App Config Test

Configuration Validation Automation

Technical Implementation Guide

MQE Unified OAO Test Automation Framework
Date: October 2025

App Config Test - Technical Implementation Guide

⚠️ **Confidentiality Notice**
This document contains sensitive technical information including internal architecture details, proprietary implementation strategies, and code references. It is intended for internal use only and should not be shared outside the organization.

Table of Contents

1. Executive Summary

1.1 What is App Config?

Think of App Config as a recipe book that tells the application how to behave. Just like a recipe tells you what ingredients to use and how much, App Config tells the app:

- Which API endpoints to connect to
- What features should be enabled or disabled
- How the app should look and behave
- What settings are available for each platform

Why Test It? If the recipe is wrong, the app won't work correctly. This test makes sure the "recipe" matches what we expect!

1.2 What Does This Test Do?

The `AppConfigTest` class is like a **quality checker** that validates application configuration parameters across multiple platforms and brands. It ensures that configuration values match expected baselines, detecting discrepancies that could cause application issues in production.

Simple Analogy:
Imagine you're a restaurant inspector checking if a restaurant follows the recipe correctly. You have a list of expected ingredients (baseline config), and you check if the actual recipe matches. If anything is different, you flag it!

Key Benefits:

- **Proactive Detection:** Identifies configuration issues before production deployment
- **Fast Validation:** Validates configurations in seconds via API or UI-based tests
- **Change Detection:** Detects all changes from last release, reviewed and approved by Dev
- **Multi-Platform Support:** Validates configs across Android, iOS, Apple TV, Android TV, Fire TV, Roku, and Web Connected TVs
- **Automated Reporting:** Generates detailed comparison reports and email notifications

2. Overview

2.1 Quick Start - How It Works

Step-by-Step Process:

1. **Get Expected Config:** We have a list of what config values SHOULD be (baseline)
2. **Get Actual Config:** We fetch what config values ACTUALLY are (from app or API)
3. **Compare:** We check if Expected = Actual
4. **Report:** We generate a report showing any differences

That's it! Simple concept, powerful results.

2.2 Purpose

The AppConfigTest class validates application configuration parameters to ensure:

- Configuration values match expected baselines
- Platform-specific configurations are correctly applied
- Brand-specific settings are properly configured
- API endpoints and URLs are correctly set
- Feature flags and screen flags are appropriately enabled/disabled
- No unauthorized changes are introduced between releases

2.2 Problem Statement

Configuration discrepancies can cause application issues in production. This test suite addresses this by:

- Validating configurations proactively during CI/CD pipelines
- Comparing configurations between environments (RC vs Production)
- Detecting unexpected changes and flagging them for review
- Maintaining baseline configurations for regression testing

3. Technical Architecture

3.1 Class Hierarchy (Simplified)

> **Think of it like Russian nesting dolls:**
>
> - **TestContext** (outermost doll) - Basic test setup
> - **BaseTest** - Common test functionality
> - **AnalyticsBaseTest** - Analytics and validation helpers
> - **AppConfigTest** (innermost doll) - Our specific config test
>
> Each layer adds more functionality. AppConfigTest gets all the features from parent classes!

```
AppConfigTest extends AnalyticsBaseTest extends BaseTest extends TestContext
```

3.2 Core Components (What They Actually Do)

3.2.1 AnalyticsBaseTest (The Helper Class)

Location: `com.viacom.unified.tests.common.AnalyticsBaseTest`

What it does: This is like a toolbox with useful tools for testing. It provides:

- `validator()` : **What it does:** Returns a validator object that checks if configs match
  Think of it as: Your quality inspector who compares expected vs actual
- `configMatcher()` : **What it does:** Returns a matcher that compares two configs
  Think of it as: A diff tool that shows what changed between versions
- `matchConfig()` : **What it does:** Helper method to create a test rule (what to check)
  Think of it as: A way to say "I expect this config key to have this value"
- `brandMapping()` : **What it does:** Helper to specify different values for different brands
  Think of it as: A way to say "BET_PLUS should have value X, but VH1 should have value Y"
- `platforms()` : **What it does:** Helper to specify which platforms to test
  Think of it as: A way to say "only check this on Android and iOS"

3.2.2 AnalyticsValidator

Location: `com.viacom.unified.service.analytics.AnalyticsValidator`

Key Methods:

- `validateAppConfigParameters(String eventType, List<MatchConfig> expectedList, SoftAssert softAssert, boolean overrideExtraCheck)` : Validates config from proxy logs
- `validateAppConfigParameters(String eventType, String appVersion, String configUrl, List<MatchConfig> expectedList, SoftAssert softAssert, boolean overrideExtraCheck)` : Validates config from API URL

3.2.3 ConfigMatcher

Location: `com.viacom.unified.service.analytics.ConfigMatcher`

Key Method:

- `compareAppConfigParameters(String configUrlRc, String configUrlProd, SoftAssert softAssert)` : Compares two configurations and generates DTR report

3.2.4 AnalyticsDataCollector

Location: `com.viacom.unified.service.analytics.AnalyticsDataCollector`

Key Method:

- `getActualAppConfig(String configUrl)`: Fetches actual configuration from API endpoint

3.3 Data Flow

UI-Based Approach

```
 Latest App Details → Synergy Driver Creation →
Actual Config from Proxy log → Analytics Validator → Test Report
```

API-Based Approach

```
 Apache HTTPClient → Generate Request and fetch directly from Neutron →
Analytics Validator → Test Report
```

Common Input

```
 Expected Config (Baselined from current) → Analytics Validator
```

4. Test Methods

4.1 appConfigValidationTest

Test ID: TMS-23124254

Platforms: Android, Android TV, Fire TV, iOS, Apple TV

Brands: BET_PLUS, VH1

Locales: ALL_LOCALES

Group: CONFIG

Implementation:

- Launches the application
- Verifies page load (Welcome screen for SVoD, Home screen for AVoD)
- Validates app configuration parameters from proxy logs
- Uses `initExpectedMatchConfig()` for expected values

```
 @Test(groups = {GroupConstants.CONFIG})
@TmsLink("23124254")
@Feature(FeatureConstants.CONFIG)
@Platforms({ANDROID, ANDROID_TV, FIRE_TV, IOS, APPLE_TV})
@AppBrand({BET_PLUS, VH1})
@Locales({ALL_LOCALES})
public void appConfigValidationTest() {
    launchApp();
    verifyPageLoad(TestRunInfo.isSVoD() ? welcomeScreen() : homeScreen());
    SoftAssert softAssert = new SoftAssert();
    validator().validateAppConfigParameters(APP_CONFIG_EVENT_TYPE,
        initExpectedMatchConfig(), softAssert, false);
    softAssert.assertAll();
}
```

4.2 appConfigValidationAPITest

Test ID: TMS-23181578

Platforms: Android, Android TV, Fire TV, iOS, Apple TV

Brands: BET_PLUS, BET, VH1

Locales: ALL_LOCALES

Group: CONFIG

Implementation:

- Retrieves latest app version from AppHub
- Constructs config URL with dynamic parameters (brand, platform, version, etc.)
- Validates configuration via API call to Neutron endpoint
- Generates dummy session ID for the request
- Uses `initExpectedMatchConfig()` for expected values

Config URL Format:

```
 https://neutron-api.paramount.tech/api/bff/mobile/2.2/config?
brand=[brand]
&platform=[platform]
®ion=us
&version=[version]
&deviceWidth=[width]
&deviceType=[deviceType]
&accessLevel=unsubscribed
&profile.firstInstalledAppVersion=[version]
[store]
[profile]
&sessionId=[session]
```

4.3 appConfigValidationRokuS3Test

Test ID: TMS-23181578

Platforms: Roku

Brands: BET_PLUS, VH1

Locales: ALL_LOCALES

Group: CONFIG

Implementation:

- Uses RC version from build number
- Fetches config from S3 bucket:
  `https://emergingplatforms627274808695public.s3.amazonaws.com/roku/configs/[brand]/US/`

- Validates Roku-specific configuration keys
- Uses `initExpMatchFileConfig()` for expected values from file

4.4 appConfigValidationWCTVBetPlusTest

Test ID: TMS-23181578

Platforms: VIZIO, TIZEN_TV, COMCAST, COX, HISENSE_TV, LG_WEBOS

Brands: BET_PLUS

Locales: ALL_LOCALES

Group: CONFIG

Implementation:

Based on `WCTV_LEGACY_REPORT` flag:

- **Legacy Mode:** Validates config against baseline
- **Comparison Mode:** Compares RC config with Production config

4.4.1 Comparison Mode (Default)

Compares configurations between Preview (RC) and Production environments:

- Fetches config from both environments
- Generates DTR (Data Test Report) with differences
- Reports only changed values, not entire config
- Sends email notification with changes

5. Configuration Details

5.1 Constants

Constant	Value	Description
NEUTRON_HOST	https://neutron-api.paramount.tech	Base URL for Neutron API
NEUTRON_VERSION	3.5	API version for Neutron endpoints
APP_CONFIG_EVENT_TYPE	AppConfigParameters	Event type identifier for app config validation
CONFIG_URL	Neutron API config endpoint template	Template URL with placeholders for dynamic values
WCTV_LEGACY_REPORT	false	Flag to enable legacy reporting mode

5.2 Platform-Specific URL Construction

5.2.1 Session ID Format

- **iOS/Apple TV:** Uppercase UUID
- **Other Platforms:** Standard UUID format

5.2.2 Store Parameter

- **Android/Android TV:** `&store=google`
- **Fire TV:** `&store=amazon`
- **iOS/Apple TV:** No store parameter

5.2.3 Device Width

- **Android/Fire:** 1440
- **iOS:** 1500
- **Android TV/Fire TV:** 0
- **Apple TV:** 1920

5.2.4 Device Type

- **Android/Fire:** PHONE
- **All TV Platforms:** TV

5.2.5 Profile Parameter

- **iOS/Apple TV:** `&profile.hasProfile=false`
- **Other Platforms:** No profile parameter

5.3 WCTV Platform URLs

Production URLs:

- **Comcast:** `https://comcast.bet.plus/?json=true`
- **Vizio:** `https://vizio.bet.com/?json=true`
- **Tizen TV:** `https://samsungtv.bet.plus/?json=true`
- **Hisense TV:** `https://hisensetv.bet.plus/?json=true`
- **LG WebOS:** `https://lgtv.bet.plus/?json=true`
- **Cox:** `https://cox.bet.plus/?json=true`

Preview/RC URLs:

Format: `https://[platform]-bet-plus-10558-webplex-app.webplex.vmn.io/?json=true`

6. Validation Logic

6.1 Match Types

MatchType	Description	Usage
EQUAL	Exact string match	Most config values (URLs, IDs, boolean flags)
MATCH	Regex pattern match	Dynamic values (IDs, timestamps, URLs with variables)
SKIP	Skip validation	Optional or unstable configs
IGNORE	Ignore if missing	Platform-specific optional configs
OPTIONAL_EQUAL	Optional exact match	Configs that may not exist

OPTIONAL_MATCH	Optional regex match	Dynamic values that may not exist

6.2 Validation Process

1. **Fetch Actual Config:** Retrieves actual configuration from API or proxy logs
2. **Load Expected Config:** Loads baseline expected values
3. **Iterate Through Expected:** For each expected parameter:
    - Check if parameter exists in actual config
    - Validate value based on MatchType
    - Report success or failure
4. **Check Extra Parameters:** Identifies unexpected parameters in actual config
5. **Generate Report:** Creates detailed validation report

6.3 Validation Failure Handling

Uses TestNG's `SoftAssert` to:

- Collect all validation failures
- Continue validation even after failures
- Report all failures at the end
- Generate comprehensive failure report

7. Platform-Specific Implementation

7.1 Mobile Platforms (Android, iOS)

Supported Brands: BET_PLUS, BET, VH1

Key Configurations Validated:

- App name and identifiers
- API endpoints (Neutron-based)
- Feature flags and screen flags
- Player configurations
- Authentication options
- Tracker configurations (Braze, Adjust, Comscore, etc.)

7.2 TV Platforms (Android TV, Fire TV, Apple TV)

Supported Brands: BET_PLUS, BET, VH1

Additional Validations:

- TV-specific screen flags
- Enhanced navigation features
- Remote control configurations
- Playback features (pause ads, still watching overlay)

7.3 Roku Platform

Supported Brands: BET_PLUS, VH1

Special Considerations:

- Config stored in S3 bucket
- JSON format configuration

- Filtered key validation (only validates specific keys)
- File-based expected config

7.4 Web Connected TV (WCTV)

Supported Platforms: VIZIO, TIZEN_TV, COMCAST, COX, HISENSE_TV, LG_WEBOS

Supported Brands: BET_PLUS

Implementation:

- Uses comparison mode by default
- Compares RC vs Production configs
- Generates DTR report with differences
- Sends email notifications with changes

8. Match Configuration System

8.1 What is MatchConfig? (In Simple Terms)

MatchConfig is like a checklist item:

- **Parameter:** What config key are we checking? (e.g., "ageGatePass")
- **MatchType:** How should we check it? (exact match, pattern match, skip?)
- **Expected:** What value should it have? (e.g., "18")
- **Description:** Optional note about why we're checking this

Real Example: "Check if ageGatePass equals '18' for BET_PLUS, BET, and VH1 brands"

8.2 Helper Methods Explained

setMatchConfig() - The Main Helper

Purpose: Adds a validation rule to our checklist

Basic Syntax:

```
setMatchConfig(expectedAppConfig,
    "config\\key\\path",       // What to check
    MatchType.EQUAL,          // How to check
    brandMapping("value", BRAND));  // Expected value
```

What it does: Adds a rule saying "I expect this config key to equal this value for this brand"

brandMapping() - Specify Different Values Per Brand

Purpose: Set different expected values for different brands

Examples:

- `brandMapping("18", VH1, BET, BET_PLUS)` - All three brands should have "18"
- `brandMapping("true", platforms(ANDROID, IOS), BET_PLUS)` - BET_PLUS on Android/iOS should have "true"

platforms() - Specify Which Platforms

Purpose: Limit validation to specific platforms

Example:

- `platforms(ANDROID, IOS)` - Only check on Android and iOS
- `platforms(ANDROID_TV, FIRE_TV, APPLE_TV)` - Only check on TV platforms

8.3 initExpectedMatchConfig() Method

This method builds the expected configuration baseline programmatically. It contains over 100 configuration validations covering:

- **Application Identity:** App name, ID, URL key
- **API Endpoints:** AB tests, favorites, content items, property feeds, search
- **Event Collector:** Heartbeat intervals, app IDs, main URLs
- **Module URLs:** Privacy policy, terms of service, copyright notices
- **Feature Flags:** Search service, onboarding, trackers
- **Screen Flags:** Home screen features, carousels, popups
- **Video Configuration:** Player settings, overlay configurations
- **Authentication:** MVPD settings, subscription options
- **Purchase Descriptions:** Subscription messaging

8.4 Real-World Example - Understanding the Code

Example Code:

```
setMatchConfig(expectedAppConfig,
    "\\data\\appConfiguration\\ageGatePass",
    MatchType.EQUAL,
    brandMapping("18", VH1, BET, BET_PLUS));
```

What This Means in Plain English:

1. Add a validation rule to our checklist
2. Check the config key: `data.appConfiguration.ageGatePass`
3. Use exact match (EQUAL) - must match exactly
4. Expected value is "18" for brands VH1, BET, and BET_PLUS

Result: The test will verify that for VH1, BET, and BET_PLUS apps, the ageGatePass config equals "18"

8.5 More Complex Example

```
setMatchConfig(expectedAppConfig,
    "\\data\\appConfiguration\\ageGatePass",
    MatchType.EQUAL,
    brandMapping("18", VH1, BET, BET_PLUS));
```

This validates that `ageGatePass` equals "18" for VH1, BET, and BET_PLUS brands.

8.4 Dynamic URL Generation

The test includes helper methods to generate expected URLs dynamically:

- `getAbTestsNotifyUrl()`: Generates AB tests notify URL
- `getAddFavoriteContentUrl()`: Generates favorite content URL
- `getPropertyFeedUrl()`: Generates property feed URL
- `getSearchServiceUrl()`: Generates search service URL
- `getModuleUrl()`: Generates module URLs with regex patterns

9. Code Examples

9.1 Beginner's Guide to Adding a New Config Validation

Step-by-Step Process:

1. **Find the right place:** Open `initExpectedMatchConfig()` method
2. **Identify the config key:** Find the config key path (e.g., `\\data\\appConfiguration\\yourKey`)
3. **Decide the match type:** Exact match (EQUAL) or pattern match (MATCH)?
4. **Determine expected value:** What should the value be?
5. **Specify brands/platforms:** Which brands/platforms should this apply to?
6. **Add the code:** Use setMatchConfig with brandMapping

9.2 Adding a New Configuration Validation

```
 // Add to initExpectedMatchConfig() method
setMatchConfig(expectedAppConfig,
    "\\data\\appConfiguration\\yourConfigKey",
    MatchType.EQUAL,
    brandMapping("expectedValue", VH1, BET, BET_PLUS));
```

9.2 Platform-Specific Validation

```
setMatchConfig(expectedAppConfig,
    "\\data\\appConfiguration\\featureFlag",
    MatchType.EQUAL,
    brandMapping(TRUE, platforms(ANDROID, IOS), BET_PLUS),
    brandMapping(FALSE, platforms(ANDROID_TV, FIRE_TV), BET_PLUS));
```

9.3 Regex Pattern Validation

```
setMatchConfig(expectedAppConfig,
    "\\data\\appConfiguration\\id",
    MatchType.MATCH,
    brandMapping(ID_PATTERN, VH1, BET, BET_PLUS));
```

9.4 Skipping Optional Config

```
setMatchConfig(expectedAppConfig,
    "\\data\\appConfiguration\\optionalConfig",
    MatchType.SKIP,
    brandMapping("", BET_PLUS));
```

9.5 Custom URL Generation

```
private String getCustomUrl() {
    return String.format("%s/api/%s/endpoint?region=US&brand=%s&platform=%s&version
        NEUTRON_HOST, NEUTRON_VERSION,
        getConfigBrand(), getConfigPlatform(),
        TestRunInfo.getAppVersion());
}
```

10. Troubleshooting Guide

10.1 Test Failure: Config Not Found

Symptoms:

- Test fails with "Could not find the AppConfig event"
- Actual config is null or empty

Diagnosis:

- Verify app launched successfully
- Check proxy logs for AppConfigParameters event
- For API tests, verify URL is accessible
- Check network connectivity

Solutions:

- Ensure app is launched before validation
- Verify proxy is capturing network traffic
- Check API endpoint accessibility
- Validate URL construction logic

10.2 Test Failure: Value Mismatch

Symptoms:

- Specific config values don't match expected
- Regex patterns not matching

Diagnosis:

- Compare actual vs expected values
- Check if value changed intentionally
- Verify regex pattern correctness
- Check platform/brand mapping

Solutions:

- Update expected value if change is intentional
- Fix regex pattern if incorrect
- Verify brand/platform mapping logic
- Check for environment-specific differences

10.3 Test Failure: Unexpected Parameters

Symptoms:

- Test reports unexpected parameters
- New config keys added to actual config

Diagnosis:

- Review actual config for new keys
- Check if new keys are expected
- Verify if keys should be added to baseline

Solutions:

- Add new keys to expected config if valid
- Use SKIP or IGNORE match type if optional
- Update baseline configuration

10.4 API Test Failures

Symptoms:

- HTTP errors when fetching config
- Invalid URL construction
- Authentication failures

Solutions:

- Verify URL template correctness
- Check parameter replacement logic
- Validate session ID format
- Ensure proper authentication headers

10.5 WCTV Comparison Issues

Symptoms:

- Comparison report shows incorrect differences
- RC config not accessible

Solutions:

- Verify RC URL is correct
- Check build version mapping
- Ensure both environments are accessible
- Review DTR report generation logic

11. Best Practices

11.0 Glossary - Common Terms Explained

Term	Meaning	Simple Explanation
App Config	Application Configuration	Settings that tell the app how to behave
Baseline	Expected Configuration	The "correct" config values we expect
MatchConfig	Validation Rule	A single rule saying "check if config X equals value Y"
MatchType	How to Compare	EQUAL = exact match, MATCH = pattern match, SKIP = don't check
brandMapping	Brand-Specific Values	Different values for different brands (BET_PLUS vs VH1)
Proxy Logs	Network Traffic Capture	Records of network requests the app makes
RC	Release Candidate	Pre-production version for testing
DTR Report	Data Test Report	Report showing differences between configs

11.1 Configuration Maintenance

- **Update Baselines Regularly:** Keep expected configs updated with approved changes
- **Version Control:** Maintain configuration baselines in version control
- **Document Changes:** Document all intentional configuration changes
- **Review Changes:** Review configuration changes before updating baselines

11.2 Test Execution

- **Run in CI/CD:** Execute config validation in CI/CD pipelines
- **Early Detection:** Run tests early in the release cycle
- **Parallel Execution:** Run tests in parallel for faster feedback
- **Report Review:** Review reports promptly and act on failures

11.3 Code Organization

- **Group Related Configs:** Group related configurations together
- **Use Helper Methods:** Create helper methods for common patterns
- **Comment Complex Logic:** Add comments for complex validation logic
- **Maintain Consistency:** Keep naming and structure consistent

11.4 Reporting

- **Detailed Reports:** Generate detailed validation reports
- **Email Notifications:** Send email notifications for failures
- **Dashboard Integration:** Integrate with dashboard tools
- **Historical Tracking:** Track configuration changes over time

---

Documentation Generated by: Cursor AI (Composer)

Date: October 2025

Version: 1.0