# Rotas

## Parâmetros opcionais e valores padrões

Os parâmetros são passados entre {}. Os parâmetros opcionais terminam com “?” e costumam ser inseridos no final. Sendo a ordem: os obrigatórios seguidos dos opcionais.



## Agrupando Rotas

Ao chamar a classe Route, ao invés de chamar o método get chamaremos o método prefix, e a seguir o método group com uma função de call-back.



## Nomeando Rotas

É possível definir um nome para a rota que pode ser utilizado dentro da aplicação como um alias:

Web.php



principal.blade.php



Este é um recurso que evita que os links tenham dependências diretas com as rotas.

OBS: ela não pode ser usada na URL

## Redirecionando Rotas

É possível usar o **Redirect** de algumas formas:

1ª do objeto Route



2ª da função de callback



## Rota de contingência (fallback)

**Fallback** é chamado quando a página não existe:



# Controladores

Encaminhando parâmetros da rota para o controlador

Os parâmetros para o controlador são passados da seguinte forma:

web.php



TesteController.php  


Note que os parâmetros devem ser passados na mesma ordem.

## Encaminhando parâmetros do controlador para visualização

TesteController.php

teste.blade.php

# Views

## Funções do blade

[Documentação do blade](https://laravel.com/docs/7.x/blade)

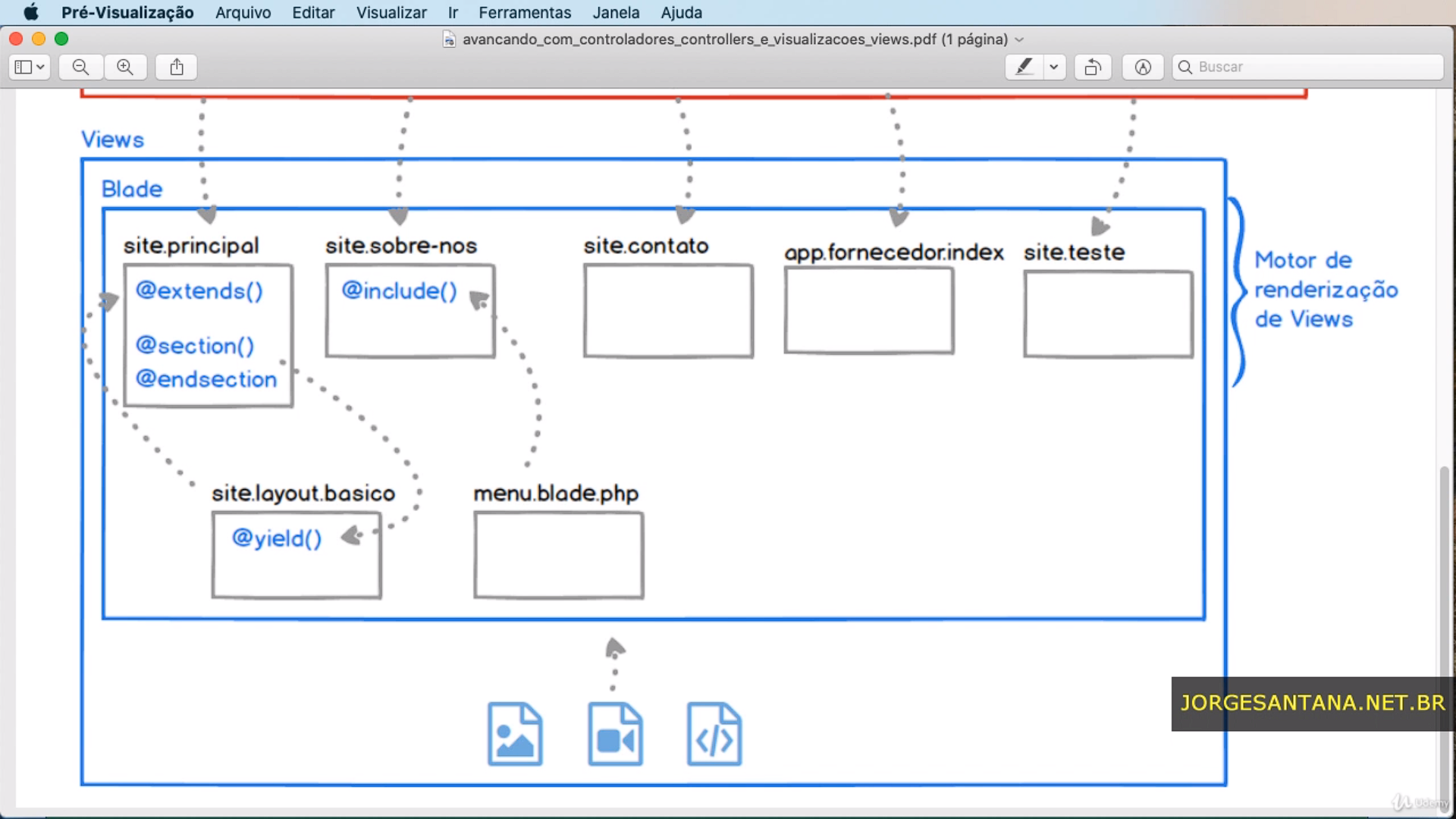
## Assets

É tudo aquilo que é utilizado para complementar o conteúdo do frontend das aplicações web. Ex: Midia, js, css.

Os assets por padrão se encontram no diretório ‘public’.

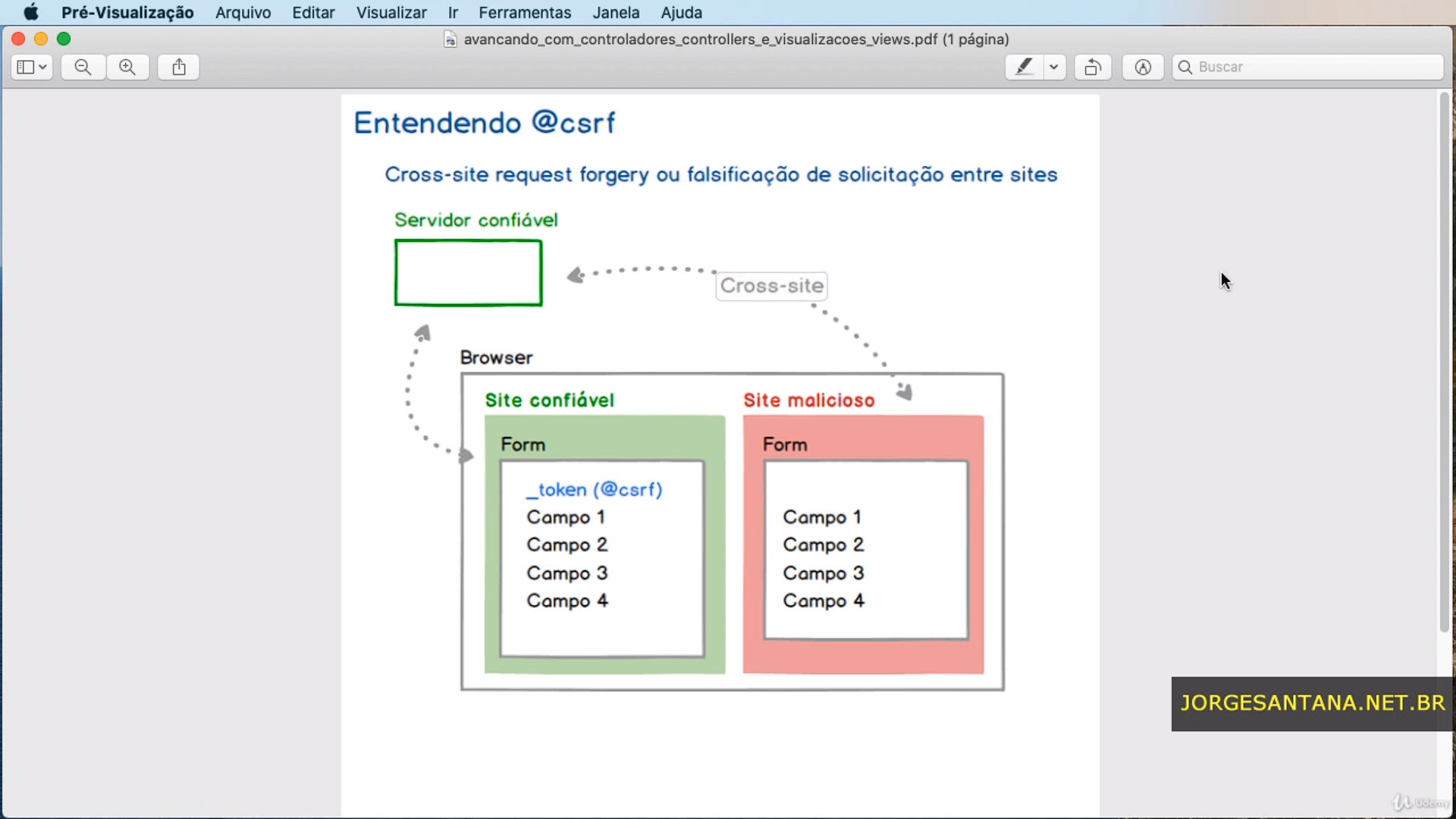


## @extends, @section, @yield e @include



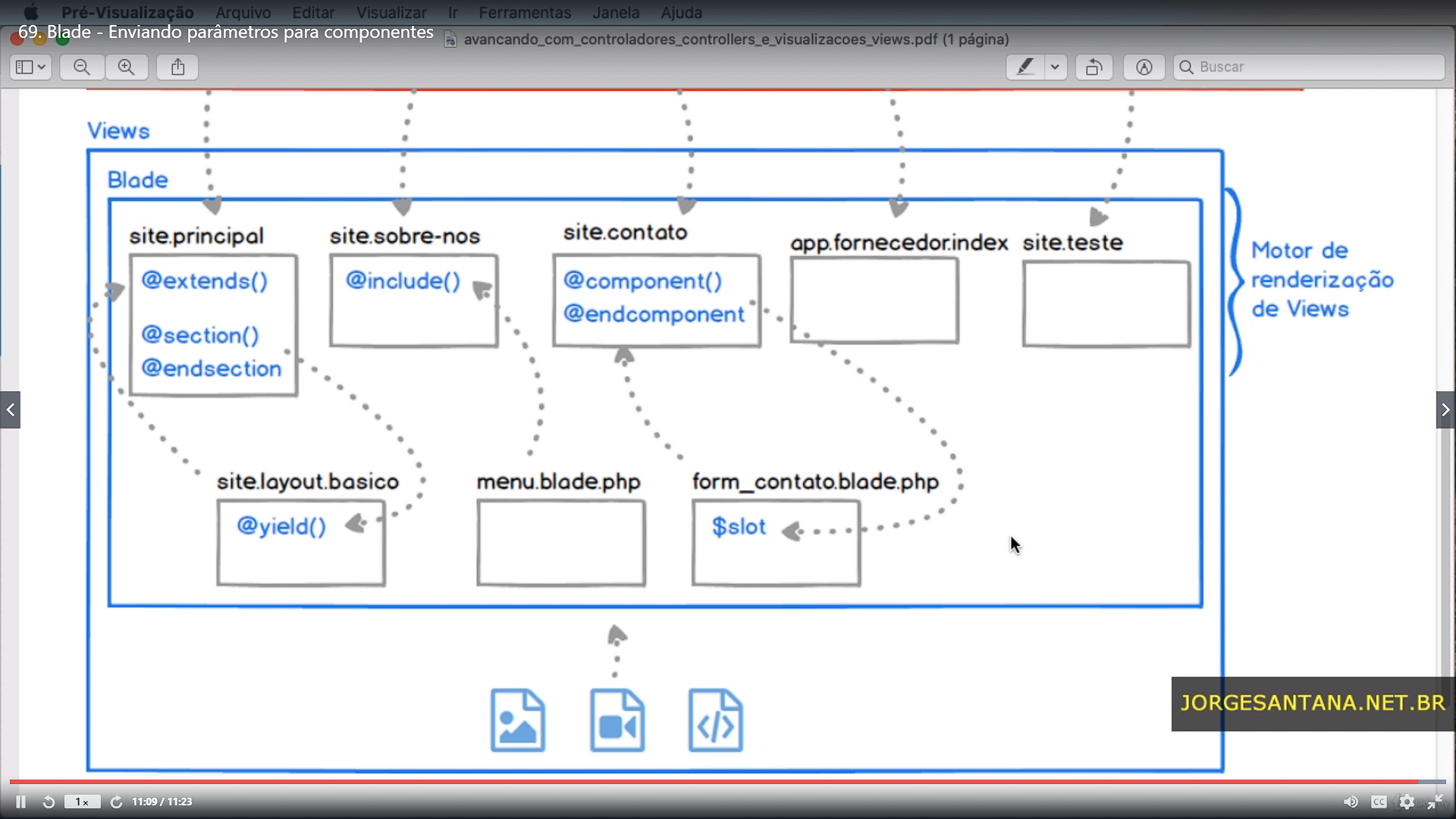
## @csrf

O csrf (cross-site request forgery ou Falsificação de solicitação entre sites) é um token é adicionado ao formulário da pagina legítima ao realizar uma requisição para exibição de um formulário de transferencia de valores, o servidor retorna a pagina aplicando ao formulario um token exclusivo transformando o formulario em algo legitimo.



## @component

**@component** e **@endcomponent** é parecido com ‘include’, porém é possível adicionar informações adicionais para o componente e mostrar o conteúdo enviado pela view para o componente através da variável **$slot** criadaautomaticamente pelo component .



View

Componente



Ou então passando mais variáveis após a chamada do componente como array associativo:

View

Componente



# Models

php artisan make:model create\_produtos\_table

php artisan make:model create\_produtos\_table -m para criar uma migration também

# Migrations

[Documentação migrations](https://laravel.com/docs/7.x/migrations)

Migrations são como controle de versão para seu banco de dados, permitindo uma equipe de modificar e compartilhar o schema da aplicação facilmente. Fazem toda a parte de DDL do banco de dados.

## Criando uma migration

php artisan make:migration create\_produtos\_table

Por padrão esse comando cria um arquivo no diretório /dataqbase/migrations contendo uma classe com o nome da migration criada com dois métodos up() e down() vazios que são usados, respectivamente, para execução da migration e reversão da migration.

Se o arquivo começar com create\_[tabela]\_table, é chamado automaticamente o método create da classe Schema dentro do método up() e o método dropIfExists da classe Schema dentro do método down() .

## Comandos

php artisan migrate:status

atalho para listagem das migrations que já foram executadas. E além disso podemos ver se a migration já foi executada.

php artisan migrate:reset

reverte todas as migrações do banco, ou seja, executa o rollback de todas as migrações independente do batch/lote.

php artisan migrate:refresh

reverte todas as migrações do banco, ou seja, executa o rollback de todas as migrações independente do batch/lote. Após isso executa novamente as migrations.

php artisan migrate:fresh

dropa todas as tabelas. Após isso executa novamente as migrations.

# Tinker

É um **console interativo** que possibilita o **acesso as classes** do projeto pelo cmd. Permite testar os models e o Eloquent ORM. Tudo que é feito no tinker será aplicado nos Controladores.

# Eloquent ORM

Para o laravel converter o nome do model para nome da tabela no banco ele segue os seguintes passos:

* 1º coloca '\_' antes de tudo que está em CamelCase na classe
* 2º coloca tudo em minúsculo
* 3º coloca um 's' no final

Se ele não encontrar o nome da tabela ele vai gerar um erro. Para resolver isso é necessário criar a variável protegida $table com o valor correto do nome da classe para que ela sobreponha essa nomeação automática.



## Inserindo registros

\App\Fornecedor::create(['nome'=>'Fornecedor','ABC','site'=>'fornecedorabc.com.br','uf'=>'RJ','email'=>'contato@fornecedorabc.com.br']);

## Selecionando registros com all( )

retorna uma coleção Collection de objetos:

Fornecedor::all()

## Selecionando registros com find( )

Parecido com o all() porem espera um ou mais PK. Ele não retorna uma coleção de objetos caso pesquise apenas por uma PK:

Fornecedor::find(1)

Fornecedor::find([1,2])

## Selecionando registros com where( )

retorna um builder, ou seja, a contrução da consulta:

SiteContato::where('nome\_coluna','operador\_comparacao','valor')

retorna uma collection:

SiteContato::where('nome\_coluna','operador\_comparacao','valor')->get()

## Selecionando registros com whereIn( ) e whereNotIn( )

SiteContato::whereIn('motivo\_contato',[1,3])

SiteContato::whereNotIn('motivo\_contato',[1,3])

## Selecionando registros com whereBetween( ) e whereNotBetween( )

usado para números e datas

## Selecionando registros com dois ou mais Wheres

SiteContato::where('nome','!=','Fernando')->whereIn('motivo\_contato',[1,2])->whereBetween('created\_at',['2020-08-01 00:00:00','2020-08-31 23:59:59'])->get();

## Selecionando registros com orWhere( )

SiteContato::where('nome','!=','Fernando')->orWhereIn('motivo\_contato',[1,2])->orWhereBetween('created\_at',['2020-08-01 00:00:00','2020-08-31 23:59:59'])->get();

## Selecionando registros com base em parâmetros do tipo data e hora

SiteContato::whereDate('created\_at','2020-08-31');

SiteContato::whereDay('created\_at','31');

SiteContato::whereMonth('created\_at','8');

SiteContato::whereYear('created\_at','2020');

SiteContato::whereTime('created\_at','=','22:30:00');

## Selecionando registros com whereColumn( )

SiteContato::whereColumn('created\_at','updated\_at')

## Selecionando registros aplicando precedência em operações lógicas

SiteContato::where(function ($query){ $query->where('nome','jorge')->orWhere('nome','ana') })->where(function ($query){ $query->whereIn('motivo\_contato',[1,2])->whereBetween('id',[4,6]) })->get()

## Paginate

Fornecedor::all()->paginate()

[documentação](https://laravel.com/docs/8.x/pagination)

## Atualizando registros (save)



## Atualizando registros (fill e save)

Fillable precisa estar setado.



## Atualizando vários registros (where e update)



## Deletando registros (delete e destroy)



## Deletando registros (Soft Delete)



Após isso deve-se adicionar uma migration



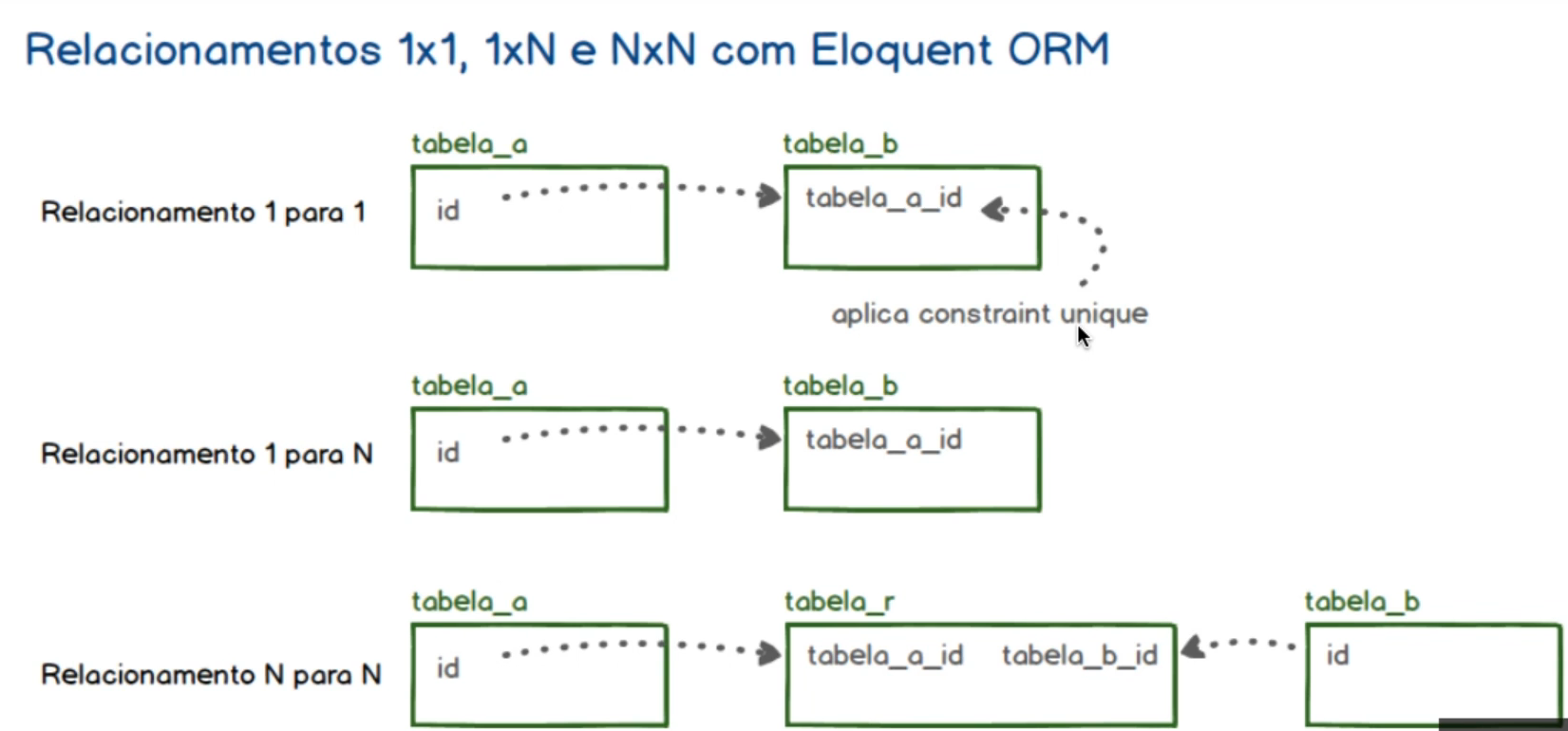
Agora, ao deletar o registro, ele adiciona ao campo deleted\_at uma data significando que o campo foi deletado. Se estiver null, é porque ele não foi removido.



## Selecionando e restaurando registros com soft delete



## Relacionamentos



# Seeders

São responsáveis por **semear o banco de dados** da aplicação **com os dados iniciais de configuração** **ou** mesmo com **dados de testes**. São classes sementes com instruções para semear as tabelas do BD.

* 1 - php artisan make:seeder FornecedorSeeder
* 2 - Depois de criado, fazer as inserções dos registros.

OBS: abaixo existem 3 formas de fazer insert nas tabelas



* 3 - Fazer a chamada do método na classe DatabaseSeeder

$this->call(FornecedorSeeder::class);

* 4 - Executar o comando php artisan db:seed

Ou php artisan db:seed –class=[nome\_da\_classe\_seeder] para selecionar uma classe especifica.

# Factories

Factories são usadas para popular tabelas em massa

php artisan make:factory SiteContatoFactory --model=SiteContato



Após isso, incluir a factory dentro da classe seeder, no metodo run()

factory(SiteContato::class,100)->create();

E então executar

php artisan db:seed --class=SiteContatoSeeder

# Formulários

## Objeto Request

Recupera todas as informações do formulário



## Gravando os dados no banco

É possível fazer essa gravação com o método fill(), create() ou passar pelos atributos e, após isso efetuar a gravação com save().



## Validando campos required, mínimo e máximo de caracteres, email e unique

[Validation documentação](https://laravel.com/docs/7.x/validation#available-validation-rules)

Controller

View

## Repululando o formulário com old( )

View

## Customizando apresentação de erros

View

## Customizando mensagens de feedback

Controller

# Middlewares

Middlewares são camadas de software implementadas entre aplicações diferentes. Intermediando comunicação de entrada e saída de aplicações diferentes.

Eles atuam na interceptação de requisições http feitas por navegadores e também na manipulação da resposta que é dada pela aplicação a esses mesmos browsers.

Os Middlewares de **request** http capturam a requisição antes que ela seja submetida para o núcleo da aplicação, tomando inúmeras ações nesse momento. Ex:

* verificar autenticação do usuário para bloquear/liberar acesso;
* registrar cada um dos acessos as rotas, incluindo informação do usuário que fez o acesso e quando ele acessou, o ip origem, o browser utilizado;
* verificar se o usuário aceitou os termos de uso para então liberar a navegação.

Os Middlewares de **response** http manipulam a resposta http para um propósito especifico, padronizando os cabeçalhos quando trabalhando com CORS (Compartilhamento de recursos com origens diferentes).

As aplicações CORS são aplicações web que se comunicam com domínios, protocolos ou portas diferentes da própria origem. É comum quando nossa aplicação expõe APIs para aplicações externas (acessos a certas rotas).

## Criando um middleware

php artisan make:middleware LogAcessoMiddleware

Está localizado em app/http/Middleware

A função handle() é executada automaticamente quando o middleware é ativado.



O método $next($request) passa a requisição para a aplicação.

Usando a middleware no contexto de rotas:



OBS: A ordem dos métodos não importa



## Implementado os middlewares no construtor dos controladores



## Implementando middleware para todas as rotas

A classe App/Http/Kernel.php é responsável por iniciar diversos middlewares e defini-los para grupos de rotas.

Os grupos de rotas:

* WEB
* API

Os middlewares podem ser alocados diretamente a esses grupos de rotas de modo que funcione a todas as rotas desse grupo, podendo ser também dos 2 grupos, sem necessidade de fazer a chamada desses middlewares individualmente nas rotas ou nos controladores.



OBS: Acessar o arquivo App/Http/Kernel.php

## Apelidando middlewares

Kernel.php

Web.php

SobreNosController.php

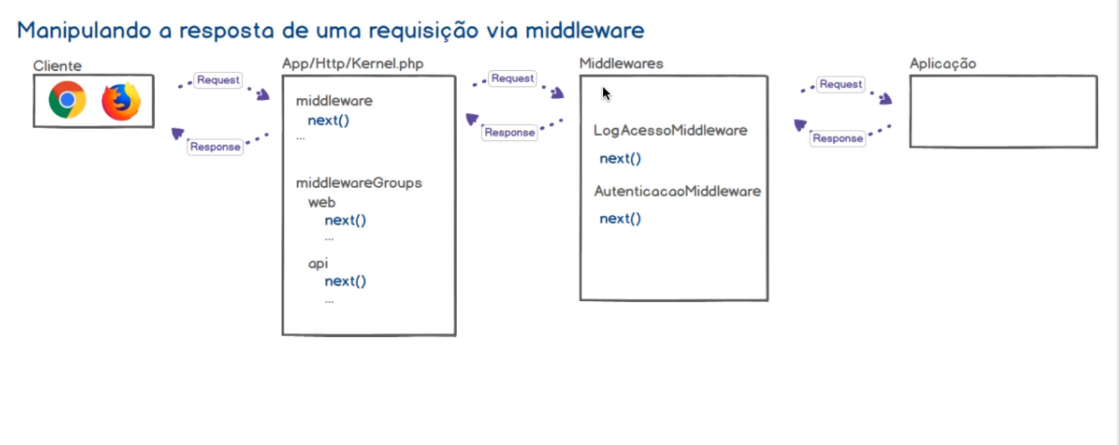
## Passando parâmetros para o middleware

Os parâmetros serão a partir do 3º parâmetro do método handle()

AutenticacaoMiddleware.php

Web.php

## Manipulando a resposta da requisição com o middleware



# Laravel UI

Após instalar o laravel no projeto. Intalar o laravel UI

Composer require laravel/ui:^3.2

Novos grupos de comandos serão adicionados no artisan.

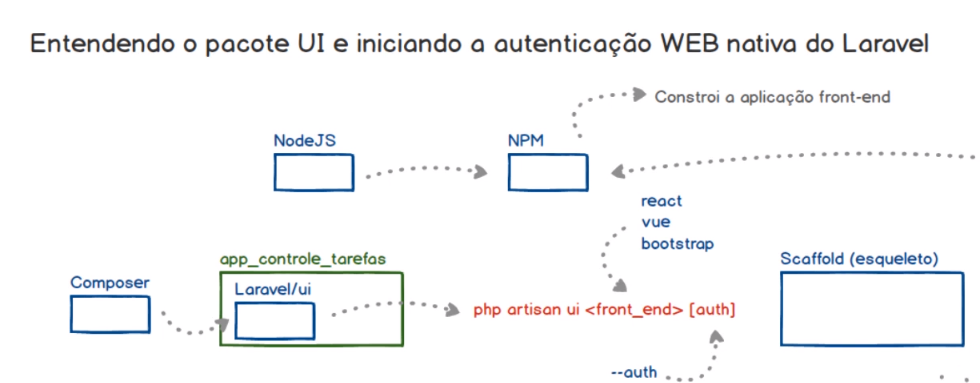
Ao executar o comando artisan para inserir dependências do laravel ui com bootstrap ao projeto ele criará um Scaffold.

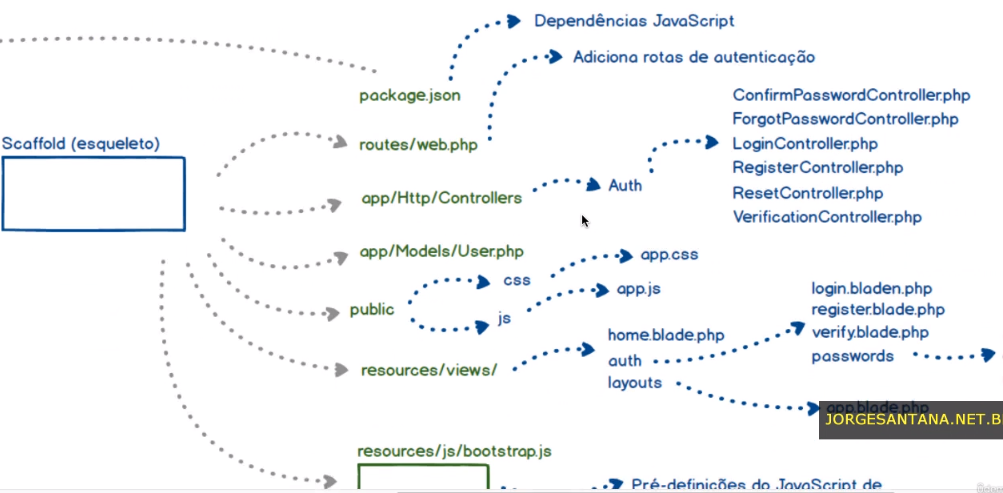
Php artisan ui bootstrap –auth

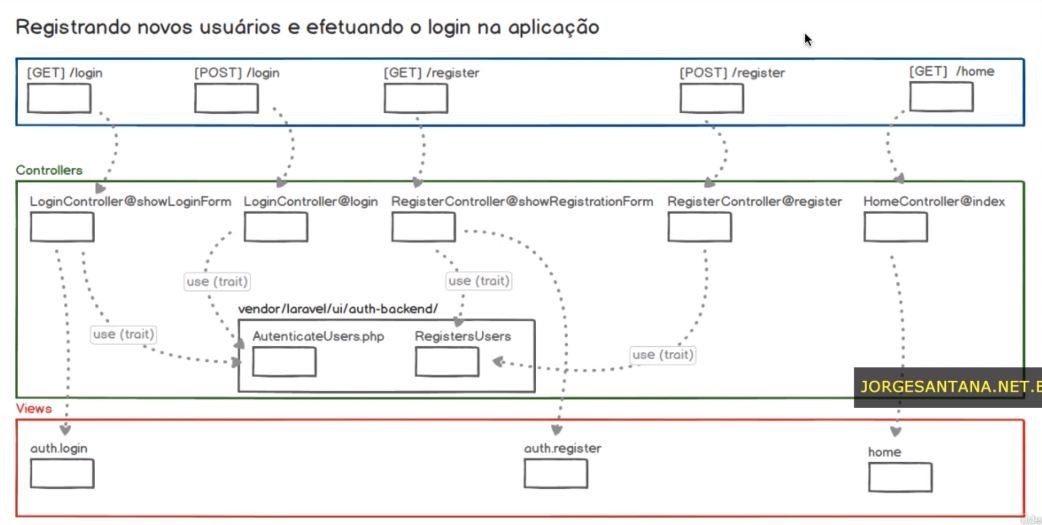
Após o comando acima o laravel pedira para rodar os comandos npm install e npm run dev para compilar esse scaffold (bundle).

* npm install -> Le o package.json para download das dependencias
* npm run dev -> gera os assets do front de acordo com a tecnologia selecionada (bootstrap).

OBS: pode ser necessário instalar umas dependências a mais contidas no mix do terminal e rodar o npm rundev novamente.





# Emails

Acesse o arquivo .env e altere as configurações



Criar uma classe de email com um novo template markdown para ele.

php artisan make:mail MensagemTesteMail --markdown emails.mensagem-teste

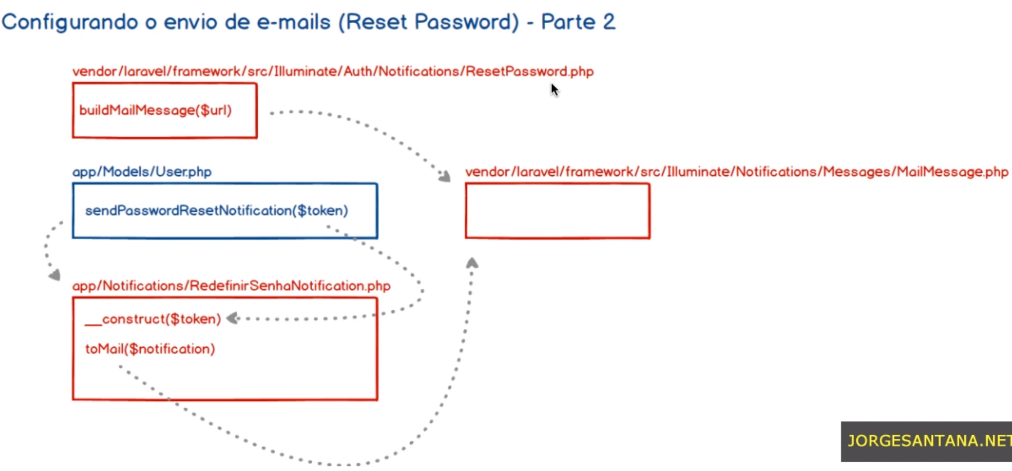
Será ciado um novo diretório no app/ chamado mail/ .

Também um outro diretório dentro de views: emails/ com o arquivo mensagem-test.blade.php .

* Markdown são views prontas com componentes e layouts que poder ser reutilizados.
* Blade permite renderizar as views.

Web.php





App/Models/User.php

Criando uma classe de notificação

php artisan make:notification RedefinirSenhaNotification



MustVerifyEmail

Adicionar a interface MustVerifyEmail a classe User



Web.php



# Importação e Exportação de arquivos

[Documentação do Laravel Excel](https://docs.laravel-excel.com/3.1/getting-started/installation.html)

Publish – para fazer configurações específicas desse provedor dentro da aplicação para não ter que mexer na classe original dentro do diretório vendor sobrepondo as configurações de acordo com o serviço

[Uso Rápido do plugin](https://docs.laravel-excel.com/3.1/exports/)

## Exportando para pdf

Instalar o pacote pdf composer require mpdf/mpdf

Acessar o arquivo config/excel.php e modificar a linha com index ‘pdf’

## Definindo títulos na exportação

Incluir a interface [WithHeadings](https://docs.laravel-excel.com/3.1/exports/mapping.html#adding-a-heading-row) a classe de exportação.

## Corrigindo caracteres especiais no CSV

Acessar o arquivo config/excel.php e modificar a linha com index ‘csv[use\_bom]’ para true

## Manipulando os dados exportados linha por linha e formatando datas

Incluir a interface [WithMapping](https://docs.laravel-excel.com/3.1/exports/mapping.html#mapping-rows) a classe de exportação.



## Instalando DOMPDF

[Documentação do Laravel DOMPDF](https://github.com/barryvdh/laravel-dompdf)

Instalar o pacote pdf composer require barryvdh/laravel-dompdf

php artisan vendor:publish --provider="Barryvdh\DomPDF\ServiceProvider"

Será criado um arquivo config/dompdf.php

## Exportando um arquivo no formato PDF com o DOMPDF no controlador



## Download vs Stream



# API

php artisan make:model Marca -m -c -r (migration e controllers com recursos)

php artisan make:model Marca -mcr

php artisan make:model Marca --all

php artisan make:model Marca -a (migration, controller, seeder e factory)

As APIs encontran-se em /routes/api.php e para chama-las é necessário chamar a rota **api/**

No retorno, sempre deve-se retornar um array pois automaticamente o framework o transforma em json, inclusive muda o content-type.

## Resource x apiResource

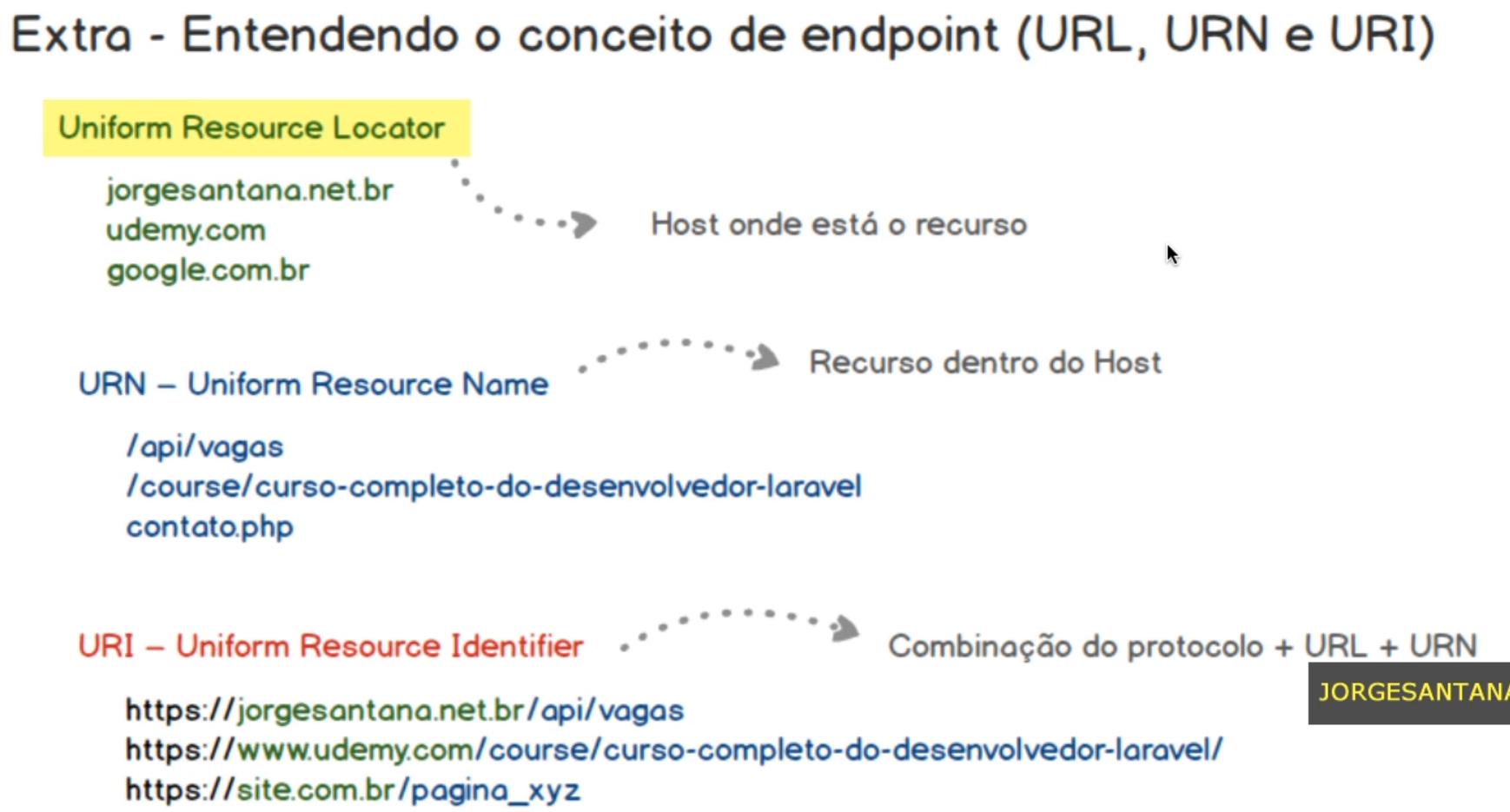
Na **apiResource** não temos as rotas create e edit

## PUT x PATCH

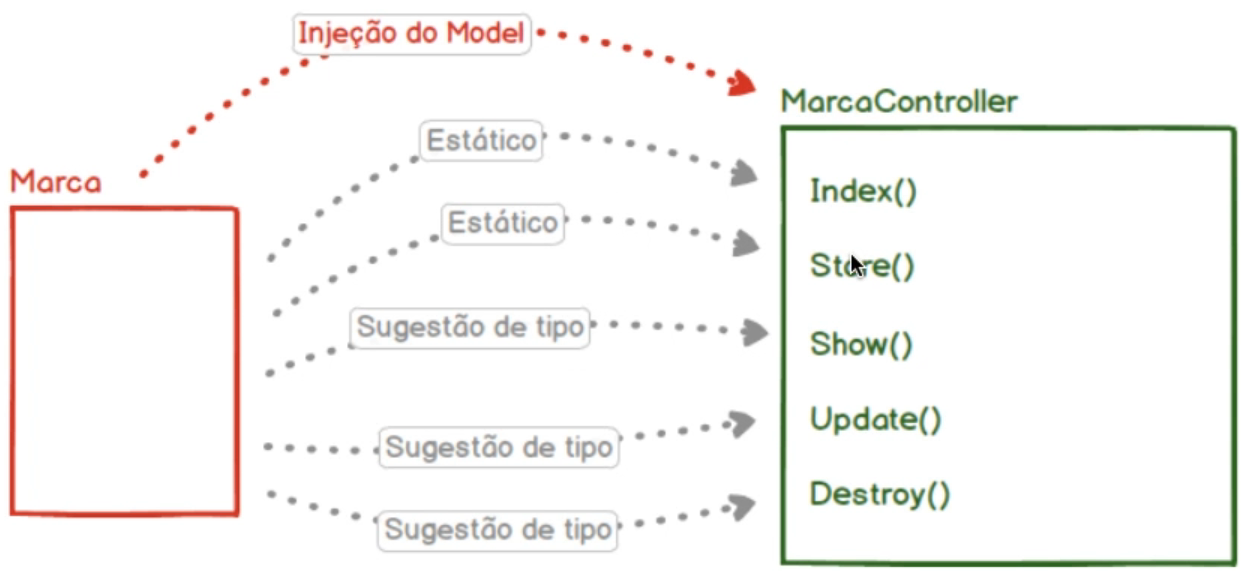
PUT - Atualizar parte do recurso semanticamente

PATCH - Atualizar todo o recurso semanticamente

## Endpoint (URL, URN e URI)



## Injetando a instância do Model no Controller e como lidar com o Type Hinting



* Estático

Model::all()

Model::create()

* Sugestao de tipo (typing hinting)



* Injeção de model



## Validações nos endpoints

Status code



Para fazer a validação é necessário que o cliente (browser) informe no header que o atributo **accept** seja igual a **application/json**

### Implementando validações no model



Após isso é necessário chamá-las no controller

Store



Update



Validação [unique](https://laravel.com/docs/5.1/validation#rule-unique):

1. Nome da tabela
2. Nome da coluna
3. Id do campo

Lidando com PUT/PATCH



# Upload de arquivos

Lado cliente:

accept = application/json

ecntype=form-data

Lado servidor:



Validação da imagem



#### php artisan storage:link

para criar um link simbólico para o disco

Para remover um arquivo, é preciso incluir a facade Storage

