Expense Application Setup Guide

Complete step-by-step guide for setting up MySQL, Backend (NodeJS), and Frontend (Nginx) servers.

1️⃣ MYSQL DATABASE SERVER

Why MySQL?

MySQL stores all application data (expenses, users, categories). The backend server connects to MySQL to insert and retrieve data.

Step 1: Install MySQL Server

```
bash

dnf install mysql-server -y
```

What this does:

- `dnf` - Package manager for Red Hat/CentOS/Fedora (like apt for Ubuntu)
- `install mysql-server` - Installs the MySQL database server software
- `-y` - Automatically answers "yes" to all prompts

Why we need this: Without MySQL installed, we have no database to store application data. This command downloads and installs all necessary MySQL packages.

Step 2: Enable and Start MySQL Service

```
bash

systemctl enable mysqld
```

What this does:

- `systemctl` - System service manager command
- `enable` - Configures MySQL to start automatically on system boot
- `mysqld` - MySQL daemon (service name)

Why we need this: If the server reboots, MySQL will automatically start. Without this, you'd need to manually start MySQL after every reboot.

```bash
systemctl start mysqld
```

What this does:

- `start` - Starts the MySQL service immediately (right now)

Why we need this: The MySQL service is installed but not running yet. This command starts it so we can begin using the database.

Step 3: Set Root Password

```bash
mysql_secure_installation --set-root-pass ExpenseApp@1
```

What this does:

- `mysql_secure_installation` - MySQL security configuration script
- `--set-root-pass` - Sets the password for the MySQL root user
- `ExpenseApp@1` - The password being set

Why we need this:

- Security: An unprotected database is a major security risk
- Authentication: Backend needs this password to connect to MySQL
- Without a password, anyone can access and modify your data

Step 4: Connect to MySQL

```bash
mysql -h <MYSQL-IP> -u root -pExpenseApp@1
```

What this does:

- `mysql` - MySQL client command-line tool
- `-h <MYSQL-IP>` - Hostname/IP address of MySQL server (use actual IP like 172.31.21.99)

- `-u root` - Username to connect as (root user)
- `-pExpenseApp@1` - Password (note: NO space between -p and password)

Why we need this: To connect from the backend server to the MySQL server. If MySQL is on the same server, you can use `mysql -u root -pExpenseApp@1` without the `-h` flag.

Example:

```bash
mysql -h 172.31.21.99 -u root -pExpenseApp@1
```

Verify Database Installation

```bash
show databases;
```

What this does (inside MySQL prompt): Displays all databases in MySQL server

---

```bash
show tables;
```

What this does (inside MySQL prompt): Shows all tables in the currently selected database

---

2️⃣ BACKEND (NodeJS) SERVER

Why Backend?

The backend is the API service layer that:

- Accepts requests from frontend
- Processes business logic
- Connects to MySQL database
- Stores/retrieves data
- Returns responses to frontend

Step 1: Disable Default NodeJS Module

```bash
dnf module disable nodejs -y
```

What this does:

- `dnf module` - Manages software module streams
- `disable nodejs` - Disables the default NodeJS version provided by system
- `-y` - Auto-confirm

Why we need this: The default NodeJS version might be older (like v16 or v18). We need to disable it first before installing a specific version.

Step 2: Enable NodeJS 20 Module

```bash
dnf module enable nodejs:20 -y
```

What this does:

- `enable nodejs:20` - Enables NodeJS version 20 module stream
- The `:20` specifies the version

Why we need this: The application developer specified that the backend requires NodeJS version 20 or higher for compatibility with dependencies and features.

Step 3: Install NodeJS

```bash
dnf install nodejs -y
```

What this does:

- Installs NodeJS version 20 (the version we just enabled)
- Also installs npm (Node Package Manager)

Why we need this: NodeJS is the runtime environment needed to execute the backend JavaScript code.

Step 4: Update SSH Packages

```bash
dnf update -y openssh openssh-server openssh-clients
```

What this does:

- `update` - Updates existing packages to latest versions
- `openssh*` - All SSH-related packages

Why we need this:

- Prevents SSH connection issues: When installing packages, SSL/SSH libraries might update, causing version mismatches
- Security: Keeps SSH components secure with latest patches
- Stability: Ensures you don't get locked out after package installations or reboots
- Without this, your SSH connection might break after system updates

---

Step 5: Create Application User

```bash
useradd expense
```

What this does:

- `useradd` - Creates a new system user
- `expense` - Username for the application

Why we need this:

- Security: Never run applications as root user (root has unlimited permissions)
- Isolation: If the app is compromised, damage is limited to the expense user
- Best Practice: Each application should have its own user account

---

Step 6: Create Application Directory

```bash
```

```
mkdir /app
```

What this does:

- `mkdir` - Make directory command
- `/app` - Creates folder at root level called "app"

Why we need this:

- Standard Practice: Keeps application code in one dedicated location
- Organization: Separates app code from system files
- Clean Structure: Not mixed with user home directories
- Easy Management: All developers know where to find the application

---

Step 7: Download Backend Code

```
bash

curl -o /tmp/backend.zip https://expense-joindevops.s3.us-east-1.amazonaws.com/expense-backend-v2.zip
```

What this does:

- `curl` - Command-line tool to download from internet
- `-o /tmp/backend.zip` - Output (save) the file as this name
- `URL` - Location of the backend code zip file on AWS S3

Why we need this:

- Downloads the actual application code
- `/tmp` is temporary storage (clean location for downloads)
- Best practice: Download first, then extract to final location

---

Step 8: Extract Backend Code

```
bash

cd /app
```

What this does:

- Changes current directory to /app

```bash
unzip /tmp/backend.zip
```

What this does:

- `unzip` - Extracts zip archive
- Extracts contents of backend.zip into current directory (/app)

Why we need this:

- Places all backend source code files into /app directory
- After extraction, /app contains: index.js, package.json, schema/, etc.

Folder structure after extraction:

```
/app/
  ├── index.js
  ├── package.json
  ├── package-lock.json
  └── schema/
    └── backend.sql
```

Step 9: Install Backend Dependencies

```bash
cd /app
```

What this does:

- Ensures we're in the /app directory

```bash
npm install
```

What this does:

- `npm install` - Reads package.json and installs all required libraries
- Downloads dependencies into /app/node_modules/

Why we need this:

- Backend needs external libraries (Express, MySQL driver, etc.)
- These dependencies are listed in package.json
- Without this, the application won't run (missing required modules)

Folder structure after npm install:

```
/app/
  ├── index.js
  ├── package.json
  ├── package-lock.json
  ├── schema/
  │   └── backend.sql
  └── node_modules/     ← New folder with libraries
```

Step 10: Create Systemd Service File

```
bash

vim /etc/systemd/system/backend.service
```

What this does:

- Opens vim editor to create a new service file
- `/etc/systemd/system/` - Location for custom service definitions

Content to add:

```
ini
```

```
[Unit]
Description = Backend Service

[Service]
User=expense
Environment=DB_HOST="<MYSQL-SERVER-IPADDRESS>"
ExecStart=/bin/node /app/index.js
SyslogIdentifier=backend

[Install]
WantedBy=multi-user.target
```

Why we need this: Creates a system service so backend runs automatically and can be controlled with systemctl commands (start/stop/restart).

Explanation of each line:

`[Unit]` Section:

- Metadata about the service

`Description = Backend Service`

- Human-readable description of what this service does

`[Service]` Section:

- Defines how the service runs

`User=expense`

- Runs the backend process as the "expense" user (not root)
- Security: Limits damage if application is compromised

`Environment=DB_HOST="<MYSQL-SERVER-IPADDRESS>"`

- Sets environment variable DB_HOST
- Backend code reads this to know where MySQL is located
- **Replace with actual IP:** `Environment=DB_HOST="172.31.21.99"`
- Backend uses this to connect: `mysql -h ${DB_HOST}`

`ExecStart=/bin/node /app/index.js`

- Command to start the service
- `/bin/node` - NodeJS executable

- `/app/index.js` - Main backend application file

`SyslogIdentifier=backend`

- Tags log messages with "backend" identifier
- Makes filtering logs easier: `journalctl -u backend`

`[Install]` Section:

- Defines when service should start

`WantedBy=multi-user.target`

- Start automatically during system boot
- multi-user.target = normal system startup mode

Step 11: Reload Systemd Daemon

```
bash

systemctl daemon-reload
```

What this does:

- Tells systemd to reload its configuration
- Scans for new or modified service files

Why we need this: After creating a new service file, systemd doesn't automatically know about it. This command makes systemd recognize the new backend.service file.

Step 12: Start Backend Service

```
bash

systemctl start backend
```

What this does:

- Starts the backend service immediately

Why we need this: Launches the backend application so it begins listening for API requests.

Step 13: Enable Backend Service

```
bash

systemctl enable backend
```

What this does:

- Configures backend to start automatically on boot

Why we need this: If the server reboots, backend will start automatically. Without this, you'd need to manually start it after every reboot.

Step 14: Load Database Schema

```
bash

mysql -h <MYSQL-IP> -uroot -pExpenseApp@1 < /app/schema/backend.sql
```

What this does:

- `mysql -h <MYSQL-IP> -uroot -pExpenseApp@1` - Connects to MySQL
- `<` - Redirects file content as input
- `/app/schema/backend.sql` - SQL file containing database structure

Why we need this:

- MySQL is empty after installation (no databases or tables)
- This command executes SQL statements to create:
  - Database
  - Tables (expenses, users, categories)
  - Table structure (columns, data types)
- Without this: Backend will fail with "table not found" errors

Example:

```
bash

mysql -h 172.31.21.99 -uroot -pExpenseApp@1 < /app/schema/backend.sql
```

What happens inside: The SQL file contains commands like:

```sql
CREATE DATABASE expense;
USE expense;
CREATE TABLE expenses (...);
CREATE TABLE users (...);
```

3️⃣ FRONTEND (Nginx) SERVER

Why Frontend + Nginx?

- Frontend contains the UI (HTML, CSS, JavaScript files)
- Nginx serves as:
  - Web server (hosts static files)
  - Reverse proxy (forwards API calls to backend)

Step 1: Install Nginx

```bash
dnf install nginx -y
```

What this does:

- Installs Nginx web server
- `-y` - Auto-confirm installation

Why we need this: Nginx is the web server that hosts and serves the frontend application to users' browsers.

Step 2: Enable Nginx Service

```bash
systemctl enable nginx
```

What this does:

- Configures Nginx to start automatically on boot

Why we need this: Ensures the website comes back online automatically after server reboots.

Step 3: Start Nginx Service

```bash
systemctl start nginx
```

What this does:

- Starts Nginx web server immediately

Why we need this: Nginx is installed but not running. This starts the web server so it can begin serving web pages.

Step 4: Remove Default Nginx Content

```bash
rm -rf /usr/share/nginx/html/*
```

What this does:

- `rm` - Remove command
- `-rf` - Recursive (folders) and force (no confirmation)
- `/usr/share/nginx/html/*` - All files in Nginx's web root directory
- `*` - Wildcard (all files and folders)

Why we need this:

- Nginx comes with default "Welcome to Nginx" page
- We remove it to avoid conflicts with our application
- Ensures only our Expense app is displayed
- Clears space for our frontend files

Step 5: Download Frontend Code

```bash
```

```
curl -o /tmp/frontend.zip https://expense-joindevops.s3.us-east-1.amazonaws.com/expense-frontend-v2.zip
```

What this does:

- Downloads frontend zip file from AWS S3
- Saves to /tmp/frontend.zip

Why we need this: Downloads the actual frontend UI files (HTML, CSS, JS).

Step 6: Navigate to Nginx Web Directory

```
bash

cd /usr/share/nginx/html
```

What this does:

- Changes to Nginx's default web root directory

Why we need this: This is where Nginx looks for files to serve. We need to extract frontend files here.

Step 7: Extract Frontend Code

```
bash

unzip /tmp/frontend.zip
```

What this does:

- Extracts frontend.zip into current directory (/usr/share/nginx/html)

Why we need this: Places all frontend files (index.html, CSS, JS) where Nginx can serve them.

Files after extraction:

```
/usr/share/nginx/html/
  ├── index.html
  ├── css/
  ├── js/
  └── images/
```

Step 8: Create Reverse Proxy Configuration

```
bash

vim /etc/nginx/default.d/expense.conf
```

What this does:

- Creates a new Nginx configuration file
- `/etc/nginx/default.d/` - Directory for additional Nginx configs

Content to add:

```
nginx

proxy_http_version 1.1;

location /api/ {
   proxy_pass http://localhost:8080/;
}

location /health {
   stub_status on;
   access_log off;
}
```

Why we need this: Configures Nginx to forward API requests to the backend server.

Explanation of each section:

`proxy_http_version 1.1;`

- Sets HTTP version for proxy connections
- HTTP/1.1 supports keep-alive connections (better performance)

`location /api/ { ... }`

- Matches any URL starting with `/api/`
- Example: `http://frontend-ip/api/expenses` matches this

`proxy_pass http://localhost:8080/;`

- Forwards the request to backend server

- `localhost:8080` - Backend runs on same server, port 8080
- Acts as a middleman between frontend and backend

Flow example:

1. User's browser: `GET http://website.com/api/expenses`
2. Nginx receives request
3. Nginx forwards to: `http://localhost:8080/expenses`
4. Backend processes and returns data
5. Nginx sends response back to browser

If backend is on different server: Replace `localhost` with backend server's private IP:

```
nginx

proxy_pass http://172.31.45.89:8080/;
```

`location /health { ... }`

- Matches URL: `http://frontend-ip/health`

`stub_status on;`

- Enables Nginx status page
- Shows: active connections, requests, etc.
- Used for monitoring server health

`access_log off;`

- Disables logging for /health endpoint
- Prevents log flooding from monitoring tools
- Monitoring systems check /health frequently (every few seconds)

Step 9: Restart Nginx

```
bash

systemctl restart nginx
```

What this does:

- Stops and starts Nginx service
- Applies the new configuration from expense.conf

Why we need this: Nginx needs to be restarted to load the reverse proxy configuration. Without restart, the proxy rules won't work.

🎯 Complete Application Flow

User Opens Website:

User Browser → http://frontend-ip → Nginx (port 80) → Serves HTML/CSS/JS

UI Makes API Call:

Browser → http://frontend-ip/api/expenses → Nginx → Forwards to Backend (port 8080)

Backend Processes Request:

Backend (NodeJS) → Connects to MySQL (port 3306) → Queries database → Returns data

Response Flow:

MySQL → Backend → Nginx → Browser → User sees data

📌 Quick Reference

Check Service Status:

bash

```
systemctl status mysqld
systemctl status backend
systemctl status nginx
```

View Logs:

bash

```
journalctl -u mysqld -f
journalctl -u backend -f
journalctl -u nginx -f
```

Test Connectivity:

```bash
# Test MySQL connection
mysql -h <MYSQL-IP> -uroot -pExpenseApp@1

# Test backend
curl http://localhost:8080/api/health

# Test frontend
curl http://localhost/
```

⚠️ Important Notes

1. **Replace placeholders:**
   - `<MYSQL-IP>` with actual MySQL server IP (e.g., 172.31.21.99)
   - `<BACKEND-IP>` with actual backend server IP if on different server
2. **Security Groups:**
   - MySQL: Allow port 3306 from backend server
   - Backend: Allow port 8080 from frontend server
   - Frontend: Allow port 80 from internet (0.0.0.0/0)
3. **Firewall (if enabled):**

```bash
firewall-cmd --permanent --add-port=3306/tcp  # MySQL
firewall-cmd --permanent --add-port=8080/tcp  # Backend
firewall-cmd --permanent --add-port=80/tcp    # Frontend
firewall-cmd --reload
```

End of Guide