each(sub { print encode('utf8', $->as_string) }); * $bookmarks それぞれを標準出力へ * $r->stash->param(bookmarks => $bookmarks); * $bookmarks をビューに渡す → 後述 * ビューによって出力の仕方は変わる

3.2.2.1 RidgeのAPI

- $r は Bookmark (lib/Bookmark.pm) のインスタンス
  - Bookmark は Ridge を継承している
- リクエストに関する処理 $r->req
- レスポンスに関する処理 $r->res
- その他もろもろ
- 詳細は https://github.com/hatena/Ridge

よくある手法: Ridge オブジェクトにメソッドを追加する

- どのページでも共通のデータを $r のメソッドとして定義しておく
  - → どのエンジンでも `$r->method` で同じデータを取れる

例: ログインユーザ

```
 # lib/Bookmark.pm
sub user {
    my $self = shift;
    moco('User')->retrieve_by_name('motemen');
}
```

とすると、

先の Index.pm も

```
 # lib/Bookmark/Engine/Index.pm
sub default : Public {
    my ($self, $r) = @_;
    $r->stash->param(
        bookmarks => $r->user->bookmarks
    );
}
```

と書ける (テンプレートからも r.user でアクセスできる)

エンジンをきれいに書くコツ

- エンジンにはロジックを書かない
- そもそもエンジンはテストを書きにくい
- → 大きなアプリケーションだったら Engine と Model の中間層をひとつ作っておくのもアリ

Index 用のテンプレートを書く

- 以下の2種類の知識が必要
  - HTMLの知識
  - Template-Toolkitの知識 (また別の言語か!)

4.2.2.2 （必要最低限の）HTML入門

- 今日は以下の２つしか教えません
  - リンク
  - フォーム

リンク

- ご存知 aタグ

```
<a href="/bookmark?id=1">リンク</a>です
```

* Webのナビゲーションは基本的にリンクで

フォーム

- ご存知 formタグ

```
 <form action="/bookmark.delete" method="POST">
<input type="hidden" name="id" value="1">
<input type="submit" value="削除">
</form>
```

- POSTでリソースを更新する場合は基本的にフォームで

4.2.2.3 Template-Toolkit入門

- Template-Toolkit (TT)
  - Ridgeのデフォルト
  - perlの名前空間ではTemplate
  - Template::以下はTT用のモジュールです
  - 他のテンプレートエンジンに比べると遅いと言われることが多い
  - その分高機能
- Perlテンプレートエンジン他にも多数
  - HTML::Template (Diary, Groupでは現役), Text::MicroTempate, Text::Xslate, Tenjin...

Template の機能

変数呼び出し

```
[% foo.bar %]
```

* $r->stash->param('foo')->{bar} とか$r->stash->param('foo')->bar() とか $r->stash->param('foo')->param('bar') とかよしなに

繰り返し処理 * 配列に対する繰り返し

```
[% FOREACH item IN items %] ... [% END %]
```

分岐処理

```
[% IF x %] ... [% ELSE %] ... [% END %]
```

フィルタ

- HTMLフィルタ

```
 [% bookmark.as_string | html %]
# <script>alert(document.cookie)<script> → &lt;script&gt;alert(&quot;document.
cookie&quot;)&lt;/script&gt;
```

- URIフィルタ

```
 <a href="http://d.hatena.ne.jp/keyword/[% word | uri %]">
# キーワード → %C3%A3%C2%82%C2%AD%C3%A3%C2%83%C2%BC%C3%A3%C2%83%C2%AF%C3%A3%C2%8
3%C2%BC %C3%A3%C2%83%C2%89
```

```
 % perldoc Template::Manual::Filters
```

外部テンプレートからの読み込み

```
 [% INCLUDE header.html %]
```

マクロ

```
 [% MACRO show_title(title) BLOCK %]
<h1>[% title | html %]</h1>
[% END %]
```

VMethods, 特殊な変数...

```
 # arrayのVMethods
[% array.first %] [% array.size %]
# loop変数
[% loop.first %] [% loop.count %]
```

```
 % perldoc Template::Manual::VMethods
% perldoc Template::Manual::Variables
```

参考

- 配列、ハッシュ、スカラーに対しては大抵の操作があります
  - http://template-toolkit.org/docs/manual/VMethods.html
- 他にもマニアックな機能多数
  - □

使ってみよう

- ふつうの HTML に [% ... %] で命令を書くと展開される

templates/index.html

```
 <ul>
[%- FOREACH bookmark IN bookmarks %]
    <li>[% bookmark.as_string | html %]</li>
[%- END %]
</ul>
```

- `[% bookmark.as_string | html %]` : フィルタ
  - `<script>alert('unko')<script>` → `<script>alert(&quot;unko&quot;)</script>`
  - 特にユーザからの入力をそのまま吐くようなときは注意

Engine::Bookmark の作成

- /bookmark へのアクセスの処理

```
 % perl script/create.pl engine Bookmark
created "lib/Bookmark/Engine/Bookmark.pm"
created "t/app/bookmark.t"
```

/bookmark にアクセスすると server.pl では :URI: http://local.hatena.ne.jp:3000/bookmark :Engine: Bookmark::Engine::Bookmark :action: default

defaultアクションを書く

- /bookmark?id=*id*

```
 package Bookmark::Engine::Bookmark;
use strict;
use warnings;
use HTTP::Status;
use Bookmark::Engine -Base;
use Bookmark::MoCo;

sub default : Public {
    my ($self, $r) = @_;
    my $bookmark = moco('Bookmark')->retrieve($r->req->param('id'))
        or Ridge::Exception::RequestError->throw(code => RC_NOT_FOUND);

    $r->stash->param(
        bookmark => $bookmark
    );
}

1;
```

テンプレートも

```
 [% bookmark.as_string | html %]
```

ブックマークの削除 (Engine::Bookmark::delete)

- /bookmark.delete?entry*id=entry*id でブックマークを削除

```
 # bookmark.pl
sub del_events {
    my ($user, $entry_id) = @_;
    $user->delete_bookmark($entry_id)
        or die "Couldn't delete bookmark with $entry_id";
}
```

```
 # lib/Bookmark/Engine/Bookmark.pm
sub delete : Public {
    my ($self, $r) = @_;
    $r->user->delete_bookmark($r->req->param('entry_id'))
        or Ridge::Exception::RequestError->throw(code => RC_NOT_FOUND);
    $r->res->redirect('/');
}
```

削除ボタンをブクマページに作っておく

```
 <form method="POST" action="/bookmark.delete">
    <input type="hidden" name="entry_id" value="[% bookmark.entry.id | html %]
">
    <input type="submit" value="削除">
</form>
```

HTTPメソッドによる振り分け

- POSTでしか削除してほしくない

```
 Ridge::Exception::RequestError->throw(code => RC_BAD_REQUEST)
```

```
    unless $r->req->request_method eq 'POST';
```

これでもいいけど

- GET/POST で処理が変わる場合、見通しが悪くなる
  - リクエストメソッドによって処理を分ける API があります

$r->follow_method

- アクションの終わりにこれを書くと
  - アクション + リクエストメソッドにより他のメソッドに移動
- default → _get, _post, ...
- hoge → *hoge*get, *hoge*post, ...
  - メソッドが定義されてなかったら 405 Method Not Allowed なのでこう書けます:

```
 sub delete : Public {
    my ($self, $r) = @_;
    $r->follow_method;
}

sub _delete_post {
    my ($self, $r) = @_;
    $r->user->delete_bookmark($r->req->param('entry_id'))
        or Ridge::Exception::RequestError->throw(code => RC_NOT_FOUND);
    $r->res->redirect('/');
}
```

* (BK)返値が$r->follow_methodの返値でないといけない * 以下は×

```
 sub delete : Public {
    my ($self, $r) = @_;
    $r->follow_method;
    return 1;
}
```

ブックマークの追加 (Engine::Bookmark::add)

- GET /bookmark.add でブックマーク追加フォームを表示
- POST /bookmark.add?url=*url*&comment=*comment* でブックマークを追加

```
 # bookmark.pl
sub add_events {
    my ($user, $url, $comment) = @_;
    return unless $url;
    my $bookmark = $user->add_bookmark(
        url     => $url,
        comment => $comment,
    ) or return;
    print encode('utf8', $bookmark->as_string);
}
```

```
 <!-- templates/bookmark.add.html -->
<form method="POST" action="/bookmark.add">
<dl>
    <dt>URL</dt>
    <dd><input type="text" name="url"></dd>
    <dt>Comment</dt>
    <dd><input type="text" name="comment"></dd>
</dl>
<p><input type="submit"></p>
</form>
```

```
 # lib/Bookmark/Engine/Bookmark.pm
```

```
sub add : Public {
    my ($self, $r) = @_;
    $r->follow_method;
}

sub _add_get {
}

sub _add_post {
    my ($self, $r) = @_;

    $r->req->form(
        url => [ 'NOT_BLANK', 'HTTP_URL' ],
    );

    if (not $r->req->form->has_error) {
        my $bookmark = $r->user->add_bookmark(
            url     => $r->req->param('url'),
            comment => $r->req->param('comment'),
        );
        $r->res->redirect('/bookmark?id=' . $bookmark->id);
    }
}
```

4.3 その他 Ridgeの便利機能たち

フォームのバリデート

- フォームの入力値が正しいかをチェックする
- $r->req->form が FormValidator::Simple のインスタンスになっている
- url => ['NOT*BLANK*', *'HTTP*URL']
    - url フィールドには値が必要、かつ URL になっていないとダメ
- $r->req->form->has_error でバリデーションを通ったかどうか分かる

HTML 以外のビュー (JSON での出力)

アクションの中で

```
$r->view->available(qw/html json/);
```

とすると /bookmark.json?id=1 のように.json付きでのアクセスで、$r->req->uri->viewが'json'になる * 返したいコンテンツをJSON::XSでデコードしよう * モデルに to_hash みたいなメソッドを用意してそれをデコードするときれいですね

```
if ($r->req->uri->view eq 'json') {
    $r->res->content_type('application/json');
    $r->res->content(JSON::XS::encode_json($bookmark->to_hash));
}
```

レスポンス

```
{"bookmark":{"comment":"google","created_on":"2008-08-06 12:18:10","id":"1","
user_id":"1","entry_id":"1"}}
```

URI の変更 (uri_filter)

- /bookmark?id=1 よりも /bookmark/1 のほうがよりクール
    - → uri_filter

```
# lib/Bookmark/Config.pm
__PACKAGE__->setup({
    URI => { filter => 38e45271a03476a0b7702cc95d71df8039292644amp;uri_filter
},
});
```

```
sub uri_filter {
    my $uri = shift;
    my $path = $uri->path;
    if ($path =~ m{^/bookmark/(\d+)$}) {
        $uri->path('/bookmark');
        $uri->param(id => $1);
    }
}
```

- uri_filter は Ridge::URI (URI を継承) のインスタンスを受け取る
- uri_filter でリクエスト URI に手を入れることができる
- $r->req->param('id') ではなく $r->req->uri->param('id') になります
    - $bookmark = moco('Bookmark')->retrieve($r->req->uri->param('id') || $r->req->param('id'))

フィルタ

- アクションに入る前の前処理、後処理を処理をフィルタにまとめることができます

プラグイン

- Ridgeにもプラグイン機構があります

ここまでのまとめ

- Ridgeによる開発は
    - まずURI設計
    - URIに対応したEngineを書く
    - それに対応したTemplateを書く
- ビジネスロジックはできるだけモデルに入れてコントローラには書かない
- よく使う処理はものは適宜Ridge.pmを継承したクラス（今回はBookmark.pm）にフィルタ、プラグインに振り分ける

5. Perl WAF界隈の最新動向

PSGI/Plack

- モチベーション
    - WAFを作る度にサーバリクエスト、レスポンスを抽象化するコードを書いている
    - サーバーレスポンス、リクエストとPerlとの共通インターフェイスが欲しい

PSGI

- Perl向けサーバレスポンス、リクエストの**仕様**
- RubyのRack, PythonのWSGIにインスパイアされて作られた
- リクエストは$envにハッシュリファレンスで渡ってくる

```
my $env = {
            'psgi.version' => [1, 0],
            'psgi.url_scheme' => 'http',
            'psgi.input' => 'hoge',
            'psgi.errors' => '',
            'psgi.multithread' => 0,
            'psgi.multiprocess' => 0,
            'REQUEST_METHOD' => 'GET',
            'SCRIPT_NAME' => '/fuga',
            'PATH_INFO' => '/fuga',
            'QUERY_STRING' => 'id=1',
        };
```

* レスポンスは配列リファレンス

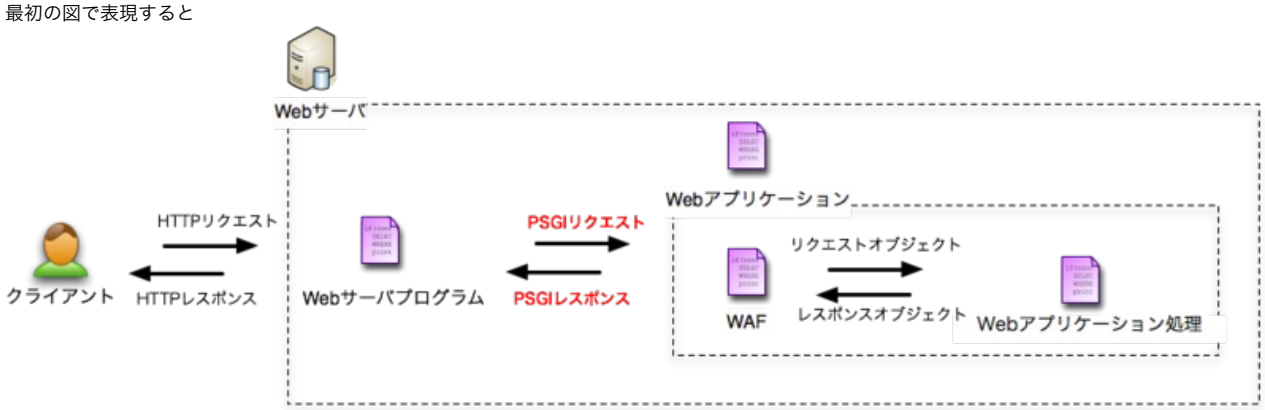
```
[
       200,
       [ 'Content-Type' => 'text/plain' ],
       [ "Hello stranger from $env->{REMOTE_ADDR}!"],
]
```

- PSGIでHello World

```
sub app {
    my $env = shift;
    return [
        '200',
        [ 'Content-Type' => 'text/plain' ],
        [ "Hello World" ], # or IO::Handle-like object
    ];
}
```

* PSGIを喋るサーバ * Starman, HTTP::Server::PSGI, Corona, Twiggy... (mod_psgiはまだ開発中?) * PSGIを喋るWAF * Catalyst, Jifty, Mojolicious, **Ridge**...

- 最初の図で表現すると



Plack

- PSGIの**リファレンス実装**
- Plack::Request, Plack::Response, plackup
- PSGIを喋らないサーバも抽象化
  - Plack::Handler::Apache2, Plack::Handler::FCGI...
- WAFを作るための部品

Plack Ridge

- RidgeにPlackを組み込んだ実装
- Ridge::Request/ResponseをPlack::Request/Responseのアダプタに

PSGI/Plackで嬉しいこと

- Apache以外の選択肢が使える
- より高速なサーバで運用も可能に
  - ラボサービスでテスト中
  - Hatean::Let, Hatena::Copie
  - サーバをApache2→Starmanに
  - Apache2と比べて1割ほど高速
  - メモリ消費量も減
- ホットデプロイ
  - 再起動せずにモジュール更新
- Plack::Middlerware::*が利用できる
  - 機能を追加できる (アプリケーションをラップする形)

ここまでのまとめ

- PSGIはサーバーレスポンス、リクエストとPerlとの共通インターフェイスの仕様
- Plackはその実装
- PSGI対応サーバ、WAFは増えている
  - Ridgeも対応
  - 選択肢が増えるのはサービス的にも○
- 自分でWAF作るときはPlack使うと楽だよ

課題内容

- Ridge を利用して、前回作った diary.pl を Web アプリケーションにして下さい

課題1 (4)

- ブラウザで読めるように (とりあえず読める=1)
  - テンプレートが使えている (1)
  - 設計がちゃんとしている (1)
  - ページャを実装 (1)
  - OFFSET / LIMIT と ?page=? というクエリパラメータを使います
  - **明日課題に繋がるので必須です**

ページングの典型的実装例

```
 my $page = $r->req->param('page') || 1;
my $limit = 3;
my $offset = ($page - 1) * $limit;

my $entries = moco("Entry")->search(
  where => { user_id => ... },
  offset => $offset,
  limit    => $limit,
  order   => 'created DESC',
)
```

課題2 (3)

- ブラウザで書けるように (1)
- ブラウザで更新できるように (1)
- ブラウザで削除できるように (1)

課題3 (3)

- 認証 (Hatena/Twitter OAuth)
- フィードを吐く (Atom, RSS)
- デザイン
- 管理画面
- さらに独自機能

注意

最初にやるべきこと：(ridge.pl plack ブランチの最新版を使ってください。以下の通りにすると、いい感じに Ridge アプリのスケルトンができるはず...)

```
 git submodule add https://github.com/hatena/Ridge Intern-Diary/modules/Ridge
perl Intern-Diary/modules/Ridge/bin/ridge.pl Intern::Diary
cd Intern-Diary
```

Can't locate Foo/Bar.pm in @INC というのがでたら、cpanm Foo::Bar します。例: Can't locate IO/Prompt.pm なら cpanm IO::Prompt

うまくいっていれば Intern-Diary/lib/Intern/Diary.pm や Intern-Diary/script/server.pl ができているはず。

提出前に、昨日と同様にmysqldumpの結果も付けておいてください！

参考：ユーザ認証層

```
 package Plack::Middleware::HatenaOAuth;
use strict;
use warnings;

our $VERSION   = '0.01';

use parent 'Plack::Middleware';
use Plack::Util::Accessor qw( consumer_key consumer_secret consumer login_path
 );
use Plack::Request;
use Plack::Session;

use OAuth::Lite::Consumer;
use JSON::XS;
```

```
sub prepare_app {
    my ($self) = @_;
    die 'require consumer_key and consumer_secret'
        unless $self->consumer_key and $self->consumer_secret;

    $self->consumer(OAuth::Lite::Consumer->new(
        consumer_key       => $self->consumer_key,
        consumer_secret    => $self->consumer_secret,
        site               => q{https://www.hatena.com},
        request_token_path => q{/oauth/initiate},
        access_token_path  => q{/oauth/token},
        authorize_path     => q{https://www.hatena.ne.jp/oauth/authorize},
        ($self->{ua} ? (ua => $self->{ua}) : ()),
    ));
}

sub call {
    my ($self, $env) = @_;
    my $session = Plack::Session->new($env);

    my $handlers = {
        $self->login_path => sub {
            my $req = Plack::Request->new($env);
            my $res = $req->new_response(200);
            my $consumer = $self->consumer;
            my $verifier = $req->param('oauth_verifier');

            if ( $verifier ) {
                my $access_token = $consumer->get_access_token(
                    token    => $session->get('hatenaoauth_request_token'),
                    verifier => $verifier,
                ) or die $consumer->errstr;
                $session->remove('hatenaoauth_request_token');

                {
                    my $res = $consumer->request(
                        method => 'POST',
                        url    => qq{http://n.hatena.com/applications/my.json}
,
                        token  => $access_token,
                    );
                    $res->is_success or die;
                    $session->set('hatenaoauth_user_info', decode_json($res->d
ecoded_content || $res->content));
                }
                $res->redirect( $session->get('hatenaoauth_location') || '/' )
;
                $session->remove('hatenaoauth_location');
            } else {
                my $request_token = $self->consumer->get_request_token(
                    callback_url => [ split /\?/, $req->uri, 2]->[0],
                    scope        => 'read_public',
                ) or die $consumer->errstr;

                $session->set(hatenaoauth_request_token => $request_token);
                $session->set(hatenaoauth_location => $req->param('location'))
;
                $res->redirect($consumer->url_to_authorize(token => $request_t
oken));
```

```
            }
            return $res->finalize;
        },
    };

    $env->{'hatena.user'} = ($session->get('hatenaoauth_user_info') || {})->{u
rl_name};
    return ($handlers->{$env->{PATH_INFO}} || $self->app)->($env);
}

1;

__END__

=head1 SYNOPSIS

  use Plack::Builder;

  my $app = sub {
      my $env = shift;
      my $session = $env->{'psgix.session'};
      return [
          200,
          [ 'Content-Type' => 'text/html' ],
          [
              "<html><head><title>Hello</title><body>",
              $env->{'hatena.user'}
                  ? ('Hello, id:' , $env->{'hatena.user'}, ' !')
                  : "<a href='/login?location=/'>Login</a>"
          ],
      ];
  };

  builder {
      enable 'Session';
      enable 'Plack::Middleware::HatenaOAuth',
           consumer_key       => 'vUarxVrr0NHiTg==',
           consumer_secret    => 'RqbbFaPN2ubYqL/+0F5gKUe7dHc=',
           login_path         => '/login',
           # ua                 => LWP::UserAgent->new(...);
      $app;
  };

=cut
```

参考資料

- https://github.com/hatena/Ridge
- perldoc -m Ridge
- http://template-toolkit.org/docs/manual/VMethods.html

Ridgeの処理の流れ

1. 初期化処理
    - Ridge::Request
    - Ridge::Response
    - Ridge::View
    - etc.
2. URLから、エンジン、アクション、ビューを決定
3. プラグイン初期化
4. before_dispatchトリガ実行
5. before_filter実行
6. アクションを実行
7. コンテントを生成

8. after_filter実行
9. after_dispatchトリガ実行
10. ヘッダ生成
11. ビューに従って結果を表示