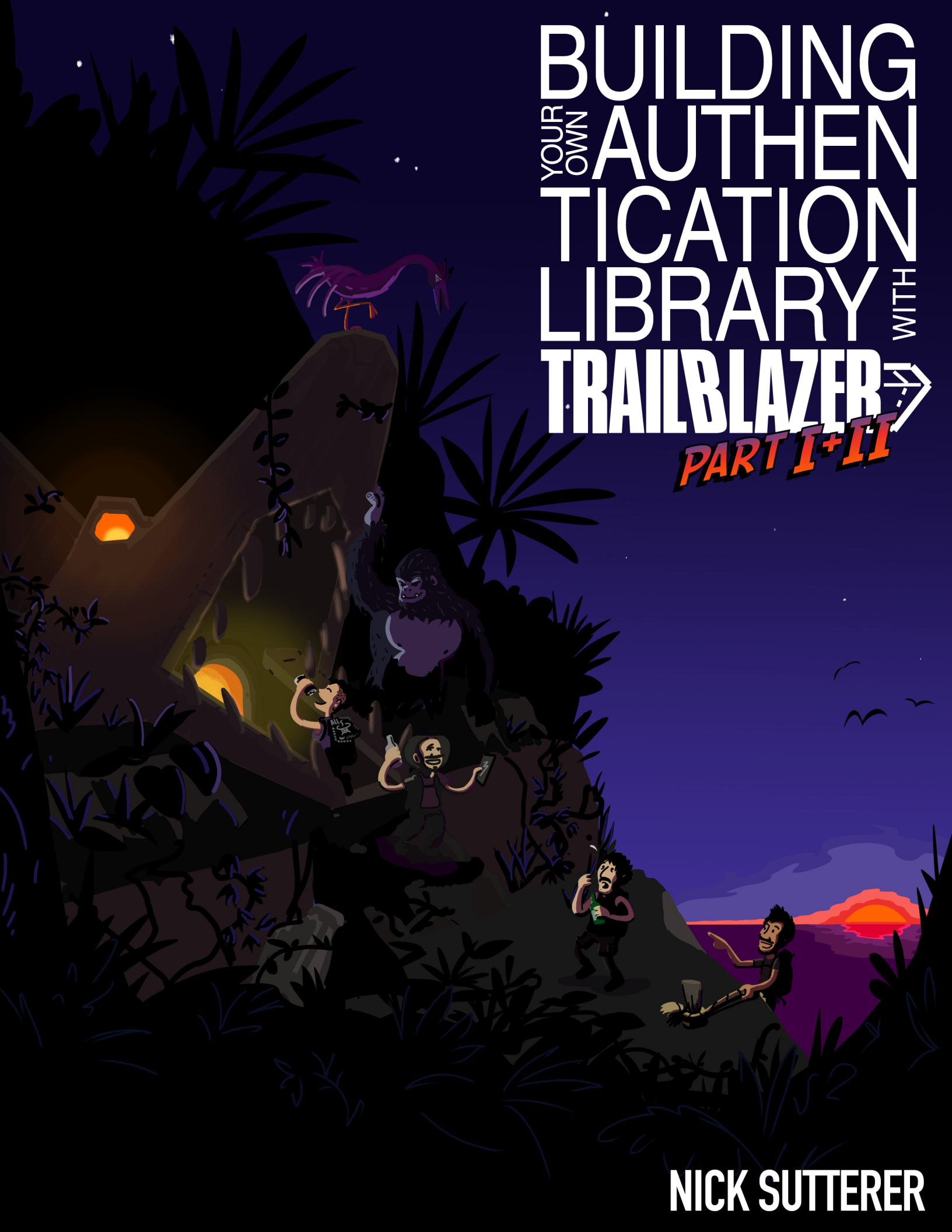


# BUILDING YOUR OWN AUTHEN TICATION LIBRARY WITH TRAILBLAZER

**PART I+II**



**NICK SUTTERER**

# Building your own authentication library with Trailblazer

Nick Sutterer

This book is for sale at <http://leanpub.com/buildalib>

This version was published on 2021-06-23



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2021 Nick Sutterer

# Contents

<b>A Brief Introduction</b>	<b>1</b>
Another layer of unnecessary abstractions	1
Operation - the heart of Trailblazer	1
<b>Operation Basics</b>	<b>2</b>
What is an operation?	2
Debugging with #wtf?	3
Railway Basics	3
Wrap up	9
<b>Create Account</b>	<b>10</b>
Password hashing	10
Persisting data	10
Uniqueness validations	10
Exceptions and flow control	11
Account verification	11
Using dependency injection	12
Sending emails	13
<b>Verify Account</b>	<b>15</b>
Operations and controllers	15
Delegation	15
Avoiding timing-attacks	16
Operation tests	16
<b>Reset Password</b>	<b>18</b>
The operation	18
Testing is fun	19
Nested operations	19
Variable Mapping	20
<b>Changing the password</b>	<b>21</b>
Extracting the token check	21
Checking password reset tokens	21
Update password	22

## CONTENTS

Fast-track wiring . . . . .	23
Signing In . . . . .	24
Wrap up . . . . .	24

# A Brief Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## And this book?

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Another layer of unnecessary abstractions

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Operation - the heart of Trailblazer

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## How are we doing this?

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## There's more than one book!

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Authentication is what we do!

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

# Operation Basics

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **Sing us the song / of the signup form**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **What is an operation?**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **Help, I see god objects!**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **It's about flow control**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **Step-wise, please.**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **Return values matter**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **Step signature**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Running an operation

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Running a test

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Understanding ctx and keyword arguments

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Debugging with `#wtf?`

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Test Assertions

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## The result object

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Is tracing a test tool?

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Railway Basics

While green tests are fun, let's learn more about operations by breaking things, again. One question you might be asking yourself, and we really need to answer is: how does an operation actually *know* if it was successful or if it failed miserably?

```

1  it "fails on invalid email" do
2    result = Auth::Operation::CreateAccount.wtf?({email: "yogi@trb"})
3    assert result.failure? # because invalid email!
4  end

```

```

-- Auth::Operation::CreateAccount
|-- Start.default
|-- check_email
|-- End.failure

```

Check the above test. When passing an invalid email into the operation, `#check_email` fails. This happens obviously because the regular expression matching returns `false` - it doesn't recognize an email.

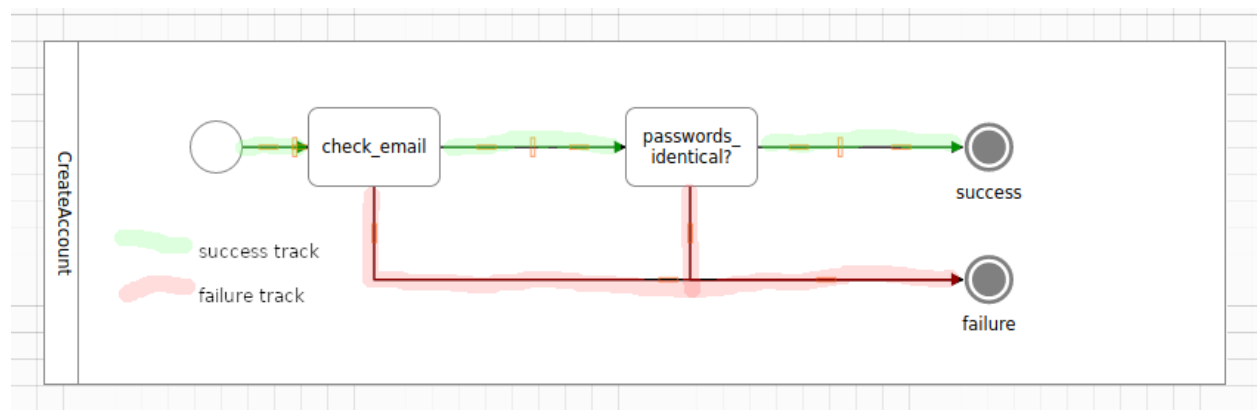
## Falsey return values

Since that very expression is the only statement in `#check_email`, the method itself returns `false`, which is indicating to Trailblazer that this step has “failed”. You can see the actual failing step marked orange in the trace. Or is it brown? We will never know. Just keep in mind, whenever a step returns a falsey value, this will be its trace color.

Now, looking at the above trace again also illustrates how all remaining steps are skipped whenever a step fails. This leads to the operation outcome being a `failure`. Why is this? Why does the operation know what steps to skip, and how does it interpret the outcome?

## What's a Railway?

The simple answer is: the underlying flow model of an operation is a *railway*. If we had the time to draw a diagram it'd look like this.



A railway is a concept [originating from functional programming](https://fsharpforfunandprofit.com/rop/)<sup>1</sup> which has one main goal: reducing error handling code. This works by providing two *tracks* to place your logical components upon.

<sup>1</sup><https://fsharpforfunandprofit.com/rop/>



Those code blocks are executed from left to right. The flow deviates to the lower track should a particular code unit fail.

And this is where we leave the official definition and formal terminology to swing back to Trailblazer and the `CreateAccount` operation.

```
1 class CreateAccount < Trailblazer::Operation
2   step :check_email
3   step :passwords_identical?
```

It's dead simple. Using `step` will place the step on the “success” track (green) which leads to a success terminus. This is why - with proper input - your `CreateAccount` is successful.

However, if a step returns a `nil` or `false` value, the flow deviates to the “failure” track (red), leading to the `failure` terminus. That is the reason the remaining steps on the success track are not executed. In our example, when providing an invalid email address, the `passwords_identical?` step is skipped.

You could implement a railway-like behavior yourself by using a bunch of `ifs` and `elses`. You'd hate it pretty quickly as it's hard to follow and even harder to extend later. What is even better about the railway: you can place steps on the failure track, too.



## Go out and play!

You may play with this code right away in the [A-branch<sup>2</sup>](#) of our example application.

## Error handling

Striving to be a good library, and considering the mental capacities of our dear CSO, we want to provide helpful error messages to the library user in case of an invalid validation. Believe it or not, but this can be done without any `if`. By using the `#fail` method, you can put logic on the failure track.

```
1 class CreateAccount < Trailblazer::Operation
2   step :check_email
3   fail :email_invalid_msg      # {fail} places steps on the failure track.
4   step :passwords_identical?
5   fail :passwords_invalid_msg
6   # ...
7   def email_invalid_msg(ctx, **)
8     ctx[:error] = "Email invalid."
9   end
10
```

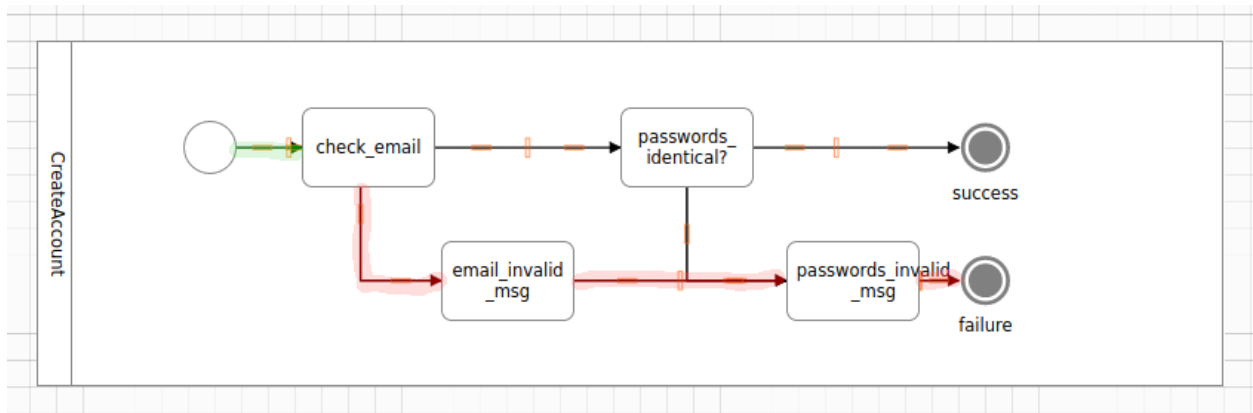
<sup>2</sup>[https://github.com/apotonick/buildalib/blob/A-branch/test/operations/auth\\_operation\\_test.rb#L3](https://github.com/apotonick/buildalib/blob/A-branch/test/operations/auth_operation_test.rb#L3)

```

11  def passwords_invalid_msg(ctx, **)
12      ctx[:error] = "Passwords do not match."
13  end
14  end

```

In the new version, we add two error handling steps. `#email_invalid_msg` and `#passwords_invalid_msg` are placed directly after their respective steps. However, note that those new steps sit on the failure track.



## Writing to ctx

To expose values to the following steps or eventually to the operation user, we can write variables to the `ctx` object. The error messages we're planning to communicate are good examples how this is done.

```

1  def email_invalid_msg(ctx, **)
2      ctx[:error] = "Email invalid."
3  end

```

You should use symbol key names such as `:error` so the variable can be used as a keyword argument in the following steps. In `fail` steps, the return value doesn't matter, if not otherwise configured.

As responsible developers, there's no other way to continue from here than writing a new test case to assert the error messages are correct.

```

1  it "returns error message for invalid email" do
2    result = Auth::Operation::CreateAccount.wtf?(
3      {
4        email:      "yogi@trb", # invalid email.
5        password:    "1234",
6        password_confirm: "1234",
7      }
8    )
9    assert result.failure?
10   assert_equal "Email invalid.", result[:error]
11   #=> Expected: "Email invalid."
12   # Actual: "Passwords do not match."
13 end

```

Passing an incomplete email to `CreateAccount`, we'd expect the error message to complain about just that. However, the `:error` variable seems overridden, the wrong error handler for the passwords must've been called! The error message is "Passwords do not match.", which clearly belongs to the second error handler.

Is it a glitch in the matrix? Is it a bug? Is lunch-time over, yet? Let's check the trace.

```

-- Auth::Operation::CreateAccount
| -- Start.default
| -- check_email
| -- email_invalid_msg
| -- passwords_invalid_msg
| -- End.failure

```

Both error handlers have been called, with the second overriding the first one's error message. While this is not exactly what we expected, it's pretty obvious when you look at the operation flow diagram above: once deviated to the failure track, steps placed with `fail` will be executed one after another.

## Fail fast!

There are many different ways to resolve this drama. You could simply quit your job and stop being a programmer. The merry days of your wild life would be over, you'd have to do some *actual* work. Sucks! Isn't there a way to change how error handlers are connected, maybe?

In part II, we will simply use a contract object in one single step to get around this. For now, let's quickly explore how re-wiring steps can help.

Trailblazer's [Wiring API](https://trailblazer.to/2.1/docs/activity.html#activity-wiring-api)<sup>3</sup> allows for any connections you desire. Steps can be simply placed on the railway, can go back, link to itself, form entire new pathes, connect to new termini, and so on.

<sup>3</sup><https://trailblazer.to/2.1/docs/activity.html#activity-wiring-api>

If we'd need a punchline for marketing the Wiring API, it'd be “what you can draw, you can code”. Ok, well, maybe we should allocate some dollars for the marketing budget and let a bunch of expensive marketing experts with Apple gadgets on their wrists take care of that.

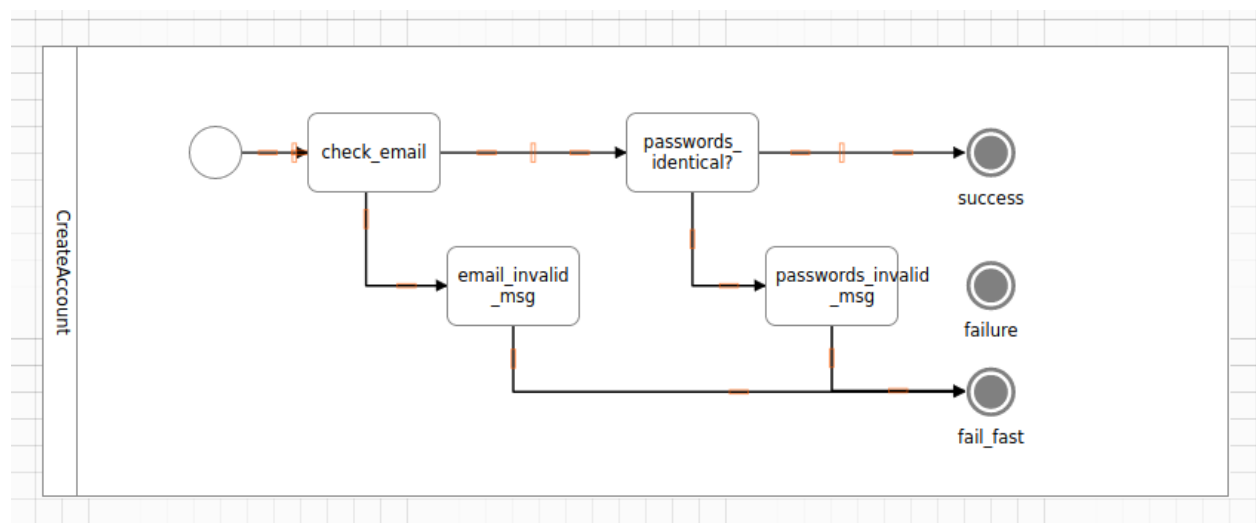
Anyway, for this specific case, we make every `fail` step connect directly to the special `fail_fast` terminus, avoiding the remaining failure track once a `fail` is hit.

```

1 class CreateAccount < Trailblazer::Operation
2   step :check_email
3   fail :email_invalid_msg, fail_fast: true
4   step :passwords_identical?
5   fail :passwords_invalid_msg, fail_fast: true
6   # ...
7 end

```

Adding `fail_fast: true` for both `fail` steps results in a slightly changed flow.



Following the outgoing connections from the error handler, both go directly to a new terminus `fail_fast`. That's what we wanted. Again, there are other ways to achieve this. You don't even have to have separate steps for error messages. The `fail_fast` is just a quick way to solve that very problem.

## Testing `fail_fast`

With the new wiring in place we have to add another test to make sure the error message for passwords mismatch is correct, too.

```
1  it "validates passwords" do
2    result = Auth::Operation::CreateAccount.wtf?(
3      {
4        email:      "yogi@trb.to",
5        password:    "12345678",
6        password_confirm: "1234",
7      }
8    )
9    assert result.failure?
10   assert_equal "Passwords do not match.", result[:error]
11 end
```

Here, we do provide a valid email, but two differing passwords, making our validation code smoke.

Re-running the three tests we already have, all pass with lovely green characters running over the dark and mystical terminal. Is it time to put on the Neo sunglasses just yet? That might also help to not getting recognized by our CSO, who will definitely not be happy with us “having finished a simple, primitive validation, only”.

## Wrap up

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

# Create Account

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Password hashing

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Defaulting keyword arguments

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Setting a state

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Persisting data

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Testing side-effects

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Uniqueness validations

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Exceptions are here to crash

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Exceptions and flow control

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Catching exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Account verification

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Database schema for verify keys

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Computing the random token

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Unique random key

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Testing the happy path

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Using dependency injection

Of course, for a grown-up developer, there's no way around testing the `begin/rescue` part of `#save_verify_account_key`. You wouldn't leave that untested, would ya?

Those tests are crucial for a good system integrity. Often, and in many applications I've seen, developers tend to "forget" to test those edge cases due to the complexity of setting up the tests. Luckily, with Traiblazer and its built-in dependency injection, this is really simple. You need to find better excuses from now on.

### Stubbing the random generator

As we want to override the way a random key is generated, I add a `NotRandom` class to the test file.

```
1 # test/operations/auth_operation_test.rb
2 class NotRandom
3   def self.urlsafe_base64(*)
4     "this is not random"
5   end
6 end
```

The pseudo generator exposes the same interface (the public `#urlsafe_base64` method) as the original generator. We're ready to crash our code.

### Injecting a test dependency

In a new, failing test case, we provoke the uniqueness violation in `#save_verify_account_key` by injecting the pseudo random generator. Check out how simple that is, and how different it is to stubbing.

```
1 it "fails when inserting the same {verify_account_key} twice" do
2   options = {
3     email: "fred@trb.to",
4     password: "1234",
5     password_confirm: "1234",
6     secure_random: NotRandom # inject a test dependency.
7   }
8   # ...
9 end
```

The injected `:secure_random` variable is simply added along with the other input variables. Remember, the options hash passed to an operation call is transformed into the `ctx` object, all variables are available as keyword arguments to the steps. By providing `:secure_random` we override the default value we set in the method definition earlier.

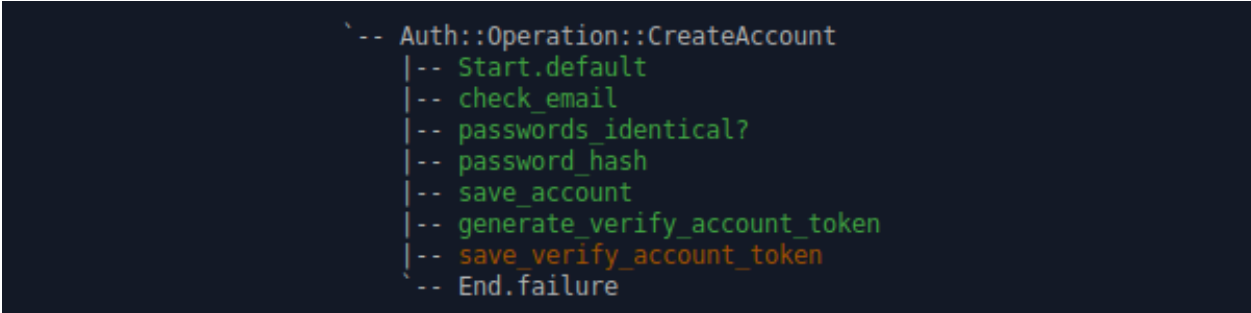
What follows is the remaining test case body with the invocations and the assertions.



```
1 it "fails when inserting the same {verify_account_key} twice" do
2   # ...
3   result = Auth::Operation::CreateAccount.wtf?(options)
4   assert result.success?
5   assert_equal "this is not random", result[:verify_account_key]
6
7   result = Auth::Operation::CreateAccount.wtf?(options.merge(email: "celso@trb.to"))
8   assert result.failure? # verify account key is not unique.
9   assert_equal "Please try again.", result[:error]
10 end
```

The first invocation with the pseudo generator goes through and passes. Check out how the “random” key is the actual string, not an unguessable token.

The second invocation ends on the failure terminus with an error message. That was expected, so that’s a good thing. Since we’re using the same “random” key here, it obviously fails due to the uniqueness violation. However, the database exception is caught and the error is handled gracefully.



```
-- Auth::Operation::CreateAccount
  |-- Start.default
  |-- check_email
  |-- passwords_identical?
  |-- password_hash
  |-- save_account
  |-- generate_verify_account_token
  |-- save_verify_account_token
  -- End.failure
```

## Injectons: cleaner than stubbing!

You are witnessing the beauty of a dependency injection right there! Yes, a random generator object is a dependency, too.

While in many OOP environments you’d temporarily change global state to stub the random generator, Trailblazer has a clean, functional way of handling dependencies. By using NotRandom as our random string generator we can make sure we get two identical tokens.

We have all logic covered by tests, and can move on to the last piece of the CreateAccount operation: sending the verification email that contains the token link we computed over the last pages.

## Sending emails

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **Mind your ctx**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **Using ActionMailer**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **Rails and its routing**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **Testing the entire operation**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **Rails and its mailers**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## **Wrap up**

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

# Verify Account

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Operations and controllers

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Business logic: anything but HTTP

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Controller actions

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Delegation

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Splitting the token

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Utility components

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Finding the verify-account key

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Finding the user

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Avoiding timing-attacks

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Eql comparisons

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Safe comparison algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Verifying the account

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Finalizing the account

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Operation tests

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Testing edge cases

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Test with invalid tokens

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Token, have you expired?

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Wrap Up

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

# Reset Password

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## The operation

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### State of the user

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Computing the reset token

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Another table, another key

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Emailing the “change password” link.

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Extending our existing mailer

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Adding a route

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Testing is fun

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Happy path test

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Factory operations

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Is the email correct?

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Nested operations

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Reusable logic lives in operations

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### Configuration by injection

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

### How do we nest operations?

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Subprocess

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Missing keywords and interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Variable Mapping

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## The DSL provides mapping

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Input

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Injection problem

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Mapping output variables

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Refactoring and the pleasure of deleting code

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Wrap up

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.



# Changing the password

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Technical flow

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Things to do

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

# Extracting the token check

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Implementing a generic `CheckToken`

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Instance method instead of injection

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Utility operation test? Later!

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

# Checking password reset tokens

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Configuration via subclassing

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Testing the token check

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Testing incorrect tokens

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

# Update password

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Tokens keep the hacker out

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Extracting the password processing

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Better encapsulation with nesting

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Updating the user

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Happy people on happy paths

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Testing failing outcome

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Fast-track wiring

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Test-first, isn't that the way to go?

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Fast-track wiring

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Error messages and Fail-fast wiring

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Fast-track and nesting

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Understanding terminus signals

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## The `:fail_fast` option

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## A terminus represents an outcome

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Refactoring with a vengeance

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## The :fast\_track option

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Checking the token in `VerifyAccount`

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

# Signing In

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## An operation for checking credentials

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

## Testing the check

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.

# Wrap up

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/buildalib>.