

# Feuille de route pour créer un smart contract

Laurent Garnier

September 5, 2019

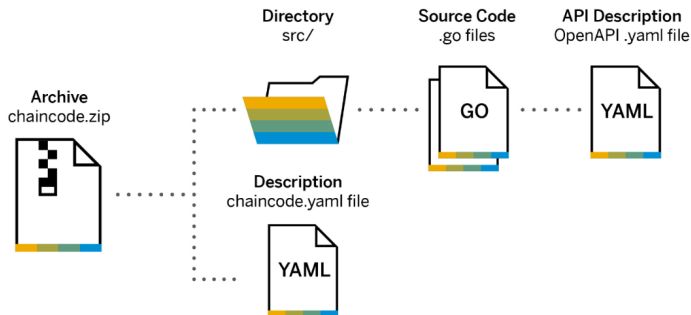
# Outline

# Structure générale du dossier

Dans tout dossier chaincode il doit y avoir :

- ▶ un fichier nommé `chaincode.yaml`
- ▶ un dossier `src`
- ▶ dans le dossier `src` il doit y avoir :
  1. Le code source `nom_du_fichier.go`
  2. L'API de description `openapi.yaml`

Voir la figure ci-dessous :



## Le fichier `chaincode.yaml`

Dans ce fichier nous fournirons les méta-données de notre chaincode.

Les balises `Id` et `Version` sont importantes ici.

Chaque fois qu'un chaincode est appelé, soit depuis une API REST soit depuis un autre chaincode, l'identifiant (ID) du chaincode doit être connu.

Cet ID doit aussi être fourni lorsque le chaincode est déployé.

Nous recommandons de spécifier tous les IDs de chaincode dans le format DNS inverse, comme pour les classes Java.

Par exemple, l'Id de notre chaincode est

`blockchain-example-chaincode_test`.

Cette syntaxe pour les IDs de chaincode : caractères alphanumérique et les tirets - et `_`.

De la même manière, chaque chaincode se voit attribué un numéro de version.

Puisqu'un chaincode est déployé "pour toujours" sur la blockchain et ne peut être effacé, pour remplacer un chaincode on ré-utilise son numéro de version.

# Code du fichier chaincode.yaml

```
# THIS SAMPLE CODE MAY BE USED SOLELY AS PART OF A TEST  
# BLOCKCHAIN SERVICE (THE "SERVICE") AND IN ACCORDANCE  
# WITH THE TERMS OF THE AGREEMENT FOR THE SERVICE.  
# THIS SAMPLE CODE PROVIDED "AS IS", WITHOUT ANY WARRANTY,  
# ESCROW, TRAINING, MAINTENANCE, OR SERVICE OBLIGATIONS  
# WHATSOEVER ON THE PART OF ALMERYYS/BE|YS.
```

```
Id:          blockchain-example-chaincode_test  
Version:    1
```

# Création du fichier de description API

ALMERYYS/BE|YS fournit une porte d'entrée qui expose toutes les fonctions chaincode comme des APIs REST normales.

Vous pouvez accomplir cela en fournissant une description OpenAPI de l'API REST qui est liée aux fonctions du chaincode. Vous faites cela avec le document YAML qui est écrit avec le chaincode en golang.

Le document YAML décrit exactement comment chaque fonction peut accéder via un appel REST, quels paramètres sont disponibles et comment les paramètres doivent être transmis au chaincode.

Dans le dossier src créer un fichier `hello_world.yaml` :

# Code du fichier hello\_world.yaml

```
swagger: "2.0"
info:
  description: |
    The Hello World! chain code shows the first steps
    in developing a chaincode that can read/write
    strings onto the blockchain and can expose these
    functions as REST API.
    THIS SAMPLE CODE MAY BE USED SOLELY AS PART OF
    THE TEST AND EVALUATION OF THE ALMERYYS/BE|YS
    BLOCKCHAIN SERVICE (THE "SERVICE") AND IN
    ACCORDANCE WITH THE AGREEMENT FOR THE SERVICE.
    THIS SAMPLE CODE PROVIDED "AS IS", WITHOUT ANY
    WARRANTY, ESCROW, TRAINING, MAINTENANCE, OR
    SERVICE OBLIGATIONS WHATSOEVER ON THE PART OF
    ALMERYYS/BE|YS.
  version: "1.0"
  title: "Hello World!"
```

# Création du fichier du code source (en Golang)

Lors du développement d'un chaincode, l'étape suivante est d'écrire le programme GO lui-même (ce qui est notre chaincode).

Dans le programme, il peut y avoir un nombre quelconque de fonctions, chaque fonction ayant un nombre quelconque de paramètres "non nommés".

L'appelant de toute fonction doit connaître le nom de la fonction et de la séquence exacte de paramètres.

Dans les configurations Hyperledger Fabric standards, l'accès au chaincode se fait uniquement via un SDK ce qui requiert un accès du chaincode au HTTPS/gRPC.



## Code du fichier hello\_world.go :

```
// DISCLAIMER:  
// THIS SAMPLE CODE MAY BE USED SOLELY AS PART OF THE TEST  
// AND EVALUATION OF THE ALMERYYS/BE|YS BLOCKCHAIN SERVICE  
// (THE "SERVICE") AND IN ACCORDANCE WITH THE TERMS OF THE  
// AGREEMENT FOR THE SERVICE. THIS SAMPLE CODE PROVIDED  
// "AS IS", WITHOUT ANY WARRANTY, ESCROW, TRAINING,  
// MAINTENANCE, OR SERVICE OBLIGATIONS WHATSOEVER ON THE  
// PART OF ALMERYYS/BE|YS.
```

# Comprendre le fichier de description API

Le fichier de description API `hello_world.yaml` est utilisé pour décrire l'interface HTTP exacte pour le chaincode.

C'est important de comprendre que le fichier `.yaml` est ensuite utilisé dans deux contextes différents :

- ▶ Pour générer la page de test API depuis laquelle les APIs de chaincode peuvent être testées directement
- ▶ La passerelle d'API utilise des aspects spécifiques du fichier YAML pour décider de la méthode d'extraction des paramètres de la requête HTTP entrante. Ensuite, ils associent ces éléments à une fonction du chaincode à appeler.

# Comprendre la correspondance verbes HTTP et appels de chaincode

Un ensemble riche de verbes HTTP est utilisé de manière spécifique pour que les appels REST imitent les opérations CRUD (Create Read Update Delete) typiques des bases de données.

Du côté Hyperledger Fabric, les fonctions chaincode peuvent être appelées comme suit :

- ▶ Par un appel `Invoke` qui écrit une transaction avec des ensembles lecture/écriture dans la blockchain
- ▶ Par un appel `Query` pour un type de fonction en lecture seule

Pour toutes les demandes HTTP entrantes, chaque verbe HTTP spécifique correspond à un appel `Invoke` ou `Query` de Hyperledger Fabric.

Dans cet exemple de chaincode, nous utiliserons des appels `POST` et `GET`.

# Tableau des correspondances HTTP/Chaincode

Verbe HTTP	Action type	Correspond à	Effet sur la Blockchain
POST	Créer	Invoke	Ecrit une transaction
GET	Lire	Query	Aucun

# Comprendre les chemins de chaincode

La section des chemins de chaincode est utilisée pour définir une définition enrichie de l'API basée sur REST et toutes les fonctionnalités de Swagger peuvent être utilisées pour décrire l'API. Deux cas spéciaux s'appliquent :

- ▶ Pour chaque chemin, vous devez préciser l'`operationId`. C'est le nom de direct de la fonction chaincode qui doit être appelée pour ce chemin.
- ▶ Pour les paramètres, les cinq emplacements de paramètres sont pris en charge. Les paramètres peuvent être `path`, `query`, `header`, `form`, ou `body`. Pour les types de paramètres, vous pouvez utiliser uniquement des paramètres simples qui peuvent être mis en correspondance avec des type de chaîne de caractères en entrée de chaincode. Les types acceptés sont : `string`, `number`, `integer`, `boolean`, et `file`.

# Définir le chemin du chaincode de POST

Pour définir le chemin d'accès au chaincode (utilisé pour appeler le chaincode), ouvrir le fichier `hello_world.yaml` avec un éditeur de texte et copiez-collez les lignes suivantes :

# Code du fichier hello\_world.yaml

```
consumes:
  - application/x-www-form-urlencoded

paths:

  /{id}:

    post:
      operationId: write
      summary: Write a text (once) by ID
      parameters:
        - name: id
          in: path
          required: true
          type: string
        - name: text
          in: formData
          required: true
          type: string
      responses:
        200:
          description: Text Written
        500:
          description: Failed
```

# Explications

Ce chemin de publication (POST) inclut la fonction `write` (`operationID: write`), deux paramètres pouvant être écrits (`id` et `text`) et deux codes de réponse (200 pour la réussite de la publication et 500 pour l'échec).

Notez que cet exemple inclut également la section `consumes`.

Ceci définit les types de contenu par défaut qui seront acceptés pour tous les appels d'API, s'ils ne sont pas définis spécifiquement.

Par défaut, il doit être défini sur

`application/x-www-form-urlencoded` pour signaler que les demandes HTTP entrantes auront des paramètres au format `nom / valeur`.



# Définir le chemin GET

Pour définir le chemin de chaincode GET (utilisé pour interroger le chaincode), copiez et collez les lignes suivantes sous le code précédent :

# Code du fichier hello\_world.yaml

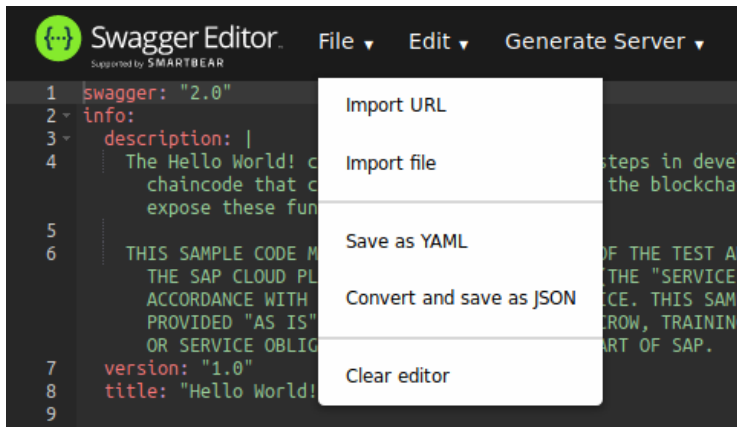
```
get:
  operationId: read
  summary: Read text by ID
  parameters:
    - name: id
      in: path
      required: true
      type: string
  produces:
    - text/plain
  responses:
    200:
      description: OK
    500:
      description: Failed
```

# Explications

Ce chemin pour GET comprend les fonctions de lecture (`operationID: read`), un paramètre à lire (`id`) et deux codes de réponse (200 pour une lecture réussie et 500 pour un échec de lecture).

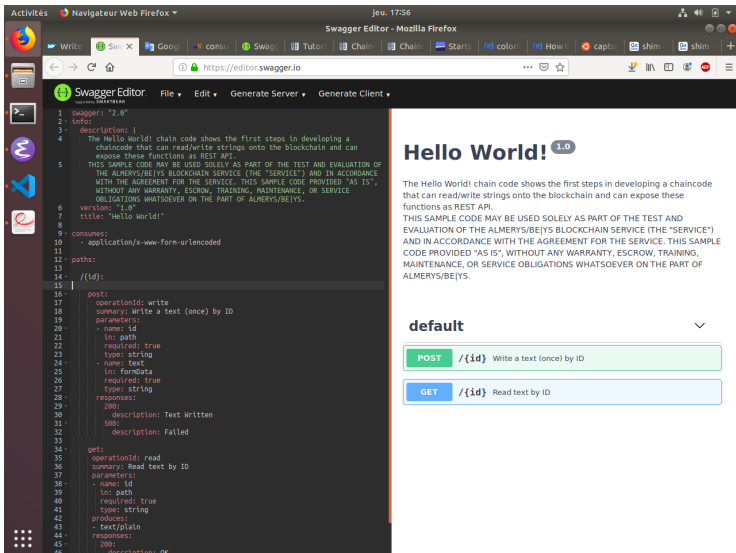
# Valider le `hello_world.yaml` avec Swagger.io

Ouvrir un navigateur web et naviguer jusqu'à l'éditeur Swagger.io  
Cliquer sur File > Clear Editor



Puis copier le code du fichier `hello_world.yaml` à l'intérieur

# Swagger complet



Swagger Editor - Mozilla Firefox

https://editor.swagger.io

## Hello World! <sup>1.0</sup>

The Hello World! chain code shows the first steps in developing a chaincode that can read/write strings onto the blockchain and can expose these functions as REST API.

THIS SAMPLE CODE MAY BE USED SOLELY AS PART OF THE TEST AND EVALUATION OF THE ALMERY/BEIYS BLOCKCHAIN SERVICE (THE "SERVICE") AND IN ACCORDANCE WITH THE AGREEMENT FOR THE SERVICE. THIS SAMPLE CODE PROVIDED "AS IS", WITHOUT ANY WARRANTY, ESCROW, TRAINING, MAINTENANCE, OR SERVICE OBLIGATIONS WHATSOEVER ON THE PART OF ALMERY/BEIYS.

**default**

**POST** /{id} Write a text (once) by ID

**GET** /{id} Read text by ID

```
1 swagger: "2.0"
2 info:
3   description: |
4     The Hello World! chain code shows the first steps in developing a
5     chaincode that can read/write strings onto the blockchain and can
6     expose these functions as REST API.
7     THIS SAMPLE CODE MAY BE USED SOLELY AS PART OF THE TEST AND EVALUATION OF
8     THE ALMERY/BEIYS BLOCKCHAIN SERVICE (THE "SERVICE") AND IN ACCORDANCE
9     WITH THE AGREEMENT FOR THE SERVICE. THIS SAMPLE CODE PROVIDED "AS IS",
10    WITHOUT ANY WARRANTY, ESCROW, TRAINING, MAINTENANCE, OR SERVICE
11    OBLIGATIONS WHATSOEVER ON THE PART OF ALMERY/BEIYS.
12  version: "1.0"
13  title: "Hello World!"
14  consumes:
15    - application/x-www-form-urlencoded
16  paths:
17    /{id}:
18      post:
19        operationId: write
20        summary: Write a text (once) by ID
21        parameters:
22          - name: id
23            in: path
24            required: true
25            type: string
26          - name: text
27            in: formData
28            required: true
29            type: string
30        responses:
31          200:
32            description: Text Written
33          500:
34            description: Failed
35      get:
36        operationId: read
37        summary: Read text by ID
38        parameters:
39          - name: id
40            in: path
41            required: true
42            type: string
43        produces:
44          - text/plain
45        responses:
46          200:
47            description: OK
```

# Comprendre les avantages des outils de développement de chaincode

Un chaincode est un programme développé en utilisant le langage de programmation GO. Pour le cycle de développement, il est plus efficace de pouvoir vérifier la syntaxe des programmes GO avant de les télécharger et de les déployer sur le noeud Hyperledger Fabric. Vous pouvez facilement le faire avec une installation minimale.

# Installer Git et Golang

## Télécharger

- ▶ Télécharger Git : <https://git-scm.com/downloads>
- ▶ Télécharger Go : <https://golang.org/dl/>

## Valider

Pour valider les installations, ouvrir un terminal et taper :

- ▶ `git`
- ▶ `go`

Dans les deux cas des documentations s'afficheront à l'écran

# Installer les paquets Hyperledger

Utilisons go et git :

```
go get github.com/hyperledger/fabric/common/util
```

```
go get github.com/hyperledger/fabric/core/chaincode/shim
```

```
go get github.com/hyperledger/fabric/protos/peer
```



## Création du package main

Le chaincode de Hyperledger Fabric est essentiellement un morceau de code exécuté sur un nœud homologue.

Lors de la création de ce code, vous utiliserez le package «main» pour en faire un programme exécutable.

Ce paquetage «main» indique au compilateur de Go, installé dans le tutoriel précédent, que le paquet doit être compilé en tant que programme exécutable au lieu d'une bibliothèque partagée (utilisée pour les morceaux de code réutilisables).

La fonction principale du paquet «main» sera le point d'entrée de notre programme exécutable.

Ceci est utilisé pour créer un binaire avec `go build`.

Ouvrez votre fichier `hello_world.go`, copiez et collez le texte ci-dessous sous l'avertissement :

```
package main
```

# Importer des packages

Dans Go, les fichiers source sont organisés dans des répertoires système appelés packages.

Ceux-ci permettent la réutilisation de code dans les applications Go. Le mot clé "import" est utilisé pour importer un package.

Nous pouvons télécharger et installer des packages Go tiers en utilisant la commande `go get`.

La commande `go get` va extraire les packages du référentiel source et les placer sur l'emplacement `GOPATH`.

Ouvrez votre fichier `hello_world.go`, copiez et collez ce qui suit sous le paquet principal :

# Code du fichier hello\_world.go

```
package main {  
    import (  
        "github.com/hyperledger/fabric/core/chaincode/shim"  
        "github.com/hyperledger/fabric/protos/peer"  
    )  
}
```

# Définir un chaincode

Lorsque le code de chaîne est démarré sur un nœud homologue, la fonction principale est exécutée.

Le chaincode est ensuite démarré avec `shim.Start`, qui attend une structure qui implémente l'interface `shim.Chaincode`.

Ouvrez votre fichier `hello_world.go`, copiez et collez le type et les fonctions principales sous les packages d'importation :

```
type Chaincode struct {  
}  
  
func main() {  
    shim.Start(new(Chaincode))  
}
```

# Implémentation d'une interface chaincode

Tout chaincode a besoin de l'interface suivante :

```
type Chaincode interface {  
    // Init is called during Instantiate transaction after  
    // the chaincode container  
    // has been established for the first time, allowing  
    // the chaincode to initialize its internal data  
    Init(stub ChaincodeStubInterface) peer.Response  
  
    // Invoke is called to update or query the ledger  
    // in a proposal transaction.  
    // Updated state variables are not committed to  
    // the ledger until the transaction is committed.  
    Invoke(stub ChaincodeStubInterface) peer.Response  
}
```

# Explications

Lorsque vous instanciez ou mettez à niveau un code de chaîne, la fonction `Init` de ce dernier est exécutée.

Vous pouvez l'utiliser pour initialiser des valeurs sur le grand livre. Lorsque vous appelez ou interrogez un code de chaîne, la fonction `Invoke` de ce dernier est exécutée.

Vous pouvez l'utiliser pour mettre à jour ou interroger le grand livre.

REMARQUE: vous trouverez des informations supplémentaires sur cette interface à l'adresse [GoDoc.org](http://GoDoc.org).

Les fonctions `Init` et `Invoke` sont toutes deux appelées à l'aide de l'interface `stub`, qui permet d'accéder aux paramètres de transaction et de modifier le grand livre.

Ouvrez votre fichier `hello_world.go`, copiez et collez les fonctions `init` et `invoke` sous les packages d'importation :

## Code à ajouter au fichier hello\_world.go

```
// Init is called during Instantiate transaction.
func (ptr *Chaincode) Init(stub shim.ChaincodeStubInterface) peer.Response {
    return shim.Success(nil)
}

// Invoke is called to update or query the ledger in a proposal transaction.
func (ptr *Chaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    return shim.Error("Not yet implemented!")
}
```

# Ajouter des fonctions au chaincode



Dans Hyperledger Fabric, nous devons faire la distinction entre les différents contextes dans lesquels le mot «Invoke» est utilisé. Dans Fabric, plus précisément dans le SDK utilisé pour accéder à toutes les fonctionnalités de Fabric, il existe le concept / accès de Invoke à Query. Un appel à la fonctionnalité Invoke au niveau du SDK Fabric est une opération d'écriture dans la blockchain sous-jacente. Après l'appel, toutes les mises à jour sont validées dans la blockchain. Cependant, un appel à la fonctionnalité Query est en réalité une opération de lecture sur la blockchain, qui ne renvoie que des informations.



## Exlications [1/2]

Étant donné les concepts Invoke et Query dans Fabric (au niveau du SDK), le monde HTTP REST peut être mis en correspondance sur ces deux appels. En réalité, toutes les requêtes HTTP GET sont considérées comme des opérations de lecture et sont mises en correspondance sur un appel Query dans le SDK de Fabric. Tous les appels HTTP POST et PUT sont considérés comme des opérations d'écriture et sont mis en correspondance sur l'appel Fabric Invoke. Toutefois, les concept Invoke et Query sur le SDK de Fabric ne correspondent pas de façon biunivoque dans le chaincode de Fabric.

## Explications [2/2]

Dans le chaincode, il n'y a qu'une seule fonction `Invoke` (implémentée dans le chaincode à l'aide du langage GO). Ainsi, dans le chaincode, il n'y a que des fonctionnalités de chaincode (également appelées fonctions), qui sont toutes accessibles via le point d'entrée central de la fonction `Invoke` (langage GO). Cette fonction GO `Invoke` ne sait en aucun cas si l'appel de l'extérieur était un `Invoke` (équivalent à écrire / valider) ou un `Query` (équivalent à une lecture). La fonction du chaincode doit simplement exécuter et renvoyer les réponses appropriées (correctes).

Lors de l'appel de chaincode via `invoke` ou `query`, la fonction `Invoke` du chaincode est exécutée.

Pour accéder au nom de la fonction et aux arguments de l'appel, vous pouvez utiliser le `stub.GetFunctionAndParameters`. Vous pouvez ensuite utiliser le nom de la fonction pour appeler différentes fonctions.

# Code de la fonction Invoke

```
func (ptr *Chaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {  
    function, args := stub.GetFunctionAndParameters()  
    switch function {  
        case "write":  
            return write(stub, args)  
        case "read":  
            return read(stub, args)  
        default:  
            return shim.Error("unknown function")  
    }  
}
```

## Explications [1/2]

Dans notre exemple de code de chaîne, nous souhaitons mettre à jour notre fonction Invoke afin de différencier les actions, telles qu'écrire et lire. Nous faisons cela en créant deux nouvelles fonctions, intitulées écrire et lire. La fonction d'écriture vous permet d'écrire quelque chose dans le grand livre, alors que la fonction de lecture extrait des données et en fournit l'état.

Une fois que la fonction GO de Invoke () a identifié la fonctionnalité de code de chaîne appelée (Lecture ou Ecriture), elle appelle la fonction GO correspondante, en fournissant en tant que paramètres l'interface de stub et le tableau complet de chaînes, qui sont des paramètres d'entrée pour la fonction de code chaîne.

L'interface de module de remplacement est une interface de rappel qui peut être utilisée dans le code de chaîne pour accéder aux fonctionnalités de Fabric. Elle est décrite en détail dans la documentation de Fabric.

## Explications [2/2]

Tous les paramètres de toute fonctionnalité de code de chaîne sont organisés en chaînes dans un tableau du code de chaîne. Il n'y a pas de contrôle sur le nombre de paramètres fournis, ni sur le type de chaque paramètre. Il appartient à la fonctionnalité de code de chaîne, en réalité à chaque fonction GO, de valider d'abord que le nombre et les types des paramètres d'entrée fournis correspondent, puis de décompresser et de convertir les chaînes en paramètres correspondants. Dans cet exemple, comme seuls les paramètres de chaîne sont utilisés, la seule vérification requise est de s'assurer que le nombre correct de paramètres a été fourni. De plus, tous les paramètres d'ID sont systématiquement remplacés par des minuscules, afin de garantir que la lecture et l'écriture fonctionneront toujours, indépendamment des chaînes d'identifiant fournies. (Notez que cela n'a pas à être fait, c'est simplement un exemple de la façon dont les paramètres peuvent être transformés avec le chaincode avant d'être écrits dans le blockchain.)

# Code de la fonction write

```
// Write text by ID
func write(stub shim.ChaincodeStubInterface, args []string) peer.Response {

    if len(args) != 2 || len(args[0]) < 3 || len(args[1]) == 0 {
        return shim.Error("Parameter Mismatch")
    }
    id := strings.ToLower(args[0])
    txt := args[1]

    if err := stub.PutState(id, []byte(txt)); err != nil {
        return shim.Error(err.Error())
    }

    return shim.Success(nil)
}
```

## Explications [1/3]

La fonction `write` prend deux paramètres, à savoir l'ID de la chaîne et la valeur de chaîne elle-même. Un appel peut par exemple être écrit ("TXT001", "Hello World!"). La première étape de la fonction `write` consiste à vérifier que deux paramètres (de type chaîne) sont fournis, que le premier paramètre (ID) comporte au moins 3 caractères et que le second paramètre (la valeur du texte) est fourni (longueur supérieure à zéro).

Une fois les paramètres validés, nous extrayons les deux paramètres du tableau de chaînes (`args []`) et les affectons à des variables GO locales. Habituellement, à cette étape, toutes les vérifications de type et les conversions supplémentaires sont effectuées, par exemple, la conversion de chaînes en entiers, etc., le cas échéant. Dans cet exemple, la seule manipulation de paramètre effectuée consiste à traduire l'ID en minuscule, afin de s'assurer qu'il est insensible à la casse.

## Explications [2/3]

Dans l'étape suivante, la chaîne de texte doit être écrite dans la chaîne de blocs. La façon dont cela fonctionne, est que toutes les informations écrites dans la chaîne de blocs sont regroupées dans un blob et stockées sous une clé à l'aide de la méthode `PutState` sur l'interface de stub. Dans les applications complexes, la clé sera indexée sur l'objet métier et la charge utile sera une chaîne sérialisée (JSON) pour l'objet. Dans cet exemple, nous utilisons simplement l'ID directement en tant que clé et la valeur de texte directement en tant que charge utile. Ces deux paramètres sont utilisés comme entrée dans `PutState`. Le seul aspect mineur est que cette méthode attend un tableau d'octets (`[] octets`) en tant que paramètre d'entrée. Nous effectuons donc un transtypage en langage GO uniquement pour convertir la chaîne en tableau d'octets.



## Explications [3/3]

Dans une dernière étape, nous retournons simplement un message de réussite.

Comment savons-nous / déterminons-nous que l'ID est le premier paramètre (args [0]) et que la valeur textuelle est le deuxième paramètre (args [1])?

Ceci est appliqué et documenté dans l'interface HTTP REST correspondante (fichier YAML) qui décrit l'interface vers cette fonction.

Là, deux paramètres ont été définis en tant qu'entrée dans l'appel HTTP REST / write et les paramètres sont transmis au code de chaîne dans exactement la séquence dans laquelle ils ont été définis dans le fichier YAML.

# Code de la fonction read

```
// Read text by ID
func read(stub shim.ChaincodeStubInterface, args []string) peer.Response {

    if len(args) != 1 {
        return shim.Error("Parameter Mismatch")
    }
    id := strings.ToLower(args[0])

    if value, err := stub.GetState(id); err == nil && value != nil {
        return shim.Success(value)
    }

    return shim.Error("Not Found")
}
```

## Explications [1/2]

La fonction de lecture est un peu plus simple.

Il ne prend qu'un seul paramètre en entrée (l'ID de la chaîne à lire) et renvoie la valeur textuelle de la chaîne.

Ce sera normalement appelé `text = read ("TXT001")`.

Dans le premier, on vérifie qu'un seul paramètre est fourni.

Notez que nous avons simplement ignoré la vérification de longueur ici, car nous allons valider l'ID implicitement un peu plus tard lors de la lecture de la blockchain.

## Explications [2/2]

Dans l'étape suivante, nous extrayons à nouveau l'ID en tant que variable GO du tableau de chaînes (args []).

La lecture de la blockchain est effectuée avec la méthode GetState. L'ID est fourni en tant que paramètre d'entrée et GetState renverra un tableau d'octets avec la valeur de blob correspondante dans la chaîne de blocs. Si nous réussissons (pas d'erreur et une valeur retournée), nous renverrons ce blob à l'appelant. Notez que l'appel Success prend également un tableau d'octets ([] byte) en tant que paramètre de retour, ce qui signifie que nous pouvons renvoyer directement l'objet blob récupéré, qui est notre valeur de texte. L'appelant (à l'extérieur) le convertira en une chaîne à renvoyer à l'appelant HTTP.

Si la lecture a échoué, par exemple parce que l'ID n'existe pas, un message d'erreur est renvoyé.

## Comprendre les exigences de packages [1/2]

Lorsque le chaincode Hyperledger Fabric est prêt à être déployé, il doit être emballé dans une archive ZIP (sans exigences de nommage strictes). Cette archive zip doit inclure les fichiers suivants :

- ▶ `manifeste chaincode.yaml`
- ▶ `code GO de chaincode (.go)`
- ▶ `Description OpenAPI (.yaml)`

Le fichier `chaincode.yaml` doit porter exactement ce nom de fichier et doit être stocké dans le répertoire racine de l'archive ZIP. En outre, il doit exister un dossier `/src` dans l'archive ZIP contenant les fichiers `.go` et `.yaml` qui représentent le code de chaîne.

Vous n'êtes pas obligé de placer le code source du chaincode directement dans le dossier `/src`.

## Comprendre les exigences de packages [2/2]

Vous pouvez créer un chemin source plus profond (qui commence toujours par `/src`) dans l'archive ZIP, par exemple :

`/src/myChaincode/`.

Les fichiers `.go` et `.yaml` du chaincode doivent ensuite être stockés dans ce chemin source.

Si d'autres packages sont requis pour le chaincode, ils peuvent également être placés dans le dossier `/src` dans leurs propres sous-dossiers, en suivant l'approche du langage GO habituel pour la structuration du code source.