Variant RCT Container

User Guide

Version 0.5 Beta, September 2015

© 2015 Variant, Inc.

# Introduction

## About Variant

Variant Container provides capabilities for instrumenting programmatic *Randomized Control Trials*, or, simply, *controlled experiments*. The need for such instrumentation frequently arises in interactive applications where a value transaction is involved. Consider, for instance, the checkout experience of an eCommerce Web application: each time a change is made to this experience, it may directly affect the sales. In the ideal case, we wouldn’t want to simply replace the existing checkout experience with a new one, but to run the two in parallel and compare the performance of the new experience with that of the existing one. Moreover, we’d want to expose the new experience to as few users as possible, so that if, at the conclusion of the experiment, we decide to discard it, most users will never have seen it.

Implementing this concurrency is harder than it may seem: the many challenges include reliability of the event logging mechanism, the need for flexible experience targeting (if we want only certain users to be eligible for the experiment), handling of multiple experiments. And the list goes on.

Variant solves all these problems in a consistent fashion, thanks to its groundbreaking experiment definition model (XDM) that allows the experiment designer to define the experiment without worrying about any implementation details. In the nutshell, XDM operates on states and tests. States represent application states, e.g. pages of a Web application. Tests are experiments instrumented over these states.

For Variant, a state is a completely opaque entity. All that is known about it is its name and that it may contain properties. The names and semantics of these properties are up to the test designer.

Tests, on the other hand, are actualized and managed by Variant. A test has a control experience—the one Variant assumes will be mapped to the current experience,—and at least one variant experience that represents the new experience under test. Tests also declare on what states they are instrumented as well as relationships with other tests, if they share instrumented states.

## Architecture And Integration

### Life Of an Experiment

A Variant experiment has three distinct phases: design, run and analysis. During design, the test designer creates the test schema file, which contains definitions for all tests to be run on a Variant instance.

At run time, a Variant instance has the following three responsibilities:

* Experience Targeting

The decision what test experience to present to the user. Even the simple case of a single test, the decision may not be entirely straight forward as Variant must also consider user’s qualification for the test as well as experience stability, i.e. if this user has already seen an experience of this test recently, we want to target the user to the same experience. The complexity of this process grows dramatically if multiple tests must be considered.

* Event Logging

Variant instance will do most of the work of logging events like Web page views or button clicks. This experiment data is will later be used at the analysis time.

* Traffic Routing

In some cases Variant will be able to go beyond targeting and actually do the work of sending the user to the targeted experience automatically.

At the analysis time, the test designer will be responsible for transforming the raw event data, generated at run time, into a meaningful test metric. Variant makes no assumptions about the nature of the experiment and what the suitable metric is going to be. For example, many experiments may be interested in the next page conversion rate. The raw data will contain page view events but to transform those to the conversion rate the raw data will have to be manipulated either programmatically or via a SQL query. A data visualization tool then may be used to present the test metrics in an easily comprehensible form.

### Core, Domain and Service APIs.

Variant can be exercised via three different application program interfaces (APIs), Fig. 1.1.

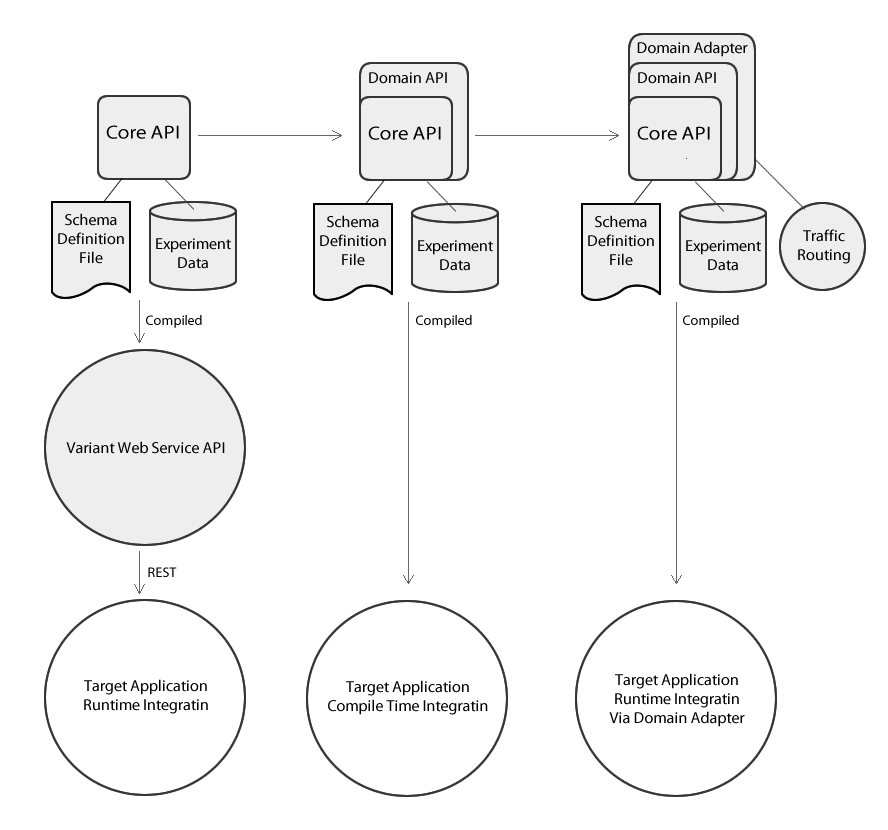


Fig. 1.1 Core API, Routing API and Web Service API.

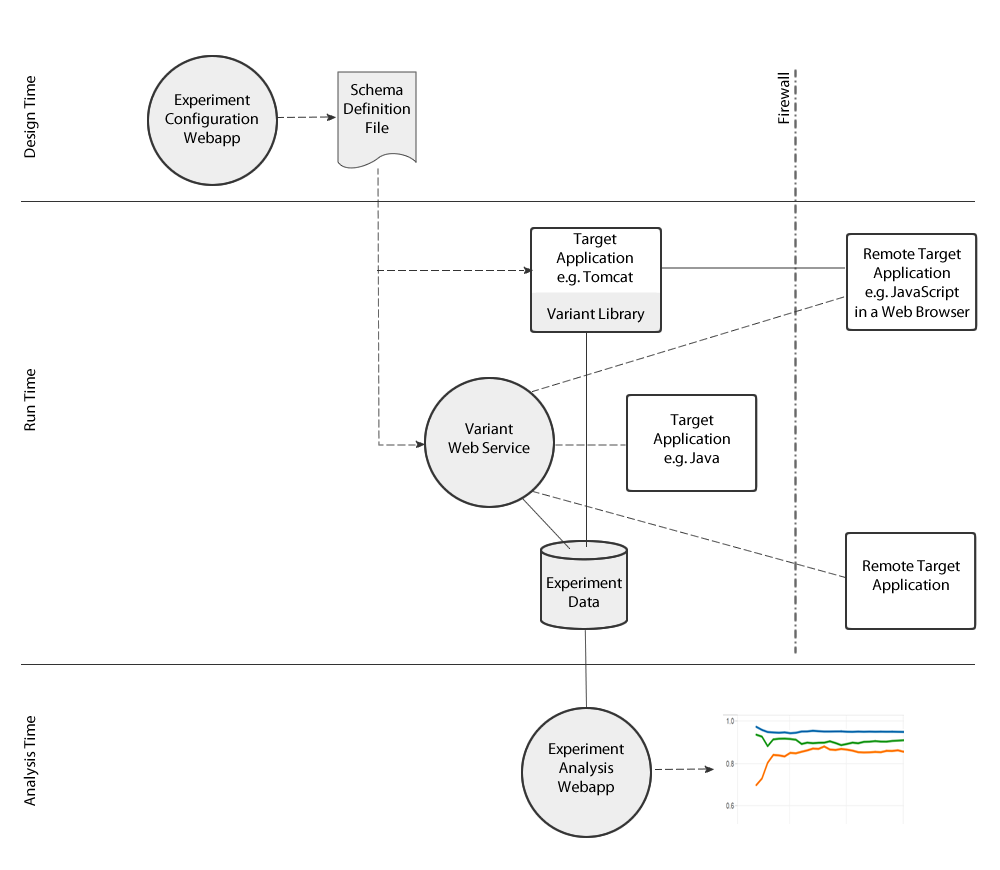
At the center of all is the Core API which implements the Variant XDM without making any further assumptions about the operating domain of the target application. The Core API reads the experiment schema definitions, targets user sessions to test experiences and logs events. But, knowing nothing about the target operating domain, it cannot help in routing the user to the chosen experience. In other words, the application code under test may ask the Core API a question like “what test experience am I?” but it will be up to the application to send the user to the corresponding.

The Core API has very few dependencies, is very small (~120Kb) and extremely fast. Every effort has been made to minimize its overhead, such as its asynchronous event writer.

The Core API’s abstract nature makes it applicable to any operating domain, e.g. a Servlet container, or a native mobile app, but also somewhat hard to use. To make the Core API more relevant to a particular operating domain, it is typically wrapped into a Domain API, which simplifies many of the Core API’s abstractions as they relate to a particular domain. For instance, the Web Domain API is links with the Servlet API and is directly applicable to JEE Servlet containers like Tomcat. Similar API can be developed for other JVM based Web frameworks, such as Play or Lift.

The next step in domain specialization is the Domain Adapter. It is a reusable code component that can further aid the test designer by connecting the Domain API to the domain’s session routing capabilities. For instance, the Web Domain Adapter is, basically, a Servlet filter that handles all aspects of experiment instrumentation: experience targeting, event logging, and (because it knows enough about the operating domain) event traffic routing.

The Core API and the Domain API are both compile time APIs. In cases when direct integration is not feasible, e.g. a non-JVM application, Variant provides a RESTful wrapper of the Core API, called Variant Web Service. It closely mimics the functionality of the Core API and can be integrated with at run time by any client, such as client- or server-side JavaScript, Ruby on Rails or a native mobile app.



asdfasdfadfasdfasdf

# Core

## Overview

The Core API implements the Variant XDM without making any assumptions about the operating domain of the target application. The

## Experiment Design

### Introduction

…

## Experiment Execution

### System Configuration

The Variant engine is configured with a set of application properties. Typically, they will be packaged into properties files and passed to the engine at either compilation or run time. Properties may also be overridden individually at run time, as explained below.

Compile time configuration involves supplying the names of application properties files to the Variant.bootstrap() call. If supplied, each argument is understood as the file name, as a Java classpath resource. When, at run time, the engine is looking for the value of a particular property, these files will be scanned left to right until the first match is found. If a value wasn’t found in any of the supplied files, or if no files were supplied, the default value is used, as defiled in the /variant-default.props file found inside the core jar file.

You may override compile time configuration in the following ways:

* Place the /variant.props file on the class path. The Variant engine will always look for this file and, if found, consult it first, before any of the files supplied at compilation time.
* Pass the –Dvariant.props.resource=... command line parameter to the JVM. The supplied value is understood to be a classpath resource name, which must exist.
* Pass the –Dvariant.props.file=... command line parameter to the JVM. The supplied value is understood to be an OS file, which must exist.

It is an error to supply both ‑Dvariant.props.resource and ‑Dvariant.props.resource.

## Experiment Analysis

## Core API Reference

### Public Packages

|  |  |
| --- | --- |
| Package | Description |
| com.variant.core |  |
| com.variant.core.annotations |  |
| com.variant.core.ext |  |
| com.variant.core.schema |  |
|  |  |
|  |  |

### Package com.variant.core

#### Class Variant

static void bootstrap(String...propsResourceName) : Variant

Bootstrap the Variant engine with an optional list of application properties files as resource names. If more than one resource file is passed, they are scanned left to right until the first match is found. If a property was not found in any of the supplied properties files (or none was supplied), a default value is used, as specified by the /variant-default.props file found inside the core jar file.

Each argument is interpreted as the name of a classpath resource. If one of argument names a resource file that cannot be located on the classpath, the bootstrap() call fails.

Note that at run time, these values may be overridden from the command line, as explained in TODO.

### Class …

# Web API

## Path Matching

Web API has the following implementation of the path matcher:

* Symbol ‘/’ always stands for the path separator. Path must start with ‘/’.
* Any sequence of symbols between two consequtive symbols '/' is taken to be a literal, unless the first letter of the sequence is a tilde '~' in which case the string immediately following the tilde and until, but not including, the next unescaped ‘/’ is considered a regular expression. Complete regular expression syntax is supported[[1]](#footnote--1).
* Although symbol ‘/’ does not have a special meaning in the regular expression grammar, it does for Variant: this is how Variant decides where a regular expression ends. Therefore, if ‘/’ must be included in the regular expression, it must be escaped with the ‘\’ symbol, like any other special character. By including ‘/’ symbols in the regular expression, it is possible to match variable sections in the middle of a path.
* Symbol ‘//’ can be used anywhere, where ‘/’ can be used, and is a shortcut for ‘/~.\*/’. On other words, ‘//’ will match any string. Note that ‘///’ is legal but superfluous because it will be expanded to ‘/~.\*/~.\*/’.
* The very last ‘/’ of the pattern is not significant, i.e. Variant will remove it, if present, from both the pattern and the path after all ‘//’ are expanded. This enables easy prefix match: ‘/user//’ will match any path that starts with ‘/user/’.

Examples:

|  |  |  |
| --- | --- | --- |
| Path | Will Match | Will Not Match |
| /user | /user  /user/ | /user/new |
| /user// | /user  /user/  /user/new | /service/user/ |
| /user//.html | /user/new/error.html | /user/error |

1. See, e.g. http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html. [↑](#footnote-ref--1)