Workshop – SALA A

Ballerina

Cloud Native Programming Language

Updated Nov, 2018

Sumário

Ballerina Introdução

Ballerina é uma linguagem de programação simples, cuja sintaxe e plataforma endereçam os difíceis problemas de integração. Ballerina é uma linguagem de programação de uso geral, compilada, concorrente, transacional, estática e fortemente tipada com sintaxes textual e gráficas.

Ela provê todas as funcionalidades esperadas de uma linguagem de programação moderna, porém foi desenhada especificamente para integração: traz conceitos fundamentais, ideias e ferramentas de integração de sistemas distribuídos na linguagem com suporte nativo para fornecer e consumir serviços de rede, transações distribuídas, mensagens confiáveis, processamento de streaming, segurança e workflows.

Ballerina foi inspirada em muitas tecnologias como: C, C++, Java, Go, Rust, Haskell, Kotlin, Dart, JavaScript, TypeScript, Flow, Swift, Relax NG, Perl e outras linguagens.

Pretende-se ser uma linguagem pragmática adequada para adoção comercial em massa; ela tenta se familiarizar com programadores que estão acostumados com as linguagens populares e modernas da família C, especialmente Java, C# e Java Script.

Demo 1 – Hello World init

Iniciando um projeto

Comece o seu projeto navegando para um diretório de sua escolha e execute o seguinte comando:

```
$ ballerina init
```

Você vai ver uma resposta dizendo que o seu projeto foi iniciado. Isto automaticamente cria um típico serviço "*Hello World*' para você no seu diretório. Um serviço Ballerina representa uma coleção de "*entry points*" de rede acessíveis. Um recurso dentro de um serviço representa um desses "*entry points*". O serviço de exemplo gerado expõe um "entry point" de rede na porta 9090.

Você pode executar este serviço rodando o comando:

```
$ ballerina run hello_service.bal
```

Você receberá a seguinte saída:

```
ballerina: initiating service(s) in 'hello_service.bal'

ballerina: started HTTP/WS endpoint 0.0.0.0:9090
```

Isto significa que o seu serviço está no ar e rodando. Você pode chamar o serviço usando um HTTP client. Neste caso usaremos o cURL.

```
$ curl http://localhost:9090/hello/sayHello
```

Você receberá a seguinte resposta:

```
Hello Ballerina!
```

Você acabou de iniciar com Ballerina, criou um projeto, iniciou um serviço, chamou o serviço criado e recebeu uma resposta.

Demo 2 – Hello World Code

Na aba de terminal do VS Code (ou em uma janela de terminal), dentro da pasta Demo, abra o demo.bal vazio que você criou:

```
$ code demo.bal
```

Este é o código que você precisa digitar (ou copiar de 1_demo_hello.bal) e explicações para cada linha:

```
// The simplest hello world REST API
// Ballerina has packages that can be imported
// This one adds services, endpoints, objects, and
// annotations for HTTP

import ballerina/http;

// Services, endpoint, resources are built into the language
// This one is HTTP (other options include WebSockets, Protobuf, gRPC, etc)
// We bind it to port 9090
service<http:Service> hello bind {port:9090} {

  // The service exposes one resource (hi)
  // It gets the endpoint that called it - so we can pass response back
  // and the request struct to extract payload, etc.
  hi (endpoint caller, http:Request request) {

      // Create the Response object
      http:Response res;
      // Put the data into it
      res.setPayload("Hello World!\n");
      // Send it back. -> means remote call (. means local)
      // _ means ignore the value that the call returns
      _ = caller->respond(res);
  }
}
```

No terminal do VS Code execute:

```
$ ballerina run demo.bal
ballerina: initiating service(s) in 'demo.bal'
ballerina: started HTTP/WS endpoint 0.0.0.0:9090
```

Agora você pode chamar o serviço na janela de terminal:

```
$ curl localhost:9090/hello/hi
Hello World!
```

Você agora tem um serviço Hello World REST executando e ouvindo na porta 9090.

Demo 3 – Annotations

Agora vamos arrumar as coisas:

Queremos que o serviço esteja apenas no caminho raiz / - por isso, vamos adicionar ServiceConfig para sobrescrever o caminho base padrão (que é o nome do serviço). Adicione a seguinte anotação antes do serviço:

```
@http:ServiceConfig {
   basePath: "/"
}
```

Disponibilize o recurso na raiz e mude os métodos para POST - vamos passar alguns parâmetros!

```
@http:ResourceConfig {
    methods: ["POST"],
    path: "/"
}
```

Na função hello, obtenha o payload como string (filtre possíveis erros):

```
 string payload = check req.getTextPayload();
```

Em seguida, adicione o nome na string de saída:

```
 response.setPayload("Hello " + untaint payload + "!\n");
```

Seu código final deve ser (veja os comentários para as novas linhas que você adicionar nesta fase):

```
// Add annotations for @ServiceConfig & @ResourceConfig
// to provide custom path and limit to POST
// Get payload from the POST request
// To run it:
// ballerina run demo.bal
// To invoke:
// curl -X POST -d "Demo" localhost:9090

import ballerina/http;

// Add this annotation to the service to change the base path
@http:ServiceConfig {
  basePath: "/"
}
```

```
service<http:Service> hello bind {port:9090} {

  // Add this annotation to the resource to change its path
  // and to limit the calls to POST only
  @http:ResourceConfig {
      path: "/",
      methods: ["POST"]
  }

  hi (endpoint caller, http:Request request) {

      // extract the payload from the request
      // getTextPayload actually returns a union of string | error
      // we will show how to handle the error later in the demo
      // for now, just use check that "removes" the error
      // (in reality, if there is error it will pass it up the caller stack)
      string payload = check request.getTextPayload();

      http:Response res;

      // use it in the response
      res.setPayload("Hello "+untaint payload+"!\n");

      _ = caller->respond(res);
  }
}
```

Execute-o novamente e invoque desta vez como POST:

```
$ curl -X POST -d "Ballerina" localhost:9090
Hello Ballerina!
```

Resumo:

- As anotações são nativas e integradas - não são add-on complementares, mas parte integrante da linguagem que permite que você ajuste significativamente o comportamento e forneça parâmetros.
- A linguagem completa ajuda você a lidar intuitivamente com qualquer transformação necessária (como manipular strings vazia em nosso caso),
- O Code completion e os tipos fortes ajudam você a localizar facilmente os métodos necessários (como o getTextPayload) e usá-los,
- Todo o poder do HTTP, REST, WebSockets, gRPC, etc. está ao seu alcance.
- As iterações de edição / execução / teste funcionam muito bem e nos mantêm produtivos.

Demo 4 – Twitter Connector (para casa)

Até agora, demonstramos a riqueza da criação de serviços da Web, mas não houve uma integração real. Estamos no espaço de integração há muito tempo e sabemos que os conectores são essenciais para a produtividade, permitindo que os desenvolvedores trabalhem com vários sistemas de maneira unificada, com uma curva de aprendizado mínima.

Ballerina Central é o lugar onde a comunidade do Ballerina está compartilhando esses conectores. WSO2 é um dos contribuintes. Vamos trabalhar com o Twitter com a ajuda do pacote WSO2/Twitter. Procure o pacote:

```
$ ballerina search twitter
```

Execute o Pull para obtermos o code completion (os pulls também acontecem automaticamente no build do aplicativo):

```
$ ballerina pull wso2/twitter
```

```
MacBook-Roberto:demo4-twitter rm$ ballerina search twitter

Ballerina Central
=================

|NAME            | DESCRIPTION                   | DATE           | VERSION |
|----------------| ------------------------------| ---------------| --------|
|wso2/twitter    | Connects to Twitter from Ba...| 2018-08-10-Sex | 0.9.18  |
|ketharan/twit...| Twitter integration using b...| 2018-07-03-Ter | 0.0.1   |

MacBook-Roberto:demo4-twitter rm$ ballerina pull wso2/twitter
wso2/twitter:0.9.18 [central.ballerina.io -> home repo]
```

https://apps.twitter.com/

Explain, that if you do not do this and just import the package in the code, this is also fine (will be pulled on the first compilation) but we want to do this before we do coding so we get the richness of the code completion.

Se você não fizer isso e apenas importar o pacote no código, isso também funciona (será puxado na primeira compilação), mas queremos fazer isso antes de codificarmos, para obtermos a riqueza do code completion.

No código, adicione:

```
import wso2/twitter;
```

Agora, vamos criar um endpoint do twitter e inicializá-lo. O Twitter requer credenciais OAuth que você pode obter de apps.twitter.com, mas colocá-las diretamente no seu código é uma má idéia (dica: ninguém as encontra no github!), Então preferimos lê-las a partir de um arquivo de configuração.

Importe a configuração para podermos ler o arquivo de configuração:

```
import ballerina/config;
```

Este código deve ficar logo abaixo da importação:

```
endpoint twitter:Client tw {
   clientId: config:getAsString("clientId"),
   clientSecret: config:getAsString("clientSecret"),
   accessToken: config:getAsString("accessToken"),
   accessTokenSecret: config:getAsString("accessTokenSecret"),
   clientConfig:{}
};
```

Agora temos o endpoint do twitter nas nossas mãos, vamos em frente e twittar!

Agora, podemos obter nossa resposta do Twitter apenas chamando seu método de tweet. A palavra-chave check significa - Eu entendo que isso pode retornar um erro. Eu não quero lidar com isso hear-pass. (para a função do caller, ou se esta é uma função de nível superior - gerar uma falha em tempo de execução).

```
twitter:Status st = check tw->tweet(payload);
```

Seu código agora ficará assim:

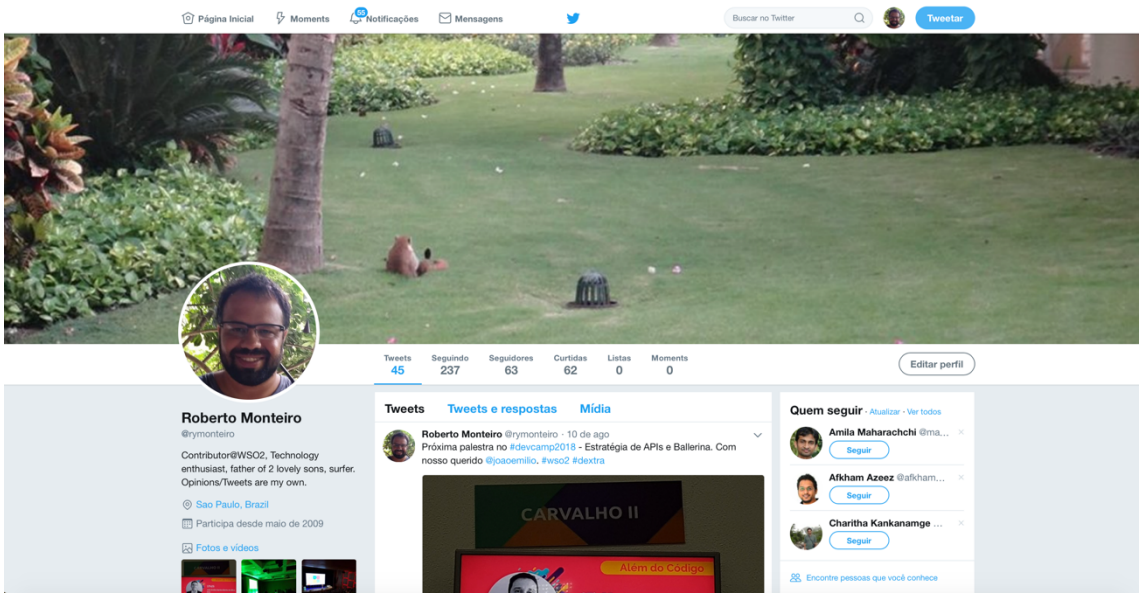
```
// Add twitter connector: import, create endpoint,
// create a new resource that invoke it
// To find it:
// ballerina search twitter
// To get it for tab completion:
// ballerina pull wso2/twitter
// To run it:
// ballerina run demo.bal --config twitter.toml
// To invoke:
// curl -X POST -d "Demo" localhost:9090

import ballerina/http;
```

```
// Pull and use wso2/twitter connector from http://central.ballerina.io
// It has the objects and APIs to make working with Twitter easy
import wso2/twitter;

// this package helps read config files
import ballerina/config;

// twitter package defines this type of endpoint
// that incorporates the twitter API.
// We need to initialize it with OAuth data from apps.twitter.com.
// Instead of providing this confidential data in the code
// we read it from a toml file
endpoint twitter:Client tw {
  clientId: config:getAsString("clientId"),
  clientSecret: config:getAsString("clientSecret"),
  accessToken: config:getAsString("accessToken"),
  accessTokenSecret: config:getAsString("accessTokenSecret"),
  clientConfig: {}
};

@http:ServiceConfig {
  basePath: "/"
}
service<http:Service> hello bind {port:9090} {
  @http:ResourceConfig {
      path: "/",
      methods: ["POST"]
  }
  hi (endpoint caller, http:Request request) {
      http:Response res;
      string payload = check request.getTextPayload();

      // Use the twitter connector to do the tweet
      twitter:Status st = check tw->tweet(payload);

      // Change the response back
      res.setPayload("Tweeted: " + st.text);

      _ = caller->respond(res);
  }
}
```

Vá em frente e execute-o e, desta vez, passe o arquivo de configuração:

```
$ ballerina run demo.bal --config twitter.toml
```

Verifique agora como está o seu twitter:



Agora vá para a janela do terminal e faça um tweet:

```
$ curl -X POST -d "Ballerina" localhost:9090
Tweeted: Ballerina
```

Vamos em frente e confira o seu feed do Twetter:

Muito legal. Em apenas algumas linhas de código, nossa integração com o Twitter começou a funcionar!

Agora vamos voltar ao código e torná-lo ainda mais legal adicionando lógica de transformação. Essa é uma tarefa muito frequente em apps de integração, pois o formato que seu backend expõe e retorna costuma ser diferente do que o aplicativo ou outros serviços precisam.

Vamos adicionar a lógica de transformação na ida para o serviço Twitter e na volta do serviço remoto para o caller.

Na ida para o Twitter, se a string não tiver a hashtag #ballerina, vamos adicioná-la, isso é tão fácil quanto:

```
if (!payload.contains("#ballerina")){payload=payload+" #ballerina";}
```

E obviamente faz sentido retornar não apenas uma string, mas um JSON significativo com o id do tweet, etc. Isso é fácil com o tipo nativo json do Ballerina:

```
      json myJson = {
          text: payload,
          id: st.id,
          agent: "ballerina"
      };

      res.setPayload(untaint myJson);
```

Vá em frente e execute:

```
$ ballerina run demo.bal --config twitter.toml
ballerina: initiating service(s) in 'demo.bal'
ballerina: started HTTP/WS endpoint 0.0.0.0:9090
```

Agora temos um JSON muito melhor:

```
$ curl -d "My new tweet" -X POST localhost:9090
{"text":"My new tweet
#ballerina","id":978399924428550145,"agent":"ballerina"}
```

Agora seu código será parecido com:

```
// Add transformation: #ballerina to input, and JSON to output
// To run it:
// ballerina run demo.bal --config twitter.toml
// To invoke:
// curl -X POST -d "Demo" localhost:9090

import ballerina/http;
import wso2/twitter;
import ballerina/config;

endpoint twitter:Client tw {
  clientId: config:getAsString("clientId"),
  clientSecret: config:getAsString("clientSecret"),
  accessToken: config:getAsString("accessToken"),
  accessTokenSecret: config:getAsString("accessTokenSecret"),
  clientConfig:{}
};

@http:ServiceConfig {
  basePath: "/"
}
service<http:Service> hello bind {port:9090} {

  @http:ResourceConfig {
      path: "/",
      methods: ["POST"]
  }
  hi (endpoint caller, http:Request request) {
      http:Response res;
      string payload = check request.getTextPayload();
```

```
        // transformation on the way to the twitter service - add hashtag
        if (!payload.contains("#ballerina")){payload=payload+" #ballerina";}

        twitter:Status st = check tw->tweet(payload);

        // transformation on the way out - generate a JSON and pass it back
        // note that json is a first-class citizen
        // and we can construct it from variables, data, fields
        json myJson = {
            text: payload,
            id: st.id,
            agent: "ballerina"
        };

        // pass back JSON instead of text
        res.setPayload(untaint myJson);

        _ = caller->respond(res);
    }
}
```

Resumo:

- Usando um conector fez uma interação com um serviço externo.
- O json nativo é superpoderoso – alguma outra linguagem que possua todos os formatos e protocolos modernos da web nativos? ;)
- A linguagem completa facilita muito o manuseio de ramificação lógica, transformação de dados e assim por diante.

Demo 5 – Docker/Kubernets (para casa)

Agora, que tipo de cloud-native programming language seria sem suporte nativo para as modernas plataformas de microsserviços? Ballerina tem suporte nativo embutido para docker e Kubernetes.

Vamos apenas adicionar algumas anotações e executá-lo no Kubernetes!

Primeiro, como de costume, adicione o pacote correspondente:

```
import ballerinax/kubernetes;
```

Adicione geração de artefatos do Kubernetes ao **serviço**:

```
@kubernetes:Deployment {
    image: "demo/ballerina-demo",
    name: "ballerina-demo"
}
```

Nota especial para usuários minikube no Ubuntu: Você precisa adicionar o caminho do host e certs do minikube na anotação "Deployment".

```
@kubernetes:Deployment {
    image: "demo/ballerina-demo",
    name: "ballerina-demo",
    dockerHost:"tcp://192.168.99.100:2376",
    dockerCertPath:"/home/chanaka/.minikube/certs"
}
```

Logo abaixo dessa anotação, adicione outra para passar a nossa configuração (aquela com as chaves do Twitter):

```
@kubernetes:ConfigMap{
    ballerinaConf:"twitter.toml"
}
```

Isso cria uma imagem docker e um deployment na qual ela é colocada.

E também precisamos criar um listener http e informar ao Kubernetes para expor externamente:

```
@kubernetes:Service {
```

```
  serviceType: "NodePort",
  name: "ballerina-demo"
}

endpoint http:Listener listener {
   port : 9090
};
```

Obviamente, o serviço agora precisa estar vinculado a esse listener e não apenas à declaração anônima inline:

```
service<http:Service> hello bind listener {
```

Seu código deve agora ser assim:

```
// Add kubernetes package and annotations
// To build kubernetes artifacts:
// ballerina build demo.bal
// To run it:
// kubectl apply -f kubernetes/
// To see the pod:
// kubectl get pods
// To see the service:
// kubectl get svc
// To invoke:
// curl -X POST -d "Hello from K8S" localhost:<put the port that you get from
kubectl get svc>
// To clean up:
// kubectl delete -f kubernetes/

import ballerina/http;
import wso2/twitter;
import ballerina/config;

// Add kubernetes package
import ballerinax/kubernetes;

endpoint twitter:Client tw {
  clientId: config:getAsString("clientId"),
  clientSecret: config:getAsString("clientSecret"),
  accessToken: config:getAsString("accessToken"),
  accessTokenSecret: config:getAsString("accessTokenSecret"),
  clientConfig:{}
};

// Now instead of inline {port:9090} bind we create a separate endpoint
// We need this so we can add Kubernetes notation to it and tell the compiler
// to generate a Kubernetes services (expose it to the outside world)
@kubernetes:Service {
  serviceType: "NodePort",
  name: "ballerina-demo"
}
endpoint http:Listener listener {
  port: 9090
};
```

```
// Instruct the compiler to generate Kubernetes deployment artifacts
// and a docker image out of this Ballerina service
@kubernetes:Deployment {
  image: "demo/ballerina-demo",
  name: "ballerina-demo"
}

// Pass our config file into the image
@kubernetes:ConfigMap{
  ballerinaConf: "twitter.toml"
}
@http:ServiceConfig {
 basePath: "/"
}
service<http:Service> hello bind listener {
  @http:ResourceConfig {
      path: "/",
      methods: ["POST"]
  }
  hi (endpoint caller, http:Request request) {
      http:Response res;
      string payload = check request.getTextPayload();
      if (!payload.contains("#ballerina")){payload=payload+" #ballerina";}

      twitter:Status st = check tw->tweet(payload);

      json myJson = {
          text: payload,
          id: st.id,
          agent: "ballerina"
      };

      res.setPayload(untaint myJson);
      _ = caller->respond(res);
  }
}
```

É isso aí - vamos em frente e construí-lo:

```
$ ballerina build demo.bal
Compiling source
    demo.bal

Generating executable
    ./target/demo.balx
        @kubernetes:Service                      - complete 1/1
        @kubernetes:ConfigMap                    - complete 1/1
        @kubernetes:Deployment                   - complete 1/1
        @kubernetes:Docker                       - complete 3/3

        Run following command to deploy kubernetes artifacts:
        kubectl apply -f /Users/anuruddha/workspace/ballerinax/ballerina-
demo/demo/target/kubernetes/demo
```

Você pode ver que criou uma pasta chamada kubernetes e colocou os artefatos de implantação e a imagem do docker lá:

```
$ tree
```

E você pode fazer o deploy no Kubernetes:

```
$ kubectl apply -f target/kubernetes/demo/
configmap "hello-ballerina-conf-config-map" created
deployment.extensions "ballerina-demo" created
service "ballerina-demo" created
```

Vamos ver se está funcionando:

```
$ kubectl get pods
NAME                              READY     STATUS    RESTARTS   AGE
ballerina-demo-74b6fb687c-mbrq2   1/1       Running   0          10s

$ kubectl get svc
NAME             TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)  AGE
ballerina-demo   NodePort    10.98.238.0   <none>        9090:**31977**/TCP
24s
kubernetes       ClusterIP   10.96.0.1     <none>        443/TCP  2d
```

Agora podemos ir em frente e invocar o serviço - estou usando o Kubernetes local, mas o mesmo comportamento aconteceria na nuvem. Usando a porta 31977 que o Kubernetes me deu:

```
$ curl -d "Tweet from Kubernetes" -X POST
http://localhost:**31977**
{"text":"Tweet from Kubernetes #ballerina",
"id":978399924428550145, "agent":"Ballerina"}
```

Se você está rodando no minikube com o Ubuntu, você precisa usar o ip do minikube para enviar o comando curl.

```
$ curl -d "Tweet from Kubernetes" -X POST  http://192.168.99.100:31977
{"text":"Tweet from Kubernetes #ballerina", "id":978399924428550145,
"agent":"Ballerina"}
```

Agora, remova a implantação do Kubernetes:

```
$ kubectl delete -f target/kubernetes/demo/
configmap "hello-ballerina-conf-config-map" deleted
deployment.extensions "ballerina-demo" deleted
service "ballerina-demo" deleted
```

Resumo:

- Com suporte nativo do Kubernetes e Docker, as imagens e os artefatos correspondentes simplesmente são gerados como parte do processo de criação!

Demo 6 – Circuit Breaking (para casa)

Para o restante da demonstração, faremos um copy and past das notas de slide para agilizar as coisas.

Para demonstrar uma lógica de integração mais avançada, vamos dar um passo além e usar uma API da web pública externa para integrar ao Twitter.

Que tal em vez de twittar nós mesmos, vamos apenas passar uma citação de Homer Simpson?

Nós vamos usar http://www.simpsonquotes.xyz/quote para fazer isso!

Para facilitar (e acelerar) a demonstração, baseamos as alterações no código pré-Kubernetes e tornamos as demonstrações locais (para reduzir o tempo de regeneração e reimplantação das imagens).

Adicionamos um endpoint para representar o serviço externo:

```
endpoint http:Client homer {
 url: "http://www.simpsonquotes.xyz"
};
```

E usamos esse endpoint (e não o payload) para obter o status do nosso tweet:

```
http:Response hResp = check homer->get("/quote");
string payload = check hResp.getTextPayload();
```

Seu código será agora parecido com:

```
// Add another external web service endpoint
// to compensate for slowness use circuit breaker
// To run it:
// ballerina run demo.bal --config twitter.toml
// To invoke:
// curl -X POST localhost:9090
// Invoke a few times to show that it is often slow

import ballerina/http;
import wso2/twitter;
import ballerina/config;

// create an endpoint for the external web service to use
endpoint http:Client homer {
 url: "http://www.simpsonquotes.xyz"
};

endpoint twitter:Client tw {
 clientId: config:getAsString("clientId"),
 clientSecret: config:getAsString("clientSecret"),
 accessToken: config:getAsString("accessToken"),
 accessTokenSecret: config:getAsString("accessTokenSecret"),
 clientConfig: {}
```

```
};

@http:ServiceConfig {
 basePath: "/"
}
service<http:Service> hello bind {port:9090} {

 @http:ResourceConfig {
     path: "/",
     methods: ["POST"]
 }
 hi (endpoint caller, http:Request request) {
     http:Response res;

// Call the remote service and get the payload from it and not the request
     http:Response hResp = check homer->get("/quote");
     string payload = check hResp.getTextPayload();

     if (!payload.contains("#ballerina")){payload=payload+" #ballerina";}

     twitter:Status st = check tw->tweet(payload);

     json myJson = {
         text: payload,
         id: st.id,
         agent: "ballerina"
     };

     res.setPayload(untaint myJson);
     _ = caller->respond(res);
 }
}
```

Agora podemos citar Homer no Twitter:

```
$ curl -X POST localhost:9090
{"text":"It’s not easy to juggle a pregnant wife and a troubled child, but
somehow I managed to fit in eight hours of TV a
day.","id":978405287928348672,"agent":"Ballerina"}

$ curl -X POST localhost:9090
{"text":"Just because I don’t care doesn’t mean that I don’t
understand.","id":978405308232957952,"agent":"Ballerina"}
```

Agora você pode perceber que metade das requisições estão demorando muito. Isso ocorre porque implementamos esse serviço remoto de citações simpson como não confiável e demoramos cerca de 5 segundos para responder na metade das invocações.

Isso simula uma situação típica quando você depende de um serviço externo, mas não é confiável.

Quando um serviço externo fica excessivamente ocupado e não responde, uma estratégia popular para lidar com isso é não esperar até que ele responda, mas apenas desconectar a chamada, fornecer tratamento padrão e suspender usando o endpoint até que ele se recupere.

Esse pattern é chamado de Circuit Breaker - e o Ballerina o construiu nativamente - não é necessário usar frameworks externos, servisse ashes, etc.:

- Se ocorrer uma determinada porcentagem de falhas, o sistema interromperá a chamada do endpoint e, em vez disso, chamará sua lógica "plano B" por algum tempo (permitindo que o endpoint se recupere caso esteja sobrecarregado).
- Existe um período de tempo durante o qual apenas a lógica do "Plano B" é usada.
- Após esse período de tempo, o Circuit Breaker envia uma nova solicitação ao endpoint remoto para verificar se recuperou ou não.

Esse serviço Simpson que usamos às vezes é muito lento. Vamos colocar um Circuit Breaker em torno dele.

Fazemos a inicialização do endpoint incluir lógica do circuit breaker:

```
endpoint http:Client homer {
 url: "http://www.simpsonquotes.xyz",
 circuitBreaker: {
     failureThreshold: 0,
     resetTimeMillis: 3000,
     statusCodes: [500, 501, 502]
 },
 timeoutMillis: 500
};
```

E o handler muda para:

```
var v = homer->get("/quote");

match v {
   http:Response hResp => {
       string payload = check hResp.getTextPayload();
       if (!payload.contains("#ballerina")){payload=payload+" #ballerina";}
       twitter:Status st = check tw->tweet(payload);

       json myJson = {
           text: payload,
           id: st.id,
           agent: "ballerina"
       };
       res.setPayload(myJson);
   }
   error err => {
       res.setPayload("Circuit is open. Invoking default behavior.\n");
   }
}
```

Observe como a correspondência nos fornece a capacidade de lidar com erros e definir o comportamento para o happy path e para a falha.

Além disso, no nosso caso, o caminho da falha será chamado não apenas quando a falha real ou o tempo limite de 0,5 segundo ocorrerem, mas também por 3 segundos depois disso - enquanto o Circuit Breaker estiver no estado Aberto.

Código Completo:

```
// To compensate for slowness use circuit breaker
// To run it:
// ballerina run demo_circuitbreaker.bal --config twitter.toml
// To invoke:
// curl -X POST localhost:9090
// Invoke many times to show how circuit breaker works

import ballerina/http;
import wso2/twitter;
import ballerina/config;

// Change the endpoint initialization to add timeout (half-send)
// and circuit breaker logic
endpoint http:Client homer {
 url: "http://www.simpsonquotes.xyz",
 circuitBreaker: {
     failureThreshold: 0,
     resetTimeMillis: 3000,
     statusCodes: [500, 501, 502]
 },
 timeoutMillis: 500
};

endpoint twitter:Client tw {
 clientId: config:getAsString("clientId"),
 clientSecret: config:getAsString("clientSecret"),
 accessToken: config:getAsString("accessToken"),
 accessTokenSecret: config:getAsString("accessTokenSecret"),
 clientConfig: {}
};

@http:ServiceConfig {
 basePath: "/"
}
service<http:Service> hello bind {port: 9090} {

 @http:ResourceConfig {
     path: "/",
     methods: ["POST"]
 }
 hi (endpoint caller, http:Request request) {
     http:Response res;

     // use var as a shorthand for http:Response | error union type
     // Compiler is smart enough to use the actual type
     var v = homer->get("/quote");

     // match is the way to provide different handling of error vs normal
output
     match v {
         http:Response hResp => {

             // if proper http response use our old code
             string payload = check hResp.getTextPayload();
             if (!payload.contains("#ballerina")){payload=payload+"
#ballerina";}
             twitter:Status st = check tw->tweet(payload);
```

```
            json myJson = {
                text: payload,
                id: st.id,
                agent: "ballerina"
            };
            res.setPayload(untaint myJson);
        }
        error err => {
            // this block gets invoked if there is error or if circuit
breaker is Open
            res.setPayload("Circuit is open. Invoking default behavior.\n");
        }
    }
    _ = caller->respond(res);
 }
}
```

Vamos em frente e executar. Observe como não apenas um erro/timeout leva ao nosso default handler ("O circuito está aberto. Invocando o comportamento padrão".) - mas como as próximas chamadas após ele (nos próximos 3 segundos no nosso caso) estão automaticamente usando esse caminho sem invocar o backend:

```
$ curl -X POST localhost:9090
{"text":"Marge, don't discourage the boy! Weaseling out of things is
important to learn. It's what separates us from the animals! Except the
weasel. #ballerina","id":986740441532936192,"agent":"ballerina"}

$ curl -X POST localhost:9090
Circuit is open. Invoking default behavior.

$ curl -X POST localhost:9090
Circuit is open. Invoking default behavior.

$ curl -X POST localhost:9090
{"text":"Marge, don't discourage the boy! Weaseling out of things is
important to learn. It's what separates us from the animals! Except the
weasel. #ballerina","id":986740441532936192,"agent":"ballerina"}
```

Resumo:

- Ballerina possui todos os padrões de integração e resiliência incorporados: message broker, chamadas assíncronas, transações distribuídas, automated retries e assim por diante.
- Usamos apenas um desses padrões - Circuit Breaker e isso foi necessário apenas adicionar algumas anotações.
- Ao contrário de frameworks e meshes, ter esses mecanismos embutidos é uma grande vantagem porque você pode - direto no código! - manipular adequadamente as falhas e fornecer a lógica necessária. Frameworks externos não têm visibilidade do que seu código está tentando realizar.

Demo 7 – Asynchronous Execution (para casa)

O Circuit-Breaker não é a única maneira de lidar com endpoints lentos ou não confiáveis. Ballerina também possui retries, lógica de compensação, transações distribuídas e execução assíncrona.

Nesse caso específico, o endpoint remoto costuma ser muito lento, então, por que não apenas chamá-lo de forma assíncrona e deixar a execução real demorar o tempo que for necessário.

Nós movemos toda a lógica para uma função (podemos deixá-la como estava ou simplificar removendo todo o tratamento de erro e construindo e passando o response - isso será uma chamada assíncrona para não nos importarmos com a resposta):

```
function doTweet() {
   http:Response hResp = check homer->get("/quote");
   string payload = check hResp.getTextPayload();
   if (!payload.contains("#ballerina")){ payload = payload+" #ballerina";}
   _ = tw->tweet(payload);
}
```

Nosso endpoint não precisa mais de circuit-breaker:

```
endpoint http:Client homer {
   url:"http://www.simpsonquotes.xyz"
};
```

E usamos a palavra-chave start para invocar a função em um thread separado de forma assíncrona:

```
hi (endpoint caller, http:Request request) {
     _ = start doTweet();
     http:Response res;
     res.setPayload("Async call\n");
     _ = caller->respond(res);
 }
```

Seu código completo agora se parece com:

```
// Move all the invocation and tweeting functionality to another function
// call it asynchronously
// To run it:
// ballerina run demo_async.bal --config twitter.toml
// To invoke:
// curl -X POST localhost:9090
// Invoke many times to show how quickly the function returns
// then go to the browser and refresh a few times to see how gradually new
tweets appear

import ballerina/http;
```

```
import wso2/twitter;
import ballerina/config;

endpoint twitter:Client tw {
 clientId: config:getAsString("clientId"),
 clientSecret: config:getAsString("clientSecret"),
 accessToken: config:getAsString("accessToken"),
 accessTokenSecret: config:getAsString("accessTokenSecret"),
 clientConfig: {}
};

endpoint http:Client homer {
 url: "http://www.simpsonquotes.xyz"
};

@http:ServiceConfig {
 basePath: "/"
}
service<http:Service> hello bind {port: 9090} {

 @http:ResourceConfig {
     path: "/",
     methods: ["POST"]
 }

 hi (endpoint caller, http:Request request) {

     // start is the keyword to make the call asynchronously
     _ = start doTweet();

     http:Response res;

     // just respond back with the text
     res.setPayload("Async call\n");

     _ = caller->respond(res);
 }
}

// Move the logic of getting the quote and pushing it to twitter
// into a separate function to be called asynchronously.
function doTweet() {
   // We can remove all the error handling here because we call
   // it asynchronously, don't want to get any output and
   // don't care if it takes too long or fails
   http:Response hResp = check homer->get("/quote");
   string payload = check hResp.getTextPayload();
   if (!payload.contains("#ballerina")){ payload = payload+" #ballerina"; }
   _ = tw->tweet(payload);
}
```

Agora podemos chamar rapidamente 10 vezes seguidas:

```
$ curl -X POST localhost:9090
Async call

$ curl -X POST localhost:9090
Async call

$ curl -X POST localhost:9090
Async call

$ curl -X POST localhost:9090
Async call

$ curl -X POST localhost:9090
Async call

$ curl -X POST localhost:9090
Async call

$ curl -X POST localhost:9090
Async call

$ curl -X POST localhost:9090
Async call
```

Agora vá para a janela do navegador do twitter, continue atualizando e vendo novos tweets ainda chegando.

Demo 8 – Swagger Export

API Management é uma parte importante das arquiteturas modernas e vem padronizado com Ballerina.

Inclui a capacidade de usar gateways de API externos ou microgateway com políticas de segurança, como basic authentication, OAuth e scopes.

Ele também entende Swagger - pode gerar serviços baseados em Swagger e exportar interfaces como Swagger.

Para gerar a definição de Swagger do nosso serviço, basta executar:

```
$ ballerina swagger export demo.bal
successfully generated swagger definition for input file - demo.bal

$ ls
ballerina-internal.log  demo.swagger.yaml
demo.bal                kubernetes
demo.balx               twitter.toml
```

Agora abra o yaml e observe o serviço e dois recursos:

```
$ code demo.swagger.yaml
```

Demo 9 – Sequence Diagrammatic

Os diagramas de sequência provaram ser a melhor maneira de documentar projetos de integração (lembre-se de como os utilizamos para ilustrar cada etapa dessa demonstração no slide deck?)

A sintaxe da Ballerina é projetada em torno de diagramas de sequência e, subsequentemente, a maneira como um desenvolvedor pensa ao escrever o código da Ballerina incentiva práticas recomendadas de interação.

Ballerina tooling está facilitando a compreensão dos projetos, gerando automaticamente diagramas de seqüência para o seu código. No VS Code, clique no botão do canto superior direito **Ballerina: Show Diagram** 

Aqui está o diagrama automatizado que Ballerina gerou para a função do Tweet que acabamos de usar: