

**AV2 - Guia do Aluno - API Marketplace Enxuto – DSW – 3º Bimestre 1) Objetivo**

Criar uma API REST simulando um **marketplace simples**, com dois relacionamentos:

• **1-1:** User → Store (cada usuário tem uma única loja)

• **1-N:** Store → Product (cada loja tem vários produtos)

A tabela **User já existe** no projeto base, você só vai estender o modelo.

**2) Modelos (schema.prisma)**

| model User {  id Int @id @default(autoincrement())  email String @unique  name String?  store Store? *// 1-1: um usuário tem uma loja*  createdAt DateTime @default(now())  updatedAt DateTime @updatedAt  }  model Store {  id Int @id @default(autoincrement())  name String  userId Int @unique  user User @relation(fields: [userId], references: [id], onDelete: Cascade) products Product[] *// 1-N: uma loja tem vários produtos*  createdAt DateTime @default(now())  updatedAt DateTime @updatedAt  }  model Product {  id Int @id @default(autoincrement())  name String  price Decimal @db.Decimal(10,2)  storeId Int  store Store @relation(fields: [storeId], references: [id], onDelete: Cascade)  createdAt DateTime @default(now())  updatedAt DateTime @updatedAt  } |
| --- |

**3) Configuração para AlwaysData**

No arquivo **.env**:

| DATABASE\_URL="mysql://USUARIO:SENHA@mysql-USUARIO.alwaysdata.net/NOME\_DO\_BANCO" |
| --- |

**Comandos principais**

****

| *# gerar cliente*  npx prisma generate  *# aplicar schema no banco remoto (AlwaysData)*  npx prisma db push |
| --- |

**Atenção**: O comando prisma migrate dev não funciona no AlwaysData porque exige permissões extras.

**Para a AV2, use prisma db push sempre que ajustar o schema.prisma.**

**4) Rotas de Exemplo (Express + Prisma)**

**src/db.js**

| *// Conexão com o banco de dados usando Prisma*  **import** { PrismaClient } **from** "@prisma/client";  *// Criar uma única instância do Prisma (padrão Singleton)*  **const** prisma = **new** PrismaClient();  *// Conectar ao banco quando o módulo for carregado*  prisma  .$connect()  .**then**(() => {  console.log("✅ Conectado ao banco de dados!");  })  .**catch**((error) => {  console.error("❌ Erro ao conectar:", error.message);  });  *// Exportar a instância para usar nas rotas*  **export default** prisma; |
| --- |

**src/index.js**

**Stores (1-1 com User)**

*// POST /stores body: { name, userId }*

app.post('/stores', **async** (req, res) => {

**try** {

**const** { name, userId } = req.body

**const** store = **await** prisma.store.create({

data: { name, userId: Number(userId) }

})

res.status(201).json(store)

} **catch** (e) { res.status(400).json({ error: e.message }) }

})

*// GET /stores/:id -> retorna loja + user (dono) + produtos*

app.**get**('/stores/:id', **async** (req, res) => {



**try** {

**const** store = **await** prisma.store.findUnique({

**where**: { id: Number(req.**params**.id) },

include: { user: **true**, products: **true** }

})

**if** (!store) **return** res.status(404).json({ error: 'Loja não encontrada' }) res.json(store)

} **catch** (e) { res.status(400).json({ error: e.message }) }

})

Products (1-N com Store)

*// POST /products body: { name, price, storeId }*

app.post('/products', **async** (req, res) => {

**try** {

**const** { name, price, storeId } = req.body

**const** product = **await** prisma.product.create({

data: { name, price: Number(price), storeId: Number(storeId) } })

res.status(201).json(product)

} **catch** (e) { res.status(400).json({ error: e.message }) }

})

*// GET /products -> inclui a loja e o dono da loja*

app.**get**('/products', **async** (req, res) => {

**try** {

**const** products = **await** prisma.product.findMany({

include: { store: { include: { user: **true** } } }

})

res.json(products)

} **catch** (e) { res.status(400).json({ error: e.message }) }

})

**Importante:** além destas rotas, você deverá implementar obrigatoriamente os métodos **PUT** (atualização) e **DELETE** (remoção) tanto para **Stores** quanto para **Products**.

**5) Checklist de Entrega**

• Schema atualizado e aplicado no **AlwaysData** via db push.

• Criar **Store** vinculada a um User (1-1 funcionando).

• Criar **Products** vinculados a uma Store (1-N funcionando).

• Implementar **CRUD completo**:

• POST e GET (já exemplificados).

• **PUT e DELETE (obrigatórios a inclusão no código).**

• **GET /stores/:id** retorna **dono (User)** e **produtos** da loja.

• **GET /products** retorna **loja** e **dono da loja**.

• Código organizado, com tratamento básico de erros.

**6) Critérios de Avaliação**

****

• **Modelagem & Schema (2,0):** Store (1-1 com User) e Product (1-N com Store) funcionando; db push aplicado.

• **CRUD (4,0):** criação, listagem, atualização (PUT) e exclusão (DELETE) de lojas e produtos.

• **Consultas com include (1,5):** retorno de dados relacionados (store + user, products + store + user).

• **Organização & Boas práticas (1,5):** código limpo, nomes claros, erros tratados.

• **Apresentação Individual (1,0):** explicação rápida do código.