# Linux Server Configuration for Banking Operations:

Certainly, configuring Linux servers for critical banking operations requires careful attention to security and reliability. Below are steps you can follow to achieve this objective:

1. Installation of CentOS & Ubuntu

* Prerequisites for CentOS installation
  + Recommended minimum of 10GB of free disk space
  + CentOS 7 ISO install file

## Step 1: Download CentOS 7

To download the official and up-to-date CentOS 7 ISO file, navigate to https://www.centos.org/download/.  
  
Our recommendation for non-enterprise environments is to download the DVD ISO option, which includes the GUI. We recommend the Minimal ISO option only for production enterprise environments.

## Step 2: Create Bootable USB or DVD

Now that you have downloaded the ISO image, you can create a bootable USB, burn it on a DVD or load the image on a VM.  
Several applications can help you create a bootable USB. We recommend using Etcher. Download the application for your system (Windows, macOS or Linux), install and run.  
  
The setup is intuitive and easy:  
  
Select the CentOS 7 ISO image.  
Insert the USB flash.  
Find the USB and select it in the Select drive step.  
Click Flash.

## Step 3: Boot the CentOS ISO File

Upon booting the CentOS 7 ISO file, you can begin the installation process. To do so, select Install CentOS 7. That will start the installer’s graphical interface.  
  
If you are booting from a USB, click the Install to Hard Drive icon on the desktop. That will open the installation wizard.

## Step 4: Install CentOS

Before starting the installation process itself, select which language you would like to use during installation. The default option is English.  
  
Click Continue to confirm your selection.  
  
Follow the below link to complete the CentOS installation  
  
How to Install CentOS 7 (Easiest Guide With Screenshots) (phoenixnap.com)

## Prerequisites for UBUNTU installation

* System requirements (recommended):
  + 2 GHz dual-core processor
  + 4GB memory
  + 25GB available disk space for storage (less if installing the minimal version)
  + DVD drive or USB port
  + At least a 4GB USB drive

## Step 1: Download the Installation Media

1. In a web browser, visit the Ubuntu download page and pick the Ubuntu version suitable for your machine. The most popular versions include:  
  
Ubuntu Desktop  
Ubuntu Server  
Ubuntu Derivatives  
  
2. Once you find the version you need, click the green Download button. You’ll be taken to a thank-you page, and your download should start. (We will download and install Ubuntu 20.04 for desktops.)  
  
The download is an .iso file. You can use it to create a bootable USB drive.  
  
3. Save the file to a location of your choice.  
  
Step 2: Create Bootable USB

You will need a USB drive with 4GB or more. This process will delete all data on the USB drive. Make sure to backup any existing data on the USB drive.  
  
Option 1: Create a Bootable USB Drive on Ubuntu  
  
Use the Create startup disk tool:  
  
Open a search dialog, and type create startup.  
If it’s not installed, the Software Center will offer the option to install it – choose the option for USB drive, then open the utility.  
In the top pane, click Other, then browse and select the Ubuntu 20.04 .iso file you downloaded.  
In the bottom pane, select your USB drive.  
Click Make startup disk.  
  
Option 2: Create Bootable USB Drive on Windows  
  
You’ll need to install a third-party utility called Rufus to create a USB bootable drive.  
  
1. Download the Rufus utility. Scroll down to the download section and click the link to download the latest version of Rufus.  
  
2. Run the file once downloaded.  
  
3. A pop-up dialog opens. You will be prompted whether you want to check for online updates. Select No.  
  
4. The Rufus utility launches. Plug in the USB drive – you should see the drive pop up in the device field.  
  
Set the USB as the device you wish to write to.  
In the Boot Selection drop-down, click Disk or ISO Image.  
Click the Select button to the right.  
Browse and select the .iso Ubuntu file you downloaded earlier.  
  
5. Click Start.  
  
Step 3: Boot up Ubuntu from USB

1. Turn off your system. Make sure you remove all other USB devices, such as printers, memory cards, etc.  
  
2. Insert the Ubuntu USB drive into the system and turn on your machine.  
  
There are two possible scenarios:  
  
The computer boots the USB drive automatically.  
You need to manually configure USB booting in the Boot Menu or BIOS/UEFI.  
  
3. To manually configure the boot order, tap the boot menu key about once or twice per second as soon as the computer powers on.  
  
The boot menu key may be different depending on your computer manufacturer.   
4. Once you see your boot menu, use the arrows to pick the Ubuntu media to boot from. For a DVD, the entry will usually have DVD or Optical in the name. USB is usually labeled USB.  
  
Your system should start loading the Ubuntu live disc menu.  
  
Note: If you are experiencing issues when booting the USB from the boot menu, try to boot the USB from BIOS/UEFI.  
  
Step 4: Run Ubuntu  
  
You can test Ubuntu 20.04 before you commit to installing it. The .iso includes a live mode that only runs in memory.  
  
Launch this mode by clicking Try Ubuntu.  
  
Step 5: Install Ubuntu 20.04 LTS Desktop  
  
To begin the installation, click Install Ubuntu.  
Follow the below link to complete the Ubuntu installation  
  
How to Install Ubuntu 20.04 LTS {With Screenshots} (phoenixnap.com)

Configure basic settings on the servers

Here are some basic steps to configure basic settings on servers  
CentOS Configuration Basic Settings  
# Update system packages  
sudo yum update  
  
# Install essential tools  
sudo yum install nano wget net-tools  
  
# Configure network settings  
sudo nano /etc/sysconfig/network-scripts/ifcfg-<interface>  
  
Ubuntu Configuration Basic Settings  
# Update system packages  
sudo apt update  
  
# Install essential tools  
sudo apt install nano wget net-tools  
  
# Configure network settings  
sudo nano /etc/network/interfaces

Secure Servers - Create Usernames and Passwords**:**  
  
Create a new user with sudo privileges:  
# CentOS  
sudo adduser <username>  
sudo usermod -aG wheel <username>  
  
# Ubuntu  
sudo adduser <username>  
sudo usermod -aG sudo <username>  
  
Set passwords for the new users:  
sudo passwd <username>  
  
Utilize strong passwords with a minimum of 12 characters, including a mix of numbers, uppercase and lowercase letters, and special characters.

Set up firewall rules  
  
# CentOS Firewall Set up  
sudo systemctl start firewalld  
sudo systemctl enable firewalld  
sudo firewall-cmd --zone=public --add-port=22/tcp --permanent  
sudo firewall-cmd --reload  
  
# Ubuntu Firewall Set up  
sudo ufw allow 22/tcp  
sudo ufw enable  
Implement Automated Backup Scripts:  
  
Create backup scripts to automate the backup process. Use tools like rsync or rsnapshot.  
# Example rsync backup script  
sudo nano /usr/local/bin/backup.sh  
#!/bin/bash  
rsync -av --delete /source/directory /backup/directory  
  
Make the script executable:  
sudo chmod +x /usr/local/bin/backup.sh  
  
Schedule the script using cron jobs:  
sudo crontab -e  
  
Add a line to schedule daily backups:  
0 2 \* \* \* /usr/local/bin/backup.sh

# 2.Java Application Performance Requirements for Financial Transactions

When setting performance requirements for a Java application handling financial transaction, it's essential to consider the criticality of these transactions and the expected user loads. Below are some guidelines to help identify critical financial transactions, define their performance criteria, and outline scenarios for performance testing:

Performance requirements for a Java application handling financial transactions are crucial to ensure the system's reliability, responsiveness, and scalability. Below are some key performance requirements to consider:

1. Response Time Requirements:

a. Critical Transactions:

Define maximum response times for critical financial transactions.

Ensure that response times meet or exceed user expectations.

Example: Fund transfers must complete within 3 seconds.

b. Normal Operations:

Specify acceptable response times for routine operations.

Differentiate between read and write operations.

Example: Account balance inquiries should have a response time of less than 2 seconds.

2. Throughput Requirements:

a. Transaction Volume:

Define the maximum number of financial transactions the system should handle per second, minute, or hour.

Consider peak hours and expected growth.

Example: Process a minimum of 1,000 transactions per second during peak hours.

b. Concurrency:

Specify the maximum number of concurrent users performing financial transactions.

Ensure the system can handle simultaneous transactions without degradation.

Example: Support a minimum of 5,000 concurrent users during peak loads.

3. Availability and Reliability Requirements:

a. Uptime:

Specify the required system uptime.

Consider scheduled maintenance windows.

Example: Achieve 99.9% uptime over a month.

b. Fault Tolerance:

Define how the application should handle server or component failures.

Specify recovery times for various failure scenarios.

Example: Recover from a server failure within 5 minutes without data loss.

Identify critical financial transactions and their expected performance criteria.

Identifying critical financial transactions and establishing their expected performance criteria is essential for designing a robust and reliable system. Here are some examples of critical financial transactions along with their potential performance criteria:

Fund Transfers:

Expected Response Time: Process fund transfers within 2 seconds.

Throughput Requirement: Support a minimum of 500 fund transfers per minute.

Concurrency: Handle at least 1,000 simultaneous fund transfer requests.

Account Balances:

Expected Response Time: Retrieve account balances within 1 second.

Throughput Requirement: Support a minimum of 1,000 account balance inquiries per minute.

Concurrency: Allow at least 2,000 concurrent users checking account balances.

Payment Processing:

Expected Response Time: Authorize and process payments within 3 seconds.

Throughput Requirement: Handle a minimum of 300 payment transactions per minute.

Concurrency: Support at least 500 concurrent payment transactions.

Credit Card Transactions:

Expected Response Time: Authorize credit card transactions within 2 seconds.

Throughput Requirement: Handle a minimum of 400 credit card transactions per minute.

Concurrency: Accommodate at least 700 concurrent credit card transactions.

Derive performance requirements, such as response times and transaction throughput.

* Decide which performance parameters to tune: These are the metrics—such as response time, transaction throughput, memory consumption, CPU usage, etc.—that indicate how well your application operates. The parameters you select must take into account the resources on your system as well as the requirements of your application. You might wish to concentrate on response speed and transaction throughput, for instance, if your application is managing financial transactions.
* Determine the system's limits: These are the restrictions on hardware specs, network speed, database size, etc. that limit the performance of your program. Setting reasonable objectives for your application and measuring the highest values your system can manage for each performance parameter are essential.
* Use a profiler to address bottlenecks: A profiler is a tool that gathers and examines performance statistics about your program, including thread behavior, memory allocations, method execution times, CPU utilization, etc. To find the areas of your code that are creating performance problems, utilize a profiler, and then optimize those sections appropriately.
* Build a performance test suite: A performance test suite is a collection of tests designed to measure the performance of your application under various situations and to mimic various workloads and scenarios. To track and enhance the performance of your application, you must build a performance test suite that includes all of its essential features and use cases. You should then execute this suite on a regular basis.

Define scenarios for performance testing, including expected user loads.

Performance testing scenarios involve simulating real-world conditions to assess how a system performs under different circumstances. Here are some performance testing scenarios for a Java application handling critical financial transactions, along with expected user loads:

1. Normal Load Scenario:

Objective: Simulate day-to-day operations.

Expected User Load:

Concurrent Users: 500

Transaction Rate: 100 transactions per minute

Activities:

Account balance inquiries

Fund transfers

Payment

2. Peak Load Scenario:

Objective: Test the system under the highest expected load.

Expected User Load:

Concurrent Users: 1,000

Transaction Rate: 200 transactions per minute

Activities:

Intensive fund transfers

High-frequency stock trading

Peak payment processing

3. Stress Testing:

Objective: Evaluate system behavior beyond normal and peak loads.

Expected User Load:

Concurrent Users: 1,500 (increasing gradually)

Transaction Rate: 300 transactions per minute (increasing gradually)

Activities:

Simulate sudden spikes in transaction volume

Evaluate system recovery after stress conditions

4. Ramp-Up Testing:

Objective: Gradually increase user load to identify system saturation.

Expected User Load:

Start with 200 users and increase by 50 users every minute.

Activities:

Gradually increasing account balance inquiries and fund transfers.

# **3.Software Testing Strategies for Banking Services:**

A test plan is a document that outlines the scope of testing, including functional and performance aspects, for testing banking services. Here are some steps to create a test plan for banking services:

1. **Define the scope of testing**: Identify the banking services that need to be tested. This includes identifying the functional and performance aspects of the banking services that need to be tested.
2. **Identify the testing approach**: Determine the testing approach that will be used to test the banking services. This includes identifying the types of tests that will be performed, such as functional testing, performance testing, and security testing.
3. **Define the test environment**: Identify the test environment that will be used to test the banking services. This includes identifying the hardware, software, and network configurations that will be used to test the banking services.
4. **Identify the test data**: Identify the test data that will be used to test the banking services. This includes identifying the types of data that will be used to test the banking services, such as customer data, transaction data, and account data.
5. **Define the test scenarios**: Define the test scenarios that will be used to test the banking services. This includes identifying the steps that will be taken to test the banking services, such as logging in, creating an account, and making a transaction.
6. **Identify the test cases**: Identify the test cases that will be used to test the banking services. This includes identifying the specific tests that will be performed to test the banking services, such as testing the login functionality, testing the account creation functionality, and testing the transaction functionality.
7. **Define the test schedule**: Define the test schedule that will be used to test the banking services. This includes identifying the start and end dates for testing, as well as the duration of the testing period.
8. **Identify the test team**: Identify the test team that will be responsible for testing the banking services. This includes identifying the roles and responsibilities of each team member, as well as the skills and experience required for each role.
9. **Define the test deliverables**: Define the test deliverables that will be produced during testing. This includes identifying the test reports, test scripts, and other documentation that will be produced during testing.

Here are some core functionalities that can be tested for banking services:

1. **Account management**: This includes testing the creation of new accounts, updating account information, and deleting accounts. It also includes testing the ability to view account balances, transaction history, and other account details.
2. **Fund transfers**: This includes testing the ability to transfer funds between accounts, both within the same bank and between different banks. It also includes testing the ability to schedule recurring transfers and to cancel transfers.
3. **Transaction processing**: This includes testing the ability to process transactions, such as deposits, withdrawals, and bill payments. It also includes testing the ability to handle errors and exceptions that may occur during transaction processing.

These are just a few examples of the core functionalities that can be tested for banking services. A comprehensive test plan should include additional functionalities, such as security testing, performance testing, and usability testing, to ensure that the banking services are fully functional and meet the needs of the users.

Manual testing

To conduct manual tests for user interfaces (UI) and functional scenarios to banking services, you can follow these techniques:

1. **Exploratory testing**: This technique involves exploring the application to identify defects and issues that may not be covered by the test cases. It is an unscripted approach to testing that allows testers to use their creativity and intuition to identify issues.
2. **Usability testing**: This technique involves testing the usability of the application by observing users as they interact with the application. It helps to identify issues related to the user interface, such as navigation, layout, and design.
3. **Regression testing**: This technique involves testing the application after changes have been made to ensure that the existing functionality has not been affected. It helps to identify issues related to the integration of new features with existing functionality.
4. **Functional testing**: This technique involves testing the functionality of the application to ensure that it meets the requirements. It helps to identify issues related to the core functionality of the application.
5. **Integration testing**: This technique involves testing the integration of different components of the application to ensure that they work together as expected. It helps to identify issues related to the interaction between different components of the application.
6. **Performance testing**: This technique involves testing the performance of the application under different conditions, such as high load and stress. It helps to identify issues related to the performance of the application.
7. **Security testing**: This technique involves testing the security of the application to ensure that it is secure from external threats. It helps to identify issues related to the security of the application.

These are just a few examples of the testing techniques that can be used to conduct manual tests for user interfaces (UI) and functional scenarios to banking services. A comprehensive test plan should include additional techniques, such as automated testing, to ensure that the banking services are fully functional and meet the needs of the users.

Automation tools or frameworks that can be used for repetitive testing tasks:

1. **Cigniti’s Test Automation Framework (CTAF)**: This is a tool-agnostic framework that can be used across multiple applications and mobile devices with minimal re-work. It is reusable and modular, reducing effort by around 25%. It enables test teams to jump-start test events around 30% faster to deliver solutions quickly and realize greater ROI. [The framework is also pro-agile with CI tools such as Jenkins](https://www.lambdatest.com/blog/automation-testing-frameworks/)
2. **QA Source**: QA Source provides software testing services for banking applications. [They recommend selecting tools that provide key advantages such as supporting test objectives and the longevity of the product](https://testrigor.com/blog/test-automation-framework/)
3. **Delimit-Ex**: This is a one-step solution that helps decode complex financial files by automatically decrypting them and exporting them in a readable format such as EFT and ABA. This reduces the testing time by 40%, relieving the testers to focus on finding defects instead of reading the files. [The tool is also reverse engineered to create test files in various formats and is embedded with basic auto validation capabilities](https://www.lambdatest.com/blog/automation-testing-frameworks/)

These are just a few examples of simple automation tools or frameworks that can be used for repetitive testing tasks. A comprehensive test plan should include additional tools and frameworks, such as Selenium, Appium, and TestComplete, to ensure that the banking services are fully functional and meet the needs of the users.

Integration testing is a critical step in verifying the seamless integration of banking application components. It ensures the proper interaction between the various components of a banking application, such as user interface, database, and transaction processor. Integration testing is also essential to verify that the application integrates well with other systems and applications in the banking environment.

Integration Testing

To perform integration testing, you can follow these steps:

1. **Identify the components**: Identify the components of the banking application that need to be tested. This includes identifying the user interface, database, and transaction processor.
2. **Create test cases**: Create test cases for each of the components. These test cases should include both positive and negative scenarios to ensure that the components are working as expected.
3. **Perform integration testing**: Perform integration testing to verify the proper interaction between the components of the banking application. This includes testing the user interface, database, and transaction processor to ensure that they are working together as expected.
4. **Analyze the results**: Analyze the results of the integration testing to identify any defects or issues that need to be addressed. This includes identifying the root cause of the defects and developing a plan to fix them.
5. **Report the results**: Report the results of the integration testing to the relevant stakeholders. This includes providing a summary of the test results, as well as any defects or issues that were identified.

These are just a few steps that can be followed to perform integration testing for banking application components. A comprehensive test plan should include additional steps, such as identifying the test environment, test data, and test team, to ensure that the banking services are fully functional and meet the needs of the users.

Security testing

Inspecting input fields for validation checks is an important aspect of security testing for banking services. [Input validation is performed to ensure that only properly formed data is entering the workflow in an information system, preventing malformed data from persisting in the database and triggering malfunction of various downstream components](https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html) .

Input validation should be applied on both syntactical and semantic level. Syntactic validation should enforce correct syntax of structured fields (e.g. SSN, date, currency symbol). Semantic validation should enforce correctness of their values in the specific business context (e.g. [start date is before end date, price is within expected range](https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html)).

[To ensure proper input validation, you can use a variety of techniques such as data type validators available natively in web application frameworks, validation against JSON Schema and XML Schema (XSD) for input in these formats, type conversion with strict exception handling, minimum and maximum value range check for numerical parameters and dates, minimum and maximum length check for strings, array of allowed values for small sets of string parameters, and regular expressions for any other structured data covering the whole input string](https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html).

[It is also important to note that input validation should not be used as the primary method of preventing XSS, SQL Injection and other attacks which are covered in respective cheat sheets but can significantly contribute to reducing their impact if implemented properly](https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html) .

Ensuring secure handling of sensitive data is a critical aspect of security testing in applications. Sensitive data includes any information that is confidential or private, such as personal identification numbers, credit card numbers, and bank account information.

To ensure secure handling of sensitive data, you can follow these best practices:

1. **Encrypt sensitive data**: Encrypting sensitive data is an effective way to protect it from unauthorized access. Encryption ensures that the data is unreadable without the appropriate decryption key.
2. **Implement access controls**: Access controls should be implemented to ensure that only authorized users have access to sensitive data. This includes implementing role-based access controls, password policies, and multi-factor authentication.
3. **Perform regular security audits**: Regular security audits should be performed to identify any vulnerabilities or weaknesses in the system. This includes conducting penetration testing, vulnerability assessments, and code reviews.
4. **Use secure communication channels**: Secure communication channels, such as SSL/TLS, should be used to transmit sensitive data over the internet. This ensures that the data is protected from interception and eavesdropping.
5. **Implement data retention policies**: Data retention policies should be implemented to ensure that sensitive data is not retained longer than necessary. This includes defining data retention periods and securely deleting data when it is no longer needed.

Exploratory testing is an approach to testing that involves exploring the application to identify defects and issues that may not be covered by the test cases. It is an unscripted approach to testing that allows testers to use their creativity and intuition to identify issues.

To conduct exploratory testing for banking services, you can follow these steps:

1. **Identify the areas to be tested**: Identify the areas of the banking services that need to be tested. This includes identifying the user interface, database, and transaction processor.
2. **Create test scenarios**: Create test scenarios that cover a wide range of scenarios, including both positive and negative scenarios. These scenarios should be designed to test the functionality, reliability, and performance of the banking services.
3. **Perform exploratory testing**: Perform exploratory testing to identify any defects or issues that may not be covered by the test cases. This includes using your creativity and intuition to identify issues that may not be immediately apparent.
4. **Document the issues**: Document any defects or issues that are identified during exploratory testing. This includes providing a detailed description of the issue, as well as steps to reproduce the issue.
5. **Report the issues**: Report the issues to the relevant stakeholders. This includes providing a summary of the issues, as well as any recommendations for addressing the issues.

# **4.Postman API Testing for Healthcare Data Integration:**

Validate each endpoint returns the expected data or performs the intended action.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **#** | **Route** | **Method** | **Type** | **Full route** | **Description** | **Status** |
| **1** | /employee | GET | JSON | <https://dummy.restapiexample.com/api/v1/employees> | Get all employee data | Returned all the Employee data  "Successfully! Record has been fetched." |
| **2** | /employee/{id} | GET | JSON | <https://dummy.restapiexample.com/api/v1/employee/1> | Get a single employee data | Returned only one Employee data  "Successfully! Record has been fetched." |
| **3** | /create | POST | JSON | <https://dummy.restapiexample.com/api/v1/create> | Create new record in database | New record has been added  Successfully! Record has been added. |
| **4** | /update/{id} | PUT | JSON | [https://dummy.restapiexample.com/api/v1/update/21](https://dummy.restapiexample.com/api/v1/update/21/) | Update an employee record | Record has been updated  "Successfully! Record has been updated." |
| **5** | /delete/{id} | DELETE | JSON | [https://dummy.restapiexample.com/api/v1/delete/2](https://dummy.restapiexample.com/api/v1/delete/2/) | Delete an employee record | Record has been deleted  Successfully! Record has been deleted |