

# Engenharia de Software

**Prof. Vinicius Cardoso Garcia**

[vcg@cin.ufpe.br](mailto:vcg@cin.ufpe.br) :: [@vinicius3w](https://twitter.com/vinicius3w) :: [viniciusgarcia.me](http://viniciusgarcia.me)

[IF977] Engenharia de Software

<http://bit.ly/vcg-es>

# Licença do material

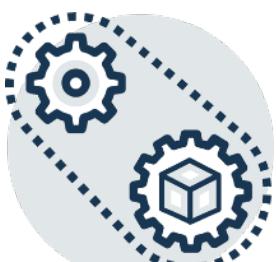
Este Trabalho foi licenciado com uma Licença

Creative Commons - Atribuição-NãoComercial-  
Compartilhual 3.0 Não Adaptada



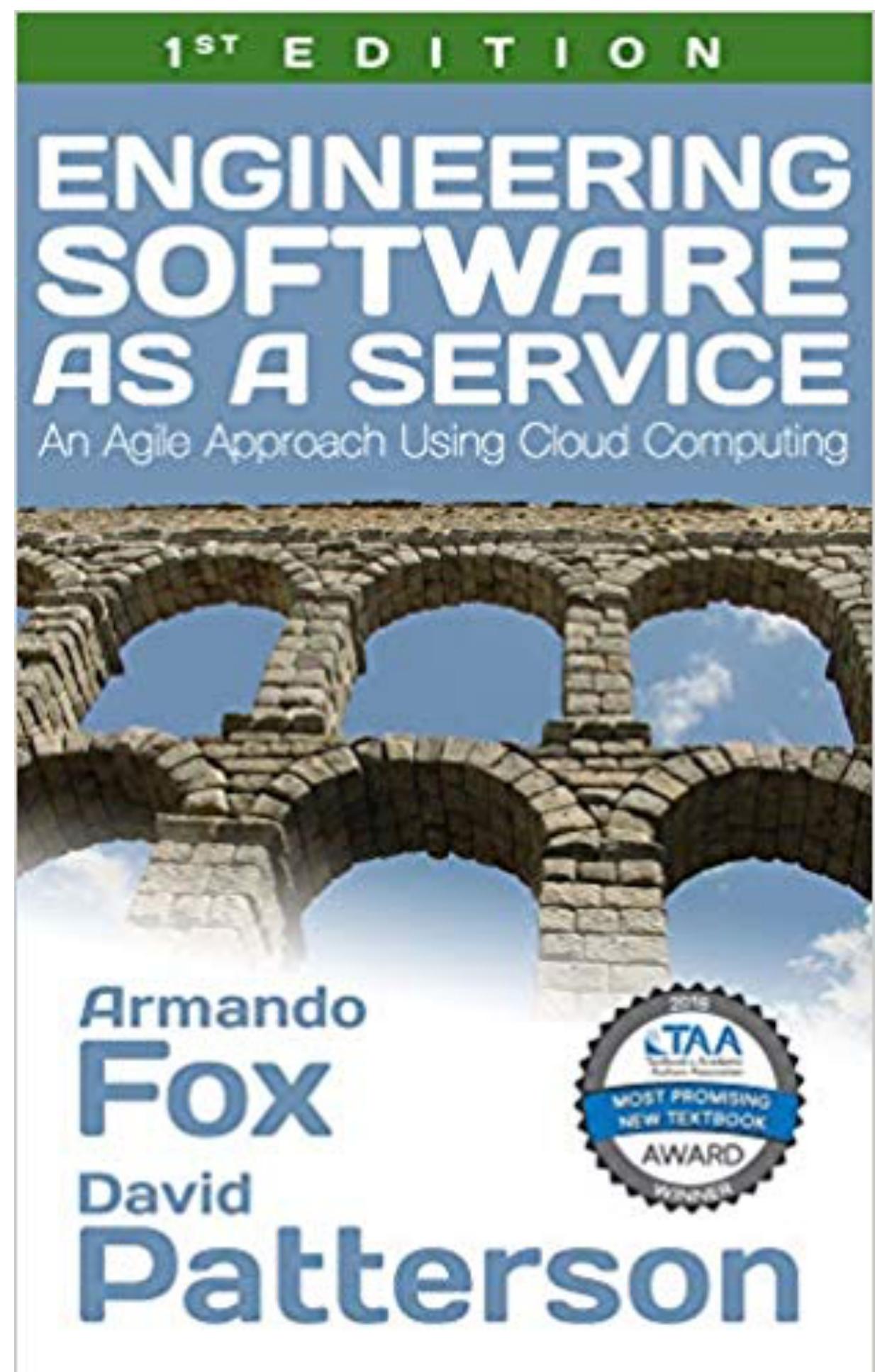
Mais informações visite

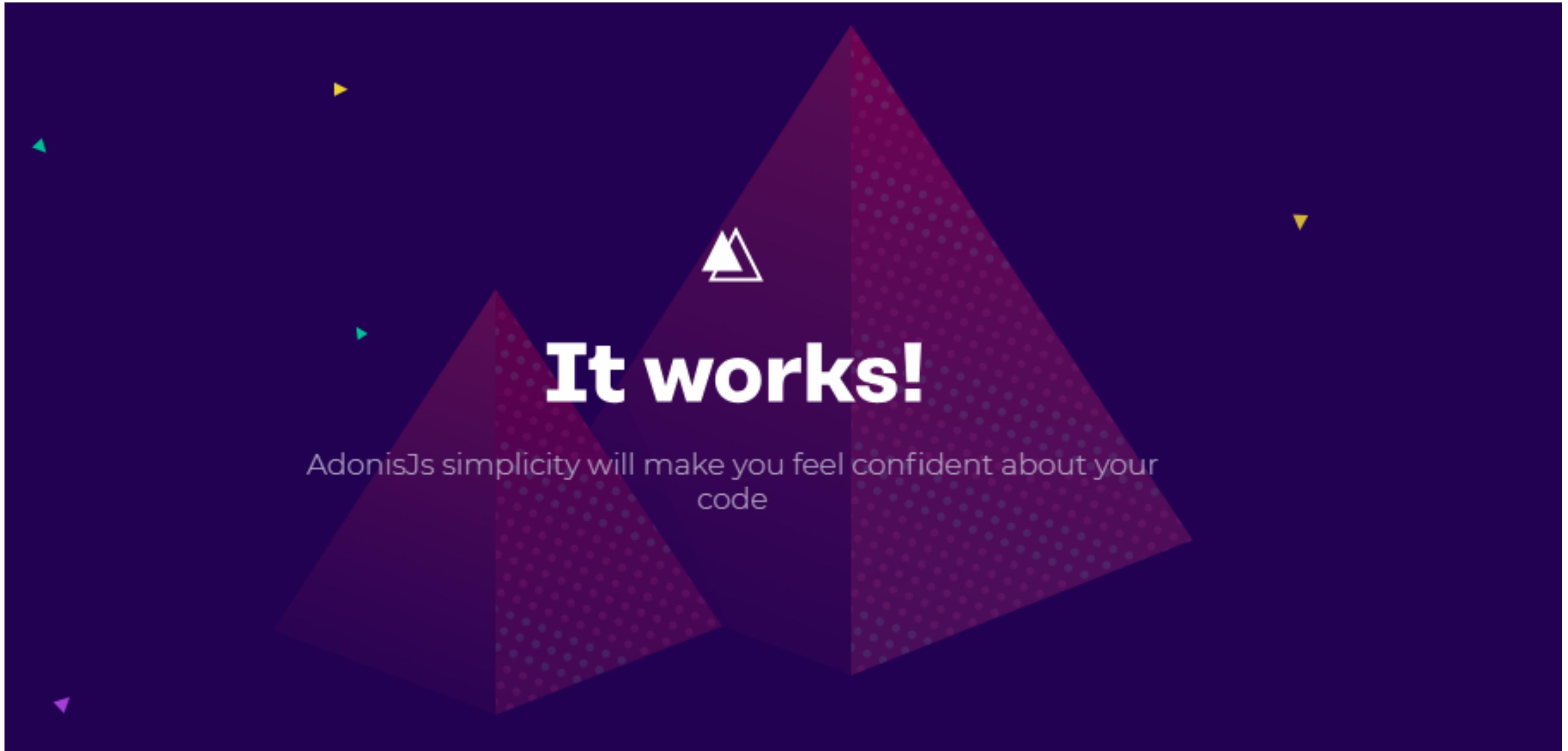
[http://creativecommons.org/licenses/by-nc-sa/  
3.0/deed.pt](http://creativecommons.org/licenses/by-nc-sa/3.0/deed.pt)



# Referências

- A biblioteca do Desenvolvedor de Software dos dias de hoje
  - <http://bit.ly/31WYK5f>
- SWEBOK: Guide to the Software Engineering Body of Knowledge (SWEBOK)
  - <http://www.computer.org/web/swebok>
- Engineering Software as a Service: An Agile Approach Using Cloud Computing
  - <http://www.saasbook.info/>
- Marco Tulio Valente. Engenharia de Software Moderna
  - <https://engsoftmoderna.info/>





# Criando CRUDI e relações em API REST no AdonisJS



4

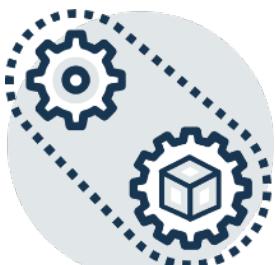
# Sprint Review

- A API já está mais do que **pronta** para partirmos para **os próximos passos** como **criação de imóveis**, **upload de imagens** e **retorno de imóveis próximos baseados na latitude e longitude do usuário**
- Com esse **token** que recebemos no **login** vamos conseguir **validar nas requisições que precisam de autenticação**, como um cadastro de imóvel, se o usuário é válido e está autenticado em nossa aplicação
- Criamos toda nossa API com AdonisJS do total zero realizando **cadastro e autenticação** via JWT e salvando nossos dados no banco.
- Aprendemos também alguns conceitos importantes como o **ORM**, **models**, **controllers**, **migrations**, **JWT**, etc...
- Código dessa parte
  - <https://github.com/vinicius3w/easylanding-server/tree/v0.1-alpha>



# Próximos passos

- Nessa segunda etapa iremos criar os seguintes recursos:
  - Listagem de imóveis;
  - Exibição de um único imóvel com imagens;
  - Remoção de imóveis;
  - Relacionamento entre usuários e imóveis (um para muitos);
  - Relacionamento entre imóveis e imagens (um para muitas);
  - (tudo isso utilizando API REST)



# Criando models e migrations

- Temos que trabalhar na nova entidade da nossa aplicação, **Imóvel**
- Precisamos **criar a tabela** no banco de dados que irá **armazenar** esses valores além de criar o seu respectivo **model** que será usado para realizarmos as **ações do ORM** como busca, cadastro, edição, remoção, filtros, etc...
- Chamaremos nosso Imóvel de **Property**



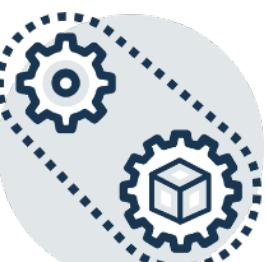
# Criando models e migrations

- Para iniciar vamos criar nosso model, migration e controller executando o comando:

```
adonis make:model Property -m -c
```

- Você receberá um feedback como o seguinte:

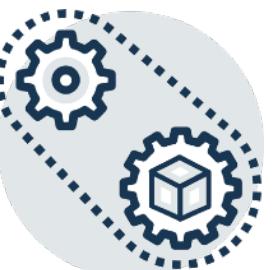
- ✓ create app/Models/Property.js
- ✓ create database/migrations/1530569796148\_property\_schema.js
- ✓ create app/Controllers/Http/PropertyController.js



# Adicionando os campos

- com a estrutura pronta, vamos adicionar os campos à tabela de imóveis alterando nossa **migration** (`*_property_schema.js`) para incluir também os campos:
  - `user_id` (Referência ao usuário que criou o imóvel);
  - `title` (Título do imóvel);
  - `address` (Endereço completo);
  - `price` (Preço da diária);
  - `latitude`;
  - `longitude`;
- Note que o campo `user_id` que possui um relacionamento com a tabela de usuários
- `adonis migration:run` para criar a tabela

```
1  'use strict'          You, 14 days ago • Creating CRUD features and relations in RE
2
3  /** @type {import('@adonisjs/lucid/src/Schema')} */
4  const Schema = use('Schema')
5
6  class PropertySchema extends Schema {
7    up () {
8      this.create('properties', (table) => {
9        table.increments()
10       table
11         .integer('user_id')
12         .unsigned()
13         .references('id')
14         .inTable('users')
15         .onUpdate('CASCADE')
16         .onDelete('CASCADE')
17       table.string('title').notNullable()
18       table.string('address').notNullable()
19       table.decimal('price').notNullable()
20       table.decimal('latitude', 9, 6).notNullable()
21       table.decimal('longitude', 9, 6).notNullable()
22       table.timestamps()
23     })
24   }
25
26   down () {
27     this.drop('properties')
28   }
29 }
30
31 module.exports = PropertySchema
32 ...
```



# Entidade para Imagens

- Como os imóveis poderão ter inúmeras imagens vamos separar isso em outra tabela com novamente um relacionamento 1-N
- Dessa vez não criaremos o controller já que as imagens serão enviadas junto com os demais dados na criação de um imóvel
  - adonis make:model Image -m
- Agora vamos alterar a migration para conter a referência à tabela de imóveis e também um campo para armazenar o caminho físico da imagem na nossa aplicação

```
1 'use strict'          You, 14 days ago • Creating CRUD features and re...
2
3 /** @type {import('adonisjs/lucid/src/Schema')} */
4 const Schema = use('Schema')
5
6 class ImageSchema extends Schema {
7   up () {
8     this.create('images', (table) => {
9       table.increments()
10      table
11        .integer('property_id')
12        .unsigned()
13        .references('id')
14        .inTable('properties')
15        .onUpdate('CASCADE')
16        .onDelete('CASCADE')
17      table.string('path').notNullable()
18      table.timestamps()
19    })
20  }
21
22  down () {
23    this.drop('images')
24  }
25 }
26
27 module.exports = ImageSchema
```



# Relacionamentos no AdonisJS

- No arquivo app/Models/User.js vamos informar que um **usuário pode ter muitos imóveis cadastrados**, adicionando um método no fim da classe
- Veja que para criamos um relacionamento no model sempre criamos um novo método retornando uma relação que pode ser “**1-N**”, “**1-1**”, “**N-N**” ou até relacionamentos polimórficos ou mais avançados
  - Outro ponto é que referenciamos outro model da aplicação em formato de **string** mesmo

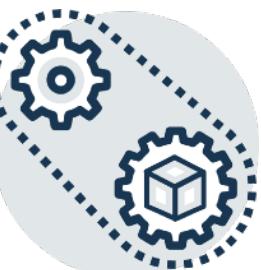
```
'use strict'

const Model = use('Model')
const Hash = use('Hash')

class User extends Model {

    properties () {
        return this.hasMany('App/Models/Property')
    }
}

module.exports = User
```



# Relacionamentos no AdonisJS

- No model de **imóvel** vamos adicionar o relacionamento contrário informando que o **imóvel sempre pertence a um usuário** e aproveitando pra adicionar o relacionamento que um **imóvel possui muitas imagens**
- Um último caso seria adicionarmos o relacionamento que a imagem pertence à um imóvel, porém nesse caso não é necessário
  - Isso porque **jamais iremos criar a imagem antes para depois relacioná-la com um imóvel**, o caminho é sempre inverso, criaremos o imóvel e junto com ele suas imagens, ou seja, **não precisamos do relacionamento contrário**

```
'use strict'

const Model = use('Model')

class Property extends Model {
    user () {
        return this.belongsTo('App/Models/User')
    }

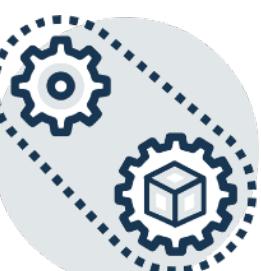
    images () {
        return thishasMany('App/Models/Image')
    }
}

module.exports = Property
```



# Rotas e controllers

- Todas nossas ações daqui pra frente vão girar em cima do controller de imóvel, por isso, abra o arquivo **PropertyController**
- Vamos começar removendo dois métodos desse arquivo: **create** e **edit** que são funções inúteis em um ambiente de API REST já que servem apenas para exibir os formulários de criação e edição
- Cada método que sobrou tem uma responsabilidade:
  - **index**: Listar todos registros;
  - **show**: Exibir um registro;
  - **store**: Criar novo registro;
  - **update**: Alterar um registro;
  - **destroy**: Remover um registro;

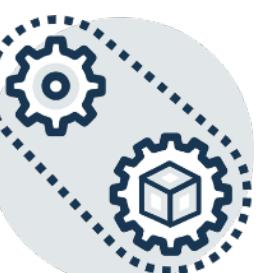


# Criando as rotas

- Temos que criar as rotas para cada um desses métodos no arquivo start/routes.js
- Ao invés de criar **uma rota para cada método** o Adonis nos oferece um **helper** chamado **resource** que pode ser usado

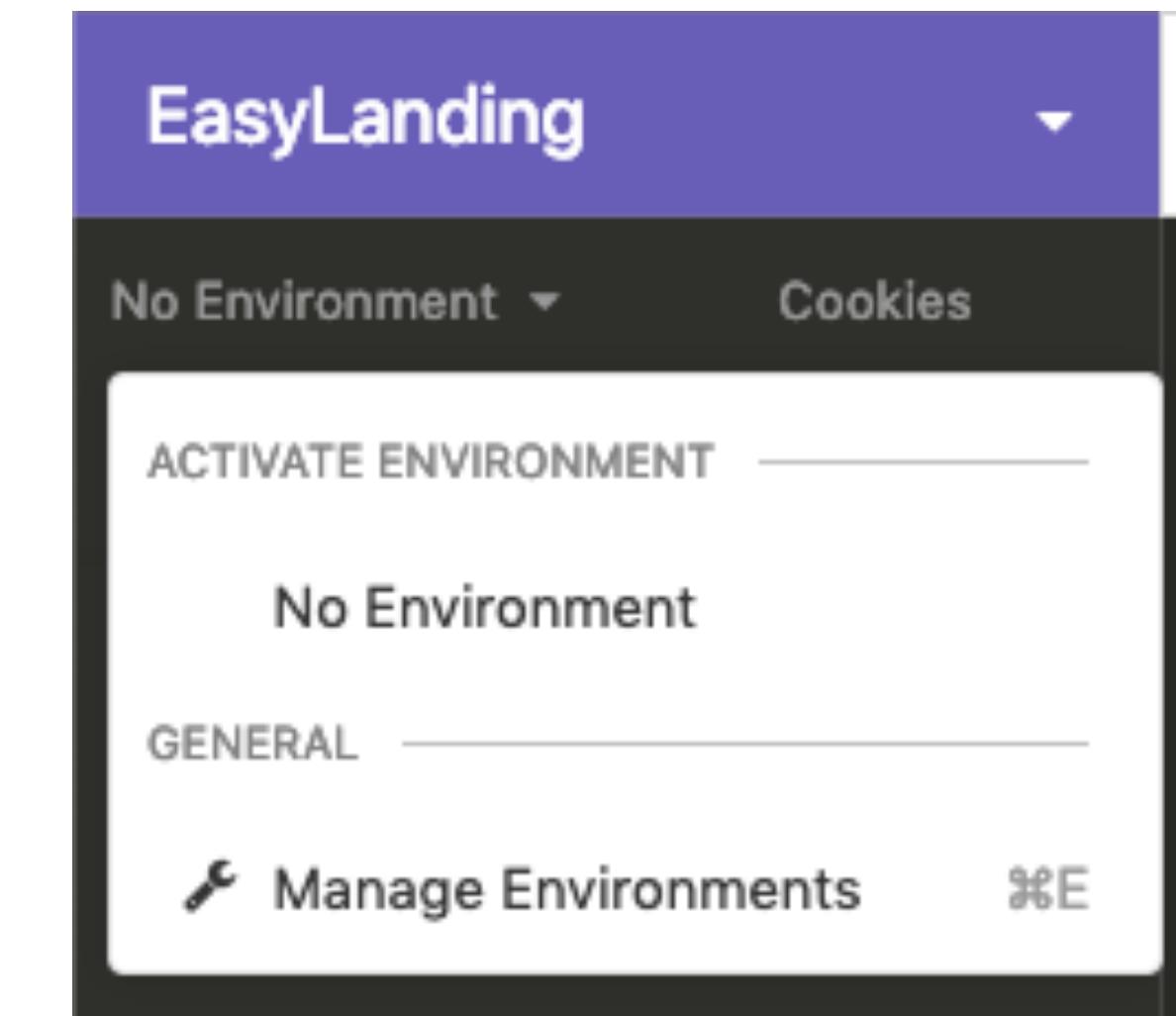
```
Route.resource('properties', 'PropertyController')
  .apiOnly()
  .middleware('auth')
```

- Estamos informando para o Adonis criar **todas** as rotas de listagem, exibição, criação, edição e remoção de imóveis em um único comando
- O método `apiOnly()` garante as rotas `create` e `edit` que deletamos anteriormente **não tenham rota**, já o `middleware auth` vai garantir que usuários **não autenticados não possam utilizar essas rotas**



# Configurando Insomnia

- Para testar a API que estamos desenvolvendo eu deixei um arquivo que você pode importar no seu Insomnia que já contém todos métodos que iremos ter no nosso controller
- Basta ir na seção “Import” no Insomnia na opção de “Import from URL” e informar o seguinte endereço:
  - [https://raw.githubusercontent.com/vinicius3w/easylanding-server/master/Insomnia\\_2019-08-31.json](https://raw.githubusercontent.com/vinicius3w/easylanding-server/master/Insomnia_2019-08-31.json)
- Pronto, você terá todas as rotas disponíveis na aplicação já com parâmetros fictícios para serem enviados em cada rota
- A única coisa que você precisa fazer é **se autenticar com um usuário, copiar o token JWT e colar na seção environment do Insomnia**



## Base Environment

```
1 ▾ {  
2   "base_url": "http://localhost:3333",  
3   "token":  
4     "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQ  
5       i0jEsImlhdCI6MTUzMDU3OTA0MH0.Q0GavD6_6oadYy0M  
6       mUDTD0z4HH-4chMflbhHLybKNio"  
7 }
```



# Criando CRUD

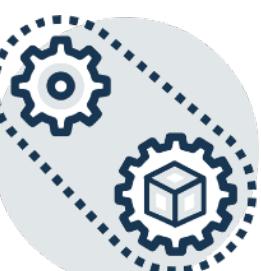
- No **controller de imóveis** adicione a seguinte linha **logo após a notação 'use strict'** para termos acesso ao model de imóvel dentro do código:

```
const Property = use('App/Models/Property')
```

- Vamos começar com os métodos `index`, `show` e `destroy` que são mais simples

```
async index () {  
  const properties = Property.all()  
  return properties  
}
```

- Por enquanto vamos apenas retornar **todos os imóveis nesse método** (mais tarde iremos buscar apenas imóveis próximos baseados na localização do usuário)
- Execute no **Insomnia** a requisição “**Index**” na pasta “**Imóveis**”, um **array vazio** deverá ser retornado



# Método show

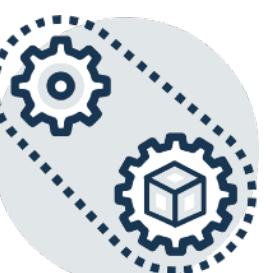
```
async show ({ params }) {  
  const property = await Property.findOrFail(params.id)  
  
  await property.load('images')  
  
  return property  
}
```

- O método show possui algumas peculiaridades comparado ao anterior. Nesse caso, podemos acessar os parâmetros enviados na URL (ex: /properties/1) através da variável **params** e assim buscar exatamente o registro no banco com esse **ID**
- Logo após, executamos o método **load** para fazer o **carregamento de um relacionamento do model**, dessa forma, nossa API retornará também as imagens do imóvel buscado
- Executando a ação “**Show**” na pasta “**Imóvel**” no Insomnia obteremos um **erro HTTP** já que **não temos nenhum imóvel cadastrado**, mas você pode experimentar cadastrar um imóvel com imagens e terá um resultado semelhante a esse

200 OK   TIME 40.2 ms   SIZE 482 B

Preview ▾   Header 4   Cookie   Timeline

```
1 {  
2   "id": 1,  
3   "user_id": 1,  
4   "title": "Apartamento 201",  
5   "address": "Rua sem nome, Rio do Sul - SC",  
6   "price": "120.00",  
7   "latitude": "-27.204534",  
8   "longitude": "-27.204534",  
9   "created_at": "2018-07-02 23:10:40",  
10  "updated_at": "2018-07-02 23:10:40",  
11  "images": [  
12    {  
13      "id": 1,  
14      "property_id": 1,  
15      "path": "/caminho/imagem.jpg",  
16      "created_at": "2018-07-02 23:11:00",  
17      "updated_at": "2018-07-02 23:11:00"  
18    },  
19    {  
20      "id": 2,  
21      "property_id": 1,  
22      "path": "/caminho/outra.jpg",  
23      "created_at": "2018-07-02 23:11:11",  
24      "updated_at": "2018-07-02 23:11:11"  
25    }  
26  ]  
27 }
```



# Método destroy

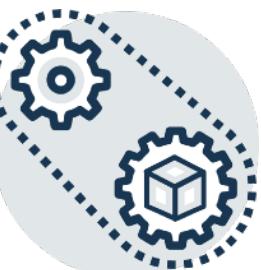
- Esse método por sua vez novamente **busca o imóvel** através do parâmetro **ID** da **URL** e faz uma **verificação se o dono do imóvel é o mesmo usuário** tentando fazer a requisição, caso contrário não permite a operação retornando a mensagem de erro
- Caso validado, o imóvel é removido

```
async destroy ({ params, auth, response }) {  
  const property = await Property.findOrFail(params.id)  
  
  if (property.user_id !== auth.user.id) {  
    return response.status(401).send({ error: 'Not authorized' })  
  }  
  
  await property.delete()  
}
```

401 Unauthorized    TIME 721 ms    SIZE 26 B

Preview ▾    Header 4    Cookie    Timeline

```
1 ▾ {  
2   "error": "Not authorized"  
3 }
```



# O que fizemos até aqui?

- Demos mais um passo para a finalização da nossa aplicação
  - API para parte do CRUDI de Imóveis
- Código dessa sprint:
  - <https://github.com/vinicius3w/easylanding-server/tree/v0.2-alpha>
- Na próxima sprint vamos mais a fundo na nossa API construindo as seguintes funcionalidades:
  - Criação/edição de imóveis com upload
  - Busca por distância através de GPS



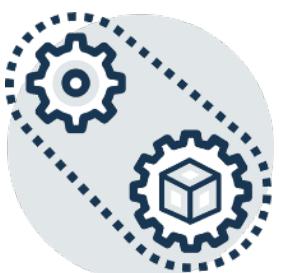


PRACTICAL

RELIABLE

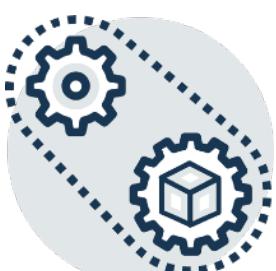
PRODUCTIVE

# Upload de imagens e geolocalização



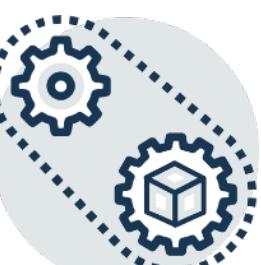
# Sprint planning

- Essa é terceira e última sprint da construção da API de backend com AdonisJS antes de partirmos para o frontend
- Vamos terminar a API REST da nossa aplicação criando a parte de **cadastro/edição** de imóveis com **upload** de imagens e filtro por **geolocalização** (latitude/longitude)



# Geolocalização

- **Usabilidade:** O ideal é que assim que acesse, o usuário veja em um mapa os imóveis mais próximos dele, vamos manter um valor **padrão de 10km**
- Precisamos adicionar uma nova regra de negócio ao nosso **Model de Property**
  - Mas por que no Model?
  - Vamos criar um **filtro por distância** que pode ser utilizado em **mais locais** do nosso código
  - Vamos trabalhar com o conceito de [Query Scopes](#) do **Adonis** que permite criarmos **nossos próprios métodos no construtor de queries**

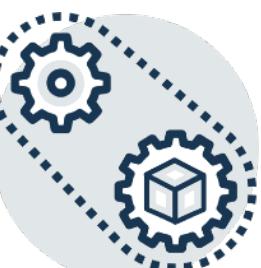


# Query Scope

- Vamos criar nosso **Query Scope** no início da classe app/models/Property.js

```
static scopeNearBy (query, latitude, longitude, distance) {  
  return query;  
}
```

- Precisamos perceber duas coisas aqui:
  - Todo Query Scope é **estático**, ou seja, você não tem acesso ao `this`
  - O nome do método deve **iniciar com scope** e seguir o padrão **Camel Case**
  - O primeiro parâmetro sempre deve ser a `query`, nela temos acesso à construção de `query atual` e podemos adicionar `where's`, `limit`, `order`, etc...



# Ainda na classe Property

- Vamos importar a classe Database do **Adonis** antes da classe já que vamos utilizar um recurso dela logo logo

```
const Database = use('Database')
```

- Com o **Scope** criado, podemos adicionar algumas regras a ele
- O código que acabamos de adicionar utiliza um **cálculo naval de distância através de latitude e longitude** conhecido como [Haversine](#). Esse valor é multiplicado por **6371** que o transforma em **quilômetros**
- O **whereRaw**, que é utilizado quando não estamos utilizando as colunas comuns das tabelas e sim valores gerados por nós

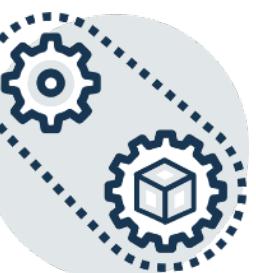
```
static scopeNearBy(query, latitude, longitude, distance) {  
    const haversine = `(6371 * acos(cos(radians(${latitude}))  
        * cos(radians(latitude))  
        * cos(radians(longitude))  
        - radians(${longitude}))  
        + sin(radians(${latitude}))  
        * sin(radians(latitude))))`  
  
    return query  
        .select('*', Database.raw(`${haversine} as distance`))  
        .whereRaw(`${haversine} < ${distance}`)  
}
```



# Configurando controller

- Até agora em nosso PropertyController nós estamos retornando **TODOS** os imóveis para os usuários, mas vamos alterar esse comportamento utilizando essa **Scope Query** que acabamos de criar para filtrar apenas os **imóveis próximos** alterando o **método index**
- O que estamos fazendo aqui é **buscar os dados de latitude e longitude do corpo da nossa requisição** (que serão enviados através do front-end depois) e utilizando nosso método nearBy para buscar apenas imóveis com no **máximo 10km de distância**

```
async index ({ request }) {  
  const { latitude, longitude } = request.all()  
  
  const properties = Property.query()  
    .nearBy(latitude, longitude, 10)  
    .fetch()  
  
  return properties  
}
```

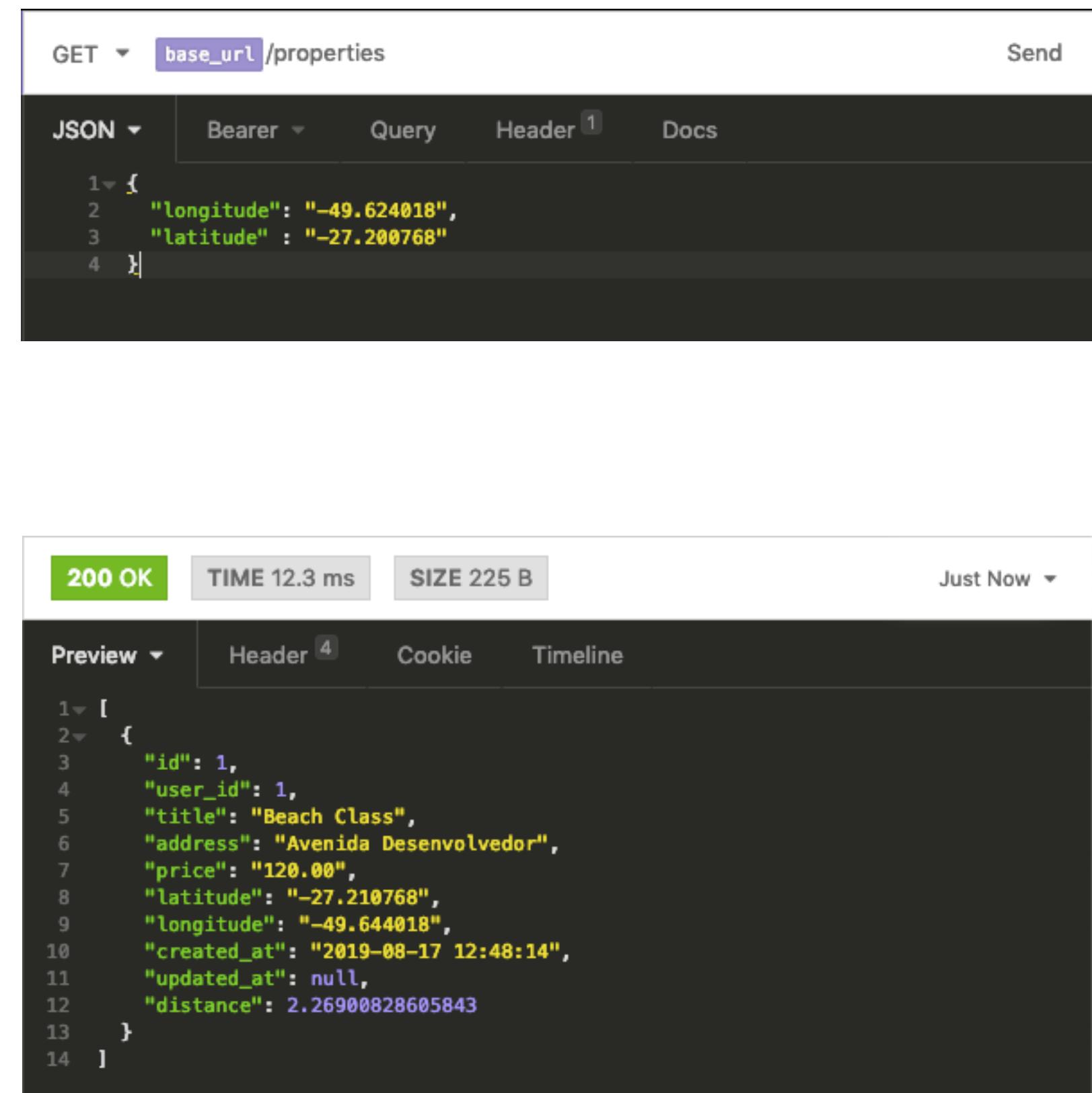


# Testando

- Para testarmos esse comportamento vamos criar um imóvel no banco de dados, para isso, crie um registro no banco de dados com os seguinte dados:

```
"title": "Beach Class"  
"address": "Avenida Desenvolvedor, 260"  
"longitude": -49.644018  
"latitude": -27.210768  
"price": 120
```

- Agora vamos acessar através da API a rota de listagem de imóveis passando uma latitude e longitude com uma pequena distância
- Veja que além dos campos comuns do imóvel temos agora também o campo **distance** e esse registro só retornou porque a distância é **menor que 10km**.

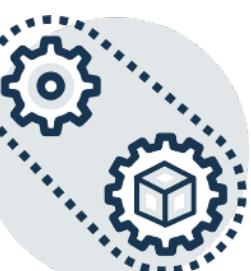


The screenshot shows a REST client interface. At the top, there's a header bar with 'GET' and 'base\_url /properties'. Below it, a 'JSON' tab is selected, showing the following JSON payload:

```
1 {  
2   "longitude": "-49.624018",  
3   "latitude": "-27.200768"  
4 }
```

Below the payload, there are tabs for 'Bearer', 'Query', 'Header 1', and 'Docs'. The main area shows a '200 OK' status with a green button, a 'TIME 12.3 ms' button, and a 'SIZE 225 B' button. The timestamp 'Just Now' is also visible. The 'Preview' tab is selected, displaying the response body:

```
1 [  
2 {  
3   "id": 1,  
4   "user_id": 1,  
5   "title": "Beach Class",  
6   "address": "Avenida Desenvolvedor",  
7   "price": "120.00",  
8   "latitude": "-27.210768",  
9   "longitude": "-49.644018",  
10  "created_at": "2019-08-17 12:48:14",  
11  "updated_at": null,  
12  "distance": 2.26900828605843  
13 }  
14 ]
```



# Criação/edição de imóveis

- Agora, além de criar os métodos de **cadastro** e **edição** de imóveis no controller precisamos permitir que o **usuário envie imagens do imóvel** para **cadastrar na tabela images** que criamos anteriormente
- Buscamos o **ID do usuário logado** através do objeto auth **embutido automaticamente em todos métodos dos controllers**, e também os campos da requisição para criação do imóvel
- Logo após, criamos o imóvel com o método **create** utilizando **todos campos da requisição mais o ID do usuário**

```
async store ({ auth, request, response }) {  
  const { id } = auth.user  
  const data = request.only([  
    'title',  
    'address',  
    'latitude',  
    'longitude',  
    'price'  
  ])  
  
  const property = await Property.create({ ...data, user_id: id })  
  
  return property  
}
```



# Testando no Insomnia

POST [base\\_url /properties](#) Send

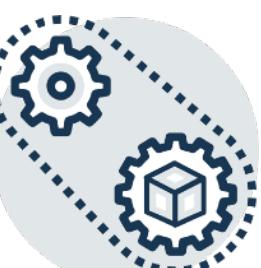
JSON Bearer Query Header 1 Docs

```
1 {  
2   "title": "React House",  
3   "address": "Pertindo do Beach Class",  
4   "price": 220,  
5   "latitude": -27.200368,  
6   "longitude": -49.621018  
7 }
```

200 OK TIME 52.8 ms SIZE 205 B Just Now

Preview Header 4 Cookie Timeline

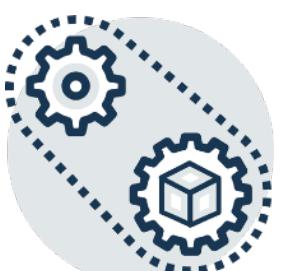
```
1 {  
2   "title": "React House",  
3   "address": "Pertindo do Beach Class",  
4   "latitude": -27.200368,  
5   "longitude": -49.621018,  
6   "price": 220,  
7   "user_id": 1,  
8   "created_at": "2019-08-17 16:14:56",  
9   "updated_at": "2019-08-17 16:14:56",  
10  "id": 2  
11 }
```



# Criação/edição de imóveis

- A principal diferença desse método para o `create` é que precisamos buscar o **ID** que vem na **URL** para buscar um imóvel que já existe e atualiza-lo
- Para isso utilizamos o método `findOneOrFail` que retornará **erro caso o imóvel não exista**
- Logo após, buscamos novamente os dados da requisição e **realizamos um `merge`** no registro **salvando os novos dados**

```
async update ({ params, request, response }) {  
  const property = await Property.findOneOrFail(params.id)  
  
  const data = request.only([  
    'title',  
    'address',  
    'latitude',  
    'longitude',  
    'price'  
  ])  
  
  property.merge(data)  
  
  await property.save()  
  
  return property  
}
```



# Testando no Insomnia

PUT `base_url /properties/2` Send

JSON Bearer Query Header 1 Docs

```
1 {  
2   "title": "React House",  
3   "address": "Pertindo do Beach Class",  
4   "price": 300,  
5   "latitude": -27.200368,  
6   "longitude": -49.621018  
7 }
```

200 OK TIME 627 ms SIZE 205 B Just Now

Preview Header 4 Cookie Timeline

```
1 {  
2   "id": 2,  
3   "user_id": 1,  
4   "title": "React House",  
5   "address": "Pertindo do Beach Class",  
6   "price": 300,  
7   "latitude": -27.200368,  
8   "longitude": -49.621018,  
9   "created_at": "2019-08-17 16:14:56",  
10  "updated_at": "2019-08-17 16:17:56"  
11 }
```



# Upload de imagens no AdonisJs

- No caso dos **uploads para API REST** o fluxo é um pouco diferente do upload comum através de formulário
- Nesse caso, quando fizermos o front-end precisaremos enviar os dados das imagens através do `FormData`, mas como eu não quero ter que enviar todos os dados do imóvel dessa forma, vou separar a parte de upload em outro controller
- Para isso, vamos criar um novo controller chamado `ImageController` com um único método `store`
- Agora, vamos criar **a rota** no arquivo `app/routes.js` para acessar esse método no controller para adicionar novas imagens ao imóvel

```
'use strict'

const Image = use('App/Models/Image')
const Property = use('App/Models/Property')

class ImageController {

    async store ({ request }) {

    }

}

module.exports = ImageController

Route.post('properties/:id/images', 'ImageController.store')
    .middleware('auth')
```

Estamos utilizando uma **rota encadeada**, ou seja, como a imagem **sempre precisa do ID do imóvel associado a ela**, podemos utilizar a rota que já temos de imóveis seguida da palavra `images`



# Upload de imagens no AdonisJs

- Agora no controller de imagens importar a classe `Helpers` do Adonis que vai **nos dar acesso ao caminho da pasta de uploads** chamada `tmp`  
`const Helpers = use('Helpers')`
- E agora no método `store` vamos começar buscando o imóvel pelo **ID recebido na URL** assim como fizemos na edição e buscando também as imagens da requisição
- O `request.file` retorna um ou mais arquivos com o nome do primeiro parâmetro e ainda podemos limitar para arquivos apenas do **tipo de imagem** com **tamanho até 2mb**
- Veja que aqui estou **movendo TODAS imagens** para uma pasta `tmp/uploads` no Adonis e **para cada arquivo estou alterando o nome do mesmo com o timestamp atual** evitando arquivos duplicados.
- Caso aconteça qualquer **erro no upload**, o processo para por aí e nosso front-end fica sabendo dos problemas

```
async store({ params, request }) {
  const property = await Property.findOrFail(params.id)
  /**
   * o request.file que nos trás um ou mais arquivos com o nome do primeiro
   * parâmetro e ainda podemos limitar para arquivos apenas do tipo de imagem
   * com tamanho até 2mb.
   */
  const images = request.file('image', {
    types: ['image'],
    size: '2mb'
  })

  /**
   * aqui estou movendo TODAS imagens para uma pasta tmp/uploads no Adonis e
   * para cada arquivo estou alterando o nome do mesmo com o timestamp atual
   * evitando arquivos duplicados.
   */
  await images.moveAll(Helpers.tmpPath('uploads'), file => {
    name: `${Date.now()}-${file.clientName}`
  })

  if (!images.movedAll()) {
    return images.errors()
  }
}
```



# Criando os registros de imagens

- Agora que já temos os arquivos vamos criar os registros de imagens no banco de dados associados com o imóvel
- Com esse código estamos percorrendo **todas imagens salvas e cadastrando dentro do imóvel**, isso só é possível pois dentro do nosso model de imóvel temos um método `images()` que é o **relacionamento de imóvel com imagens**
- Utilizamos a técnica de `await Promise.all` pois temos uma **iteração (map) assíncrona**, ou seja, cada `create` que **estamos dando retorna uma Promise**, ou seja, pode demorar, com o `await` evitamos que nossa requisição retorne sucesso antes mesmo de terminar o processo

```
/**  
 * estamos percorrendo todas imagens salvas e cadastrando dentro do imóvel,  
 * isso só é possível pois dentro do nosso model de imóvel temos um método  
 * images() que é o relacionamento de imóvel com imagens.  
 */  
await Promise.all(  
  images  
    .movedList()  
    .map(image => property.images().create({ path: image.fileName }))  
)
```



# O método store

```
async store ({ params, request }) {
  const property = await Property.findOrFail(params.id)

  const images = request.file('image', {
    types: ['image'],
    size: '2mb'
  })

  await images.moveAll(Helper.tmpPath('uploads'), file => ({
    name: `${Date.now()}-${file.clientName}`
  }))

  if (!images.movedAll()) {
    return images.errors()
  }

  await Promise.all(
    images
      .movedList()
      .map(image => property.images().create({ path: image.fileName }))
  )
}
```



# Testando no Insomnia

- Para testarmos o **upload** vamos criar **um novo método no Insomnia** e diferente dos outros até agora vamos utilizar o formato do corpo Multipart Form enviando duas imagens
- Enviando a requisição vamos ter apenas uma resposta de sucesso sem corpo já que não retornamos nada
- Nós vamos mostrar as imagens do imóvel no método show e index do imóvel, por isso, no `PropertyController` vamos alterar a query do método `index`
- O `.with` realiza um processo de Eager Loading nas imagens adicionando-as ao retorno de cada imóvel

The screenshot shows the Insomnia interface for a POST request. The URL is `base_url /properties/2/images`. The method is set to `Multipart` with two parts. The header `Bearer` is present. The request body contains two files named `image[0]` and `image[1]`, both of which are screenshots from the presentation.

```
const properties = Property.query()
  .with('images')
  .nearBy(latitude, longitude, 10)
  .fetch()
```

The screenshot shows the response details for a successful `204 No Content` request. The duration was `TIME 572 ms` and the size was `SIZE 0 B`. The preview tab shows the message `No body returned for response`.

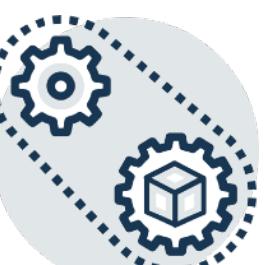


# Testando no Insomnia

200 OK   TIME 569 ms   SIZE 539 B   Just Now ▾

Preview ▾   Header 4   Cookie   Timeline

```
1 {  
2   "id": 2,  
3   "user_id": 1,  
4   "title": "React House",  
5   "address": "Pertindo do Beach Class",  
6   "price": "300.00",  
7   "latitude": "-27.200368",  
8   "longitude": "-49.621018",  
9   "created_at": "2019-08-17 16:14:56",  
10  "updated_at": "2019-08-17 16:17:56",  
11  "images": [  
12    {  
13      "id": 2,  
14      "property_id": 2,  
15      "path": "1566070226998-Screen Shot 2019-08-17 at 16.28.43.png",  
16      "created_at": "2019-08-17 16:30:27",  
17      "updated_at": "2019-08-17 16:30:27"  
18    },  
19    {  
20      "id": 1,  
21      "property_id": 2,  
22      "path": "1566070226998-Screen Shot 2019-08-17 at 16.28.31.png",  
23      "created_at": "2019-08-17 16:30:27",  
24      "updated_at": "2019-08-17 16:30:27"  
25    }  
26  ]  
27 }
```



# Criando um “campo virtual”

- O último ponto que vamos fazer é **criar um campo “virtual”** no nosso model de imagem para **retornar o caminho completo da imagem** para nosso front-end
- Criamos um campo “computed” chamado **url** e adicionamos seu valor com um método com **prefixo get seguido do campo em camel case**
- Utilizamos também o Env para recuperar a **URL da nossa API**
- Se tentarmos acessar esse link ainda não temos uma rota para exibir a imagem, por isso vamos criar uma nova rota que apenas mostra a imagem

```
Route.get('images/:path',  
  'ImageController.show')
```

```
'use strict'  
  
const Env = use('Env')  
const Model = use('Model')  
  
class Image extends Model {  
  static get computed () {  
    return ['url']  
  }  
  
  getUrl ({ path }) {  
    return `${Env.get('APP_URL')}/images/${path}`  
  }  
}  
  
module.exports = Image
```

# Rota das imagens

- Agora no ImageController adicione um método show com o código:

```
'use strict'

const Helpers = use('Helpers')
const Image = use('App/Models/Image')
const Property = use('App/Models/Property')

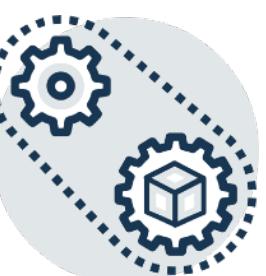
class ImageController {

    async show({ params, response }) {
        return response.download(Helpers.tmpPath(`uploads/${params.path}`))
    }
}
```

200 OK   TIME 521 ms   SIZE 719 B   Just Now ▾

Preview Header 4 Cookie Timeline

```
1 {  
2     "id": 2,  
3     "user_id": 1,  
4     "title": "React House",  
5     "address": "Pertindo do Beach Class",  
6     "price": "300.00",  
7     "latitude": "-27.200368",  
8     "longitude": "-49.621018",  
9     "created_at": "2019-08-17 16:14:56",  
10    "updated_at": "2019-08-17 16:17:56",  
11    "images": [  
12        {  
13            "id": 2,  
14            "property_id": 2,  
15            "path": "1566070226998-Screen Shot 2019-08-17 at 16.28.43.png",  
16            "created_at": "2019-08-17 16:30:27",  
17            "updated_at": "2019-08-17 16:30:27",  
18            "url": "http://127.0.0.1:3333/images/1566070226998-Screen Shot 2019-08-17 at  
16.28.43.png"  
19        },  
20        {  
21            "id": 1,  
22            "property_id": 2,  
23            "path": "1566070226998-Screen Shot 2019-08-17 at 16.28.31.png",  
24            "created_at": "2019-08-17 16:30:27",  
25            "updated_at": "2019-08-17 16:30:27",  
26            "url": "http://127.0.0.1:3333/images/1566070226998-Screen Shot 2019-08-17 at  
16.28.31.png"  
27        }  
28    ]  
29 }
```



# Temos uma API!

- Terminamos a API com Adonis e agora podemos continuar essa aplicação no front-end consumindo essa API e criando todo seu funcionamento com ReactJS, Vue.js, Angular, React Native para web/mobile...
- Código dessa sprint
  - <https://github.com/vinicius3w/easylanding-server/tree/v0.3-alpha>

