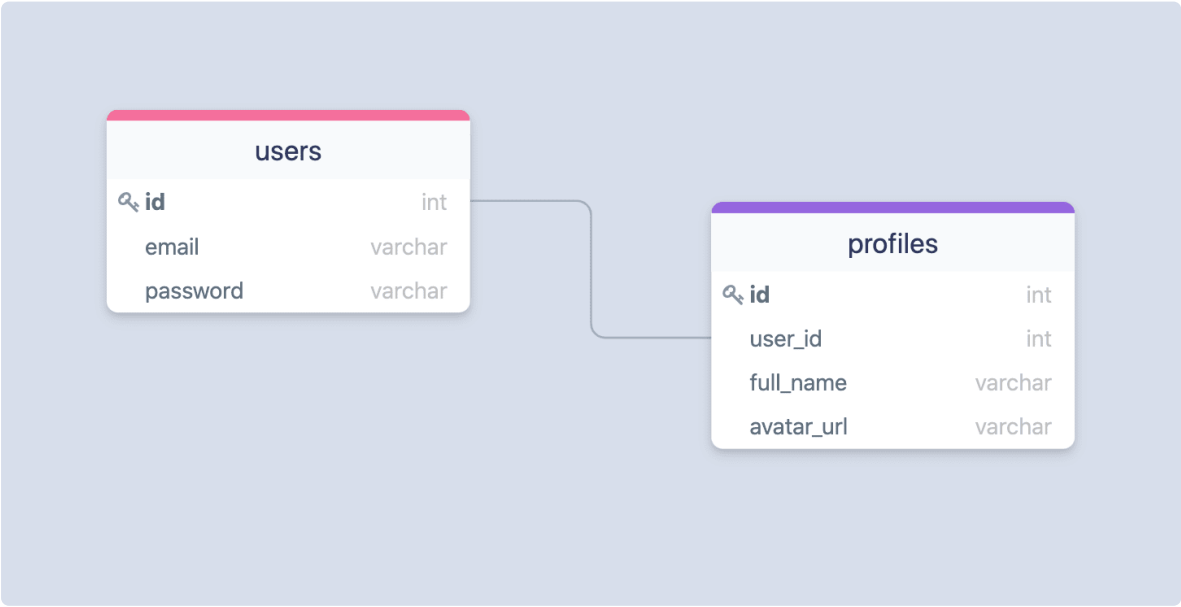
Relacionamentos

Os modelos de dados Lucid têm suporte pronto para trabalhar com relacionamentos. Você precisa definir os relacionamentos em seus modelos, e o Lucid fará todo o trabalho pesado de construção das consultas SQL subjacentes.

Tem um

HasOne cria um `one-to-one` relacionamento entre dois modelos. Por exemplo, **um usuário tem um perfil** . O relacionamento que tem um precisa de uma chave estrangeira na tabela relacionada.

A seguir está um exemplo de estrutura de tabela para o relacionamento. A `profiles.user_id` é a chave estrangeira e forma o relacionamento com a `users.id` coluna.



Usuários perfis

```
import BaseSchema from '@ioc:Adonis/Lucid/Schema'

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

  public async up () {
    this.schema.createTable(this.tableName, (table) => {
      table.increments('id').primary()
      table.timestamp('created_at', { useTz: true })
      table.timestamp('updated_at', { useTz: true })
    })
  }
}
```

Definindo relacionamento no modelo

Depois de criar as tabelas com as colunas obrigatórias, você também terá que definir o relacionamento no modelo Lucid.

O relacionamento has one é definido usando o decorador **@hasOne** em uma propriedade de modelo.

```
import {
  column,
  BaseModel,
  hasOne,
  HasOne
} from '@ioc:Adonis/Lucid/Orm'

export default class User extends BaseModel {
  @hasOne(() => Profile)
  public profile: HasOne<typeof Profile>
}
```

Chaves de relacionamento personalizadas

Por padrão, `foreignKey` é a **representação camelCase do nome do modelo pai e sua chave primária** . No entanto, você também pode definir uma chave estrangeira personalizada.

```
@hasOne(() => Profile, {
  foreignKey: 'profileUserId', // defaults to userId
})
public profile: HasOne<typeof Profile>
```

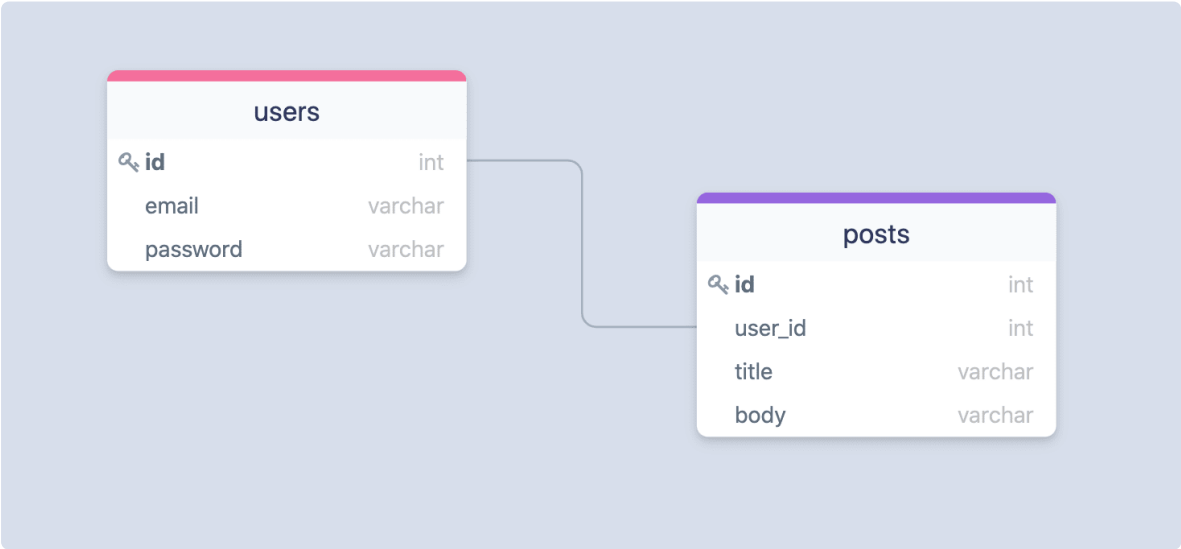
Lembre-se, se você pretende usar camelCase para definição de chave estrangeira, lembre-se de que a estratégia de nomenclatura padrão irá convertê-la automaticamente para Snake_case.

```
@hasOne(() => Profile, {
  localKey: 'uuid', // defaults to id
})
public profile: HasOne<typeof Profile>
```

Tem muitos

HasMany cria um `one-to-many` relacionamento entre dois modelos. Por exemplo, **um usuário tem muitas postagens** . O relacionamento hasMany precisa de uma chave estrangeira na tabela relacionada.

A seguir está um exemplo de estrutura de tabela para o relacionamento hasMany. A `posts.user_id` é a chave estrangeira e forma o relacionamento com a `users.id` coluna.



A seguir estão os exemplos de migrações para as tabelas `users` e `posts` .

```
import BaseSchema from '@ioc:Adonis/Lucid/Schema'

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

  public async up () {
    this.schema.createTable(this.tableName, (table) => {
      table.increments('id').primary()
      table.timestamp('created_at', { useTz: true })
      table.timestamp('updated_at', { useTz: true })
    })
  }
}
```

Definindo relacionamento no modelo

Depois de criar as tabelas com as colunas obrigatórias, você também terá que definir o relacionamento no modelo Lucid.

O relacionamento has many é definido usando o decorador **@hasMany** em uma propriedade de modelo.

```
import {
  column,
  BaseModel,
  hasMany,
  HasMany
} from '@ioc:Adonis/Lucid/Orm'

export default class User extends BaseModel {
  @hasMany(() => Post)
  public posts: HasMany<typeof Post>
}
```

Chaves de relacionamento personalizadas

Por padrão, `foreignKey` é a **representação camelCase do nome do modelo pai e sua chave primária** . No entanto, você também pode definir uma chave estrangeira personalizada.

```
@hasMany(() => Post, {
  foreignKey: 'authorId', // defaults to userId
})
public posts: HasMany<typeof Post>
```

Lembre-se, se você pretende usar camelCase para definição de chave estrangeira, lembre-se de que a estratégia de nomenclatura padrão irá convertê-la automaticamente para Snake_case.

```
@hasMany(() => Post, {
  localKey: 'uuid', // defaults to id
})
public posts: HasMany<typeof Post>
```

Pertence a

BelongsTo é o inverso do `hasOne` e do `hasMany` relacionamento. Assim, por exemplo, **o perfil pertence a um usuário** e **uma postagem pertence a um usuário** .

Você pode aproveitar a mesma estrutura de tabela e as mesmas convenções de chave estrangeira para definir um relacionamento pertence a.

O relacionamento pertence a é definido usando o decorador **@belongsTo** em uma propriedade de modelo.

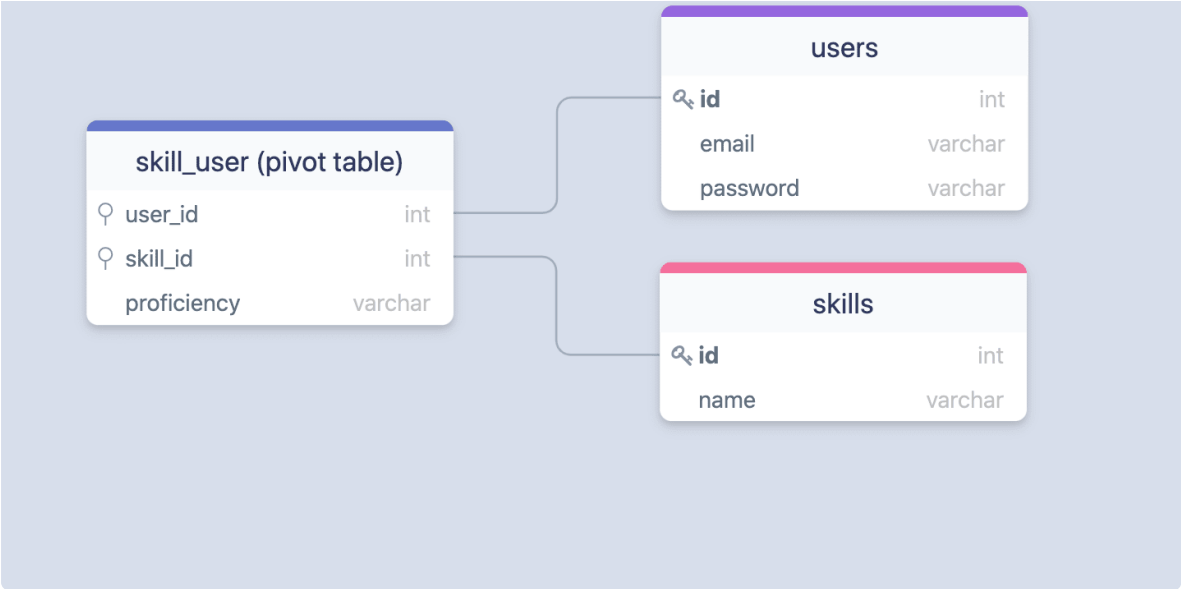
```
import {
  column,
  BaseModel,
  belongsTo,
  BelongsTo
} from '@ioc:Adonis/Lucid/Orm'

export default class Profile extends BaseModel {
  // Foreign key is still on the same model
  @column()
  public userId: number

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

Muitos para muitos

Um relacionamento muitos-para-muitos é um pouco complexo, pois permite que ambos os lados tenham mais de um relacionamento entre si. Por exemplo: **um usuário pode ter muitas habilidades** e **uma habilidade também pode pertencer a muitos usuários** .

Você precisa de uma terceira tabela (geralmente conhecida como tabela dinâmica) para que esse relacionamento funcione. A tabela dinâmica contém as chaves estrangeiras de ambas as outras tabelas.

No exemplo a seguir, a `skill_user` tabela possui as chaves estrangeiras tanto para a tabela `users` quanto para a `skills` tabela, permitindo que cada usuário tenha diversas habilidades e vice-versa.



A seguir estão os exemplos de migrações para as tabelas `users`, `skills` e `skill_user`.

Usuários habilidades usuário_habilidade

```
import BaseSchema from '@ioc:Adonis/Lucid/Schema'

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

  public async up () {
    this.schema.createTable(this.tableName, (table) => {
      table.increments('id').primary()
      table.timestamp('created_at', { useTz: true })
      table.timestamp('updated_at', { useTz: true })
    })
  }
}
```

Definindo relacionamento no modelo

O relacionamento muitos para muitos é definido usando o decorador **@manyToMany** em uma propriedade de modelo.

Não há necessidade de criar um modelo para a tabela dinâmica.

```
import Skill from 'App/Models/Skill'
import {
  column,
  BaseModel,
  manyToMany,
  ManyToMany,
} from '@ioc:Adonis/Lucid/Orm'

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

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

Chaves de relacionamento personalizadas

Uma relação manyToMany depende de muitas chaves diferentes para configurar adequadamente o relacionamento. Todas essas chaves são calculadas usando convenções padrão. No entanto, você está livre para substituí-los.

- `localKey` é a chave primária do modelo pai (ou seja, Usuário)

- `pivotForeignKey` é a chave estrangeira para estabelecer o relacionamento com o modelo pai. O valor padrão é a `snake_case` versão do nome do modelo pai e sua chave primária.
- `pivotRelatedForeignKey` é a chave estrangeira para estabelecer o relacionamento com o modelo relacionado. O valor padrão é a `snake_case` versão do nome do modelo relacionado e sua chave primária.

```
@manyToMany(() => Skill, {
  localKey: 'id',
  pivotForeignKey: 'user_id',
  relatedKey: 'id',
  pivotRelatedForeignKey: 'skill_id',
})
public skills: ManyToMany<typeof Skill>
```

Lembre-se, se você pretende usar camelCase para definição de chave estrangeira, lembre-se de que a **estratégia de nomenclatura** padrão irá convertê-la automaticamente para Snake_case.

Tabela dinâmica personalizada

O valor padrão para o nome da tabela dinâmica é calculado **combinando** o **nome do modelo pai** e o **nome do modelo relacionado** . No entanto, você também pode definir uma tabela dinâmica personalizada.

```
  pivotTable: 'user_skills',
})
public skills: ManyToMany<typeof Skill>
```

Colunas dinâmicas adicionais

Às vezes, sua tabela dinâmica terá colunas adicionais. Por exemplo, você está armazenando `proficiency` junto com a habilidade do usuário.

Você terá que informar um relacionamento manyToMany sobre esta coluna extra. Caso contrário, o Lucid não o selecionará durante as consultas de busca.

```
@manyToMany(() => Skill, {
  pivotColumns: ['proficiency'],
})
public skills: ManyToMany<typeof Skill>
```

Carimbos de data e hora da tabela dinâmica

Você pode ativar o suporte para carimbos **de data/hora criados** e **atualizados em** suas tabelas dinâmicas usando a `pivotTimestamps` propriedade.

- Uma vez definido, o Lucid definirá/atualizará automaticamente esses carimbos de data/hora nas consultas de inserção e atualização.
- Converte-os em uma instância da classe **Luxon Datetime** durante a busca.

```
  pivotTimestamps: true
})
public skills: ManyToMany<typeof Skill>
```

As configurações `pivotTimestamps = true` assumem que os nomes das colunas são definidos como `created_at` e `updated_at`. No entanto, você também pode definir nomes de colunas personalizados.

```
@manyToMany(() => Skill, {
  pivotTimestamps: {
    createdAt: 'creation_date',
    updatedAt: 'updation_date'
  }
})
public skills: ManyToMany<typeof Skill>
```

Para desabilitar um carimbo de data/hora específico, você pode definir seu valor como `false`.

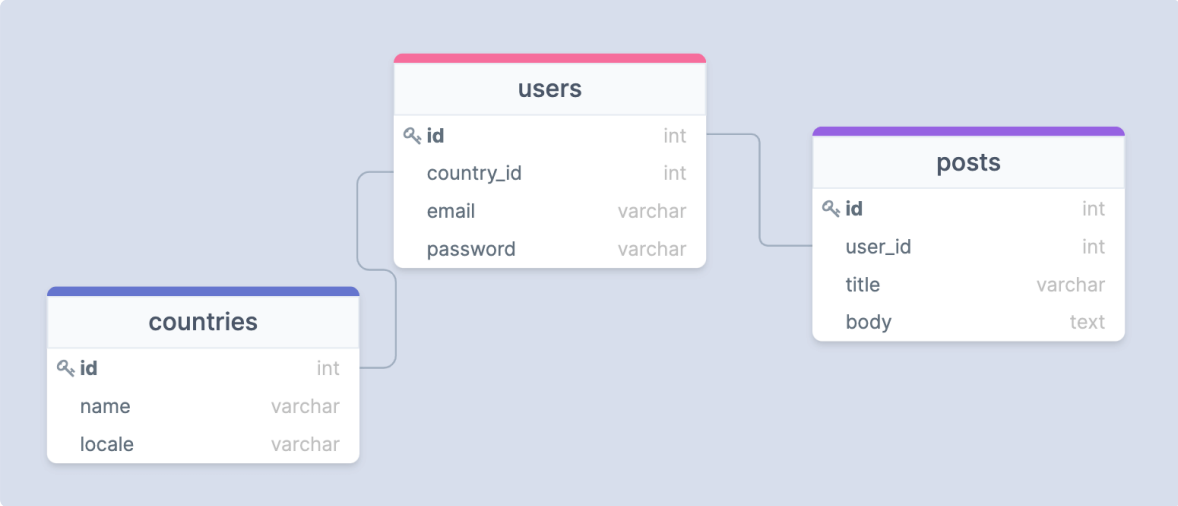
```
@manyToMany(() => Skill, {
  pivotTimestamps: {
    createdAt: 'creation_date',
    updatedAt: false // turn off update at timestamp field
  }
})
public skills: ManyToMany<typeof Skill>
```

HasManyThrough

O relacionamento HasManyThrough é semelhante ao `HasMany` relacionamento, mas cria o relacionamento por meio de um modelo intermediário. Por exemplo, **um país tem muitas postagens por meio de usuários** .

- Este relacionamento precisa que o modelo direto (ou seja, Usuário) tenha uma referência de chave estrangeira com o modelo atual (ou seja, País).

O modelo relacionado (ou seja, Post) possui uma referência de chave estrangeira com o modelo direto (ou seja, Usuário).



A seguir estão os exemplos de migrações para as tabelas `countries` , `users` e `posts` .

países Usuários Postagens

```
export default class Countries extends BaseSchema {
  protected tableName = 'countries'

  public async up () {
    this.schema.createTable(this.tableName, (table) => {
      table.increments('id').primary()
      table.timestamp('created_at', { useTz: true })
      table.timestamp('updated_at', { useTz: true })
    })
  }
}
```

Definindo relacionamento no modelo

Depois de criar as tabelas com as colunas obrigatórias, você também terá que definir o relacionamento no modelo Lucid.

O relacionamento has many through é definido usando o decorador **@hasManyThrough** em uma propriedade de modelo.

```
import User from 'App/Models/User'
import {
  BaseModel,
  column,
  hasManyThrough,
  HasManyThrough
} from '@ioc:Adonis/Lucid/Orm'

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

  @hasManyThrough([
    () => Post,
    () => User,
  ])
  public posts: HasManyThrough<typeof Post>
}
```

Relacionamento de pré-carregamento

O pré-carregamento permite buscar os dados do relacionamento junto com a consulta principal. Por exemplo: Selecione todos os usuários e `preload` seus perfis ao mesmo tempo.

- O `preload` método aceita o nome do relacionamento definido no modelo.
- O valor da propriedade de relacionamento para `hasOne` e o `belongsTo` relacionamento é definido para a instância de modelo relacionada ou `null` quando nenhum registro é encontrado.

```
const users = await User
  .query()
  .preload('profile')

users.forEach((user) => {
  console.log(user.profile)
})
```

Você pode modificar a consulta de relacionamento passando um retorno de chamada opcional para o `preload` método.

```
const users = await User
  .query()
  .preload('profile', (profileQuery) => {
    profileQuery.where('isActive', true)
  })
```

Pré-carregar vários relacionamentos

Você pode `preload` criar vários relacionamentos chamando o `preload` método várias vezes. Por exemplo:

```
const users = await User
  .query()
  .preload('profile')
  .preload('posts')
```

Você pode pre-carregar relacionamentos aninhados usando o construtor de consultas de relacionamento acessível por meio do retorno de chamada opcional.

No exemplo a seguir, buscamos todos os usuários, pré-carregamos suas postagens e, em seguida, buscamos todos os comentários de cada postagem, junto com o usuário do comentário.

```
const users = await User
  .query()
  .preload('posts', (postsQuery) => {
    postsQuery.preload('comments', (commentsQuery) => {
      commentsQuery.preload('user')
    })
  })
```

Muitas para muitas colunas dinâmicas

Ao pré-carregar um relacionamento manyToMany, as colunas da tabela dinâmica são movidas para o `$extras` objeto na instância do relacionamento.

Por padrão, selecionamos apenas as chaves estrangeiras da tabela dinâmica. No entanto, é possível definir colunas dinâmicas adicionais para selecionar ao **definir o relacionamento** ou o tempo de execução.

```
  .query()
  .preload('skills', (query) => {
    query.pivotColumns(['proficiency'])
  })

users.forEach((user) => {
  user.skills.forEach((skill) => {
    console.log(skill.$extras.pivot_proficiency)
    console.log(skill.$extras.pivot_user_id)
    console.log(skill.$extras.pivot_skill_id)
    console.log(skill.$extras.pivot_created_at)
  })
})
```

Relacionamentos de carga lenta

Junto com o pré-carregamento, você também pode carregar relacionamentos diretamente de uma instância de modelo.

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

// Lazy load the profile
await user.load('profile')
console.log(user.profile) // Profile | null

// Lazy load the posts
await user.load('posts')
console.log(user.posts) // Post[]
```

Assim como o `preload` método, o `load` método também aceita um retorno de chamada opcional para modificar a consulta de relacionamento.

```
  profileQuery.where('isActive', true)
})
```

Você pode carregar vários relacionamentos chamando o `load` método várias vezes ou capturando uma instância do carregador de relacionamento subjacente.

```
// Calling "load" method multiple times
await user.load('profile')
await user.load('posts')
```

```
// Using the relationships loader
await user.load((loader) => {
  loader.load('profile').load('posts')
})
```

Limitar relacionamentos pré-carregados

Digamos que você queira carregar todas as postagens e buscar os três comentários recentes de cada postagem.

Usar o `limit` método do construtor de consultas não fornecerá o resultado desejado, pois o limite é aplicado a todo o conjunto de dados e não aos comentários de uma postagem individual.

Portanto, você deve usar o `groupLimit` método que utiliza **funções de janela SQL** para aplicar um limite em cada registro pai separadamente.

```
  .query()
  .preload('comments', (query) => {
    query.groupLimit(3)
  })
```

Construtor de consultas de relacionamento

> Certifique-se de ler a **documentação da API de relacionamento** para visualizar todos os métodos/propriedades disponíveis no construtor de consultas.

Você também pode acessar o construtor de consultas para um relacionamento usando o `related` método. As consultas de relacionamento sempre têm como escopo uma determinada instância do modelo pai.

O Lucid adicionará automaticamente a `where` cláusula para limitar as postagens para um determinado usuário no exemplo a seguir.

```
const user = await User.find(1)
const posts = await user.related('posts').query()
```

O `query` método retorna uma instância do construtor de consultas padrão e você pode encadear qualquer método a ela para adicionar restrições adicionais.

```
  .related('posts')
  .query()
  .where('isPublished', true)
  .paginate(1)
```

Você também pode usar o construtor de consultas de relacionamento para `update` linhas `delete` relacionadas. No entanto, fazer isso **não executará** nenhum dos ganchos do modelo.

Filtrar por relacionamentos

You can also filter the records of the main query by checking for the existence or absence of a relationship. For example, **select all posts that have received one or more comments**.

You can filter by relationship using the `has` or the `whereHas` methods. They accept the relationship name as the first argument. Optionally you can also pass an operator and number of expected rows.

```
// Get posts with one or more comments
const posts = await Post
  .query()
  .has('comments')

// Get posts with more than 2 comments
const posts = await Post
  .query()
  .has('comments', '>', 2)
```

have one or more approved comments.

```
const posts = await Post
  .query()
  .whereHas('comments', (query) => {
    query.where('isApproved', true)
  })
```

Similar to the `has` method, the `whereHas` also accepts an optional operator and the count of expected rows.

```
const posts = await Post
  .query()
  .whereHas('comments', (query) => {
    query.where('isApproved', true)
  }, '>', 2)
```

Following is the list of `has` and `whereHas` variations.

- `orHas | orWhereHas` adds an OR clause for the relationship existence.
- `doesntHave | whereDoesntHave` checks for the absence of the relationship.
- `orDoesntHave | orWhereDoesntHave` adds an OR clause for the relationship absence.

Relationship aggregates

comments for each post.

withAggregate

The `withAggregate` method accepts the relationship as the first argument and a mandatory callback to define the value's aggregate function and property name.

In the following example, the `comments_count` property is moved to the `$extras` object because it is not defined as a property on the model.

```
const posts = await Post
  .query()
  .withAggregate('comments', (query) => {
    query.count('*').as('comments_count')
  })

posts.forEach((post) => {
  console.log(post.$extras.comments_count)
})
```

---

withCount

Since counting relationship rows is a very common requirement, you can instead use the `withCount` method.

```
posts.forEach((post) => {
  console.log(post.$extras.comments_count)
})
```

You can also provide a custom name for the count property using the `as` method.

```
const posts = await Post
  .query()
  .withCount('comments', (query) => {
    query.as('commentsCount')
  })

posts.forEach((post) => {
  console.log(post.$extras.commentsCount)
})
```

You can define constraints to the count query by passing an optional callback to the `withCount` method.

```
const posts = await Post
  .query()
  .withCount('comments', (query) => {
    query.where('isApproved', true)
  })
```

Lazy load relationship aggregates

`loadAggregate` methods.

```
const post = await Post.findOrFail()
await post.loadCount('comments')

console.log(post.$extras.comments_count)
```

```
const post = await Post.findOrFail()
await post.loadAggregate('comments', (query) => {
  query.count('*').as('commentsCount')
})

console.log(post.$extras.commentsCount)
```

Make sure you are using the `loadCount` method only when working with a single model instance. If there are multiple model instances, it is better to use the query builder `withCount` method.

Relationship query hook

You can define an `onQuery` relationship hook at the time of defining a relationship. Then, the query hooks get executed for all the **select**, **update**, and **delete** queries executed by the relationship query builder.

The `onQuery` method is usually helpful when you always apply certain constraints to the relationship query.

```
import {
  column,
  BaseModel,
  hasMany,
  HasMany
} from '@ioc:Adonis/Lucid/Orm'

export default class User extends BaseModel {
  @hasMany(() => UserEmail)
  public emails: HasMany<typeof UserEmail>

  @hasMany(() => UserEmail, {
    onQuery: (query) => {
      query.where('isActive', true)
    }
  })
  public activeEmails: HasMany<typeof UserEmail>
}
```

Create relationships

You can create relationships between two models using the relationships persistence API. Make sure to also check out the **API docs** to view all the available methods.

create

In the following example, we create a new comment and link it to the post at the same time. The `create` method accepts a plain JavaScript object to persist. The foreign key value is defined automatically.

```
const comment = await post.related('comments').create({
  body: 'This is a great post'
})

console.log(comment.postId === post.id) // true
```

save

Following is an example using the `save` method. The `save` method needs an instance of the related model. The foreign key value is defined automatically.

```
const post = await Post.findOrFail(1)

const comment = new Comment()
comment.body = 'This is a great post'

await post.related('comments').save(comment)

console.log(comment.postId === post.id) // true
```

createMany

You can also create multiple relationships using the `createMany` method. The method is only available for `hasMany` and `manyToMany` relationships.

The `createMany` method returns an array of persisted model instances.

```
  .related('comments')
  .createMany([
    {
      body: 'This is a great post.'
    },
    {
      body: 'Well written.'
    }
  ])
```

saveMany

Similar to the `save` method. The `saveMany` method allows persisting multiple relationships together.

```
const comment1 = new Comment()
comment1.body = 'This is a great post'

const comment2 = new Comment()
comment2.body = 'Well written'

await Post
  .related('comments')
  .saveMany([comment1, comment2])
```

associate

The `associate` method is exclusive to the `belongsTo` relationship. It let you associate two models with each other.

```
const profile = new Profile()
profile.avatarUrl = 'foo.jpg'
await profile.related('user').associate(user)
```

dissociate

The `dissociate` removes the relationship by setting the foreign key to `null`. Thus, the method is exclusive to the `belongsTo` relationship.

```
await profile = await Profile.findOrFail(1)
await profile.related('user').dissociate()
```

attach

The `attach` method is exclusive to a `manyToMany` relationship. It allows you to create a relationship between two persisted models inside the pivot table.

The `attach` method just needs the `id` of the related model to form the relationship inside the pivot table.

```
const user = await User.find(1)
const skill = await Skill.find(1)

// Performs insert query inside the pivot table
await user.related('skills').attach([skill.id])
```

columns.

```
await user.related('skills').attach({
  [skill.id]: {
    proficiency: 'Beginner'
  }
})
```

detach

The `detach` method is the opposite of the `attach` method and allows you to remove the relationship **from the pivot table**.

It optionally accepts an array of `ids` to remove. Calling the method without any arguments will remove all the relationships from the pivot table.

```
const user = await User.find(1)
const skill = await Skill.find(1)

await user.related('skills').detach([skill.id])

// Remove all skills for the user
await user.related('skills').detach()
```

sync

The `sync` method allows you to sync the pivot rows. The payload provided to the `sync` method is considered the source of truth, and we compute a diff internally to execute the following SQL queries.

- Update the rows present in the pivot table and the sync payload but has one or more changed arguments.
- Remove the rows present in the pivot table but missing in the sync payload.
- Ignore rows present in both the pivot table and the sync payload.

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

// Only skills with id 1, 2, 3 will stay in the pivot table
await user.related('skills').sync([1, 2, 3])
```

You can also define additional pivot columns as an object of key-value pair.

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

await user.related('skills').sync({
  [1]: {
    proficiency: 'Beginner',
  },
  [2]: {
    proficiency: 'Master'
  },
  [3]: {
    proficiency: 'Master'
  }
})
```

You can disable the `detach` option to sync rows without removing any rows from the pivot table.

```
.related('skills')
// Add skills with id 1,2,3, but do not remove any
// rows from the pivot table
.sync([1, 2, 3], false)
```

Delete relationship

Na maioria das vezes, você pode excluir linhas relacionadas diretamente de seu modelo. Por exemplo: **Você pode excluir um comentário pelo seu id, diretamente utilizando o modelo Comment** , não há necessidade de acionar a exclusão do comentário via post.

- Para um `manyToMany` relacionamento, você pode usar o `detach` método para remover a linha da tabela dinâmica.
- Use o `dissociate` método para remover um relacionamento pertence a sem excluir a linha da tabela do banco de dados.

Usando a ação onDelete

Você também pode usar a `onDelete` ação do banco de dados para remover os dados relacionados do banco de dados. Por exemplo: Exclua as postagens de um usuário quando o próprio usuário for excluído.

A seguir está um exemplo de migração para definir a `onDelete` ação.

```
    table.increments('id')
    table
      .integer('user_id')
      .unsigned()
      .references('users.id')
      .onDelete('CASCADE')
  })
```

Edite esta página no GitHub