Behat

Release 3.0.12

February 15, 2017

Conteúdo

Esta é uma tradução não oficial feita e mantida por Diego Santos, portanto pode estar desatualizada. Sinta-se encorajado para me ajudar a mantê-la atualizada.

Behat é um framework open source de Behavior Driven Development (BDD - Desenvolvimento Dirigido por Comportamento) para PHP 5.3+. O que é *Behavior Driven Development*, você pergunta? É uma maneira de desenvolver software através de uma comunicação constante com as partes interessadas em forma de exemplos; exemplos de como este programa deve ajudá-los, e você, para alcançar seus objetivos.

Por exemplo, imagine que você está prestes a criar o famoso comando UNIX `ls`. Antes de começar, você vai ter uma conversa com seus stakeholders (usuários de UNIX) e eles poderiam dizer que mesmo que eles gostem muito do UNIX, eles precisam de uma maneira de ver todos os arquivos no diretório de trabalho atual. Você então tem um bate-papo vai-e-vem com eles sobre como eles vêem esta característica trabalhando e que você venha com o seu primeiro cenário (um nome alternativo, por exemplo, na metodologia BDD):

```
# language: pt
Funcionalidade: Comando de listagem
  A fim de alterar a estrutura da pasta em que estou atualmente
  Como um usuário de UNIX
  Eu preciso ser capaz de ver os arquivos e pastas disponíveis atualmente lá

  Cenário: Listando dois arquivos em um diretório
    Dado que estou em um diretório "teste"
    E tenho um arquivo chamado "foo"
    E tenho um arquivo chamado "bar"
    Quando executar o "ls"
    Então eu devo ter:
      """
      bar
      foo
      """
```

Se você é um stakeholder, esta é a sua prova de que os desenvolvedores entenderam exatamente como você quer que esse recurso funcione. Se você é um desenvolvedor, esta é a sua prova de que o stakeholder espera que você implemente este recurso exatamente da maneira que você está planejando implementá-lo.

Então, como um desenvolvedor seu trabalho estará terminado tão logo você faça o comando `ls` e faça o comportamento descrito no cenário "Comando de listagem".

Você provavelmente já ouviu sobre a prática moderna de desenvolvimento chamada TDD, onde você escreve testes para o seu código antes, não depois, do código. Bem, BDD é parecido, exceto pelo fato de que você não precisa começar com um teste - seus *cenários* são seus testes. Isto é exatamente o que o Behat faz! Como você vai ver, Behat é fácil de aprender, rápido de usar e vai trazer a diversão de volta para os seus testes.

CAPÍTULO 1

Intro Rápida

Para começar a ser um *Behat'er* em 30 minutos, basta apenas mergulhar no guia de início rápido e desfrutar!

Construindo Modelo de Domínio

Bem vindo ao Behat! Behat é uma ferramenta para fechar o laço de comunicação do *Desenvolvimento Dirigido por Comportamento (BDD)*. BDD é uma metodologia de desenvolvimento de software baseado em exemplo por meio da comunicação contínua entre desenvolvedores e a área de negócios que esta aplicação suporta. Esta comunicação acontece de uma forma que a área de negócios e os desenvolvedores podem claramente entender - exemplos. Exemplos são estruturados entorno do padrão `Contexto-Ação-Resultado` e são escritos em um formato especial chamado *Gherkin*. O fato do Guerkin ser muito estrutural torna muito fácil automatizar testes de comportamento contra uma aplicação em desenvolvimento. Exemplos automatizados são utilizados atualmente para guiar o desenvolvimento de aplicações TDD-style.

Exemplo

Vamos imaginar que você está construindo uma plataforma totalmente nova de e-commerce. Uma das características fundamentais de qualquer plataforma de compras online é a habilidade de comprar produtos. Mas antes de comprar algo, os clientes devem poder informar ao sistema quais produtos eles têm interesse de comprar. Voc precisa de um carrinho de produtos. Então vamos escrever sua primeira user-story:

```
# language: pt
Funcionalidade: Carrinho de produtos
  A fim de comprar produtos
  Como um cliente
  Eu preciso colocar produtos do meu interesse no carrinho
```

Antes de nós começarmos a trabalhar nesta feature, nós precisamos preencher uma promessa de user-story e ter uma conversa de verdade com nossos stakeholders da área de negócios. Eles podem dizer que eles querem que os clientes vejam o preço combinado do produto no carrinho, mas que o preço reflita o imposto (20%) e o valor do frete (que depende da soma total dos produtos):

```
# language: pt
Funcionalidade: Carrinho de produtos
  A fim de comprar produtos
  Como um cliente
  Eu preciso colocar produtos do meu interesse no carrinho

  Regras:
  - O imposto é de 20%
```

```
  - O frete para um carrinho de compras até R$10 é R$3
  - O frete para um carrinho de compras maior que R$10 é R$2
```

Então como você pode ver, está ficando complicado (ambíguo, pelo menos) falar sobre está feature, em termos de *regras*. O que você entende por adicionar imposto? O que acontece quando nós tivermos dois produtos, um com valor menor que R$10 e outro de maior valor? Ao invés de você prosseguir em ter um leva e traz de conversas entre os stakeholders na forma dos exemplos atuais de um *cliente* adicionando produtos ao carrinho. Depois de algum tempo, você vai levantar seus primeiros exemplos de comportamentos (no BDD isto é chamado de *Cenários*):

```
# language: pt
Funcionalidade: Carrinho de produtos
  A fim de comprar produtos
  Como um cliente
  Eu preciso colocar produtos do meu interesse no carrinho

  Regras:
  - O imposto é de 20%
  - O frete para um carrinho de compras até R$10 é R$3
  - O frete para um carrinho de compras maior que R$10 é R$2

  Cenário: Comprando um único produto que custe menos que R$10
    Dado que exista um "Sabre de luz do Lorde Sith", que custe R$5
    Quando Eu adicionar o "Sabre de luz do Lorde Sith" ao carrinho
    Então Eu devo ter 1 produto no carrinho
    E o valor total do carrinho deve ser de R$9

  Cenário: Comprando um único produto que custe mais que R$10
    Dado que exista um "Sabre de luz do Lorde Sith", que custe R$15
    Quando Eu adicionar o "Sabre de luz do Lorde Sith" ao carrinho
    Então Eu devo ter 1 produto no carrinho
    E o valor total do carrinho deve ser de R$20

  Cenário: Comprando dois produtos que custem mais que R$10
    Dado que exista um "Sabre de luz do Lorde Sith", que custe R$10
    E que exista um "Sabre de luz Jedi", que custe R$5
    Quando Eu adicionar o "Sabre de luz do Lorde Sith" ao carrinho
    E Eu adicionar o "Sabre de luz Jedi" ao carrinho
    Então Eu devo ter 2 products no carrinho
    E o valor total do carrinho deve ser de R$20
```

Isto é seu e seus stakeholders compartilham da mesma escrita em um formato estruturado do projeto. Tudo é baseado na clara e construtiva conversa que vocês tiveram juntos. Agora você pode colocar este texto em um arquivo simples - `features/carrinho.feature` - dentro do diretório do seu projeto e começar a implementar a funcionalidade checando manualmente se se encaixa no cenário definido. Não é necessário nenhuma ferramenta (Behat em seu caso). Isto é, na essência, o que o BDD é.

Se você ainda está lendo, significa que você ainda espera mais. Ótimo! Porque apesar das ferramentas não serem a peça central do quebra-cabeça do BDD, elas melhoram todo o processo e adicionam muitos benefícios ao topo disto. Para isso, ferramentas como o Behat atualmente fecham o ciclo de comunicação da história. Isto significa que não somente você e seu stakeholder podem juntos definir como sua feature deveria trabalhar após ser implementada, ferramentas de BDD permitem a você automatizar a checagem do comportamento após a funcionalidade ser implementada. Então todo mundo sabe quando isto está feito e quando o time pode parar de escrever código. Isto, na essência, é oque o Behat é.

Behat é um executável que quando você o executa da linha de comando ele irá testar como a sua aplicação se comporta exatamente como você descreveu nos seus ‘‘.feature*‘‘ cenários.**

Indo adiante, nós vamos mostrar a você como o Behat pode ser usado para automatizar em particular esta feature

do carrinho de compras como um teste verificando se aquela aplicação (existindo ou não) trabalha como você e seus stakeholders esperam (de acordo com a conversa de vocês) também.

É isso ai! O Behat pode ser usado para automatizar qualquer coisa, inclusive relacionadas a funcionalidades web via Mink library.

Instalação

Antes de você começar, garanta que você tem uma versão superior a 5.3.3 do PHP instalada.

Método #1 - Composer (o recomendado)

O caminho oficial para instalar o Behat é através do Composer. Composer é um gerenciador de pacotes para PHP. Ele não irá lhe ser útil somente para instalar o Behat para você agora, ele será capaz de atualizar facilmente para a última versão mais tarde, quando for lançada. Se você ainda não tem o Composer, veja a documentação do Composer para instruções. Depois disto, basta ir ao diretório do projeto (ou criar um novo) e rodar:

```
$ php composer.phar require --dev behat/behat=~3.0.4
```

Então voc estará apto a checar a versão instalada do Behat:

```
$ vendor/bin/behat -V
```

Método #2 - PHAR (um caminho fácil)

Um caminho fácil para instalar o Behat é pegar a última `behat.phar` na página de download. Certifique-se de fazer o download de uma versão `3+`. Depois de baixar isto, basta colocá-lo no diretório do seu projeto (ou criar um novo) e checar a versão instalada usando:

```
$ php behat.phar -V
```

Desenvolvendo

Agora nós vamos usar nosso recém instalado Behat para automatizar a feature escrita anteriormente em `features/carrinho.feature`.

Nosso primeiro passo após descrever a feature e instalar o Behat é configurar a suite de teste. Uma suite de teste é um conceito chave em Behat. Suites são uma forma do Behat saber onde achar e como testar sua aplicação com as suas features. Por padrão, Behat vem com uma suite `default`, que diz ao Behat para procurar por features no diretório `features/` e os teste usando a classe `FeatureContext`. Vamos inicializar esta suite:

```
$ vendor/bin/behat --init
```

O comando `--init` diz ao Behat para prover para você com coisas faltando para começar a testar sua feature. Em nosso caso - é apenas uma classe `FeatureContext` no arquivo `features/bootstrap/FeatureContext.php`.

Executando o Behat

Eu acho que nós estamos prontos para ver o Behat em ação! Vamos rodar isto:

```
$ vendor/bin/behat
```

Voc deve ver que o Behat reconheceu que você tem 3 cenários. o Behat deve também contar a você que na sua classe `FeatureContext` faltam passos e propor trechos para etapas para você. `FeatureContext` é seu ambiente de teste. É um objeto através do qual você descreve como você deve testar sua aplicação através de suas features. Isso foi gerado através do comando `--init` e agora se parece com isso:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

class FeatureContext implements SnippetAcceptingContext
{
    /**
     * Initializes context.
     */
    public function __construct()
    {
    }
}
```

Definindo Steps

Finalmente, nós chegamos a parte de automação. Como é que o Behat sabe o que fazer quando vê `Dado que exista um "Sabre de luz do Lorde Sith", que custe R$5`? Diga a ele. Você escreve em PHP dentro da sua classe de contexto (`FeatureContext` no seu caso) e diz ao Behat que este código representa um passo específico do cenário (através de uma anotação com um padrão):

```
/**
 * @Given que exista um :arg1, que custe R$:arg2
 */
public function queExistaUmQueCusteR($arg1, $arg2)
{
    throw new PendingException();
}
```

`@Given que exista um :arg1, que custe R$:arg2` sobre o método diz ao Behat que este método em particular deve ser executado sempre que o Behat ver um step que se pareça com `...  que exista um ..., que custe R$...`. Este padrão combina qualquer um dos seguintes steps:

```
Dado que exista um "Sabre de luz do Lorde Sith", que custe R$5
Quando que exista um "Sabre de luz do Lorde Sith", que custe R$10
Então que exista um "Sabre de luz do Anakin", que custe R$10
E que exista um "Sabre de luz", que custe R$2
Mas que exista um "Sabre de luz", que custe R$25
```

Não somente estes, mas o Behat irá capturar tokens (palavras iniciadas com `:`, por exemplo `:arg1`) a partir do step e passar seu valor para o método como argumentos:

```
// Dado que exista um "Sabre de luz do Lorde Sith", que custe R$5
$context->queExistaUmQueCusteR('Sabre de luz do Lorde Sith', '5');

// Então que exista um "Sabre de luz Jedi", que custe R$10
$context->queExistaUmQueCusteR('Sabre de luz Jedi', '10');

// Mas que exista um "Sabre de luz", que custe R$25
$context->queExistaUmQueCusteR('Sabre de luz', '25');
```

Estes padrões podem ser muito poderosos, mas ao mesmo tempo, escreve-los por todos steps possíveis manualmente pode ser extremamente tedioso e chato. É por isso que o Behat faz isto para você. Relembre quando você executou anteriormente `vendor/bin/behat` você teve:

```
--- FeatureContext has missing steps. Define them with these snippets:

    /**
     * @Given que exista um :arg1, que custe R$:arg2
     */
    public function queExistaUmQueCusteR($arg1, $arg2)
    {
        throw new PendingException();
    }
```

O Behat gera automaticamente trechos para etapas que faltam e tudo que você precisa para os copiar e colar em sua classe context. Ou há ainda um caminho mais fácil - pasta rodar:

```
$ vendor/bin/behat --dry-run --append-snippets
```

E o Behat vai automaticamente acrescentar todos os métodos das etapas que faltam em sua classe `FeatureContext`. Como isso é legal?

Se voc executou *–append-snippets*', sua `FeatureContext` deve se parecer com:

```php
// features/bootstrap/FeatureContext.php

use Behat\Behat\Tester\Exception\PendingException;
use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

class FeatureContext implements SnippetAcceptingContext
{
    /**
     * @Given que exista um :arg1, que custe R$:arg2
     */
    public function queExistaUmQueCusteR($arg1, $arg2)
    {
        throw new PendingException();
    }

    /**
     * @When Eu adicionar o :arg1 ao carrinho
     */
    public function euAdicionarOAoCarrinho($arg1)
    {
        throw new PendingException();
    }

    /**
     * @Then Eu devo ter :arg1 produto(s) no carrinho
     */
    public function euDevoTerProdutoNoCarrinho($arg1)
    {
        throw new PendingException();
    }

    /**
     * @Then o valor total do carrinho deve ser de R$:arg1
     */
```

```
    public function oValorTotalDoCarrinhoDeveSerDeR($arg1)
    {
        throw new PendingException();
    }
}
```

Automating Steps

Agora finalmente é o tempo de começar a implementar nossa feature do carrinho de compras. A abordagem quando você usa testes para dirigir o desenvolvimento da sua aplicação é chamada de Test-Driven Development (ou simplesmente TDD). Com o TDD você inicia definindo casos de testes para a funcionalidade que você vai desenvolver, em seguida você preenche estes casos de teste com o melhor código da aplicação que você poderia chegar (use suas habilidades de design e imaginação).

No caso do Behat, você já tem casos de teste definidos (step definitions em sua `FeatureContext`) e a unica coisa que está faltando é o melhor código da aplicação que poderíamos chegar para cumprir o nosso cenário. Algo assim:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Tester\Exception\PendingException;
use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

class FeatureContext implements SnippetAcceptingContext
{
    private $prateleira;
    private $carrinho;

    public function __construct()
    {
        $this->prateleira = new Prateleira();
        $this->carrinho = new Carrinho($this->prateleira);
    }

    /**
     * @Given que exista um :produto, que custe R$:valor
     */
    public function queExistaUmQueCuste($produto, $valor)
    {
        $this->prateleira->colocaValorProduto($produto, floatval($valor));
    }

    /**
     * @When Eu adicionar o :produto ao carrinho
     */
    public function euAdicionarOAoCarrinho($produto)
    {
        $this->carrinho->adicionaProduto($produto);
    }

    /**
     * @Then Eu devo ter :quantidade produto(s) no carrinho
     */
    public function euDevoTerProdutoNoCarrinho($quantidade)
    {
        PHPUnit_Framework_Assert::assertCount(
```

```
            intval($quantidade),
            $this->carrinho
        );
    }

    /**
     * @Then o valor total do carrinho deve ser de R$:valor
     */
    public function oValorTotalDoCarrinhoDeveSerDeR($valor)
    {
        PHPUnit_Framework_Assert::assertSame(
            floatval($valor),
            $this->carrinho->pegaValorTotal()
        );
    }
}
```

Como você pode ver, afim de implementar e testar nossa aplicação, nós introduzimos 2 objetos - `Prateleira` and `Carrinho`. O primeiro responsavel por armazenar produtos e seus preços, o segundo é responsável pela representação do carrinho do nosso cliente. Através do step definitions apropriado nós declaramos produtos' preços e adicionamos ao carrinho. Nós então comparamos o estado de nosso objeto `Carrinho` com a nossa expectativa usando asserções do PHPUnit.

Agora vamos tentar executar seu teste funcional:

```
$ vendor/bin/behat
```

Você deve ver o início da feature e em seguida um erro dizendo que a classe `Prateleira` não existe. Isso significa que estamos prontos para começar a efetivamente escrever código da aplicação!

Implementando a Feature

Então agora nós temos 2 coisas muito importantes:

1. Uma concreta descrição da funcionalidade que estamos tentando entregar.
2. Ao falhar, o teste nos diz o que fazer a seguir.

Agora a parte mais fácil do desenvolvimento da aplicação - implementação da feature. Sim, com TDD e BDD a implementação se torna uma rotina, devido você já ter a maioria do trabalho nas fases anteriores - você escreveu os testes, voc veio com uma solução elegante (tanto quanto você poderia dar no contexto atual) e você escolhe os atores (objetos) e ações (métodos) que estão envolvidos. Agora é a hora de escrever um punhado de palavras chave em PHP para colar tudo junto. Ferramentas como o Behat, quando usadas da forma correta, vão ajudar voc a escrever esta fase, lhe dando um simples conjunto de instruções que você precisa para seguir. Você fez seu pensamento e projeto,agora está na hora de sentar, rodar a ferramenta e seguir as instruções na ordem para escrever seu código de produção.

Vamos começar! Rode:

```
$ vendor/bin/behat
```

O Behat vai tentar testar a sua aplicação com o `FeatureContext` mas vai falhar logo, exibindo algum evento como este em sua tela:

```
Fatal error: Class 'Prateleira' not found
```

Agora nosso trabalho é reinterpretar esta frase em uma instrução executável. Como "Criar a classe `Prateleira`". Vamos criar isto em `features/bootstrap`:

```
// features/bootstrap/Shelf.php

final class Prateleira
{
}
```

Vamos executar o Behat novamente:

```
$ vendor/bin/behat
```

Nós vamos ter uma mensagem diferente em nossa tela:

```
Fatal error: Class 'Carrinho' not found
```

Ótimo, nós estamos progredindo! Reinterpretando a mensagem como "Criar a classe `Carrinho`". Vamos seguir nossa nova instrução:

```
// features/bootstrap/Carrinho.php

final class Carrinho
{
}
```

Rode o Behat novamente:

```
$> vendor/bin/behat
```

Maravilha! Outra "instrução":

```
Call to undefined method Prateleira::colocaValorProduto()
```

Seguindo estas instruções passo-a-passo você vai terminar com uma classe `Prateleira` parecida com esta:

```
// features/bootstrap/Prateleira.php

final class Prateleira
{
    private $valores = array();

    public function colocaValorProduto($produto, $valor)
    {
        $this->valores[$produto] = $valor;
    }

    public function pegaValorProduto($produto)
    {
        return $this->valores[$produto];
    }
}
```

E uma classe `Carrinho` parecida com esta:

```
// features/bootstrap/Carrinho.php

final class Carrinho implements \Countable
{
    private $prateleira;
    private $produtos;
    private $valoresProdutos = 0.0;
```

```
    public function __construct(Prateleira $prateleira)
    {
        $this->prateleira = $prateleira;
    }

    public function adicionaProduto($produto)
    {
        $this->produtos[] = $produto;
        $this->valoresProdutos += $this->prateleira->pegaValorProduto($produto);
    }

    public function pegaValorTotal()
    {
        return $this->valoresProdutos
            + ($this->valoresProdutos * 0.2)
            + ($this->valoresProdutos > 10 ? 2.0 : 3.0);
    }

    public function contador()
    {
        return contador($this->produtos);
    }
}
```

Execute o Behat novamente:

```
$ vendor/bin/behat
```

Todos os cenários devem passar agora! Parabéns, você quase terminou a sua primeira feature. O último passo é *refatorar*. Olhe para as classes `Carrinho` e `Prateleira` e tente achar um caminho para fazer um código mais limpo, fácil de ler e conciso.

Depois da refatoração pronta, voc terá:

1. Um código óbvio e claramente concebido que faz exatamente o que deveria fazer sem funcionalidades que não foram solicitadas pelos usuários.
2. Um conjunto de testes de regressão que irá ajudá-lo a ter confiança em seu código daqui para frente.
3. Uma documentação viva do comportaento do seu código,
4. Documentação viva do comportamento do seu código que vai viver, evoluir e morrer em conjunto com o seu código.
5. Um incrível nível de confiança em seu código. Não só você está confiante agora que ele faz exatamente o que é suposto fazer, você está confiante de que ele faz isso por entregar valor para os usuários finais (clientes, no nosso caso).

Existem muitos outros beneficios no BDD, mas estes são as razões chaves porque a maioria dos praticantes de BDD fazem BDD em Ruby, .Net, Java, Python e JS. Bem vindo a família!

What's Next?

Parabéns! Você agora conhece tudo o que precisa para começar com o desenvolvimento dirigido por testes e Behat. Daqui, voc pode aprender mais sobre a sintaxe *Gherkin* ou aprender como testar suas aplicações web usando Behat com Mink.

CAPÍTULO 2

Guias

Aprenda Behat com os seguintes guias:

Escrevendo Features

Behat é uma ferramenta para testar o o comportamento de sua aplicação, utilizando uma linguagem especial chamada Gherkin. Gherkin é uma Business Readable, Domain Specific Language criada especificamente para a descrição de comportamentos. Isto lhe dá a habilidade de remover detalhes lógicos dos testes de comportamento.

O Gherkin serve como documentação do seu projeto, bem como para testes automatizados. O Behat também tem uma característica bônus: Ele fala para você usando linguagem verdadeira, humana lhe dizendo o código que você deve escrever.

> Dica:
>
> Se você ainda é novo no Behat, vá para *Construindo Modelo de Domínio* primeiro e então retorne aqui para aprender mais sobre Gherkin.

Sintaxe Gherkin

Bem como YAML e Python, Gherkin é uma linguagem orientada a espaços, ela usa indentação para definir a estrutura. Os fins de linha encerram as declarações (denominados etapas) e espaços ou tabs também podem ser usados para indentação (nós sugerimos a você usar espaços para melhor portabilidade). Finalmente, a maioria das linhas em Gherkin iniciam com uma palavra chave especial:

```
# language: pt
Funcionalidade: Algum texto descritivo conciso do que é desejado
  A fim de realizar um valor de negócio
  Como ator explicito do sistema
  Eu quero ganhar algum resultado benéfico que promova a meta

 Texto adicional...

  Cenário: Uma determinada situação de negócios
    Dado uma pré condição
    E uma outra pré condição
    Quando uma ação é feita pelo ator
    E uma outra ação
    E outra ação diferente
    Então um resultado testável é alcançado
    E outra coisa que possamos verificar também acontece
```

```
  Cenário: Uma situação diferente
    ...
```

O analisador divide a entrada em funcionalidades, cenários e etapas. Vamos analisar o exemplo:

.# **`Funcionalidade: Algum texto descritivo conciso do que é desejado`** inicia a feature e lhe dá um título. Aprenda mais sobre Funcionalidades na seção "**'Features'_**".

.# **As próximas três linhas (`A fim de ...`, `Como um ...`, `Eu quero...`)** dão um contexto para as pessoas que leem a sua funcionalidade e descreve o valor do negócio derivada da inclusão da funcionalidade em seu software. Estas linhas não são analisadas pelo Behat e não requerem uma estrutura.

.# **`Cenário: Uma determinada situação de negócios` inicia o cenário e** contêm uma descrição do cenário. Aprenda mais sobre cenários na seção "**'Scenarios'_**"

.# **As próximas 7 linhas são etapas do cenário, cada uma das quais é comparada** com um padrão definido em outro lugar. Aprenda mais sobre etapas na seção "**'Steps'_**"

.# `Cenário: Uma situação diferente` inicia o próximo cenário e assim por diante.

Quando você está executando a funcionalidade, a porção da direita de cada etapa (após as palavras chaves como `Dado`, `E`, `Quando`, etc) coincide com um padrão, que executa uma função callback do PHP. Você pode ler mais sobre etapas de coincidências e execução em *Definindo Ações Reutilizáveis*.

Funcionalidades

Todos arquivos `*.feature` convencionalmente consistem em uma funcionalidade única. Linhas iniciando com a palavra chave `Funcionalidade:` seguido de três linhas identadas iniciam uma funcionalidade. Usualmente uma Funcionalidade contém uma lista de Cenários. Você pode escrever qualquer coisa que você precise até o primeiro cenário, que inicia com `Cenário:` (ou o seu equivalente) em uma nova linha. Você pode usar tags para agrupar Funcionalidades e Cenários, independente da estrutura do seu arquivo e diretório.

Todos cenários consistem em uma lista de etapas, que devem iniciar com uma das palavras chaves `Dado`, `Quando`, `Então`, `Mas` ou `E`. O Behat trata eles do mesmo modo, mas você não deve fazer isto. Aqui temos um exemplo:

```
# language: pt
Funcionalidade: Servir café
    A fim de ganhar dinheiro
    Os clientes devem ser capazes de
    comprar café a todo momento

  Cenário: Compra último café
    Dado que tenha 1 café sobrando na máquina
    E eu tenha depositado 1 real
    Quando eu pressionar o botão de café
    Então eu deveria ser servido de um café
```

Além do básico Cenário, uma funcionalidade pode conter Esquema do Cenário e Contexto.

Cenário

Cenários são uma das principais estruturas do Gherkin. Todo cenário deve iniciar com a palavra chave `Cenário:`, opcionalmente seguido de um título de cenário. Cada funcionalidade pode ter um ou mais cenários e todo cenário consiste em uma ou mais **'etapa'_**.

Os cenários seguintes tem cada um 3 etapas:

```
Cenário: Wilson posta em seu blog
  Dado que eu estou logado como Wilson
  Quando eu tento postar "A terapia cara"
  Então eu devo ver "Seu artigo foi publicado."

Cenário: Wilson falha ao postar algo no blog de outra pessoa
  Dado que eu estou logado como Wilson
  Quando eu tento postar "Greg esbraveja contra impostos"
  Então eu devo ver "Hey! Este não é o seu blog!"

Cenário: Greg posta em blog cliente
  Dado que eu estou logado como Greg
  Quando eu tento postar "Terapia Cara"
  Então eu devo ver "Seu artigo foi publicado."
```

Esquema do Cenário

Copiar e colar cenários para usar diferentes valores pode ser muito tedioso e repetitivo:

```
Cenário: Comer 5 em cada 12
  Dado que tenho 12 pepinos
  Quando eu comer 5 pepinos
  Então eu devo ter 7 pepinos

Cenário: Comer 5 em cada 20
  Dado que tenho 20 pepinos
  Quando eu comer 5 pepinos
  Então eu devo ter 15 pepinos
```

Os *Esquemas do Cenários* nos permitem formular estes exemplos com maior precisão através da utilização de um modelo com espaços reservados:

```
Esquema do Cenário: Comendo
  Dado que tenho <antes> pepinos
  Quando eu comer <come> pepinos
  Então eu devo ter <depois> pepinos

  Exemplos:
    | antes | come | depois |
    |  12   |  5   |   7    |
    |  20   |  5   |   15   |
```

As etapas do Esquema do Cenário fornecem um modelo que nunca é executado diretamente. Um Esquema do Cenário é executado uma vez para cada linha na seção de exemplos abaixo dela (exceto para a primeira linha que é o cabeçalho).

O Esquema do Cenário utiliza espaços reservados, que estão contidos < > nas etapas de saída do Cenário. Por exemplo:

```
Dado <Eu sou um espaço reservado e estou ok>
```

Pense em um espaço reservado como uma variável. Isto pode ser substituído por um valor real das linhas da tabela de `Exemplos:`, onde o texto entre os < > de espaço reservado corresponde ao cabeçalho da coluna da tabela. O valor substituído pelo espaço reservado muda a cada execução subsequente do Esquema do Cenário, até que o fim da tabela de `Exemplos` seja alcançado.

Tip: Você também pode usar os espaços reservados em Argumentos Multilineos.

Note: Sua etapa de definições nunca terá que coincidir com o próprio texto do espaço reservado, mas sim os valores terão que substituir o espaço reservado.

Então quando executamos a primeira linha do nosso exemplo:

```
Esquema do Cenário: Comer
  Dado que temos <antes> pepinos
  Quando eu comer <come> pepino
  Então teremos <depois> pepinos

  Exemplos:
    | antes | come | depois |
    |  12   |   5  |   7    |
```

O cenário que realmente é executado é:

```
Cenário: Comer
  # <antes> é substituído por 12:
  Dado que temos 12 pepinos
  # <come> é substituído por 5:
  Quando eu comer 5 pepino
  # <depois> é substituído por 7:
  Então teremos 7 pepinos
```

Contexto

Contexto permite a você adicionar algum contexto a todos os cenários em um único recurso. Um Contexto é como um Cenário sem título, que contém uma série de etapas. A diferença ocorre quando ele é executado: o contexto será executado *antes de cada* um de seus cenários, mas depois dos seus hooks `BeforeScenario` (*Hooking no Processo de Teste*).

```
# language: pt
Funcionalidade: Suporte a múltiplos sites

Contexto:
    Dado um administrador global chamado "Greg"
    E um blog chamado "Greg esbraveja contra impostos"
    E um cliente chamado "Wilson"
    E um blog chamado "Terapia Cara" de propriedade de "Wilson"

Cenário: Wilson posta em seu próprio blog
    Dado que eu esteja logado como Wilson
    Quando eu tentar postar em "Terapia Cara"
    Então eu devo ver "Seu artigo foi publicado."

Cenário: Greg posta no blog de um cliente
    Dado que eu esteja logado como Greg
    Quando eu tentar postar em "Terapia Cara"
    Então eu devo ver "Seu artigo foi publicado"
```

Etapas

Funcionalidades consistem em etapas, também conhecidas como Dado, Quando e Então.

O Behat não faz distinção técnica entre estes três tipos de etapas, contudo, nós recomendamos fortemente que você faça! Estas palavras foram cuidadosamente selecionadas para o seu propósito e você deve saber que o objetivo é entrar na mentalidade BDD.

Robert C. Martin escreveu um ótimo post sobre o conceito de BDD Dado-Quando-Então onde ele pensa neles como uma máquina de estados finitos.

Dado

O propósito da etapa **Dado** é **colocar o sistema em um estado conhecido** antes do usuário (ou sistema externo) iniciar a interação com o sistema (na etapa Quando). Evite falar sobre a interação em Dado. Se você trabalhou com casos de uso, Dado é a sua pré condição.

Exemplos de Dado

Dois bons exemplos do uso de **Dado** são:

- Para criar registros (instâncias de modelo) ou de configuração do banco de dados:

```
Dado que não tenha usuários no site
Dado que o banco de dados esteja limpo
```

- Autenticar um usuário (uma exceção a recomendação de não-interação Coisas que "aconteceram antes" estão ok):

```
Dado que eu esteja logado como "Everzet"
```

Tip: Tudo bem chamar a camada de "dentro" da camada de interface do usuário aqui (no Symfony: falar com os modelos).

Usando Dado como massa de dados:

Se você usa ORMs como Doctrine ou Propel, nós recomendamos a utilização de uma etapa Dado com o argumento **'tabela'**_ para configurar registros em vez de objetos. Neste caminho você pode ler todos os cenários em um único lugar e fazer sentido fora dele sem ter que saltar entre arquivos:

```
Dado estes usuários:
| username | password | email               |
| everzet  | 123456   | everzet@knplabs.com |
| fabpot   | 22@222   | fabpot@symfony.com  |
```

Quando

O propósito da etapa **Quando** é **descrever a ação chave** que o usuário executa (ou, usando a metáfora de Robert C. Martin, a transição de estado).

Exemplos de Quando

Dois bons exemplos do uso de **Quando** são:

- Interagir com uma página web (a biblioteca Mink lhe dá muitas etapas `Quando` web amigáveis):

```
Quando eu estiver em "/alguma/pagina"
Quando eu preencho o campo "username" com "everzet"
Quando eu preencho o campo "password" com "123456"
Quando eu clico em "login"
```

- Interagir com alguma biblioteca CLI (chama comandos e grava saída):

```
Quando eu chamo "ls -la"
```

Então

O propósito da etapa **Então** é **observar saídas**. As observações devem estar relacionadas com o valor/benefício de negócios na sua descrição da funcionalidade. As observações devem inspecionar a saída do sistema (um relatório, interface de usuário, mensagem, saída de comando) e não alguma coisa profundamente enterrado dentro dela (que não tem valor de negócios e ao invés disso faz parte da implementação).

Exemplos de Então

Dois bons exemplos do uso de **Então** são:

- Verificar algo relacionado ao Dado + Quando está (ou não) na saída:

```
Quando eu chamo "echo hello"
Então a saída deve ser "hello"
```

- Checar se algum sistema externo recebeu a mensagem esperada:

```
Quando eu enviar um email com:
  """
  ...
  """
Então o cliente deve receber um email com:
  """
  ...
  """
```

Caution: Embora possa ser tentador implementar etapas Então para apenas olhar no banco de dados - resista à tentação. Você deve verificar somente saídas que podem ser observadas pelo usuário (ou sistema externo). Se a base de dados somente é visível internamente por sua aplicação, mas é finalmente exposta pela saída do seu sistema em um navegador web, na linha de comando ou uma mensagem de email.

E, Mas

Se você tem várias etapas Dado, Quando ou Então você pode escrever:

```
Cenário: Múltiplos Dado
  Dado uma coisa
  Dado outra coisa
  Dado mais outra coisa
  Quando eu abrir meus olhos
```

```
  Então eu verei qualquer coisa
  Então eu não verei qualquer outra coisa
```

Ou você pode usar etapas **E** ou **Mas**, permitindo uma leitura mais fluente do seu Cenário:

```
Cenário: Múltiplos Dado
  Dado uma coisa
  E outra coisa
  E mais outra coisa
  Quando eu abrir meus olhos
  Então eu verei qualquer coisa
  Mas eu não verei qualquer outra coisa
```

O Behat interpreta as etapas iniciando com E ou Mas exatamente como as outras etapas, não faz distinção entre elas - Mas você deve!

Argumentos Multilineos

A linha única etapas permite ao Behat extrair pequenas strings de suas etapas e recebê-los em suas definições de etapas. No entanto, há momentos em que você quer passar uma estrutura de dados mais rica a uma definição de etapa.

Para isto foram projetados os Argumentos Multilineos. Eles são escritos nas linhas que seguem imediatamente uma etapa e são passadas para o método definição de etapa como um último argumento.

Etapas de Argumentos Multilineos vem em dois modos: tabelas ou pystrings.

Tabelas

As tabelas são etapas de argumentos úteis para a especificação de um grande conjunto de dados - normalmente como entrada para uma saída de `Dado` ou como espera de um `Então`.

```
Cenário:
  Dado que as seguintes pessoas existem:
    | nome  | email           | fone  |
    | Aslak | aslak@email.com | 123   |
    | Joe   | joe@email.com   | 234   |
    | Bryan | bryan@email.org | 456   |
```

Attention: Não confunda tabelas com **'Esquemas do cenário'_** - sintaticamente eles são idênticos, mas eles tem propósitos diferentes. Esquemas declaram diferentes valores múltiplos ao mesmo cenário, enquanto tabelas são usadas para esperar um conjunto de dados.

Tabelas correspondentes em sua Step Definition

Uma definição correspondente para esta etapa se parece com isso:

```
use Behat\Gherkin\Node\TableNode;

// ...

/**
 * @Given as seguintes pessoas existem:
 */
public function asSeguintesPessoasExistem(TableNode $tabela)
{
    foreach ($tabela as $linha) {
        // $linha['nome'], $linha['email'], $linha['fone']
    }
}
```

Uma tabela é injetada na definição do objeto `TableNode`, com o qual você pode obter um hash de colunas (método `TableNode::getHast()`) ou por linhas (`TableNode::getRowsHash()`).

PyStrings

Strings multilineas (também conhecidas como PyStrings) são úteis para a especificação de um grande pedaço de texto. O texto deve ser fechado por delimitadores que consistem em três marcas de aspas duplas (‘‘ “”” ‘‘), colocadas em linha:

```
Cenário:
  Dado uma postagem em um blog chamado "Random" com:
    """
    Algum título, Eh?
    =================
    Aqui está o primeiro parágrafo do meu post.
    Lorem ipsum dolor sit amet, consectetur adipiscing
    elit.
    """
```

Note: A inspiração para o PyString vem do Python onde `"""` é usado para delimitar docstrings, mais ou menos como `/** ... */` é usado para docblocks em PHP.

PyStrings correspondentes em sua step definitions

Em sua step definition, não precisa procurar por este texto e corresponder com o seu padrão. O texto vai automaticamente passar pelo último argumento no método step definition. Por exemplo:

```
use Behat\Gherkin\Node\PyStringNode;

// ...

/**
 * @Given um post em um blog chamado :titulo com:
 */
public function umPostEmUmBlogChamado($titulo, PyStringNode $texto)
{
    $this->criarPost($titulo, $texto->getRaw());
}
```

PyStrings são armazenadas em uma instancia `PyStringNode`, que você pode simplesmente converter a uma string com `(string) $pystring` ou `$pystring->getRaw()` como no exemplo acima.

Note: A indentação para abrir `"""` não é importante, apesar de ser uma prática comum deixar dois espaços da etapa de fechamento. A identação dentro das aspas triplas, entretanto, é significante. Cada linha da string passa pela chamada da definição de etapa e será re-indentada de acordo com a abertura `"""`. A indentação além da coluna de abertura `"""` por conseguinte, será preservada.

Tags

Tags são uma ótima forma de organizar suas funcionalidades e cenários. Considere este exemplo:

```
@faturamento
Feature: Verifica o faturamento

  @importante
  Cenário: Falta da descrição do produto

  Cenário: Vários produtos
```

Um Cenário ou Funcionalidade pode ter quantas tags você quiser, basta apenas separá-los com espaços:

```
@faturamento @brigar @incomodar
Funcionalidade: Verificar o faturamento
```

Note: Se uma tag existe em uma `Funcionalidade`, o Behat irá atribuir essa tag para todos os `Cenários` filhos e `Esquemas do Cenário` também.

Gherkin em Muitas Línguas

O Gherkin está disponível em muitas linguagens, permitindo você escrever histórias usando as palavras chave de sua linguagem. Em outras palavras, se você fala Francês, você pode usar a palavra `Fonctionnalité` ao invés de `Funcionalidade`.

Para checar se o Behat e o Gherkin suportam a sua língua (Francês, por exemplo), execute:

```
behat --story-syntax --lang=fr
```

Note: Guarde em sua mente que qualquer linguagem diferente de `en` precisa ser explicitada com um comentário `#language: ...` no início de seu arquivo `*.feature`:

```
# language: fr
Fonctionnalité: ...ta
  ...
```

Desta forma, suas funcionalidades realizarão todas as informações sobre o seu tipo de conteúdo, o que é muito importante para metodologias como BDD e também dá ao Behat a capacidade de ter recursos de vários idiomas em uma suíte.

Definindo Ações Reutilizáveis

A linguagem Gherkin fornece uma maneira de descrever o comportamento da sua aplicação em uma linguagem compreensível de negócios. Mas como você testa se o comportamento descrito realmente foi implementado? Ou se esta aplicação satisfaz as expectativas de negócios descritas nos cenários da funcionalidade? O Behat provê uma maneira para mapear suas etapas de cenário (ações) uma a uma com o código PHP chamado definições de etapa:

```
/**
 * @When eu fizer algo com :argumento
 */
public function euFizerAlgoCom($argumento)
{
    // fazer algo com o $argumento
}
```

A Casa das Definições - A Classe `FeatureContext`

As definições de etapa são apenas métodos normais PHP. Eles são métodos de instâncias de em uma classe especial chamada *FeatureContext*. Esta classe pode ser facilmente criada executando `behat` com o comando `--init` do diretório do seu projeto:

```
$ vendor/bin/behat --init
```

Depois de você executar este comando, o Behat irá configurar um diretório `features` dentro do seu projeto:

A recentemente criada `features/bootstrap/FeatureContext.php` terá uma classe contexto inicial para você começar:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

class FeatureContext implements SnippetAcceptingContext
{
    /**
     * Inicializa o contexto
     */
    public function __construct()
    {
```

```
    }
}
```

Todas definições de etapa e *hooks* necessárias para testar seu projeto contra suas funcionalidades serão representadas por métodos dentro desta classe.

Criando Sua Primeira Definição de Etapa

O principal objetivo da Definição de etapa é ser executada quando o Behat ver a sua etapa correspondente no cenário de execução. Contudo, não é apenas porque um método existe em `FeatureContext` que o Behat irá encontrá-lo. O Behat precisa de uma forma para verificar se um método de classe concreta é adequado a uma etapa concreta em um cenário. O Behat corresponde os métodos do `FeatureContext` com a definição de etapa utilizando padrão de correspondência.

Quando o Behat é executado, ele compara as linhas de etapas do Gherkin de cada cenário com os padrões vinculados a cada método em seu `FeatureContext`. Se a linha do Gherkin satisfaz o padrão vinculado, a sua definição de etapa correspondente é executada. Simples assim!

O Behat utiliza anotações php-doc para vincular padrões com os métodos do `FeatureContext`:

```
/**
 * @When eu fizer algo com :argumentoDoMetodo
 */
public function algumMetodo($argumentoDoMetodo) {}
```

Vamos dar uma olhada neste código:

1. `@When` é uma palavra chave definida. Há 3 palavras-chave suportadas em anotações: `@Given`/`@When`/`@Then`. Estas três palavras-chave de definição atualmente são equivalentes, mas todas as três permanecem disponíveis para que a sua definição de etapa permaneça legível.
2. O texto depois da palavra-chave é a etapa padrão de texto(por exemplo: `eu fizer algo com :argumentoDoMetodo`).
3. Todos os valores símbolos do padrão (por exemplo `:argumentoDoMetodo`) são capturados e passados ao método como um argumento com o mesmo nome (`$argumentoDoMetodo`).

Note: Note que o bloco de comentário inicia com `/**`, e não o usual `/*`. Isto é importante para o Behat ser capaz de analisar tais comentários como anotações!

Como você já deve ter notado, este padrão é bastante geral e seu método correspondente será chamado pelas etapas que contenham `...eu fizer algo com...`, incluindo:

```
Given eu fizer algo com "string1"
When eu fizer algo com 'alguma outra string'
Then eu fizer algo com 25
```

A única diferença real entre essas etapas aos olhos do Behat é o texto símbolo capturado. Este texto será passado para as etapas do método o correspondente valor de argumento. No exemplo acima, `FeatureContext::algumMetodo()` será chamado três vezes, em cada vez com um argumento diferente:

1. `$context->algumMetodo($argumentoDoMetodo = 'string1');`.
2. `$context->algumMetodo($argumentoDoMetodo = 'alguma outra string');`.
3. `$context->algumMetodo($argumentoDoMetodo = '25');`.

Note: Um padrão não pode determinar automaticamente o tipo de dados de suas correspondências, então todos os

argumentos dos métodos são passados como strings. Até mesmo que seu padrão corresponda a "500", que pode ser considerado como um inteiro, '500' será passado como um argumento string para o método

definições de etapa.

Isto não é uma funcionalidade ou limitação do Behat, mas sim uma forma inerente da string corresponder. É sua responsabilidade converter os argumentos string para inteiro, ponto flutuante ou booleano onde for aplicável dado o código que você está testando.

A conversão de argumentos para tipos específicos pode ser feita usando **'etapa transformações de argumento'_**.

Note: O Behat não diferencia palavras-chave da etapa quando corresponde padrões para métodos. Assim uma etapa definida com `@When` também poderia ser correspondida com `@Given ...`, `@Then ...`, `@And ...`, `@But ...`, etc.

Sua definições de etapa também pode definir múltiplos argumentos para passar para o método `FeatureContext` correspondente:

```
/**
 * @When eu fizer algo com :argumentoString e com :argumentoNumero
 */
public function algumMetodo($argumentoString, $argumentoNumero) {}
```

Você também pode especificar palavras alternativas e partes opcionais de palavras, como esta:

```
/**
 * @When aqui esta/estão :quantidade monstro(s)
 */
public function aquiEstaoMonstros($quantidade) {}
```

Se você precisa de um algoritmo de correspondência muito mais complicado, você sempre pode usar a boa e velha expressão regular:

```
/**
 * @When /^aqui (?:esta|estao) (\d+) monstros?$/i
 */
public function aquiEstaoMonstros($quantidade) {}
```

Definição de Fragmentos

Agora você sabe como escrever definições de etapa à mão, mas escrever todos estes métodos raiz, anotações e padrões à mão é tedioso. O Behat torna esta tarefa rotineira muito fácil e divertida com a geração de Definição de Fragmentos para você! Vamos fingir que você tenha esta funcionalidade:

```
Funcionalidade:
  Cenário:
    Dado alguma etapa com um argumento "string"
    E uma etapa com número 23
```

Se a sua classe contexto implementa a interface `Behat\Behat\Context\SnippetAcceptingContext` e você testa uma funcionalidade com etapas em falta no Behat:

```
$ vendor/bin/behat features/exemplo.feature
```

O Behat irá providenciar fragmentos gerados automaticamente para sua classe contexto.

Ele não somente gera o tipo de definição adequada (`@Given`), mas também propõe um padrão com o símbolo capturado (`:arg1`, `:arg2`), nome do método (`algumaEtapaComUmArgumento()`, `umaEtapaComNumero()`) e argumentos (`$arg1`, `$arg2`), todos baseados no texto da etapa. Não é legal?

A única coisa que falta para você fazer é copiar estes fragmentos de métodos para a sua classe `FeatureContext` e fornecer um corpo útil para eles. Ou melhor ainda, executar o behat com a opção `--append-snippets`:

```
$ vendor/bin/behat features/exemplo.feature --dry-run --append-snippets
```

`--append-snippets` diz ao behat para adicionar automaticamente fragmentos dentro de sua classe contexto.

Note: A implementação da interface `SnippetAcceptingContext` diz ao Behat que seu contexto espera fragmentos a serem gerados no seu interior. O Behat vai gerar padrões simples de fragmentos para você, mas se a sua for uma expressão regular, o Behat pode gerar por você, se você implementar a interface `Behat\Behat\Context\CustomSnippetAcceptingContext` e adicionar o método `getAcceptedSnippetType()` irá retornar a string `"regex"`:

```
public static function getAcceptedSnippetType()
{
    return 'regex';
}
```

Tipos de resultado da execução da etapa

Agora você sabe como mapear o código atual do PHP que vai ser executado. Mas como você pode falar exatamente o que "falhou" ou "passou" quando executou uma etapa? E como atualmente o Behat verifica se um passo é executado corretamente?

Para isto, nós temos tipos de execução de etapa . O Behat diferencia sete tipos de resultados de execuções de etapa: "Successful Steps", "**'Undefined Steps'_**", "**'Pending Steps'_**", "**'Failed Steps'_**", "**'Skipped Steps'_**", "**'Ambiguous Steps'_**" e "**'Redundant Step Definitions'_**".

Vamos usar nossa funcionalidade introduzida anteriormente para todos os exemplos a seguir:

```
# features/exemplo.feature
Funcionalidade:
  Cenário:
    Dado alguma etapa com um argumento "string"
    E uma etapa com número 23
```

Successful Steps

Quando o Behat encontra uma step definition correspondente ele vai executá-la. Se o método definido **não** joga nenhuma `Exceção`, a etapa é marcada como bem sucedida (verde). O que você retornar de um método de definição não tem efeito sobre o status de aprovação ou reprovação do próprio.

Vamos simular que nossa classe contexto contenha o código abaixo:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
    /** @Given alguma etapa com um argumento :argumento1 */
    public function algumaEtapaComUmArgumento($argumento1)
```

```
    {
    }

    /** @Given uma etapa com numero :argumento1 */
    public function umaEtapaComNumero($argumento1)
    {
    }
}
```

Quando você executar sua funcionalidade, você vai ver todas as etapas passadas serem marcadas de verde. Isso simplesmente porque não foram lançadas exceções durante a sua execução.

Note: Etapas passadas sempre são marcadas de **verde** se o seu console suportar cores.

Tip: Habilite a extensão PHP "posix" para ver a saída colorida do Behat. Dependendo do seu Linux, Mac OS ou outro sistema Unix pode fazer parte da instalação padrão do PHP ou um pacote `php5-posix` a parte.

Etapas Indefinidas

Quando o Behat não pode achar uma definição correspondente, a etapa é marcada como **indefinida**, e todas as etapas subsequentes do cenário são **ignoradas**.

Vamos supor que temos uma classe contexto vazia:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
}
```

Quando você executar a sua funcionalidade, você terá 2 etapas indefinidas marcadas de amarelo.

Note: Etapas indefinidas sempre são marcadas de **amarelo** se o seu console suportar cores.

Note: Todas as etapas seguintes de uma etapa indefinida não são executadas, como o seguinte comportamento é imprevisível. Estas etapas são marcadas como **ignoradas** (ciano).

Tip: Se você usar a opção `--strict` com o Behat, etapas não definidas vão fazer o Behat imprimir o código `1`.

Etapas Pendentes

Quando uma definição de um método lança uma exceção `Behat\Behat\Tester\Exception\PendingException`, a etapa é marcada como **pendente**, lembrando que você tem trabalho a fazer.

Vamos supor que sua `FeatureContext` se pareça com isto:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;
use Behat\Behat\Tester\Exception\PendingException;
```

```
class FeatureContext implements Context
{
    /** @Given alguma etapa com um argumento :argumento1 */
    public function algumaEtapaComUmArgumento($argumento1)
    {
        throw new PendingException('Fazer algum trabalho de string');
    }

    /** @Given uma etapa com numero :argumento1 */
    public function umaEtapaComNumero($argumento1)
    {
        throw new PendingException('Fazer algum trabalho de numero');
    }
}
```

Quando você executar sua funcionalidade, você terá 1 etapa pendente marcada de amarelo e uma etapa seguinte que é marcada de ciano.

Note: Etapas pendentes sempre são marcadas de **amarelo** se o seu console suportar cores, porque são logicamente semelhante aos passos **indefinidos**

Note: Todas as etapas seguintes a uma etapa pendente não são executadas, como o comportamento seguinte é imprevisível. Essas etapas são marcadas como **ignoradas**

Tip: Se você usar a opção `--strict` com o Behat, etapas pendentes vão fazer o Behat imprimir o código `1`.

Etapas Falhas

Quando uma definição de um método lança uma `Exceção` (exceto `PendingException`) durante a execução, a etapa é marcada como **falha**. Novamente, o que você retornar de uma definição não afeta a passagem ou falha da etapa. Retornando `null` ou `false` não vai causar a falha da etapa.

Vamos supor, que sua `FeatureContext` possua o seguinte código:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
    /** @Given alguma etapa com um argumento :argumento1 */
    public function algumaEtapaComUmArgumento($argumento1)
    {
        throw new Exception('alguma exceção');
    }

    /** @Given uma etapa com numero :argumento1 */
    public function umaEtapaComNumero($argumento1)
    {
    }
}
```

Quando você executar sua funcionalidade, você terá 1 etapa falha marcada de vermelho e será seguida de uma etapa ignorada marcada de ciano.

Note: Etapas falhas são marcadas de **vermelho** se o seu console suportar cores.

Note: Todas as etapas seguintes a uma etapa pendente não são executadas, como o comportamento seguinte é imprevisível. Essas etapas são marcadas como **ignoradas**

Tip: Se você usar a opção `--strict` com o Behat, etapas falhas vão fazer o Behat imprimir o código `1`.

Tip: O Behat não vem com uma ferramenta própria de asserção, mas você pode usar qualquer ferramenta de asserção externa. Uma ferramenta própria para asserção é uma biblioteca, na qual asserções lancem exceções em caso de falha. Por exemplo, se você está familiarizado com o PHPUnit, você pode utilizar suas asserções no Behat o instalando via composer:

```
$ php composer.phar require --dev phpunit/phpunit='~4.1.0'
```

e então simplesmente utilizar asserções em suas etapas:

```
PHPUnit_Framework_Assert::assertCount(valorInteiro($contador), $this->cesta);
```

Tip: Você pode ter uma exceção stack trace com a opção `-vv` fornecido pelo Behat:

```
$ vendor/bin/behat features/exemplo.feature -vv
```

Etapas Ignoradas

Etapas que seguem etapas **indefinidas**, **pendentes** ou **falhas** nunca são executadas, mesmo que tenham correspondência definida. Essas etapas são marcadas como **ignoradas**:

Note: Etapas ignoradas são marcadas de **ciano** se o seu console suportar cores.

Etapas Ambíguas

Quando o Behat encontra duas ou mais definições correspondentes a uma única etapa, esta etapa é marcada como **ambígua**.

Considere que sua `FeatureContext` tenha o seguinte código:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{

    /** @Given /^.* etapa com .*$/ */
    public function algumaEtapaComUmArgumento()
    {
    }

    /** @Given /^uma etapa com .*$/ */
    public function umaEtapaComNumero($argument1)
    {
```

```
    }
}
```

A execução do Behat com este contexto da funcionalidade irá resultar no lançamento de uma exceção `Ambígua`.

O Behat não vai tomar uma decisão sobre qual definição irá executar. Este é o seu trabalho! Mas como você pode ver, o Behat vai fornecer informações para ajudar você a eliminar o tais problemas.

Definições de etapa Redundante

O Behat não vai deixar você definir uma expressão de etapa correspondente a um padrão mais de uma vez. Por exemplo, olhe para dois padrões definidos `@Given` em seu contexto de funcionalidade:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
    /** @Given /^etapa com numero (\d+)$/ */
    public function trabalhandoComUmNumero($numero1)
    {
    }

    /** @Given /^etapa com numero (\d+)$/ */
    public function trabalhandoComUmNumeroDiferente($numero1)
    {
    }
}
```

Executando o Behat com este contexto de funcionalidade irá resultar no lançamento de uma exceção `Redundante`.

Transformações da Etapa Argumento

Transformações da etapa argumento permite você abstrair operações comuns executadas em argumentos no método de definição da etapa, em um dado mais específico ou em um objeto.

Cada método de transformação deve retornar um valor novo. Este valor, em seguida, substitui o valor original da string que estava sendo utilizado como um argumento para um método de definição da etapa.

Métodos de transformação são definidos utilizando o mesmo estilo de anotação como métodos de definição, mas deve-se usar a palavra-chave `@Transform`, seguido de um padrão correspondente.

Como um exemplo básico, você pode automaticamente converter todos os argumentos numéricos para inteiro com o seguinte código na classe de contexto:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
    /**
     * @Transform /^(\d+)$/
     */
    public function converteStringParaNumero($string)
    {
```

```
        return intval($string);
    }

    /**
     * @Then um usuário :nome, deve ter :quantidade seguidores
     */
    public function confirmaUsuarioTemSeguidores($nome, $quantidade)
    {
        if ('inteiro' !== gettype($quantidade)) {
            throw new Exception('Um número inteiro é esperado');
        }
    }
}
```

Note: Assim como em definições de etapa, você também pode usar ambos os simples padrões e expressões regulares.

Vamos a uma etapa mais distante e criar um método de transformação que pegue um argumento string de entrada e retorne um objeto específico. No exemplo a seguir, nosso método de transformação vai passar um nome de usuário e o método vai criar e retornar um novo objeto `Usuario`:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
    /**
     * @Transform :usuario
     */
    public function converteNomeDeUsuarioEmUmUsuario($usuario)
    {
        return new Usuario($usuario);
    }

    /**
     * @Then um :usuario, deve ter :quantidade seguidores
     */
    public function confirmaUsuarioTemSeguidores(Usuario $usuario, $quantidade)
    {
        if ('integer' !== gettype($quantidade)) {
            throw new Exception('Um número inteiro é esperado');
        }
    }
}
```

Transformando Tabelas

Vamos supor que nós escrevemos a seguinte funcionalidade:

```
# features/table.feature
Funcionalidade: Usuários

  Cenário: Criando Usuários
    Dado os seguintes usuários:
      | nome          | seguidores |
      | everzet       | 147        |
```

```
      | avalanche123  | 142        |
      | kriswallsmith | 274        |
      | dgosantos89   | 962        |
```

E nossa classe `FeatureContext` parecida com esta:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;
use Behat\Gherkin\Node\TableNode;

class FeatureContext implements Context
{
    /**
     * @Given os seguintes usuários:
     */
    public function incluiUsuarios(TableNode $tabelaUsuarios)
    {
        $users = array();
        foreach ($tabelaUsuarios as $hashUsuario) {
            $usuario = new Usuario();
            $usuario->colocaNomeUsuario($hashUsuario['nome']);
            $usuario->colocaContadorDeSeguidores($hashUsuario['seguidores']);
            $usuarios[] = $usuario;
        }

        // fazer a mesma coisa com $usuarios
    }
}
```

Uma tabela como esta pode ser necessária em uma etapa que teste a criação dos próprios objetos `Usuario`, e mais tarde usada novamente para validar outras partes de nosso código que dependa de múltiplos objetos `Usuario` que já existam. Em ambos os casos, nosso método de transformação pode usar nossa tabela de nomes de usuários e quantidade de seguidores e construir os usuários fictícios. Ao usar um método de transformação nós eliminamos a necessidade de duplicar o código que cria nossos objetos `Usuario`, e ao invés disso podemos contar com o método de transformação em cada momento que esta funcionalidade for necessária.

Transformações também podem ser usadas com tabelas. Uma transformação de tabela é correspondida por vírgulas que delimitam a lista de cabeçalho das colunas prefixadas com `table:`:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;
use Behat\Gherkin\Node\TableNode;

class FeatureContext implements Context
{
    /**
     * @Transform table:nome,seguidores
     */
    public function converteTabelaUsuarios(TableNode $tabelaUsuarios)
    {
        $usuarios = array();
        foreach ($tabelaUsuarios->getHash() as $hashUsuario) {
            $usuario = new Usuario();
            $usuario->colocaNomeUsuario($hashUsuario['nome']);
            $usuario->colocaContadorDeSeguidores($hashUsuario['seguidores']);
            $usuarios[] = $usuario;
        }
```

```
        return $usuarios;
    }

    /**
     * @Given os seguintes usuários:
     */
    public function incluiUsuarios(array $usuarios)
    {
        // faça alguma coisa com $usuarios
    }

    /**
     * @Then eu espero que os seguintes usuários:
     */
    public function confirmaUsuarios(array $usuarios)
    {
        // faça alguma coisa com $usuarios
    }
}
```

Note: Transformações são poderosas e é importante ter cuidado como você as implementa. Um erro pode frequentemente introduzir um estranho e inesperado comportamento. Também, eles são por natureza difíceis de serem depurados devido a sua natureza altamente dinâmica.

Procure no seu dicionário de etapas

Tal como o seu conjunto de cenários irá crescer, há uma boa chance de que a quantidade de etapas diferentes que você terá à sua disposição para escrever novos cenários também irá crescer.

O Behat provem uma opção de linha de comando `--definitions` ou simplesmente `-d` para navegar facilmente nas definições, a fim de reutilizá-los ou adaptá-los (introdução de novos espaços reservados por exemplo).

Por exemplo, quando utilizamos o contexto Mink provido pela extensão Mink, você terá acesso a este dicionário de etapas executando:

```
$ behat -di
web_features | Given /^(?:|I )am on (?:|the )homepage$/
             | Opens homepage.
             | at `Behat\MinkExtension\Context\MinkContext::iAmOnHomepage()`

web_features | When /^(?:|I )go to (?:|the )homepage$/
             | Opens homepage.
             | at `Behat\MinkExtension\Context\MinkContext::iAmOnHomepage()`

web_features | Given /^(?:|I )am on "(?P<page>[^"]+)"$/
             | Opens specified page.
             | at `Behat\MinkExtension\Context\MinkContext::visit()`

# ...
```

ou, pela saída curta:

```
$ behat -dl
web_features | Given /^(?:|I )am on (?:|the )homepage$/
web_features |  When /^(?:|I )go to (?:|the )homepage$/
web_features | Given /^(?:|I )am on "(?P<page>[^"]+)"$/
```

```
web_features |  When /^(?:|I )go to "(?P<page>[^"]+)"$/
web_features |  When /^(?:|I )reload the page$/
web_features |  When /^(?:|I )move backward one page$/
web_features |  When /^(?:|I )move forward one page$/
# ...
```

Você também pode procurar por um padrão específico executando:

```
$ behat --definitions="field" (ou simplesmente behat -dfield)
web_features | When /^(?:|I )fill in "(?P<field>(?:[^"]|\\")*)" with "(?P<value>(?:[^"]|\\")*)"$/
             | Fills in form field with specified id|name|label|value.
             | at `Behat\MinkExtension\Context\MinkContext::fillField()`

web_features | When /^(?:|I )fill in "(?P<field>(?:[^"]|\\")*)" with:$/
             | Fills in form field with specified id|name|label|value.
             | at `Behat\MinkExtension\Context\MinkContext::fillField()`

#...
```

É isso aí, agora você pode procurar e navegar pelo seu dicionário de etapas inteiro.

Hooking no Processo de Teste

Você aprendeu *como escrever definições de etapas* e que com *Gherkin* você pode mover etapas comuns em um bloco de fundo, e fazer suas funcionalidades DRY - Don't repeat yourself (livre de ambiguidades). Mas e se isso não for suficiente? E se você precisar executar algum código antes de toda a suite de testes ou depois de um cenário específico? Hooks é a salvação:

```php
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;
use Behat\Testwork\Hook\Scope\BeforeSuiteScope;
use Behat\Behat\Hook\Scope\AfterScenarioScope;

class FeatureContext implements Context
{
    /**
     * @BeforeSuite
     */
     public static function preparar(BeforeSuiteScope $scope)
     {
         // preparar o sistema para a suite de testes
         // antes de executar isto
     }

     /**
      * @AfterScenario @database
      */
     public function limparDB(AfterScenarioScope $scope)
     {
         // limpa a database depois dos cenários,
         // marcados com @database
     }
}
```

Sistema de Hooks no Behat

O Behat fornece um número de pontos de hook que nos permitem executar a lógica arbitrária em vários pontos no ciclo de teste Behat. Hooks são muito parecidos com definições de etapas ou transformações - eles são métodos simples com anotações especiais dentro de suas classes de contexto. Não há associação entre o local onde o hook é definido e qual nó é executado, mas você pode usar tags ou hooks nomeados se você quiser um controle mais refinado.

Todos os hooks definidos serão executados sempre que ações relevantes ocorram. A árvore de ações deve ser algo como isto:

```
-- Suite #1
|   -- Funcionalidade #1
|   |   -- Cenário #1
|   |   |   -- Etapa #1
|   |   |   -- Etapa #2
|   |   -- Cenário #2
|   |       -- Etapa #1
|   |       -- Etapa #2
|   -- Funcionalidade #2
|       -- Cenário #1
|           -- Etapa #1
-- Suite #2
    -- Funcionalidade #1
        -- Cenário #1
            -- Etapa #1
```

Este é o ciclo básico do teste no Behat. Há muitas suites de testes, cada uma das quais tem muitas funcionalidades, que tem muitos cenários com muitas etapas. Note que quando o Behat realmente é executado, exemplos de esquemas de cenário são interpretados como cenários - ou seja, cada exemplo torna-se um cenário real nesta árvore ação.

Hooks

Os Hooks permitem você executar seu código customizado pouco antes ou pouco depois de cada uma destas ações. O Behat permite você utilizar os seguintes hooks:

1. O hook `BeforeSuite` é executado antes que qualquer funcionalidade em sua suite rode. Por exemplo, você pode usar isto para configurar o projeto que você está testando. Este hook recebe um argumento opcional com uma instância da classe `Behat\Testwork\Hook\Scope\BeforeSuiteScope`.
2. O hook `AfterSuite` é executado depois que todas as funcionalidades da sua suite tenham sido executadas. Este hook é útil para despejar ou imprimir algum tipo de estatística ou derrubar a sua aplicação após o teste. Este hook recebe um argumento opcional com uma instância da classe `Behat\Testwork\Hook\Scope\AfterSuiteScope`.
3. O hook `BeforeFeature` é executado antes que a funcionalidade seja executada. Este hook recebe um argumento opcional com uma instância da classe `Behat\Behat\Hook\Scope\BeforeFeatureScope`.
4. O hook `AfterFeature` é executado após o Behat finalizar a execução da funcionalidade. Este hook recebe um argumento opcional com uma instância da classe `Behat\Behat\Hook\Scope\AfterFeatureScope`.
5. O hook `BeforeScenario` é executado antes que um cenário específico seja executado. Este hook recebe um argumento opcional com uma instância da classe `Behat\Behat\Hook\Scope\BeforeScenarioScope`.
6. O hook `AfterScenario` é executado depois que o Behat termina a execução de um cenário. Este hook recebe um argumento com uma instância da classe `Behat\Behat\Hook\Scope\AfterScenarioScope`.
7. O hook `BeforeStep` é executado antes que uma etapa é executada. Este hook recebe um argumento opcional com uma instância da classe `Behat\Behat\Hook\Scope\BeforeStepScope`.

8. O hook `AfterStep` é executado depois que o Behat termina de executar uma etapa. Este hook recebe um argumento opcional com uma instância da classe `Behat\Behat\Hook\Scope\AfterStepScope`.

Você pode utilizar qualquer um destes hooks colocando como anotação em qualquer um dos seus métodos na classe contexto:

```
/**
 * @BeforeSuite
 */
public static function preparar($scope)
{
    // preparar o sistema para suite de testes
    // antes de executar isto
}
```

Nós utilizamos anotações como fizemos antes com *definitions*. Simplesmente utilizando a anotação do nome da hook que você deseja usar (por exemplo `@BeforeSuite`).

Hooks de Suite

Suite hooks são executadas fora do contexto do cenário. Isso significa que sua classe de contexto (por exemplo `FeatureContext`) ainda não foi instanciada e a única maneira que o Behat pode executar o código é através de chamadas estáticas. Este é o motivo das suite hooks precisarem ser definidas com métodos estáticos na classe de contexto:

```
use Behat\Testwork\Hook\Scope\BeforeSuiteScope;
use Behat\Testwork\Hook\Scope\AfterSuiteScope;

/** @BeforeSuite */
public static function configurar(BeforeSuiteScope $scope)
{
}

/** @AfterSuite */
public static function destruir(AfterSuiteScope $scope)
{
}
```

Aqui estão dois tipos de suite hook disponíveis:

- `@BeforeSuite` - executado antes de qualquer funcionalidade.
- `@AfterSuite` - executado após a execução de todas as funcionalidades.

Hooks de Funcionalidade

Como as hooks de suite, hooks de funcionalidade também são executadas fora do contexto de cenário. Então como uma hook de suite, sua hook de funcionalidade precisa ser definida como método estático em seu contexto:

```
use Behat\Behat\Hook\Scope\BeforeFeatureScope;
use Behat\Behat\Hook\Scope\AfterFeatureScope;

/** @BeforeFeature */
public static function configurarFuncionalidade(BeforeFeatureScope $scope)
{
}
```

```
/** @AfterFeature */
public static function destruirFuncionalidade(AfterFeatureScope $scope)
{
}
```

Aqui estão dois tipos de hook de funcionalidade disponíveis:

- `@BeforeFeature` - é executado antes de cada funcionalidade na suite.
- `@AfterFeature` - é executado depois de cada funcionalidade na suite.

Hooks de Cenário

Hooks de cenário são disparadas antes ou depois que cada cenário é executado. Estes hooks são executados dentro da inicialização da instância do contexto, assim eles não só poderiam ser uma instância simples de métodos de contexto, eles também terão acesso a qualquer propriedade do objeto que você definiu durante o seu cenário:

```
use Behat\Behat\Hook\Scope\BeforeScenarioScope;
use Behat\Behat\Hook\Scope\AfterScenarioScope;

/** @BeforeScenario */
public function antes(BeforeScenarioScope $scope)
{
}

/** @AfterScenario */
public function depois(AfterScenarioScope $scope)
{
}
```

Aqui estão dois tipos de hook de cenário disponíveis:

- `@BeforeScenario` - executado antes da execução de todos os cenário em cada funcionalidade.
- `@AfterScenario` - executado após a execução de todos os cenário em cada funcionalidade.

Agora, a parte interessante:

O hook `@BeforeScenario` é executada não só antes de cada cenário em cada funcionalidade, mas antes de **cada linha de exemplo** no esquemas do cenário. Sim, cada linha de exemplo do esquema do cenário trabalha quase do mesmo modo que um cenário comum.

`@AfterScenario` funciona exatamente do mesmo modo, também sendo executado após cenários habituais e exemplos de saída.

Hooks de Etapas

Hooks de etapas são disparadas antes ou depois que cada etapa é executada. Estes hooks são executados dentro da inicialização da instância do contexto, por isso eles são métodos de instância do mesmo modo que os hooks de cenário são:

```
use Behat\Behat\Hook\Scope\BeforeStepScope;
use Behat\Behat\Hook\Scope\AfterStepScope;

/** @BeforeStep */
public function antesDaEtapa(BeforeStepScope $scope)
{
}
```

```
/** @AfterStep */
public function depoisDaEtapa(AfterStepScope $scope)
{
}
```

Aqui estão dois tipos disponíveis de hook de etapa:

- `@BeforeStep` - executado antes de cada etapa em cada cenário.
- `@AfterStep` - executado depois de cada etapa em cada cenário.

Hooks Tagueadas

Talvez as vezes você queira executar somente um certo hook para certos cenários, funcionalidades ou etapas. Isto pode ser obtido através da associação da hook `@BeforeFeature`, `@AfterFeature`, `@BeforeScenario`, `@AfterScenario`, `@BeforeStep` ou `@AfterStep` com uma ou mais tags. Você pode também usar tags `OR` (`||`) e `AND` (`&&`):

```
/**
 * @BeforeScenario @database,@orm
 */
public function limparBancoDeDados()
{
    // limpar banco de dados antes
    // do cenario @database ou @orm
}
```

Utilize a tag `&&` para executar somente uma hook quando tem a tag *all*:

```
/**
 * @BeforeScenario @database&&@acessorios
 */
public function limparBancoDeDadosAcessorios()
{
    // limpar banco de dados acessorios
    // antes dos cenários @database @acessorios
}
```

Funcionalidades de teste

Nós já usamos esta estranha classe `FeatureContext` como uma casa para nossa *definição de etapas* e *hooks*, mas nós não temos feito muito para explicar o que realmente é.

Classes de contexto são uma pedra angular do meio ambiente de testes em Behat. A classe de contexto é uma simples POPO - Plain Old PHP Object - que diz ao Behat como testar as suas funcionalidades. Se todos arquivos `*.feature` descrevem *como* sua aplicação se comporta, então a classe de contexto é sobre como testar isso.

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
    public function __construct($parametro)
    {
```

```
        // instanciar contexto
    }

    /** @BeforeFeature */
    public static function prepararParaAFuncionalidade()
    {
        // limpar o banco de dados ou fazer outras preparações
    }

    /** @Given nós temos algum contexto */
    public function prepararContexto()
    {
        // fazer algo
    }

    /** @When ocorrer um evento */
    public function fazerAlgumaAcao()
    {
        // fazer algo
    }

    /** @Then algo deve ser feito */
    public function checar()
    {
        // fazer algo
    }
}
```

Um mnemônico simples para classes de contexto é "testar funcionalidades *em um contexto*". Descrições de funcionalidades tendem ser muito alto nível. Isto significa que não há muito detalhe técnico exposto neles, então o caminho para você testar essa linda funcionalidade depende do contexto em que seu teste está contido. Isso é o que as classes de contexto são.

Requerimentos de Classe de Contexto

A fim de ser usado pelo Behat, sua classe de contexto deve seguir as seguintes regras:

1. A classe de contexto deve implementar a interface `Behat\Behat\Context\Context`.
2. A classe de contexto deve ser chamada `FeatureContext`. É uma simples convenção dentro da infraestrutura do Behat. `FeatureContext` é o nome da sua classe de contexto para uma suíte padrão. Isto pode ser facilmente alterado através da configuração da suite dentro de `behat.yml`.
3. A classe de contexto deve ser detectável e carregável pelo Behat. Isso significa que você deve de alguma forma dizer ao Behat sobre o arquivo de classe. O Behat vem com uma PSR-0 carregamento automático fora e o carregamento automático de diretório padrão é `features/bootstrap`. É por isso que é carregado o padrão `FeatureContext` tão fácil pelo Behat. Você pode colocar suas próprias classes sob `features/bootstrap` seguindo a convenção PSR-0 ou você pode até definir seu próprio arquivo auto carregável em `behat.yml`.

Note: `Behat\Behat\Context\SnippetAcceptingContext` e `Behat\Behat\Context\CustomSnippetAcceptin` são versões especiais da interface `Behat\Behat\Context\Context` que dizem ao Behat neste contexto, espera fragmentos a ser gerado por ele.

A forma mais fácil de começar a utilizar o Behat em seu projeto é chamar o `behat` com a opção `--init` dentro do diretório do seu projeto:

```
$ vendor/bin/behat --init
```

O Behat irá criar alguns diretórios e uma classe esqueleto `FeatureContext` dentro do seu projeto.

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

class FeatureContext implements SnippetAcceptingContext
{
    /**
     * Initializes context.
     */
    public function __construct()
    {
    }
}
```

Tempo de vida do Contexto

Sua classe de contexto é inicializada antes de cada cenário ser executado, e todos os cenários tem a sua própria instância do contexto. Isto significa 2 coisas:

1. Todos os cenários são isolados de cada um dos outros cenários de contexto. Você pode fazer quase qualquer coisa dentro da instância do seu cenário de contexto sem medo de afetar outros cenários - todos os cenários tem a sua própria instância do contexto.
2. Todas as etapas em um único cenário são executados dentro de uma instância de contexto comum. Isto significa que você pode colocar instâncias `privadas` variáveis dentro de sua etapa `@Given` e você será capaz de ler seus novos valores dentro de suas etapas `@When` e `@Then`.

Contextos Múltiplos

Em algum momento, manter tudo em uma única classe *step definitions* e *hooks* poderia se tornar muito difícil. Você poderia utilizar herança de classes e dividir as definições em múltiplas classes, mas fazer isto poderia tornar muito difícil de seguir o seu código e utilizá-lo.

À luz destas questões, o Behat provê um caminho mais flexível para ajudar a fazer um código mais estruturado, permitindo que você utilize múltiplos contextos em uma única suíte de teste.

A fim de personalizar a lista de contextos que sua suíte de teste requer, você precisa ajustar a configuração da suíte dentro de ‘‘ behat.yml‘‘:

```
# behat.yml

default:
    suites:
        default:
            contexts:
                - FeatureContext
                - SecondContext
                - ThirdContext
```

A primeira `default` nesta configuração é um nome do perfil. Nós vamos discutir em *profiles* um pouco mais tarde. Sobre o perfil específico, nós temos uma seção especial `suites`, que configura suítes dentro deste perfil. Nós vamos

falar sobre suítes de teste com mais detalhes no *próximo capítulo*, por hora basta ter em sua mente que uma suíte é uma forma de dizer ao Behat onde encontrar suas funcionalidades e como as testar. A parte interessante para nós agora é a seção `contextos` - esta é uma matriz de nomes de classes de contexto. O Behat utilizará as classes especificadas em seu contexto de funcionalidades. Isto significa que a cada vez que o Behat ver um cenário em sua suíte de testes, ele irá:

1. Pegar a lista de todas as classes de contexto da opção `contexts`.
2. Tentará inicializar todas estas classes de contexto em Objetos.
3. Buscará por *step definitions* e *hooks* em todos eles.

Note: Não se esqueça que cada uma destas classes de contexto deve seguir todos os requerimentos de uma classe de contexto. Especificamente - todos eles devem implementar a interface `Behat\Behat\Context\Context` e ser autocarregadas pelo Behat.

Basicamente, todos os contextos sob a seção `contexts` em seu `behat.yml` são os mesmos para o Behat. Ele vai encontrar e utilizar os métodos da mesma forma que faz na `FeatureContext` padrão. E se você estiver feliz com uma única classe de contexto, mas você não gosta do nome `FeatureContext`, aqui está como você muda isto:

```
# behat.yml

default:
    suites:
        default:
            contexts:
                - MyAwesomeContext
```

Esta configuração irá dizer ao Behat para olhar para `MyAwesomeContext` ao invés da padrão `FeatureContext`.

Note: Ao contrário de *profiles*, o Behat não irá herdar qualquer configuração de sua suíte `default`. O nome `default` é utilizado somente para demonstração neste guia. Se você tem múltiplas suítes onde todas devem utilizar o mesmo contexto, você deverá definir este contexto específico para cada suíte específica:

```
# behat.yml

default:
    suites:
        default:
            contexts:
                - MyAwesomeContext
                - MyWickedContext
        suite_a:
            contexts:
                - MyAwesomeContext
                - MyWickedContext
        suite_b:
            contexts:
                - MyAwesomeContext
```

Esta configuração irá dizer ao Behat para olhar para `MyAwesomeContext` e `MyWickedContext` quando testar `suite_a` e `MyAwesomeContext` quando testar `suite_b`. Neste exemplo, `suite_b` não será capaz de chamar etapas que estão definidas em `MyWickedContext`. Como você pode ver, mesmo se você utilizar o nome `default` como o nome de uma suíte, o Behat não irá herdar qualquer configuração desta suíte.

Parâmetros de Contexto

Classes de contexto podem ser muito flexíveis, dependendo de quão longe você quer ir em fazê-las dinâmicas. A maioria de nós irá querer fazer nossas contextos ambiente-independente; onde nós colocaremos arquivos temporários, como URLs que serão utilizadas para acessar a aplicação? Estas são as opções de configuração de contexto altamente dependentes do ambiente que você irá testar as suas funcionalidades.

Já dissemos que classes de contexto são simplesmente velhas classes PHP. Como você incorporaria parâmetros ambiente-dependentes em sua classe PHP? Utilize *argumentos no construtor*:

```
// features/bootstrap/MyAwesomeContext.php

use Behat\Behat\Context\Context;

class MyAwesomeContext implements Context
{
    public function __construct($baseUrl, $tempPath)
    {
        $this->baseUrl = $baseUrl;
        $this->tempPath = $tempPath;
    }
}
```

Na realidade, o Behat lhe dá a habilidade de fazer exatamente isto. Você pode especificar argumentos requiridos para instanciar sua classe de contexto através de alguma configuração `contexts` em seu `behat.yml`:

```
# behat.yml

default:
    suites:
        default:
            contexts:
                - MyAwesomeContext:
                    - http://localhost:8080
                    - /var/tmp
```

Note: Nota sobre identação para parâmetros. É significativo:

```
contexts:
    - MyAwesomeContext:
        - http://localhost:8080
        - /var/tmp
```

Alinhado a quatro espaços da própria classe de contexto.

Argumentos seriam passados ao construtor `MyAwesomeContext` na ordem especificada aqui. Se você não está feliz com a ideia de manter uma ordem de argumentos em sua cabeça, você pode utilizar nomes de argumentos em vez disso:

```
# behat.yml

default:
    suites:
        default:
            contexts:
                - MyAwesomeContext:
                    baseUrl: http://localhost:8080
                    tempPath: /var/tmp
```

Na realidade, se você o fizer, a ordem em que você especificar estes argumentos se torna irrelevante:

```
# behat.yml

default:
    suites:
        default:
            contexts:
                - MyAwesomeContext:
                    tempPath: /var/tmp
                    baseUrl: http://localhost:8080
```

Levando isso um passo adiante, se os seus argumentos de construtor de contexto são opcionais:

```
public function __construct($baseUrl = 'http://localhost', $tempPath = '/var/tmp')
{
    $this->baseUrl = $baseUrl;
    $this->tempPath = $tempPath;
}
```

Você então pode especificar somente o parâmetro que você precisa mudar atualmente:

```
# behat.yml

default:
    suites:
        default:
            contexts:
                - MyAwesomeContext:
                    tempPath: /var/tmp
```

Neste caso, o valor padrão seria utilizado para outros parâmetros.

Traits de Contexto

O PHP 5.4 trouxe uma funcionalidade interessante para a linguagem - traits. Traits são um mecanismo para reutilização de código em linguagens de herança simples como o PHP. Traços são implementados em PHP como um `copia e cola` em tempo de compilação. Que significa se você colocar alguma definição de etapa ou hooks dentro de uma trait:

```
// features/bootstrap/DicionarioDeProdutos.php

trait DicionarioDeProdutos
{
    /**
     * @Given que tenha uma(s) :arg1, que custe R$:arg2
     */
    public function queTenhaUmaQueCusteR($arg1, $arg2)
    {
        throw new PendingException();
    }
}
```

E então utilize isto em seu contexto:

```
// features/bootstrap/MyAwesomeContext.php

use Behat\Behat\Context\Context;
```

```
class MyAwesomeContext implements Context
{
    use DicionarioDeProdutos;
}
```

Ele só funcionará como você espera que ele funcione.

Traits de contexto vem a calhar se você gostaria de ter contextos diferentes, mas ainda precisa utilizar a mesma etapa de definições em ambos. Ao invés de terem o mesmo código em ambos - você deve criar uma única Trait que você `use` em ambas classes de contexto.

Note: Dado que a etapa de definições *não pode ser duplicada dentro de uma Suíte*, isso só vai funcionar para contextos utilizados em suítes separadas.

Em outras palavras, se a sua Suíte utiliza no mínimo dois Contextos diferentes, estas classes de contexto `use` a mesma Trait, isto irá resultar em uma definição de etapa duplicada e o behat irá queixar-se lançando uma exceção `Redundant`.

Configurando Suíte de Testes

Nós já falamos sobre a configuração de contextos múltiplos para uma única suíte de testes em um *capítulo anterior*. Agora é a hora de falarmos sobre suíte de testes mesmo. Uma suíte de teste representa um grupo de funcionalidades concretas juntas com a informação de como as testar.

Com suítes você pode configurar o Behat para testar diferentes tipos de funcionalidades utilizando diferentes tipos de contextos e fazendo-o em uma única execução. Suítes de Testes são realmente poderosas e o `behat.yml` faz delas muito mais poderosas:

```
# behat.yml

default:
    suites:
        principal_features:
            paths:    [ %paths.base%/features/principal ]
            contexts: [ PrincipalContext ]
        usario_features:
            paths:    [ %paths.base%/features/web ]
            filters:  { role: usuario }
            contexts: [ UsuarioContext ]
        administrador_features:
            paths:    [ %paths.base%/features/web ]
            filters:  { role: administrador }
            contexts: [ AdministradorContext ]
```

Caminhos de Suíte

Uma das configurações mais óbvias das suítes é a configuração de `caminhos`:

```
# behat.yml

default:
    suites:
        principal_features:
            paths:
```

```
            - %paths.base%/features
            - %paths.base%/test/features
            - %paths.base%/vendor/.../features
```

Como você pode imaginar, esta opção diz ao Behat onde é para buscar as funcionalidades de teste. Você poderia, por exemplo, dizer ao Behat para procurar no arquivo `features/web` por funcionalidades e testá-los com `WebContext`:

```
# behat.yml

default:
    suites:
        web_features:
            paths:    [ %paths.base%/features/web ]
            contexts: [ WebContext ]
```

Você então pode precisar também descrever alguma funcionalidade para uma API-específica em `features/api` e testá-las com um `ApiContext`. Fácil:

```
# behat.yml

default:
    suites:
        web_features:
            paths:    [ %paths.base%/features/web ]
            contexts: [ WebContext ]
        api_features:
            paths:    [ %paths.base%/features/api ]
            contexts: [ ApiContext ]
```

Isto fará com que o Behat:

1. Encontre todas as funcionalidades em `features/web` e testá-las usando sua `WebContext`.
2. Encontre todas as funcionalidades em `features/api` e testá-las usando sua `ApiContext`.

Note: `%paths.base%` é uma variável especial em `behat.yml` que se refere ao arquivo em que o `behat.yml` está armazenado.

As suítes Path-Based são um fácil modo de testar aplicações altamente modulares onde as funcionalidades são entregues por componentes altamente desacoplados. Com suítes você pode testar todos eles juntos.

Filtros de Suíte

Além de ser capaz de executar funcionalidades de diretórios diferentes, nós podemos executar cenários do mesmo diretório, mas filtrado por critério específico. O analisador do Gherkin vem empacotado com uma coleção de filtros legais tais como filtros de *tags* e *nome*. Você pode utilizar estes filtros ao executar funcionalidades com uma tag (ou nome) específica em contextos específicos:

```
# behat.yml

default:
    suites:
        web_features:
            paths:    [ %paths.base%/features ]
            contexts: [ WebContext ]
            filters:
```

```
            tags: @web
        api_features:
            paths:    [ %paths.base%/features ]
            contexts: [ ApiContext ]
            filters:
                tags: @api
```

Esta configuração irá dizer ao Behat para executar funcionalidades e cenários com a tag `@web` em `WebContext` e funcionalidades e cenários com a tag `@api` em `ApiContext`. Mesmo se todos eles estão armazenados no mesmo arquivo. Achou isso legal? Isso ficará ainda mais, por que o Gherkin 4+ (usado no Behat 3+) trouxe um filtro muito especial *role*. Que significa, que você agora pode ter uma boa suíte baseada em ator:

```
# behat.yml

default:
    suites:
        usuario_features:
            paths:    [ %paths.base%/features ]
            contexts: [ UsuarioContext ]
            filters:
                role: usuario
        administrador_features:
            paths:    [ %paths.base%/features ]
            contexts: [ AdministradorContext ]
            filters:
                role: administrador
```

Uma Função filtro olha para o bloco da descrição da funcionalidade:

```
Funcionalidade: Registrando usuários
  A fim de ajudar mais pessoas a utilizarem nosso sistema
  Como um administrador
  Eu preciso ser capaz de registrar mais usuários
```

Ele procura por um padrão `Como um...` e supõe um ator a partir dele. Ele então filtra funcionalidades que não tenham um ator no conjunto. No caso do nosso exemplo, isso basicamente significa que a funcionalidade descrita a partir da perspectiva do ator *usuário* irá ser testada em `UsarioContext` e funcionalidades descritas a partir da perspectiva do ator *administrador* serão testadas em `AdministradorContext`. mesmo se eles estiverem no mesmo arquivo.

Contextos de Suítes

São capazes de especificar um conjunto de funcionalidades com um conjunto de contextos para estas funcionalidades dentro da suíte tem um efeito colateral muito interessante. Você pode especificar as mesmas funcionalidades em duas suítes diferentes sendo testadas por contextos diferentes *ou* o mesmo contexto configurado diferentemente. Isto basicamente significa que você pode utilizar o mesmo subconjunto de funcionalidades para desenvolver diferentes camadas da sua aplicação com o Behat:

```
# behat.yml

default:
    suites:
        domain_features:
            paths:    [ %paths.base%/features ]
            contexts: [ DomainContext ]
        web_features:
            paths:    [ %paths.base%/features ]
```

```
            contexts: [ WebContext ]
            filters:
                tags: @web
```

Neste caso, o Behat irá primeiramente executar todas as funcionalidades de `features/` do arquivo `DomainContext` e em seguida somente aqueles com a tag `@web` em `WebContext`.

Executando Suítes

Por padrão, quando você executa o Behat ele irá executar todas as suítes registradas uma a uma. Se ao invés disso você quiser executar uma única suíte, utilize a opção `--suite`:

```
$ vendor/bin/behat --suite=web_features
```

Inicialização de Suíte

Suítes são a principal parte do Behat. Qualquer funcionalidade do Behat sabe sobre elas e pode lhe dar uma mão com elas. Por exemplo, se você definir suas suítes em `behat.yml` antes de executar `--init`, ele realmente irá criar os arquivos e suítes que você configurou, ao invés do padrão.

Configuração - `behat.yml`

O Behat tem um sistema de configuração muito poderoso baseado na configuração de arquivos `YAML` e perfis.

`behat.yml`

Todas as configurações acontecem dentro de um único arquivo de configuração no formato `YAML`. O Behat tenta carregar `behat.yml` ou `config/behat.yml` por padrão, ou você pode dizer ao Behat onde está o seu arquivo de configuração com a opção `--config`:

```
$ behat --config custom-config.yml
```

Todos os parâmetros de configuração neste arquivo são definidos sob um perfil com nome root (`default:` por exemplo). Um perfil é justamente um nome customizado que você pode usar para mudar rapidamente as configurações de teste utilizando a opção `--profile` quando executar a sua suíte de funcionalidades.

O perfil padrão sempre é `default`. Todos os outros perfis herdam parâmetros do perfil `default`. Se você só precisa de um perfil, defina todos os seus parâmetros sob o `default:` root:

```
# behat.yml
default:
    #...
```

Variável de Ambiente - BEHAT_PARAMS

Se você quiser definir configurações do Behat, utilize a variável de ambiente `BEHAT_PARAMS`:

```
export BEHAT_PARAMS='{"extensions" : {"Behat\\MinkExtension" : {"base_url" : "https://www.exemplo.cor
```

Você pode configurar qualquer valor para qualquer opção que esteja disponível no arquivo `behat.yml`. Basta fornecer opções no formato *JSON*. O Behat utilizará essa opção como padrão. Você sempre pode sobrepô-las com as configurações no arquivo `behat.yml` do projeto (este tem maior prioridade).

Tip: A fim de especificar um parâmetro em uma variável de desenvolvimento, o valor *não deve* existir em seu `behat.yml`

Tip: NOTA: No Behat 2.x esta variável estava no formato *URL*. Ela foi modificada para utilizar o formato *JSON*.

Filtros Globais

Enquanto é possível especificar filtros em uma parte da configuração da suíte, por vezes você irá querer excluir certos cenários através da suíte, com a opção de sobrepor os filtros pela linha de comando.

Isto é conseguido através da especificação de filtros na configuração do *gherkin*:

```
# behat.yml

default:
    gherkin:
        filters:
            tags: ~@fazendo
```

Nesta instância, cenários com a tag *@fazendo* serão ignorados a menos que o comando seja executado com um filtro personalizado, por exemplo:

```
vendor/bin/behat --tags=fazendo
```

Autoloading Personalizado

Algumas vezes você irá precisar colocar a sua pasta de `funcionalidades` em outro lugar que não seja o padrão (por exemplo `app/features`). Tudo que você precisa fazer é especificar o caminho que você precisa carregar automaticamente pelo `behat.yml`:

```
# behat.yml

default:
    autoload:
        '': %paths.base%/app/features/bootstrap
```

Se você deseja o namespace de suas funcionalidades (por exemplo: para ser compatível com a PSR-1) você irá precisar adicionar o namespace às classes e também dizer ao behat onde carregá-las. Aqui `contextos` são um array de classes:

```
# behat.yml

default:
    autoload:
        '': %paths.base%/app/features/bootstrap
    suites:
        default:
            contexts: [My\Application\Namespace\Bootstrap\FeatureContext]
```

Note: A utilização do `behat.yml` para auto carregar somente é permitida pela `PSR-0`. Você pode também utilizar

o `composer.json` para auto carregar, que será também permitido pela `PSR-4`

Formatadores

Formatadores padrão podem ser habilitados especificando-os no perfil.

```
# behat.yml

default:
    formatters:
        pretty: true
```

Extensões

Extensões podem ser configuradas como esta:

```
# behat.yml

default:
    extensions:
        Behat\MinkExtension:
            base_url: http://www.example.com
            selenium2: ~
```

Executando Testes

Opções de formato

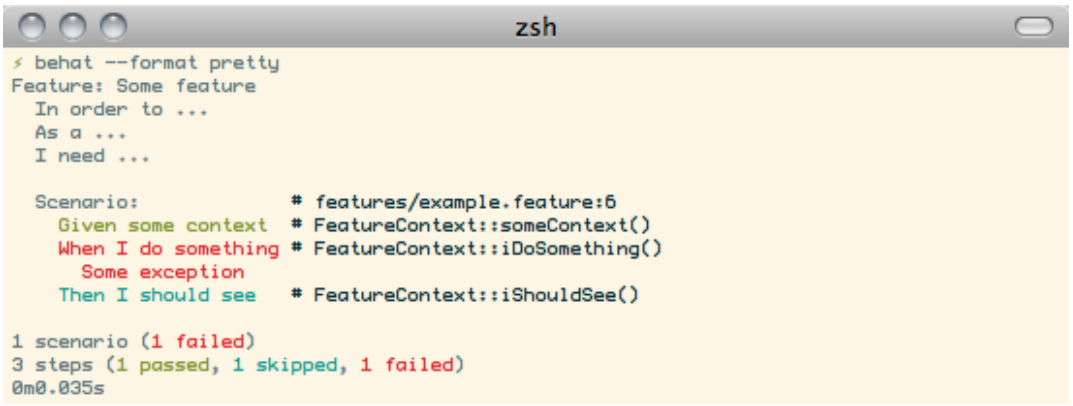
O Behat suporta diferentes formas de saídas de informações. As saídas no `behat` são chamadas *formatos* ou *formatadores*. Você pode dizer ao `behat` para executar com um formatador específico pelo fornecimento da opção `--format`:

```
$ behat --format progress
```

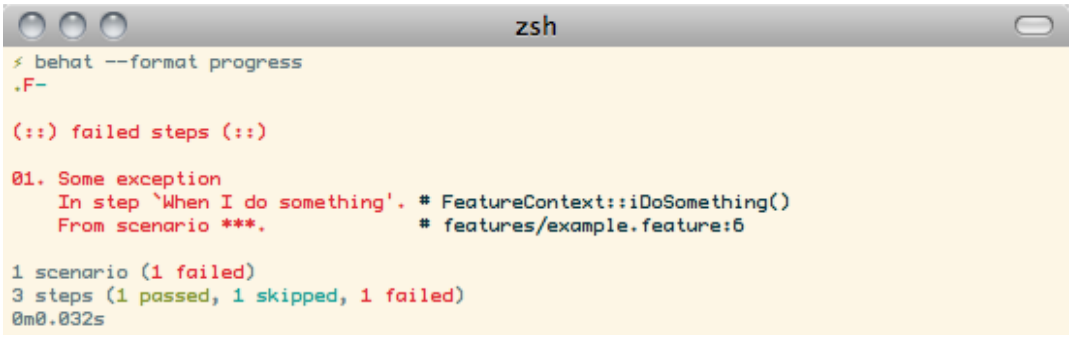
Note: O formatador padrão é o `pretty`.

O `behat` suporta 2 formatadores de saída:

- `pretty` - imprime a funcionalidade como é:



- `progress` - imprime um caractere por etapa:



Se você não quer imprimir a saída no console, você pode dizer ao `behat` para imprimir a saída em um arquivo ao invés de `STDOUT` com a opção `--out`:

```
$ behat --format pretty --out relatorio.txt
```

Note: Alguns formatadores, como `junit`, sempre requerem a opção especificar a opção `--out`. O formatador `junit` gera um arquivo `*.xml` para cada suíte, então precisa de um diretório de destino para colocar estes arquivos XML.

Você também pode especificar múltiplos formatos para serem usados pelo Behat utilizando múltiplas opções –format:

```
$ behat --format pretty --format progress
```

Neste caso, a saída padrão irá ser utilizada como saída para cada formato. Mas se você quiser que eles usem diferentes - especifique com `--out`:

```
$ behat -f pretty -o ~/pretty.out -f progress -o std -f junit -o xml
```

Neste caso, a saída do formatador pretty escreverá no arquivo `~/pretty.out`, a saída do formatador junit escreverá no arquivo `xml` e o formatador progress somente imprimira no console.

O Behat tentará identificar arduamente se o seu terminal suporta cores ou não, mas as vezes ele ainda falhará. Nestes casos, você pode forçar o `behat` a utilizar cores (ou não) com as opções `--colors` ou `--no-colors`, respectivamente:

```
$ behat --no-colors
```

CAPÍTULO 3

Cookbook

Aprenda soluções específicas para necessidades específicas:

Integrando Symfony2 com o Behat

Symfony2 é um Framework para Aplicações Web que pode ser facilmente integrado e perfeitamente utilizado com Behat 3. Como um pré-requisito para este livro de receitas você precisa ter a aplicação Symfony2 funcionando.

Neste livro de receitas nós cobriremos:

1. Instalando a dependência do Behat com Composer.
2. Inicializando uma suíte Behat.
3. Instalando e habilitando a extensão Symfony2.
4. Acesso aos serviços de aplicação em Contextos.
5. Utilizando o cliente de teste Symfony2 como um driver Mink.

Instalando o Behat em seu projeto Symfony2

A forma recomendada de gerir a dependência do Behat em seu projeto é usar o Composer. Presumindo que você já tenha o arquivo `composer.json` em seu projeto você precisa somente adicionar uma nova entrada para ele e instalar. Isto pode ser feito automaticamente para você com este comando:

```
$ php composer.phar require --dev behat/behat
```

Note: Note que nós utilizamos a switch `--dev` para o Composer. Isto significa que o Behat será instalado como uma dependência em seu projeto, e não estará presente em produção. Para mais informações, por favor verifique a documentação do Composer.

Inicializando o Behat

Depois de executar este comando você deve ver informações sobre arquivos inicializados em seu projeto, e você deve ser capaz de escrever o seu primeiro cenário. A fim de verificar a inicialização do Behat você pode simplesmente executar o seguinte comando:

```
$ bin/behat
```

Tip: Se você não está familiarizado o suficiente com o Behat, por favor leia *Construindo Modelo de Domínio* primeiro.

Instalando e habilitando a extensão Symfony2

Ótimo, você tem uma suite Behat trabalhando em seu projeto, agora é a hora de instalar a Extensão Symfony2. Para fazer isto você precisa adicionar outra dependência, mas da mesma forma que nós o fizemos há algum tempo atrás:

```
$ php composer.phar require --dev behat/symfony2-extension
```

Agora é a hora de habilitar a extensão em seu arquivo `behat.yml`. Se ele não existe, apenas crie este arquivo na raiz do seu projeto e preenchê-lo com o seguinte conteúdo:

```
default:
    extensions:
        Behat\Symfony2Extension: ~
```

Se este arquivo já existe, simplesmente mude o conteúdo adequadamente. A partir deste ponto você deve ser capaz de executar o Behat e a extensão Symfony2 será carregada e estará pronta para trabalhar.

Acessando serviços de aplicações em contextos

A extensão que acabou de ser instalada detecta a configuração padrão do Symfony e permite utilizar seu serviço de aplicação em classes de contexto. Para disponibilizar um serviço em uma contexto você precisa mudar sua configuração `behat.yml` e dizer a extensão que os serviços injetar:

```
default:
    suites:
        default:
            contexts:
                - FeatureContext:
                    session:   '@session'
    extensions:
        Behat\Symfony2Extension: ~
```

Está configuração irá tentar corresponder a `$session` dependência de seu construtor `FeatureContext` pela injeção do serviço `session` no contexto. Seja cuidadoso porque se tal serviço não existir ou seu nome não corresponder, ele não irá funcionar e você terminará com uma exceção.

Utilizando kernelDriver com sua suíte Behat

Symfony2 tem uma compilação interna de Cliente de Teste, que pode ajudar você com o teste de aceite web, porque não o utilizar? Especialmente devido ao Behat ter uma Extensão Mink que faz este tipo de teste ainda mais fácil.

A vantagem de utilizar o KernelDriver ao invés do Mink driver padrão é que você não precisa executar um servidor web a fim de acessar uma página. Você também sempre pode utilizar o Symfony Profiler e inspecionar sua aplicação diretamente. Você pode ler mais sobre cliente de teste na Documentação do Symfony.

Se você não tem o Mink e a MinkExtension ainda, você pode instalá-los de duas formas:

```
$ php composer.phar require --dev behat/mink
$ php composer.phar require --dev behat/mink-extension
```

Para instalar o Driver BrowserKit você precisa executar o seguinte comando:

```
$ php composer.phar require --dev behat/mink-browserkit-driver
```

Agora você está há somente uma etapa para estar pronto para fazer uso completo da extensão Symfony2 em seu projeto. Você precisa habilitar a extensão em seu arquivo `behat.yml` como a seguir:

```
default:
    extensions:
        Behat\Symfony2Extension: ~
        Behat\MinkExtension:
            sessions:
                default:
                    symfony2: ~
```

Et voilà! Agora você está pronto para você conduzir o seu desenvolvimento de aplicativo Symfony2 com o Behat3!

Acessando um contexto de outro

Quando dividimos as definições em múltiplos contextos, pode ser útil acessar um contexto apartir de outro. Isto é particularmente útil ao migrar do Behat 2.x para substituir sub contextos.

O Behat permite acessar o ambiente em *hooks*, então outros contextos podem ser recuperados utilizando a hook `BeforeScenario`:

```
use Behat\Behat\Context\Context;
use Behat\Behat\Hook\Scope\BeforeScenarioScope;

class FeatureContext implements Context
{
    /** @var \Behat\MinkExtension\Context\MinkContext */
    private $minkContext;

    /** @BeforeScenario */
    public function reunirContextos(BeforeScenarioScope $scope)
    {
        $environment = $scope->getEnvironment();

        $this->minkContext = $environment->getContext('Behat\MinkExtension\Context\MinkContext');
    }
}
```

Caution: Referências circulares em objetos de contexto impediriam a referência PHP contagem de contextos da recolha até o fim de cada cenário, forçando a aguardar o garbage collector ser executado. Isso aumentaria o uso de memória utilizada pela execução do Behat. Para prevenir isto, é melhor evitar o armazenamento do ambiente em suas classes de contexto. Também é melhor evitar a criação de referências circulares entre diferentes contextos.

CAPÍTULO 4

Mais sobre BDD

Quando estiver com o Behat instalado e funcionando, você pode aprender mais sobre o BDD através dos seguintes links (em inglês). Embora ambos os tutoriais sejão específicos do Cucumber, Behat Though both tutorials are specific to Cucumber, Behat compartilha muito com Cucumber e as filosofias são as mesmas.

- Dan North's "What's in a Story?"
- Cucumber's "Backgrounder"