INSTITUTO FEDERAL
Paraná
Campus Paranaguá

Ministério da Educação

TECNÓLOGO EM ANÁLISE E DESENVOLVIMENTO
Disciplina de Desenvolvimento Web
Aula 06: Adonis JS - Migration / Seeder / Lucid ORM

Gil Eduardo de Andrade

Material TypeScrip - Básico
Material Docker (Adonis + MySQL + PhpMyAdmin)
Passo a Passo - Aplicação Aula

Conceitos Preliminares

(https://adonisjs.com/)
(https://lucid.adonisjs.com/docs/migrations)
(https://lucid.adonisjs.com/docs/seeders)
(https://lucid.adonisjs.com/docs/crud-operations)
(https://lucid.adonisjs.com/docs/relationships)
(https://lucid.adonisjs.com/docs/model-query-builder)

Migration

As *"migrations"* permitem controlar a versão da base de dados que está sendo utilizada pela aplicação. Elas podem ser vistas como scripts independentes escritos em TypeScript para criar e alterar o esquema da base de dados ao longo do tempo.

Com funcionam as *Migrations*

As *migrations* funcionam da seguinte forma:

1. Um novo arquivo de *migration* é criado para cada alteração no esquema do banco de dados (criar ou alterar tabela);
2. Dentro do arquivo de *migration* são escritas as instruções que permitem realizar as ações necessárias;
3. As *migrations* são executadas usando a ferramenta de linha de comando do AdonisJS (ace);
4. O AdonisJS mantém o controle das migrations executadas, garantindo que cada migration seja executada apenas uma vez;
5. Durante o desenvolvimento, você também pode reverter migrations para editá-las.

Criando uma *Migration*

Uma nova *migration* pode ser criada através da execução do seguinte comando Ace.

```shell
node ace make:migration users

# CREATE: database/migrations/1630981615472_create_users_table.ts
```

Também é possível criar um modelo Lucid, juntamente com a *migration*, executando o comando com a flag *"-m"*

```shell
node ace make:model User -m
```

Estrutura da classe *Migration*

Uma classe de *migration* sempre estende a classe *"BaseSchema"* e deve implementar os métodos ***up*** e ***down***.

- O método ***up*** é usado para evoluir o esquema do banco de dados. Geralmente, você criará novas tabelas ou alterará tabelas existentes dentro deste método;

- O método ***down*** é usado para reverter as ações executadas pelo método ***up***. Por exemplo, se o método ***up*** cria uma tabela, o método ***down*** deve excluir a mesma tabela.

```typescript
import { BaseSchema } from '@adonisjs/lucid/schema'

export default class extends BaseSchema {
      protected tableName = 'users'

      async up() {
            this.schema.createTable(this.tableName, (table) => {
                  table.increments('id')
                  table.string('username').unique()
```

```
            table.string('email').unique()
            table.string('password')
            table.timestamp('created_at', { useTz: true })
            table.timestamp('updated_at', { useTz: true })
        })
    }

    async down() {
        this.schema.dropTable(this.tableName)
    }
}
```

Executando e Revertendo Migrations
Para executar as migrations, use o comando:

Shell

```
node ace migration:run
```

Para reverter migrations:

Shell

```
# Reverter o último lote
node ace migration:rollback

# Reverter até o início das migrations
node ace migration:rollback --batch=0

# Reverter até o lote 1
node ace migration:rollback --batch=1

# Reverter as últimas 3 migrations
node ace migration:rollback --step=3
```

Comandos úteis de Migration

```shell
# Reverter todas as migrations e executar novamente
node ace migration:refresh

# Reverter, migrar e executar seeders
node ace migration:refresh --seed

# Excluir todas as tabelas e migrar novamente
node ace migration:fresh

# Excluir todas as tabelas, migrar e executar seeders
node ace migration:fresh --seed

# Resetar todas as migrations
node ace migration:reset
```

Criando Tabelas

```typescript
import { BaseSchema } from '@adonisjs/lucid/schema'

export default class extends BaseSchema {
  protected tableName = 'posts'

  async up() {
    this.schema.createTable(this.tableName, (table) => {
      table.increments('id')
      table.string('title')
      table.text('content')
      table.string('status').defaultTo('draft')

      // Relacionamento (chave estrangeira)
      table.integer('user_id').unsigned().references
      ('id').inTable('users')
```

```
                table.timestamp('published_at', { useTz: true
                }).nullable()
                table.timestamp('created_at', { useTz: true })
                table.timestamp('updated_at', { useTz: true })
        })
    }

    async down() {
        this.schema.dropTable(this.tableName)
    }
}
```

Alterando Tabelas

TypeScript

```
export default class extends BaseSchema {
    protected tableName = 'users'

    async up() {
        this.schema.alterTable(this.tableName, (table) => {
            table.string('phone').nullable()
            table.date('birth_date').nullable()
            table.dropColumn('old_column')
        })
    }

    async down() {
        this.schema.alterTable(this.tableName, (table) => {
            table.dropColumn('phone')
            table.dropColumn('birth_date')
            table.string('old_column')
        })
    }
}
```

Renomeando Tabelas

```typescript
export default class extends BaseSchema {

      async up() {
            this.schema.renameTable('old_table_name',
      'new_table_name')
      }

      async down() {
            this.schema.renameTable('new_table_name',
      'old_table_name')
      }
}
```

Seeders (Povoamento de Dados)

Database seeding é uma forma de configurar sua aplicação com alguns dados iniciais necessários para executar e usar a aplicação. Os seeders são úteis para:

- Criar dados iniciais como países, estados e cidades antes de implantar e executar sua aplicação;
- Inserir usuários no banco de dados para desenvolvimento local;
- Criar dados de teste para desenvolvimento.

Criando um Seeder

Os seeders são armazenados no diretório database/seeders, e podem ser criados através da execução do seguinte comando Ace:

```shell
node ace make:seeder User

# CREATE: database/seeders/user_seeder.ts
```

Estrutura do Seeder

Todo arquivo seeder deve estender a classe *"BaseSeeder"* e implementar o método run.

```typescript
import { BaseSeeder } from '@adonisjs/lucid/seeders'
import User from '#models/user'

export default class UserSeeder extends BaseSeeder {
      async run() {
            await User.createMany([
                  {
                        username: 'admin',
                        email: 'admin@example.com',
                        password: 'secret',
                  },
                  {
                        username: 'user1',
                        email: 'user1@example.com',
                        password: 'secret',
                  },
            ])
      }
}
```

Executando Seeders

Para executar todos ou seeders selecionados utiliza-se o seguinte comando Ace:

```shell
# Executa todos os seeders
node ace db:seed

# Executa um seeder específico
node ace db:seed --files "./database/seeders/user_seeder.ts"

# Executa seeders de forma interativa
node ace db:seed -i
```

Seeder com Relacionamentos

```typescript
import { BaseSeeder } from '@adonisjs/lucid/seeders'
import User from '#models/user'
import Post from '#models/post'

export default class PostSeeder extends BaseSeeder {
      async run() {
            // Buscar usuários existentes
            const users = await User.all()

            for (const user of users) {
                  // Criar posts para cada usuário
                  await user.related('posts').createMany([
                        {
                              title: `Post 1 do ${user.username}`,
                              content: 'Conteúdo do primeiro post',
                              status: 'published',
                        },
                        {
                              title: `Post 2 do ${user.username}`,
                              content: 'Conteúdo do segundo post',
                              status: 'draft',
                        },
                  ])
            }
      }
}
```

Seeder com Factory (se estiver usando Model Factories)

```typescript
import { BaseSeeder } from '@adonisjs/lucid/seeders'
import User from '#models/user'

export default class UserSeeder extends BaseSeeder {
      async run() {
            // Criar 10 usuários usando factory
            await User.factory().count(10).create()
      }
```

```
}
```

Models (Modelos)

Os modelos de dados do Lucid são construídos sobre o padrão ***Active Record (*)*** e facilitam muito a realização de operações CRUD e gerenciamento de relacionamentos entre modelos.

(*) Active Record é um padrão de projeto de ORM (Object-Relational Mapping) em que uma classe de objeto representa uma única linha (ou registo) de uma tabela de banco de dados, permitindo que os desenvolvedores manipulem a base de dados de forma orientada a objetos.

Criando um *Model*

É possível criar um modelo Lucid usando o comando Ace *"make:model"*:

Shell

```
node ace make:model User

# CREATE: app/Models/User.ts
```

Também é possível gerar uma *migration* vinculada com o modelo, utilizando a flag *"-m"*:

Shell

```
node ace make:model User -m

# CREATE: database/migrations/1618903673925_users.ts
# CREATE: app/Models/User.ts
```

Também é possível criar uma *factory* vinculada com o modelo, utilizando a flag *"-f"*:

Shell

```
node ace make:model User -f

# CREATE: app/Models/User.ts
# CREATE: database/factories/User.ts
```

Estrutura Básica de um *Model*

Todo modelo deve estender a classe *"BaseModel"* para herdar funcionalidades adicionais:

TypeScript

```
import { DateTime } from 'luxon'
import { BaseModel, column } from '@adonisjs/lucid/orm'

export default class User extends BaseModel {
      @column({ isPrimary: true })
      declare id: number

      @column()
      declare username: string

      @column()
      declare email: string

      @column({ serializeAs: null })
      declare password: string

      @column()
      declare avatarUrl: string | null

      @column.dateTime({ autoCreate: true })
      declare createdAt: DateTime

      @column.dateTime({ autoCreate: true, autoUpdate: true })
      declare updatedAt: DateTime
      }
}
```

Definindo Colunas

As colunas do banco de dados são definidas como propriedades na classe, e são decoradas pelo uso do *decorator* ***@column***.

Opções do Decorator @column

```typescript
export default class User extends BaseModel {
	// Chave primária
	@column({ isPrimary: true })
	declare id: number

	// Coluna normal
	@column()
	declare username: string

	// Coluna que não aparece na serialização
	@column({ serializeAs: null })
	declare password: string

	// Coluna com nome personalizado no banco
	@column({ columnName: 'user_email' })
	declare email: string

	// Coluna com preparação e consumo de dados
	@column({
		prepare: (value: string) => value.toLowerCase(),
		consume: (value: string) => value.toUpperCase(),
	})

	declare status: string
}
```

Colunas de Data

O Lucid permite aprimorar as propriedades de data e data-hora, convertendo os valores do driver do banco de dados para uma instância de "*luxon.DateTime*":

```typescript
import { DateTime } from 'luxon'
import { BaseModel, column } from '@adonisjs/lucid/orm'

export default class User extends BaseModel {
      @column.date()
      declare birthDate: DateTime

      @column.dateTime({ autoCreate: true })
      declare createdAt: DateTime

      @column.dateTime({ autoCreate: true, autoUpdate: true })
      declare updatedAt: DateTime

      // Para usar Date nativo ao invés de Luxon
      @column({
            consume: (value: string) => new Date(value),
            prepare: (value: Date) => value.toISOString(),
      })

      declare lastLogin: Date
}
```

Configurações do Model

Nome da Tabela Personalizado

```typescript
export default class User extends BaseModel {
      static table = 'app_users'
}
```

Chave Primária Personalizada

```typescript
export default class User extends BaseModel {
```

```
      static primaryKey = 'uuid'

      @column({ isPrimary: true })
      declare uuid: string
}
```

Auto-atribuição de Chave Primária

TypeScript
```
import { randomUUID } from 'node:crypto'
import { BaseModel, beforeCreate, column } from '@adonisjs/lucid/orm'
export default class User extends BaseModel {

      static selfAssignPrimaryKey = true

      @column({ isPrimary: true })
      declare id: string

      @beforeCreate()
      static assignUuid(user: User) {
            user.id = randomUUID()
      }
}
```

Hooks de Ciclo de Vida

Os hooks permitem executar código em pontos específicos do ciclo de vida do modelo:

TypeScript
```
import {
      BaseModel,
      beforeSave,
      afterCreate,
      beforeCreate,
      afterUpdate
```

```
}
from '@adonisjs/lucid/orm'
import Hash from '@adonisjs/core/services/hash'

export default class User extends BaseModel {

      @column({ serializeAs: null })
      declare password: string

      // Executado antes de salvar (criar ou atualizar)
      @beforeSave()
      static async hashPassword(user: User) {
            if (user.$dirty.password) {
                  user.password = await Hash.make(user.password)
            }
      }

      // Executado antes de criar
      @beforeCreate()
            static async setDefaults(user: User) {
            user.status = user.status || 'active'
      }

      // Executado após criar
      @afterCreate()
      static async sendWelcomeEmail(user: User) {
            // Enviar email de boas-vindas
            console.log(`Enviando email de boas-vindas para ${user.email}`)
      }

      // Executado após atualizar
      @afterUpdate()
            static async logUpdate(user: User) {
            console.log(`Usuário ${user.id} foi atualizado`)
      }
}
```

Hooks Disponíveis

TypeScript

```typescript
import {
	BaseModel,
	beforeSave,
	afterSave,
	beforeCreate,
	afterCreate,
	beforeUpdate,
	afterUpdate,
	beforeDelete,
	afterDelete,
	beforeFind,
	afterFind,
	beforeFetch,
	afterFetch
} from '@adonisjs/lucid/orm'

export default class User extends BaseModel {

	@beforeSave()
	static async beforeSaveHook(user: User) {
		// Executado antes de salvar (criar ou atualizar)
	}

	@afterSave()
	static async afterSaveHook(user: User) {
		// Executado após salvar (criar ou atualizar)
	}

	@beforeCreate()
	static async beforeCreateHook(user: User) {
		// Executado antes de criar
	}

	@afterCreate()
	static async afterCreateHook(user: User) {
		// Executado após criar
```

```
    }

    @beforeUpdate()
    static async beforeUpdateHook(user: User) {
        // Executado antes de atualizar
    }

    @afterUpdate()
    static async afterUpdateHook(user: User) {
        // Executado após atualizar
    }

    @beforeDelete()
    static async beforeDeleteHook(user: User) {
        // Executado antes de deletar
    }

    @afterDelete()
    static async afterDeleteHook(user: User) {
        // Executado após deletar
    }
}
```

Model Factories (Fábrica de Modelos)

As *Model Factories* são uma ferramenta poderosa para gerar dados falsos em massa, especialmente úteis durante testes, possibilitando povoar o banco de dados com dados aleatórios. Com as *factories*, é possível extrair toda a configuração de dados de teste para um arquivo dedicado e escrever o mínimo de código necessário para configurar o estado do banco de dados.

Criando Factories

As *Model Factories* são armazenadas no diretório database/factories. Você pode definir todas as *factories* em um único arquivo ou criar arquivos dedicados para cada modelo.

```shell
node ace make:factory User

# CREATE: database/factories/user.ts
```

Também é possível criar uma *factory* vinculada ao modelo, usando a flag *"-f"*:

```shell
node ace make:model User -f

# CREATE: app/Models/User.tsc
# CREATE: database/factories/user.ts
```

Estrutura Básica de uma Factory

```typescript
import User from '#models/user'
import Factory from '@adonisjs/lucid/factories'

export const UserFactory = Factory.define(User, ({ faker }) => {
      return {
            username: faker.internet.userName(),
            email: faker.internet.email(),
            password: faker.internet.password(),
            firstName: faker.person.firstName(),
            lastName: faker.person.lastName(),
            birthDate: faker.date.birthdate(),
      }
}).build()
```

Usando Factories

Criar um único registro:

```typescript
import { UserFactory } from '#database/factories/user'

const user = await UserFactory.create()
console.log(user.email) // email gerado pelo faker
```

Criar múltiplos registros

```typescript
const users = await UserFactory.createMany(10)
console.log(users.length) // 10
```

Factory States (Estados)

Os states permitem definir variações das suas factories. Por exemplo, em uma *factory* de *"Post"*, é possível ter diferentes states para representar posts publicados e rascunhos.

```typescript
import Post from '#models/post'
import Factory from '@adonisjs/lucid/factories'

export const PostFactory = Factory.define(Post, ({ faker }) => {
      return {
            title: faker.lorem.sentence(),
            content: faker.lorem.paragraphs(4),
            status: 'draft',
            viewCount: 0,
            }
})
.state('published', (post) => {
      post.status = 'published'
      post.publishedAt = new Date()
})
.state('popular', (post) => {
      post.viewCount = faker.number.int({ min: 1000, max: 10000})
})
```

```
.build()
```

Usando States

```typescript
// Criar posts com state padrão (draft)
const draftPosts = await PostFactory.createMany(3)

// Criar posts publicados (published)
const  publishedPosts = await
      PostFactory.apply('published').createMany(3)

// Criar posts populares e publicados
const popularPosts = await PostFactory
      .apply('published')
      .apply('popular')
      .createMany(2)
```

Relacionamentos com Factories

As factories facilitam o trabalho com relacionamentos entre modelos.

```typescript
import Post from '#models/post'
import Comment from '#models/comment'
import Factory from '@adonisjs/lucid/factories'

export const CommentFactory = Factory.define(Comment, ({ faker }) => {
      return {
            content: faker.lorem.paragraph(),
            authorName: faker.person.fullName(),
      }
}).build()

export const PostFactory = Factory.define(Post, ({ faker }) => {
      return {
```

```
            title: faker.lorem.sentence(),
            content: faker.lorem.paragraphs(4),
            status: 'draft',
      }
})
.relation('comments', () => CommentFactory)
.build()

export const UserFactory = Factory.define(User, ({ faker }) => {
      return {
            username: faker.internet.userName(),
            email: faker.internet.email(),
            password: faker.internet.password(),
      }
})
.relation('posts', () => PostFactory)
.relation('profile', () => ProfileFactory)
.build()
```

Criando Modelos com Relacionamentos

TypeScript

```
// Criar usuário com 3 posts
const user = await UserFactory.with('posts', 3).create()
console.log(user.posts.length) // 3

// Criar usuário com posts e comentários
const user = await UserFactory.with(
      'posts', 2, (post) => post.with('comments', 5)
)
.create()

// Criar usuário com posts publicados
const user = await UserFactory.with(
      'posts', 3, (post) => post.apply('published')
)
```

```
.create()

// Criar usuário com posts mistos
const user = await UserFactory
.with('posts', 2, (post) => post.apply('published'))
.with('posts', 3) // posts em draft
.create()
```

Lucid ORM

O Lucid é um ORM (*Object-Relational Mapper*) para Node.js, construído sobre o Knex.js. Ele faz parte do framework AdonisJS, mas pode ser usado de forma independente. Ele implementa o padrão *Active Record*, que facilita a interação com o banco de dados de forma orientada a objetos.
Esta aula tem como objetivo apresentar os métodos básicos de CRUD (Create, Read, Update, Delete) e os métodos de relacionamento.

Métodos Básicos (CRUD)

Os métodos de CRUD (Create, Read, Update, Delete) são a base de qualquer ORM. O Lucid oferece uma API intuitiva para realizar essas operações.

Create (Criar)

Para criar novos registros no banco de dados, podem ser usados os métodos *"create"*, *"createMany"*, ou instanciar um novo modelo e usar o método *"save"*.

create(data)
Cria e persiste um novo registro no banco de dados.

JavaScript
```
import User from '#models/user'

const user = await User.create({
      username: 'virk',
      email: 'virk@adonisjs.com',
})
```

createMany(data)
Cria e persiste múltiplos registros no banco de dados.

```JavaScript
const users = await User.createMany([
      {
            email: 'virk@adonisjs.com',
            password: 'secret',
      },
      {
            email: 'romain@adonisjs.com',
            password: 'secret',
      },
])
```

new Model() + save()
Cria uma nova instância do modelo, atribui os valores e então persiste no banco de dados.

```JavaScript
import User from '#models/user'

const user = new User()
user.username = 'virk'
user.email = 'virk@adonisjs.com'
await user.save()
```

Read (Ler)
O Lucid oferece vários métodos para buscar dados do banco de dados.

all()
Busca todos os registros da tabela.

```JavaScript
const users = await User.all()
```

find(id)
Busca um registro pela sua chave primária.

```javascript
const user = await User.find(1)
```

findBy(column, value)
Busca um registro por uma coluna específica.

```javascript
const user = await User.findBy('email', 'virk@adonisjs.com')
```

findMany(ids)
Busca múltiplos registros por um array de chaves primárias.

```javascript
const users = await User.findMany([1, 2])
```

first()
Busca o primeiro registro da tabela.

```javascript
const user = await User.first()
```

findOrFail(id) , firstOrFail() , findByOrFail(column, value)
As variações apresentadas acima, para os métodos de busca, lançam uma exceção E_ROW_NOT_FOUND se nenhum registro for encontrado.

```javascript
const user = await User.findOrFail(1)
```

Update (Atualizar)
Para atualizar um registro, é necessário, primeiro, buscá-lo no banco de dados. Após essa etapa, os valores obtidos podem ser alterados e o método *"save"* utilizado para consolidar a mudança na base de dados.

```javascript
const user = await User.findOrFail(1)
user.username = 'lucid'
await user.save()
```

Também é possível usar o método *"merge"* para atribuir múltiplos valores de uma vez.

```javascript
const user = await User.findOrFail(1)
await user.merge({ username: 'lucid' }).save()
```

Delete (Deletar)
Para deletar um registro, é necessário, primeiro, buscálo na base de dados, para então usar o método *"delete"*.

```javascript
const user = await User.findOrFail(1)
await user.delete()
```

Para deletar múltiplos registros, é possível usar o query builder.

```javascript
await User.query().where('isVerified', false).delete()
```

Métodos de Relacionamento

O Lucid ORM oferece um poderoso sistema de relacionamentos para definir e consultar dados relacionados.

HasOne (Um para Um)

Define um relacionamento ***um-para-um***. Por exemplo, um *"User"* tem um *"Profile"*.

```typescript
// app/models/User.ts

import Profile from '#models/profile'
import type { HasOne } from '@adonisjs/lucid/types/relations'
import { BaseModel, column, hasOne } from '@adonisjs/lucid/orm'

export default class User extends BaseModel {
      @column({ isPrimary: true })
      declare id: number

      @hasOne(() => Profile)
      declare profile: HasOne<typeof Profile>
}
```

HasMany (Um para Muitos)

Define um relacionamento ***um-para-muitos***. Por exemplo, um *"User"* tem muitos *"Posts"*.

```typescript
// app/models/User.ts

import Post from '#models/post'
import type { HasMany } from '@adonisjs/lucid/types/relations'
import { BaseModel, column, hasMany } from '@adonisjs/lucid/orm'

export default class User extends BaseModel {
      @column({ isPrimary: true })
      declare id: number

      @hasMany(() => Post)
      declare posts: HasMany<typeof Post>
}
```

BelongsTo (Pertence a)

Define o inverso de um relacionamento hasOne ou hasMany. Por exemplo, um *"Post"* pertence a um *"User"*.

```typescript
// app/models/Post.ts

import User from '#models/user'
import type { BelongsTo } from '@adonisjs/lucid/types/relations'
import { BaseModel, column, belongsTo } from '@adonisjs/lucid/orm'

export default class Post extends BaseModel {
      @column({ isPrimary: true })
      declare id: number

      @column()
      declare userId: number

      @belongsTo(() => User)
      declare user: BelongsTo<typeof User>
}
```

ManyToMany (Muitos para Muitos)

Define um relacionamento ***muitos-para-muitos***. Por exemplo, um *"User"* pode ter muitas *"Skills"*, e uma *"Skill"* pode pertencer a muitos *"Users"*. Este relacionamento requer uma tabela *"pivot"*.

```typescript
// app/models/User.ts

import Skill from '#models/skill'
import type { ManyToMany } from '@adonisjs/lucid/types/relations'
import { BaseModel, column, manyToMany } from '@adonisjs/lucid/orm'

export default class User extends BaseModel {
      @column({ isPrimary: true })
```

```
    declare id: number

    @manyToMany(() => Skill)
    declare skills: ManyToMany<typeof Skill>
}
```

Consultando Relacionamentos

O Lucid oferece métodos para carregar e filtrar relacionamentos de forma eficiente.

preload(relationship)

Carrega os dados de um relacionamento junto com a consulta principal.

```typescript
const users = await User.query().preload('posts')
users.forEach(user => {
    console.log(user.posts) // Array de posts do usuário
})
```

Também é possível adicionar *"constraints"* (restrições) à consulta do relacionamento.

```typescript
const users = await User.query().preload('posts', (postsQuery) => {
    postsQuery.where('status', 'published')
})
```

whereHas(relationship, callback)

Filtra os resultados da consulta principal com base na existência de um relacionamento que satisfaça uma determinada condição.

```typescript
// Busca usuários que têm posts publicados
```

```
const users = await User.query().whereHas('posts', (postsQuery) => {
      postsQuery.where('status', 'published')
})
```

has(relationship)
Filtra os resultados da consulta principal com base na existência de um relacionamento.

```
TypeScript
// Busca usuários que têm pelo menos um post
const users = await User.query().has('posts')

// Busca usuários que têm pelo menos 2 posts
const users = await User.query().has('posts', '>=', 2)
```

Métodos Idempotentes

O Lucid oferece métodos que simplificam a criação de registros verificando primeiro se eles já existem.

firstOrCreate(searchPayload, savePayload)
Busca um registro no banco de dados ou cria um novo se não encontrar.

```
TypeScript
const user = await User.firstOrCreate(
      { email: 'virk@adonisjs.com' },
      { password: 'secret' }
)
```

updateOrCreate(searchPayload, updatePayload)
Busca um registro e o atualiza, ou cria um novo se não encontrar.

```typescript
const user = await User.updateOrCreate(
      { email: 'virk@adonisjs.com' },
      { lastLoginAt: DateTime.local() }
)
```

Query Builder Avançado

O Lucid oferece um query builder poderoso que permite construir consultas SQL complexas.

```typescript
const users = await User
      .query()
      .where('countryCode', 'BR')
      .orWhereNull('countryCode')
      .orderBy('createdAt', 'desc')
      .limit(10)
```

Paginação

O Lucid oferece suporte nativo à paginação.

```typescript
const users = await User.query().paginate(1, 20)

console.log(users.total) // Total de registros
console.log(users.perPage) // Registros por página
console.log(users.currentPage) // Página atual
```

Transações

Para operações que envolvem múltiplas consultas, é possível utilizar transações.

TypeScript

```typescript
import Database from '@adonisjs/lucid/services/db'

const trx = await Database.transaction()
try {
      const user = await User.create({
            email: 'virk@adonisjs.com'
      }, { client: trx })

      await user.related('profile').create({
            fullName: 'Harminder Virk'
      }, { client: trx })

      await trx.commit()
} catch(error) {
      await trx.rollback()
      throw error
}
```