

---

## Devise-Two-Factor Authentication



Devise-Two-Factor is a minimalist extension to Devise which offers support for two-factor authentication, through the TOTP scheme. It:

- Allows you to incorporate two-factor authentication into your existing models
- Is opinionated about security, so you don't have to be
- Integrates easily with two-factor applications like Google Authenticator and Authy
- Is extensible, and includes two-factor backup codes as an example of how plugins can be structured

### Contributing

We welcome pull requests, bug reports, and other contributions. We're especially looking for help getting this gem fully compatible with Rails 5+ and squashing any deprecation messages.

### Example App

An example Rails 4 application is provided in the [demo](#) directory. It showcases a minimal example of Devise-Two-Factor in action, and can act as a reference for integrating the gem into your own application.

For the demo app to work, create an encryption key and store it as an environment variable. One way to do this is to create a file named `local_env.yml` in the application root. Set the value of `ENCRYPTION_KEY` in the YML file. That value will be loaded into the application environment by `application.rb`.

### Getting Started

Devise-Two-Factor doesn't require much to get started, but there are two prerequisites before you can start using it in your application:

1. A Rails application with devise installed
2. Secrets configured for ActiveRecord encrypted attributes

First, you'll need a Rails application setup with Devise. Visit the Devise homepage for instructions.

---

Devise-Two-Factor uses ActiveRecord encrypted attributes. If you haven't already set up ActiveRecord encryption you must generate a key set and configure your application to use them either with Rails' encrypted credentials or from another source such as environment variables.

```
1 # Generates a random key set and outputs it to stdout
2 ./bin/rails db:encryption:init
```

You can load the key set using Rails' credentials.

```
1 # Copy the generated key set into your encrypted credentials file
2 # Setting the EDITOR environment variable is optional, but without it
  your default editor will open
3 EDITOR="code --wait" ./bin/rails credentials:edit
```

To learn more about credentials run `./bin/rails credentials:help`.

Alternatively, you can configure your application with environment variables rather than Rails' credentials.

```
1 # Copy the generate key set and set them as environment variables
2
3 config.active_record.encrypted_primary_key = ENV['
  ACTIVE_RECORD_ENCRYPTION_PRIMARY_KEY']
4 config.active_record.encrypted_deterministic_key = ENV['
  ACTIVE_RECORD_ENCRYPTION_DETERMINISTIC_KEY']
5 config.active_record.encrypted_key_derivation_salt = ENV['
  ACTIVE_RECORD_ENCRYPTION_KEY_DERIVATION_SALT']
```

Add Devise-Two-Factor to your Gemfile with:

```
1 # Gemfile
2
3 gem 'devise-two-factor'
```

There is a generator which automates most of the setup:

```
1 # MODEL is the name of the model you wish to configure
  devise_two_factor e.g. User or Admin
2 ./bin/rails generate devise_two_factor MODEL
```

Where **MODEL** is the name of the model you wish to add two-factor functionality to (for example `user`)

This generator will:

1. Create a new migration which adds a few columns to the specified model:

```
1 add_column :users, :otp_secret, :string
2 add_column :users, :consumed_timestep, :integer
3 add_column :users, :otp_required_for_login, :boolean
```

---

2. Edit `app/models/MODEL.rb` (where MODEL is your model name):

- add the `:two_factor_authenticatable` devise module
- remove the `:database_authenticatable` if present because it is incompatible with `:two_factor_authenticatable`

3. Add a Warden config block to your Devise initializer, which enables the strategies required for two-factor authentication.

Remember to apply the new migration after you run the generator:

```
1 ./bin/rails db:migrate
```

Next you need to whitelist `:otp_attempt` as a permitted parameter in Devise `:sign_in` controller. You can do this by adding the following to your `application_controller.rb`:

```
1 # app/controllers/application_controller.rb
2
3 before_action :configure_permitted_parameters, if: :devise_controller?
4
5 # ...
6
7 protected
8
9 def configure_permitted_parameters
10   devise_parameter_sanitizer.permit(:sign_in, keys: [:otp_attempt])
11 end
```

Finally you should verify that `:database_authenticatable` is **not** being loaded by your model. The generator will try to remove it, but if you have a non-standard Devise setup, this step may fail.

**Loading both `:database_authenticatable` and `:two_factor_authenticatable` in a model is a security issue** It will allow users to bypass two-factor authenticatable due to the way Warden handles cascading strategies!

## Designing Your Workflow

Devise-Two-Factor only worries about the backend, leaving the details of the integration up to you. This means that you're responsible for building the UI that drives the gem. While there is an example Rails application included in the gem, it is important to remember that this gem is intentionally very open-ended, and you should build a user experience which fits your individual application.

There are two key workflows you'll have to think about:

1. Logging in with two-factor authentication

---

## 2. Enabling two-factor authentication for a given user

We chose to keep things as simple as possible, and our implementation can be found by registering at Tinfoil Security, and enabling two-factor authentication from the security settings page.

### Logging In

Logging in with two-factor authentication works extremely similarly to regular database authentication in Devise. The `TwoFactorAuthenticatable` strategy accepts three parameters:

1. email
2. password
3. otp\_attempt (Their one-time password for this session)

These parameters can be submitted to the standard Devise login route, and the strategy will handle the authentication of the user for you.

### Disabling Automatic Login After Password Resets

If you use the Devise `recoverable` strategy, the default behavior after a password reset is to automatically authenticate the user and log them in. This is obviously a problem if a user has two-factor authentication enabled, as resetting the password would get around the two-factor requirement.

Because of this, you need to set `sign_in_after_reset_password` to `false` (either globally in your Devise initializer or via `devise_for`).

### Enabling Two-Factor Authentication

Enabling two-factor authentication for a user is easy. For example, if my user model were named `User`, I could do the following:

```
1 current_user.otp_required_for_login = true
2 current_user.otp_secret = User.generate_otp_secret
3 current_user.save!
```

Before you can do this however, you need to decide how you're going to transmit two-factor tokens to a user. Common strategies include sending an SMS, or using a mobile application such as Google Authenticator.

At Tinfoil Security, we opted to use the excellent `qrqr-code-rails3` gem to generate a QR-code representing the user's secret key, which can then be scanned by any mobile two-factor authentication client.

---

If you decide to do this you'll need to generate a URI to act as the source for the QR code. This can be done using the `User#otp_provisioning_uri` method.

```
1 issuer = 'Your App'
2 label = "#{issuer}:#{current_user.email}"
3
4 current_user.otp_provisioning_uri(label, issuer: issuer)
5
6 # > "otpauth://totp/Your%20App:user@example.com?secret=[otp_secret]&
    issuer=Your+App"
```

If you instead decide to send the one-time password to the user directly, such as via SMS, you'll need a mechanism for generating the one-time password on the server:

```
1 current_user.current_otp
```

The generated code will be valid for the duration specified by `otp_allowed_drift`. This value can be modified by adding a config in `config/initializers/devise.rb`.

```
1 Devise.otp_allowed_drift = 240 # value in seconds
2 Devise.setup do |config|
3   ...
4 end
```

However you decide to handle enrollment, there are a few important considerations to be made:

- Whether you'll force the use of two-factor authentication, and if so, how you'll migrate existing users to system, and what your on-boarding experience will look like
- If you authenticate using SMS, you'll want to verify the user's ownership of the phone, in much the same way you're probably verifying their email address
- How you'll handle device revocation in the event that a user loses access to their device, or that device is rendered temporarily unavailable (This gem includes `TwoFactorBackupable` as an example extension meant to solve this problem)

It sounds like a lot of work, but most of these problems have been very elegantly solved by other people. We recommend taking a look at the excellent workflows used by Heroku and Google for inspiration.

### Filtering sensitive parameters from the logs

To prevent two-factor authentication codes from leaking if your application logs get breached, you'll want to filter sensitive parameters from the Rails logs. Add the following to `config/initializers/filter_parameter_logging.rb`:

---

```
1 Rails.application.config.filter_parameters += [:otp_attempt]
```

## Preventing Brute-Force Attacks

With any authentication solution it is also important to protect your users from brute-force attacks. For Devise-Two-Factor specifically if a user's username and password have already been compromised an attacker would be able to try possible TOTP codes and see if they can hit a lucky collision to log in. While Devise-Two-Factor is open-ended by design and cannot solve this for all applications natively there are some possible mitigations to consider. A non-exhaustive list follows:

1. Use the `lockable` strategy from Devise to lock a user after a certain number of failed login attempts. See <https://www.rubydoc.info/github/heartcombo/devise/main/Devise/Models/Lockable> for more information.
2. Configure a rate limit for your application, especially on the endpoints used to log in. One such library to accomplish this is rack-attack.
3. When displaying authentication errors hide whether validating a username/password combination failed or a two-factor code failed behind a more generic error message.

**Acknowledgements** Thank you to Christian Reitter (Radically Open Security) and Chris MacNaughton (Centauri Solutions) for reporting the issue.

## Backup Codes

Devise-Two-Factor is designed with extensibility in mind. One such extension, `TwoFactorBackupable`, is included and serves as a good example of how to extend this gem. This plugin allows you to add the ability to generate single-use backup codes for a user, which they may use to bypass two-factor authentication, in the event that they lose access to their device.

To install it, you need to add the `:two_factor_backupable` directive to your model.

```
1 devise :two_factor_backupable
```

You'll also be required to enable the `:two_factor_backupable` strategy, by adding the following line to your Warden config in your Devise initializer, substituting `:user` for the name of your Devise scope.

```
1 manager.default_strategies(:scope => :user).unshift :  
  two_factor_backupable
```

---

The final installation step is dependent on your version of Rails. If you're not running Rails 4, skip to the next section. Otherwise, create the following migration:

```
1 class AddDeviseTwoFactorBackupableToUsers < ActiveRecord::Migration
2   def change
3     # Change type from :string to :text if using MySQL database
4     add_column :users, :otp_backup_codes, :string, array: true
5   end
6 end
```

You can then generate backup codes for a user:

```
1 codes = current_user.generate_otp_backup_codes!
2 current_user.save!
3 # Display codes to the user somehow!
```

The backup codes are stored in the database as bcrypt hashes, so be sure to display them to the user at this point. If all went well, the user should be able to login using each of the generated codes in place of their two-factor token. Each code is single-use, and generating a new set of backup codes for that user will invalidate all of the old ones.

You can customize the length of each code, and the number of codes generated by passing the options into `:two_factor_backupable` in the Devise directive:

```
1 devise :two_factor_backupable, otp_backup_code_length: 32,
2                               otp_number_of_backup_codes: 10
```

## Testing

Devise-Two-Factor includes shared-examples for both `TwoFactorAuthenticatable` and `TwoFactorBackupable`. Adding the following two lines to the specs for your two-factor enabled models will allow you to test your models for two-factor functionality:

```
1 require 'devise_two_factor/spec_helpers'
2
3 it_behaves_like "two_factor_authenticatable"
4 it_behaves_like "two_factor_backupable"
```

## Troubleshooting

If you are using Rails 4.x and Ruby  $\geq 2.7$ , you may get an error like

```
1 An error occurred while loading ./spec/devise/models/
  two_factor_authenticatable_spec.rb.
2 Failure/Error: require 'devise'
```

---

```
3
4 NoMethodError:
5   undefined method `new' for BigDecimal:Class
```

see <https://github.com/ruby/bigdecimal#which-version-should-you-select> and <https://github.com/ruby/bigdecimal/> for more details, but you should be able to solve this by explicitly requiring an older version of bigdecimal in your gemfile like

```
1 gem "bigdecimal", "~> 1.4"
```