CHAPTER 1

TinyChat

Pharo permet la définition en quelques dizaines de lignes un server REST grâce au package Teapot qui étend Zinc, le superbe client/server HTTP de Pharo développé par la société BetaNine et offert gracieusement à la communauté. L'objectif de ce chapitre est de vous faire développer en cinq classes, une application de chat client/server avec un client graphique. Cette petite aventure vous permettra de vous familiariser avec Pharo et de voir l'aisance avec laquelle un server REST peut être défini. Développée en quelques heures, TinyChat a été conçu comme une application pédagogique. Dans cet objectif, à la fin de l'article nous proposons une liste d'améliorations possibles.

1.1 Objectifs et architecture

Nous allons donc construire un serveur de discussion (chat) et un client permettant de s'y connecter (voir figure 1.1).

La communication entre le client et le serveur sera basée sur HTTP et REST. En plus des classes `TCServer` et `TinyChat` (le client), nous définirons trois autres classes : la classe `TCMessage` qui représente les messages échangés (dans le future vous pourrez étendre TinyChat pour échanger des éléments plus structurés comme JSON ou STON le format textuel Pharo.), la classe `TCMessageQueue` qui stocke les messages et `TCConsole` l'interface graphique.

1.2 Chargeons Teapot

Teapot est hébergé sur https://github.com/zeroflag/Teapot Vous pouvez utiliser l'expression suivante pour le charger.

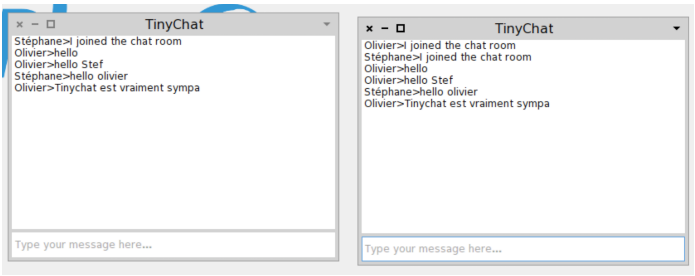

Figure 1.1 Chatting avec TinyChat

```
Metacello new
  baseline: 'Teapot';
  repository: 'github://zeroflag/teapot:master/source';
  load.
```

1.3 Représentation d'un message

Un message est un objet très simple avec un texte et un identifiant pour l'émetteur.

Classe TCMessage

Nous définissons la classe `TCMessage` dans le package `TinyChat`.

```
Object subclass: #TCMessage
  instanceVariableNames: 'sender text separator'
  classVariableNames: ''
  category: 'TinyChat'
```

Les variables d'instances sont les suivantes:

- `sender` : le login de l'expéditeur,
- `text` : le texte du message,
- `separator` : un caractère séparateur pour l'affichage.

Créez les accesseurs pour les variables d'instance 'sender' et 'text'

Nous créons les accesseurs suivants:

```
TCMessage >> sender
  ^ sender

TCMessage >> sender: anObject
  sender := anObject
```

```
TCMessage >> text
  ^ text

TCMessage >> text: anObject
  text := anObject
```

Initialisation de la classe

La méthode `initialize` définit la valeur du caractère séparateur.

```
TCMessage >> initialize
  super initialize.
  separator := '>'.
```

La méthode de classe `TCMessage class>>from:text:` permet d'instancier un message :

```
TCMessage class >> from: aSender text: aText
  ^ self new sender: aSender; text: aText; yourself
```

Le message `yourself` rend le receveur du message : c'est une manière de s'assurer que le nouvel objet créé sera bien retourné par le message `from:text:` et non le résultat du message `text:`.

Convertir un message en chaîne de caractères

Nous ajoutons une méthode `printOn:` pour transformer le message en une chaîne de caractères. Le modèle de la chaîne est sender-separator-text-crlf. Exemple: 'john>hello !!!'. La méthode `printOn:` est invoquée par la méthode `printString`. Il est important de comprendre que la méthode `printOn:` est invoquée par les outils tels que le débogueur ou l'inspecteur d'objets.

```
TCMessage >> printOn: aStream

  aStream
    << self sender; << separator;
    << self text; << String crlf
```

Construire un message à partir d'une chaîne de caractères

Nous devons également définir deux méthodes pour créer un message à partir d'une chaine, ayant la forme: `'olivier>tinychat est cool'`. Tout d'abord, créons une méthode de classe qui sera invoquée de la manière suivante: `TCMessage fromString: 'olivier>tinychat est cool'`, puis la méthode d'instance remplissant les variables de l'objet préalablement créé.

```
TCMessage class >> fromString: aString
  ^ self new
    fromString: aString;
    yourself
```

```
TCMessage >> fromString: aString
  "Compose a message from a string of this form 'sender>message'."
  | items |
  items := aString substrings: separator.
  self sender: items first.
  self text: items second.
```

Maintenant nous sommes prêts pour définir le serveur.

1.4 Le serveur

Pour le serveur, nous allons définir une classe pour gérer une queue de messages. Ce n'est pas vraiment nécessaire mais cela permet de bien identifier les responsabilités.

Stockage des messages

Créez la classe `TCMessageQueue` dans le package TinyChat-Server.

```
Object subclass: #TCMessageQueue
  instanceVariableNames: 'messages'
  classVariableNames: ''
  category: 'TinyChat-server'
```

La variable d'instance `messages` est une collection ordonnée donc le contenu est composé d'instances de `TCMessage`. Une OrderedCollection est une collection qui s'agrandit dynamiquement lors d'ajouts.

```
TCMessageQueue >> initialize
  super initialize.
  messages := OrderedCollection new.
```

Opérations de bases sur la liste des messages

On doit pouvoir ajouter un message `add:`, effacer la liste avec `reset` et connaître le nombre de messages avec `size`.

```
TCMessageQueue >> add: aMessage
  messages add: aMessage

TCMessageQueue >> reset
  messages removeAll

TCMessageQueue >> size
  ^ messages size
```

Obtenir la liste des messages à partir d'une position

Lorsqu'un client demande au serveur la liste des derniers messages échangés, il indique au serveur l'index du dernier message qu'il connaît. Le serveur répond alors la liste des messages reçus depuis cet index.

```
TCMessageQueue >> listFrom: aIndex
  ^ (aIndex > 0 and: [ aIndex <= messages size])
    ifTrue: [ messages copyFrom: aIndex to: messages size ]
    ifFalse: [ #() ]
```

Formatage des messages

La classe `TCMessageQueue` doit pouvoir formater une liste de messages (à partir d'un index) en une chaîne de caractères que le serveur pourra transmettre au client. On ajoute ensuite une méthode à la classe `TCMessageQueue` pour construire une seule chaine de caractères à partir de chaque chaîne de caractères produite par chaque message :

```
TCMessageQueue >> formattedMessagesFrom: aMessageNumber

  ^ String streamContents: [ :formattedMessagesStream |
    (self listFrom: aMessageNumber)
      do: [ :m | formattedMessagesStream << m printString ]
    ]
```

Le Serveur de Chat

Le coeur du serveur est basé sur le framework REST Teapot, permettant l'envoi et la réception des messages. Il maintient en plus une liste de messages qu'il communique aux clients.

Créez la classe TCServer dans le package TinyChat-Server

```
Object subclass: #TCServer
  instanceVariableNames: 'teapotServer messagesQueue'
  classVariableNames: ''
  category: 'TinyChat-Server'
```

La variable d'instance `messagesQueue` référence la liste des messages reçus et envoyés par le serveur.

```
TCServer >> initialize
  super initialize.
  messagesQueue := TCMessageQueue new.
```

La variable d'instance `teapotServer` référence l'instance du serveur TeaPot que l'on créée à l'aide de la méthode `initializePort:`

```
TCServer >> initializePort: anInteger
  teapotServer := Teapot configure: {
    #defaultOutput -> #text.
    #port -> anInteger.
    #debugMode -> true
  }.
  teapotServer start.
```

Le routage HTTP est défini dans la méthode `registerRoutes`. Trois opérations sont définies :

- GET `messages/count` : retourne au client le nombre de messages reçus par le serveur,
- GET `messages/<id:IsInteger>` : le serveur retourne les messages à partir de l'index indiqué dans la requête HTTP,
- POST `/message/add` : le client envoie un message au serveur.

```
TCServer >> registerRoutes
  teapotServer
    GET: '/messages/count' -> (Send message: #messageCount to: self);
    GET: '/messages/<id:IsInteger>' -> (Send message: #messagesFrom:
    to: self);
    POST: '/messages/add' -> (Send message: #addMessage: to: self)
```

Nous exprimons ici que le chemin message/count va donner lieu à l'execution du message `messageCount` sur le serveur lui-même. Le pattern `<id:IsInteger>` indique que l'argument doit être exprimé sous forme de nombre et qu'il sera converti en un entier.

La gestion des erreurs est construite dans la méthode `registerErrorHandlers`. Ici on voit comment on construit une instance de la classe `TeaResponse`.

```
TCServer >> registerErrorHandlers
  teapotServer
    exception: KeyNotFound -> (TeaResponse notFound body: 'No such
    message')
```

Le démarrage du serveur est confié à la méthode `TCServer class>>startOn:` qui reçoit le numéro de port TCP en paramètre.

```
TCServer class >> startOn: aPortNumber
  ^self new
    initializePort: aPortNumber;
    registerRoutes;
    registerErrorHandlers;
    yourself
```

Il faut également gérer l'arrêt du serveur. La méthode `stop` met fin à l'exécution du serveur TeaPot et vide la liste des message.

```
TCServer >> stop
  teapotServer stop.
  messagesQueue reset.
```

Comme il est probable que vous executiez plusieurs fois l'expression `TCServer startOn:`, nous définissons la méthode de classe `stopAll` qui stoppe tous les serveurs en récuperant toutes les instance de la classe. La méthode `TCServer class>>stopAll` demande l'arrêt de chaque instance du serveur.

```
TCServer class >> stopAll
  self allInstancesDo: #stop
```

Traitements réalisés par le serveur

La méthode `addMessage` extrait de la requête du client le message posté. Elle ajoute à la liste des messages une nouvelle instance de `TCMessage`.

```
TCServer >> addMessage: aRequest
  messagesQueue add: (TCMessage from: (aRequest at: #sender) text:
    (aRequest at: #text)).
```

La méthode `messageCount` retourne le nombre de messages reçus.

```
TCServer >> messageCount
  ^ messagesQueue size
```

La méthode `messageFrom:` retourne la liste des messages reçus par le serveur depuis l'index indiqué par le client. Les messages sont retournés au client sous la forme d'une chaine de caractères. Ce point sera définitivement à améliorer.

```
TCServer >> messagesFrom: request
  ^ messagesQueue formattedMessagesFrom: (request at: #id)
```

Nous en avons fini avec le server. Nous pouvons maintenant le tester un peu. Commençons par le lancer :

```
  TCServer startOn: 8181
```

Maintenant nous pouvons soit vérifier avec un navigateur web (figure 1.2), soit à l'aide du client/serveur Zinc disponible par défaut dans Pharo.

```
ZnClient new url: 'http://localhost:8181/messages/count' ; get
```

Les amateurs du shell peuvent également utiliser la commande curl

```
  curl http://localhost:8181/messages/count
```

Nous pouvons aussi ajouter un message de la manière suivante :

```
ZnClient new
  url: 'http://localhost:8181/messages/add';
  formAt: 'sender' put: 'olivier';
```

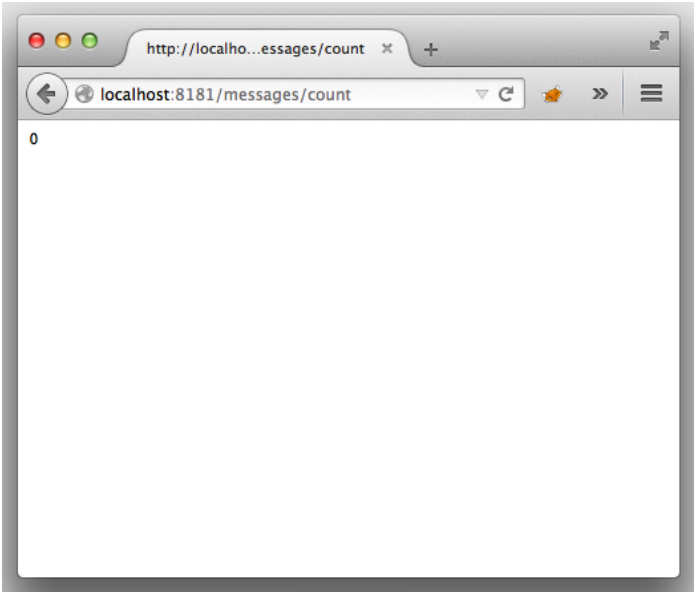

Figure 1.2 Testons le serveur.

```
formAt: 'text' put: 'Super cool ce tinychat' ; post
```

1.5 Le client

Maintenant, nous pouvons nous concentrer sur la partie client de TinyChat. Le client se compose de deux classes:

- `TinyChat` est la classe contenant la logique métier (connexion, envoi et réception des messages),
- `TCConsole` est une classe définissant l'interface graphique.

La logique du client est la suivante:

- Au lancement du client, celui-ci demande au serveur l'index du dernier message reçu,
- Toutes les deux secondes, le client se connecte au serveur pour lire les messages échangés depuis sa dernière connexion. Pour cela, il transmet au serveur l'index du dernier message dont il a eu connaissance.

De plus, lorsque le client transmet un message au serveur, il en profite pour également lire les messages échangés depuis sa dernière connexion.

La classe TinyChat

Nous créons la classe `TinyChat` dans le package `TinyChat-client`.

```
Object subclass: #TinyChat
  instanceVariableNames: 'url login exit messages console
    lastMessageIndex'
  classVariableNames: ''
  category: 'TinyChat-client'
```

Cette classe définit les variables suivantes:

- url contient l'url HTTP permettant au client de se connecter au serveur,
- login est une chaîne de caractères identifiant le client,
- messages est une variable d'instance contenant les messages lus par le client,
- lastMessageIndex est le numéro du dernier message lu par le client,
- exit est une valeur booléenne. Tant que cette valeur est vraie, le client se connecte à intervalle régulier au serveur pour lire les messages échangés depuis sa dernière connexion,
- console pointe sur l'instance de la console graphique permettant à l'utilisateur de saisir et de consulter les messages.

Nous initialisons les variables qui le nécessitent dans la méthode `initialize` suivante.

```
TinyChat >> initialize
  super initialize.
  exit := false.
  lastMessageIndex := 0.
  messages := OrderedCollection new.
```

Définir les commandes HTTP

Nous définissons les méthodes pour communiquer avec le serveur. Elles respectent le protocole HTTP.

Deux méthodes permettent de formater la requête. L'une n'a pas d'argument et permet de construire les requêtes `/messages/add` et `/messages/count`. L'autre a un argument qui est utilisé pour la lecture des messages à partir d'une position.

```
TinyChat >> command: aPath
  ^'{1}{2}' format: { url . aPath }

TinyChat >> command: aPath argument: anArgument
  ^'{1}{2}/{3}' format: { url . aPath . anArgument asString }
```

Il suffit ensuite de définir les trois commandes HTTP du client:

```
TinyChat >> cmdLastMessageID
  ^ self command: '/messages/count'

TinyChat >> cmdNewMessage
  ^self command: '/messages/add'

TinyChat >> cmdMessagesFromLastIndexToEnd
  "Returns the server messages from my current last index to the
    last one on the server."
  ^ self command: '/messages' argument: lastMessageIndex
```

Gérer les opérations du client

Nous avons besoin d'émettre ces commandes et de pouvoir récupérer des informations à partir du serveur. Pour cela, nous définissons deux méthodes. La méthode `readLastMessageID` retourne l'index du dernier message reçu par le serveur.

```
TinyChat >> readLastMessageID
  | id |
  id := (ZnClient new url: self cmdLastMessageID; get) asInteger.
  id = 0 ifTrue: [ id := 1 ].
  ^ id
```

La méthode `readMissingMessages` ajoute les derniers messages reçus par le serveur à la liste des messages connus par le client. Cette méthode retourne le nombre de messages récupérés.

```
TinyChat >> readMissingMessages
  "Gets the new messages that have been posted since the last
    request."
  | response receivedMessages |
  response := (ZnClient new url: self cmdMessagesFromLastIndexToEnd;
    get).
  ^ response
    ifNil: [ 0 ]
    ifNotNil: [
      receivedMessages := response substrings: (String crlf).
      receivedMessages do: [ :msg | messages add: (TCMessage
    fromString: msg) ].
      receivedMessages size.
    ].
```

Nous sommes prêt à définir le comportement de rafraichissement du client avec la méthode `refreshMessages`. Elle utilise un processus léger pour lire à intervalle régulier les messages reçus par le serveur. Le délai est fixé à deux secondes. Le message `fork` envoyé à un bloc (une fermeture lexical en Pharo) exécute ce bloc dans un processus léger. La logique est de boucler tant que le client ne spécifie pas que veut s'arrêter via la variable `exit`. L'expression

`(Delay forSeconds: 2) wait` suspend l'exécution du processus léger dans lequel elle se trouve pendant un certain nombre de secondes.

```
TinyChat >> refreshMessages
  [
    [ exit ] whileFalse: [
      (Delay forSeconds: 2) wait.
      lastMessageIndex := lastMessageIndex + (self
    readMissingMessages).
      console print: messages.
    ]
  ] fork
```

La méthode `sendNewMessage:` poste le message de l'utilisateur au serveur.

```
TinyChat >> sendNewMessage: aMessage
  ^ ZnClient new
    url: self cmdNewMessage;
    formAt: 'sender' put:  (aMessage sender);
    formAt: 'text' put: (aMessage text);
    post
```

Cette méthode est utilisée par la méthode `send:` qui reçoit en paramètre le texte saisi par l'utilisateur. La chaine de caractères est alors convertie en une instance de `TCMessage`. Le message est envoyé. Le client met à jour l'index du dernier message connu et déclenche l'affichage du message dans l'interface graphique.

```
TinyChat >> send: aString
  "When we send a message, we push it to the server and in addition
    we update the local list of posted messages."

  | msg |
  msg := TCMessage from: login text: aString.
  self sendNewMessage: msg.
  lastMessageIndex := lastMessageIndex + (self readMissingMessages).
  console print: messages.
```

La déconnexion du client est gérée par la méthode `disconnect` qui envoie un message au serveur pour signaler le départ de l'utilisateur et met fin à la boucle de récupération périodique des messages.

```
TinyChat >> disconnect
  self sendNewMessage: (TCMessage from: login text: 'I exited from
    the chat room.').
  exit := true
```

Fixer les paramètres du client

Pour initialiser les paramètres de connexion, on définit une méthode de class `TinyChat class>>connect:port:login:`. Cette méthode permet de se con-

necter de la manière suivante : `TinyChat connect: 'localhost' port: 8080 login: 'username'`

```
TinyChat class >> connect: aHost port: aPort login: aLogin

  ^ self new
    host: aHost port: aPort login: aLogin;
    start
```

Le code appelle la méthode `host:port:login:`. Cette méthode met à jour la variable d'instance `url` en construisant l'URL et en affectant le nom de l'utilisateur à la variable d'instance `login`.

```
TinyChat >> host: aHost port: aPort login: aLogin
  url := 'http://' , aHost , ':' , aPort asString.
  login := aLogin
```

La méthode `start` envoie un message au serveur pour présenter l'utilisateur, récupèrer l'index du dernier message reçu par le serveur et mettre à jour la liste des messages connus par le client. C'est également cette méthode qui initialise l'interface graphique de l'utilisateur. Une évolution pourrait être de décoreller le modèle de son interface graphique en utilisant une conception basée sur des évenements.

```
TinyChat >> start
  console := TCConsole attach: self.
  self sendNewMessage: (TCMessage from: login text: 'I joined the
    chat room').
  lastMessageIndex := self readLastMessageID.
  self refreshMessages.
```

Création de l'interface graphique

L'interface graphique est composée d'une fenêtre contenant une liste et un champ de saisie comme montré dans la figure 1.1.

```
ComposablePresenter subclass: #TCConsole
  instanceVariableNames: 'chat list input'
  classVariableNames: ''
  category: 'TinyChat-client'
```

La variable d'instance `chat` est une référence à une instance de la classe `TinyChat` et nécessite uniquement un accesseur en écriture. Les variables d'instance `list` et `input` dispose chacune d'un accesseur en lecture. Ceci est imposé par Spec le constructeur d'interface.

```
TCConsole >> input
  ^ input

TCConsole >> list
  ^ list
```

```
TCConsole >> chat: anObject
  chat := anObject
```

L'interface graphique a un titre pour la fenêtre. Pour le définir, il faut écrire une méthode `title`.

```
TCConsole >> title
  ^ 'TinyChat'
```

La méthode de classe `TCConsole class>>attach:` reçoit en argument l'instance du client de chat avec lequel l'interface graphique va être utilisée. Cette méthode déclenche l'ouverture de la fenêtre et met en place l'événement gérant la fermeture de celle ci et donc, provoquant la déconnexion du client.

```
TCConsole class >> attach: aTinyChat
  | window |
  window := self new chat: aTinyChat.
  window openWithSpec whenClosedDo: [ aTinyChat disconnect ].
  ^ window
```

La méthode `TCConsole class>>defaultSpec` définit la mise en page des composants contenus dans la fenêtre. Ici nous avons une colonne avec une liste et un champ de saisie placé juste en dessous.

```
TCConsole class >> defaultSpec
  <spec: #default>

  ^ SpecLayout composed
    newColumn: [ :c |
      c add: #list; add: #input height: 30 ]; yourself
```

La méthode `initializeWidgets` spécifie la nature et le comportement des composants graphiques. Ainsi le `acceptBlock:` permet de définir l'action à exécuter lorsque le texte est entré dans le champ de saisie. Ici nous l'envoyons à client et nous le vidons.

```
TCConsole >> initializeWidgets

  list := ListModel new.
  input := TextInputFieldModel new
    ghostText: 'Type your message here...';
    enabled: true;
    acceptBlock: [ :string |
      chat send: string.
      input text: '' ].
  self focusOrder add: input.
```

La méthode `print` affiche les messages reçus par le client en les affectant au contenu de la liste.

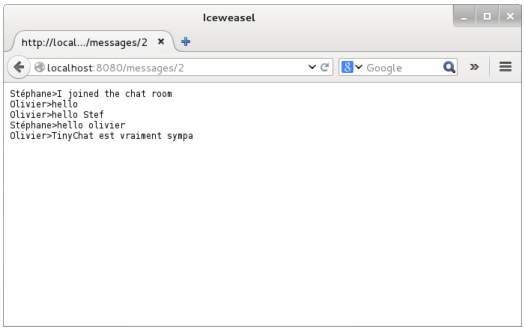

Figure 1.3 Acces direct au server.

```
TCConsole >> print: aCollectionOfMessages
  list items: (aCollectionOfMessages collect: [  :m |  m printString
    ])
```

Notez que cette méthode est invoquée par la méthode `refreshMessages` et que changer tous les éléments de la liste à chaque ajout d'un nouveau messages est peu élégant mais l'exemple se veut volontairement simple.

Voilà vous pouvez maintenant chatter avec votre serveur.

1.6 Conclusion

Nous avons montré que la création d'un serveur REST est extrêmement simple avec Teapot. La définition de TinyChat donne un cadre ludique à l'exploration de la programmation en Pharo et nous esperons que vous avez apprécié cette ballade. TinyChat est une petite application que nous avons développé de manière très simple afin de vous permettre de l'étendre et d'expérimenter. Voici une liste d'améliorations : gestion parcimonieuse des ajouts d'éléments dans la liste graphique, gestion d'accès concurrents dans la collection sur le serveur (en effet, si le serveur pouvait recevoir des requêtes concurrentes la structure de donnée utilisée n'est pas adéquate), gestion des erreurs de connexion, rendre les clients robustes à la fermeture du serveur, obtenir la liste des personnes connectées, pouvoir définir le délai de récupération des messages, utiliser JSON pour le transport des messages, afficher le nom de la personne connectée dans la fenêtre. Le projet est disponible à l'adresse http://www.smalltalkhub.com/#!/~olivierauverlot/TinyChat. A vous de jouer!