Hablemos de TDD

Sam Belmor

#Devday4w

¿Qué es TDD?

El TDD o **Test Driven Development** (desarrollo dirigido por pruebas) es una práctica de programación en la que se comienzan escribiendo las pruebas en primer lugar, escribiendo el código fuente a continuación, pasando correctamente la prueba y terminando con la refactorización del código escrito.

Beneficios

- Permite delimitar el alcance de un ticket.
- Ayuda a entender los requerimientos del cliente.
- Reduce el número de bugs en el código.
- Permite tener seguridad a la hora de refactorizar una parte del código.
- Nos aporta una gran cantidad de documentación que nos ayuda a resolver dudas.
- Ayuda a identificar ciertos escenarios que no se habían previsto en la estimación.
- Obtenemos un código limpio y depurado.

El ciclo TDD y sus etapas

01. Desarrollo de la prueba

En esta parte del proceso, se tiene que conocer de forma clara los requerimientos, condicionales y casos bordes.

02. Escritura del código

En este paso se lleva a cabo la escritura del código mediante el cual se irán validando las diferentes pruebas.

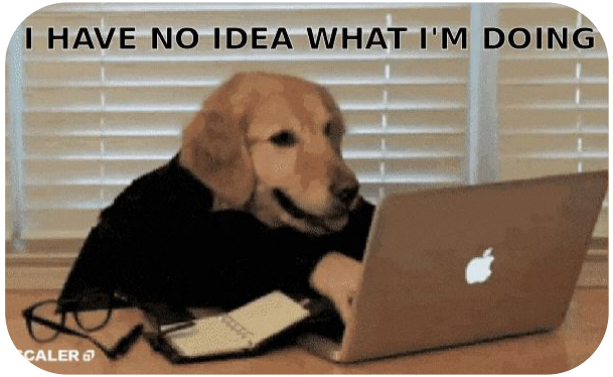
03. Validación de pruebas

Si en la fase anterior hemos conseguido pasar todas las pruebas establecidas, en este paso ya podemos decir que el software es válido.

04. Refactorización

Aquí se reconducen aquellas necesidades que han ido surgiendo a lo largo del proceso.

Mis inicios en el mundo del TDD



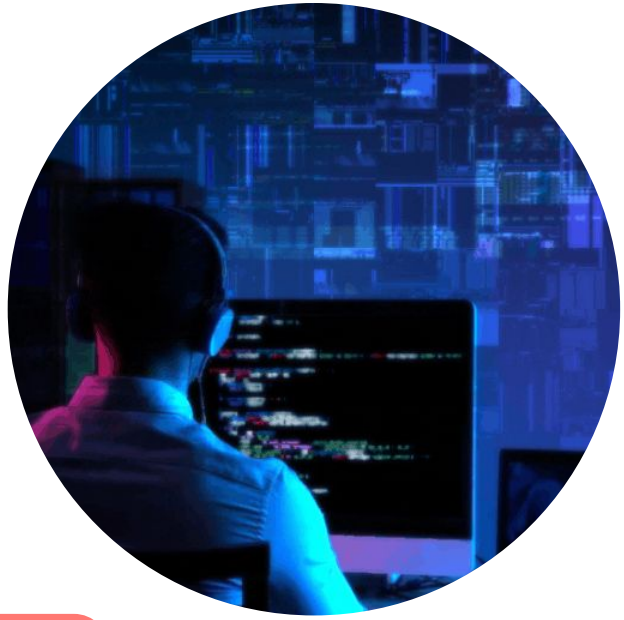
No las disfrutaba en absoluto

No las entendía del todo

Creía que era más rápida si no las escribía

No sabía qué escenarios probar

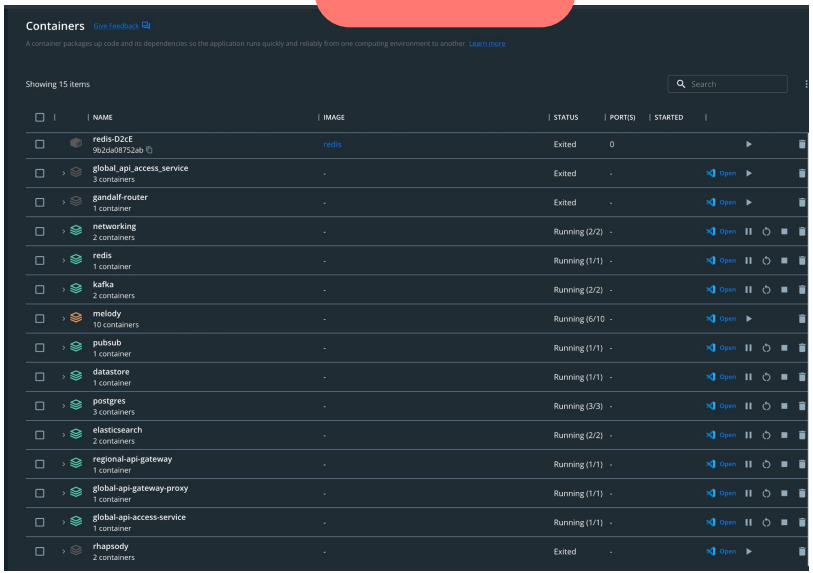
Ahora



04

Servers

Docker



Cómo las empecé a querer...

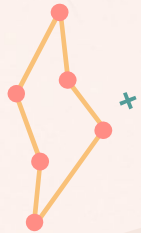
“Escribe una prueba que falle y luego hazla pasar”

- La UI en local estaba roto.
- Ya había reproducido el error en qa.
- Ya llevaba bloqueada más de un día por el error en mi local.
- El error venía de una API.

Cómo trabajamos en el equipo

- Cada vez que hacemos pair programming empezamos por las pruebas.
- Si hay un bug en la app primero tratamos de replicar el error en una prueba y después escribimos el código para hacer pasar esa prueba.
- Empezar por las pruebas me ayuda a entender mejor mi ticket e identificar escenarios que no habíamos previsto.
- En el code review es más fácil dar/recibir feedback respecto a ciertos escenarios que no se han probado y por ende nos ayuda a identificar posibles errores antes de llegar a QA.
- Nos ahorra bastante tiempo.
- Tenemos muchos ejemplos en la aplicación.

Ejemplos



Escribimos el primer escenario

```ruby
require "spec_helper"

RSpec.describe Api::V2::UsersController do
  subject { get "/api/v2/Users" }

  context "when user is an admin" do
    it "returns a 200" do
      subject
      expect(response.status).to eq 200
    end
  end
end
```

```
  returns a 200 (FAILED - 1)

Failures:

  1) UsersController when user is an admin returns a 200
     Failure/Error: subject { get "/api/v2/Users" }

     ActionController::RoutingError:
       No route matches [GET] "/api/v2/Users"
```

Creamos la ruta

```
namespace :v2 do
  get "/Users", to: "users#index"
end
```

```
class Api::V2::UsersController < ActionController::Base

end
```

```
  returns a 200 (FAILED - 1)

Failures:

  1) UsersController when user is an admin returns a 200
     Failure/Error: return @app.call(env) unless in_maintenance_mode?

     ActionController::RoutingError:
       uninitialized constant Api::V2::UsersController
```

```
  returns a 200 (FAILED - 1)

Failures:

  1) Api::V2::UsersController when user is an admin returns a 200
     Failure/Error: return @app.call(env) unless in_maintenance_mode?

     AbstractController::ActionNotFound:
       The action 'index' could not be found for Api::V2::UsersController
```

Hacemos pasar la prueba

```ruby
class Api::V2::UsersController < ActionController::Base
  def index
    render json: { users: User.all }
  end
end
```

```
/Users/sambetmor/.asdf/installs/ruby/2.7.6/lib/ruby/2.7.0/
    returns a 200

Finished in 1.42 seconds (files took 5.42 seconds to load)
1 example, 0 failures

Randomized with seed 14171
```

API para mandar emails

- Endpoint para mandar emails.
- Los emails se mandan en un array = [“admin@gmail.com”, “user@gmail.com”, “manager@gmail.com”]
- Validar que los emails sean correctos.
- Validar que los emails sean de tipo array.
- Lanzar un error si uno de los emails no es correcto.
- Lanzar un error si no es de tipo array.
- Tener ciertos permisos para poder actualizar los emails.
- Lanzar un error 403 si el usuario no tiene los permisos necesarios.

01 Validamos la autenticación

```
context "when the request is not authenticated" do
--  3 lines: let(:jwt) { "invalid-jwt" }.............
end

context "when the request is authenticated" do
  let(:jwt) { valid_gandalf_jwt(user, tid) }
```

02 Validamos permisos

```
context "when the user has manage_team and manage_act_on_behalf permissions" do
  let(:permissions) { ["manage_team", "manage_act_on_behalf"] }

  context "when the approval_strategy is invalid" do
 7 lines: let(:approval_strategy) { "not valid" }·····
  end

  context "when the approval_strategy is valid" do
 4 lines: it "updates the approval_strategy for their team" do·····
  end
```

03

Cuando el email es inválido

```
context "when the emails are invalid" do
  context "when the approval_strategy is valid" do
7   lines: let(:emails) { ["not-valid.salesloft.com"] }
  end

  context "when the emails are not in an array" do
6   lines: let(:emails) { "sam@salesloft.com" }
  end

  context "when the emails are nil" do
6   lines: let(:emails) { nil }
  end

  context "when there is a nil in the emails array " do
11  lines: let(:emails) { [nil] }
  end

  context "when there is a false in the emails array" do
6   lines: let(:emails) { [false] }
  end

  context "when there is a "" in the emails array" do
6   lines: let(:emails) { [""] }
  end

  context "when there is a [] in the emails array" do
6   lines: let(:emails) { [[]] }
  end

  context "returns an error" do
6   lines: let(:emails) { [{}] }
  end

  context "when there is at least one invalid email" do
6   lines: let(:emails) { ["sambelmor@salesloft.com", "sambelmor-salesloft.com"] }
  end
end
```

04 Cuando el email es válido

```
context "when the emails are valid" do
  let(:emails) { ["cloud9erz@salesloft.com", "cloud9erz+1@salesloft.com"] }

  it "updates the emails for this team" do
7 lines: subject·····
  end

  context "when sending an empty array" do
15 lines: let(:emails) { [] }·····
  end

  context "when the emails are duplicate" do
    context "when the emails are in lowercase" do
- 12 lines: let(:emails) { ["sambelmor@salesloft.com", "sambelmor@salesloft.com"] }·····
    end

    context "when the emails have uppercase" do
- 12 lines: let(:emails) { ["Cloud9Erz@salesloft.com", "CLOUD9erz@salesloft.com"] }·····
    end
  end
end

context "when the approval_strategy and emails are valid" do
12 lines: let(:emails) { ["cloud9erz@salesloft.com", "cloud9erz+1@salesloft.com"] }·····
end
```

Conclusión

- El hacer pruebas te ayuda a conocer el alcance del ticket.
- Te da la seguridad de no romper nada cuando refactorizas algo en el código.
- Evitas estar probando la implementación manualmente.
- Si el ticket en el que estás trabajando no es muy claro, el escribir las pruebas te dará la claridad que necesitas.
- Reduce el riesgo de agregar un bug a la aplicación.
- Te aseguras de entregar un producto de calidad al cliente.

¡Gracias!

@SamBelmor