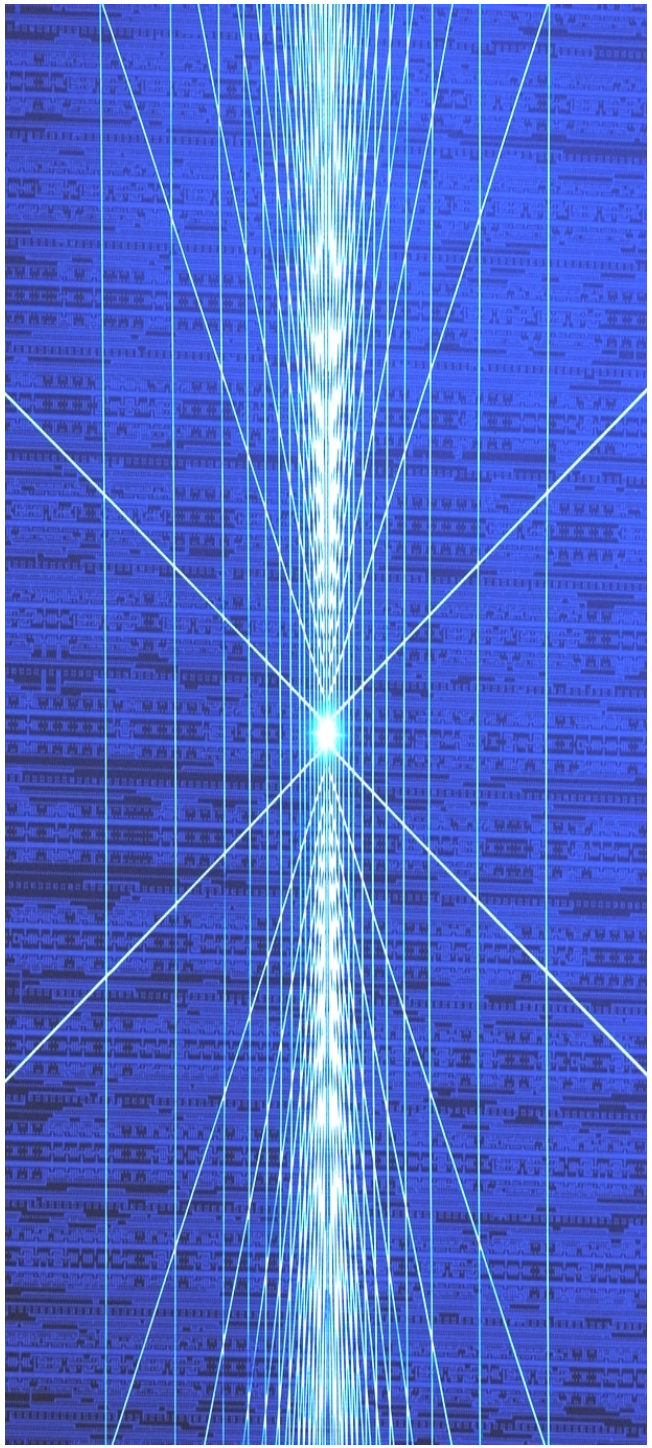


TOSHIBA
Leading Innovation

Ruby on Railsにみる RESTfulアプリケーションの方向性

2006/11/24 第九回XML開発者の日

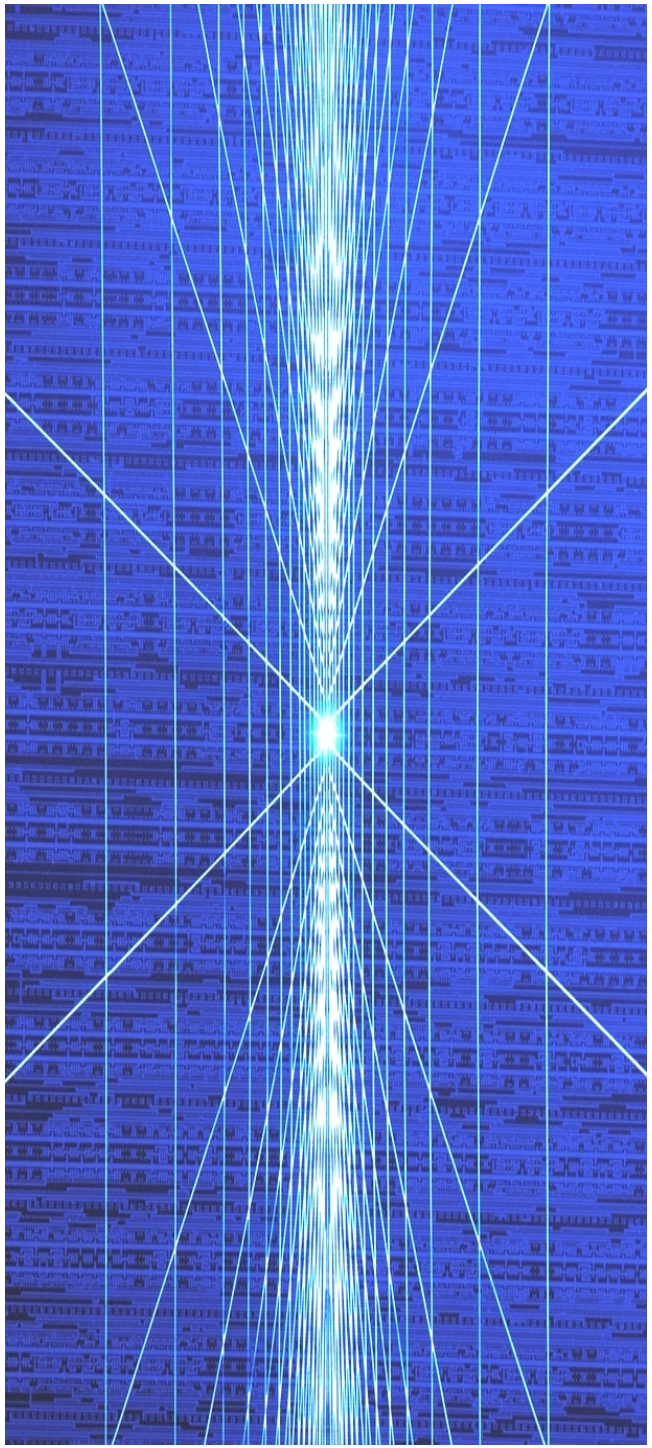
Collaborative Innovation

東芝ソリューション株式会社
TOSHIBA SOLUTIONS CORPORATION

IT技術研究所　　川村 徹

目次

- RESTful Webアプリケーションのメリット
  - WebアプリがRESTだと何がうれしいのか?
- RESTfulフレームワークとしてのRails
  - 次期バージョン1.2へ向けてREST化の方向性
- REST on Rails 設計例
  - Railsを例に、その方向性に沿ったもの
- 終わりに



RESTful Webアプリケーションのメリット

REST (REpresentational State Transfer) とは

- Webのあるべきアーキテクチャスタイル
- ステートレス
  - リクエストには処理に必要なすべての情報を含む
  - サーバ資源をすぐに解放できスケーラビリティ向上
- リソース指向
  - Web上のあらゆるものはリソースであり、識別子（URI）をもつ
- 統一インターフェース
  - URIとそれに適用するCRUD操作（POST, GET, PUT, DELETE）
  - RPCとは対極に位置する

RESTといえばWebサービス?

- Amazon ECS、はてなウェブサービス、…
- Atom Publishing Protocol
- Webアプリケーションだって重要（むしろ本流）
  - ブラウザとハイパーリンク(HTTP GET)の活用
- 今まではRESTと程遠いアプリケーションが多すぎた
  - /wiki.cgi?mode=delete&name=pagename

RESTのメリット

- ステートレスにすることでスケーラビリティの向上
  - RESTの一番のメリットとされている部分
  - クッキーでのセッション管理はNG

しかし…

- Personalized Webの時代
  - Railsや最近のWebアプリケーションはセッションを使うことがほとんど
  - クライアント（ユーザ）によって異なるレスポンスを返すためキャッシュが効かない

WebアプリケーションにRESTを適用するメリットは失われたのか?

いや、そんなことはありません!

利用者のメリット

- アプリケーション中のデータ（リソース）がURIで示せる
  →アドレス欄に入力すれば参照できる

  “エンドユーザーが、上司への報告書の中でそのシステムが提供する重要な情報（例えば今月の売り上げ一覧）を参照したい場合、彼はどうするだろうか。「ホームページを開いて、～～というリンクをクリックして、3つ目のテキスト・ボックスに『10月』と入力して一覧ボタンを押してください。」などとメールに書くのだろうか”

  出典：吉松史彰氏『Webの「正しい」アーキテクチャ』

設計面のメリット

- 統一インターフェース(Uniform Interface)
  - シンプルで一貫性のある設計
  - リソースへの操作はCRUDの4種類という**制約**
    →各リソースへの責任が適切に配分された設計が可能

Constraints are liberating
（制約が自由をもたらす）
by David Heinemeier Hansson

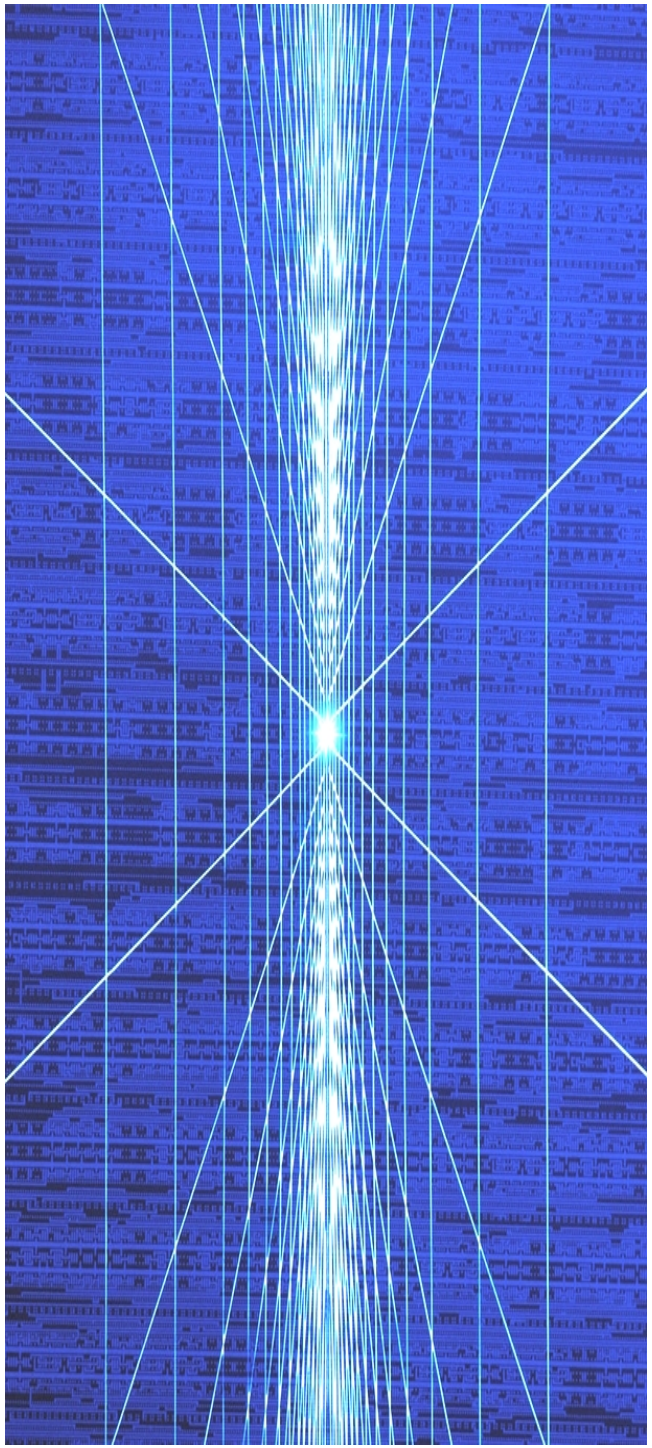
さらに

- 同じインターフェースでWebサービスになる
  - 入力：HTMLフォーム、出力：HTML

代わりにXMLやJSONを使えるようにする

↓

REST Webサービス化!



RESTfulフレームワークとしてのRails

Ruby on Railsとは

- David Heinemeier Hansson（通称DHH）氏作のWebアプリケーションフレームワーク
- これまでにもREST的なアプリケーションを開発するには適したフレームワークとされてきた
  - 第八回XML開発者の日 舘野氏の発表「RESTful Wikiの実装」

Cool URIを可能にするルーティングのしくみ

デフォルトでアプリケーションのURIは

http://www.example.com/controller/action/id

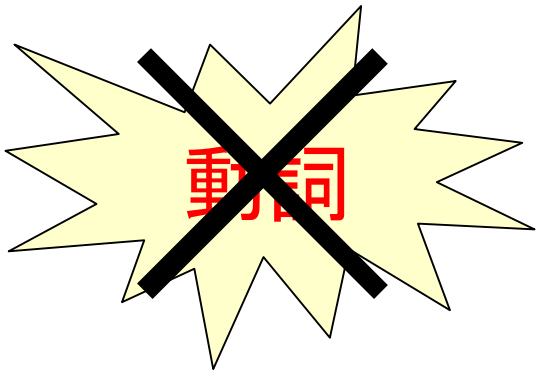
/shop/buy/1

```
class ShopController < ApplicationCon...
  def buy
    id = params[:id] # => 1
    ...
  end
```

- クエリパラメータ（?mode=buy&...）がないのがCool（?）

URIはリソース

- なんかおかしくない?
  /shop/buy/1
  /products/show/xxx
  /user/logout



- REST的にはURI＝リソース＝名詞
- 『現在のRailsについて、僕はみんなを間違った方向に導こうとしてた事に気が付いた』
  by DHH

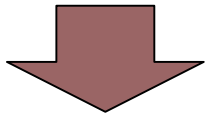
出典:「RoR Wiki 翻訳 Wiki - RubyKaigi2006」

Collaborative Innovation

CRUDへの変革

- 基本的なアクションを4つ（CRUD: create, read, update, delete）に制限

create	show	update	destroy



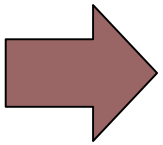
HTTPメソッドに対応!

POST	GET	PUT	DELETE

URIからアクション（動詞）を排除できる!

URIがリソースに

POST /products/create
GET /products/show/1
POST /products/update/1
POST /products/destroy/1



POST /products
GET /products/1
PUT /products/1
DELETE /products/1

URI＝リソース　が実現!

– ただし、ブラウザからPUT, DELETEメソッドは送れないため、POSTメソッドにパラメータ“_method=put”などをつけて代用する

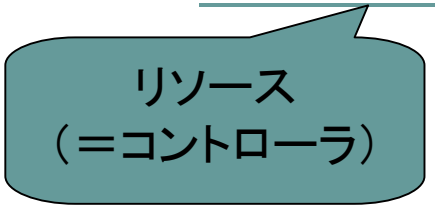
新しいルーティング機能

- コントローラをリソースとして指定

```
map.resources :products
```

アプリケーションのURIは

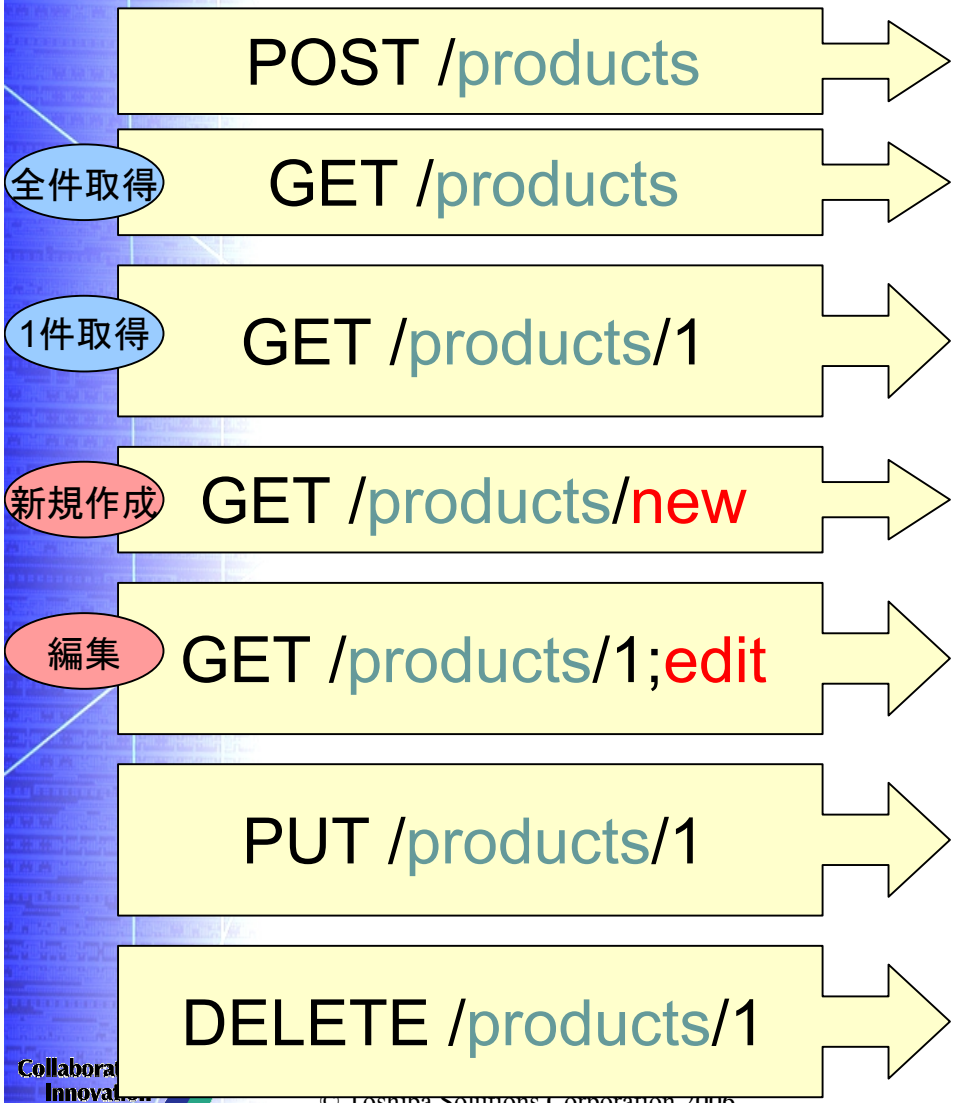
http://www.example.com/resources/id



GET /products/1

```
class ProductsController < Appl...
  def show
    id = params[:id] # => 1
    ...
  end
```

新しいルーティング機能




```
class ProductsController < Appl...
  def create
    ...
  end
  def index
    ...
  end
  def show
    id = params[:id] # => 1
    ...
  end
  def new
    ...
  end
  def edit
    id = params[:id] # => 1
    ...
  end
  def update
    id = params[:id] # => 1
    ...
  end
  def destroy
    id = params[:id] # => 1
    ...
  end
```

特徴的なURI

- ID1番の商品(products)の編集画面を表示するURIは

```
GET /products/1;edit
```

「edit」という一面から表示

- “;”（セミコロン）の後に、「どの一面から」処理するかを指示する
- あくまで例外的
- 表示形式によって、またスクリプトなどを用いれば使わないようにすることも可能

コンテントネゴシエーションとWebサービス化

- MIMEタイプによるコンテントネゴシエーション

```
class ProductsController < Applicat...
  def show
    id = params[:id] # => 1
    @product = Product.find(id)
    respond_to do |format|
      format.html
      format.xml {
        render :xml => @product.to_xml
      }
    end
    ...
  end
```

GET /products/1

```
<!DOCTYPE html PUBLIC "-//W3C//...
<html>
  <head>
    <title>商品: Ruby入門</title>
    ...
```

GET /products/1
Accept: application/xml

GET /products/1.xml

```
<product>
  <title>Ruby入門</title>
  <price>2800</price>
  ...
</product>
```

コンテントネゴシエーションとWebサービス化

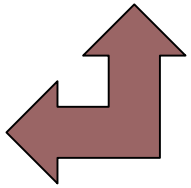
- フォーム入力とXML入力を同一視

```
POST /products
Content-Type: application/x-www-form-urlencoded

product[title]=Ruby入門&product[price]=2800&...
```

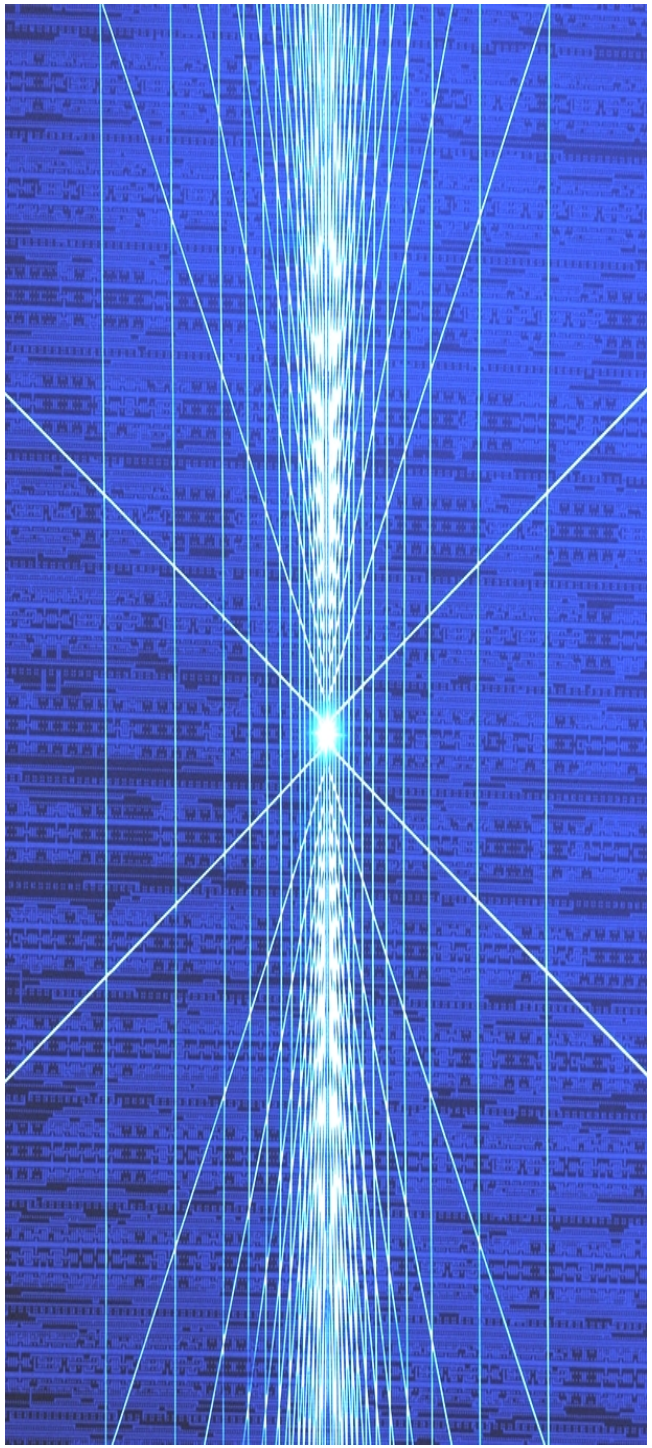
```
POST /products
Content-Type: application/xml

<product>
  <title>Ruby入門</title>
  <price>2800</price>
  ...
</product>
```



どちらも同一の入力として処理が可能

Ruby on Railsは、次期バージョン1.2に向けてRESTfulアプリケーションフレームワークとしての機能がますます充実していく。



REST on Rails 設計例

REST on Rails 設計例

- リソース同士の関連(1:多)
- リソース同士の関連(多:多)
- 状態変化
- ユーザ認証
- ウィザード
  - パターン(1)
  - パターン(2)
  - パターン(3)

リソース同士の関連(1:多)

- 例:「ユーザ」が「アイテム」を持つ

```
<user>
  <id>1</id>
  <name>Kawamura</name>
  ...
</user>
```

```
<item>
  <id>99</id>
  <name>Book</name>
  <user_id/>
  ...
</item>
```

itemの中にuser_idを持っている

PUT /items/99
item[user_id]=1

書き換えたい部分のデータを送信すればよい

```
<item>
  <id>99</id>
  <name>Book</name>
  <user_id>1</user_id>
  ...
</item>
```

Collaborative Innovation

リソース同士の関連（多：多）

- 例：「ユーザ」が「グループ」に所属する

```
<user>
  <id>1</id>
  <name>Kawamura</name>
  ...
</user>
```

```
<group>
  <id>2</id>
  <name>RESTafarian</name>
  ...
</group>
```

- パターン「関連のリソース」

```
POST /memberships
membership[user_id]=1&membership[group_id]=2
```

```
<membership>
  <id>3</id>
  <user_id>1</user_id>
  <group_id>2</group_id>
  <join_at>2006-10-12T18:42:03+09:00</join_at>
</membership>
```

リソース同士の関連(多:多)

- 例:「ユーザ」が「グループ」から離脱する

```
<user>
  <id>1</id>
  <name>Kawamura</name>
  ...
</user>
```

```
<group>
  <id>2</id>
  <name>RESTafarian</name>
  ...
</group>
```

```
<membership>
  <id>3</id>
  <user_id>1</user_id>
  <group_id>2</group_id>
  <join_at>2006-10-12T18:42:03+09:00</join_at>
</membership>
```

DELETE /memberships/3

関連のリソースを削除

関連を表現するURI

- Railsで関連を表現するURIを提供する Nested Resources

```
map.resources :users do |user|
  user.resources :items
end
```

GET /users/1/items

GET /users/1/items/99

```
class ItemsController < Applic...
  def index
    uid = params[:user_id] # => 1
    ...
  end
  def show
    uid = params[:user_id] # => 1
    id  = params[:id] # => 99
    ...
  end
```

状態変化

- 例:「記事」の状態
  - 「下書き(未公開)」
  - 「公開」
  - 「(論理)削除」

```
<article>
  <id>1</id>
  <title>About REST</title>
  <status>draft</status>
  ...
</article>
```

- 記事を公開する

```
PUT /articles/1
article[status]=public
```

```
<article>
  <id>1</id>
  <title>About REST</title>
  <status>public</status>
  ...
</article>
```

Collaborative Innovation

状態変化

- もう1つのパターン「状態のリソース」“publication”

```
<article>
  <id>1</id>
  <title>About REST</title>
  ...
</article>
```

- 記事をKawamuraによって公開する

```
POST /publications
publication[article_id]=1&publication[by]=Kawamura
```

  - 状態変化に付随する情報も保存できる
  - 状態変化の履歴が残せる

```
<publication>
  <id>3</id>
  <article_id>1</article_id>
  <by>Kawamura</by>
  <at>2006-10-12T18:42:03+09:00</at>
</publication>
```

ユーザ認証

- REST本来はリクエストごとにHTTP Basic/Digestなどで認証するのが正しい
- Personalized Web
  - 常に認証が必要とされる
  - HTTP Basic/Digestでは明示的にログアウトできないのが不便(本当はブラウザで機能を用意すべき?)

ユーザ認証

- 「セッション」もリソースとみる→CRUD対象
  - ログイン：認証セッションのcreate

```
POST /auth_session
username=kawamura&password=XXX
```

  - ログアウト：認証セッションのdestroy

```
DELETE /auth_session
```

セッションに存在する単数リソース

- Railsでは複数形のリソースにしか対応していない

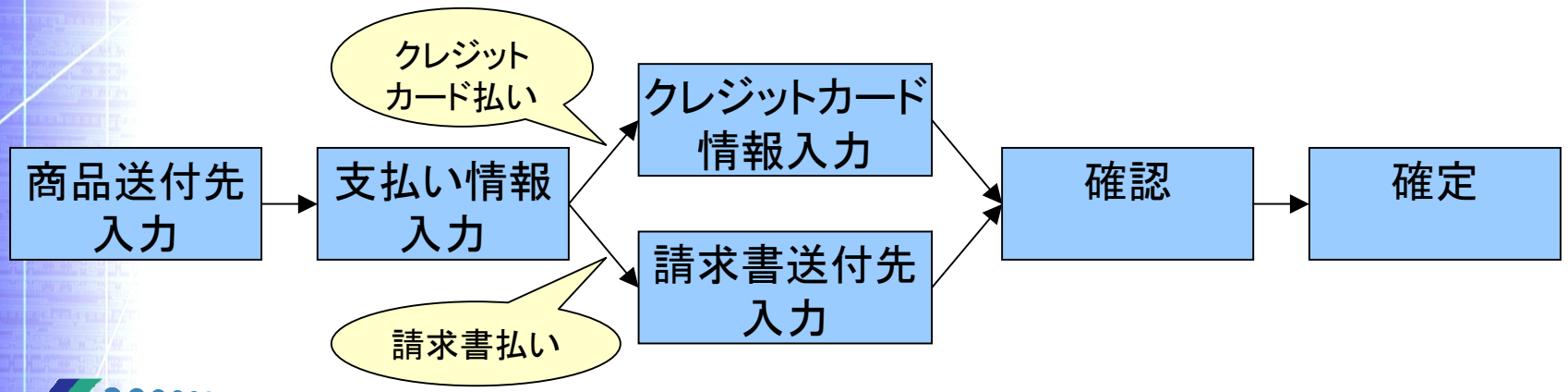
```
map.resources :products
```

- セッションに存在するリソースを表現するために単数形リソースが必要
- 単数リソース対応プラグイン“map_singular_resource”を作成

```
map.singular_resource :auth_session
```

ウィザード

- クライアントにデータを連続して入力させる
- 最後にすべてのデータが確定する
- 条件分岐もあり
- 例：注文情報入力

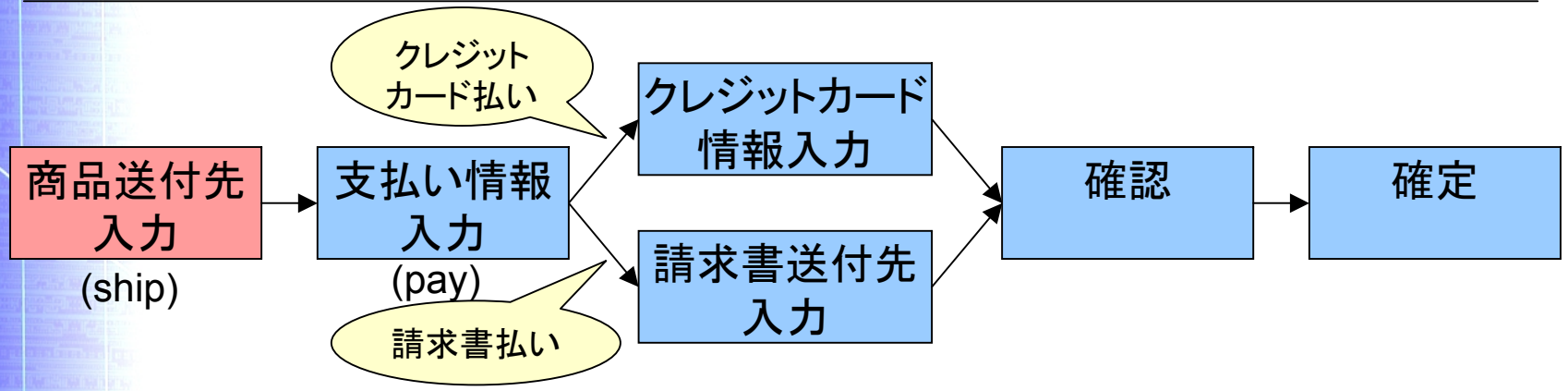


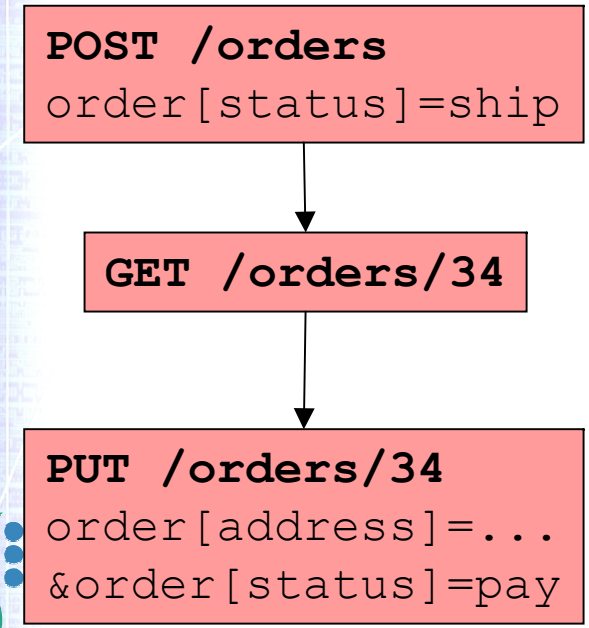
Collaborative Innovation

ウィザード(1)

- 初めに注文情報リソースをcreate
- その後画面ごとにPUTでデータを追加していく
- パラメータ“status”で状態遷移

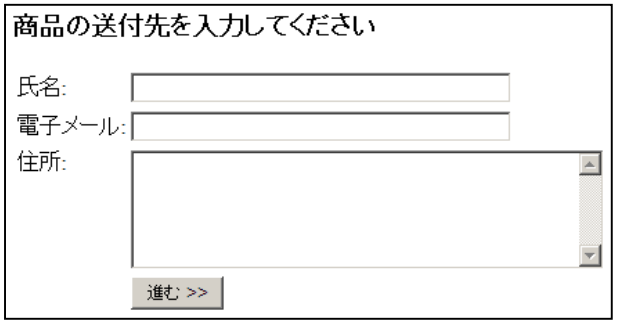
ウィザード(1)





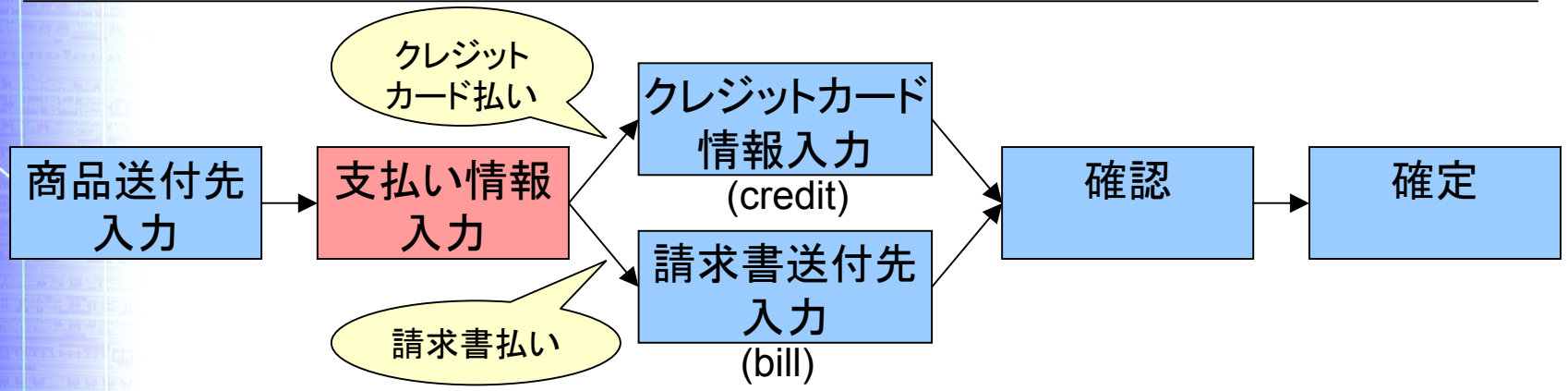

```
201 Created
Location: /orders/34
```

リソースをまず作ってしまう




```
303 See Other
Location: /orders/34
```

ウィザード(1)




```
GET /orders/34
```

支払い方法をご指定ください

支払い方法: ○ クレジットカード
○ 請求書

進む >>

<< 戻る

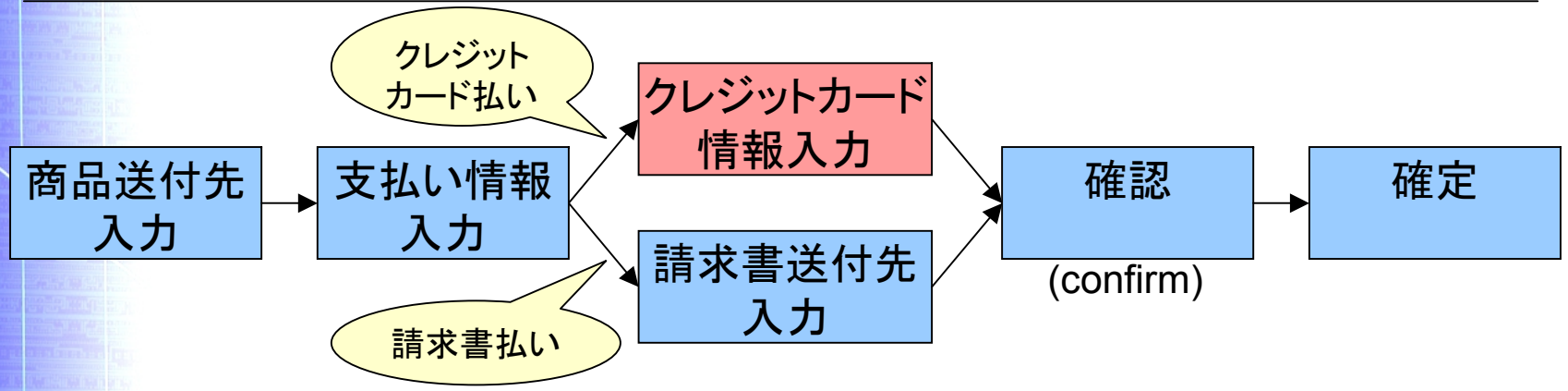
```
PUT /orders/34
order[pay_type]=...
&order[status]=credit_or_bill
```

```
303 See Other
Location: /orders/34
```

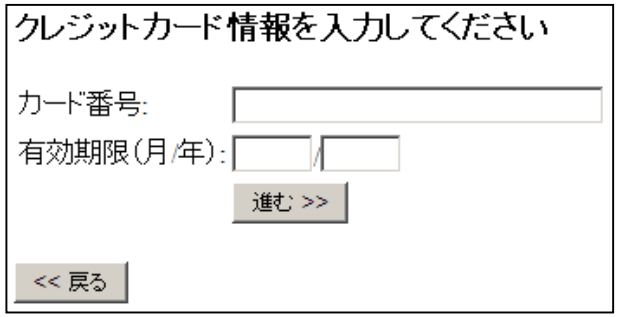
pay_typeに応じて
内部で書き換える

Collaborative Innovation

ウィザード(1)



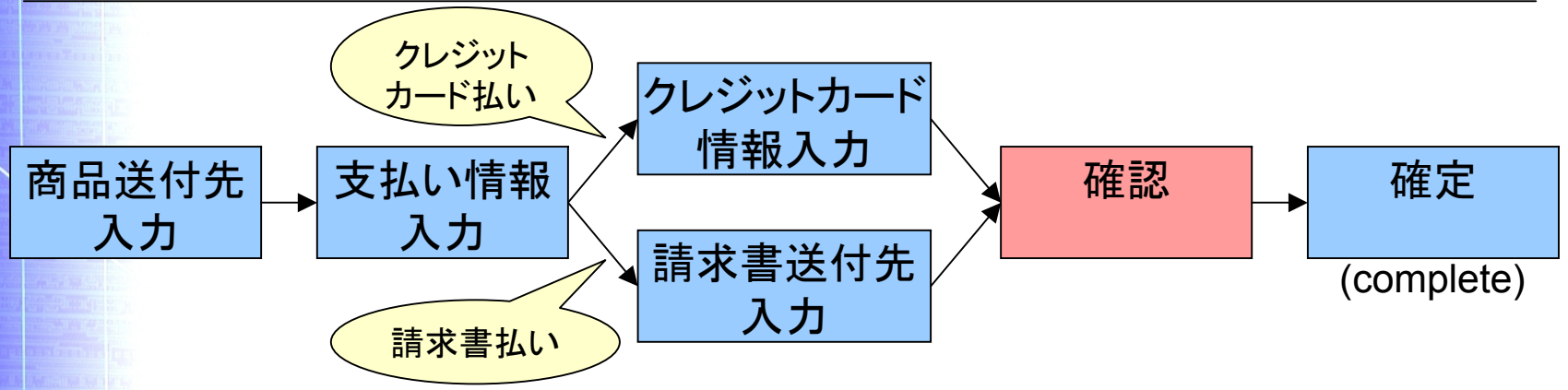

```
GET /orders/34
```




```
PUT /orders/34
order[credit_number]=...
&order[status]=confirm
```

```
303 See Other
Location: /orders/34
```

ウィザード(1)



GET /orders/34

PUT /orders/34
order[status]=complete

以下の情報でよろしいですか？

氏名:	川村 徹
電子メール:	tkawa@4bit.net
住所:	東京都府中市片町3-22
支払い方法:	クレジットカード
カード番号:	XXXXXXXXXXXXX012
有効期限(月/年):	11/12

確定

<< 戻る

「戻る」ボタンもそれぞれ
statusを書き換えるだけ

```
303 See Other
Location: /orders/34
```

Collaborative Innovation

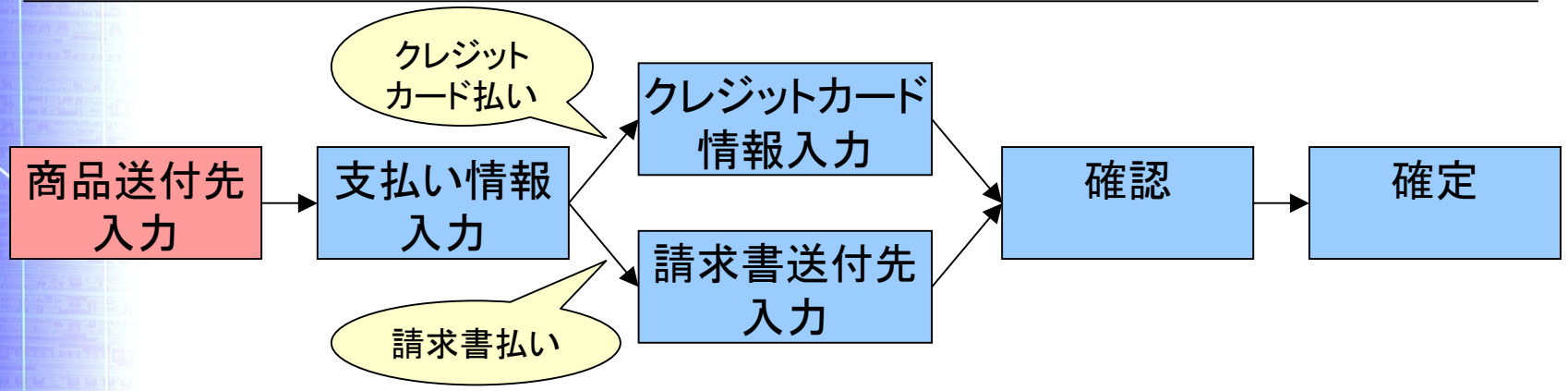
ウィザード(1)

- メリット
  - 入力途中の状態をすべて保存できる
- 問題点
  - 途中でブラウザを閉じられたら中途半端な注文情報が残ってしまう
  - 残った注文情報の管理をユーザ側に任せることになってしまう
  - ブラウザの「戻る」ボタンが使えない（本当にまったく使えないので、中途半端に使えるよりはいいかも）

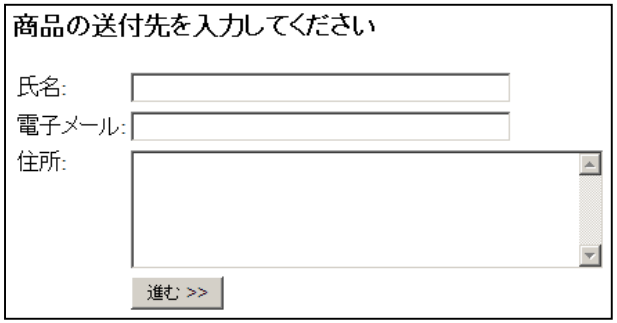
ウィザード(2)

- 途中でブラウザを閉じる問題に対処するには…
  - 途中の状態を保存しない
  - セッションを使う
- POST /orders を繰り返して画面遷移
- 途中の入力データはhiddenパラメータで保持する

ウィザード(2)




```
GET /orders/new
```

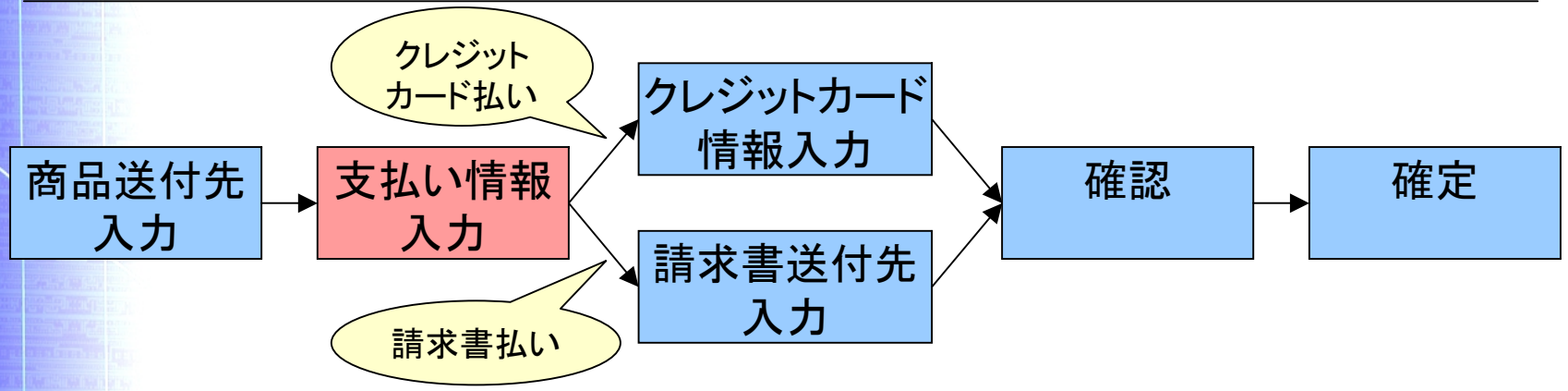



```
POST /orders
order[address]=...
&order[status]=pay
```

```
200 OK

...
```

ウィザード(2)




```
POST /orders
order[address]=...
&order[status]=pay
```

```
POST /orders
order[address]=...
&order[pay_type]=...
&order[status]=credit_or_bill
```

```
200 OK
```

支払い方法をご指定ください

支払い方法: ○ クレジットカード ○ 請求書

進む >>

<< 戻る

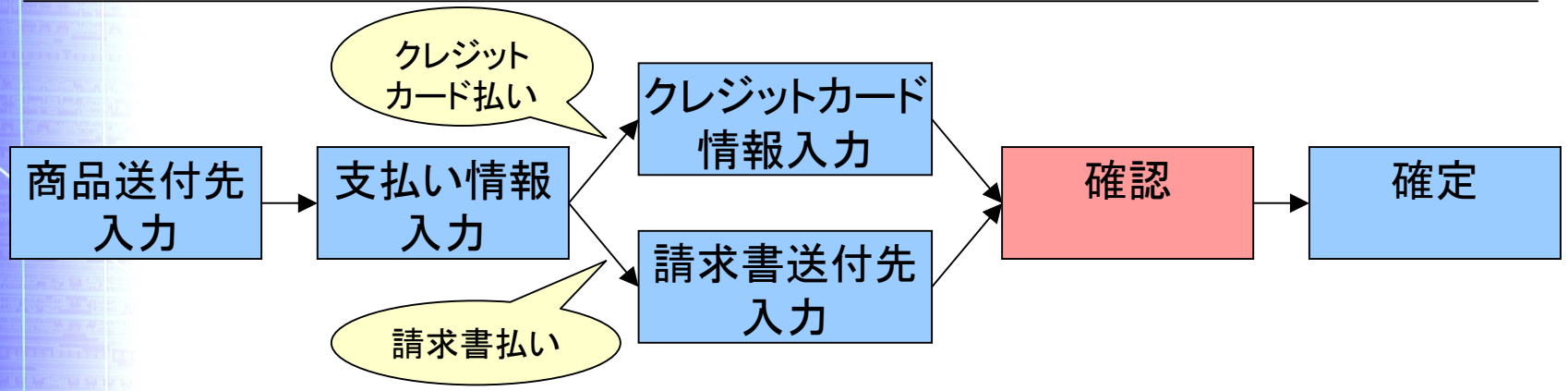
```
200 OK

...
```

<input type="hidden" name="order[address]" value="..."/>

pay_typeに応じて
内部で書き換える

ウィザード(2)




```
POST /orders
...
```

```
POST /orders
order[address]=...
&order[pay_type]=...
&order[credit_number]=...
&order[status]=complete
```

最後にすべての
パラメータを送る

```
200 OK
```

以下の情報でよろしいですか？

氏名:	川村 徹
電子メール:	tkawa@4bit.net
住所:	東京都府中市片町3-22
支払い方法:	クレジットカード
カード番号:	XXXXXXXXXXXX012
有効期限(月/年):	11/12

確定

<< 戻る

```
201 Created
Location: /orders/34
```

最後にリソース
が生成される

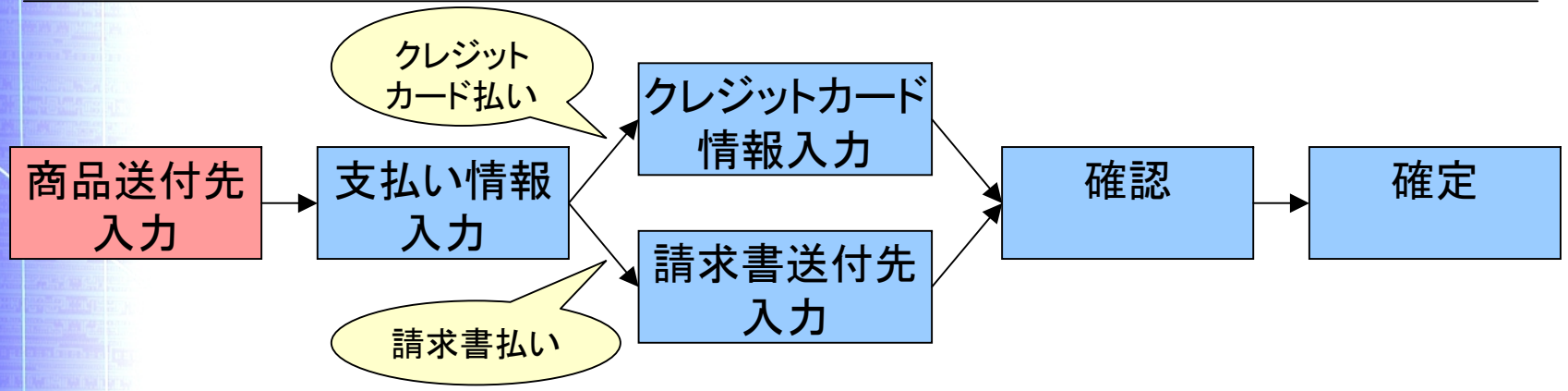
ウィザード(2)

- メリット
  - 途中でブラウザを閉じられても問題ない
  - ブラウザの「戻る」ボタンが使える
- 問題点
  - 「進む」「戻る」ボタンともに大量のhiddenパラメータを埋め込まなければならず煩雑

ウィザード(3)

- 途中でブラウザを閉じる問題に対処するには…
  - 途中の状態を保存しない
  - セッションを使う
- 初めにウィザードリソースorder_wizardをcreate（実体はセッションに保存する）、その後画面ごとにPUTでデータを追加していく
- 最後に注文情報リソースをcreateする

ウィザード(3)




```
POST /order_wizard
order[status]=ship
```

```
GET /order_wizard
```

```
PUT /order_wizard
order[address]=...
&order[status]=pay
```

```
201 Created
Location: /order_wizard
```

ウィザードリソース
作成

商品の送付先を入力してください

氏名:

電子メール:

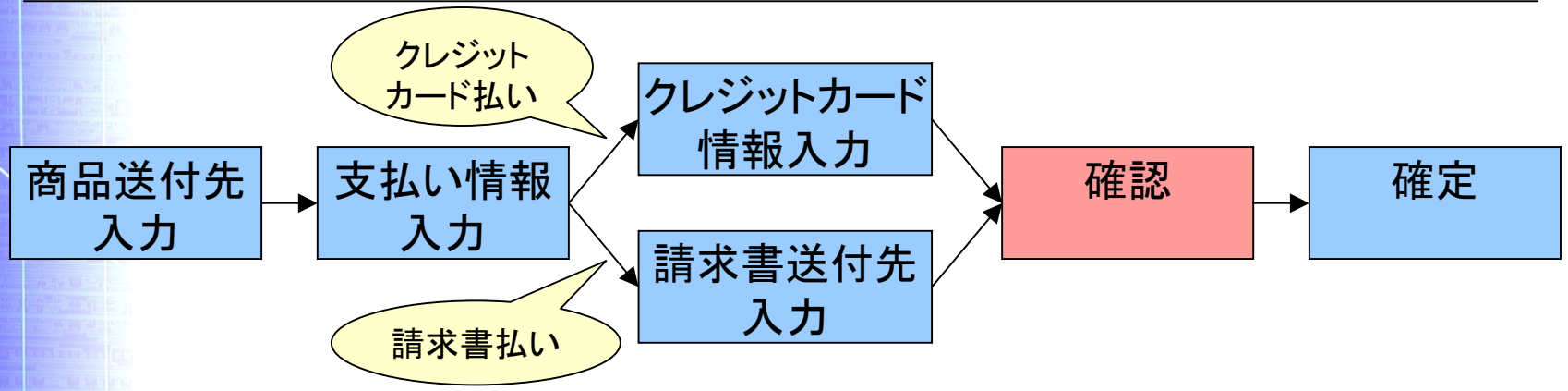
住所:

進む >>

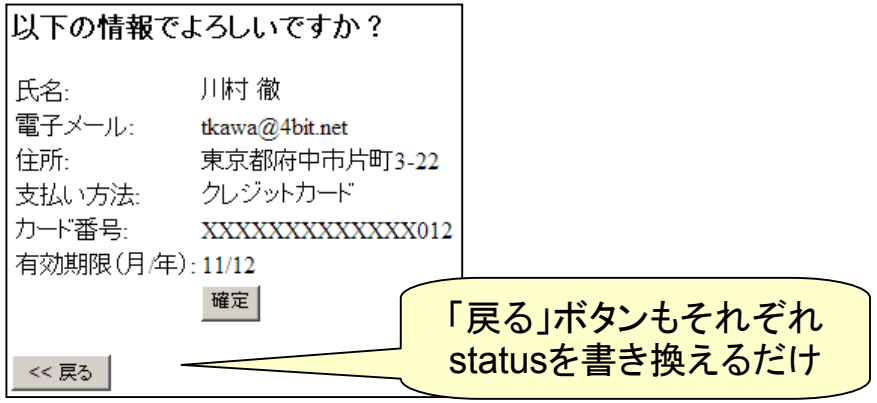
```
303 See Other
Location: /order_wizard
```

Collaborative Innovation

ウィザード(3)




```
GET /order_wizard
```




```
POST /orders
```

```
201 Created
Location: /orders/34
```

最後にリソース
が生成される

ウィザード(3)

- メリット
  - 入力途中の状態をすべて保存できる
  - 途中でブラウザを閉じられても問題ない
- 問題点
  - ブラウザの「戻る」ボタンが使えない

終わりに

- RESTをWebアプリ設計の技法として使おう
  - もし部分的にセッション（クッキー）を使っていても、RESTの恩恵は受けられるはず
- Ruby以外の言語にも、Railsの方向性・RESTを指向したアプリケーションフレームワークが出てほしい
  - 考え方はRailsだけに限らない

ご清聴ありがとうございました

参考文献等

- David Heinemeier Hansson
  “Discovering a world of Resources on Rails”
  - http://www.loudthinking.com/lt-files/worldofresources.pdf
- 「RoR Wiki 翻訳 Wiki - RubyKaigi2006」
  - http://techno.hippy.jp/rorwiki/?RubyKaigi2006
- 吉松 史彰『Webの「正しい」アーキテクチャ』
  - http://www.atmarkit.co.jp/fdotnet/opinion/yoshimatsu/onepoint05.html
- Rick Olson “Restful Authentication Plugin”
  - http://svn.techno-weenie.net/projects/plugins/restful_authentication/
- Paul Prescod
  “A Web-Centric Approach to State Transition”
  - http://www.prescod.net/rest/state_transition.html