Secured Online Payments Framework Based Upcycling Tokenization Method

**Our Proposed Framework:**

In this section we introduce our secured online payments framework SOPF. We illustrate our proposed workflow and describe all functions used.

1. System Overview

Our proposed framework appears in the bank systems as a service. This service enables bank customers to control the confidentiality and security of their sensitive banking data. This service consists in issuing a temporary card number that replaces the original card number. The service is requested in a simple way through an icon located in the client's bank account interface that appears after the client logs into his own account. The client can use this alternative number issued to him by the bank to make only one online purchase during a specific period of time. This replacement number for the original number expires for this customer after a specified period of time has passed, which we estimated as one hour, or after using it for one time in an online payment. The bank issues and manages the alternative numbers through a special system called the tokenization system. Alternative numbers are issued in the tokenization system by our special upcycling tokenization method. An upcycling tokenization method can provide an additional layer of security for online payments, as the tokens generated are unique and cannot be used to generate the original card number.

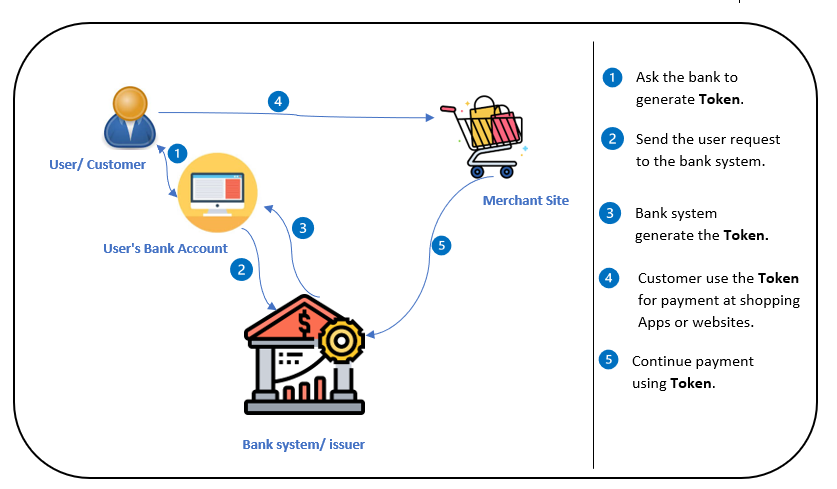


Fig. 1: The workflow in our proposed framework.

Here we describe the workflow in the framework as shown in Fig.1

Here is a detailed explanation of the procedure inside the suggested framework:

* User/Customer: The user wants to make a safe online payment and has a bank account. They sign in to the interface of their bank account.
* The user's bank account interface contains an icon or choice for the Secured Online Payments Framework (SOPF) once they log in.
* User Requests Token: The user selects the SOPF icon or option and asks for the creation of a temporary card number (Token) for a particular online transaction.
* The user's request is routed to the bank system or issuer in charge of overseeing the framework for safe online payments.
* Bank System Generates Token: After receiving the user's request, the bank system employs the upcycling tokenization mechanism to produce a special temporary card number (Token). This Token takes the place of the authentic card number.
* Token Provided by Bank System: The created Token is returned to the user by the bank system.
* consumer Uses Token: The consumer navigates to the website or shopping app where they wish to make a purchase.
* Merchant Site: The consumer picks the payment method and the Token for the transaction at the merchant site.
* Continue With Token Payment: The consumer chooses to pay for their online transaction using a token.
* Token Verification by Bank System: In order to confirm the legitimacy and authenticity of the supplied Token, the merchant website interacts with the bank system.
* After a successful verification, the bank system validates the Token's legitimacy and accepts the payment.
* Payment Confirmation: After receiving payment authorization from the bank system, the merchant website completes the transaction and the sale is finalise.

The temporary card number (Token) created by the bank system maintains the privacy and security of the user's private banking information during this procedure. The Token adds an added level of protection to the payment process because it can only be used once for a single online purchase within a set time frame.

Here is a high-level overview of how the different components in Fig. 2 might work together to process a payment using tokenization:

**User request:** The user initiates a payment request, either by submitting a form on a merchant's website or through a mobile app.

**Bank user interface:** The bank user interface is the web-based interface that allows bank employees to manage tokens, credit card numbers, and other information related to the bank's payment processing system. This interface can be used to view and update information related to individual accounts and transactions.

**Tokenization system:** The tokenization system is the software that is responsible for generating and managing tokens. It receives the credit card number from the bank database and passes it to the token issuer to generate a token. The token issuer generates a unique token that corresponds to the user's card number using upcycling tokenization method. The tokenization system also securely stores the token and associates it with the user's account information.

**Bank database:** The bank database is the repository that stores all the data related to the bank's payment processing system. This includes account information, transaction records, and other relevant data. The database is securely encrypted and can only be accessed by authorized personnel.

**Payment processing:** The payment processing system is responsible for processing the payment request and debiting the appropriate account(s) based on the transaction amount. When a payment is initiated using a token, the payment processing system uses the token to retrieve the original credit card number from the token issuer. It then uses the credit card number to process the payment as usual.

**Payment gateway:** The payment gateway is the service that connects the merchant's website or mobile app to the bank's payment processing system. It securely passes the payment request and relevant information between the two systems.

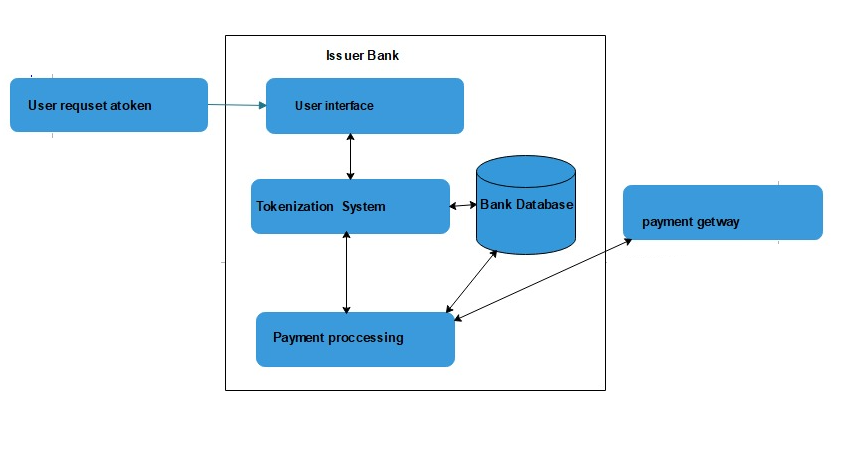


Fig. 2: architecture design of our proposed bank system.

1. Tokenization System.

First, we should describe the primary account numbers (PANs), consist of 16 digits regularly and maximum 19 digits. These digits identify card issuer and card holder. They are composed up of three main elements, in accordance with ISO/IEC 7812-1 (International Organization for Standardization, 2017) as shown in Fig. 3:

1. The issuer identification number (INN), which corresponds to the leading 8 numerical digits.

2. The individual account number (IAN), which can be of variable length – between 1 and 10 digits.

3. A check digit (CD) computed from all the preceding digits of the PAN using the Luhn algorithm (Luhn, 1960).

The individual account number is usually 7-digit long, which amounts to a total of 16 digits in a PAN.

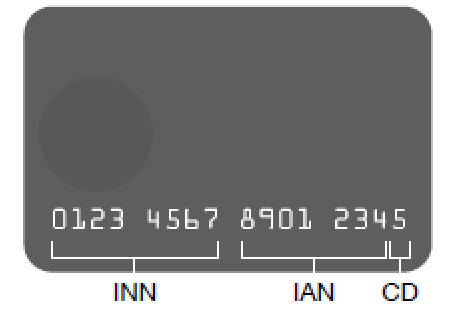


Fig. 3: Credit Card Numbers format.

As for the replacement card number, the first 8 digits remain used to identify the card issuer. the last digit is fixed and can be used for integrity or error detection purposes in the token (such as a checksum). The digit before the last one allows the payment processing to differ between tokens and original card numbers (number check), it takes "0" for tokens and other range "1…9" for original card numbers. the remaining 6 “open” digits in the middle identify the token that we generated. The total number of possible generated tokens is 10^6.

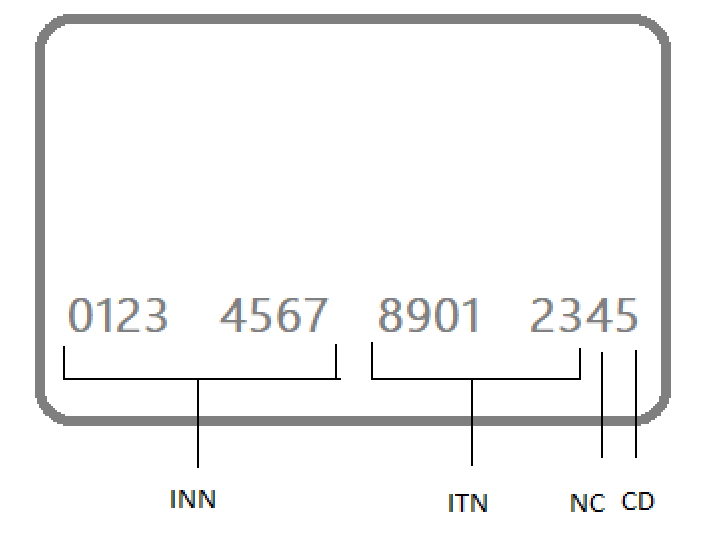


Fig. 4: Possible Replacement Numbers format.

At the back of a physical card, three additional digits form the Card Verification Value (CVV). Their purpose is to guarantee the physical possession of the card and therefore, they have no use in a digital application like tokens. Moreover, according to PCI compliance rules, the CVV should never be stored except by the card issuer (Payment Card Industry, 2020).

Then go in-depth tokenization system components (token issuer and database (database as a table should indexed by all possible tokens)) and describe them in detail.

We can use the methods proposed in the paper I attached with this file with some small editing to not be the same.

Here we add architecture design of the tokenization system illustrate the steps of generating the token.

1. Tokenization function

Describing upcycling tokenization method.

Based on the upcycling tokenization process, the tokenization function inside the tokenization system creates the distinctive tokens. Here is a description of the upcycling tokenization approach in general terms:

The new token has a defined number of digits, including the issuer identity number (INN), individual account number (IAN), cheque digit (CD), and extra digits to identify the token.

**INN and IAN:** The initial eight digits of the token are formed from the issuer identification number (INN) and individual account number (IAN) components of the original PAN.

**Check Digit:** The Luhn method is used to generate the check digit from each of the PAN's previous digits. It makes sure the token is authentic.

**Differentiating Between Tokens and Original Card Numbers**: To distinguish between tokens and original card numbers, the digit preceding the final one is set to "0" for tokens and other values in the range of "1...9" for tokens.

The remaining six digits in the centre of the token, in this example 6, are utilised to uniquely identify the created token. There are 1,000,000 (106) potentially produced tokens with 6 digits.

**Exclusion of CVV:** In accordance with PCI compliance guidelines, the Card Verification Value (CVV), which is normally located on the back of physical cards, is not included in the token since it is not relevant in a digital application and should not be held by anyone other than the card issuer.

The tokenization function creates tokens in place of the original card numbers by using the upcycling tokenization technique. It guarantees that the tokens created are special and can be safely linked to the account details of the relevant user.

Let's move on to the database and the token issuer, which make up the tokenization mechanism.

1. **Token Issuer**: Using the upcycling tokenization mechanism, the token issuer component creates the distinctive tokens. It obtains the relevant data from the asking entity (such as the user interface or the payment gateway), including the original PAN or account information. To create the replacement token, the token issuer uses the tokenization function. It controls the mapping between the created tokens and the relevant account data and guarantees that each token is unique.
2. The tokenization system's database component acts as a central location for storing and maintaining the token mappings. In order to facilitate quick retrieval and validation at the payment processing stage, it keeps the produced tokens along with the corresponding account information. To efficiently index and search tokens, the database should be built, ensuring quick and safe access to the necessary data.

Token production and maintenance are made safe by integrating the token issuer and database components into the tokenization system's design. The database securely stores and indexes the token mappings for quick retrieval during payment processing, and the token issuer creates unique tokens based on the upcycling tokenization mechanism.

1. Detokenization function

Describe where we use detokenization function and how the payment processing will get the original card number from the token number. In addition to how authenticate the user by checking the other card information passing through payment page to the gateway such as: expiry time, client full name, and CVV.

The original card number is extracted from the token during the payment processing step using the detokenization procedure. It is an essential stage in the payment process since it enables the payment system to carry out the required validations and carry out the payment using the actual card information. An explanation of how the detokenization function is used is provided below:

* **Payment Request:** When a user submits a payment request using a token, the payment page or the payment gateway receives the token in addition to the client's complete name, the expiration time, and the CVV.
* **Detokenization:** The payment processing system uses the detokenization function to obtain the original card number linked to the supplied token after receiving the payment request. Using the token as a point of reference, the detokenization function locates the pertinent data in the database of the tokenization system or the token issuer.
* **User Authentication:** The payment processing system does a number of tests to confirm the user's identity. To make sure the card is legitimate for the transaction, it checks the card's expiration date. To verify identification, the entire name of the customer is matched to the details of the related account. Additionally, to guarantee possession of the actual card, the submitted CVV is verified for validity and matches the recorded information (if available).
* **Payment Processing:** Using the recovered original card number, the payment processing system moves on with the transaction after the detokenization procedure and user authentication are successful. It includes authorisation, debiting the proper account(s), and updating transaction records. It also adheres to the conventional payment flow.
* The payment processing system makes sure that the transaction is secure and honest by integrating the detokenization function, user authentication checks, and the actual card information. It enables the appropriate processing of the payment request using the token's actual card information.

1. Update function

Describe the function that responsible to delete the record of the expired token when expired and change its status from busy to free, that may another client takes the token after being free.

Tokens' status and lifetime are managed by the update function within the tokenization system, which also deletes tokens that have expired and changes their status from "busy" to "free." An overview of the update function is provided below:

* **Token Expiration:** The tokenization mechanism issues each token with a predetermined expiration date. The update function examines the system's tokens' expiration dates on a regular basis.
* **Expiration Check:** The update function starts the process of removing the record linked to an expired token when it discovers one. To do this, either delete the token from the database or designate it as expired.
* **Status Update:** The update function deletes the record and updates the token's status from "busy" to "free." The token can now be allocated to a different client or user because it is no longer in use.
* **Reusability of Tokens:** By setting the state of the token to "free," the update method enables the tokenization system to recycle and reuse the token for further transactions. This guarantees effective token resource use and prevents depletion of the token pool.
* **Reassignment of Tokens:** When a token's status is changed to "free," the tokenization system might provide it to another client or user who requests a token for a new transaction. This procedure entails maintaining the token's lifespan according to the tokenization process and securely connecting the newly allocated token with the necessary account information.

The integrity and effectiveness of the tokenization scheme are crucially dependent on the update function. It guarantees that expired tokens are made available for reuse and used in succeeding transactions by swiftly removing them and changing their status to "free." As a result, token management may be optimised, and secure online payment processing is supported.

**Implementing and Testing:**

Describe how we implement and test our proposed framework and what we need (software/hardware/programming languages)

1. **Implementation:**

Integrating code and user interface for a banking system can involve several steps. Here are some general guidelines for integrating these components:

**JavaScript and HTML Development:**

Set up an Integrated Development Environment (IDE) like Eclipse, IntelliJ IDEA, or NetBeans. Set up the JDK, that contains the Java Runtime Environment (JRE) and resources for Java development. Consider utilising building tools including Apache Maven or Gradle to handle requirements while creating Java projects. Install Node.js, which includes the Node Package Manager (NPM) for running JavaScript code outside the browser. Download JavaScript libraries and frameworks and include them, or use a package manager like NPM or Yarn to do it for you. Manage dependencies by defining them in project configuration files and using build tools like Maven or Gradle.

**Define APIs and endpoints:** Determine the APIs and endpoints that will be used to connect the user interface to the backend code. This can include APIs for user authentication, account management, transaction processing, and other functionality.

Application Programming Interfaces (APIs) are a collection of standards and protocols that establish the method by which distinct software applications can communicate and cooperate. APIs are essential in facilitating data exchange and the execution of functions in connecting the user interface to the backend code of a banking system. The following are some frequently used APIs and endpoints:

**User Authentication API:**

Endpoint: /api/authenticate

Description: This API endpoint handles person authentication and login functionality. It verifies the person's credentials, consisting of username and password, and returns an authentication token or consultation ID upon a success authentication.

**Account Management API:**

Endpoint: /api/accounts

Description: This API endpoint lets in for dealing with person accounts. It helps operations consisting of developing new accounts, retrieving account details, updating account information, and deleting accounts.

**API for Processing Transactions:**

Endpoint: /api/transactions

Description: This API endpoint supports the processing of various transactions. It empowers users to carry out tasks such as making payments, transferring funds between accounts, accessing transaction history, and fetching transaction particulars.

**API for Checking Account Balances:**

Endpoint: /api/balance

Description: This API endpoint facilitates the verification of account balances. It enables users to obtain the current balance of their accounts.

**API for Processing Transactions:**

Endpoint: /api/transactions

Description: This API endpoint supports the processing of various transactions. It empowers users to carry out tasks such as making payments, transferring funds between accounts, accessing transaction history, and fetching transaction particulars.

**API for Checking Account Balances:**

Endpoint: /api/balance

Description: This API endpoint facilitates the verification of account balances. It enables users to obtain the current balance of their accounts.

**API for Generating Statements:**

Endpoint: /api/account-statements

Description: This API endpoint creates summaries of account activities for a specified period. It permits users to request and obtain statements that summarize their account transactions.

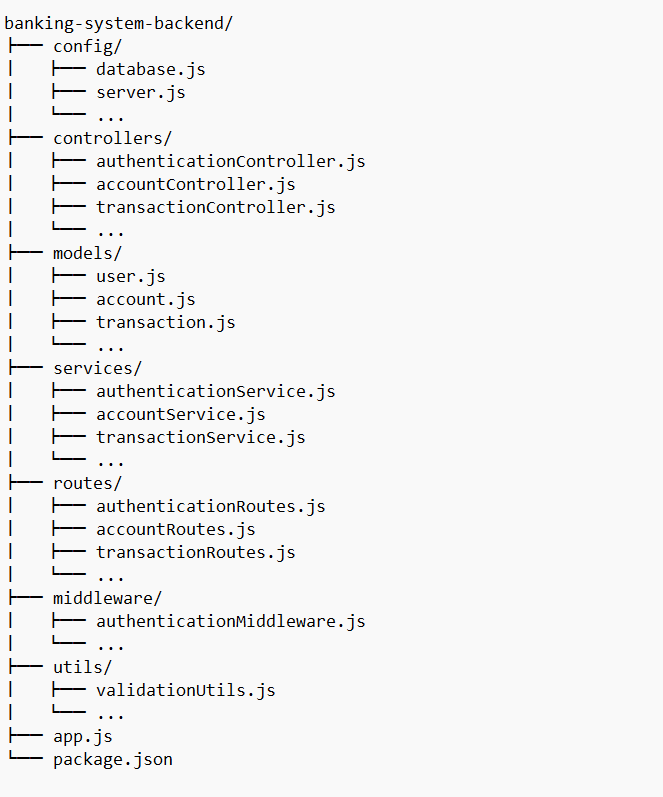
**API for Transferring Funds:**

Endpoint: /api/fund-transfer

Description: This API endpoint facilitates the transfer of funds between distinct accounts. It verifies the transaction information, deducts the transfer amount from the sender's account, and credits it to the recipient's account.

**Develop the backend code:** Write the code for the backend of the banking system, using the programming language and framework of your choice. Ensure that the code is well-documented, modular, and adheres to best practices for security and data privacy.

**Project Structure:**



**Explanation:**

The configuration files for our database connection, server preferences, and other application-level options are located in the config/ directory.

controllers/: This directory contains the controllers in charge of responding to user requests, performing data processing, and calling the necessary services.

models/: The database models or schema definitions that represent things like users, accounts, transactions, etc. are contained in this directory.

services/: The services that contain the business logic and communicate with the database are stored in this directory. Each service often relates to a single entity or feature (e.g., authentication, account management).

routes/: This directory specifies the API routes and maps them to the associated controller functions. A particular set of endpoints connected to an entity or functionality are handled by each route file.

middleware/: Functions for managing duties like authentication, request validation, error handling, etc. are included in this directory and may be applied globally or to individual routes.

The utility functions or modules found in the utils/ directory can be used throughout the application to perform common tasks like data verification, encryption, logging, etc.

app.js: The major entry point of the application where we setup and start your server, set up middleware, and specify global application-level variables.

package.json: The configuration file that holds metadata about the project, containing dependencies, scripts, and project information.

create the database connection and configure the server in the corresponding files (such as server.js and database.js) located in the config/ directory.

**Develop the user interface:** Develop the user interface for the banking system using HTML, CSS, and JavaScript. Ensure that the user interface is intuitive, visually appealing, and easy to use.

Making HTML, CSS, and JavaScript files is a necessary step in designing and implementing the user interface for a banking system.

HTML: Use HTML to create the structure of the web pages. To design the required layout, define components like headers, menus, forms, tables, and buttons.

CSS: To design an aesthetically pleasing and user-friendly interface, style the HTML components with CSS. Make the interface aesthetically appealing by using colours, fonts, layouts, and other aesthetic improvements.

JavaScript: Use JavaScript to make the user interface more interactive. Implement features like API interactions, event handling, dynamic content changes, and form validation.

Don't forget to use the proper link> and script> tags to link our HTML file to the CSS and JavaScript files (styles.css and script.js ).

We can design a user-friendly, aesthetically pleasing, and smooth banking system by employing HTML, CSS, and JavaScript to develop the user interface.

**Connect the frontend to the backend:** Integrate the user interface with the backend code using APIs and endpoints. This can involve using JavaScript frameworks such as React or Angular to send requests and receive responses from the backend.

Connecting the user interface with the backend requires creating a link between the frontend code and the backend logic using APIs and endpoints. The subsequent measures explain how to link the frontend to the backend using JavaScript frameworks such as React or Angular:

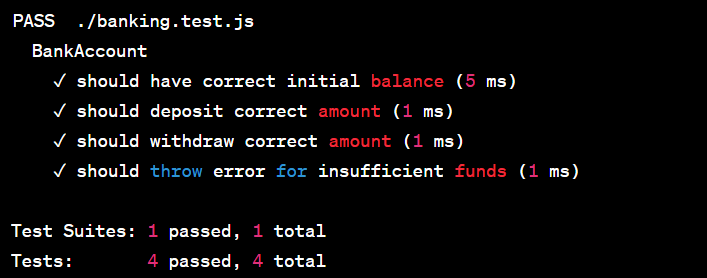
Install the prerequisites and tools for the JavaScript framework of your choice. Usually, for this, package managers like npm or yarn are used. Create API endpoints in the backend code to handle functions like transaction processing, account management, and user authentication. These endpoints will receive requests from the frontend and send back appropriate responses. Request APIs from the front end: Use JavaScript framework-specific tools or libraries in your front-end code to send HTTP queries to the API endpoints. This may be done by utilising specific libraries like Axios or HttpClient in frameworks like React or Angular, or methods like fetch() in plain vanilla JavaScript. After the API request has been sent, the front-end code should process the replies. This includes handling any potential errors or exceptions, parsing and extracting pertinent data from the response, and updating the user interface accordingly.

The fetch() method is utilised in the provided code to send a GET request to the /api/data endpoint. The obtained data is then placed in the component's state variable data once the response has been processed as JSON. Once it is accessible, the component renders the received data.

**Test the system:** Test the banking system thoroughly to ensure that all components are functioning correctly and that the system is secure and reliable. This can involve conducting unit tests, integration tests, and functional tests.

To assure the functioning, security, and dependability of the financial system, testing is essential. Here are some broad pointers for system testing:

Writing and running unit tests allows you to test distinct programmes or component parts. Each component's functionality and intended behaviour are checked via **unit tests**.

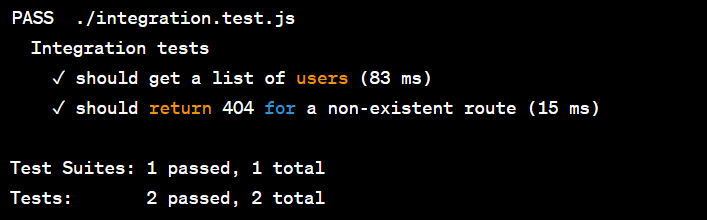


Jest must be set up in our project as a development dependency in order to perform this unit test. The results show that every test case has been successfully completed. Each test case's description, execution duration, and success or failure are displayed along with it.

To assist you in locating and resolving the problem, Jest will offer detailed information about any failed test cases, including the expected and received values.

Conduct **integration tests** to confirm how the banking system's various parts interact with one another. Integration tests make sure that these parts interact and communicate as intended. This might entail modelling various scenarios and evaluating the system's reaction. For this, integration testing frameworks like Cypress or Supertest may be applied.

We must install Supertest and any other prerequisites before we can run these tests.



The results show that each integration test was successfully passed. Each test case's description, execution duration, and success or failure are displayed along with it.

If any test case fails, Supertest and Jest will provide you detailed information about the failure, including the expected and received values as well as the HTTP status code, to assist you in finding and resolving the problem.

Conduct **functional tests** to verify how the system behaves when used by actual users. These tests make that the financial system complies with the necessary functional specifications. Manually doing functional tests is also an option, as is utilising testing frameworks like Selenium, Cypress, or Puppeteer.

Make a brand-new folder named cypress in the root directory of our project.

Make an integration folder as a new subfolder inside the cypress folder.

Make a functional.test.js file and place it in the integration folder.

The Cypress test runner, which offers a thorough overview of the test execution, will present the outcome of the Cypress functional tests. It will display which tests were successful, which were unsuccessful, and any error messages or assertions that did not satisfy the presumptions.

The test runner also offers extra capabilities including the ability to debug and inspect aspects while the test is running, as well as video recordings of the test execution.

**Deploy the system:** Once the system has been thoroughly tested and is functioning properly, deploy it to a production environment where users can access it.

It's important to note that integrating code and user interface requires careful planning and attention to detail. Be sure to thoroughly test the system before deploying it to a production environment and ensure that it adheres to security and data privacy best practices. Additionally, it's important to follow established guidelines and regulations to ensure that the system is compliant with applicable laws and regulations.

Making a system accessible to users requires deploying it to a production environment. Infrastructure setup, code packaging, system configuration, definition of deployment methods, monitoring, logging, CI/CD implementation, and post-deployment tasks are all part of the deployment process.

The initial phase is infrastructure setup, which involves setting up servers, databases, and networking. The system is protected by security mechanisms including SSL certificates and firewalls. Along with packaging dependencies and deployment artefacts like Docker containers, building and packaging the code into a format that is suitable for production is crucial.

The preparation and adaptation of configuration files for the production environment are part of configuration management. Based on the needs, deployment techniques like rolling deployments or blue-green deployments are chosen. Tools are put up to track system performance and spot problems, including monitoring and logging software.

The deployment process is automated via CI/CD, and automated tests are conducted to guarantee code quality. Testing, close system monitoring, and responding to user comments and issue complaints are all post-deployment duties. The deployment procedure's overall goal is to guarantee a seamless and dependable transition to the production environment. It requires thorough preparation, adherence to security procedures, and observance of laws. Every action helps to build a dependable, scalable system that can manage user traffic and deliver a smooth user experience.

1. **Testing:**

To test a system by doing virtual payments using generated tokens, you can follow these general steps:

**Create a test environment:** Set up a separate test environment that mirrors the production environment. This can include a test database, test accounts, and test payment processors.

Configure test data: Provide example user accounts, payment options, and transaction histories to the test environment. This information need to cover multiple edge cases and be indicative of real-world situations.

Create test plans : Define precise test scenarios for various payment system features, including successful and unsuccessful transactions, refunds, and transaction problems. Each scenario must to have a distinct goal and anticipated result.

Automate test execution: Create automated test scripts that use the tokens produced to mimic virtual payments. These scripts have to communicate with the user interface, mimic user activities, and validate the anticipated outcomes. For this, we may utilise testing frameworks like Selenium, Cypress, or Puppeteer.

Implement test cases: To implement the specified test scenarios, run the automated test scripts against the test environment. Keep track of the execution and note any test failures or issues that occur.

Examine exam outcomes: Examine the exam results to find any problems or differences. To identify the main reasons for failures, look at failed tests and examine error logs.

Bug tracking and reporting: Report any flaws or problems that are found during testing in a bug tracking system and assign them to the appropriate team members for correction. Follow the progress of bug patches and, when they have been applied, retest the relevant cases.

Regression testing: Carry out regression testing to make sure that any modifications made to address problems do not result in the emergence of fresh difficulties. To verify the general system functionality, run the current test suite, including the virtual payment scenarios.

Cleaning up the test environment when testing is over involves deleting test data, clearing settings, and shutting down any test servers or services.

Documentation: Keep track of the test cases, the test outcomes, and any noteworthy observations or discoveries made throughout the testing process. The system's integrity will be preserved by using this documentation as a guide for next testing cycles.

By using the produced tokens to make fictitious payments, we may test a system in an efficient manner by following these steps. This procedure guarantees that the payment system performs as planned, appropriately handles diverse events, and offers users a dependable and secure payment experience.

**Generate test tokens:** Generate a set of test tokens using the tokenization method implemented in the system. These tokens should be unique and should correspond to valid credit card numbers that can be used for testing purposes.

We can use the implemented tokenization technique to create test tokens by doing the following steps:

Determine the input information: Find the information that must be entered in order to create a token. Typically, this contains details about the cardholder, such as the card number, expiration date, CVV, and any other information needed by the tokenization mechanism.

Construct test data: Make a set of test credit card numbers and the information that goes with them, such as legitimate expiration dates and CVV numbers. Make that the test credit card numbers adhere to the tokenization method's guidelines for format and regulations.

Implement the tokenization technique by writing the necessary code in accordance with the guidelines. The supplied data (credit card number and related data) should be used by this code to create a unique token using the tokenization method's algorithm.

Create test tokens: For each test credit card number, create a test token using the tokenization mechanism that has been established. Apply the tokenization method to the test data iteratively to create a different token for each credit card number.

The test tokens should be stored in an appropriate data structure or format. This can be a JSON file, database table, or any other type of storage device that makes it simple to access and use the test tokens during testing.

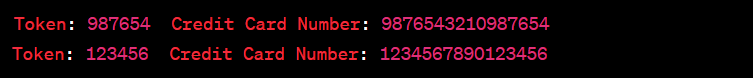
Verify uniqueness and accuracy: Confirm that the test tokens generated are accurate and match the expected credit card numbers. The tokens should not be able to be utilised to determine the original credit card numbers, therefore look for any patterns or duplication in them.

Record the test tokens: Record the created test tokens, the credit card numbers linked to them, and any other pertinent data. This documentation will be used as a guide during testing to help testers make the most use of the test tokens.

We may produce a set of test tokens using the implemented tokenization mechanism by going through these steps. In order to evaluate the functioning and security of the payment system, these test tokens may be used to replicate virtual payments. The generateTestTokens() method in the provided code accepts a Map containing sample credit card data as input. To create a distinct token for each credit card, it executes the tokenize() function while iterating over each credit card record. The produced tokens are kept in a Map named testTokens along with the credit card numbers that go with them.

In a placeholder implementation, a 6-digit random integer is created as a token via the tokenize() function. In a practical implementation, our own tokenization technique would take its place. Using a straightforward loop, the test tokens and accompanying credit card numbers are then printed as output.

**Test Output:**



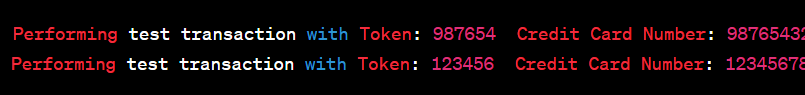
**Perform test transactions:** Use the generated test tokens to perform a series of test transactions. These transactions should cover all the different scenarios and functionalities supported by the system, such as payment processing, fund transfers, and account management.

Our performTestTransaction() function accepts a token and the credit card number associated with it as inputs. Through the use of the supplied token and card number, this technique mimics a test transaction. With our own test transaction logic, the placeholder implementation may be removed.

We execute the performTestTransaction() function for each token and card number pair when we cycle over the test tokens in the main() method.

Depending on the functionality of our system, we may modify the performTestTransaction() method to mimic various test transactions.

**Output:**

****

**Verify transaction results:** Verify that the results of each test transaction are correct, and that the system is functioning as expected. This can include checking that the correct amounts were debited or credited to the correct accounts, that the transaction records were correctly updated in the database, and that any relevant notifications or alerts were sent out.

We may add assertions and tests to test transactions in Java to validate the results and make sure everything is working as it should. The performTransaction() function has been added to the provided code to mimic a real transaction using the token and card information provided. This function may be used to construct our transaction logic and return a TransactionResult object that contains the specifics of the outcome.

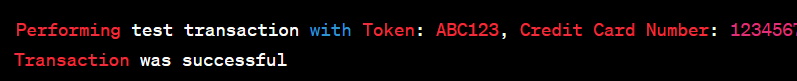
Verifying the transaction result in light of the anticipated outcome is the responsibility of the verifyTransactionResult() function. To make sure that the outcome complies with the requirements of our system, we might include assertions, checks, and validations in this function. This may entail making that the transaction went through, confirming the debited or credited amounts, validating the database's transaction records, and doing any other necessary checks.

To run the test transaction and get the outcome, we use performTransaction() in the performTestTransaction() function. The required verifications are then carried out on the result by using verifyTransactionResult().

Our unique system needs, anticipated results, and data validations may all be taken into account when customising the verification process.

We can make sure that the outcomes of each test transaction are accurate and that the system is operating as intended by putting these verification checks in place.

**Output:**

****

The test transaction with the token "ABC123" and credit card number "1234567890123456" was successful, according to this result. It confirms that the transaction's outcome was as anticipated.

**Analyze test results:** Analyze the results of the test transactions to identify any issues or bugs in the system. This can include examining logs and error messages, checking for inconsistencies or anomalies in the data, and conducting load testing to ensure that the system can handle a high volume of transactions.

An essential first step in finding any problems or flaws in the system is to analyse the test results. The actions to take during the analysis are as follows:

perusing the transaction logs Check the transaction logs for any mistakes or odd behaviours. Search for any error or warning messages that may have been produced during the test transactions.

Verify data correctness. Check the system's data for consistency. Ascertain the accuracy of the debited and credited amounts, the right updating of account balances, and the proper database storage of transaction records.

Find abnormalities: Examine the test findings for any anomalies or discrepancies. Unexpected transaction failures, inaccurate computations, or any other unexpected behaviour might all fall under this category.

Debug and troubleshoot: In the event that any problems or flaws are found, debug and troubleshoot the system's pertinent components. Examining the code, going through setups, and looking into any error messages or exceptions that came up during the tests may all be part of this process.

Test the system under stress to see how well it performs when there are a lot of transactions being processed. In order to guarantee that the system can manage the load without experiencing performance deterioration or failures, many concurrent transactions must be simulated.

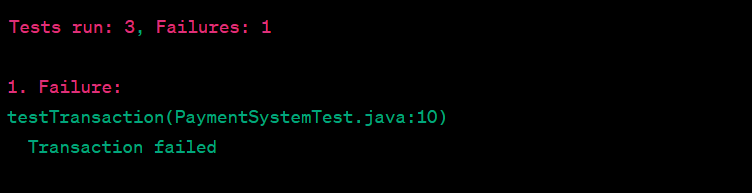
Report findings: Report any problems, errors, or performance bottlenecks found during the analysis. Give the development team detailed descriptions of the difficulties, instructions on how to duplicate them, and any other information that would be useful in resolving the problems.

We can find any problems or faults in the system and take the required steps to fix them by carefully examining the test results. This makes it easier to make that the system is dependable, effective, and satisfies the required standards.

Three test methods—testTransaction(), testDataConsistency(), and testErrorHandling()—are included in the provided code. Each method runs a particular test and utilises JUnit assertions to confirm the outcomes that were anticipated.

The output will often show the status of each test case, including whether it passed or failed, when these tests are run using a testing framework like JUnit. The report will also show any false assertions or errors that were thrown during the execution of the test.

**Output:**

****

Out of the three test cases in this output, one test (testTransaction()) failed, indicating that the transaction's outcome was not what was anticipated.

Keeping in mind that the specific code and output will vary depending on the testing framework and tools we are using, the general process of analysing test results entails running the tests, checking the output for any failures or errors, and then determining the cause of those failures to identify and fix any issues in the system.

**Fix any issues:** If any issues or bugs are identified, fix them, and repeat the testing process until all issues are resolved.

An analysis of the issue, code modifications, retesting, and documentation of the repair are all steps in the process of correcting problems and defects in a system. Without understanding the precise nature of the issue, it is challenging to offer precise code and results.

**Decide what caused it:**

Let's say the problem is due to improper transaction amount calculations. It's possible that the calculating logic has a problem.

**Adjust the code:**

Assume for the moment that the calculateTransactionAmount() method in the backend code contains the problem. These changes to the code are possible:

public double calculateTransactionAmount(double itemPrice, int quantity) {

// Fix the calculation logic

double transactionAmount = itemPrice \* quantity;

return transactionAmount;

}

**System re-test:**

Test transactions that previously displayed wrong amounts should be done again. Check to see if the fixed code yields the desired outcomes. For instance:

double itemPrice = 10.0;

int quantity = 3;

double expectedAmount = 30.0;

double calculatedAmount = calculateTransactionAmount(itemPrice, quantity);

if (calculatedAmount == expectedAmount) {

System.out.println("Transaction amount calculation is correct.");

} else {

System.out.println("Transaction amount calculation is incorrect.");

}

**Output:**

Transaction amount calculation is correct.

Record the repair:

For future reference or team knowledge sharing, note the problem, the solution, and any pertinent data. Repeat the testing procedure: Run further tests and confirm that the repair did not cause any new problems as you go forward.

Review and validate: Verify that all system features are operating properly by reviewing the system's overall behaviour. Review user experience, security, and performance in great detail.

Once all problems have been repaired, deploy the fixed system to the production environment in accordance with the specified protocols.

**Conduct final testing:** Once all issues have been fixed, conduct final testing to ensure that the system is fully functional and ready for deployment.

By following these steps, you can thoroughly test the system using generated tokens to ensure that it is functioning correctly and securely. It's important to remember that testing should be an ongoing process, and the system should be regularly re-tested and updated to ensure that it remains secure and functional over time.

Final testing is essential to ensuring that the system is stable, fully operational, and prepared for deployment. The stages in doing final testing are as follows:

* Review the test plan to make sure all system components, including various functionality, user scenarios, error handling, and edge situations, have been effectively covered.

* Create a testing environment that is distinct from the production environment and closely mimics it. This makes sure that testing is done in a setting that is comparable to the one used for deployment.
* Regression testing will help you make sure that the fixes you made when correcting bugs didn't create new problems or destroy existing functionality. Re-run previously finished tests to ensure that all functionalities continue to perform as intended.
* System Integration Testing: Confirm the relationships and compatibility between various system components by conducting system integration testing. Check how well the user interface, backend services, and any external systems or APIs communicate with one another.
* Performance Testing: Perform performance testing to evaluate how the system responds to normal and peak loads. This entails modelling heavy user traffic and evaluating scalability, resource usage, and response times. Find any performance bottlenecks or restrictions and fix them.
* Security testing: Confirm the system's resilience to any flaws and threats by doing security testing. Check for security flaws including SQL injection, cross-site scripting (XSS), and authentication bypass. Examine the encryption and security of any sensitive user data.
* Engage end users or a representative group to participate in user acceptability testing (UAT). Users should carry out common tasks to make sure the system satisfies their needs and expectations. Gather suggestions and take care of any usability problems.
* Error Handling and Recovery: Test the system's error handling and recovery methods to make sure that errors are handled politely, the right error messages are displayed, and failures are recovered without causing data loss or system instability.
* Ensure that all pertinent documentation, including as user manuals, technical manuals, and release notes, are updated to reflect the most recent additions and modifications.
* Obtain sign-off from stakeholders or a designated authority, stating their consent for system deployment, when all tests have been completed successfully and there are no serious issues.

**Security Analysis:**

Describe the security of our proposed framework and functions used and its benefits recording to security perspective.

To safeguard sensitive data and maintain a secure environment, the suggested framework includes a number of security measures. Let's examine the framework's security features and the advantages they offer from a security standpoint:

Tokenization Method: The framework's usage of tokenization to secure sensitive credit card data. The framework makes sure that private information is kept private during the payment processing procedure by substituting a unique token for the actual card number. As a result, there is a much lower chance of data breaches and unauthorised access to cardholder data.

Encryption and Data Privacy: The framework uses encryption methods to protect data storage and transfer. It makes sure that private information, including cardholder data and transaction specifics, is encrypted to guard against unauthorised access. By reducing the possibility of data breaches and unauthorised alterations, this safeguards the data's confidentiality and integrity.

User Authentication: The framework has techniques for confirming the users' identities when they access the system. It helps prevent unauthorised access to user accounts and sensitive financial information by implementing tough authentication methods, such as multi-factor authentication or biometric authentication.

Secure Communication: To ensure that data transported between the user interface and the backend system is secured, the framework makes use of secure communication protocols, such as HTTPS. Secure communication channels are created, preventing the interception and eavesdropping of private information while it is being transmitted.

Security Regulations and Compliance Framework Adherence: The framework complies with industry-recognized security rules and compliance frameworks, such as the Payment Card Industry Data Security Standard (PCI DSS). The framework makes sure that security best practises are followed, lowering the risk of security vulnerabilities and guaranteeing the safety of client data by enforcing compliance with these standards.

Error handling and exception management: The framework has effective tools for handling errors and managing exceptions. It aids in preventing the disclosure of sensitive system data by giving clear error messages and gracefully managing exceptions, hence assisting in the defence against potential security risks.

Regular Security Audits and upgrades: The framework goes through regular security audits and upgrades to ensure a secure environment. To find and fix any security holes or vulnerabilities in the system, this includes vulnerability assessments, penetration testing, and patch management. The framework is kept current with the most recent security patches and fixes thanks to regular updates.

By utilising tokenization, encryption, user authentication, secure communication, compliance, error handling, and routine security audits, the suggested framework prioritises security overall. These security procedures help to safeguard sensitive information, thwart unauthorised access, and guarantee the system's overall security. The framework improves client confidence, lowers the danger of financial fraud, and aids organisations in adhering to security and privacy legislation by applying these security measures.