FINANCIAL SERVICES CASE STUDY

Controlled Test Data for Payment Processing Applications

APPLICATION ENVIRONMENT

Billions of debit and credit card transactions take place every year representing trillions of dollars in purchases and payments that are processed by financial services companies. Each of the leading credit card companies in the US manage millions of cards issued to business and consumer card holders and process all of their payment transactions on a daily basis. The ability to accurately, efficiently and securely process these payments is the lifeblood of any financial services company.

In this case study, we examine the role of test data used by quality assurance teams to ensure their payment processing applications are rigorously tested for defects, compliance with data interchange standards and performance under heavy load conditions. To protect the privacy and security of cardholders, quality assurance testing must be conducted without the use of any *Personally Identifiable Information (PII)* during test operations.

Payment processing software has become highly sophisticated in its ability to manage complex electronic payment processes. Software must support merchants in a variety of vertical markets (e.g., restaurant, hospitality, e-commerce, etc.) and service cardholders with a wide assortment of card categories, incentive and loyalty programs, credit histories and spending limits for both consumer and business accounts.

In the restaurant industry for example, a two-step process is required – the first step is a pre-authorization for an estimated amount (the charge for the meal prior to applying a tip for the server) followed by a post-authorization in which the actual amount of the transaction is recorded.

At the end of each business day, merchant accounts must be closed or settled. Upon settlement, the movement of funds takes place with notifications to the card-issuing bank for billing the cardholder and the settling of accounts between banks. The merchant receives payment from the issuing bank for the transaction amount minus appropriate processing fees.

Payment applications must be capable of processing millions of transactions on a daily basis. Needless to say, they must calculate payments, processing fees and settle transactions with absolute precision, all while protecting the privacy of all parties involved.

To accurately collect and process this diverse transaction information, it must be structured and formatted as a data feed that conforms to a well-defined data interchange format. Sometimes these data feeds are standardized and sometimes they are not. In this case study, we profile a major financial services company that has created their own proprietary data feed format for collecting transaction data and processing payments.

TEST DATA CHALLENGES

In order to test their payment processing application, the QA team at this financial services company determined their data feeds must be simulated in a highly controlled fashion. To reproduce complex transaction data feeds, the team copied a subset of their production data and prepared it for testing. Production data is attractive because it contains real transactions in the proper data interchange format. However, to prepare the data for testing, it had to be laboriously reworked by hand to create the data variations and permutations needed for test cases while removing all sensitive customer and merchant information.

It took the QA staff 160 man-hours (an entire man month) to build a test data set. Because the data interchange format was revised every six months, the number of man-hours required for test data provisioning effectively doubles over the course of a year. The tedious nature of the provisioning process placed limits on the variety of test data available for functional, integration and regression testing. And the limits on the volume of data provisioned was impacting their ability to perform the load and performance testing required to simulate heavy transaction loads. In the end, they concluded there were too many problems associated with using production data alone for testing purposes. The following summarizes their rationale.

Production data is not controlled data

Without manual modification, test data copied from production data can only test for conditions represented by a given data subset. It does not provide the QA team with the necessary data to test edge case conditions, the presence of invalid data values, or specific input value combinations that might uncover software defects. To maximize code coverage under all potential operating conditions, test data must be controlled to simulate data feeds that contain all of the data variations required by each test case and its assertions.

Production data is not secure data

Business and IT leaders at this financial service company were very concerned about data privacy. The risk of a data breach that might expose sensitive customer credit information was too great when considering the legal and financial consequences. This risk was further compounded by the fact that much of the testing was being performed by offshore contract resources, limiting the internal control over the handling of sensitive customer data.

Secure, high volume production test data is not practical

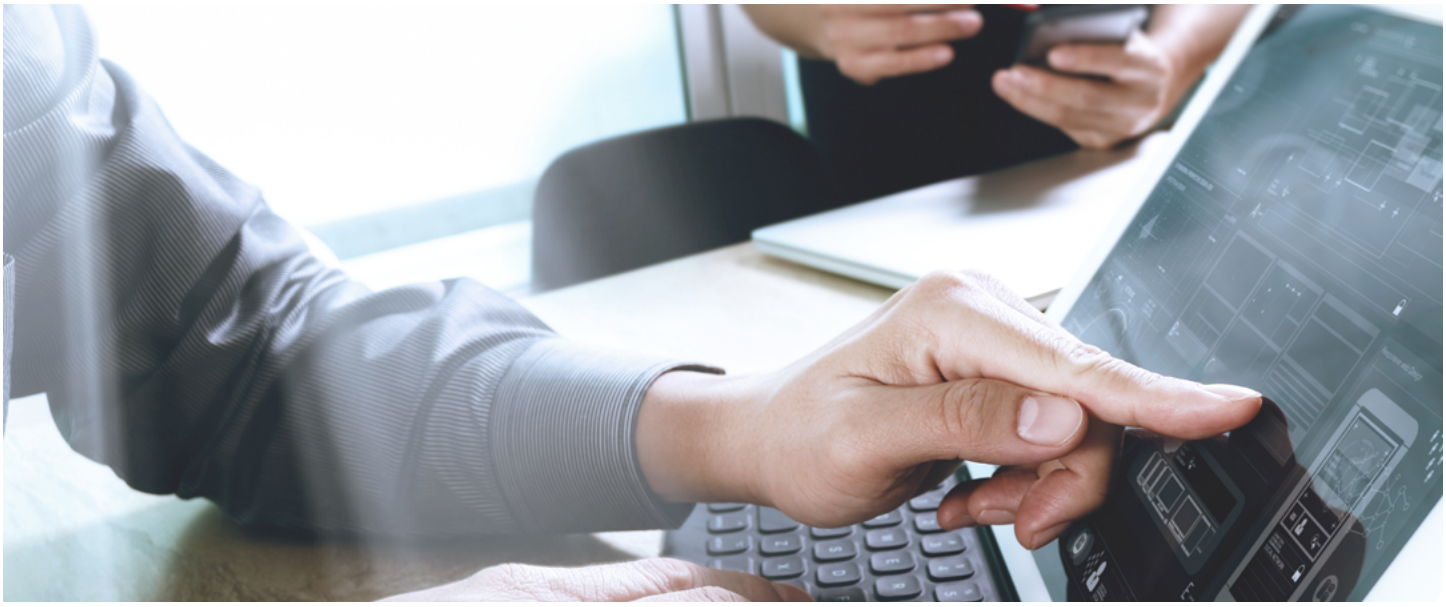
Data masking is the conventional approach often used for mitigating the security risks of working with production data. However, masking all of the PII contained in the transaction data feeds used by payment processing systems is a monumental task. Transaction data feeds are complex, nested, fixed file data structures that contain control codes, record types, accumulated transaction values, and calculations for reward points and cash-back incentives along with real card holder and merchant account numbers and credit information. Finding and masking the sensitive information in this complex data stream while preserving the referential integrity of the data values is both daunting and time consuming.

Payment Processing Test Data Requirements

With these issues in mind, the QA team decided to evaluate commercial solutions for provisioning the volume and variety of test data they needed for testing. They identified the following test data provisioning requirements for testing their payment processing application.

1. Test data must be formatted according to internal data interchange specifications
2. Specified values are needed for methodically testing control codes and input variables
3. Test data must follow specific workflow patterns used in processing payments
4. Test data must allow for assertions to validate specific test outcomes
5. Real-time calculations must be performed to validate accumulated totals
6. Queries to an external database must be integrated with the use of test data
7. A “feature file” containing test case scenarios must be used to control the data
8. Referential integrity between all columns and data tables must be assured
9. Test data must be devoid of personal customer and merchant credit information
10. Test data files must scale to 20 million rows to simulate heavy transaction loads

QA team leaders first explored using one of the major Test Data Management (TDM) systems to securely provision test data from the production environment. They encountered all of the issues cited above and found these systems were not able to successfully create 20 million rows of masked, patterned and controlled test data.



THE GENROCKET SOLUTION

The team then evaluated the GenRocket TDG platform and the use of real-time synthetic test data to meet their needs. They presented their requirements to GenRocket and within three weeks GenRocket was able to provide them with a fully working proof of concept. First, GenRocket created a custom *test data generator* to recreate the "feature file" used to control test case conditions. This new data generator works in combination with custom *test data receivers* that format the data to match the company's data interchange specification. Then a custom script was created to implement an API integration between their testing tools and the GenRocket TDG platform along with *test data scenarios* that contain instructions for generating test data in the required volume and variety that is needed for comprehensive testing.

GenRocket worked closely with Channel Partner, one of its premiere testing partners to produce an operational test environment that was ready for immediate use by the testing team. Here is a summary of the steps taken to set up their new test data provisioning platform:

1. First Channel Partner and GenRocket used the financial company's data model to create GenRocket *domains* and *attributes* to simulate their payment processing database.
2. Then the Channel Partner team used GenRocket *data generators* to model the company's business data for each GenRocket *attribute*. GenRocket created a custom *FeatureFileGen* generator for the purpose of reading "Feature File" data into GenRocket attributes.
3. The GenRocket team then implemented custom *data receivers* to create formatted data.
4. Together, GenRocket and Channel Partner created GenRocket *test data scenarios* using the above components to consume "Feature File" data and produce the test data output.
5. Finally, the GenRocket team created a groovy script that used the GenRocket API to orchestrate the entire process.

The new custom GenRocket components created for this solution are as follows:

- **FeatureFileCreatorScript:** Used to generate a "Feature File" of 1 to 1,000,000 rows or more
- **FeatureFileGen:** The GenRocket generator used to query columns in a "Feature File"
- **SegmentDataCreatorReceiver:** Creates various segment files to represent the many data elements used in a typical payment transaction process
- **SegmentMergeReceiver:** Merges multiple segment files in the proper sequence and hierarchy to produce a consolidated payment transaction file
- **GenRocket API Script (300 Lines):** Integrates the test data generation process with test cases and ensures proper relationships of data in a dynamic data hierarchy

The following workflow diagram illustrates the process used by testers to create payment transaction data for any type of testing, including functional, integration, regression, performance, security and compliance testing.

1. Generate Feature Files

• Tester creates Feature File by hand for their specific test case.

• User can also run the FeatureFileCreatorScript to generate multiple Feature Files for testing a large load.

2. Run GenRocket API Script

• User runs GenRocket API script which calls GenRocket Scenarios.

• The script manages the complex hierarchy of data for the final output file.

3. GenRocket Scenarios Run and Generate data

• Scenarios run when instructed by the API script and start generating data in a segment format.

4. FeatureFileGen

• Certain Attributes have the FeatureFileGen assigned to them.

• This Generator loads Feature File data which is used by other Generators to query a database to generate data.

5. Segment-DataCreator-Receiver

• This Receiver segments the raw data to be consumed by SegmentMergeReceiver.

6. SegmentData-MergeReceiver

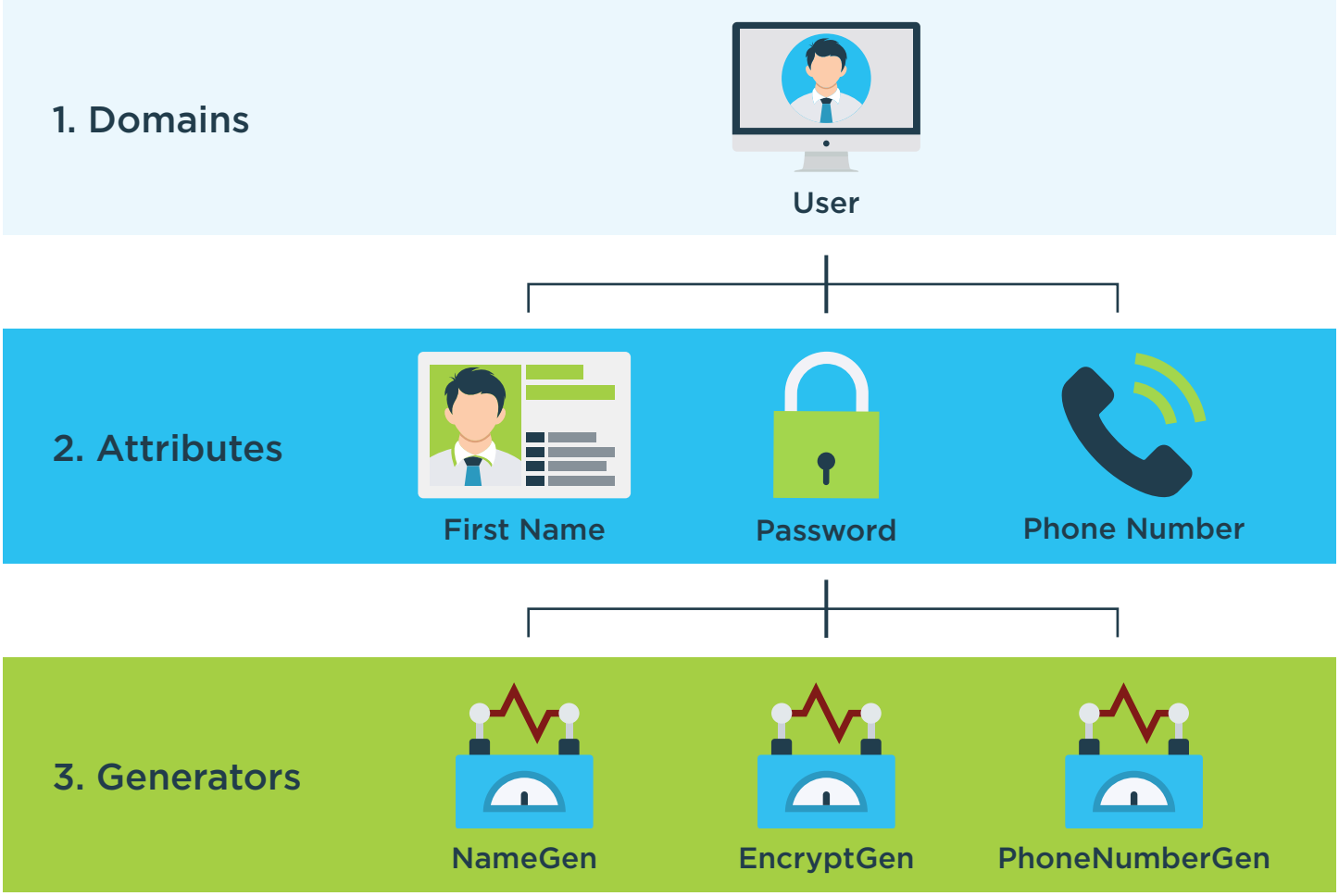
• This Receiver merges the segments from the Segment-DataCreatorReceiver into a single file.

7. File Output

• GenRocket outputs a single file that can be used for testing.

Advanced Technology Streamlines Test Data Provisioning

The test data solution created for this complex software testing challenge illustrates the power and flexibility of the component-based architecture used by the GenRocket TDG platform.



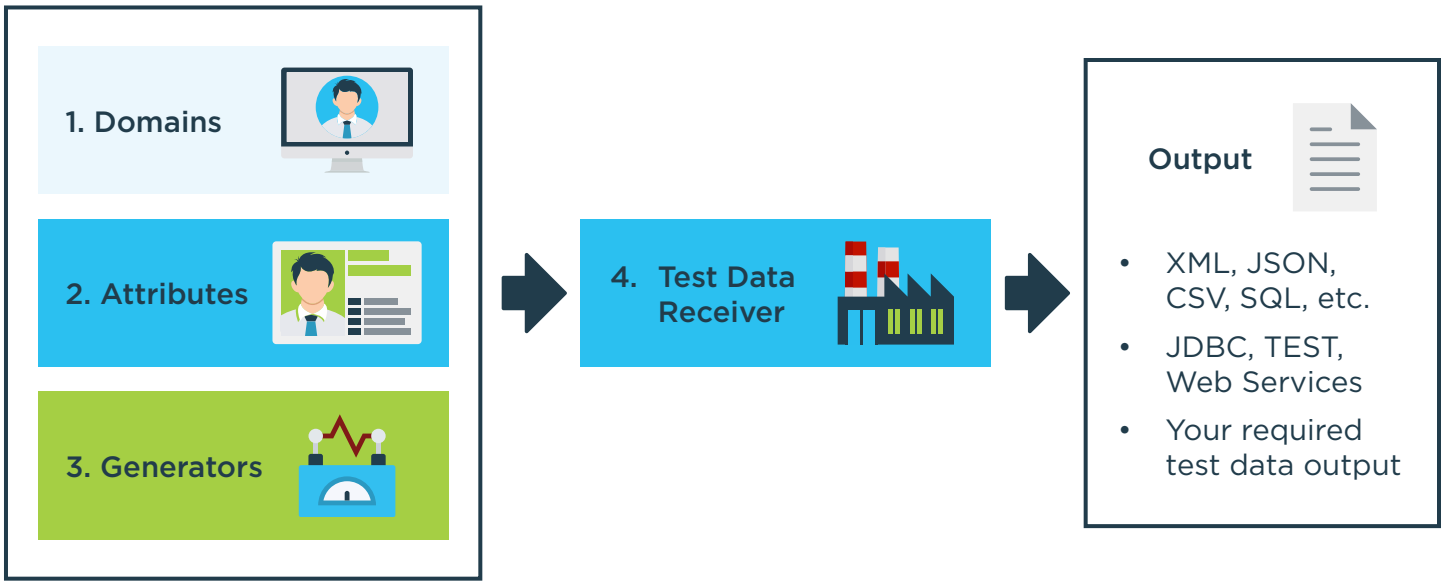
GenRocket TDG breaks down the process of test data generation into 5 components that provide total control over the nature of the data to be generated.

Domains are at the highest level and define the category of data to be generated. They are analogous to a data table.

Attributes define the data elements of the domain that will be generated and are analogous to columns in a data table.

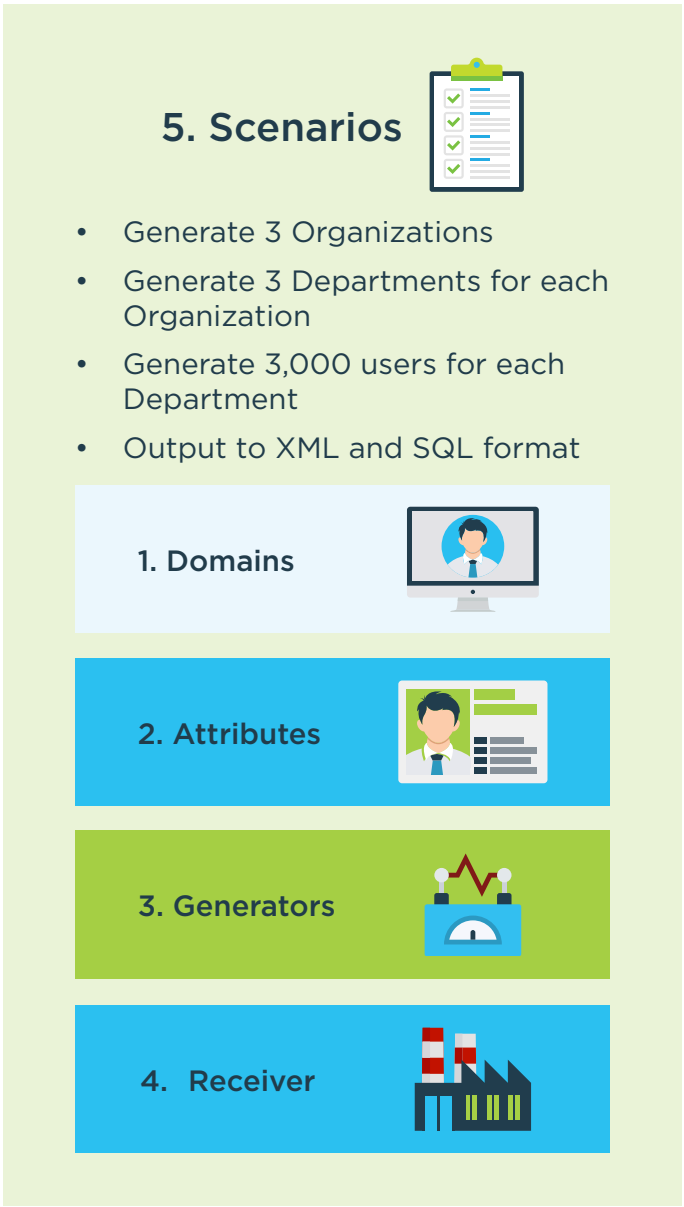
Generators are components that automate the generation of test data according to *domain* and *attribute* definitions. The GenRocket platform contains over 200 data generators and more are being developed all the time.

Receivers are GenRocket components that format data in the output needed for testing (e.g., XML, JSON, CSV, etc.). There are currently more than a dozen GenRocket receivers including the custom receivers created to simulate the payment transaction data described in this case study.

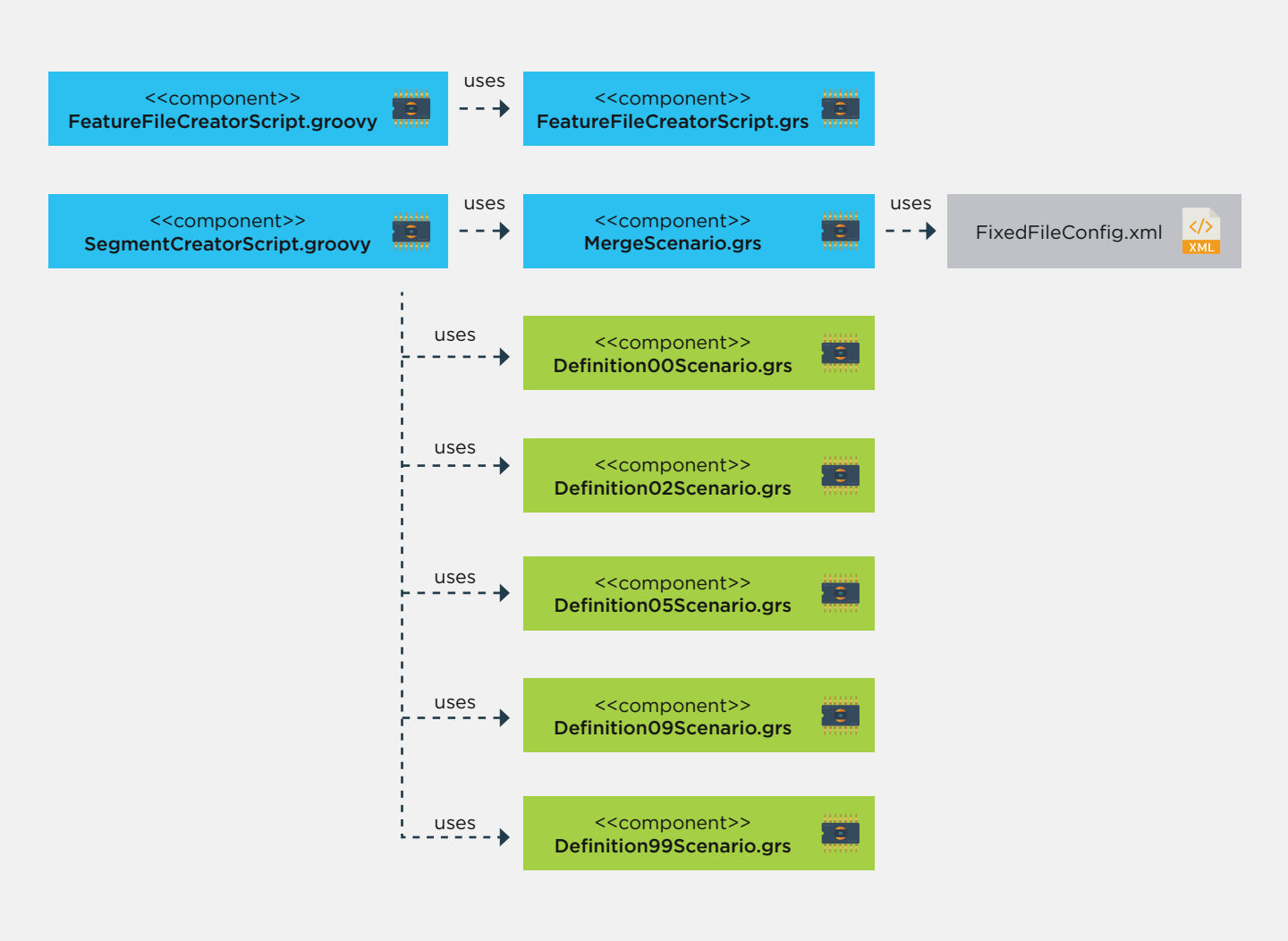


Finally, GenRocket **scenarios** provide all of the instructions for generating the final test data output needed for a given test case as defined by domains, attributes, generators and receivers. *Scenarios* can be used to invoke various data generators and receivers, perform real-time calculations, query databases, blend synthetic data with production data, create patterns and permutations of data and specify the number of data rows needed for testing.

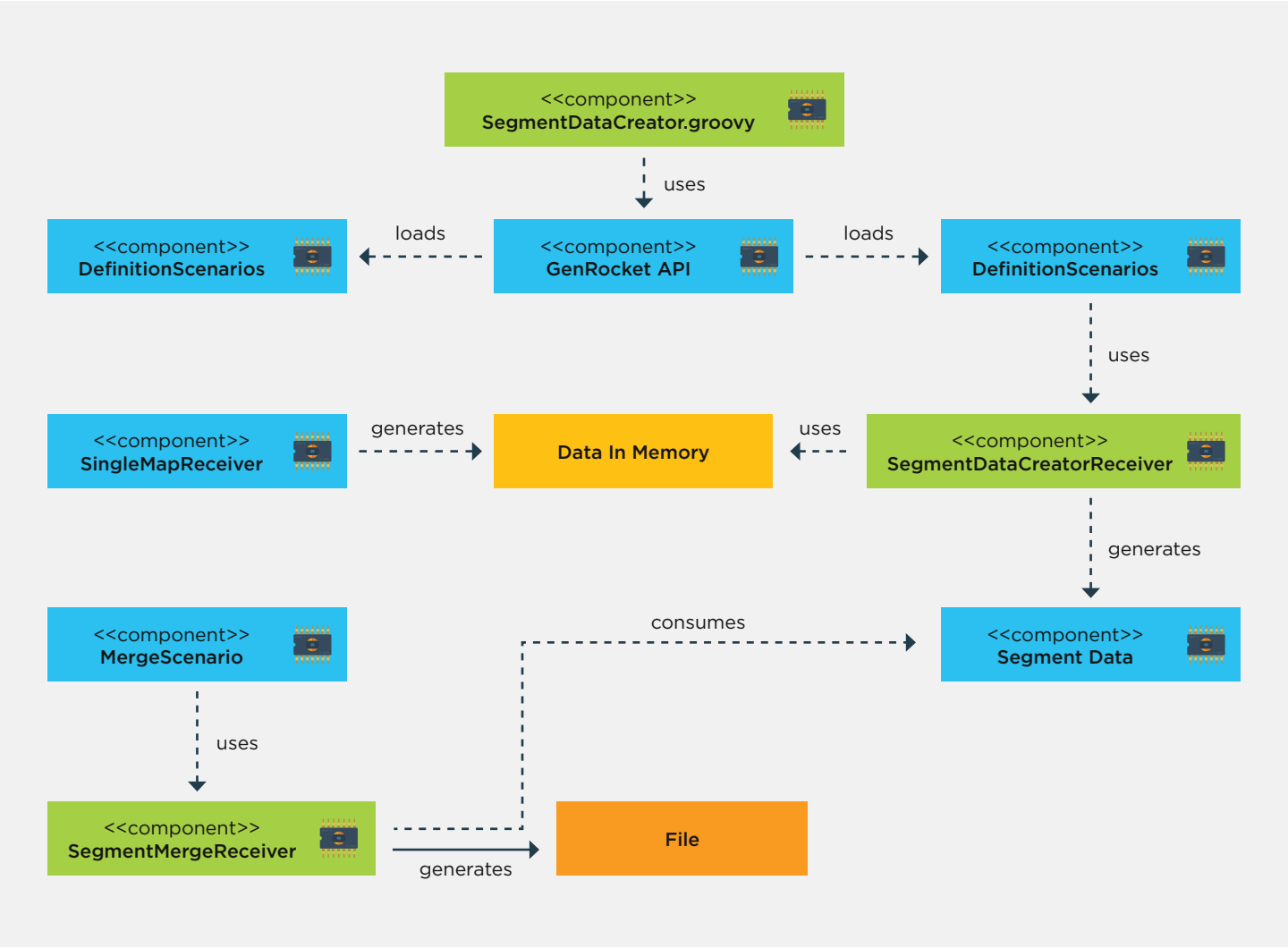
GenRocket's component-based architecture empowers GenRocket to customize test data for any complex data feed in a way no other TDM or TDG platform can duplicate. GenRocket has the ability to rapidly develop custom data generators and receivers to create controlled, patterned and conditioned test data to simulate any data feed with 100% secure synthetic data. The ability to provision test data on-demand and using a self-service model streamlines the entire testing process. And a powerful GenRocket API provides seamless integration with all of the latest test automation tools and frameworks.



The diagram below shows the relationship between the components created by GenRocket to solve this sophisticated payment transaction test data challenge. Together they enable rapid provisioning for any pattern of test data needed to simulate any payment transaction workflow with secure, real-time synthetic test data.



The process flow for how these components are used by the GenRocket platform during actual test data generation is illustrated by the following diagram. At the top of the diagram, a Groovy script using the GenRocket API controls the process in a way that is fully adaptable to a variety of testing procedures.



Using the API, the script loads and executes GenRocket components to simulate the “Feature File” and configures the test data feed with appropriate record types and input values. Multiple record types that designate data feed segments are required to simulate the entire payment transaction process and test cases used for testing them. Segments are generated by one of several *SegmentDataCreatorReceiver* components.

The *SegmentMergeReceiver* is a component used to assemble a composite test data file containing multiple nested segments as specified by the “Feature File” and generates a complete transaction data feed that is ready for use by testers.

A major benefit of the GenRocket solution is the ability to create very high volumes of data in a very short span of time. In the table below, the time to create the “Feature File” and data segments and then merge them into a composite test data file that represents complete payment transactions are given for 1 million rows to 20 million rows of data.

# Transactions	Feature File Creation Time	Segment File Creation Time	Merge File Creation Time	Feature File Size	# Segment Directories	FILE SIZE
1,000,000	2m:54s	17m:52s	1m:1s	77.4MB	11	420.4MB
5,000,000	14m:56s	1h:35m:14s	4m:55s	386.9MB	51	2.1GB
10,000,000	30m:44s	3h:13m:7s	10m:4s	773.9MB	101	4.2GB
20,000,000	57m:3s	6h:29m:32s	20m:27s	1.55GB	202	8.41GB

The total time to create 1 million rows is less than 30 minutes while the time to create 20 million rows is less than 8 hours. Once a final payment transaction test file has been created, it can be reused or regenerated in its original state for subsequent testing (e.g., nightly regression test) at any time and on-demand. When compared with the manual data provisioning process used previously, the team was saving 320 man-hours per year representing a major cost and time savings for the organization.

This sophisticated GenRocket TDG solution can be replicated for any data feed testing requirement in financial services, healthcare, eCommerce, or any environment where complex data structures must be replicated under controlled conditions using secure, real-time on-demand synthetic test data.