Extracted from:

Rails, Angular, Postgres, and Bootstrap

Powerful, Effective, and Efficient Full-Stack Web Development

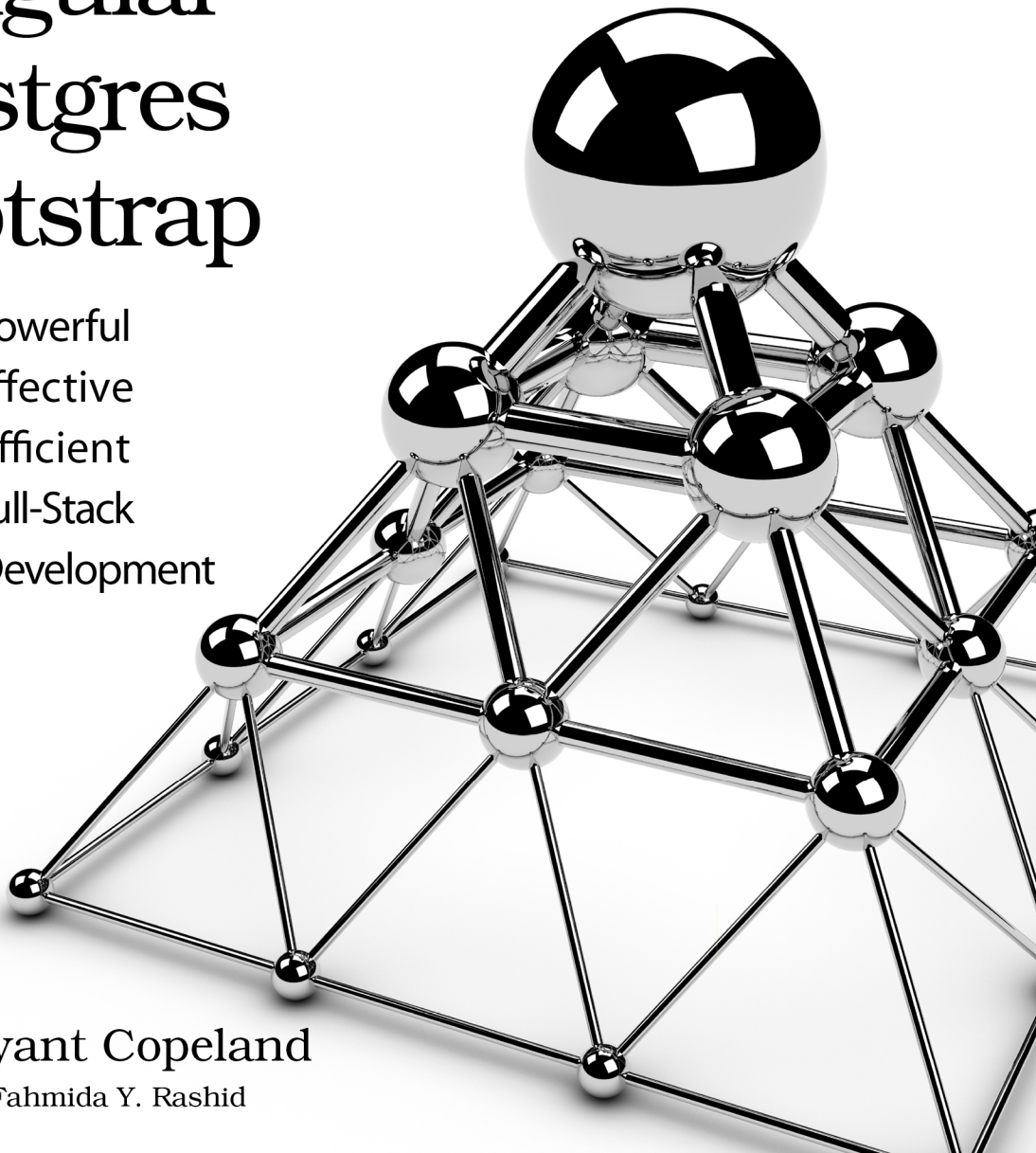
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Rails Angular Postgres Bootstrap

Powerful
Effective
Efficient
Full-Stack
Web Development



David Bryant Copeland
edited by Fahmida Y. Rashid

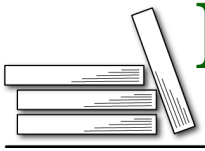
Rails, Angular, Postgres, and Bootstrap

Powerful, Effective, and Efficient Full-Stack Web Development

David Bryant Copeland

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-126-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—July 29, 2015

Prevent Bad Data Using Check Constraints

If you done any database work at all, you're no-doubt familiar with a "not null" constraint that prevents inserting null into a column of the database.

```
CREATE TABLE people (
  id        INT          NOT NULL,
  name      VARCHAR(255) NOT NULL,
  birthdate DATE             NULL
);
```

In this table,id and name may not be NULL, however birthdate may be. Postgres takes the "null constraint" concept *much* further by allowing arbitrary constraints on fields. Postgres also has support for regular expressions. This means we can create a constraint on our email field that requires its value to match the same regular expression we used in our Rails code. This would prevent non-company email addresses from being inserted into the table entirely.

First, we'll create a new migration where we can add this constraint.

```
> bundle exec rails g migration add-email-constraint-to-users
    invoke  active_record
    create    db/migrate/20150303133619_add_email_constraint_to_users.rb
```

The DSL for writing Rails migrations doesn't provide any means of creating this constraint, so we have to do it in straight SQL. Although Postgres *Data Definition Language* (DDL) looks different from what we normally use in our migrations, it's still relatively straightforward and well documented online[1].

The basic structure of our constraint is that we want to "alter" the USERS to "add" a constraint that will "check" the email column for invalid values. Here's what our migration will look like (See *Why aren't we using change in our Rails migrations?*, on page 4 for why we are using the older up and down methods).

login/add-postgres-constraint/shine/db/migrate/20150303133619_add_email_constraint_to_users.rb

```
class AddEmailConstraintToUsers < ActiveRecord::Migration
  def up
    execute %{
      ALTER TABLE
        users
      ADD CONSTRAINT
        email_must_be_company_email
      CHECK ( email ~* '[A-Za-z0-9._%-]+@example.com' )
    }
  end
```

1. http://www.postgresql.org/docs/9.4/static/ddl-constraints.html

```
  def down
    execute %{
      ALTER TABLE
        users
      DROP CONSTRAINT
        email_must_be_company_email
    }
  end
end
```

The ~* operator is how Postgres does regular expression matching. Therefore this code means that the email column's value must match the regular expression we've given or the insert or update command will fail. The regular expression is more or less identical to the one we used when configuring Devise.

Why aren't we using change in our Rails migrations?

Rails 3.1 introduced the concept of *reversible migrations* via the method change in migrations DSL. The Rails authors realized that most implementations of down were to reverse what was done inside up and Rails could figure out how to reverse the code in the up method automatically.

In order to make this work, programmers would need to constrain the contents of the change method to only those migration methods that Rails knows how to reverse, which are itemized in ActiveRecord::Migration::CommandRecorder[a].

In most of the migrations we'll write in this book, we aren't using those methods, and are typically just using execute, because we need to run Postgres-specific commands. We could work within the Reversible Migrations framework by using reversible, but the resulting code is somewhat clunky:

```
class AddEmailConstraintToUsers < ActiveRecord::Migration
  def change
    reversible do |direction|
      direction.up {
        execute %{
          ...
        }
      }
      direction.down {
        execute %{
          ...
        }
      }
    end
  end
end
```

Since up and down aren't deprecated, it ends up being easier to stick with the older syntax for the types of migrations we'll be writing.

a. http://api.rubyonrails.org/classes/ActiveRecord/Migration/CommandRecorder.html

Let's see it in action. First we'll run our migrations (if you experience a problem doing this see *Migrations Failing Because of Existing Data?*, on page 5).

```
> bundle exec rake db:migrate
== 20150303133619 AddEmailConstraintToUsers: migrating ==================
-- execute("
      ALTER TABLE
        users
      ADD CONSTRAINT
        email_must_be_company_email
      CHECK ( email ~* '[A-Za-z0-9._%-]+@example.com' )
      ;
    ")
   -> 0.0012s
== 20150303133619 AddEmailConstraintToUsers: migrated (0.0013s) ========
```

Migrations Failing Because of Existing Data?

If you ran the migrations and saw something like the error below, you'll need to do a bit more work to apply this change.

```
> bundle exec rails db:migrate
ActiveRecord::StatementInvalid: PG::CheckViolation: ERROR:
  check constraint "email_must_be_company_email" is violated by some row:
      ALTER TABLE
        users
      ADD CONSTRAINT
        email_must_be_company_email
      CHECK ( email ~* '[A-Za-z0-9._%-]+@example.com' )
      ;
```

This means that at least one row in your development database has a value for the email column that violates our new constraint. Postgres is refusing to apply the constraint because it doesn't know what to do.

In your development environment, you can easily change or remove those rows that violate the constraint. If you were doing this to an active, production dataset, you would not have that luxury. You would need to get more creative. There are several ways of handling this.

- Create a migration that deletes all users using a bad email address. This is drastic, but would work.
- Create a migration to assign bogus company email addresses to the existing bad accounts. This would prevent those users logging in but maintain their history. You could correct the accounts manually later on, but the constraint would be satisfied.

- You could also do something more complex where you demarcate active users with a new field, and prevent inactive users from logging in. Your check constraint could then only check for active users, e.g. `active = true AND email ~* '[A-Za-z0-9._%-]@example.com'`.

In any case, if you are adding constraints to a running, production system, you'll have to be more careful.

With the migration applied, let's see how it works. First, we'll insert a user whose email is on our company's domain.

```
> bundle exec rails dbconsole
shine_development> INSERT INTO
                     users (
                       email,
                       encrypted_password)
                     VALUES (
                       'foo@example.com',
                       '$abcd'
                     );
INSERT 0 1
```

This works as expected. Now let's try to insert a user using a *different* domain.

```
shine_development> INSERT INTO
                     users (
                       email,
                       encrypted_password)
                     VALUES (
                       'foo@bar.com',
                       '$abcd'
                       );
ERROR:  new row for relation "users" violates
        check constraint "email_must_be_company_email"
DETAIL: Failing row contains (4,
                              foo@bar.com,
                              $abcd,
                              null,
                              null,
                              null,
                              0,
                              null,
                              null,
                              null,
                              null,
                              null,
                              null).
```

We can see that Postgres will refuse to allow invalid data into the table (and that we get a pretty useful error message as well). This means that a rogue application, bug in our code, or even a developer at a production console will not be able to allow access to any user who doesn't have a company email address.

Given how little effort this was, and the piece of mind it gives us, it's a no-brainer to add this level of security. Postgres makes it simple, meaning the cost of securing our website is low.

There is one last thing we'll need to change, because we're using a feature that's Postgres-specific. By default, Rails stores a snapshot of the database schema in db/schema.rb, which is a Ruby source file using the DSL for Rails migrations. Rails creates this by examining the database schema and creating what is essentially a single migration, in Ruby, to create the schema from scratch. This is what tests uses to create a fresh database.

The problem is that Rails doesn't know about check constraints, so the one we just added won't be present in db/schema.rb. This is easily remedied by telling Rails to use SQL, rather than Ruby, for storing the schema. We can do this by adding one line to config/application.rb

login/add-postgres-constraint/shine/config/application.rb
```
config.active_record.schema_format = :sql
```

We'll then need to remove the old db/schema.rb file, create db/structure.sql by running migrations and finally reset our test database by dropping it and recreating it. We can do all this with rake.

```
> rm db/schema.rb
> bundle exec rake db:migrate
> RAILS_ENV=test bundle exec rake db:drop
> RAILS_ENV=test bundle exec rake db:create
```