Rails: Testing

Progetto di Sistemi WEB

Anno Accademico 2025/2026

Filippo Poltronieri

filippo.poltronieri@unife.it

Perché il testing?

Il **testing automatizzato** è una pratica fondamentale nello sviluppo software. Permette di:

1. **Migliorare il design** del codice
   → Scrivere test ti costringe a pensare in modo modulare e a separare le responsabilità.
2. **Avere feedback immediato**
   → Un test fallito ti avvisa **all'istante** se hai rotto qualcosa.
3. **Proteggere il codice da regressioni**
   → Dopo una modifica, i test garantiscono che il comportamento precedente sia preservato.
4. **Documentare il comportamento atteso**
   → I test sono **esempi eseguibili** di come usare il codice.

Testing in Rails 8 – Panoramica

Rails 8 offre un **ecosistema di testing completo, moderno e integrato**.

Tipo di Test	Cosa verifica	Strumento
Model	Logica di business, validazioni	MiniTest + ActiveSupport
Controller	Risposte HTTP, redirect	ActionDispatch
System	Interazione utente completa	**Capybara + Cuprite**
Integration	Flussi tra più controller	ActionDispatch

> **Nota importante**:
> In Rails 8, **i system test sono il modo principale per testare l'esperienza utente**.
> I controller test sono utili principalmente per testare API.

Ambiente di Test – Come funziona

Rails crea un **ambiente isolato** chiamato `test`:

```
# config/environments/test.rb
config.cache_classes = true
config.action_mailer.delivery_method = :test
```

- **Database separato**: `db/test.sqlite3` (o configurato in `database.yml`)
- **Nessun impatto** sul database di sviluppo o produzione
- **Transazioni automatiche**: ogni test è avvolto in una transazione → rollback automatico

> **Vantaggio**: i test sono **veloci** e **ripetibili**

Struttura della directory `test`

```
test/
├── models/          # Test dei modelli (logica di business)
├── controllers/     # Test delle risposte HTTP
├── system/          # Test completi con browser (Capybara)
├── integration/     # Flussi tra più controller
├── mailers/         # Test invio email
├── helpers/         # Test helper methods
└── fixtures/        # Dati di esempio (legacy)
```

MiniTest – Framework

Rails usa **MiniTest**, un framework leggero ma potente incluso in Ruby.

Come si scrive un test

```ruby
# test/models/article_test.rb
require "test_helper"

class ArticleTest < ActiveSupport::TestCase
  test "non salva un articolo senza titolo" do
    article = Article.new
    assert_not article.save
    assert_includes article.errors[:title], "can't be blank"
  end
end
```

Setup e Teardown

```ruby
class ArticleTest < ActiveSupport::TestCase
  setup do
    @user = User.create(name: "Mario", email: "mario@example.com")
  end

  test "articolo appartiene a un utente" do
    article = Article.new(title: "Test", user: @user)
    assert_equal @user, article.user
  end
end
```

- `setup`: eseguito **prima di ogni test**
- `teardown`: eseguito **dopo** (raramente necessario grazie alle transazioni)

Testing dei Modelli – Esempi Pratici

```
test "titolo deve essere presente" do
  article = Article.new(content: "Solo contenuto")
  assert_not article.valid?
  assert_includes article.errors[:title], "can't be blank"
end

test "contenuto può essere lungo" do
  long_text = "a" * 10_000
  article = Article.new(title: "Lungo", content: long_text)
  assert article.valid?
end
```

> **Best practice**: testa **ogni validazione**, **associazione**, e **metodo personalizzato**

Fixture vs Factory Bot

Fixture (Legacy)

```
# test/fixtures/articles.yml
first:
  title: Primo Post
  content: Questo è il contenuto
  created_at: <%= 1.day.ago %>
```

Problemi:

- Dati **statici**
- Difficile personalizzare
- Non scalabile

Factory Bot – Alternativa (1)

Factory Bot permette di creare dati di test **dinamici** e **personalizzabili**.

Repo

I dati dinamici vengono generati con la gemma **Faker** Repo

Faker mette a disposizione dei metodi per generare nomi, indirizzi, testi casuali e molto altro.

Installazione:

```
# Gemfile
group :test do
  gem 'factory_bot_rails'
  gem 'faker''
end
```

Factory Bot – Alternativa (1)

Definizione (`test/factories/articles.rb`):

```
FactoryBot.define do
  factory :article do
    title { "Articolo di #{Faker::Name.name}" }
    content { Faker::Lorem.paragraph }
    association :user

    trait :published do
      published_at { Time.current }
    end
  end
end
```

```
article = create(:article)                    # Nuovo record
article = build(:article)                     # Non salvato
article = create(:article, :published)        # Con trait
```

Testing dei Controller

```ruby
# test/controllers/articles_controller_test.rb
class ArticlesControllerTest < ActionDispatch::IntegrationTest
  test "dovrebbe mostrare la lista degli articoli" do
    get articles_url
    assert_response :success
    assert_select "h1", "Articles"
  end

  test "dovrebbe creare un articolo" do
    assert_difference("Article.count", 1) do
      post articles_url, params: { article: { title: "Nuovo", content: "Testo" } }
    end
    assert_redirected_to article_url(Article.last)
  end
end
```

- `assert_difference`: verifica che il numero di record aumenti
- `assert_redirected_to`: controlla il redirect
- `assert_select`: verifica il contenuto HTML

System Tests

Simulano un browser reale con **Capybara** e **Cuprite** (headless Chrome).

```ruby
# test/system/articles_test.rb
require "application_system_test_case"

class ArticlesTest < ApplicationSystemTestCase
  test "creare un articolo dall'interfaccia" do
    visit articles_url
    click_on "New Article"

    fill_in "Title", with: "Il mio primo post"
    fill_in "Content", with: "Contenuto dettagliato..."
    click_on "Create Article"

    assert_text "Article was successfully created"
    assert_text "Il mio primo post"
  end
end
```

Esecuzione dei Test

```
# Tutti i test
bin/rails test

# Solo model
bin/rails test:models

# Solo system (lenti ma completi)
bin/rails test:system

# Test paralleli (Rails 8 default)
bin/rails test --parallel

# Un test specifico
bin/rails test test/models/article_test.rb:15
```

Mock e Stub

Usati per **isolare il codice** dai servizi esterni.

```
test "invia email di benvenuto" do
  user = create(:user)

  assert_enqueued_email_with(
    UserMailer,
    :welcome,
    args: [user]
  ) do
    UserWelcomeJob.perform_later(user)
  end
end
```

- **Non testa il contenuto dell'email**, ma che sia **in coda**
- **Veloce** e **affidabile**

Best Practices (Rails 8)

Regola	Spiegazione
1 asserzione per test	Test più leggibili e focalizzati
Nomi descrittivi	`test "should create article with valid data"`
Factory Bot	Dati dinamici e realistici
System test per flussi utente	Simulano l'esperienza reale
Model test per logica	Validazioni, scope, metodi
Parallel testing	`bin/rails test --parallel`
Non testare view HTML	Testa il comportamento, non il markup

Quando usare ogni tipo di test

Tipo	Priorità	Quando usarlo
Model	Alta	Logica, validazioni
System	Alta	Flussi utente completi
Controller	Media	API, autorizzazioni
Integration	Bassa	Solo se necessario

Test Esercizi - Blog

```
# db/schema.rb
create_table "posts" do |t|
  t.string   "title"
  t.text     "ide "body"
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
end

create_table "comments" do |t|
  t.text     "content"
  t.integer  "post_id", null: false
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.index    ["post_id"], name: "index_comments_on_post_id"
end

add_foreign_key "comments", "posts"
```

Esercizio 1: Model Test – Post Validazioni

File: `test/models/post_test.rb`

Cosa codificare nel test?

1. `title` e `body` **obbligatori**
2. `title` massimo 128 caratteri
3. Cosa modificare per impostare la lunghezza massima?

Esercizio 2: Model Test – Comment e Associazioni

File: `test/models/comment_test.rb`

Cosa codificare nel test?

> `content` obbligatorio
> `post_id` obbligatorio
> `belongs_to :post`

Esercizio 3: Associazioni – Post has_many Comments

```ruby
# app/models/post.rb
class Post < ApplicationRecord
  has_many :comments, dependent: :destroy
  validates :title, :body, presence: true
  validates :title, length: { maximum: 128 }
end
```

```ruby
# app/models/comment.rb
class Comment < ApplicationRecord
  belongs_to :post
  validates :content, presence: true
end
```

- Andiamo a verificare che l'associazione `has_many` funzioni correttamente.
- Inoltre verifichiamo che cancellare un `Post` cancelli anche i `Comment` associati (Aiuto `assert_difference`).

Esercizio 4: Controller Test – PostsController

Dobbiamo scrivere un test per il `PostsController` che verifichi:

1. La risposta `index` è `success`
2. Il titolo del post è mostrato nella vista `index`
3. La vista `show` mostra il post corretto

Per implementare questi test possiamo utilizzare il metodo di setup per creare un post di esempio.

```
require "test_helper"

class PostsControllerTest < ActionDispatch::IntegrationTest
  setup do
    @post = Post.create!(title: "Test Post", body: "Contenuto del post")
  end

end
```

Esercizio 5: System Test – Navigazione e Visualizzazione

Possiamo utilizzare generator o creare il file manualmente, il file da creare è `test/system/posts_test.rb`.

```
rails generate system_test posts
```

- Andiamo a verificare che un utente possa visitare la pagina index dei post, cliccare su un post e vedere i dettagli del post insieme ai commenti associati.

Esercizio 6: Factory Bot – Dati Realistici

https://github.com/thoughtbot/factory_bot_rails

Faker, per la generazione di dati automatica: https://github.com/faker-ruby/faker

Dobbiamo installare le gemme factory_bot e rails aggiungendole al Gemfile:

```
group :test do
  gem 'factory_bot_rails'
  gem 'faker'
end
```

```
bundle install
```

Factory: `test/factories/posts.rb`

```ruby
FactoryBot.define do
  factory :post do
    title { Faker::Lorem.sentence(word_count: 3) }
    body { Faker::Lorem.paragraph(sentence_count: 3) }

    factory :post_with_comments do
      transient do
        comments_count { 3 }
      end

      after(:create) do |post, evaluator|
        create_list(:comment, evaluator.comments_count, post: post)
      end
    end
  end
end
```

Factory: `test/factories/comments.rb`

Dobbiamo anche creare la factory per i commenti:

```
FactoryBot.define do
  factory :comment do
    content { Faker::Lorem.sentence }
    post { association :post }
  end
end
```

Usiamo i Factory in un Test

Qui usiamo in grado di creare un post con commenti associati in modo semplice, andando a sfruttare la factory definita. Per esempio, possiamo modificare il file `test/models/post_test.rb` per includere un test che crea un post con 3 commenti associati:

```
include FactoryBot::Syntax::Methods  # Per rendere disponibili i metodi di FactoryBot
test "post con 3 commenti" do
  post = create(:post_with_comments, comments_count: 3)
  assert_equal 3, post.comments.count
end
```

Soluzione - Esercizio 1

```
require "test_helper"

class PostTest < ActiveSupport::TestCase
  test "valido con title e body" do
    post = Post.new(title: "Ciao", body: "Contenuto")
    assert post.valid?
  end

  test "non valido senza title" do
    post = Post.new(body: "Solo corpo")
    assert_not post.valid?
    assert_includes post.errors[:title], "can't be blank"
  end

  test "non valido senza body" do
    post = Post.new(title: "Solo titolo")
    assert_not post.valid?
    assert_includes post.errors[:body], "can't be blank"
  end

  test "title non può superare 100 caratteri" do
    long_title = "a" * 101
    post = Post.new(title: long_title, body: "Ok")
    assert_not post.valid?
    assert_includes post.errors[:title], "is too long"
  end
end
```

Soluzione - Esercizio 2

```ruby
require "test_helper"

class CommentTest < ActiveSupport::TestCase
  setup do
    @post = Post.create!(title: "Post", body: "Corpo")
  end

  test "valido con content e post" do
    comment = Comment.new(content: "Bravo!", post: @post)
    assert comment.valid?
  end

  test "non valido senza content" do
    comment = Comment.new(post: @post)
    assert_not comment.valid?
    assert_includes comment.errors[:content], "can't be blank"
  end

  test "non valido senza post" do
    comment = Comment.new(content: "Ok")
    assert_not comment.valid?
    assert_includes comment.errors[:post], "must exist"
  end

  test "appartiene a un post" do
    comment = Comment.new(content: "Ok", post: @post)
    assert_equal @post, comment.post
  end
end
```

Soluzione Esercizio 3

```
test "post ha molti commenti" do
  post = Post.create!(title: "Post", body: "Corpo")
  comment1 = post.comments.create!(content: "Primo")
  comment2 = post.comments.create!(content: "Secondo")

  assert_includes post.comments, comment1
  assert_includes post.comments, comment2
  assert_equal 2, post.comments.count
end

test "cancellare post cancella commenti" do
  post = Post.create!(title: "Post", body: "Corpo")
  post.comments.create!(content: "Commento")

  assert_difference "Comment.count", -1 do
    post.destroy
  end
end
```

Soluzione Esercizio 4

```
require "test_helper"

class PostsControllerTest < ActionDispatch::IntegrationTest
  setup do
    @post = Post.create!(title: "Test Post", body: "Contenuto del post")
  end

  test "should get index" do
                get posts_url
                assert_response :success
                assert_select "h2", text: @post.title
        end

        test "should show post" do
                get post_url(@post)
                assert_response :success
                assert_select "h1", text: @post.title
                assert_select "p", text: @post.body
        end

        test "should show post" do
                get post_url(@post)
                assert_response :success
                assert_select "h1", text: @post.title
                assert_select "p", text: @post.body
        end

end
```

Soluzione Esercizio 5

```
require "application_system_test_case"

class PostsTest < ApplicationSystemTestCase
  setup do
    @post = Post.create!(title: "Blog Post", body: "Questo è un post di esempio.")
    @comment = @post.comments.create!(content: "Ottimo articolo!")
  end

  test "visita index e vede i post" do
    visit posts_url
    assert_selector "h2", text: "Blog Post"
  end

  test "clicca sul post e vede dettagli e commenti" do
    visit posts_url
    click_on "Blog Post"

    assert_text "Questo è un post di esempio."
    assert_text "Ottimo articolo!"
    assert_current_path post_path(@post)
  end
end
```