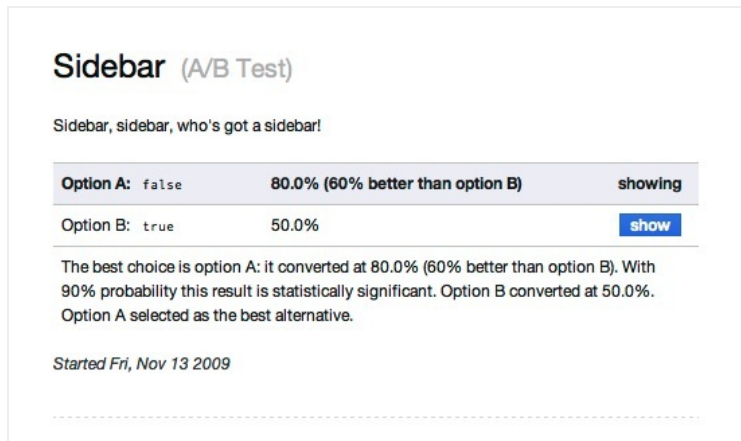


Welcome to Vanity

Vanity is an Experiment Driven Development framework for Rails.



Reading Order

- [2 Minute Demo](#)
- [Metrics](#)
- [A/B Testing](#)
- [Using with Rails](#)
- [Managing Identity](#)

Also:

- [Experiment Driven Development](#)
- [Get the code](#)
- [API reference](#)
- [Join vanity-talk list](#)
- [Contributing to Vanity](#)

A/B Testing with Rails (in 5 easy steps)

Step 1: Start using Vanity in your Rails application:

```
Rails::Initializer.run do |config|
  gem.config "vanity"

  config.after_initialize do
    require "vanity"
  end
end
```

And:

```
class ApplicationController < ActionController::Base
  use_vanity :current_user
end
```

Step 2: Define your first A/B test. This experiment goes in the file `experiments/price_options.rb`:

```
ab_test "Price options" do
  description "Mirror, mirror on the wall, who's the better price of all?"
  alternatives 19, 25, 29
  metrics :signup
end
```

Step 3: Present the different options to your users:

```
<h2>Get started for only $<%= ab_test :price_options %> a month!</h2>
```

Step 4: Measure conversion:

```
class SignupController < ApplicationController
  def signup
    @account = Account.new(params[:account])
    if @account.save
      track! :signup
      redirect_to @account
    else
      render action: :offer
    end
  end
end
```

Step 5: Check the report:

```
vanity report --output vanity.html
```

Price options (A/B Test)	
Mirror, mirror on the wall, who's the better price of them all?	
Option A: 19	9.4% (308% better than option C)
Option B: 25	6.1% (165% better than option C)
Option C: 29	2.3%
The best choice is option A: it converted at 9.4% (54% better than option B). This result is not statistically significant, suggest you continue this experiment. Option B converted at 6.1%. Option C converted at 2.3%.	
Started Mon, Nov 16	

Read more about [A/B Testing ...](#)

Metrics

- [Defining a Metric](#)
- [Metrics From Your Database](#)
- [Google Analytics](#)
- [Creating Your Own Metric](#)
- [Digging Deeper](#)

A good starting point for improving — on anything — is measuring. Vanity allows you to measure multiple metrics, best way to tell how well your experiments are doing.

Startup metrics for pirates: AARRR!

1. *Acquisition*
2. *Activation*
3. *Retention*
4. *Referral*
5. *Revenue*

Defining a Metric

Vanity always loads metrics defined in the `experiments/metrics` directory. A metric definition is a Ruby file that looks like this:

```
metric "Signup (Activation)" do
  description "Measures how many people signed up for our awesome service."
end
```

That's a basic metric and you feed it data by calling the `track!` method. For example:

```
class AccountsController < ApplicationController

  def create
    @person = Person.new(params[:person])
    if @person.save
      track! :signup # track successful sign up
      UserSession.create person
      redirect_to root_url
    else
      render :action=>:new
    end
  end
end
```

The metric identifier is the same as the file name. The above example defines the metric `:signup` in the file `experiments/metrics/signup.rb`.

You can call `track!` with a value to track. This example tracks how many items were bought during the day:

```
def checkout
  track! :items, @cart.items.count
  . . .
end
```

Calling `track!` with no value is the same as calling with one, and for convenience you can pass zero and negative numbers, both will be ignored.



Define, track, and you're ready to roll.

Metrics From Your Database

If you already have the data, why not use it?

This example defines a metric for signups, based on the number of `Account` records created each day:

```
metric "Signup (Activation)" do
  description "Measures how many people signed up for our awesome service."
  model Account
end
```

You don't need to call `track!` with this metric, all the data already exists. It's a simple query to count the number of records created, grouped by their timestamp (`created_at`). And since it's querying the database, you'll immediately see historical data for the last 90 days.

Even though the metric itself doesn't store any information, it needs to update experiments whenever new records are created. To do that, it registers itself as an `after_create` callback on the model.

Some metrics measure values, not occurrences. For example, this metric measures user satisfaction by calculating average value from the column `rating`:

```
metric "Satisfaction Survey" do
  description "Measures how many people signed up for our awesome service."
  model Survey, :average=>:rating
end
```

The aggregates you can use this way are: `:average`, `:minimum`, `:maximum` and `:sum`.

You can use a condition when the metric only applies to some records. Here's a metric that only measures unlimited accounts:

```
metric "Signups to Unlimited" do
  description "Signups to our All You Can Eat and Burp Unlimited plan."
  model Account, :conditions=>{ :plan_type=>'unlimited' }
end
```

If you have named scopes, you'll want to use them instead:

```
metric "Signups to Unlimited" do
  description "Signups to our All You Can Eat and Burp Unlimited plan."
  model Account.unlimited
end
```

When you view this metric, it calculates the number of accounts created on any given day that are currently unlimited plans. So, if ten accounts were created over the past week, and today five of them upgraded to unlimited plan, the metric will show five unlimited accounts (current state) but spread over the past week (their `created_at` timestamp).

If your metric uses aggregates or conditions, and the aggregate/conditional attributes change over time, and you need to know when the change took place, consider tracking the event.

This example tracks when accounts were created or upgraded to unlimited plan:

```
metric "Signups (Unlimited)" do
  description "Signups to our All You Can Eat and Burp Unlimited plan (including upgrades)."
  Account.after_save do |account|
    track! if account.plan_type_changed? && account.plan_type == 'unlimited'
  end
end
```

Google Analytics

You can easily include Google Analytics metrics in your Vanity dashboard. You'll need, in addition to Vanity, to use [Garb](#), a Ruby wrapper for the Google Analytics API.

Login to Google Analytics using either username and password, or OAuth authentication token. Here's a sample `config/environment` snippet:

```
Rails::Initializer.run do |config|
  gems.config "vanity"
  gems.config "garb"

  . . .
  config.after_initialize do
    require "garb"
    ga = YAML.load_file(Rails.root + "config/ga.yml")
    Garb::Session.login(ga['email'], ga['password'], account_type: "GOOGLE")
  end
end
```

To define a metric that corresponds to the Google Analytics daily visitors:

```
metric "Acquisition: Visitors" do
  description "Unique visitors on any given page, as tracked by Google Analytics"
  google_analytics "UA-1828623-6", :visitors
end
```

The first argument is the GA profile, the second argument the GA metric name (defaults to `pageviews`).

You can use the full Garb API by accessing the report directly, for example:

```
metric "Activation: Signups" do
  google_analytics "UA-1828623-6"
  report.filters do
    eql(:page_path, 'welcome')
  end
end
```

See [the Garb documentation](#) and [Google Analytics API](#) for more details.

Creating Your Own Metric

Got other ideas for metrics? Writing your own metric is fairly simple.

The easiest way to create your own metric is by adding your own `values` method, for example:

```
metric "Hours in a day" do
  description "Measures how many hours in each day."
  def values(from, to)
    (from..to).map { |i| 24 }
  end
end
```

This example is based on `Vanity::Metric`. You can, of course, base your metric on any other class.

For simplicity, a metric is any object that implements these two methods:

- `name` — Returns the metric's name, which will show up in the dashboard/report.
- `values` — Receives a start date and end date and returns an array of values for all dates in that range (inclusive).

A metric may also implement these methods:

- `description` — Returns human readable description.
- `bounds` — Returns acceptable upper and lower bounds (`nil` if unknown).
- `hook` — [A/B tests](#) use this to manage their own book keeping.

If you wrote your own metric implementation, please consider [contributing it to Vanity](#) so we can all benefit from it. Thanks.

Digging Deeper

All metrics are listed in `Vanity.playground.metrics`, a hash that maps metric identifier to metric object. Methods like `track!` and `metrics` (see [A/B tests](#)) reference metrics using their identifier.

On startup, Vanity loads all the metrics it finds in the `experiments/metrics` directory. The metric identifier is the same as the file name, so `experiments/metrics/coolness.rb` becomes `:coolness`.

You can always populate the hash with your own metrics.

When Vanity loads a metric, it evaluates the metric definition in a context that has two methods: `metric` and `playground`. The `metric` method creates a new `Vanity::Metric` object, and evaluates the block in the context of that object, so when you see the metric definition using methods like `description` or `model`, these are all invoked on the metric object itself.

A `Vanity::Metric` object responds to `track!` and increments a record in the database (an $O(1)$ operation). It creates one record for each day, accumulating that day's count. When generating reports, the `values` method fetches the values of all these keys (also $O(1)$).

You can call `track!` with a value higher than one, and it will increment the day's count by that value.

Any time you track a metric, the metric passes its identifier, timestamp and count (if more than zero) to all its hooks. [A/B tests](#) use hooks to manage their own book keeping. When you define an experiment and tell it which metric(s) to use, the experiment registers itself by calling the `hook` method.

When you call `model` on a metric, this method changes the metric definition by rewriting the `values` method to perform a query, rewriting the `track!` method to update hooks but not the database, and register an `after_create` callback that updates the hooks.

How about some tips & tricks for getting the most out of metrics (you might call them “best practices”)? Got any to share?

A/B Testing

- [True or False](#)
- [Interpreting the Results](#)
- [Multiple Alternatives](#)
- [A/B Testing and Code Testing](#)
- [Let the Experiment Decide](#)

[A/B testing](#) (or “split testing”) are experiments you can run to compare the performance of different alternatives. A classical example is using an A/B test to compare two versions of a landing page, to find out which alternative leads to more registrations.

You can use A/B tests to gauge interest in a new feature, response to a feature change, improve the site’s design and copy, and so forth. In spite of the name, you can use A/B tests to check out more than two alternatives.

“If you are not embarrassed by the first version of your product, you’ve launched too late” — Reid Hoffman, founder of LinkedIn

True or False

Let’s start with a simple experiment. We have this idea that a bigger sign-up link will increase the number of people who sign up for our service. Let’s see how well our hypothesis holds.

We already have a [metric](#) we’re monitoring, and our experiment will measure against it:

```
ab_test "Big signup link" do
  description "Testing to see if a bigger sign-up link increases number of
  signups."
  metrics :signup
end
```

Next, we’re going to show some of our visitors a bigger sign-up link:

```
<% bigger = "font:14pt;font-weight:bold" if ab_test(:big_signup_link) %>
<%= link_to "Sign up", signup_url, style: bigger %>
```

Approximately half the visitors to our site will see this link:

[Sign up](#)

The other half will see this one:

[Sign up](#)

An A/B test has two parts, we just covered the part which decides which alternative to show. The second part measures the effectiveness of each alternative. This happens as result of measuring the metric.

Remember that we’re measuring signups, so we already have this in the code:

```
class SignupController < ApplicationController
  def signup
    Account.create(params[:account])
    track! :signup
  end
end
```

Interpreting the Results

We're going to let the experiment run for a while and track the results using [the dashboard](#), or by running the command `vanity report`.

Vanity splits the audience randomly — using [cookies and other mechanisms](#) — and records who got to see each alternative, and how many in each group converted (in our case, signed up). Dividing conversions by participants gives you the conversion rate.

Option A: <code>false</code>	80.6% (11% better than option B)
Option B: <code>true</code>	72.6%
The best choice is option A: it converted at 80.6% (11% better than option B). With 90% probability this result is statistically significant. Option B converted at 72.6%. Option A selected as the best alternative.	

Vanity will show the conversion rate for each alternative, and how that conversion compares to the worst performing alternative. In the example above, option A has 80.6% conversion rate, 11% more than option B's 72.6% conversion rate ($72.6 * 111\% \sim 80.6\%$).

(These large numbers are easily explained by the fact that this report was generated from made up data)

It takes only a handful of visits before you'll see one alternative clearly performing better than all others. That's a sign that you should continue running the experiment. You see, small sample size tend to give out random results.

To get actionable results, you want a large enough sample, more specifically, you want to look at the probability. Vanity picks the top two alternatives and [calculates a z-score](#) to determine the probability that the best alternative performed better than second best. It presents that probability which should tell you when is a good time to wrap up the experiment.

This is the part that gets most people confused about A/B testing. Let's say we ran an experiment with two alternatives and we notice that option B performs 50% better than option A ($A * 150\% = B$). We calculate from the z-score a 90% probability.

"With 90% probability" does not mean 90% of the difference (50%), it does not mean B performs 45% better than A ($90\% * 50\% = 45\%$). In fact, it doesn't tell us how well B performs relative to A. Option B may perform exceptionally well during the experiment, not so well later on.

The only thing "with 90% probability" tells us is the probability that option B is somewhat better than option A. And that means 10% probability that the results we're seeing are totally random and mean nothing in the long run. In other words: 9 out of 10 times, B is indeed better than A.

If you run the test longer to collect a larger sample size you'll see the probability increase to 95%, then 99% and finally 99.9%. That's big confidence in the outcome of the experiment, but it might take a long time to get there.

You might want to instead decide on some target probability (which could change from one experiment to another). For example, if you pick 95% as the target, you're going to act on the wrong conclusion 1 out of 20 times, but you're going to finish your experiments faster, which means you'll get to iterate quickly and more often. Fast iterations are one way to improve the quality of your software.

You'll want to read more about [A/B testing and statistical significance](#)

Multiple Alternatives

Your A/B tests can have as many alternatives as you care, with two caveats. The more alternatives you have the larger the sample size you need, and so the longer it will take to find out the outcome of your experiment. You want alternatives that are significantly different from each other, testing two pricing options at \$5 and \$25 is fast, testing all the prices between \$5 and \$25 at \$1 increments will take a long time to reach any conclusive result.

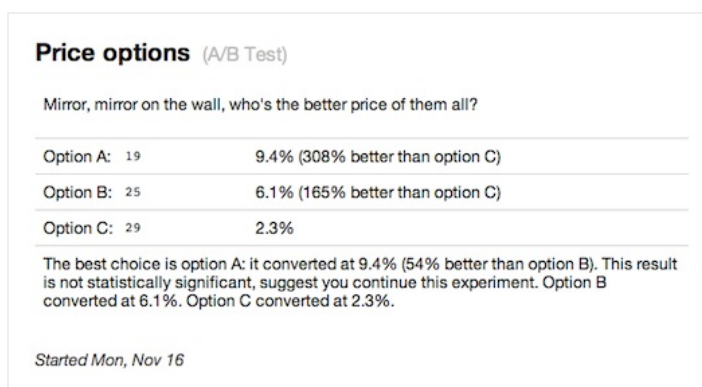
The second caveat is that right now Vanity only scores the two best performing alternatives. This may be an issue in some experiments, it may also be fixed in a future release.

To define an experiment with multiple alternatives:

```
ab_test "Price options" do
  description "Mirror, mirror on the wall, who's the better price of them all?"
  alternatives 5, 15, 25
  metrics :signup
end
```

The `ab_test` method returns the value of one of the chosen alternatives, so in your views you can write:

```
<h2>Get started for only $<%= ab_test :price_options %> a month!</h2>
```



If you don't given any values, Vanity will run your experiment with the values false and true. Here are other examples for rendering A/B tests with multiple values:

```
def index
  # alternatives are names of templates
  render template: ab_test(:new_page)
end

<%= ab_test(:greeting) %> <%= current_user.name %>

<% ab_test :features do |count| %>
  <%= count %> features to choose from!
<% end %>
```

A/B Testing and Code Testing

If you're presenting more than one alternative to visitors of your site, you'll want to test more than one alternative. Don't let A/B testing become A/broken.

You can force a functional/integration test to choose a given alternative:

```
def test_big_signup_link
  experiment(:big_signup_link).chooses(true)
  get :index
  assert_select "a[href=/signup][style^=font:14pt]", "Sign up"
end
```

Here's another example using Webrat:

```
def test_price_option
  [19, 25, 29].each do |price|
    experiment(:price_options).chooses(price)
    visit root_path
    assert_contain "Get started for only $#{price} a month!"
  end
end
```

You'll also want to test each alternative visually, from your Web browser. For that you'll have to install the [the Dashboard](#), which lets you pick which alternative is shown to you:

Option A: false	0.0%	showing
Option B: true	0.0%	<input type="button" value="show"/>

Once the experiment is over, simply remove its definition from the experiments directory and run the test suite again. You'll see errors in all the places that touch the experiment (from failing to load it), pointing you to what parts of the code you need to remove/change.

Let the Experiment Decide

Sample size and probability help you interpret the results, you can also use them to configure an experiment to automatically complete itself.

This experiment will conclude once it has 1000 participants for each alternative, or a leading alternative with probability of 95% or higher:

```
ab_test "Self completed" do
  description "This experiment will self-complete."
  metrics :coolness
  complete_if do
    alternatives.all? { |alt| alt.participants >= 1000 } ||
      (score.choice && score.choice.probability >= 95)
  end
end
```

When it reaches its end, the experiment will stop recording conversions, chose one of its alternatives as the outcome and switch every usage of `ab_test` to that alternative.

By default Vanity will choose the alternative with the highest conversion rate. This is most often, but not always, the best outcome. Imagine an experiment where option B results in less conversions, but higher quality conversion than option A. Perhaps you're interested in option B conversions if you're losing no more than 20% compared to option A. Here's a way to write that outcome:

```
ab_test "Self completed" do
  description "This experiment will self-complete."
  metrics :coolness

  complete_if do
    score(95).choice # only return choice with probability >= 95
  end

  outcome_is do
    a, b = alternatives
    b.conversion_rate >= 0.8 * a.conversion_rate ? b : a
  end
end
```

Using with Rails

- [Configuring Vanity](#)
- [Test Environment](#)
- [Dashboard](#)
- [Unicorn and Forking Servers](#)

This guide is written for Rails 2.3.5. If you have any tips for Rails 3.0, please share.

Configuring Vanity

Start by telling Rails to use the Vanity gem, either using `config.gem "vanity"` or by adding `gem "vanity"` to your Gemfile.

You will most likely need to `require "vanity"` from within `after_initialize` in order to use it everywhere in your app:

```
Rails::Initializer.run do |config|
  . . .
  config.after_initialize do
    require "vanity"
  end
end
```

If you have a `config/vanity.yml` file, Vanity will read the configuration for the current environment. For example:

```
staging:
  adapter: redis
  host: staging.internal
production:
  adapter: mongo
  host: live.internal
  database: vanity
```

If you want to use Google Analytics, you must also tell Rails to include the `garb` gem, and login for a new session. You'll want to do that for production, not for development if you like developing offline:

```
config.after_initialize do
  require "garb"
  Garb::Session.login('..ga email..', '..ga pwd..', account_type: "GOOGLE")
end
```

There's generally no need to collect metric and experiment data outside production environment. Under Rails, Vanity turns collection on only if the environment name is "production". You can control this from `config/environments` by setting `Vanity.playground.collecting` to `true/false`. When collection is off, Vanity doesn't connect to the database server, so there's no need to set a database configuration for these environments.

Dashboard

Start by adding a new resource in `config/routes.rb`:

```
map.vanity "/vanity/:action/:id", :controller=>:vanity
```

Create a new controller for Vanity:

```
class VanityController < ApplicationController
  include Vanity::Rails::Dashboard
end
```

Now open your browser to <http://localhost:3000/vanity>.

The Dashboard renders complete HTML pages with CSS and all necessary JavaScript libraries. Thankfully, HTML is forgiving enough that it will render correctly even with your existing application layout. You can decide to keep your layout, or tell the controller to set `layout false`.

Unicorn and Forking Servers

Unicorn forks the master process to create worker processes efficiently. Since the master processes opens a connection to the database, all workers end up sharing that connection, resulting in ugly contention issues.

The cure is simple, use the `after_fork` hook to reconnect each worker process. Here's the relevant part from my `config/unicorn.rb`:

```
after_fork do |server, worker|
  ActiveRecord::Base.establish_connection
  Vanity.playground.establish_connection
end
```

You'll run into this issue with other forking servers. Vanity can detect when it runs under Passenger and automatically reconnect each forked process.

Testing Emails

- [Configuring ActionMailer](#)
- [Testing email subject lines](#)
- [Testing email content](#)

Configuring ActionMailer

First setup Rails to send email. For example, if you are using GMail you can setup your SMTP settings like this:

```
ActionMailer::Base.smtp_settings = {
  :address => "smtp.gmail.com",
  :port => "587",
  :domain => "gmail.com",
  :authentication => :plain,
  :user_name => "your-email@gmail.com",
  :password => "your-pass"
}
```

Testing email subject lines

In your `RAILS_ROOT/experiments/` folder create an experiment file. For example:

```
ab_test "Invite subject" do
  description "Optimize invite subject line"
  alternatives "Join now!", "You're invited to an exclusive event."
  metrics :open
end
```

In your `RAILS_ROOT/experiments/metrics/` folder create a metric file for the metric you are testing. For example:

```
metric "Open (Activation)" do
  description "Measures how many recipients opened an email."
end
```

Create an ActionMailer class, for example:

```
class UserMailer < ActionMailer::Base
  def invite_email(user)
    use_vanity_mailer user
    mail :to => user.email, :subject => ab_test(:invite_subject)
  end
end
```

We set the identity of the “user” in the `use_vanity_mailer` method. This can take a string or an object that responds to `id`. If it's nil then it will set it as a random number. Setting the appropriate context is important to have each user consistently get the same alternative in our experiment.

Now we need to include a tracking image in the email content. We pass in the vanity identity which we set when we called `use_vanity_mailer(user)` and the metric we are tracking.

```

<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type">
  </head>
  <body>
    <h1>Hey Joseph</h1>
    <p>
      <%= vanity_tracking_image(Vanity.context.vanity_identity, :open, :host =>
"127.0.0.1:3000") %>
    </p>
  </body>
</html>

```

Last, we have to include the `TrackingImage` module into our `VanityController`. This is the same place that you can include the `Dashboard`.

`Vanity::Rails::TrackingImage` will add a `image` method that will render a blank image.

```

class VanityController < ApplicationController
  include Vanity::Rails::Dashboard
  include Vanity::Rails::TrackingImage
end

```

Testing email content

In your `RAILS_ROOT/experiments/` folder create a new experiment file:

```

ab_test "Invite text" do
  description "Optimize invite text"
  alternatives "A friend of yours invited you to use Vanity", "Vanity is the
latest and greatest in a/b testing technology"
  metrics :click
end

```

In your `RAILS_ROOT/experiments/metrics/` folder create a metric file for the metric you are testing:

```

metric "Click (Acquisition)" do
  description "Measures clickthrough on email."
end

```

A/B test your email content:

```

<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type">
  </head>
  <body>
    <h1>Hi!</h1>
    <p>
      <%= link_to ab_test(:invite_text),
vanity_track_url_for(Vanity.context.vanity_identity, :click, :controller =>
"home", :action => "index", :host => "127.0.0.1:3000") %>
    </p>
  </body>
</html>

```

Here we use the text from the “invite_text” experiment and then use the `vanity_track_url_for` helper to add the identity and the metric to track into the url so that Vanity can track the click-throughs.

Remember: By default, Vanity only collects data in production mode.

Managing Identity

For effective A/B tests, you want to:

1. Randomly show different alternatives to different people
2. Consistently show the same alternative to the same person
3. Know which alternative you're showing and tracking
4. When running multiple tests at once, keep them independent

Vanity will assign each visitor a unique identifier and store it in a cookie that persists across sessions. That way, each visitor will get to see the same alternatives on repeating visits. Assuming they use the same browser on all visits.

If you have a better way of tracking visitors, e.g. using sign in, you'll want to use that instead. That way the same person gets treated to the same experiment, even if they switch between machines and browsers.

You can choose either option using the `use_vanity` method. In the first case, just call `use_vanity` from within the `ApplicationController`. In the second case, you'll want to pass `use_vanity` either a block that returns an identity value, or the name of a method that returns an object which provides the identity.

Sounds complicated? These two examples are equivalent:

```
class ApplicationController < ActionController::Base
  use_vanity :current_user
end

class ApplicationController < ActionController::Base
  use_vanity { |c| c.current_user && c.current_user.id }
end
```

If you use either block or method name and they return `nil`, Vanity will fallback on persistent cookie mechanism.

An identity can be anything. For example, if you're running an experiment to test a new feature that will be available in some projects but not others, you'll want to slice the audience by project identifier, not user ID.

You can also give each experiment a different identity using the `identify` callback. Here's an example that tells one experiment to use a project identifier:

```
ab_test "New feature" do
  description "New feature only available to some projects"
  identify { |c| c.current_project.id }
end
```

Configuring the Playground

Vanity will work out of the box on a default configuration. Assuming you're using Redis on localhost, port 6379, there's nothing special to do.

Database connection information is loaded from `config/vanity.yml`, based on the current environment (`RACK_ENV` or `RAILS_ENV`). Example:

```
development:
  adapter: redis
  connection: redis://localhost:6379/0
production:
  adapter: mongodb
  database: analytics
```

If there's no configuration file and the application does not create a connection explicitly, Vanity will default to the Redis instance running on localhost at port 6379.

The available database adapters are:

- **redis** — This adapter is used by default. Available options are connection and password. host, port, database (defaults to 0) options are available, but deprecated.
- **mongodb** — Available options are host, port, database (defaults to "vanity"), username and password.
- **active_record** — Uses existing ActiveRecord configuration, by you can over-ride by supplying different options. To pick different underlying adapter, set `active_record_adapter`.

You want Vanity to collect information (metrics, experiments, etc) in production, but there's no point collecting data in other environments. You can turn data collection on and off by setting `Vanity.playground.collecting`. Under Rails, collection is turned off in all environments except production.

You may want to turn data collection on for integration tests, depending on what you're testing. Also, you may need to turn it on if your development server runs more than one process, e.g. if you're using Passenger for development and want to use the Vanity console to pick a particular alternative in an A/B test.

Available configuration options are:

name	Is all about ...	Default
<code>load_path</code>	Directory containing experiment files	experiments
<code>logger</code>	This should be obvious	default/Rails
<code>collecting</code>	False if you won't want data collected	true

When [running under Rails](#), Vanity defaults to using the Rails logger, locates the `load_path` relative to Rails root, uses the `config/vanity.yml` configuration file (if present) and turns collection on only in production mode.

If you run a different setup, use the `playground` object to configure Vanity. For example:

```
Vanity.playground.load_path = "exp"
Vanity.playground.establish_connection "redis://db.example.com"
```

Contributing

- [How To Contribute](#)
- [Building From Source](#)
- [Documentation](#)
- [Open Issues](#)

By all means.

How To Contribute

Pick on an [open issue](#), [experimental feature](#), suggestion from the [Google Group](#), or whatever you feel like contributing.

To contribute new code/changes:

1. [Fork the project](#)
2. Please use a topic branch to make your changes, it's easier to test them that way
3. Fix, patch, enhance, document, improve, sprinkle pixie dust
4. Tests. Please. Run `rake` and if possible CI (see below)
5. Send a pull request on GitHub

Bonus points for helping improve the documentation, writing some examples, and adding more test coverage.

Building From Source

Vanity is tested against multiple Ruby implementations, and a variety of database engines. To make life easier, we use [RVM](#) and [Bundler](#) to set up the test/development environment.

To test Vanity for the first time under whichever Ruby implementation you're currently using:

```
$ rake test:setup
$ rake
```

To test Vanity with specific database adapter:

```
$ rake DB=redis
$ rake DB=mongodb
$ rake DB=mysql
```

Before making a release, we run the full test suite against multiple Ruby VMs and using multiple database adapters. Doing this on your own is easier than it sounds:

1. [Fork the project](#)
2. Go to [Travis CI](#), setup a new account if you don't already have one
3. [In your profile page](#), tell Travis to build your fork
4. `git push` your changes into your fork and [watch Travis](#) run the tests

To package Vanity as a gem and install on your machine:

```
$ rake install
```

Documentation

Documentation is written in [Textile](#), and converted to HTML using [Jekyll](#).

API reference is [RDoc](#), converted to HTML using [Yardoc](#).

To build and view documentation:

```
$ rake docs  
$ open html/index.html
```

To clean up after yourself:

```
$ rake clobber
```

Open Issues