# Splunk

* **Splunk is a powerful platform** used to monitor, search, and analyze application logs through a single web interface.
* **It allows developers to filter logs** for specific microservices, set alerts for errors, and monitor overall system performance in real time.
* **In a microservices architecture, Splunk centralizes log management**, making troubleshooting and maintenance much easier.

**Example**

**How does Splunk help in a microservices architecture?**

* Let's say you have multiple microservices like **Order Service**, **Inventory Service**, and **Payment Service**.
* To track any issue in these services, you need to **generate logs** for each one, right?
* But if all the logs are being captured in a **single log file**, that becomes a **bad practice** or poor design.
* For example, if an issue occurs in the **Inventory Service**, you would have to **manually debug the entire log file** to find the problem.
* This process would take a lot of time because it’s difficult to **filter out the specific log flow** for the Inventory Service.
* To overcome the problem of mixed log data, **Splunk helps by segregating logs using indexes**.
* You can **create a separate index** for each microservice, such as:
* **Order Service Index**
* **Inventory Service Index**
* **Payment Service Index**
* Then, **forward all logs from each microservice** to its respective Splunk index.
* Order Service logs → *Order Service Index*
* Inventory Service logs → *Inventory Service Index*
* Payment Service logs → *Payment Service Index*
* This approach allows you to **separate and manage logs for each microservice** efficiently.
* When an issue occurs in any service, you can **search logs in Splunk by the specific index**, retrieving only the relevant service’s data.
* This makes **debugging and issue resolution much faster and easier**.
* That’s why **Splunk is highly popular in real-time, microservices-based architectures**.
* Apart from log analysis, **Splunk also offers powerful features** like **monitoring, alerting, and dashboarding**.

URL: where your Splunk server will redirect the logs

Host: What is the host where your Splunk server is running

Token: what is the security token to connect with your Splunk server

Index: In which index you want to push your application logs

Source: what is your source type (who will send your logs)

**Create Index**

1. Open Splunk in your browser → http://localhost:8000
2. Go to **Settings → Indexes** **→** Click **“New Index”**
3. Enter a name (ex: **Policy\_Dev\_API**) **→** Leave other settings as default (or adjust if you need)
4. Click **Save** ✅ That’s it — your index is created.

**Check Token**

1. Setting → Data Inputs → HTTP Event Controller → Copy Token Value
2. Source Type: Log4J

**Q1: What is Splunk and why is it used?**

Splunk is a log management and analysis tool used for collecting, indexing, and visualizing log data from different sources to monitor and troubleshoot applications.

**🔹 Q2: How do you integrate Splunk with a Spring Boot application?**

You can integrate Splunk using the HTTP Event Collector (HEC) by configuring a Logback appender that sends logs as JSON over HTTP/S to a Splunk endpoint.

**🔹 Q3: What is the role of an HTTP Event Collector (HEC)?**

HEC is a Splunk component that allows data ingestion via HTTP/HTTPS with a token-based authentication system.

**How to Implementation of Splunk?**

**Step 1: Enable Splunk HEC**

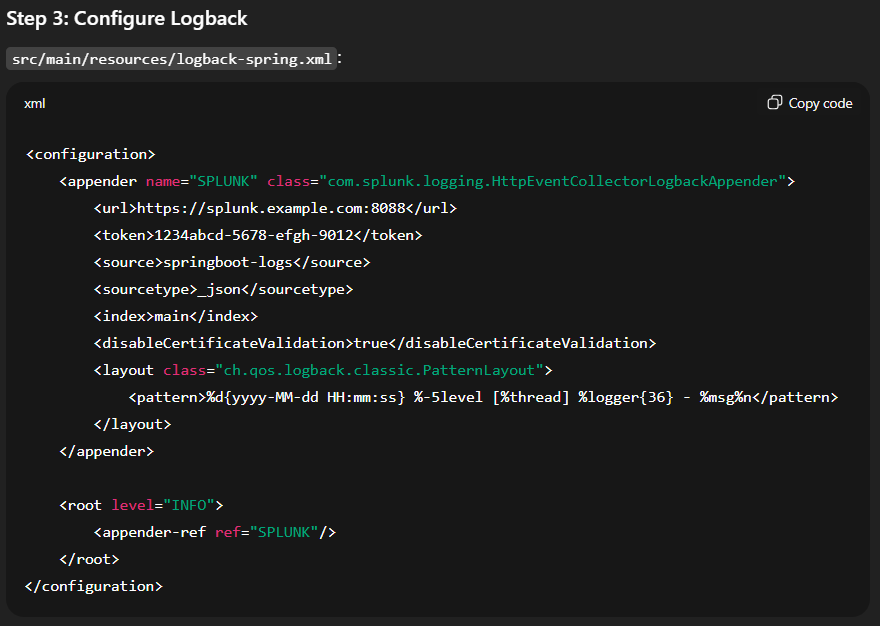
1. Log in to Splunk → **Settings → Data Inputs → HTTP Event Collector → New Token**
2. Name it, e.g., springboot-app-logs.
3. Copy the **HEC token** and **endpoint URL**, e.g.:

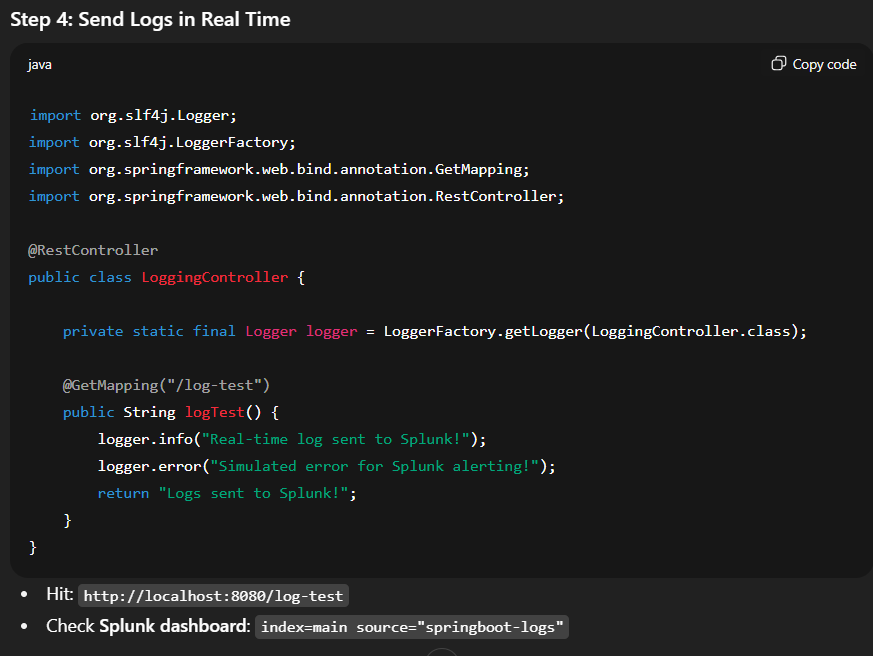
<https://splunk.example.com:8088> (Post App Logs)

Token: 1234abcd-5678-efgh-9012

2. **Add Splunk Logging Dependency:** splunk-library-javalogging

3: Configure Logback





**Open Splunk Dash Board**   
Search the search bar and provide with index name **(Index=”order\_api\_dev”)** and give time also as per your requirement.

**Index=”order\_api\_dev” AND(Exception OR Error)**

**what is splunk Alerts?**

Splunk Alerts automatically watch your data and notify you when something important happens — like an error, system failure, or security issue.

You can create an **alert** that triggers when Splunk finds more than 10 “error” events in 5 minutes.  
When it happens, Splunk can:

* Send you an **email**, Call a **webhook** Or even **run a script** automatically.

**Why Use Alerts**

* Detect system or app failures early, Notify teams of critical issues
* Automate responses (restart services, trigger scripts), Monitor security events

Use scheduled alerts for most monitoring cases (they’re lightweight).  
Use real-time alerts only for critical issues that need instant attention.

**Create Alerts using Splunk**

**Go To Splunk Dash Board**   
Settings --> Server Settings -> Email Setting -> give required details

**Mail Host:** [**smtp@gmail.com**](mailto:smtp@gmail.com) **-> Enable TLS**

**User Name:** [**svs@gmail.com**](mailto:svs@gmail.com)

**User Password: \*\*\*\*\*\*\*\*\*\***

**Send Email As: Splunk**

**Email footer: Received Email from Splunk.** **Click ON**  **Save**

**Set Alerts**

Go To Splunk Dash Board -> Save As -> Alert

**Title: Job\_Status\_Failed**

**Description: Production got failed.**

**Permission: Shared the Application**

**Alert Type: Scheduled -> run on cron Schedule.**

**Cron Expression: \*/2\*\*\*\* [ every 2 min]**

**Triger Alert: is greater Than [3] times**

**Trigger: Once**

**Trigger Acceleration: Send Mail [give team member mail]**

**Provide Priority: High Click ON SAVE**

 **PRIMARY KEY**

* Uniquely identifies each row in a table.
* Cannot contain **NULL** values.
* Only **one primary key** per table.

 **FOREIGN KEY**

* Ensures **referential integrity** between two tables.
* A column with a foreign key points to a **primary key in another table**.

 **UNIQUE**

* Ensures that all values in a column are **distinct**.
* Multiple UNIQUE constraints can exist in a table.

 **NOT NULL**

* Ensures that a column **cannot have NULL values**.

 **CHECK**

* Ensures that **column values satisfy a specific condition**.
* Example: salary > 0.

 **DEFAULT**

* Sets a **default value** for a column if no value is provided during insertion.

**1. PRIMARY KEY**

* **Purpose:** Uniquely identifies each row in a table.
* **Rules:** Cannot be NULL, only one per table.
* **Example:** emp\_id in Employees table.

**2. FOREIGN KEY**

* **Purpose:** Links a column in one table to the **primary key of another table**.
* **Rules:** Ensures **referential integrity** — the value must exist in the referenced table.
* **Example:** dept\_id in Employees table must exist in Departments table.

 **git init** ✅

* Initializes a local Git repository.
* Needed if the folder isn’t already a repo.

 **git add .** ✅

* Stages all files in the folder for commit.

 **git status** ✅

* Optional, just to check what’s staged or modified.

 **git commit -m "all files added"** ✅

* Commits the staged files with a message.

 **git checkout main** ⚠️

* Only needed if your branch is not main yet.
* If the branch doesn’t exist locally, use:
* git branch -M main

 **git pull** ⚠️

* **Problem:** You must specify the remote and branch if it exists:
* git pull origin main --allow-unrelated-histories
* This merges any remote commits into your local repo. --allow-unrelated-histories is needed if the remote has an initial commit like a README.

 **git push origin main** ✅

* Pushes your local main branch to the remote origin.

 **git push** ✅ (optional)

* Works only if you already set upstream with -u in the previous push.

1. Init → git init
2. Add files → git add .
3. Commit → git commit -m "message"
4. Set branch → git branch -M main
5. Add remote → git remote add origin <URL>
6. Pull remote → git pull origin main --allow-unrelated-histories
7. Push → git push -u origin main