



Aula 02:



Desenvolvendo o CRUD da API

Buscando registros na API

Após criarmos a base de dados da API, implementaremos agora um novo service na estrutura do projeto. Para isso, crie uma pasta na raiz do projeto chamada **services**, dentro desta pasta crie um novo arquivo chamado **gameService.js**.

No arquivo **gameService.js** criaremos uma **classe** que será responsável por conter os **métodos** de manipulação do banco de dados, como buscar, criar, alterar e deletar registros.

O primeiro método a ser criado é o **getAll()**, que irá selecionar todos os registros de games cadastrados no banco de dados. Esse método será uma **função assíncrona**, por isso utilizaremos o padrão **async/await**.

 **services**
 **gameService.js**

```
1 import Game from "../models/Games.js";
2
3 class gameService {
4   async getAll() {
5     try {
6       const games = await Game.find();
7       return games;
8     } catch (error) {
9       console.log(error);
10    }
11  }
12}
13 export default new gameService();
```



Funções assíncronas (async/await)

O padrão **async/await** no JavaScript é uma maneira moderna e mais fácil de lidar com operações **assíncronas**, como buscar dados de uma API, sem precisar usar muitos callbacks ou **promises**.

Função assíncrona (async): Quando você define uma função como `async`, ela automaticamente retorna uma promessa e permite que você use o **await** dentro dela.

await: O `await` é usado dentro de funções `async` para "**esperar**" que uma promessa seja resolvida. Isso significa que o código vai parar e esperar até que a operação assíncrona termine, mas sem bloquear o resto do programa. Depois que a promessa é resolvida, o código continua a ser executado.

```
1 import Game from "../models/Games.js";
2
3 class gameService {
4   async getAll() {
5     try {
6       const games = await Game.find();
7       return games;
8     } catch (error) {
9       console.log(error);
10    }
11  }
12}
13 export default new gameService();
```



Funções assíncronas (async/await)



Promisse:

```
1 import Game from "../models/Games.js";
2
3 class gameService {
4   getAll() {
5     return Game.find()
6       .then(games => {
7         return games;
8       })
9       .catch(error => {
10        console.log(error);
11      });
12 }
13}
14
15 export default new gameService();
```

Async/await:

```
1 import Game from "../models/Games.js";
2
3 class gameService {
4   async getAll() {
5     try {
6       const games = await Game.find();
7       return games;
8     } catch (error) {
9       console.log(error);
10    }
11  }
12}
13 export default new gameService();
```

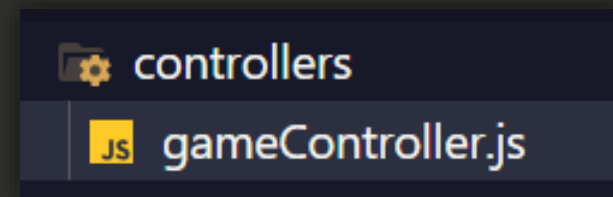


Criando o Controller

Após criarmos o service, criaremos agora nosso **controller**.

Para isso, crie uma pasta na raiz do projeto chamada **controllers**, dentro desta pasta crie um novo arquivo chamado **gameController.js**.

```
1 import gameService from "../services/gameService.js";
2
3 const getAllGames = async (req, res) => {
4   try {
5     const games = await gameService.getAll();
6     res.status(200).json({ games: games });
7   } catch (error) {
8     console.log(error);
9     res.status(500).json({ error: "Erro interno do servidor." });
10  }
11};
12 export default { getAllGames };
```



O controller será responsável por

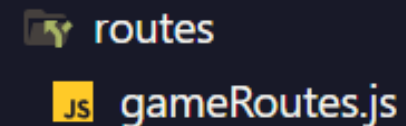
tratar as requisições do cliente.

No exemplo ao lado, **getAllGames** recebe uma *função assíncrona* que chama o método **getAll()** do **service** para buscar os registros no banco. Em seguida, o controller retorna esses registro em um **JSON**, bem como o **código de status 200**. Além disso, também é feito o tratamento de erro.



Modularizando as rotas

Visando a eficiência do projeto iremos agora modularizar as rotas da API (endpoints). Para isso, crie uma pasta na raiz do projeto chamada **routes**, dentro desta pasta crie um novo arquivo chamado **gameRoutes.js**.



Nesse arquivo iremos importar o **express**, em seguida carregar na variável **gameRoutes** o método **express.Router()**, responsável por fazer o gerenciamento de rotas. Importaremos também o **gameController** responsável por tratar as requisições.

```
1 import express from 'express'
2 const gameRoutes = express.Router()
3 import gameController from '../controllers/gameController.js'
4
5 // Endpoint para listar todos os games
6 gameRoutes.get("/games", gameController.getAllGames)
7
8 export default gameRoutes
```

Nesse arquivo criaremos todos os **endpoints** (rotas) da API.

Começaremos criando a rota **"/games"** que ao receber um requisição do tipo **GET**, chamará o método **getAllGames()** do **gameController** para fazer o tratamento da requisição.

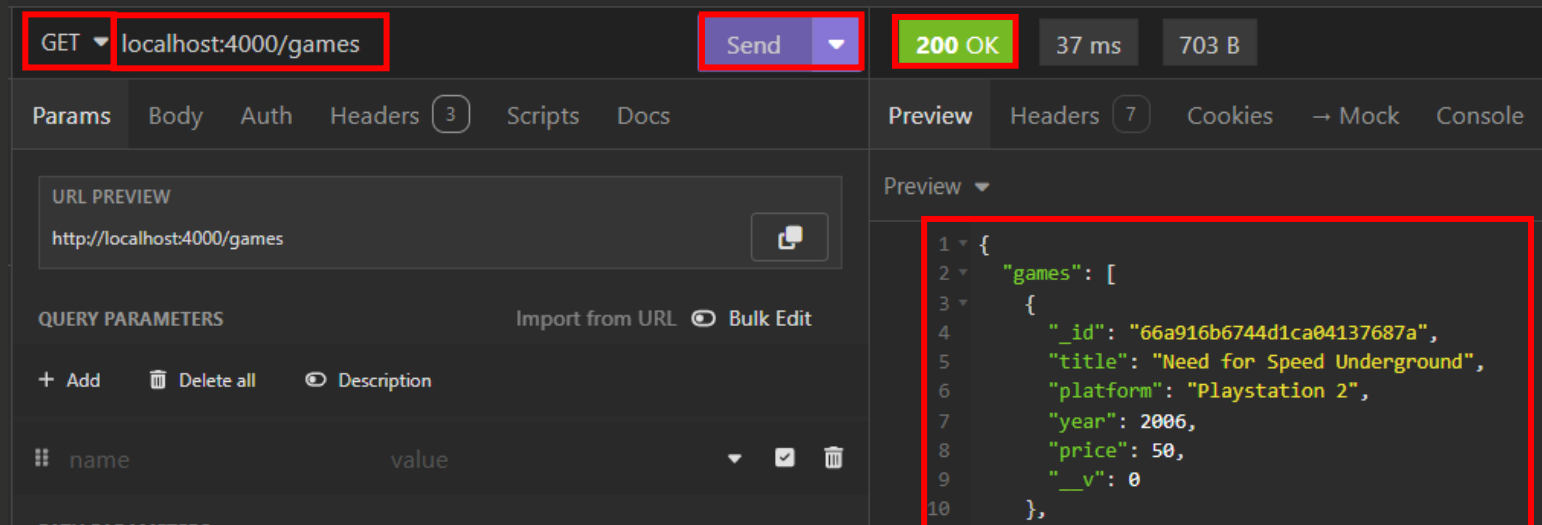


Listando os registros (GET)

Por fim, devemos importar nossas rotas no arquivo **index.js**, para que sejam acessíveis na aplicação.



```
1 const app = express();
2
3 import gameRoutes from "../routes/gameRoutes.js";
4
5 // Configurações do Express
6 app.use(express.urlencoded({ extended: false }));
7 app.use(express.json());
8 app.use("/", gameRoutes);
```

Feito isso, testaremos a requisição no **Insomnia**, enviando uma requisição **GET** para a rota **"/games"**:



Cadastrando dados na API

Para começarmos a cadastrar dados na API, criaremos um novo método chamado **Create()** na classe **gameService** que irá inserir novos registros no banco de dados.

 services
 gameService.js

```
1  async Create(title, platform, year, price) {  
2      try{  
3          const newGame = new Game({  
4              title,  
5              platform,  
6              year,  
7              price  
8          })  
9          await newGame.save()  
10     } catch (error) {  
11         console.log(error)  
12     }  
13 }
```

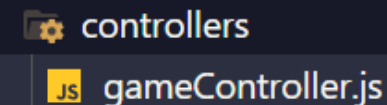


Cadastrando dados na API

Após isso, no arquivo **gameController.js**, criaremos a constante **createGame** que recebe uma função assíncrona. Nessa função será coletado os campos vindos do corpo da requisição POST, em seguida será chamado o método **Create()** do service para cadastrar os registros no banco. Em seguida, o controller retornará o código de status **201 (Created)**. Além disso, também é feito o tratamento de erro, retornando o código de status **500 (Erro interno do servidor)**. Lembre-se também de **exportar createGame** no final do arquivo.



```
1 //Cadastrando um Game
2 const createGame = async (req, res) => {
3   try {
4     const { title, platform, year, price } = req.body;
5     await gameService.Create(title, platform, year, price);
6     res.sendStatus(201); //Código 201 (CREATED)
7   } catch (error) {
8     console.log(error);
9     res.status(500).json({ error: "Erro interno do servidor." });
10  }
```



```
controllers
  gameController.js
```



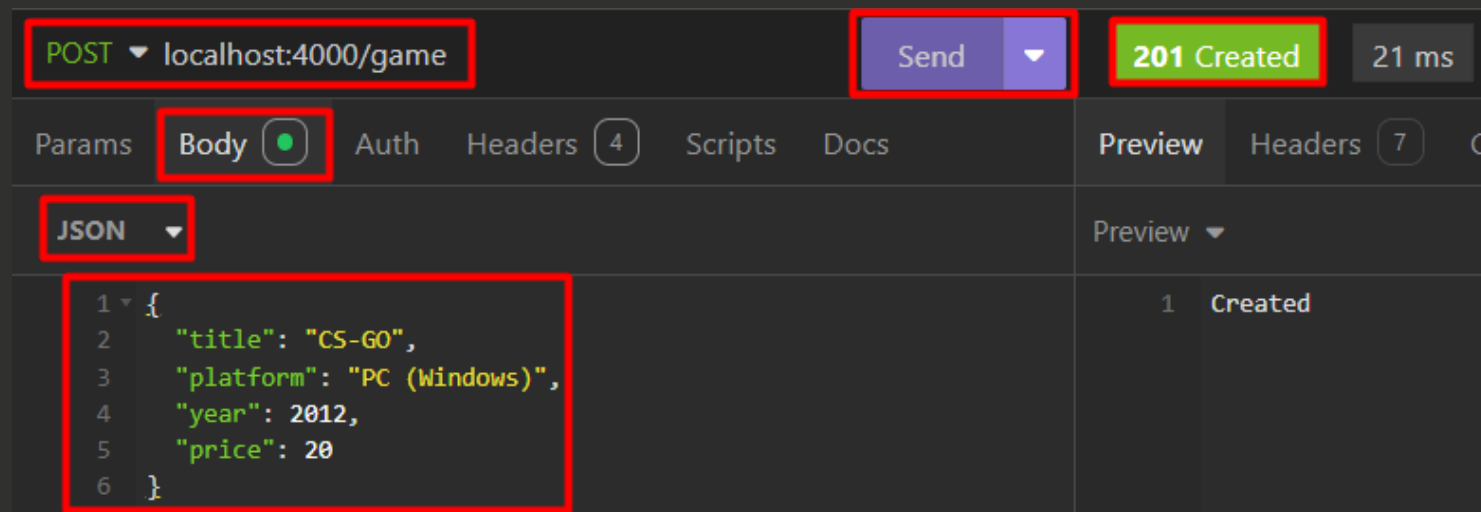
Cadastrando dados na API

Agora devemos apenas criar um novo **endpoint** no arquivo **gameRoutes.js**, que receberá a requisição **POST** e chamará o método **createGame()** do **gameController** para tratar a requisição:

```
1 // Endpoint para cadastrar um Game
2 gameRoutes.post("/game", gameController.createGame);
```



routes
gameRoutes.js

Feito isso, testaremos a requisição no **Insomnia**, enviando uma requisição **POST** para a rota **"/game"**. Será necessário enviar os dados que se deseja cadastrar no corpo da requisição (**BODY**) em formato **JSON**.



Deletando dados na API

Para deletar dados na API, criaremos um novo método chamado **Delete()** na classe **gameService** que irá excluir os registros no banco de dados.

 **services**
 **gameService.js**

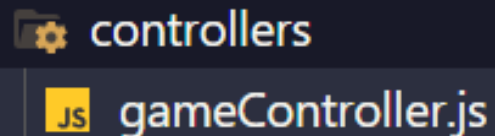
```
1  async Delete(id) {  
2      try {  
3          await Game.findByIdAndDelete(id);  
4          console.log(`Game com a id: ${id} foi deletado.`)  
5      } catch (error) {  
6          console.log(error)  
7      }  
8  }
```



Deletando dados na API

Após isso, no arquivo **gameController.js**, criaremos a constante **deleteGame** que recebe uma função assíncrona. Nessa função será verificado se o ID do jogo enviado é válido. Para isso, precisamos instalar e importar o **ObjectId** da biblioteca **mongoose**. Após essa verificação, será chamado o método **Delete()** do service para excluir o registro em questão do banco de dados. Em seguida, o controller retornará o código de status **204 (No Content)**. Além disso, também é feito o tratamento de erro, retornando o código de status **400 (Bad Request)** ou **500**. Lembre-se também de **exportar deleteGame** no final.

```
1 //Deletando um Game
2 const deleteGame = async (req, res) => {
3   try {
4     if(ObjectId.isValid(req.params.id)){
5       const id = req.params.id
6       gameService.Delete(id)
7       res.sendStatus(204) // Código 204 (NO CONTENT) :
8     }else{
9       res.sendStatus(400) // Código 400 (BAD REQUEST) :
10    }
11  } catch (error) {
12    console.log(error)
13    res.status(500).json({ error: 'Erro interno do servidor.'})
14  }
15 }
```



controllers
gameController.js

```
npm install mongoose
```

```
1 import {ObjectId} from 'mongoose'
```



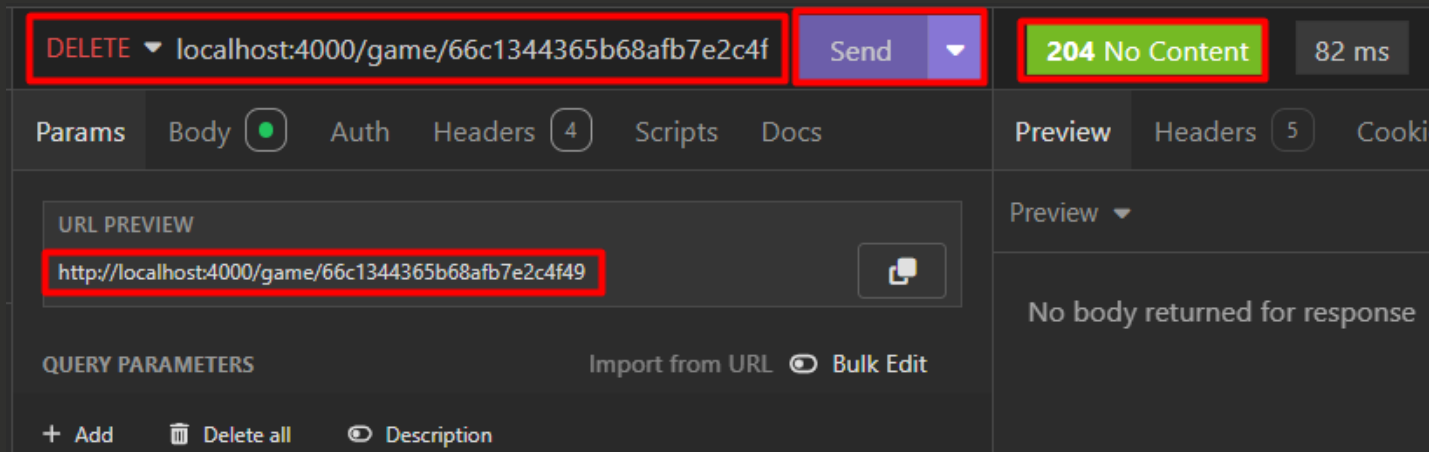
Deletando dados na API

Agora devemos apenas criar um novo **endpoint** no arquivo **gameRoutes.js**, que receberá a requisição **DELETE** e chamará o método **deleteGame()** do **gameController** para tratar a requisição. Essa rota terá um **parâmetro obrigatório** que será a **ID** do jogo que se deseja excluir:

```
1 // Endpoint para deletar um Game
2 gameRoutes.delete("/game/:id", gameController.deleteGame);
```



routes
gameRoutes.js

Feito isso, testaremos a requisição no **Insomnia**, enviando uma requisição **DELETE** para a rota **"/game"**. Lembre-se que será necessário enviar a ID do jogo que se deseja excluir na URL:



Alterando dados na API

Para começarmos a alterar dados na API, criaremos um novo método chamado **Update()** na classe **gameService** que irá alterar registros no banco de dados.

 **services**
 **gameService.js**

```
1  async Update(id, title, platform, year, price) {
2    try {
3      const updatedGame = await Game.findByIdAndUpdate(
4        id,
5        {
6          title,
7          platform,
8          year,
9          price,
10         },
11         { new: true }
12       );
13       console.log(`Dados do game com id: ${id} alterados com sucesso.`);
14       return updatedGame;
15     } catch (error) {
16       console.log(error);
17     }
18   }
```

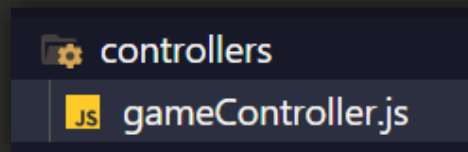


Alterando dados na API

Após isso, no arquivo **gameController.js**, criaremos a constante **updateGame** que recebe uma função assíncrona. Nessa função será coletado os campos vindos do corpo da requisição PUT, em seguida será chamado o método **Update()** do service para alterar o registro no banco. O controller retornará o código de status **200 (OK)** e um **JSON** com os dados do jogo alterado. Lembre-se também de **exportar updateGame**.



```
1 //Alterando um Game
2 const updateGame = async (req, res) => {
3   try {
4     if (ObjectId.isValid(req.params.id)) {
5       const id = req.params.id;
6       const { title, platform, year, price } = req.body;
7       const game = await gameService.Update(id, title, platform, year, price);
8       res.status(200).json({ game }); //Código 200 (OK)
9     } else {
10      res.sendStatus(400); //Código 400 (BAD REQUEST)
11    }
12  } catch (error) {
13    console.log(error);
14    res.sendStatus(500); // Erro interno do servidor
15  }
16 };
```



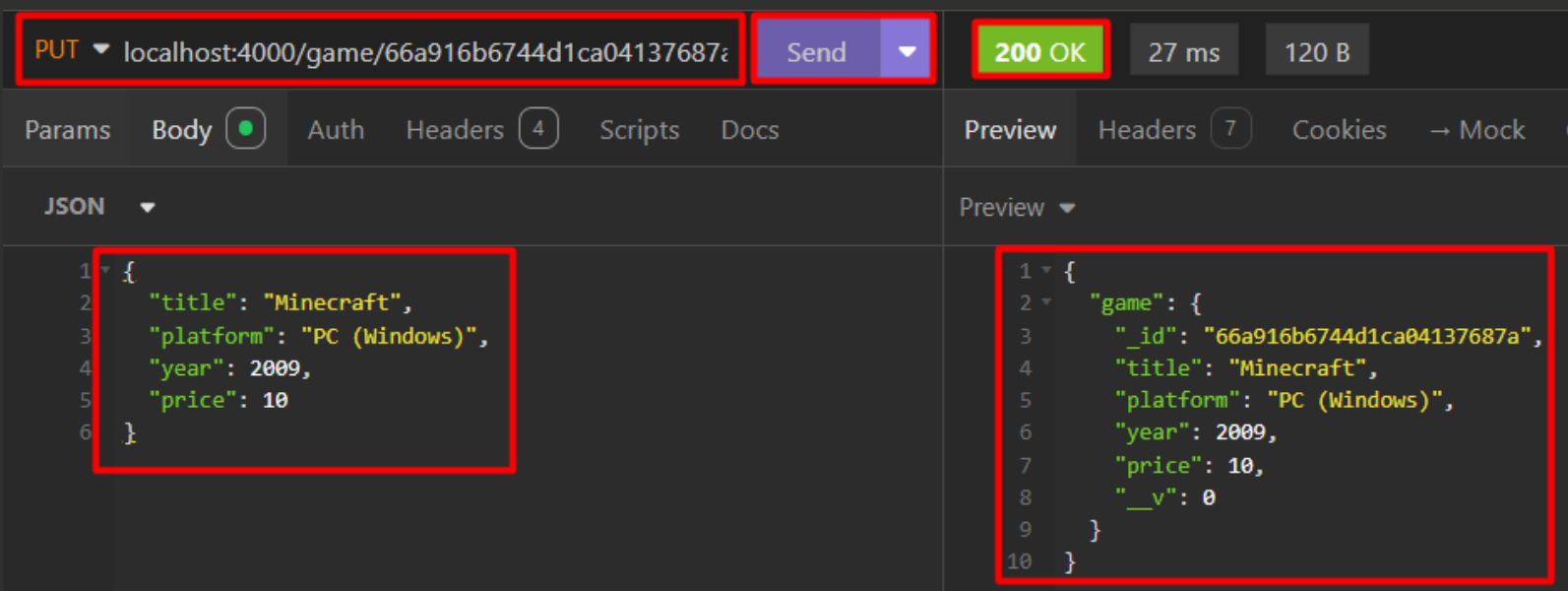
Alterando dados na API

Agora devemos apenas criar um novo **endpoint** no arquivo **gameRoutes.js**, que receberá a requisição **PUT** e chamará o método **updateGame()** do **gameController** para tratar a requisição:

```
1 // Endpoint para alterar um Game
2 gameRoutes.put("/game/:id", gameController.updateGame);
```

routes
gameRoutes.js

Feito isso, testaremos a requisição no **Insomnia**, enviando uma requisição **PUT** para a rota **"/game"**. Será necessário enviar os dados que se deseja alterar no corpo da requisição (**BODY**) em formato **JSON**, bem como a **ID do game** na URL.



The screenshot shows the Insomnia API client interface. The top bar displays the method **PUT**, the URL **localhost:4000/game/66a916b6744d1ca04137687a**, and the status **200 OK** with a response time of **27 ms** and a size of **120 B**. The **Body** tab is selected, showing the request body as a JSON object:



```
{  "title": "Minecraft",  "platform": "PC (Windows)",  "year": 2009,  "price": 10}
```

. The **Preview** tab is also selected, showing the response body as a JSON object:

```
{  "game": {    "_id": "66a916b6744d1ca04137687a",    "title": "Minecraft",    "platform": "PC (Windows)",    "year": 2009,    "price": 10,    "__v": 0  }}
```


Listando um registro único

Para listar um registro da API, criaremos um novo método chamado **getOne()** na classe **gameService** que irá listar um único registro do banco de dados.

 **services**
 **gameService.js**

```
1 async getOne(id) {  
2     try {  
3         const game = await Game.findOne({_id: id})  
4         return game  
5     } catch (error) {  
6         console.log(error)  
7     }  
8 }
```

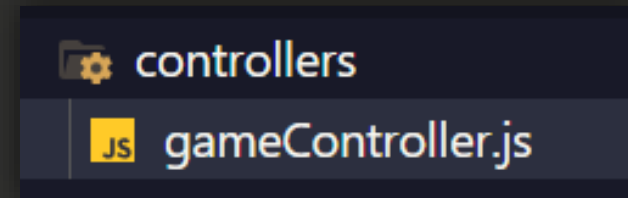


Listando um registro único

Após isso, no arquivo **gameController.js**, criaremos a constante **getOneGame** que recebe uma função assíncrona. Nessa função será verificado se o ID do jogo enviado é válido. Após essa verificação, será chamado o método **getOne()** do service para buscar o registro em questão no banco de dados. Em seguida, o controller retornará o código de status **200 (OK)**. E um **JSON** com os dados do jogo.



```
1 //Listando um único Game
2 const getOneGame = async (req, res) => {
3   try {
4     if (ObjectId.isValid(req.params.id)) {
5       const id = req.params.id
6       const game = await gameService.getOne(id)
7       if (!game) {
8         res.sendStatus(404) // Jogo não encontrado
9       } else {
10        res.status(200).json({ game })
11      }
12    } else {
13      res.sendStatus(400) // Requisição inválida - Bad request
14    }
15  } catch (error) {
16    console.log(error)
17    res.sendStatus(500) // Erro interno do servidor
18  }
19}
20
21export default { getAllGames, createGame, deleteGame, updateGame, getOneGame }
```



Listando um registro único

Agora devemos apenas criar um novo **endpoint** no arquivo **gameRoutes.js**, que receberá a requisição **GET** e chamará o método **getOneGame()** do **gameController** para tratar a requisição. Essa rota terá um **parâmetro obrigatório** que será a **ID** do jogo que se deseja buscar:

```
1 // Endpoint para listar um único Game
2 gameRoutes.get("/game/:id", gameController.getOneGame)
```

routes
gameRoutes.js

Feito isso, testaremos a requisição no **Insomnia**, enviando uma requisição **GET** para a rota **"/game"**. Lembre-se que será necessário enviar a ID do jogo que se deseja buscar na URL:

The screenshot shows the Insomnia REST client interface. At the top, a GET request is made to `localhost:4000/game/66aee852440693aeb9ec0f0a`, resulting in a `200 OK` status, `94 ms` response time, and `123 B` of data. The response body is a JSON object:

```
{
  "game": {
    "_id": "66aee852440693aeb9ec0f0a",
    "title": "Forza Horizon 4",
    "platform": "Xbox One",
    "year": 2018,
    "price": 220,
    "__v": 0
  }
}
```



Aula 02:

Desenvolvendo o CRUD da API