Directus

Got it You want to install **Directus** on **Debian**. Ill give you a clean, step-by-step guide for a typical setup. Directus is a **Node.js + SQL database** app, so youll need Node, npm (or yarn), and a database (MySQL, PostgreSQL, or SQLite).

Step 1. Update system

sudo apt update && sudo apt upgrade -y

Step 2. Install Node.js + npm

Directus requires Node.js **18+**.

Install curl if not available sudo apt install -y curl # Add Node.js LTS repo (replace 18.x with 20.x if you want the latest LTS) curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash - sudo apt install -y nodejs # Verify node -v npm -v

---

Step 3. Install a database

Directus supports PostgreSQL, MySQL/MariaDB, or SQLite.
Example with **MariaDB (MySQL-compatible)**:

sudo apt install -y mariadb-server mariadb-client sudo systemctl enable mariadb sudo systemctl start mariadb

Secure the installation:

sudo mysql_secure_installation

Create a database and user for Directus:

CREATE DATABASE directus CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci; CREATE USER 'directus_user'@'localhost' IDENTIFIED BY 'your_password'; GRANT ALL PRIVILEGES ON directus.* TO 'directus_user'@'localhost'; FLUSH PRIVILEGES; EXIT;

Step 4. Install Directus (globally or per project)

Option A: Global install

sudo npm install -g directus

Option B: Project install

mkdir ~/directus-project && cd ~/directus-project npm init directus-project .

Step 5. Configure Directus

Run the init command if you installed globally:

npx directus init

Provide:

- Database client: mysql (or postgres, sqlite3)
- Database host: 127.0.0.1
- Database name: directus
- Database user: directus_user

- Database password: your_password
- Port: 3306 (MySQL default)

This creates a .env file with config.

Step 6. Start Directus

npx directus start

By default, Directus runs on http://localhost:8055.
The first time you open it, youll be prompted to create an **admin account**.

Step 7. Run as a service (optional)

To keep Directus running in the background:

1. Create a systemd service file:

   sudo nano /etc/systemd/system/directus.service

2. Add:

   [Unit] Description=Directus Headless CMS After=network.target [Service] Type=simple User=www-data WorkingDirectory=/home/youruser/directus-project ExecStart=/usr/bin/npx directus start Restart=always [Install] WantedBy=multi-user.target

3. Enable + start service:

   sudo systemctl daemon-reload sudo systemctl enable directus sudo systemctl start directus

At this point, Directus should be running at http://your-server-ip:8055

Do you want me to also show you how to set up **NGINX reverse proxy + HTTPS (Lets Encrypt)** so Directus can be accessed on a clean domain (e.g. cms.example.com)?

You said:

http://my-ip-address:8055 does not work

ChatGPT said:

Got it if http://your-ip:8055 doesnt load, lets troubleshoot step by step.

---

1. Check if Directus is running

Run this in the project folder:

npx directus start

You should see logs like:

Server started at http://0.0.0.0:8055

If you dont, there may be a DB config or Node.js issue.

---

2. Check port binding

By default, Directus sometimes binds to localhost only (127.0.0.1), so it wont be reachable externally.

Edit your .env file in your Directus project:

PORT=8055 HOST=0.0.0.0

Then restart Directus:

npx directus start

Now it should be listening on all interfaces.

---

3. Check firewall (UFW/iptables)

On Debian, port 8055 might be blocked. Run:

sudo ufw allow 8055/tcp sudo ufw reload

Check if its open:

sudo ufw status

---

4. Verify listening process

Run:

ss -tulnp | grep 8055

You should see something like:

tcp LISTEN 0 511 0.0.0.0:8055 0.0.0.0:* users:(("node",pid=12345,fd=23))

If it only shows 127.0.0.1:8055, then its still binding to localhost fix .env as above.

---

5. Access via IP

Try again from your browser:

http://YOUR_SERVER_IP:8055

If youre on the same machine, http://localhost:8055 should also work.

Question: Do you want me to also show you how to **expose Directus on port 80/443 with Nginx + domain** (so you can use http://cms.example.com instead of :8055)? That usually solves firewall + port headaches.

You said:

can directus be used as administration of web applications modules? for example products (e-shop), newsletter, blog, events, accounts and contacts usw...

ChatGPT said:

Yes Directus can absolutely be used as the **administration backend** for web application modules like products, newsletters, blogs, events, accounts, contacts, etc. In fact, thats exactly what it was designed for: managing **structured data** through a nice admin interface, while exposing it as an API for your frontend(s).

Lets map your examples:

Example Modules in Directus

Module	How it works in Directus	Notes
Products (E-shop)	Create a products collection with fields (name, price, description, SKU, images, stock, category relation). Add categories	Directus manages product data, but youd implement checkout, cart, payments in your app (Directus is not an e-commerce

Module	How it works in Directus	Notes
	collection with relation.	engine like Magento/Shopify).
Newsletter	Create a subscribers collection (email, name, preferences). Integrate via API with email providers (e.g. Mailchimp, SendGrid).	Directus manages subscribers + campaigns metadata, external service handles sending.
Blog	Create a posts collection with fields (title, content, slug, author relation, tags). Add categories or tags collections with relations.	Editors can draft/publish posts. API delivers content to frontend.
Events	Create events collection (title, date, time, location, description). Relate to venues, organizers.	Supports filtering, queries by date range, etc.
Accounts & Contacts	Create users or contacts collection with relations to companies/organizations. Use Directus RBAC for access control.	You can separate Directus system users (admins/editors) from application users stored in a custom collection.

What Directus Does Well

- **Data modeling**: Any relational data structure (1-to-1, 1-to-many, many-to-many) can be modeled.
- **Admin UI out of the box**: Non-developers can add/edit products, blog posts, events, contacts without coding.
- **API-first**: Everything you store is instantly available via REST & GraphQL.
- **Permissions**: Fine-grained role/field/row-level access for different teams.
- **Versioning / revisions**: Built-in history of changes to records.
- **Multilingual**: Support for translations in your content.

What Directus Does *Not* Do

- **Business workflows like checkout, payments, order management** youd implement this logic in your app, using Directus just for data.
- **Complex marketing automation (newsletters, push campaigns)** integrate with specialized services.
- **Full CMS templating engine** Directus is headless, so you need to build the frontend (Vue, React, Next.js, etc.).

Positioning

Think of **Directus** as:

- **Admin UI + Database wrapper** it gives you a clean admin for structured data.
- **API layer** REST/GraphQL for your frontend(s).

- **Access control + workflows** manage who can edit what, keep revision history.

So yes: **Products, Newsletter subscribers, Blog, Events, Accounts/Contacts** all very natural to model in Directus.

Do you want me to sketch out a **sample schema** (collections & relations) in Directus for an **e-shop + blog + events** project, so you see how modules map into Directus?

You said:

yes

ChatGPT said:

Perfect Lets design a **sample schema** in **Directus** for your use case:

- **E-Shop (Products, Categories, Orders)**
- **Blog (Posts, Tags, Authors)**
- **Events (Events, Venues, Organizers)**
- **Accounts & Contacts**

This will give you a starting point for modeling your modules.

E-Shop Module

Collections

- products
    - id (primary key)
    - name (text)
    - slug (text, unique)
    - description (rich text)
    - price (decimal)
    - stock (integer)

    - image (file relation)
    - category (many-to-one categories)
    - created_at (datetime)
    - updated_at (datetime)
- categories
    - id
    - name
    - slug
- orders
    - id
    - customer (relation contacts)
    - status (enum: pending, paid, shipped, cancelled)
    - total_amount (decimal)
    - created_at
- order_items
    - id
    - order (many-to-one orders)
    - product (many-to-one products)
    - quantity (integer)
    - price (decimal snapshot at order time)

Blog Module

Collections

- posts
    - id
    - title
    - slug
    - content (rich text or markdown)
    - author (relation users or contacts)
    - tags (many-to-many tags)

    - status (enum: draft, published)
    - published_at (datetime)
- tags
    - id
    - name

Events Module

Collections

- events
    - id
    - title
    - slug
    - description
    - start_date (datetime)
    - end_date (datetime)
    - venue (relation venues)
    - organizer (relation organizers)
    - image (file relation)
- venues
    - id
    - name
    - address
    - city
    - country
- organizers
    - id
    - name
    - contact (relation contacts)

Accounts & Contacts

Collections

- contacts
    - id
    - first_name
    - last_name
    - email
    - phone
    - company (relation companies)
- companies
    - id
    - name
    - address
    - city
    - country

Relations Overview

- Products Categories (many products per category).
- Orders Order Items Products (classic e-commerce order structure).
- Posts Tags (many-to-many).
- Events Venues (many-to-one).
- Events Organizers Contacts (organizer tied to a contact).
- Contacts Companies (many contacts per company).

How it looks in Directus

- Editors will see all collections in the admin panel.
- You can configure **permissions** so, for example:
    - Marketing can manage Blog & Events.
    - Sales team can manage Products & Orders.
    - Support can manage Contacts.
- Everything is exposed automatically via **REST & GraphQL API** for your Vue/React frontend.

Do you want me to also sketch an **example Vue/React frontend flow** that consumes these Directus APIs (e.g. product listing, blog, events calendar)?

You said:

yes

ChatGPT said:

Alright Lets outline how a **Vue/React frontend** would consume your Directus backend (Products, Blog, Events, Contacts).

Directus automatically exposes **REST** and **GraphQL** endpoints. Ill show both, but most people go with REST first because its simpler.

1. Base API Setup

REST endpoint

http://your-domain.com/items/{collection}

Examples:

- Products /items/products
- Blog posts /items/posts
- Events /items/events

GraphQL endpoint

http://your-domain.com/graphql

2. Example: Product Listing (E-shop)

REST (Vue/React)

```
// Vue/React fetch example const res = await
fetch('http://your-domain.com/items/products?fields=id,name,price,image,category.name'); const
products = await res.json(); console.log(products.data);
```

GraphQL

```
query { products { id name price image { id title } category { name } } }
```

---

3. Example: Blog

REST

```
const res = await fetch('http://your-domain.com/items/posts?fields=title,slug,content,tags.name,aut
hor.first_name&filter[status][_eq]=published'); const posts = await res.json();
```

GraphQL

```
query { posts(filter: { status: { _eq: "published" } }) { id title slug content tags { name } author {
first_name last_name } } }
```

---

4. Example: Events Calendar

REST

```
const res = await fetch('http://your-
domain.com/items/events?fields=title,start_date,end_date,venue.name,organizer.name'); const
```

events = await res.json();

GraphQL

query { events { id title start_date end_date venue { name } organizer { name } } }

---

5. Example: Contacts (CRM module)

REST

const res = await fetch('http://your-domain.com/items/contacts?fields=first_name,last_name,email,company.name'); const contacts = await res.json();

GraphQL

query { contacts { id first_name last_name email company { name } } }

---

6. Frontend Flow

In **Vue/React** your flow could be:

- **Products page**
  - GET /items/products
  - Show list when user clicks GET /items/products/:id
- **Blog**
  - GET /items/posts?filter[status][_eq]=published
  - Show list of published posts
  - Blog detail GET /items/posts/:id
- **Events calendar**

- GET /items/events?sort=start_date
- Display in calendar component

- **Contacts / Accounts**
  - Admin-only views in your frontend app (protected by JWT login from Directus).

---

7. Authentication

- Public content (products, blog, events) allow **public role** in Directus with read permissions.
- Private content (contacts, orders, admin functions) require **JWT auth**:

// Login to Directus const res = await fetch('http://your-domain.com/auth/login', { method: 'POST', headers: { 'Content-Type': 'application/json' }, body: JSON.stringify({ email: 'admin@example.com', password: 'yourpassword' }) }); const { data } = await res.json(); const token = data.access_token; // Use token for protected routes const res2 = await fetch('http://your-domain.com/items/contacts', { headers: { Authorization: `Bearer ${token}` } });

Would you like me to prepare a **Vue 3 starter project** that already connects to Directus (products/blog/events) so you can see working components?

You said:

how to use web frontend (vue or react) to call directus backend

ChatGPT said:

Great question You already have Directus running as the backend. Now, lets connect it to a **Vue** or **React** frontend. Ill give you both approaches, and you can pick the framework your team prefers.

---

1. API Options in Directus

Directus exposes:

- **REST API** easier to start with

- **GraphQL API** more flexible for custom queries
- **JavaScript SDK** (@directus/sdk) recommended, works in Vue & React

Ill show you the SDK way (cleaner) and also raw fetch for REST/GraphQL.

---

2. Install Directus SDK

In your frontend project:

npm install @directus/sdk

---

3. Vue Example (Vue 3 + Composition API)

import { ref, onMounted } from 'vue' import { Directus } from '@directus/sdk' // Connect to Directus const directus = new Directus('http://your-domain.com') // State const products = ref([]) onMounted(async () => { // Fetch products from Directus const response = await directus.items('products').readByQuery({ fields: ['id', 'name', 'price', 'image', 'category.name'], limit: 10, }) products.value = response.data })

Products

- {{ p.name }} - {{ p.price }} ({{ p.category.name }})

---

4. React Example (React + Hooks)

import React, { useEffect, useState } from "react"; import { Directus } from "@directus/sdk"; export default function Products() { const [products, setProducts] = useState([]); const directus = new Directus("http://your-domain.com"); useEffect(() => { async function fetchProducts() { const response = await directus.items("products").readByQuery({ fields: ["id", "name", "price", "category.name"], limit: 10, }); setProducts(response.data); } fetchProducts(); }, []); return (Products {products.map((p) => ({p.name} {p.price} {p.category && `(${p.category.name})`}))}); }

---

5. Authentication

For protected endpoints (like orders, contacts), you need a login flow.

// Login const auth = await directus.auth.login({ email: "admin@example.com", password: "yourpassword", }); // Token stored automatically // Now you can fetch private data const contacts = await directus.items("contacts").readByQuery();

You can also persist sessions in **localStorage** or **cookies**.

---

6. REST without SDK (if you prefer)

// Vue/React fetch const res = await fetch('http://your-domain.com/items/products?fields=id,name,price'); const data = await res.json(); console.log(data);

Summary:

- Use @directus/sdk easiest way in Vue/React.
- Public content = no login needed (configure Public role in Directus).
- Private content = authenticate once, then call API with JWT.

Unique solution ID: #1181
Author: n/a
Last update: 2025-11-23 13:58