SERVICE-ORIENTED ARCHITECTURE (SOA)

SIN5005 — TÓPICOS EM ENGENHARIA DE SOFTWARE

Daniel Cordeiro

11 de setembro de 2019

Escola de Artes, Ciências e Humanidades | EACH | USP

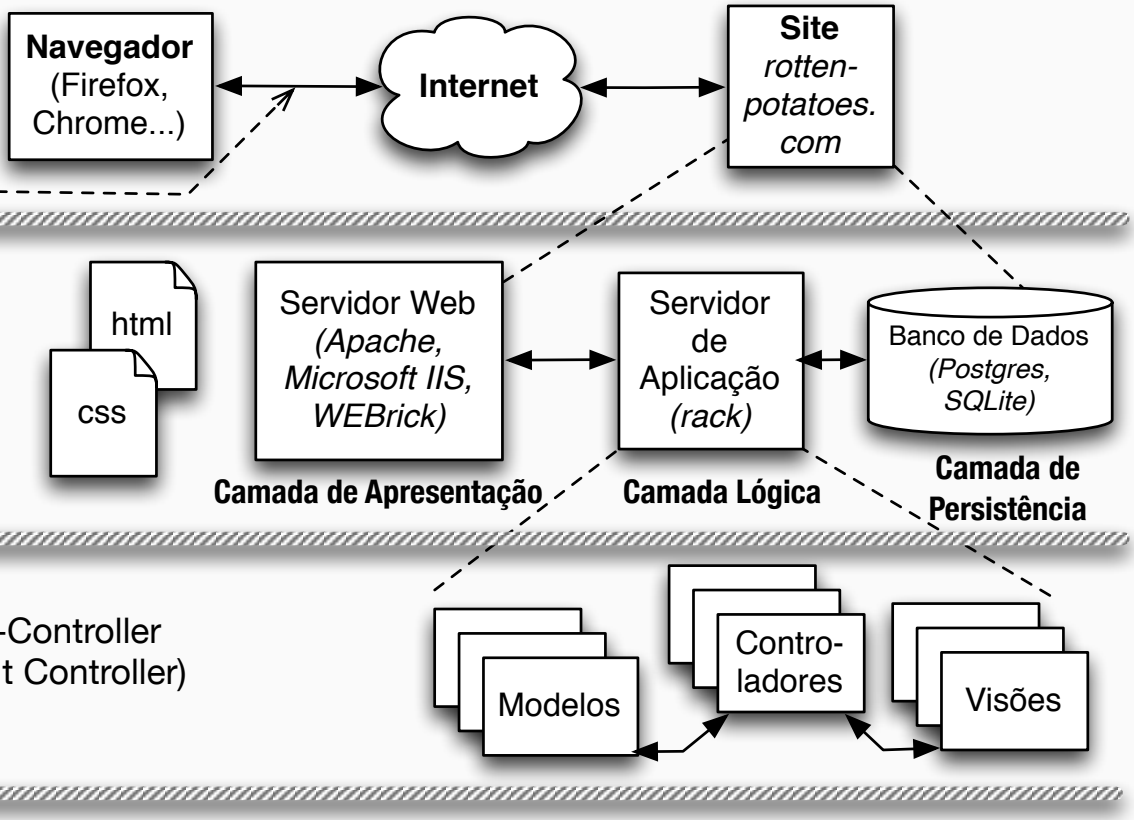
Arquitetura shared-nothing em 3 camadas

§2.1 100.000 pés
• Cliente-servidor (vs. P2P)

§2.2 50.000 pés
• HTTP e URIs

§2.3 10.000 pés
• XHTML e CSS

§2.4 5.000 pés
• Arquitetura de 3 camadas
• Escalabilidade horizontal



§2.5 1.000 pés—Model-View-Controller
(vs. Page Controller, Front Controller)

§2.6 500 pés: Modelos Active Record (vs. Data Mapper)
§2.7 500 pés: Controladores REST (*Representational State Transfer* para ações auto-contidas)
§2.8 500 pés: Template View (vs. Transform View)

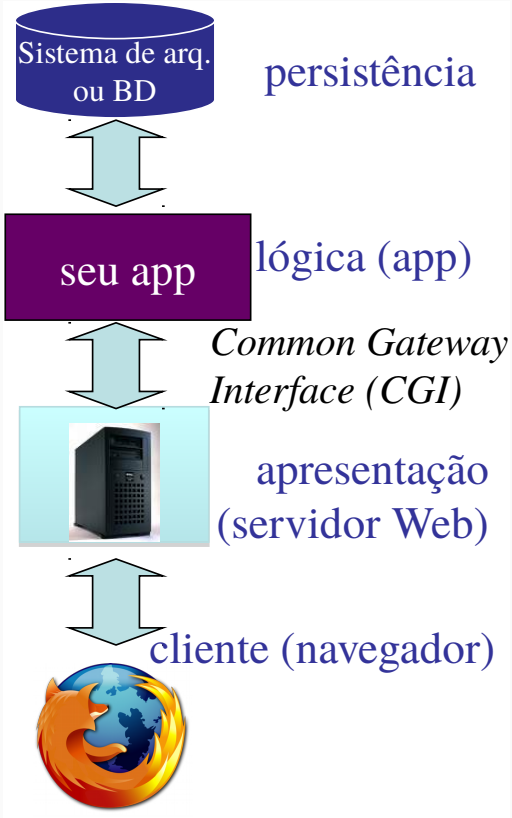
• **Active Record** • **REST** • **Template View**
• Data Mapper • Transform View

CRIAÇÃO DE CONTEÚDO DINÂMICO

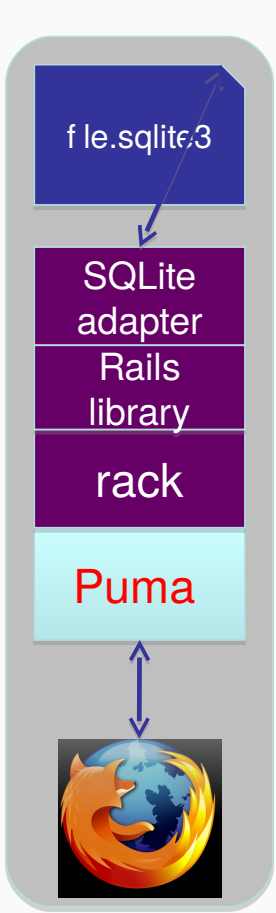
- Antigamente, a maior parte das páginas Web eram (coleções de) arquivos simples
- Mas os sites mais interessantes da Web 1.0/e-commerce executam um programa que cria cada “página”
- Originalmente: *templates* com código embutido (“*snippets*”)
- Eventualmente o código acabou movido para fora do servidor Web

SITES QUE NA VERDADE SÃO PROGRAMAS (SAAS)

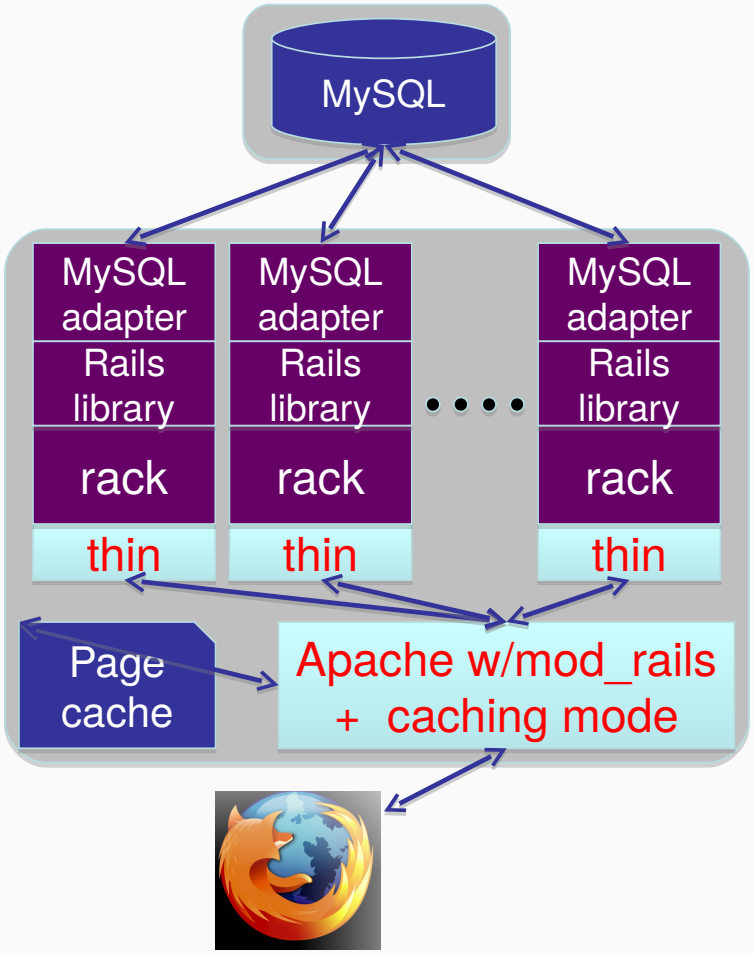
- Como você faz para:
  - "mapear" a URI para o programa & função corretos
  - passar argumentos?
  - invocar programas no servidor?
  - lidar com armazenamento persistente?
  - lidar com cookies?
  - lidar com erros?
  - empacotar a saída para o usuário?
- Usar arcabouços facilita essas tarefas mais comuns



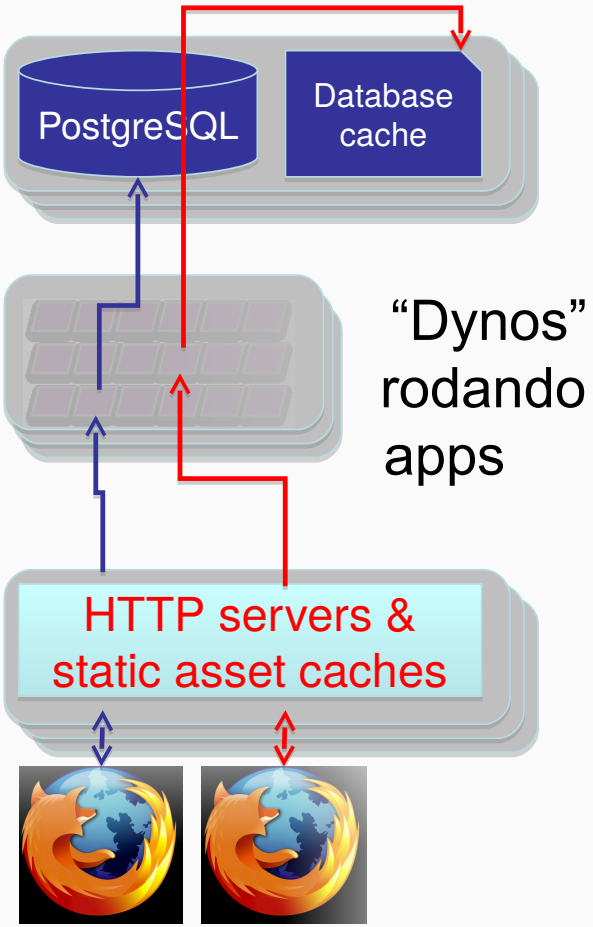
AMBIENTE DO DESENVOLVEDOR VS. IMPLANTAÇÃO DE MÉDIA ESCALA



Desenvolvedor

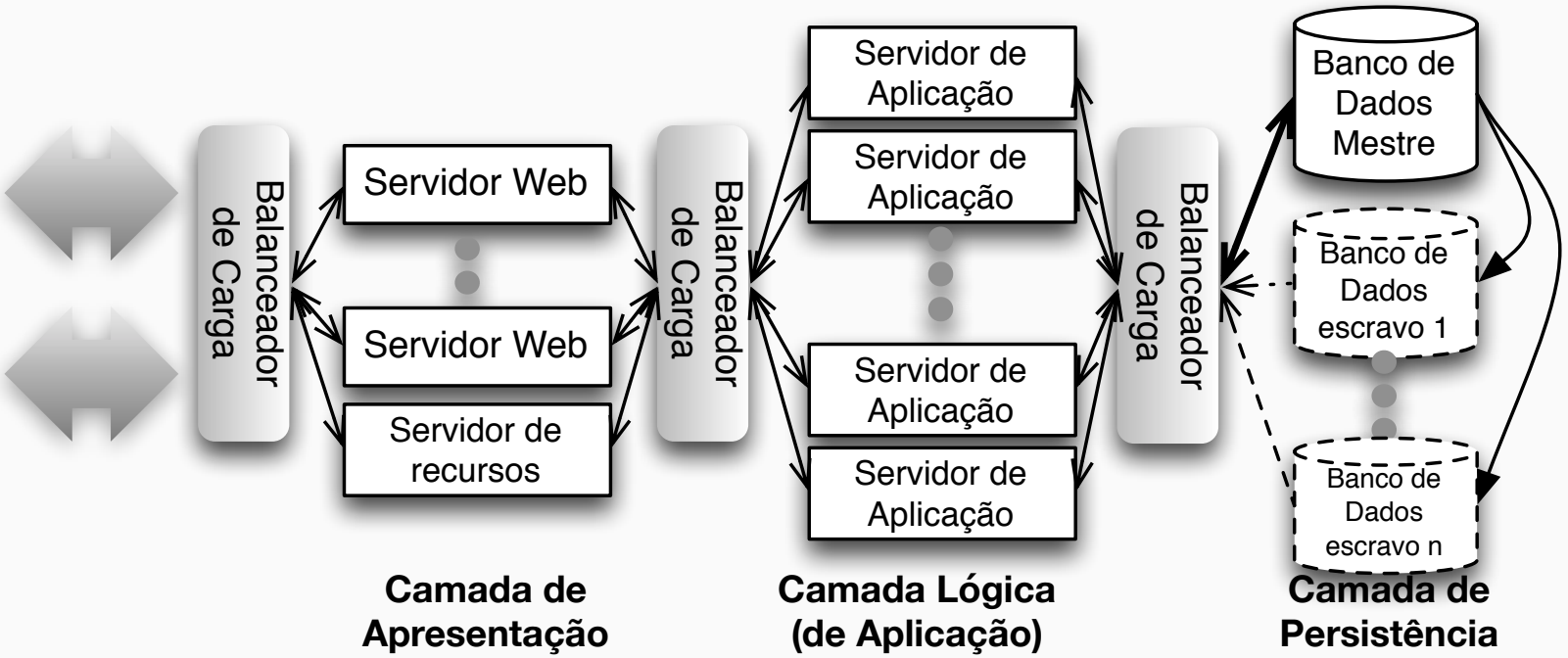


Implantação de média escala



Implantação cuidadosa em larga escala (ex: Heroku)

“SHARED NOTHING”



RESUMO: SAAS NA WEB 1.0 I

- Navegador requisita um recurso web (URI) usando HTTP
  - HTTP é um protocolo requisição–resposta simples que depende de TCP/IP
  - em SaaS, a maior parte das URIs disparam a execução de um programa
- HTML é usado para codificar o conteúdo, CSS para estilizá-lo
- Cookies permitem que o servidor acompanhe o rastro do usuário
  - o navegador automaticamente passa os cookies para o servidor em cada requisição
  - o servidor pode mudar o cookie em cada requisição
  - uso típico: cookie inclui uma forma de acessar a informação do lado do servidor
  - por isso muitos sites não funcionam quando os cookies estão totalmente desabilitados

RESUMO: SAAS NA WEB 1.0 II

- Arcabouços fazem com que essas abstrações sejam mais convenientes para o programador usar, sem que ele precise entrar em todos os detalhes
- ... e permitem mapear um app SaaS na arquitetura em 3 camadas “shared-nothing”

PERGUNTA

Quais afirmações são corretas sobre as duas requisições abaixo:

```
GET /foo/bar
POST /foo/bar
```

1. Eles são indistinguíveis para a app SaaS
2. Eles são distinguíveis e *devem* ter comportamentos diferentes
3. Eles são distinguíveis e *podem* ter comportamentos diferentes
4. Um dado app pode ser configurado para lidar com um ou com outro, mas não ambos

Model–View–Controller

ATÉ QUE PONTO...

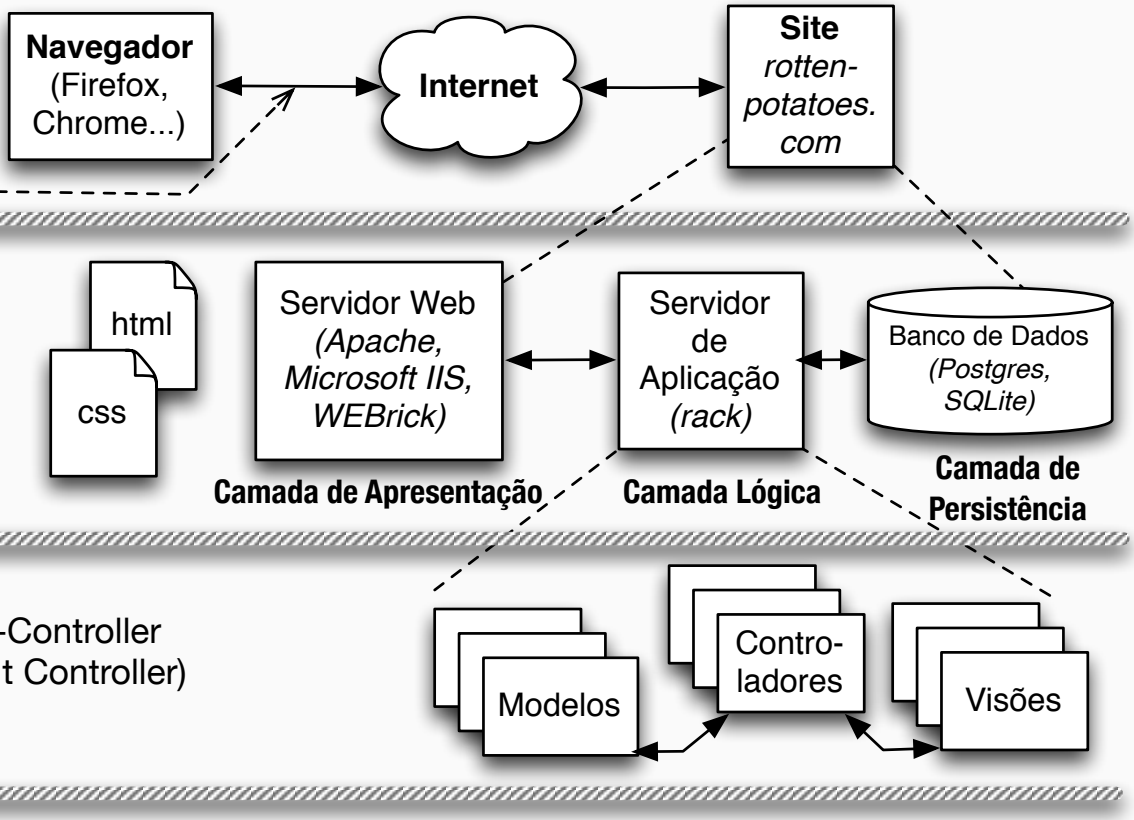
- há alguma estrutura comum em aplicações...
- ... interativas ...
- ... que pode simplificar o desenvolvimento de apps se nós a capturarmos em um arcabouço?

§2.1 100.000 pés
• Cliente-servidor (vs. P2P)

§2.2 50.000 pés
• HTTP e URIs

§2.3 10.000 pés
• XHTML e CSS

§2.4 5.000 pés
• Arquitetura de 3 camadas
• Escalabilidade horizontal



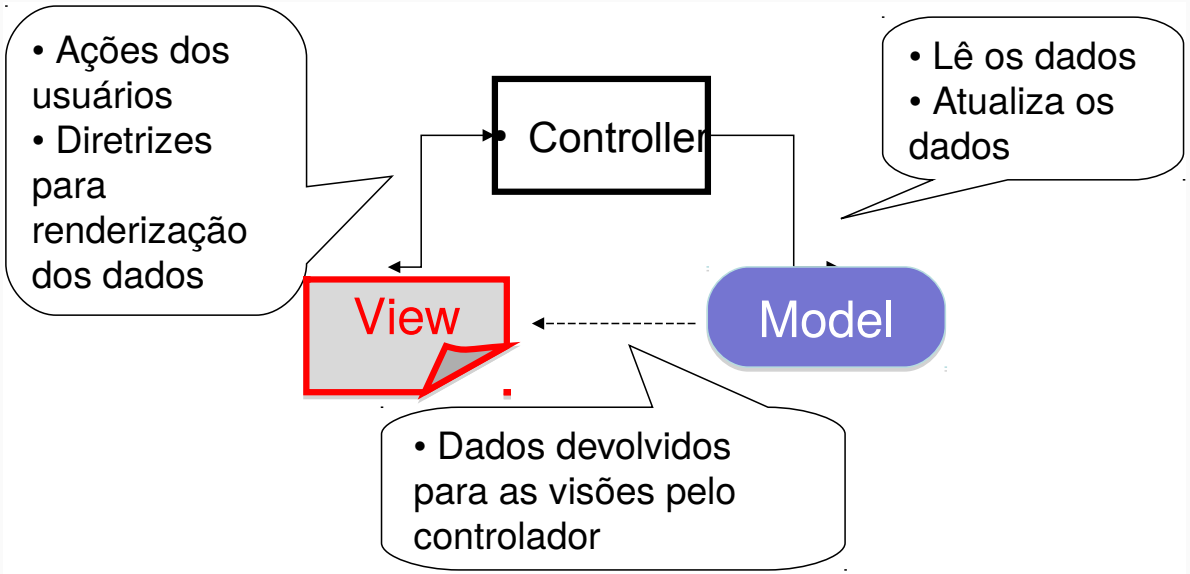
§2.5 1.000 pés—Model-View-Controller
(vs. Page Controller, Front Controller)

§2.6 500 pés: Modelos Active Record (vs. Data Mapper)
§2.7 500 pés: Controladores REST (*Representational State Transfer* para ações auto-contidas)
§2.8 500 pés: Template View (vs. Transform View)

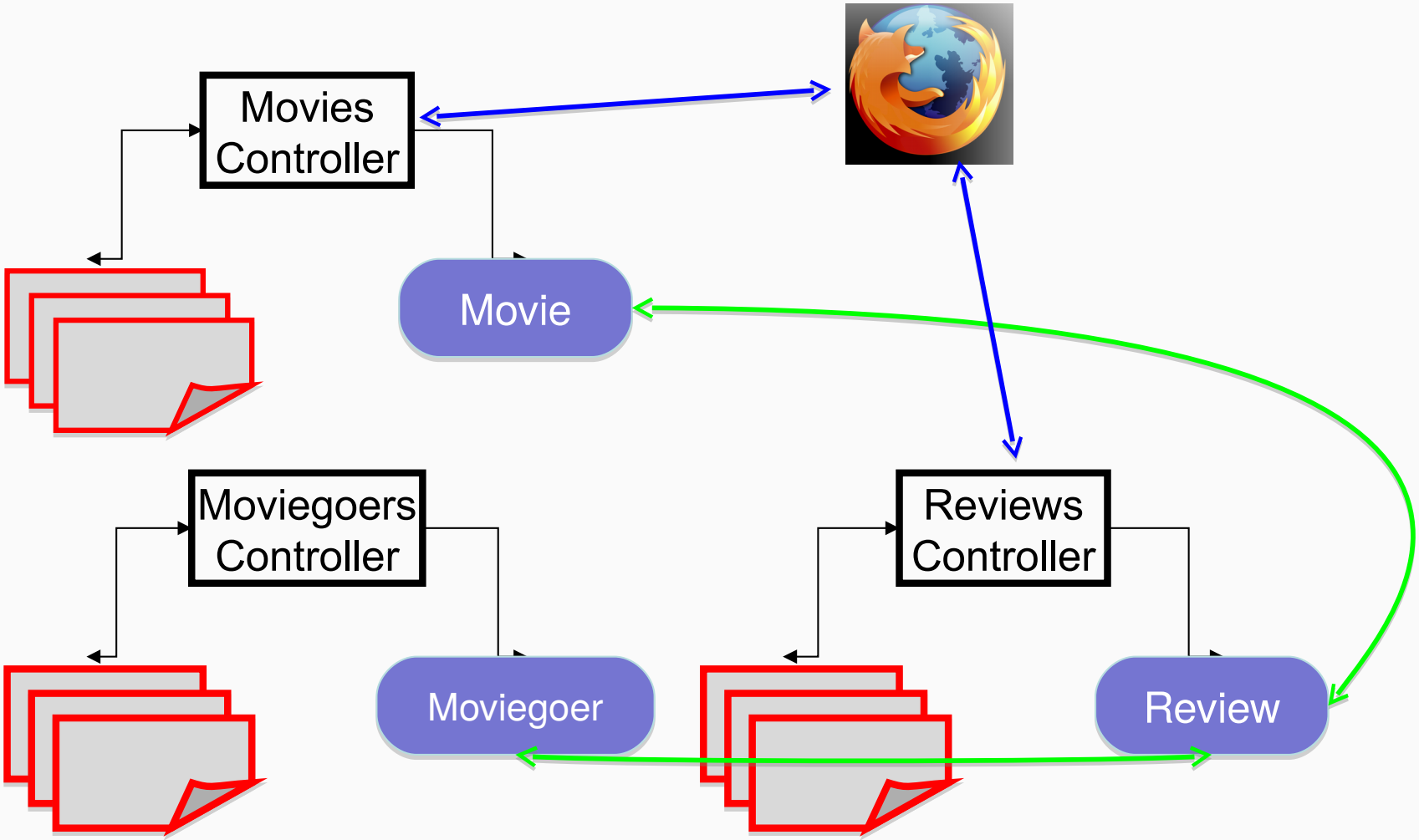
• **Active Record** • **REST** • **Template View**
• Data Mapper • Transform View

O padrão de projeto MVC

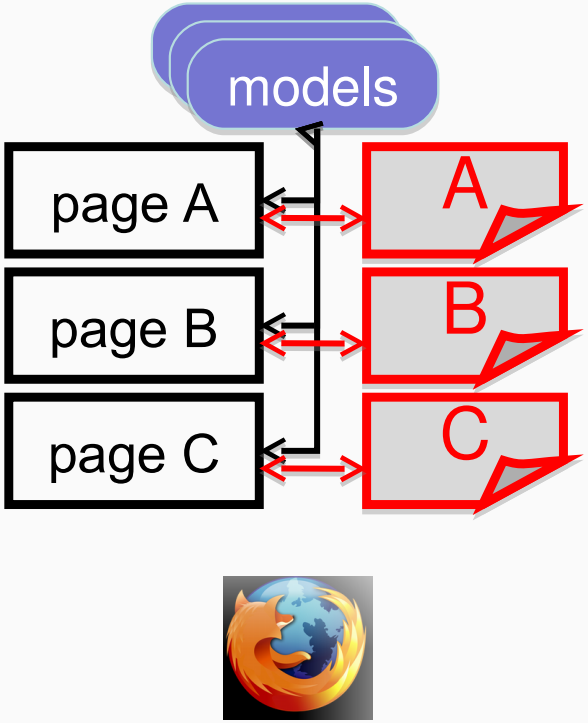
- Objetivo: separar os dados (*modelo*) da UI & apresentação (*visão*) com o uso de um *controlador*
  - intercede as ações dos usuários que pedem acesso aos dados
  - expõe os dados para a renderização (a ser realizada pela visão)
- Apps Web podem parecer "obviamente" MVC por definição, mas outras alternativas são possíveis...



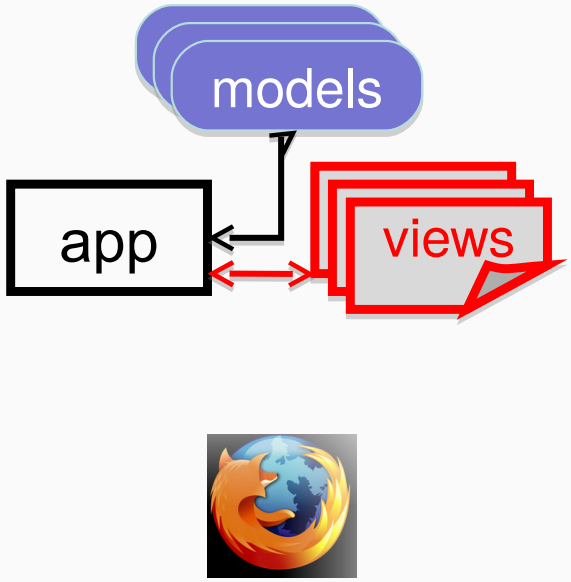
CADA ENTIDADE TEM UM MODELO, CONTROLE & CONJUNTO DE VISÕES



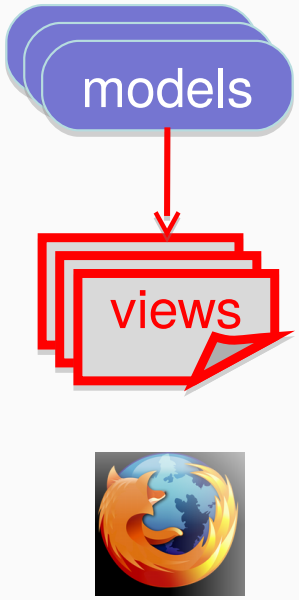
Page Controller (Ruby Sinatra)



Front Controller (J2EE servlet)



Template View (PHP)



Rails
Usa por padrão apps estruturados como MVC, mas outras arquiteturas podem ser melhores para certos apps.

PERGUNTA

Qual afirmação não é verdade sobre o padrão arquitetural Model–View–Controller:

- Em apps SaaS na Web, o conteúdo relacionado às ações do controlador e visão são transmitidos via HTTP
- Todos os apps MVC possuem uma parte “cliente” (ex: navegador) e uma parte “na nuvem” (ex: app Rails em uma plataforma de computação em nuvem)
- Model–View–Controller é só um dentre vários modos possíveis de estruturar uma app SaaS
- Apps Peer-to-Peer (o contrário de apps cliente–servidor) podem ser estruturados com Model–View–Controller

- Como armazenar e recuperar dados restruturados como registros?
- Qual a relação entre a forma que os dados estão armazenados com a forma como eles são manipulados em uma linguagem de programação?

Objetos in-memory vs. in-storage

- Como representar objetos que foram gravados em um dispositivo de armazenamento?
  - Exemplo: `Movie` com os atributos `name` e `rating`
- Operações básicas em um objeto: CRUD (`Create, Read, Update, Delete`)
- ActiveRecord: todo modelo sabe como realizar as operações CRUD em si mesmos, usando mecanismos comuns

- Cada tipo de modelo possui sua própria tabela no BD
  - todas as linhas da tabela tem uma estrutura idêntica
  - uma linha na tabela == uma instância da classe do modelo
  - cada coluna armazena o valor de um atributo do modelo
  - cada linha tem um único valor como *chave primária* (por convenção, o Rails usa um inteiro e o chama de `id`

`id`	`rating`	`title`	`release_date`
2	Livre	Pets – A Vida Secreta dos Bichos	2016-08-25
11	Livre	Procurando Dory	2016-06-30
22	14 anos	Quando as Luzes se Apagam	2016-08-18

- *Schema*: coleção de todas as tabelas e de suas estruturas

Controladores, Rotas e recursos REST

- Quais decisões de projeto permitirão ao nosso app implementar uma Arquitetura Orientada a Serviços (SOA)?

ROTAS

- Em MVC, cada interação do usuário pode ser tratada por um `controller action`
  - método Ruby que trata a interação
- Uma *rota* mapeia uma tupla `<método HTTP, URI>` para uma ação do controlador

Rota	Ação
`GET /movies/3`	Mostra informação sobre o filme cujo ID=3
`POST /movies`	Cria novo filme usando os dados da requisição
`PUT /movies/5`	Atualiza o filme com ID=5 usando dados da req.
`DELETE /movies/5`	Apaga o filme cujo ID=5

SISTEMA DE ROTEAMENTO DO RAILS

- despacha <método, URI> para a ação do controlador correto
- provê métodos auxiliares (`helper methods`) que geram um par <método,URI> dada uma ação de controlador
- analisa parâmetros da requisição (da URI ou formulário) e cria um hash para acessá-los
- métodos `built-in` para gerar todas as rotas CRUD

`rake routes`

```
  GET /movies          {:action=>"index", :controller=>"movies"}
C POST /movies         {:action=>"create", :controller=>"movies"}
  GET /movies/new      {:action=>"new", :controller=>"movies"}
  GET /movies/:id/edit {:action=>"edit", :controller=>"movies"}
R GET /movies/:id      {:action=>"show", :controller=>"movies"}
U PUT /movies/:id      {:action=>"update", :controller=>"movies"}
D DELETE /movies/:id   {:action=>"destroy", :controller=>"movies"}
```

GET /movies/3/edit HTTP/1.0

- Casa com a rota `GET /movies/:id/edit :action=>"edit", :controller=>"movies"`
- Analisa os parâmetros curinga: `params[:id] = "3"`
- Despacha a requisição para o método `edit` em `movies_controller.rb`
- Para incluir na visão gerada uma URI que irá submeter o formulário para a ação `update` do controlador com `params[:id] = "3"`, chame o método auxiliar `update movie path(3) # => PUT /movies/3`

rake routes

```
  GET /movies          {:action=>"index", :controller=>"movies"}
C POST /movies         {:action=>"create", :controller=>"movies"}
  GET /movies/new      {:action=>"new", :controller=>"movies"}
  GET /movies/:id/edit {:action=>"edit", :controller=>"movies"}
R GET /movies/:id      {:action=>"show", :controller=>"movies"}
U PUT /movies/:id      {:action=>"update", :controller=>"movies"}
D DELETE /movies/:id   {:action=>"destroy", :controller=>"movies"}
```

PERGUNTA

Qual afirmação não é verdadeira em relação às rotas RESTful do Rails e os recursos referenciados por elas:

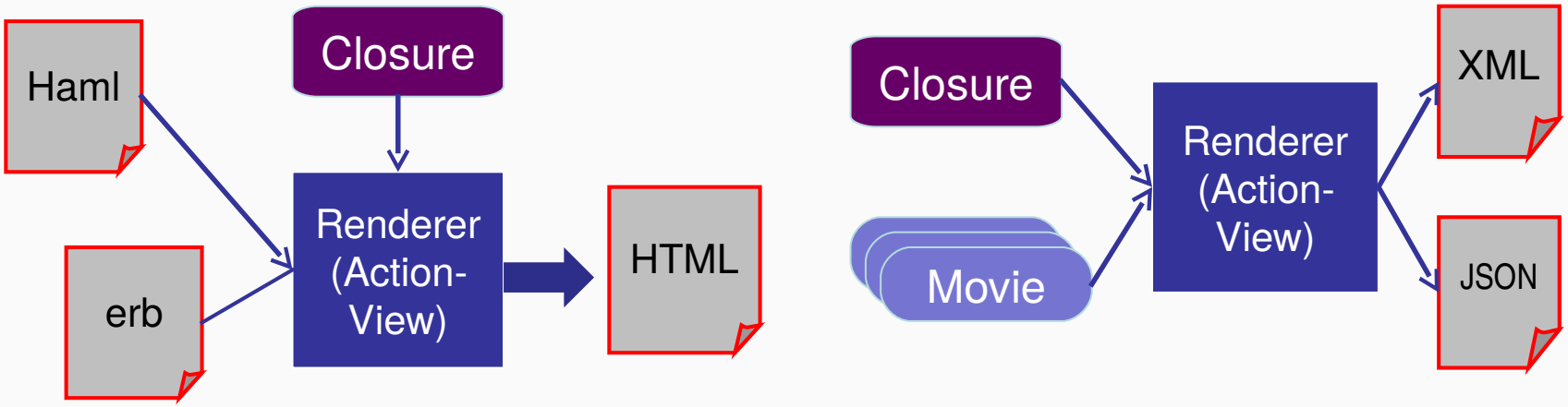
- Um recurso pode representar um conteúdo existente ou uma requisição para modificar algo
- Em um app MVC, toda rota deve disparar uma ação do controlador
- Um conjunto comum de ações RESTful são as ações CRUD nos modelos
- Uma rota sempre deve conter um ou mais parâmetros "curingas" tais como `:id`, para identificar uma instância particular de um recurso

Template Views e Haml

- HTML é como representamos conteúdo para os navegadores
- ... mas como é o processo pelo qual a saída do nosso app se torna HTML?

PADRÃO TEMPLATE VIEW

- A visão consiste de um *markup* intercalado com trechos que serão preenchidos em tempo de execução
  - Normalmente, valores de variáveis ou o resultado da avaliação de um pedaço pequeno de código
- Antigamente, *isso era o app* (ex: PHP)
- Alternativa: padrão *Transform View*



HAML É UM "HTML DE REGIME"

```
%h1.pagename All Movies
%table#movies
  %thead
    %tr
      %th Título do livro
      %th Data de lançamento
      %th Mais informações
  %tbody
    - @movies.each do |movie|
      %tr
        %td= movie.title
        %td= movie.release_date
        %td= link_to "Mais em #{movie.title}",  |
               movie_path(movie) |
= link_to 'Adicionar novo filme', new_movie_path
```

NÃO COLOQUE CÓDIGO NAS SUAS VIEWS

- Sintaticamente, você pode colocar qualquer código na view
- Mas MVC defende que a visão e controle devem ser enxutos
    - A sintaxe de Haml, de propósito, dificulta incluir um monte de código
- *Helpers* (métodos que "embelezam" os objetos para inclusão nas views) tem um espaço reservado em uma app Rails
- Alternativa para Haml: *templates* html.erb (Embedded Ruby), têm mais cara de PHP

Sumário & Reflexões: arquitetura SaaS

Visão geral (arquitetura)

View	HTML & CSS; XML & XPath
Controller	URIs, HTTP, TCP/IP + REST & rotas RESTful
Model	Bancos de dados e migrações + CRUD

2008: “rails não é escalável”

- Escalabilidade é uma preocupação arquitetural; não é restrita à linguagem ou arcabouço
- Arquiteturas de 3 camadas escalam, desde que *stateless*
- Bancos de dados **relacionais** tradicionais não escalam bem
- Várias soluções que combinam sistemas relacionais com não-relacionais (“NoSQL”) escalam muito melhor
- Uso inteligente de *caching* pode melhorar consideravelmente os custos constantes

ARCABOUÇOS, APPS, PADRÕES DE PROJETOS

- Vimos vários padrões de projeto, veremos mais no futuro
- Em 1995, a situação era caótica: os maiores sites Web eram minicomputadores e não 3-camadas/nuvem
- As melhores práticas (padrões) foram “extraídos” e capturados em arcabouços
- Mas a API transcende isso: protocolos de 1969 + linguagem de marcação de 1960 + navegador de 1990 + servidores Web de 1992 continuam funcionando até hoje

ARQUITETURA: SEMPRE HÁ ALTERNATIVAS

Padrão usado	Alternativas
Cliente–servidor	Peer-to-Peer
Shared-nothing (cloud)	Multiprocessamento simétrico (SMP), espaço de endereçamento global e compartilhado
Model–View–Controller	Page controller, Front controller, Template View
ActiveRecord	Data Mapper
URIs RESTful	a mesma URI faz coisas diferentes dependendo do estado do sistema

No futuro sua experiência com apps SaaS vai te fazer questionar qual arquitetura é mais adequada à cada situação.

SUMÁRIO: ARQUITETURA & RAILS

- Model–View–Controller é um padrão arquitetural muito usado para a estruturação de apps
- Rails assume que a app SaaS segue MVC
- Views são Haml com código Ruby embutido que são convertidos em HTML antes de serem enviados ao navegador
- Models são armazenados em tabelas em um BD relacional, acessados com ActiveRecord
- Controllers amarram as views aos modelos com rotas e código nos métodos de controle