Table of Contents

JPay - Payment Gateway Information

1. Home

JPay is a secure and user-friendly payment gateway designed to simplify online transactions.
With JPay, businesses and individuals can process payments seamlessly and efficiently.
Our platform offers robust security measures, ensuring the safety of your transactions.
Key features of JPay include:
- Easy integration with websites and applications.
- Comprehensive support for various payment methods, including credit/debit cards and mobile wallets.
- Real-time transaction tracking and analytics.
- Dedicated support to help you set up and manage your account.

2. About

JPay was established with the vision of revolutionizing online payment systems. Our mission is to
provide a hassle-free payment gateway solution that caters to businesses of all sizes. At JPay,
we prioritize security, efficiency, and ease of use.
Why choose JPay?
- Trusted by thousands of merchants worldwide.
- Cutting-edge encryption technologies to safeguard your data.
- 24/7 customer support and a dedicated team of payment experts.

3. Contact

For any inquiries, please reach out to us:
- Email: support@jpay.com
- Phone: +123-456-7890
- Address: 123 Payment Street, Gateway City, Paymentland
Our services include:
- Merchant onboarding and training.
- Technical support for integration.
- Assistance with payment disputes and chargebacks.

4. Help

Need help? JPay provides a comprehensive knowledge base and dedicated support team to address your concerns.
Some common topics include:
- How to set up your account.
- Troubleshooting failed transactions.
- Updating payment methods.
Visit our Help Center or contact our support team for assistance.

5. Login

The login functionality allows registered users to securely access their accounts.
Steps to login:
1. Enter your registered ID, mobile number, or email address.
2. Enter your password.
3. Click on the "Login" button.
If your credentials are correct, you will be redirected to the merchant dashboard, where you can manage
your transactions and account details. In case of an error, an appropriate message will be displayed.

6. Register

To register for JPay, please provide the following details:
- Email: Enter a valid email address.
- Password: Create a secure password.
- Mobile: Provide your mobile number for verification.
- Do you want to collect payments on your website?: Check "Yes" or "No".
- Website URL: If applicable, enter your website URL.
Additional Steps:
1. Click on "Send OTP" to receive a One-Time Password for verification.
2. Enter the OTP and click on "Create Account" to complete the registration process.
Once registered, you can log in and start using JPay's services.

Step2

Detailed Explanation of Login Flow

This document describes the login flow process for accessing the merchant's dashboard:

1. **Navigation to Login Page**
   The user begins the login process by clicking the "Login Nav Button," which redirects them to the Login Page.
2. **Authentication on the Login Page**
   On the Login Page, users provide credentials such as a username and password.
   - If the credentials are valid, the user is authenticated and logged in.
   - If the credentials are invalid, the user is redirected to a JWT error page with an appropriate error message.
3. **Redirect to Merchant's Dashboard**
   Upon successful authentication, the user is redirected to the Merchant's Dashboard. A JWT token is passed to the dashboard to grant access to protected resources and ensure secure client-server communication.
4. **Error Handling**
   If authentication fails (e.g., invalid credentials), the user is redirected to a JWT error page. This page informs the user about the error and allows them to retry the login process.

Step3

Detailed Explanation of Token Handling

1. **Authentication Request**
   The client (e.g., browser or mobile app) initiates an authentication request to the Authentication Server. This request includes credentials such as a username and password or an API key to verify the user's identity.
2. **Access Token Issuance**
   Upon successful authentication, the Authentication Server issues an Access Token to the client. This token acts as a temporary credential to access protected resources.
3. **Accessing Protected Resources**
   The client uses the Access Token to request data or services from the Resource Server. The server verifies the token's validity before granting access to the requested resources.
4. **Refresh Token Mechanism**
   When the Access Token expires, the client requests a new Access Token from the Authentication Server using a Refresh Token. The Refresh Token, issued along with the Access Token, ensures uninterrupted access to protected resources without requiring re-login.

Additional Details:

- The client sends its credentials to the `/api/login_check` endpoint to obtain an Access Token and a Refresh Token.
- When the Access Token expires, the client sends a request to `/api/refresh/token` with the Refresh Token.
- The server validates the Refresh Token and issues a new Access Token and a refreshed Refresh Token.
  This process ensures secure and efficient token handling.

Step4

Detailed Explanation of JSON Web Token (JWT) Structure

This document provides a detailed explanation of the structure of a JSON Web Token (JWT) based on the provided diagram. A JWT is a compact, URL-safe token that represents claims between two parties. It is commonly used for authentication and securely transmitting information.

1. Header

The Header contains metadata about the token, including the type of token (JWT) and the hashing algorithm used to secure the token. For example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

In this example:
- `alg` specifies the algorithm (HMAC SHA-256).
- `typ` specifies the type of token (JWT).

2. Payload

The Payload contains claims or statements about an entity (typically the user) and additional data. Claims are typically divided into three types:
- Registered claims: Predefined claims such as `iss` (issuer), `exp` (expiration time), `sub` (subject), and `iat` (issued at).

- Public claims: Custom claims defined by the user.
- Private claims: Claims shared between parties.

Example Payload:
{
"sub": "1234567890",
"name": "George White",
"admin": true,
"iat": 1516239022
}

In this example:
- `sub`: Subject identifier (user ID).
- `name`: Name of the user.
- `admin`: Indicates whether the user has admin privileges.
- `iat`: Timestamp when the token was issued.

3. Signature

The Signature is used to verify the authenticity of the token and ensure that it has not been tampered with. It is created by taking the encoded Header, the encoded Payload, a secret key, and the algorithm specified in the Header. The resulting string is then Base64URL encoded. For example:

HMACSHA256(
base64UrlEncode(header) + '.' + base64UrlEncode(payload),
secret
)

The Signature ensures the integrity and authenticity of the token. If any part of the token is altered, the signature will no longer match.

4. Complete JWT

A complete JWT is composed of three Base64URL encoded parts separated by periods (`.`):
1. Header
2. Payload
3. Signature
Example:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikdlb3JnZSBXaGl0ZSIsImFkbWluIjp0cnVlLCJpYXQiOjE1MTYyMzkwMjJ9.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Each part is decoded as follows:
- **Header**: {"alg":"HS256","typ":"JWT"}
- **Payload**: {"sub":"1234567890","name":"George

White","admin":true,"iat":1516239022}
- **Signature**: Cryptographic signature ensuring the token's integrity.

Conclusion

The JSON Web Token (JWT) structure provides a secure way to transmit information between parties. Its three-part structure (Header, Payload, and Signature) ensures the integrity and authenticity of the data, making it an essential tool for modern authentication systems.

Step 6

Flowchart Description: Node.js Application Interacting with MongoDB

This flowchart outlines the data flow and operational steps in a Node.js application that interacts with a MongoDB database.

1. User Interaction with Web/User Application

- A user interacts with the application via a web interface or user application.
- The application generates a token containing the data payload and sends it to the backend.
- This data is forwarded to the Node.js server for processing.

2. Data Decryption and Validation by Node.js

- The Node.js server receives the encrypted data and decrypts it.
- The server processes the request through the schema, route, and controller layers.
- It validates the data as part of the second step in the authentication process.

3. Database Interaction via Mongoose

- After validation, the server performs one of the following operations on the MongoDB database:

- Store: Inserts new data into the database.

- Fetch: Retrieves data from the database.

- Update: Modifies existing data in the database.

- The server uses Mongoose to access and manipulate specific collections and documents.

4. Error Handling

- If data validation fails, the Node.js server sends an error response to the user application.
- This step ensures that invalid or unauthorized requests are not processed further.

5. Dynamic Collection and Schema Management

- If the required collections do not exist in the database:

- The Node.js server creates the necessary collections.

- It also handles schema creation, updates, and deletion of JSON data as needed.

6. MongoDB Database

- The MongoDB database serves as the data storage layer.
- It organizes data into collections and documents.

7. Collections

- Collections are groups of related data records in the database.
- Represented as vertical rectangles in the flowchart.

8. Documents

- Documents are individual data records within a collection.
- Represented as horizontal rectangles within the corresponding collection.

9. Successful Interaction

- If the database operation is successful, the flow proceeds.
- The processed data is delivered back to the web or user application for further use.

This flowchart highlights the systematic flow of operations, from user interaction to data storage, retrieval, and processing, ensuring efficient and secure data management in a Node.js application interacting with MongoDB.

Step7

MONGODB WORKING ARCHITECHTURE

MongoDB is a NoSQL database that stores data in a flexible, document-oriented way, unlike traditional relational databases. Here's an explanation of the architecture components:

MongoDB Database

A MongoDB database is a container that holds collections. It's the top-level unit in MongoDB where all the data resides. You can have one or more databases in a MongoDB instance. Each database is isolated and independent from others, meaning collections in one database can't directly reference those in another.

Collections

A **collection** is a group of documents in MongoDB. Think of it as equivalent to a table in a relational database. Collections are not rigidly defined with a schema, meaning each document can have a different structure. Collections reside within a database, and each database can contain multiple collections. For example, in an e-commerce application, you might have collections like `users`, `orders`, and `products` within a database.

Documents

A **document** is the basic unit of data in MongoDB. Each document is a set of key-value pairs, similar to a dictionary in programming languages like Python or JSON objects. A document can also include arrays or even nested documents (objects within objects). Documents are the data that you store in MongoDB, and they reside within collections.

Key Characteristics of Documents:

- **Flexible Structure:** MongoDB allows you to store documents with varying structures, unlike relational databases that require a predefined schema.
- **Key-Value Pairs:** Each document contains fields (keys) and values. The values can be simple data types (strings, numbers), arrays, or even other embedded documents.

Step8

VPS Hosting Explained

Physical Server

It represents a dedicated physical server that is powerful and capable of hosting multiple virtual servers.

Virtual Servers

These are the virtual servers that are created and hosted on the physical server. They can be used for various purposes, such as running websites, applications, and databases.

VPS Functionality

Each VPS can be configured with different ports and functionalities, allowing users to tailor them to their specific needs.

Website Hosting

The image shows three websites hosted on different VPS servers, highlighting the ability of each VPS to host multiple websites.

Web Server Installation

After creating a VPS, a web server software (like Apache or Nginx) needs to be installed on the VPS to enable website hosting.

VPS Port IP

Each VPS has its own unique IP address and can be configured with different ports to run specific services.

Benefits of VPS Hosting

Shared Resources

VPS hosting offers a dedicated environment, although it shares some resources with other VPSs on the same physical server.

Flexibility

VPS allows for customization, offering more control over resources and configuration.

Scalability

VPS hosting offers the ability to upgrade resources as your needs grow.

Step9

Nginx (pronounced as "Engine-X") is a high-performance web server and reverse proxy server, widely used for handling web traffic. It operates based on an asynchronous event-driven architecture, which allows it to serve static and dynamic content efficiently. Here's an overview of its structure and workflow:

Nginx Structure

Nginx is typically divided into several key components:

- **Worker Processes**: These are the processes that handle incoming requests. Nginx uses multiple worker processes to handle multiple requests concurrently. Each worker is responsible for handling multiple connections using non-blocking I/O.
- **Master Process**: The master process is responsible for managing the worker processes. It controls the configuration of the Nginx instance, including starting, stopping, and managing worker processes.
- **Configuration File**: Nginx's behavior is controlled by a central configuration file, usually located at `/etc/nginx/nginx.conf`. The file is divided into different contexts like:
    - **http**: Defines HTTP-specific settings.
    - **server**: Defines settings for individual server blocks (virtual hosts).
    - **location**: Defines how to handle requests for specific resources.
- **Modules**: Nginx uses modules to extend its functionality. These modules can be dynamically loaded or compiled into the Nginx binary, handling tasks like SSL termination, proxying, load balancing, and more.

Nginx Workflow

Nginx processes incoming requests in the following steps:

- **Step 1: Request Handling**
    - A client (browser or other service) sends a request to the Nginx server.
    - Nginx receives the request in one of its worker processes.

- **Step 2: Request Matching**
  - Nginx looks at the request and matches it against the configurations defined in the `server` block.
  - If the request matches a defined domain or IP address, it proceeds to find a matching location block.
- **Step 3: Location Matching**
  - Nginx checks the request against `location` blocks to determine how to handle it. The `location` block defines how specific types of content (URLs, file types, etc.) should be handled.
- **Step 4: Content Serving**
  - If the request is for static content (e.g., HTML, images), Nginx serves it directly from the file system.
  - If the request is for dynamic content (e.g., PHP, Python), Nginx will forward the request to a backend application server (like PHP-FPM, uWSGI) using a reverse proxy setup.
- **Step 5: Proxying (if applicable)**
  - If Nginx is acting as a reverse proxy, it forwards the request to the appropriate backend server (e.g., an application server or another web server).
  - The backend processes the request, generates a response, and sends it back to Nginx.
- **Step 6: Response Handling**
  - Nginx receives the response from the backend or serves the content itself if it's static.
  - Nginx can then apply further transformations or manipulations (e.g., compression, caching).
- **Step 7: Response Delivery**
  - Nginx sends the response back to the client over the network. If caching is enabled, the response may be stored for future requests.

Additional Features

- **Load Balancing**: Nginx can distribute incoming traffic across multiple backend servers using round-robin, least connections, or other algorithms.
- **Caching**: Nginx can cache content, reducing the need for repeated requests to backend servers for static or dynamic content.
- **SSL Termination**: Nginx can manage SSL/TLS encryption, offloading the SSL handshake and encryption from backend servers.
- **Security**: Nginx provides features like access control, rate limiting, and IP filtering to protect servers from malicious requests.
- **Logging**: Nginx logs all requests to access and error logs, providing detailed insights into server activity.

Step10

NGINX Reverse Proxy with Dynamic IP Upstream

A **reverse proxy** is a server that sits between client devices and a web server, intercepting requests from clients and forwarding them to the appropriate backend server. **NGINX**, one of the most popular open-source web servers, is widely used to implement reverse proxies. When combined with **dynamic IP upstream**, it can handle requests even when the backend server's IP address changes dynamically (for instance, in cloud environments or when the backend server has a dynamic IP address). Here's a detailed explanation of how NGINX reverse proxy works with dynamic IP upstream.

Understanding Reverse Proxy

Reverse Proxy: A reverse proxy accepts incoming client requests and forwards them to a backend server. It hides the real identity and IP address of the backend server, which is useful for load balancing, security, or caching. With a reverse proxy in place, users never directly connect to the backend servers but instead interact with the reverse proxy.

NGINX Reverse Proxy: NGINX is a highly efficient reverse proxy due to its ability to handle many concurrent connections with low resource consumption. It serves as an intermediary between users and backend servers, forwarding requests based on rules configured in the NGINX server block.

Dynamic IP Upstream

Dynamic IP Address: In many modern infrastructures (e.g., cloud environments or home networks), the IP address of the backend server may change over time. This is especially true when servers are dynamically assigned IP addresses (as opposed to static ones). For example, cloud providers may change an instance's IP when it is restarted.

Dynamic IP Upstream in NGINX: NGINX can be configured to forward requests to a dynamic upstream server, even if its IP address changes. NGINX achieves this by resolving the domain name of the backend server at the time of the request or at regular intervals, ensuring that the IP address always points to the correct server.

Connecting Public Domain DNS to Dynamic IP

A **Public Domain DNS** (Domain Name System) is a system that maps a human-readable domain name (like `example.com`) to an IP address. When using dynamic IPs, the DNS record for the domain can be updated automatically when the backend server's IP changes, thus ensuring that the domain always points to the correct server.

The domain DNS needs to be configured to resolve to the correct public IP address, which may point to a load balancer or reverse proxy that uses NGINX.

Port Assignment: When a reverse proxy is set up, it forwards traffic from the public domain (e.g., `example.com:80` or `example.com:443`) to a backend server listening on a specific port. In NGINX, this is set up in the configuration file using the `proxy_pass`

directive, where requests to a domain and port (like `http://example.com:80`) are forwarded to the backend server on the desired port.

NGINX Configuration Example

Here's an example of how to set up NGINX as a reverse proxy for a dynamic IP upstream:

nginx

Copy code

```
http {

    upstream backend {

        # NGINX will resolve the domain dynamically

        server backend.example.com:80;

        # Use multiple servers if needed, NGINX will balance load

        server backend2.example.com:80;

    }


    server {

        listen 80;

        server_name example.com;


        location / {

            proxy_pass http://backend;  # Forward requests to the
backend

            proxy_set_header Host $host;  # Pass original Host header

            proxy_set_header X-Real-IP $remote_addr;  # Pass client's
IP

            proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;

        }

    }

}
```

upstream backend: This block configures NGINX to dynamically resolve the IP address of `backend.example.com`. If the IP address of the server changes, NGINX will resolve the domain name again to get the new IP.

proxy_pass http://backend: This tells NGINX to forward requests to the upstream backend.

proxy_set_header directives: These pass necessary headers to the backend server, such as the original host and the client's IP.

Handling DNS and IP Changes

DNS TTL (Time to Live): The TTL value determines how long DNS records are cached. With a short TTL, NGINX will frequently re-resolve the DNS to check for any changes in the backend server's IP address.

DNS Resolution: NGINX periodically checks the DNS records for the upstream servers, so if the backend IP changes, NGINX will automatically use the new IP without needing to restart.

Benefits of Using NGINX Reverse Proxy with Dynamic IP Upstream

Flexibility: It allows the use of dynamic IP addresses for backend servers while still providing reliable reverse proxy functionality.

Load Balancing: NGINX can distribute incoming requests across multiple backend servers (even with dynamic IPs), providing high availability and load balancing.

Security: By using a reverse proxy, the backend server's IP is hidden from the client, improving security.

Scalability: As the number of backend servers grows, NGINX can be configured to handle multiple dynamic upstreams, enabling seamless scaling of the infrastructure.

Challenges

DNS Propagation Delays: If the DNS record changes frequently, clients may experience a delay in resolving the new IP address due to caching.

Dynamic IP Handling: If the backend server is restarted and the IP changes but DNS resolution is slow, the reverse proxy may temporarily fail to connect to the new IP. Configuring a low TTL can help reduce this issue.

Step11

DNS SSL Connection to a VPS Server Port

A **DNS SSL connection** to a VPS (Virtual Private Server) refers to securely connecting a domain (DNS) to a server over an encrypted SSL/TLS connection, typically through HTTPS, on a specific port. This ensures that data transmitted between the client (browser) and the server remains private and secure. Here's a step-by-step breakdown of how DNS, SSL, and port configurations work together to establish a secure connection to a VPS:

Understanding the Components

DNS (Domain Name System): DNS translates human-readable domain names (like `example.com`) into IP addresses that computers use to locate each other on the internet. When a client enters a domain name into their browser, the DNS resolves it to an IP address, allowing the browser to connect to the server.

SSL/TLS (Secure Socket Layer/Transport Layer Security): SSL and its successor, TLS, are cryptographic protocols designed to provide secure communication over a computer network. When used for websites, SSL/TLS encrypts data between the client (browser) and the server, ensuring the confidentiality and integrity of data, such as login credentials, financial transactions, etc.

VPS (Virtual Private Server): A VPS is a virtualized server hosted on a physical machine. It's a private server environment that allows users to install software and manage services independently, offering more control than shared hosting but at a lower cost than dedicated servers.

Port: A port is a logical connection point used by the operating system to manage incoming and outgoing network traffic. By default, HTTPS traffic (which uses SSL/TLS) operates on port **443**.

Steps for Setting Up DNS SSL Connection to a VPS Server Port

Step 1: Configure DNS Records

First, you need to ensure that your domain points to your VPS server's IP address. This is done by setting up **A records** or **CNAME records** in your domain's DNS settings.

The **A record** points your domain (e.g., `example.com`) to the public IP address of your VPS.

Example of an A record:

rust

Copy code

```
example.com -> 192.168.1.100 (VPS IP)
```

If you're using subdomains, you may also need to create additional DNS records (e.g., `www.example.com` or `sub.example.com`).

Step 2: Install SSL/TLS Certificate on Your VPS

Obtain SSL Certificate: To establish a secure connection, you need an SSL/TLS certificate for your domain. You can obtain one from a trusted Certificate Authority (CA) like **Let's Encrypt** (free), **Comodo**, or **Symantec**.

Install SSL Certificate: After obtaining the SSL certificate, you need to install it on your VPS. This involves placing the certificate files (public and private keys) in specific directories and configuring your web server to use them.

For **NGINX** or **Apache**, this typically involves modifying the configuration files to point to the certificate files.

Example for NGINX configuration:

nginx

Copy code

```
server {
    listen 443 ssl;
    server_name example.com;

    ssl_certificate /etc/nginx/ssl/example.com.crt;
    ssl_certificate_key /etc/nginx/ssl/example.com.key;

    # Other SSL settings for stronger security (optional)
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers 'TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384';

    location / {
        root /var/www/html;
        index index.html;
    }
}
```

Step 3: Configure Your VPS Firewall to Allow Port 443

Your VPS should allow inbound traffic on port 443, which is the default port for HTTPS connections. This can be configured through the firewall settings on the VPS.

For example, if you're using **UFW** (Uncomplicated Firewall) on a Linux VPS, you would run:

bash

Copy code

```
sudo ufw allow 443/tcp
```

Step 4: Bind SSL to Your Web Server (Port 443)

After configuring SSL, you need to ensure your web server is set up to accept traffic on port 443.

For **NGINX** or **Apache**, the web server should be configured to listen on port 443, as shown in the earlier configuration example for NGINX.

Step 5: Redirect HTTP Traffic to HTTPS (Optional but Recommended)

It's a good practice to redirect all HTTP traffic (on port 80) to HTTPS (on port 443) to ensure secure connections by default.

This can be done by adding a redirect rule in the server configuration.

Example for NGINX:

nginx

Copy code

```
server {
    listen 80;
    server_name example.com;
    return 301 https://$host$request_uri;
}
```

Step 6: Verify SSL Connection

Once everything is set up, test the SSL connection by visiting your domain with `https://`. Your browser should show a secure connection with a padlock icon.

You can also use online tools like **SSL Labs** (by Qualys) to check your server's SSL/TLS setup for security and proper configuration.

How the DNS SSL Connection Works

DNS Resolution: When a user enters `https://example.com` in their browser, the browser first queries the DNS to resolve the domain to the VPS server's IP address.

SSL Handshake: After the DNS resolution, the browser initiates a connection to the server's IP on port 443. During this process, an **SSL handshake** occurs, which involves:

The server sending its SSL certificate to the client.

The client verifying the certificate's authenticity using the CA's public key.

Establishing a shared encryption key for secure communication.

Encrypted Communication: Once the SSL handshake is complete, the communication between the client and the VPS server is encrypted, ensuring data integrity and privacy.

Benefits of DNS SSL Connection to VPS Port

Security: SSL/TLS encryption ensures that data sent between the client and the server is private and cannot be intercepted or tampered with.

Trust and SEO: SSL/TLS is crucial for user trust, especially for e-commerce or login sites. Google also ranks HTTPS sites higher in search results.

Compliance: Many industries and regulations (such as GDPR) require secure data transmission, and using SSL ensures compliance.

Common Issues

Expired SSL Certificate: If the SSL certificate expires, users will see a warning in the browser. Ensure to renew it before expiration.

Mixed Content: This occurs when a website loads some resources (e.g., images or scripts) over HTTP instead of HTTPS. This can cause browsers to block those resources or display warnings.

Step12

Configuring PM2 to Work with NGINX

PM2 is a process manager for Node.js applications that helps keep applications alive forever, manage their processes, and monitor them. It's commonly used to run Node.js applications in production environments. **NGINX**, on the other hand, is a high-performance web server and reverse proxy server that can distribute traffic to multiple backend services (e.g., Node.js applications) running on different ports.

In this context, PM2 is used to run Node.js applications, while NGINX is used as a reverse proxy to handle incoming web traffic and forward it to the appropriate Node.js application managed by PM2. This configuration improves scalability, security, and performance. Below is a step-by-step guide on how to configure PM2 and NGINX to work together.

Install PM2 and Start the Node.js Application

Install PM2

First, ensure that **Node.js** and **npm** (Node Package Manager) are installed on your VPS server. Then install **PM2** globally using npm:

```
bash
Copy code
sudo npm install -g pm2
```

Start Your Node.js Application with PM2

You can now start your Node.js application using PM2. For example, if your application is located in `/var/www/myapp`, you can start it like this:

```
bash
Copy code
cd /var/www/myapp
pm2 start app.js --name "myapp"
```

- `app.js` is the entry point of your Node.js application (replace it with your app's main file if different).
- `--name "myapp"` assigns a name to the process, making it easier to manage.

PM2 will keep the application running in the background and ensure that it remains online even after system restarts.

Check the Application Status

To confirm that your Node.js application is running, you can check the PM2 status:

```
bash
Copy code
pm2 status
```

This will show the current status of all running applications under PM2's management.

Set Up NGINX as a Reverse Proxy for Your Node.js Application

NGINX will forward incoming HTTP(S) requests to your Node.js application running under PM2. Below are the steps for configuring NGINX.

Install NGINX

If NGINX isn't installed on your VPS, you can install it by running:

```
bash
Copy code
sudo apt update
sudo apt install nginx
```

Configure NGINX for Reverse Proxy

You need to create a configuration file for your application in the `/etc/nginx/sites-available/` directory and then link it to `/etc/nginx/sites-enabled/`.

Create a new configuration file for your app (e.g., `myapp`):

```
bash
Copy code
sudo nano /etc/nginx/sites-available/myapp
```

Add the following configuration to proxy requests to PM2:

```
nginx
Copy code
server {
    listen 80;
    server_name example.com;  # Replace with your domain name or IP
address

    location / {
        proxy_pass http://127.0.0.1:3000;  # PM2 app running on port
3000 (change to your PM2 app's port)
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }

    error_log /var/log/nginx/myapp_error.log;
    access_log /var/log/nginx/myapp_access.log;
}
```

server_name: Replace `example.com` with your actual domain or IP address.

proxy_pass http://127.0.0.1:3000: This is where the NGINX will forward the requests. The Node.js application running under PM2 is typically listening on a local port (e.g., `3000`).

proxy_http_version 1.1: Ensure that HTTP/1.1 is used for WebSocket support (if you're using WebSockets).

proxy_set_header: These headers help forward the correct client information (like `Host` and `X-Real-IP`) to the Node.js application, which may be required for logging or generating proper URLs.

error_log and **access_log**: These define the locations where the error and access logs will be stored for troubleshooting.

Enable the NGINX Configuration

After creating the configuration file, you need to create a symbolic link from `/etc/nginx/sites-available/` to `/etc/nginx/sites-enabled/` to activate it.

```
bash
Copy code
sudo ln -s /etc/nginx/sites-available/myapp /etc/nginx/sites-enabled/
```

Test NGINX Configuration

To ensure there are no syntax errors in your NGINX configuration, run:

```
bash
Copy code
sudo nginx -t
```

If there are no errors, you'll see something like:

```
bash
Copy code
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

Restart NGINX

To apply the changes, restart NGINX:

```
bash
Copy code
sudo systemctl restart nginx
```

Configure PM2 to Automatically Start on Server Reboot

You want your application to start automatically when the server reboots. PM2 provides a way to generate startup scripts for this purpose.

```
bash
Copy code
pm2 startup
```

This command generates a system-specific startup script, and it will give you a command to run. For example, it might suggest running something like:

```
bash
Copy code
sudo env PATH=$PATH:/usr/local/bin pm2 startup systemd -u youruser --hp
/home/youruser
```

Run the command given by PM2 to set up the startup script.

Next, save the PM2 process list so that it can restart your app on boot:

```
bash
Copy code
pm2 save
```

This ensures that PM2 remembers the apps that are running.

Testing the Configuration

At this point, your application should be accessible via your domain or IP address. You can test it by:

- Visiting `http://example.com` (or `http://your-vps-ip`) in your browser.
- If you've set up SSL, you should visit `https://example.com`.

NGINX will route traffic from port 80 (HTTP) or 443 (HTTPS) to the application running on a specific port (e.g., 3000), where PM2 ensures the Node.js application is always running.

PM2 Log Management and Monitoring

You can monitor and manage logs for your Node.js application using PM2:

```
bash
Copy code
pm2 logs myapp   # View logs for your app
```

You can also manage your Node.js application with PM2:

Stop the application:

bash

Copy code

```
pm2 stop myapp
```

Restart the application:

bash

Copy code

```
pm2 restart myapp
```

Delete the application from PM2:

bash

Copy code

```
pm2 delete myapp
```

PM2 and NGINX Best Practices

Performance: PM2 can manage multiple Node.js applications, so you can use it to handle more than one app at once, each listening on a different port. NGINX can proxy to each one.

Security: If you're using NGINX with SSL/TLS (HTTPS), ensure the communication between the client and NGINX is secure. Use Let's Encrypt for free SSL certificates or configure your own.

Scaling: NGINX can be used to load balance traffic to multiple PM2-managed instances of your Node.js application, improving availability and scalability.

Step13

Basic React Architecture with React-Redux and React View

React, as a library for building user interfaces, provides a flexible architecture to manage components, state, and interactions. When building larger applications, it's common to introduce state management solutions like **Redux** to manage the application state in a predictable way.

The combination of **React**, **Redux**, and **React View** creates a powerful architecture that helps developers organize the UI and manage state efficiently. Below is a detailed explanation of the **basic architecture** with these components and how they are connected.

React Architecture

At the core of React's architecture is the **Component-based architecture**. This means that the UI is broken down into small, reusable components that can manage their own state or rely on a central state management system like Redux.

Key Elements of React Architecture:

1. **Components**:
    - React components are the building blocks of the UI. They can be **functional components** (introduced with React hooks) or **class components** (traditional way).
    - Components handle the **rendering of the UI** based on the **state** and **props**.
    - A React component can be **stateful** (having its own internal state) or **stateless** (relying entirely on props).
2. **JSX**:
    - React uses **JSX** (JavaScript XML), a syntax extension that allows writing HTML elements within JavaScript code. JSX is ultimately compiled into `React.createElement()` calls.
3. **Props**:
    - **Props** are the mechanism through which data is passed from parent components to child components. They are **immutable** within the child component.
4. **State**:
    - **State** represents data that can change over time and triggers re-rendering of the component when updated. This can be local to a component or can be managed globally using tools like Redux.

React with Redux (State Management)

Redux is a predictable state container for JavaScript applications, used primarily with React to manage and centralize the application's state.

Key Concepts in Redux:

1. **Store**:
    - The **store** is the central place where all the application's state is stored. It holds the entire state tree of the application.
    - React components can access the state from the store using **Redux connect** (via `connect` or `useSelector` with hooks).
2. **Actions**:
    - **Actions** are plain JavaScript objects that describe an event or action that has occurred in the app. Actions have a **type** and **payload** (optional).
    - Example of an action:

```
javascript
Copy code
const loginAction = {
  type: 'LOGIN',
  payload: { userId: 123, username: 'JohnDoe' }
};
```

3. **Reducers**:
   - **Reducers** are pure functions that specify how the state changes in response to an action. They receive the current state and an action, and return a new state.
   - Reducers determine how the state is updated based on the action type.
4. **Dispatch**:
   - The **dispatch** function is used to send actions to the store. When an action is dispatched, it triggers the reducer function to compute the next state.

React-Redux: Connecting React and Redux

To integrate Redux into React, we typically use the following tools:

1. **Provider**:
   - The `<Provider>` component is used to pass the Redux store down to the component tree. It is placed at the top level of the application.
   - Example:

```
javascript
Copy code
import { Provider } from 'react-redux';
import store from './store';

function App() {
  return (
    <Provider store={store}>
      <AppComponents />
    </Provider>
  );
}
```

2. **connect (Higher-Order Component)**:
   - `connect` is used to **connect React components to the Redux store**. It allows components to access the state and dispatch actions.
   - Example:

```
javascript
Copy code
import { connect } from 'react-redux';

function MyComponent({ username, login }) {
  return (
    <div>
      <h1>{username}</h1>
      <button onClick={login}>Login</button>
```

```
    </div>
  );
}

const mapStateToProps = (state) => ({
  username: state.username
});

const mapDispatchToProps = (dispatch) => ({
  login: () => dispatch({ type: 'LOGIN', payload: { userId:
123, username: 'JohnDoe' } })
});

export default connect(mapStateToProps,
mapDispatchToProps)(MyComponent);
```

3. **useSelector & useDispatch (React-Redux Hooks)**:
   - React-Redux also provides **hooks** (`useSelector` and `useDispatch`) to directly access Redux store data and dispatch actions from functional components.
   - Example with hooks:

```
javascript
Copy code
import { useSelector, useDispatch } from 'react-redux';

function MyComponent() {
  const username = useSelector(state => state.username);
  const dispatch = useDispatch();

  const handleLogin = () => {
    dispatch({ type: 'LOGIN', payload: { userId: 123,
username: 'JohnDoe' } });
  };

  return (
    <div>
      <h1>{username}</h1>
      <button onClick={handleLogin}>Login</button>
    </div>
  );
}
```

React View (UI Layer)

React provides a **view layer** where the UI is rendered dynamically based on the current state of the application. The view is **declarative**, meaning you describe what the UI should look like for any given state, and React will take care of updating the actual DOM.

- **React Views** are essentially **components** that use JSX to define how the UI should be rendered.

- The **UI updates automatically** when the state changes, thanks to React's **reconciliation algorithm** and **virtual DOM**.

Example of a simple React view (component):

```
javascript
Copy code
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

Here, the `Counter` component is responsible for rendering the UI. The `useState` hook is used to manage the local state of the component, and the UI updates automatically when the state changes.

Connection Between React, Redux, and React View

The connection between **React**, **Redux**, and **React View** works as follows:

1. **React Components (View Layer)**:
   - React components are responsible for rendering the UI. They either manage their own local state or get data from the Redux store.
   - When a component needs data that is not local (e.g., user data), it connects to Redux using `connect`, `useSelector`, or `useDispatch`.
2. **React-Redux**:
   - React-Redux acts as the bridge between React and Redux, allowing React components to interact with the Redux store.
   - React components can **read data** from the Redux store using `useSelector` or `connect` and **dispatch actions** using `useDispatch` or `connect`.
   - This enables the components to update the UI based on the state managed by Redux.
3. **Redux Store**:
   - The Redux store holds the **global state** for the entire application.
   - Actions are dispatched from React components to modify the store, and the Redux state is updated by reducers based on the dispatched actions.
4. **React View Updates**:
   - When the Redux state changes (via dispatching actions), React components connected to the store are **re-rendered** to reflect the new state. React

automatically ensures that only the components that rely on the updated state are re-rendered.

Step14

Understanding bcrypt for Password Hashing

bcrypt is a widely used algorithm for hashing passwords in a secure way. Unlike traditional hashing algorithms like MD5 or SHA, **bcrypt** is specifically designed to be slow and computationally expensive, which makes it resistant to brute-force and rainbow table attacks. bcrypt also includes a salt (random data added to the password before hashing) to ensure that even identical passwords will generate different hash outputs.

How bcrypt Works

Salting:

bcrypt automatically generates a **salt** for each password. This salt is a random string of characters that is combined with the password before hashing. The salt ensures that even if two users have the same password, their stored password hashes will be different.

Hashing:

bcrypt applies the **Blowfish cipher** with multiple rounds of encryption to create the hash. The algorithm is intentionally slow to make brute-force attacks more time-consuming. The number of rounds (known as the **cost factor**) can be adjusted to make hashing even slower as computing power increases.

Storing:

bcrypt stores the resulting **hash** and **salt** together in a single string in the format:

swift

Copy code

$2b$10$Saltsaltsaltsaltsaltsaltsaltsalt$hashedpassword

$2b$: The bcrypt version used.

10: The cost factor (i.e., the number of rounds).

Salt: The random salt value.

Hashed password: The resulting hash after applying bcrypt.

bcrypt in the Registration and Login Processes

1. Registration (User Signup)

During **user registration**, the goal is to take the plaintext password provided by the user, hash it securely, and store the hash in the database so that it's not directly exposed.

Steps in Registration:

User submits their password:

The user provides a password during signup (e.g., "mySecurePassword").

Generate Salt:

bcrypt generates a random salt for the password. This is an important step because it ensures that each user's password hash will be unique, even if they use the same password.

Hash the Password:

The salt and the password are combined, and bcrypt performs several rounds of hashing to create a final hash. This makes it computationally expensive to reverse-engineer the hash.

Store the Hash and Salt:

After hashing the password, bcrypt returns a string containing the hash and salt. This string is saved in the database. Example:

swift

Copy code

$2b$10$randomsaltvalue$hashedpassword

Save the hash in the database:

The hash and salt are stored in the database. Note that the actual password is **not stored**, only the hashed version is saved.

Example Registration Code (Node.js):

```
javascript
Copy code
const bcrypt = require('bcrypt');
const saltRounds = 10; // The number of rounds for bcrypt hashing

function registerUser(plainPassword) {
    // Generate a salt and hash the password
    bcrypt.hash(plainPassword, saltRounds, function(err, hash) {
        if (err) throw err;

        // Store 'hash' in the database
```

```
        console.log('Hashed Password:', hash);
        // Example: Save hash to database (e.g., MongoDB, MySQL, etc.)
    });
}
```

2. Login (User Authentication)

During **user login**, the goal is to verify that the password the user enters matches the one stored in the database. To do this, we never compare the actual passwords directly. Instead, we hash the entered password and compare the hash values.

Steps in Login:

User submits their login password:

The user provides their password during login (e.g., "mySecurePassword").

Retrieve the stored hash:

The hash and salt are retrieved from the database. The salt is embedded in the hash string.

Hash the entered password:

The salt from the stored hash is extracted and used to hash the entered password in the same way it was done during registration.

Compare the hashes:

bcrypt compares the newly generated hash (from the entered password) with the stored hash in the database. If the hashes match, the password is correct.

Example Login Code (Node.js):

```
javascript
Copy code
const bcrypt = require('bcrypt');

function loginUser(plainPassword, storedHash) {
    bcrypt.compare(plainPassword, storedHash, function(err, result) {
        if (err) throw err;

        if (result) {
            console.log('Password match! User is authenticated.');
            // Proceed with user login
        } else {
            console.log('Incorrect password.');
            // Reject login attempt
        }
    });
}
```

In this example:

- **bcrypt.compare** takes the entered password (`plainPassword`) and the stored hash (`storedHash`).
- It hashes the entered password using the salt from the stored hash and compares the two hashes.
- If they match, the password is correct; otherwise, the login attempt fails.

Advantages of bcrypt for Password Hashing

Security:

bcrypt uses **salting** to prevent rainbow table attacks. Even if two users have the same password, their hashes will be different because of unique salts.

The **cost factor** (number of rounds) makes bcrypt slow enough to resist brute-force and dictionary attacks.

Scalability:

The cost factor can be adjusted to make bcrypt more or less computationally expensive as hardware capabilities improve over time.

Simplicity:

bcrypt is easy to implement in most programming languages and is well-supported by libraries (like `bcrypt` in Node.js).

Prevents Storage of Plaintext Passwords:

With bcrypt, passwords are never stored in plaintext, ensuring that even if an attacker gains access to the database, the actual passwords are protected.

Step 15

JPay User Authentication Process: Detailed Explanation

The JPay authentication process involves a series of steps that ensure secure user access to the system's resources, leveraging **JSON Web Tokens (JWT)** for token-based authentication. This process ensures that users (whether customers, merchants, or administrators) can only access the resources they are authorized for, based on their roles and permissions. Below is a detailed breakdown of each step:

Step 1: User Authentication Request

1. **Initiating the Login Request**:
    - The authentication process begins when an **end user** (a customer, merchant, or administrator) attempts to log into the system. This could happen through a **client application** such as a website or mobile app.
    - The user typically provides credentials (username, email, or password) to authenticate their identity.
    - The client application collects the credentials and prepares a **login request** to be sent to the **JPay API Gateway**.
2. **Data Sent**:
    - The client sends an **authentication request** to the **JPay API Gateway**, which includes the user's login credentials (e.g., username and password).

Step 2: JWT Issuer

1. **Client Application Requests Authentication**:
    - The client application forwards the user's login details to the **JPay API Gateway**.
    - The **JPay API Gateway** plays a crucial role in managing the authentication flow and can either handle the JWT token creation itself or delegate this process to a third-party JWT issuer.
2. **Role of API Gateway**:
    - If the API Gateway is responsible for authentication, it will validate the credentials by querying the backend server or a database.
    - Alternatively, the API Gateway may redirect the user's authentication request to a **third-party service** that specializes in issuing and validating JWT tokens (such as an identity provider).

Step 3: JWT Token Generation

1. **Authentication Validation**:
    - The API Gateway or third-party issuer processes the **authentication request** and validates the user's credentials.
    - If the user's credentials are valid, the API Gateway generates a **JWT (JSON Web Token)** to represent the user's authenticated session.
2. **JWT Structure**:
    - A JWT is a compact, URL-safe token that consists of three parts:
        - **Header**: Specifies the algorithm used for signing (e.g., HMAC SHA256 or RSA).
        - **Payload**: Contains the claims (e.g., user ID, role, permissions, expiration time).
        - **Signature**: Used to verify that the token has not been tampered with. It is signed using a secret key or a public/private key pair.

3. **JWT Issuance**:
    - The **JWT** generated by the API Gateway contains information (claims) about the **user**, such as:
        - The user's **unique identifier**.
        - The **user's role** (e.g., customer, merchant, admin).
        - Permissions or access levels (specific rights or data the user can access).
        - **Expiration** time of the token (when the token will no longer be valid).
    - Once the JWT is generated, it is sent back to the client application, which stores it (usually in memory or local storage).

Step 4: JWT Token Validation and Application Identification

1. **Client Sends JWT to Access Protected Resource**:
    - When the user (client application) attempts to access a **protected resource** (e.g., user data, merchant sales information, or admin control panel), the **JWT** is sent to the **JPay API Gateway** in the **Authorization header** of the HTTP request.
2. **Token Validation**:
    - The JPay API Gateway, which is now set up with **JWT authentication**, receives the request and **verifies the JWT** for authenticity and integrity.
        - **Authenticity**: The API Gateway checks if the JWT was issued by a trusted authority (i.e., whether it was signed with the correct secret key or public key).
        - **Integrity**: The API Gateway ensures that the token has not been tampered with. This is done by checking the **signature** of the JWT against the header and payload.
3. **Claims Examination**:
    - Once the JWT is validated, the API Gateway also inspects the **claims** contained within the JWT. Claims are pieces of information about the user or session, such as:
        - **Role** (user, merchant, admin).
        - **Permissions** (what data the user is authorized to access).
        - **Expiration** date (ensuring the token is still valid).
    - This helps the API Gateway determine whether the **requesting application** is allowed to access the requested resource.
4. **User Identification and Access Control**:
    - The API Gateway uses the claims within the JWT to **identify** the **application** making the request and determine whether the user has **appropriate access** to the requested resource. For example:
        - A **customer** may only be allowed to access their personal information.
        - A **merchant** may have access to their sales data.
        - An **administrator** may have full access to the system and all user accounts.
5. **Application Identification**:

- In addition to verifying the user's identity, the claims in the JWT might also include **application-specific identifiers** to indicate which client is making the request (e.g., which website, mobile app, or service is involved).

Step 5: Accessing the Protected Resource

1. **Access Granted**:
   - If the JWT is **valid**, and the user's role and permissions are **authorized** to access the requested resource, the **JPay API Gateway** allows access.
   - The user can now interact with the **protected resource**. For example:
     - A **customer** can access their account details.
     - A **merchant** can view their transaction history.
     - An **admin** can manage all user accounts.
2. **Access Denied** (if JWT is Invalid):
   - If the JWT is invalid, expired, or the claims do not match the expected criteria (e.g., insufficient permissions), the **API Gateway denies access** and may return an error response (e.g., HTTP 401 Unauthorized or 403 Forbidden).

User/Merchant/Admin Login Access Control

1. **User Roles and Permissions**:
   - Each user in the system is assigned a **role** (e.g., user, merchant, admin), and their access to different parts of the system is governed by these roles.
     - **Users**: Regular customers who can view their personal data, make transactions, and interact with basic features.
     - **Merchants**: Users who sell products or services, with access to sales data, order management, and inventory.
     - **Admins**: Administrative users with the highest level of access, who can manage users, view all transactions, and control system settings.
2. **Access Control with JWT**:
   - The JWT contains claims such as **user role** and **permissions**, allowing the API Gateway to enforce role-based access control (RBAC). For example:
     - Only **merchants** can access sales data.
     - **Admins** can view and modify all user accounts.
     - **Users** can only access their own profile data.
3. **Role-Based Access**:
   - The **JWT's role** claim helps to enforce the **least privilege principle**, ensuring that each user, merchant, and admin only has access to the resources they need based on their role.
4. **Multi-Role Scenarios**:
   - The system may also support **multi-role scenarios**, where a user can hold more than one role. For example, an **admin user** might also have merchant-level access, and this can be reflected in the claims of their JWT.