

# Sendo um Servlet

Ele usou uma solicitação GET para atualizar o banco de dados. A punição será a mais severa... sem aulas de "Ioga com Suzy" por 90 dias.



**Servlets vivem para servir clientes.** A função de um servlet é receber uma **solicitação** do cliente e devolver uma **resposta**. A solicitação talvez seja simples: *"traga-me a página de Boas-vindas"*. Ou pode ser complexa: *"Finalize o processo do meu carrinho de compras."* A solicitação traz consigo dados cruciais e o código do seu servlet tem que saber como *encontrá-los* e *utilizá-los*. A resposta leva a informação que o browser precisa para montar uma página (ou baixar alguns dados) e o código do seu servlet tem que saber como *enviá-los*. Ou *não*... em vez disso, seu servlet pode decidir encaminhar a solicitação *adiante* (para outra página, servlet ou JSP).

# Objetivos



## O Modelo de Tecnologia do Servlet

- 1.1 Para cada um dos Métodos HTTP (como GET, POST, HEAD e assim por diante), descrever o propósito do método e as características técnicas do protocolo do Método HTTP, listar triggers que possam levar o cliente (geralmente um browser) a usar o Método e identificar o método HttpServlet que corresponda ao Método HTTP.
- 1.2 Usando a interface HttpServletRequest, escrever o código que retira da solicitação os parâmetros do formulário HTML, a informação do header da solicitação HTTP ou os cookies.
- 1.3 Usando a interface HttpServletResponse, escrever o código que cria um header para a resposta HTTP, configura o tipo de conteúdo da resposta, recebe um stream de texto para a resposta, recebe um stream binário para a resposta, redireciona uma solicitação HTTP para outra URL, ou adiciona cookies na resposta.\*
- 1.4 Descrever o propósito e a sequência de eventos do ciclo de vida de um servlet:  
(1) carregar a classe servlet, (2) instanciar o servlet, (3) chamar o método init(), (4) chamar o método service(), e (5) chamar o método destroy().

## Notas sobre a Abrangência:

*Todos os objetivos desta seção são cobertos completamente neste capítulo, com exceção da parte dos cookies, no objetivo 1.3. Grande parte do conteúdo deste capítulo foi comentada no capítulo dois, mas lá nós dissemos: "Não se preocupe em decorar isto."*

*Neste capítulo, você TEM que ir devagar, realmente estudar e memorizar o conteúdo. Nenhum outro capítulo cobrirá estes objetivos com detalhes; então, esta é a hora.*

*Faça os exercícios, revise o material e faça seu primeiro teste preparatório no final deste capítulo. Se você não conseguir pelo menos 80% das respostas corretas, volte para descobrir o que você deixou escapar, ANTES de passar para o capítulo cinco.*

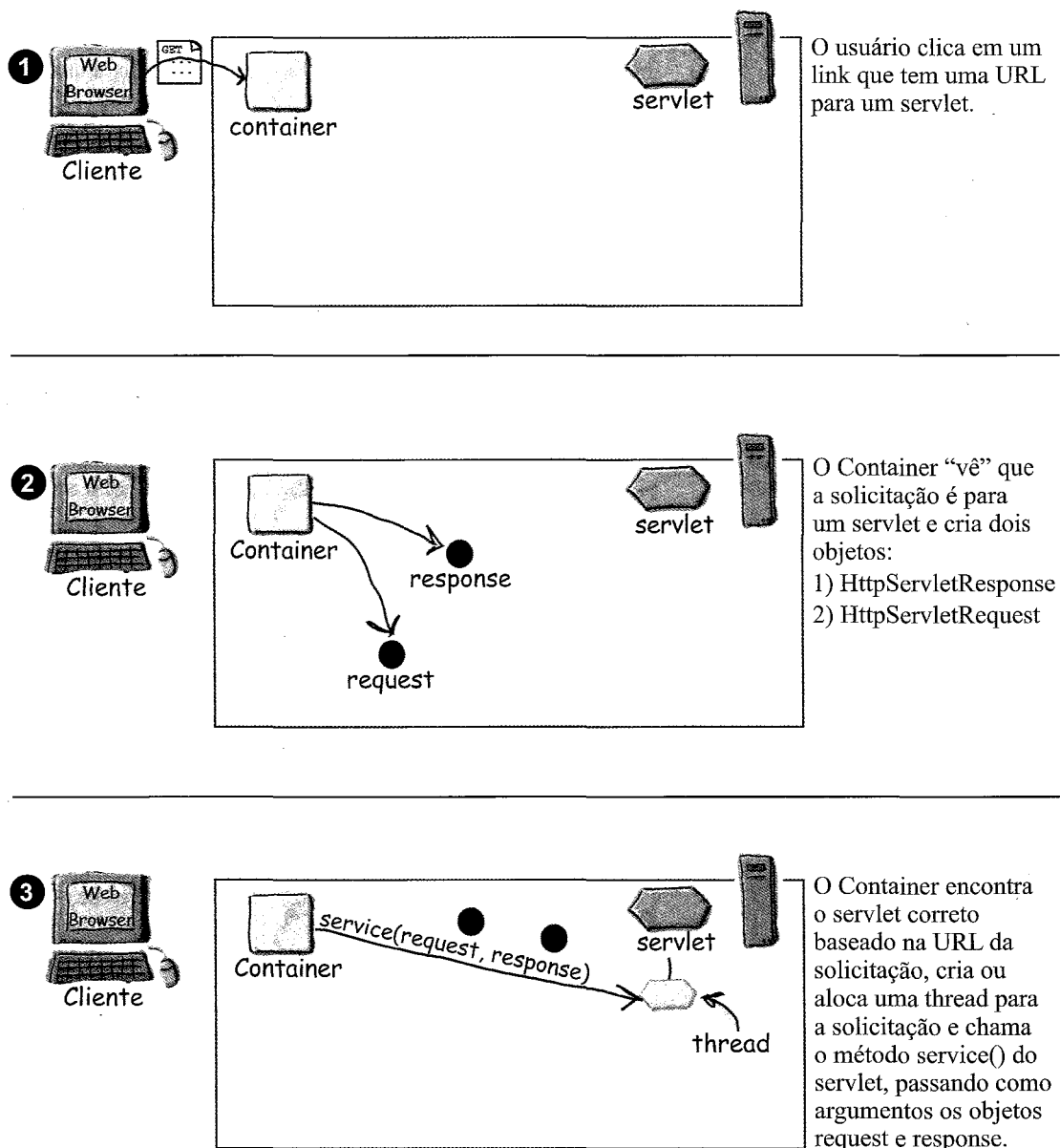
*Algumas das perguntas do teste preparatório que fazem parte destes objetivos foram colocadas nos capítulos 5 e 6, por requererem um conhecimento adicional de alguns assuntos que não explicamos até aqui. Isto significa que teremos um número menor de questões preparatórias neste capítulo e maior nos próximos, evitando testá-lo naquilo que você ainda não viu.*

*Nota importante: enquanto os três primeiros capítulos abordaram assuntos que servem de base, desta página em diante quase tudo o que você verá está diretamente relacionado ou é explicitamente parte do exame.*

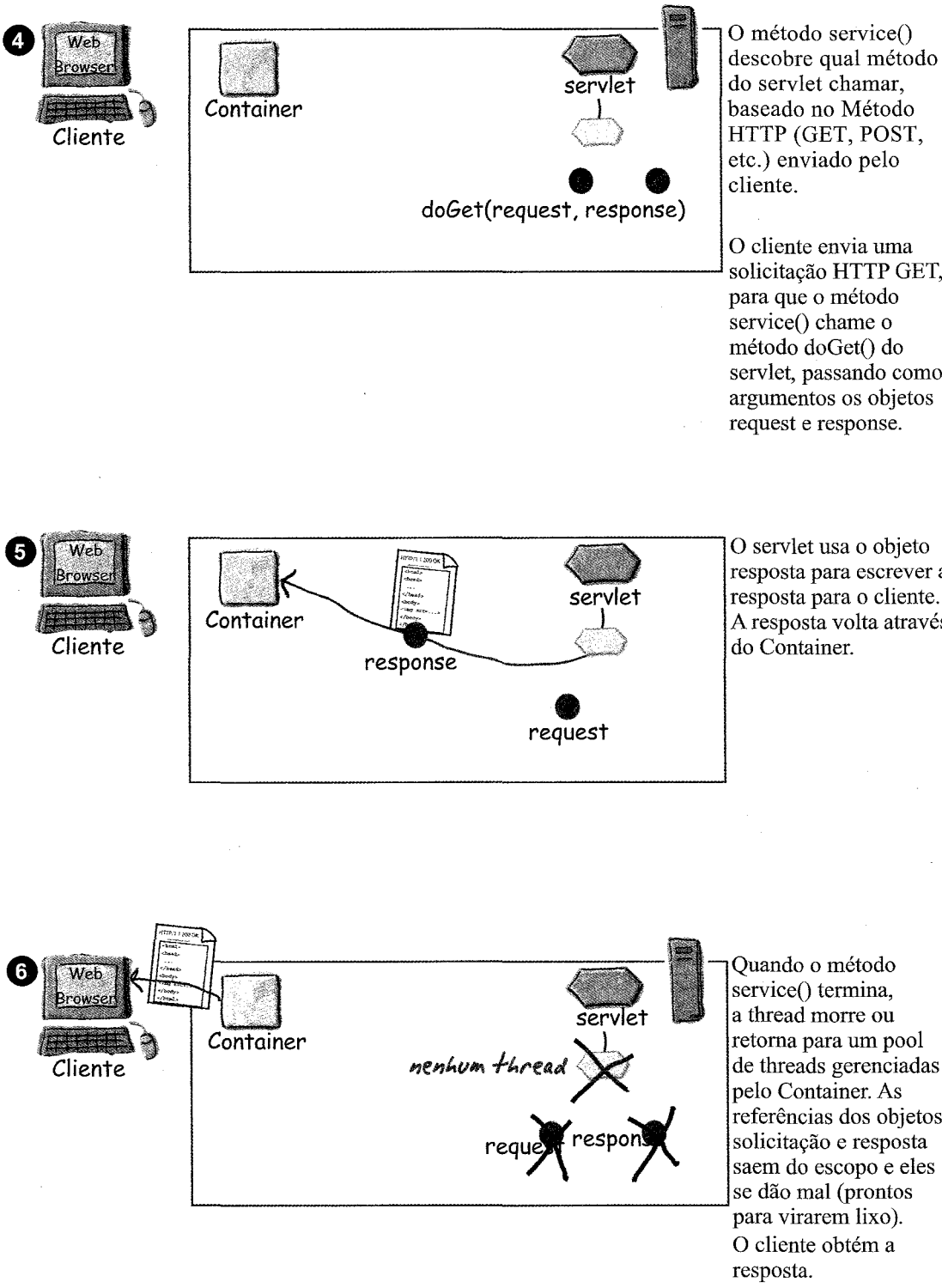
\*Não falaremos muito a respeito dos objetivos relacionados aos cookies até o capítulo que trata das Sessões.

## Os Servlets são controlados pelo Container

No capítulo dois nós vimos as funções completas do Container na vida do servlet: ele cria os objetos request e response, cria ou aloca uma nova thread para o servlet e chama o método `service()` do servlet, passando as referências de request e response como argumentos. Aqui vai uma rápida revisão...



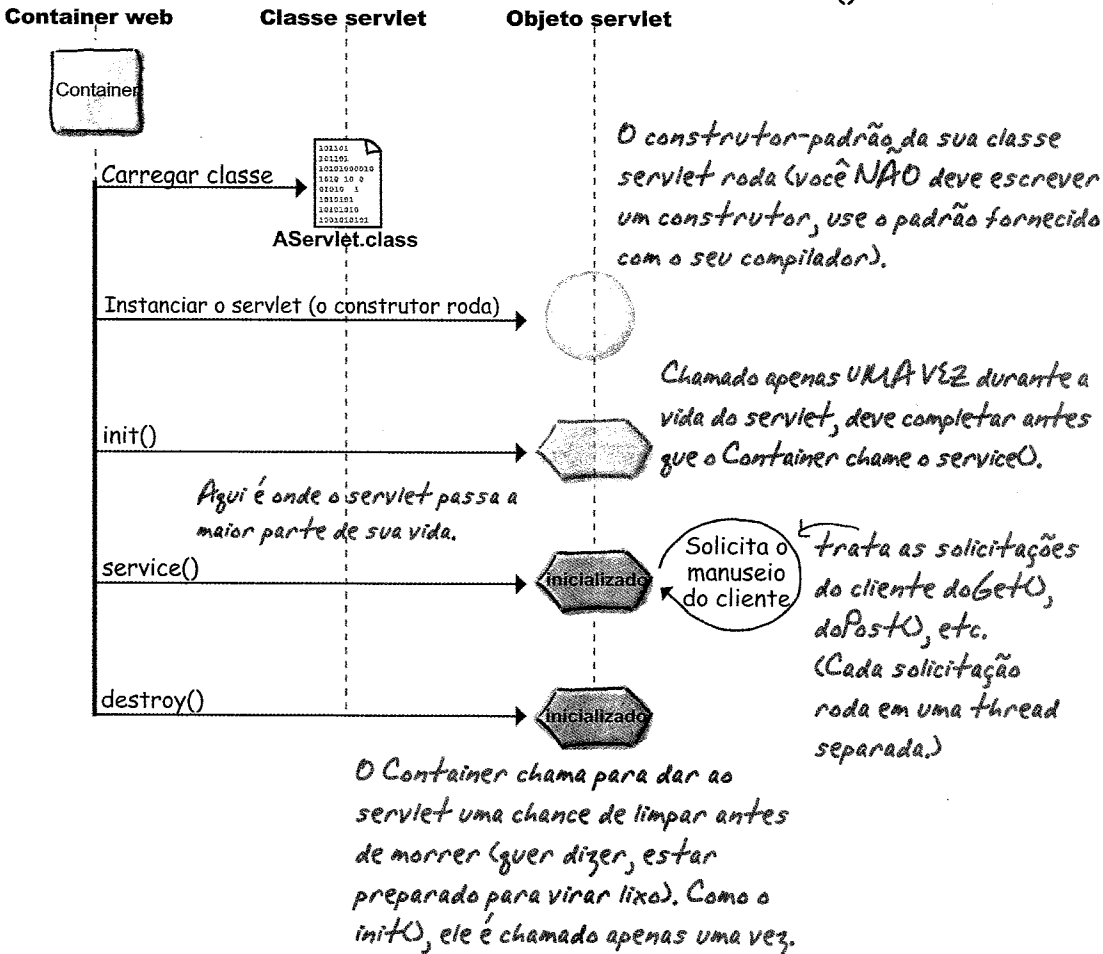
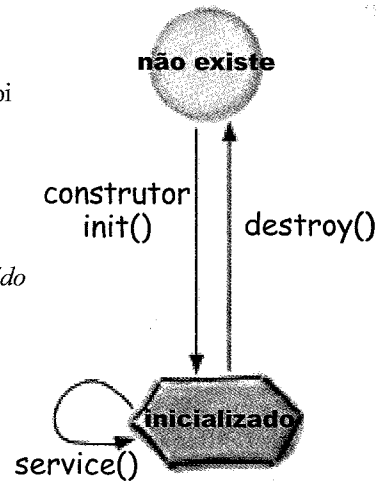
A história continua...



## Mas a vida do servlet não é só isso

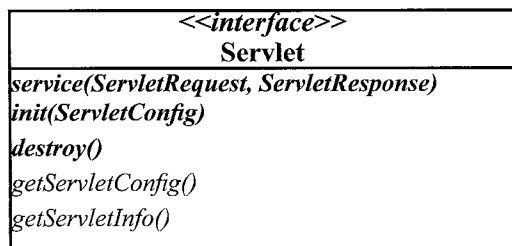
Nós fomos até o meio da vida do servlet, mas ainda existem perguntas: Quando a classe servlet foi carregada? Quando o construtor do servlet foi executado? Quanto tempo vive o objeto servlet? Quando o seu servlet deve iniciar os recursos? E quando ele deve limpá-los?

O ciclo de vida do servlet é simples: existe apenas um estado principal – **inicializado**. Se o servlet não está inicializado, ou ele está *sendo inicializado* (rodando seu construtor ou o método `init()`), *sendo destruído* (rodando seu método `destroy()`), ou simplesmente **não existe**.



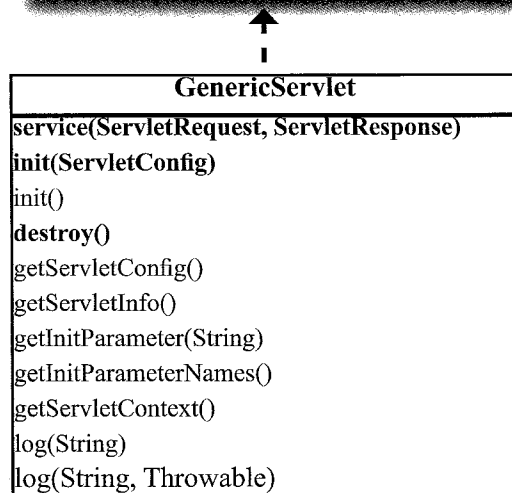
*Nota: NÃO tente memorizar tudo isto agora! Apenas sinta como a API trabalha...*

## Seu servlet herda os métodos do ciclo de vida



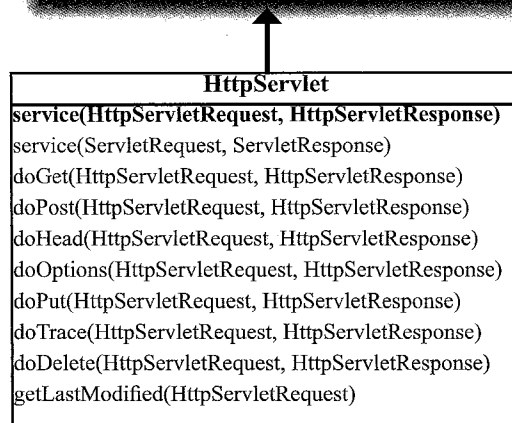
A interface Servlet  
(javax.servlet.Servlet)

*A interface Servlet diz que todos os servlets possuem estes cinco métodos (os três em negrito são métodos referentes ao ciclo de vida).*



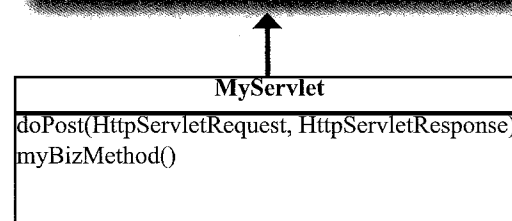
A classe GenericServlet  
(javax.servlet.GenericServlet)

*A GenericServlet é uma classe abstrata que implementa a maioria dos métodos básicos do servlet que você precisará, incluindo aqueles da interface Servlet. Você provavelmente NUNCA fará um extend nesta classe. A maioria do "comportamento servlet" do seu servlet vem daqui.*



A classe HttpServlet  
(javax.servlet.http.HttpServlet)

*A HttpServlet (também uma classe abstrata) implementa o método service() para refletir as características HTTP do servlet - o método service() não recebe NENHUMA servlet de solicitação e resposta antigo, mas uma solicitação e resposta específica para HTTP.*



A classe MyServlet  
(com.wickedlysmart.foo)

*A maioria das características do seu servlet é tratada pelos métodos das superclasses. Tudo o que você faz é anular os métodos HTTP que você precisa.*

## Os Três Grandes Momentos do Ciclo de Vida

<b>1</b> <b>init( )</b>	<b>Quando ele é chamado</b> O Container chama o <code>init()</code> na instância servlet <i>depois</i> que a instância servlet foi criada, porém, <i>antes</i> que o servlet sirva a qualquer solicitação do cliente.	<b>Para que serve</b> Possibilita que você inicialize seu servlet antes de tratar quaisquer solicitações do cliente.	<b>Ele pode ser anulado?</b> <i>Possivelmente.</i> Se você tiver código para inicialização (como estabelecer uma conexão com um banco de dados ou registrar-se em outros objetos), então você anula o método <code>init()</code> na sua classe servlet.
<b>2</b> <b>service( )</b>	<b>Quando ele é chamado</b> Quando chega a primeira solicitação do cliente, o Container inicia uma nova thread ou aloca uma thread do pool, fazendo com que o método <code>service()</code> do servlet seja ativado.	<b>Para que serve</b> Este método analisa a solicitação, determina o método HTTP (GET, POST, etc.) e chama o respectivo <code>doGet()</code> , <code>doPost()</code> , etc. no servlet.	<b>Ele pode ser anulado?</b> <i>Não. Dificilmente.</i> Você <b>NÃO</b> deve anular manualmente o método <code>service()</code> . Seu trabalho é anular os métodos <code>doGet()</code> e/ou <code>doPost()</code> e deixar que a implementação <code>service()</code> do <code>HTTPServlet</code> se preocupe em chamar o método correto.
<b>3</b> <b>doGet( )</b> and/or <b>doPost( )</b>	<b>Quando ele é chamado</b> O método <code>service()</code> chama o <code>doGet()</code> ou <code>doPost()</code> baseado no método HTTP (GET, POST, etc.) da solicitação.  (Estamos incluindo aqui apenas o <code>doGet()</code> e o <code>doPost()</code> , pois provavelmente serão os únicos que você usará.)	<b>Para que serve</b> É aqui que o <i>seu</i> código começa! Este é o método responsável por tudo que esperamos que sua aplicação FAÇA.  Você pode chamar outros métodos em outros objetos, é claro, porém tudo começa daqui.	<b>Ele pode ser anulado?</b> <i>SEMPRE, pelo menos, UM deles! (doGet() ou doPost())</i>  Aquele que você anular irá informar ao Container o que você suporta. Por exemplo, se você não anular o <code>doPost()</code> , você estará dizendo ao Container que este servlet não dá suporte às solicitações HTTP POST.

Eu acho que entendi... o Container chama o método `init()` do meu servlet, mas se eu não anular o `init()`, o método do `GenericServlet` roda. Então, quando a solicitação chega, o Container inicia ou aloca uma thread e chama o método `service()`, que eu não anulo, e o método `service()` do `HttpServlet` roda. O método `service()` do `HttpServlet` chama meu `doGet()` ou `doPost()` que foi anulado. Então, cada vez que meu `doGet()` ou `doPost()` roda, ocorre em uma thread separada.

### Inicialização do Servlet



#### Thread A

O Container chama o `init()` na instância servlet *depois* que esta é criada, mas *antes* que o servlet atenda alguma solicitação do cliente. Se você tiver código para inicialização (como estabelecer uma conexão com um banco de dados ou se registrar em outros objetos), então você anula o método `init()` na sua classe servlet. Do contrário, o método `init()` do `GenericServlet` roda.

### O método `service()` é sempre chamado em sua própria pilha...

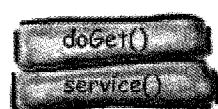
#### Solicitação do cliente 1



#### Thread B

Quando a primeira solicitação do cliente chega, o Container inicia (ou localiza) uma thread e induz o método `service()` do servlet a ser executado. Você normalmente NÃO anulará o método `service()`, e o método do `HttpServlet` é que rodará. O método `service()` descobre qual método HTTP (GET, POST, etc.) está na solicitação e chama o respectivo método `doGet()` ou `doPost()`. O `doGet()` e o `doPost()` dentro do `HttpServlet` não fazem nada, então você terá que anular um ou ambos. Esta thread morre (ou é colocada de volta em um pool gerenciado pelo Container) quando o `service()` é finalizado.

#### Solicitação do cliente 2



#### Thread C

Quando a segunda (e todas as outras) solicitações do cliente chegam, o Container novamente cria ou encontra uma outra thread e induz o método `service()` do servlet a ser executado. Então, a sequência do método `service()` --> `doGet()` ocorre cada vez que existe uma solicitação do cliente. Em um determinado momento, você terá ao menos, tantas threads sendo executadas, quantas solicitações de clientes houver, limitadas pelos recursos ou políticas/configuração do Container. (Você pode, por exemplo, ter um Container que permita especificar a quantidade máxima de threads simultâneas, e quando o número das solicitações do cliente ultrapassá-la, alguns clientes terão apenas que esperar).

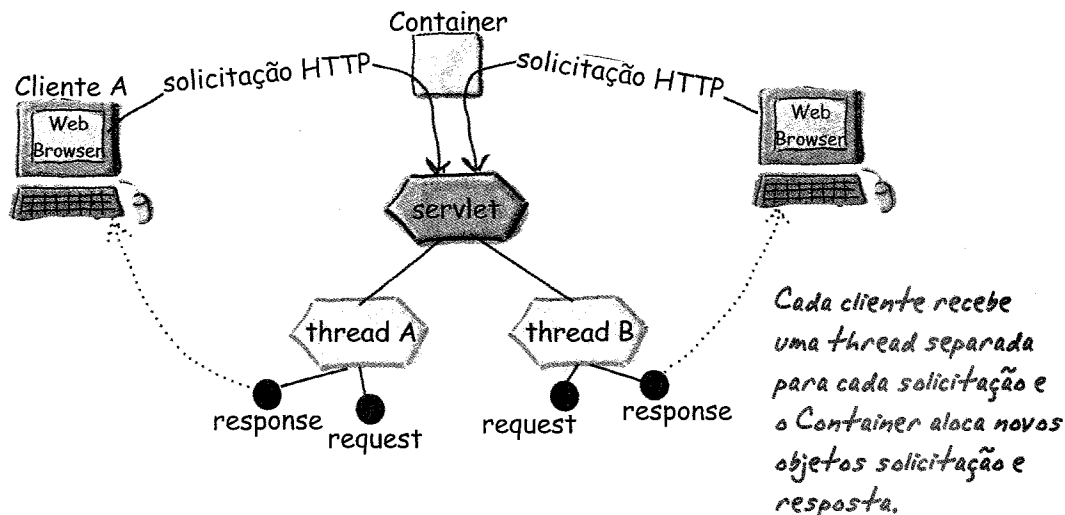


## Cada solicitação roda em uma thread separada!

Você talvez já tenha ouvido alguém dizer coisas como: “Cada instância do servlet...”, mas isto está *errado*. Não existem múltiplas *instâncias* de nenhuma classe servlet, exceto para um caso especial (chamado SingleThreadModel, de natureza perversa). Porém, ainda não estamos falando desse caso especial.

O Container roda várias *threads* para processar as várias solicitações para um único servlet.

E cada solicitação do cliente gera um novo par de objetos request e response..



## <sup>Não existem</sup> Perguntas Idiotas

**P:** Isto está confuso... na figura acima você mostra dois clientes diferentes, cada um com sua própria thread. O que acontece se o *mesmo* cliente fizer várias solicitações? É uma thread por *cliente* ou uma thread por *solicitação*?

**R:** Uma thread por solicitação. O Container não se importa com quem fez a solicitação – cada solicitação que chega significa uma nova thread/pilha.

**P:** E se o Container usar cluster e distribuir a aplicação em mais de uma JVM?

**R:** Imagine que a figura acima é para uma simples JVM e que cada JVM tenha a mesma figura. Então, para uma aplicação distribuída, existiria uma instância de um determinado servlet para cada JVM. Porém, cada JVM ainda teria apenas uma única instância daquele servlet.

**P:** Eu notei que o *HttpServlet* está num pacote diferente do *GenericServlet*... quantos pacotes servlet existem?

**R:** Tudo relacionado a servlets (exceto o que se refere ao JSP) está em *javax.servlet* ou em *javax.servlet.http*. É fácil ver a diferença... o que se refere a HTTP está no pacote *javax.servlet.http* e o restante (classes genéricas de servlet e interfaces) está em *javax.servlet*. Veremos capítulos que tratam do JSP mais adiante.

## No começo: carregando e inicializando

O servlet nasce quando o Container encontra o arquivo de classe servlet. Isto acontece quase sempre quando o Container inicia (por exemplo, quando você roda o Tomcat). Quando o Container inicia, ele procura por aplicações distribuídas e então localiza os arquivos de classe servlet. (No capítulo sobre Distribuição, nós entraremos em mais detalhes de como, por que e onde o Container procura os servlets.)

*Encontrar a classe é o primeiro passo.*

*Carregar a classe é o segundo passo. E isso acontece na inicialização do Container ou na primeira utilização do cliente. Seu Container talvez lhe possibilite escolher qual classe carregar, ou talvez carregar a classe sempre que ele quiser.*

*Independentemente de o seu Container preparar o servlet antes ou exatamente no momento que o cliente necessita, um método `service()` do servlet não rodará até que o servlet seja inteiramente inicializado.*

Seu servlet é  
sempre carregado e  
inicializado ANTES  
de servir à primeira  
solicitação do cliente.

**O `init()` sempre termina antes da primeira chamada ao `service()`**



### EXERCITE SUA MENTE

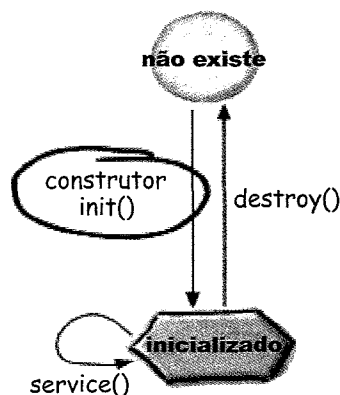
Por que existe um método `init()`? Em outras palavras, por que o *construtor* não é suficiente para inicializar um servlet?

Que tipo de código você deveria colocar no método `init()`?

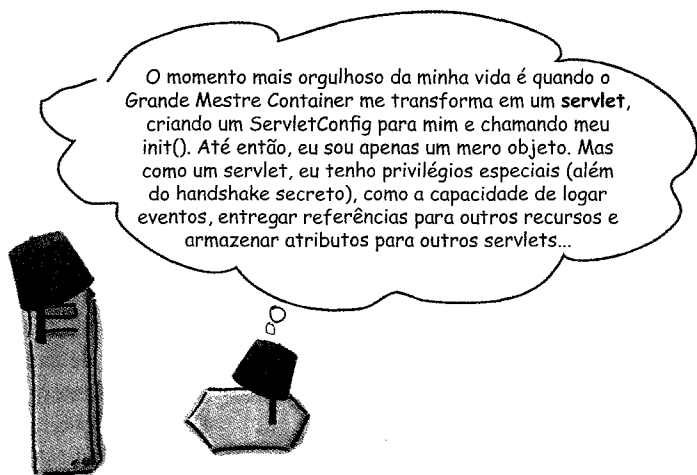
Dica: o método `init()` usa um argumento de referência ao objeto. O que você acha que seria o argumento para o método `init()` e como (ou por que) você o usaria?

*A mesma coisa com a resposta... `HttpServletResponse` acrescenta métodos que serão importantes se você estiver usando HTTP, erros, cookies e headers.*

## A Inicialização do Servlet: quando um objeto torna-se um servlet



*O init() roda apenas uma vez na vida do servlet, então não o detone! É não tente fazer nada tão cedo... está muito cedo para que o construtor faça tarefas específicas para o servlet.*



Um servlet vai do *não existe* para *inicializado* (que na verdade significa *pronto para servir às solicitações dos clientes*), começando com um construtor. Mas o construtor cria apenas um *objeto*, não um *servlet*. Para ser um servlet, o objeto precisa adquirir padrão de servlet.

Quando um objeto torna-se um servlet, ele recebe todos os privilégios que se têm quando se é um servlet, como a capacidade para usar sua referência *ServletContext* para obter informações do Container.

### *Por que nos preocupamos com os detalhes da inicialização?*

Porque em algum lugar entre o construtor e o método `init()`, o servlet está no estado *servlet Schroedinger*\*. Você pode possuir um código para inicialização do servlet, como receber informação de configuração da aplicação web, ou procurar por uma referência em outro trecho da aplicação, que irá **dar erro** se você executá-la muito *cedo* na vida do servlet. Contudo, é muito simples se você se lembrar de não colocar nada no construtor do servlet! Não há nada que não possa esperar até o `init()`.

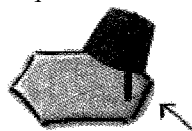
\* Se sua parte mecânica está um pouco enferrujada, talvez você queira fazer uma pesquisa no Google por "Schroedinger's Cat" (cuidado: se você ama animais, não a faça). Quando nos referimos ao estado *Schroedinger*, nos referimos a algo que não está nem totalmente morto, nem totalmente vivo, mas em algum lugar estranho entre ambos.

## Quanto vale para você “ser um servlet”?

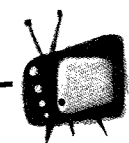
O que acontece quando  
um servlet vai daqui:



para cá?



servlet em uso, válido



Veja isto!

**Não confunda os  
parâmetros do  
ServletConfig com  
os parâmetros do  
ServletContext!**

Nós realmente não falaremos sobre isso até o próximo capítulo, mas já que tantas pessoas se confundem, vamos plantar uma semente agora: **preste atenção nas diferenças.**

Comece olhando pelos nomes:

O **ServletConfig** tem a palavra “config” no nome, que lembra “configuração”.

Ele lida com valores de tempo de distribuição que você configurou para o servlet (um por servlet). Aquilo que seu servlet pode querer acessar e que você não quer fazer hardcoded, como o nome de um banco de dados, por exemplo.

Os parâmetros do **ServletConfig** não mudarão, desde que este servlet esteja distribuído e rodando. Para alterá-los, você terá que redistribuir o servlet.

O **ServletContext** dever-se-ia chamar **AppContext** (mas eles não nos deram atenção), pois há somente um por aplicação e NÃO um por servlet. De qualquer jeito, nós entraremos neste assunto no próximo capítulo – isto é só um alerta.

### 1 Um objeto ServletConfig

- Um objeto **ServletConfig** por servlet.
- Use-o para passar informações de tempo de distribuição para o servlet (um banco de dados ou a pesquisa do nome de um enterprise bean, por exemplo) que você não queira fazer hardcoded no servlet (parâmetros init do servlet).
- Use-o para acessar o **ServletContext**.
- Os parâmetros são configurados no **Deployment Descriptor**.

### 2 Um ServletContext

- Um **ServletContext** por aplicação. (Eles deveriam tê-lo chamado de **AppContext**.)
- Use-o para acessar *parâmetros* da aplicação (também configurado no **Deployment Descriptor**).
- Use-o como se fosse um quadro de avisos da aplicação, onde você pode escrever mensagens (conhecidas como atributos) que as outras partes da aplicação possam acessar (mais sobre isto no próximo capítulo).

Use-o para obter informações do servidor, incluindo o nome e a versão do **Container** e a versão da API que é suportada.

**Mas a VERDADEIRA função de um Servlet é tratar solicitações.**

**É aí que a vida do servlet faz diferença.**

No capítulo seguinte estudaremos o `ServletConfig` e o `ServletContext`, mas por hora, estamos vendo em detalhes a solicitação e a resposta. Porque o `ServletConfig` e o `ServletContext` existem apenas para darem suporte à Única e Verdadeira Tarefa do servlet: tratar as solicitações do cliente! Portanto, antes de vermos como os seus objetos contexto e configuração podem ajudá-lo em seus trabalhos, teremos que voltar um pouco e rever os fundamentos da solicitação e resposta.

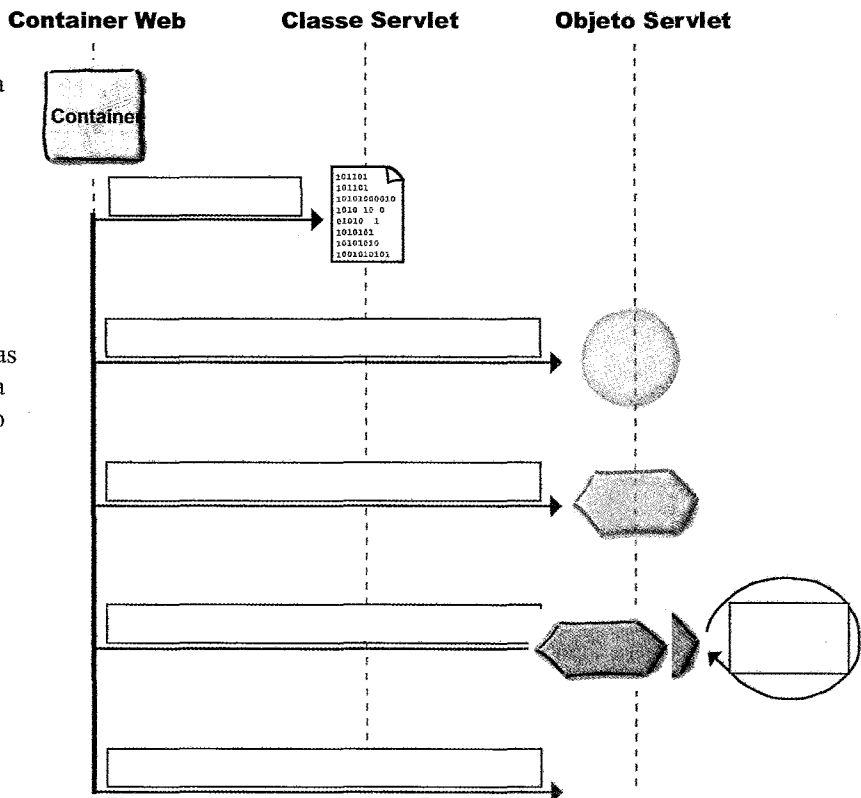
Você já sabe que a solicitação e a resposta são passadas como argumentos para o método `doGet()` ou `doPost()`, mas que *poderes* estes objetos `request` e `response` oferecem? O que você pode fazer com eles e por que você se importa com isso?



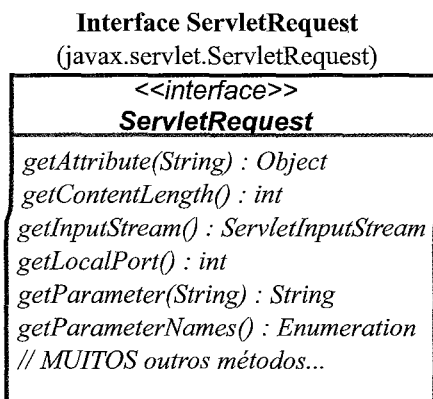
Aponte seu lápis

Coloque o nome nos trechos em branco (as caixas vazias) da linha do tempo do ciclo de vida. (Verifique suas respostas com a linha do tempo mostrada anteriormente neste capítulo.)

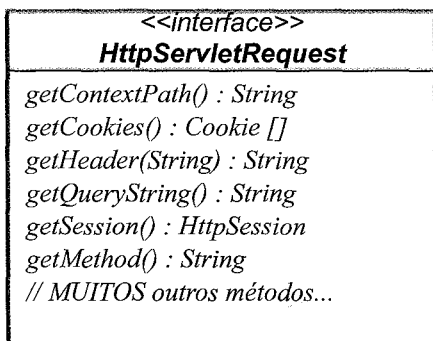
Acréscite também suas próprias anotações para facilitar a memorização dos detalhes.



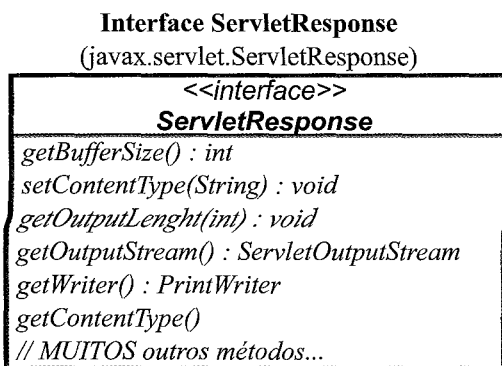
## Solicitação e Resposta: a chave para tudo, e os argumentos para o service()\*



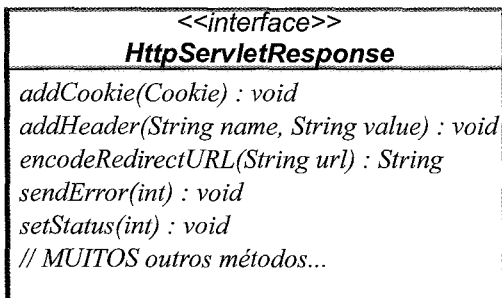
**Interface HttpServletRequest**  
(javax.servlet.http.HttpServletRequest)



Os métodos `HttpServletRequest` referem-se a assuntos HTTP, como cookies, headers e sessões. A interface `HttpServletRequest` acrescenta os métodos que se referem ao protocolo HTTP... que seu servlet utiliza para se comunicar com o cliente/brower.



**Interface HttpServletResponse**  
(javax.servlet.http.HttpServletResponse)



A mesma coisa com a resposta... o `HttpServletResponse` acrescenta métodos que serão importantes se você estiver usando HTTP - erros, cookies e headers.

\*Os objetos solicitação e resposta também são argumentos para os outros métodos `HttpServlet` que você escreve: `doGet()`, `doPost()`, etc.

## Não existem Perguntas Idiotas



**O exame não espera  
que você saiba  
desenvolver usando  
servlets não-HTTP.**

**P:** Quem implementa as interfaces para o `HttpServletRequest` e o `HttpServletResponse`? São aquelas classes na API?

**R:** Não. O Container. As classes não estão na API, pois fica a cargo dos fabricantes implementá-las. A boa notícia é que você não precisa se preocupar com isso. Apenas creia que quando o método `service()` é chamado em seu servlet, ele receberá referências para dois excelentes objetos que *implementam* o `HttpServletRequest` e o `HttpServletResponse`. Você jamais se preocupa com o nome e o tipo verdadeiros da classe envolvidos nesta implementação. Tudo que interessa é que você terá alguma coisa com todas as funcionalidades do `HttpServletRequest` e do `HttpServletResponse`.

Ou seja, tudo o que você precisa conhecer são os *métodos que você pode chamar* nos objetos que o Container oferece como parte da solicitação! A verdadeira classe na qual serão implementados não faz diferença – você se refere aos objetos `request` e `response` apenas pelo tipo da interface.

**P:** Eu estou lendo esta UML corretamente? Essas interfaces estão *estendendo* interfaces?

**R:** Sim. Lembre-se, as interfaces podem ter sua própria árvore de herança. Quando uma interface *estende* uma outra interface (e é tudo o que elas podem fazer – pois interfaces não *implementam* interfaces), significa que quem implementar uma interface deve implementar *todos* os métodos definidos na interface e em suas superinterfaces. Isto quer dizer, por exemplo, que aquele que implementar o `HttpServletRequest` deve prover métodos de implementação para os métodos declarados nas interfaces `HttpServletRequest` e `ServletRequest`.

**P:** Eu ainda estou confuso com o porquê de existir um `GenericServlet`, um `ServletRequest` e um `ServletResponse`. Se ninguém está fazendo nada, exceto os servlets HTTP... qual é a intenção?

**R:** Nós não dissemos *ninguém*. Alguém, em algum lugar, não sei, está usando o modelo de tecnologia servlet sem o protocolo HTTP. Mas nunca encontramos ou soubemos da existência desse alguém.

Além disso, o modelo servlet possui flexibilidade para atender àqueles que queiram usá-lo com, por exemplo, o SMTP, ou talvez um protocolo proprietário customizado. Porém, a API só oferece suporte nativo ao HTTP, que é o que quase todo mundo usa.

## O método de solicitação HTTP define se é o doGet() ou doPost() que rodará

Lembre-se, a solicitação do cliente sempre inclui um Método HTTP específico. Se o Método HTTP for um GET, o método service() chama o doGet(). Se for um POST, o método service() chama o doPost().

Você continua mostrando o doGet() e o doPost() como se eles fossem os únicos... mas EU SEI que existem oito métodos no HTTP 1.1.



```
GET /select/selectBeerTaste.  
do?color=dark&taste=malty  
HTTP/1.1
```

```
Host: www.wickedlysmart.com
```

```
User-Agent: Mozilla
```

```
Mac OS X Mach-O
```

```
20030624 Netscape
```

```
Accept: text/xml
```

```
application/xhtml+xml;text
```

```
plain;q=0.8,video/
```

```
POST /select/selectBeerTaste2.do  
HTTP/1.1
```

```
Host: www.wickedlysmart.com
```

```
User-Agent: Mozilla/5.0 (Macintosh; U; PPC
```

```
Mac OS X Mach-O; en-US; rv:1.4) Gecko/
```

```
20030624 Netscape/7.1
```

```
Accept: text/xml,application/xml,application/
```

```
xml;q=0.9,video/
```

```
plain;q=0.8,video/x-mng,image/png,image/
```

```
jpeg,image/gif;q=0.2,*/*;q=0.1
```

```
Accept-Language: en-us,en;q=0.5
```

*solicitações HTTP*

## É bem capaz de você não se importar com os outros métodos HTTP, exceto o GET e o POST

Sim, *existem* outros Métodos HTTP 1.1 além do GET e POST. Temos também o HEAD, TRACE, OPTIONS, PUT, DELETE e CONNECT.

Todos, exceto um, têm um método doXXX() na classe HttpServlet. Ou seja, além do doGet() e doPost(), temos o doOptions(), doHead(), doTrace(), doPut() e doDelete(). Não existe nenhum mecanismo na servlet da API para tratar o doConnect(), então ele não faz parte do HttpServlet.

Mas, enquanto os outros Métodos HTTP talvez sejam importantes para, digamos, um desenvolvedor *web*, um desenvolvedor *servlet* raramente usará outro além do GET ou do POST.

Na maior parte do desenvolvimento servlet (provavelmente *todo*), você usará o doGet() (para solicitações simples) ou o doPost() (para aceitar e processar dados de formulários), e não terá que se preocupar com os outros.



Então, se eles não são importantes para mim... é CLARO que isso significa que eles cairão na prova.



## Na verdade, é possível que algum outro Método HTTP faça uma (rápida) aparição no exame...

Se você está se preparando para o exame, deve ser capaz de reconhecer todos eles e ter pelo menos uma idéia de suas funções. Mas não perca muito tempo aqui!

No mundo servlet de verdade, só interessam o GET e o POST.

Para o exame, vai interessar também um pouquinho dos outros Métodos HTTP.

<b>GET</b>	Pede para <i>obter</i> a coisa (recurso/arquivo) na URL requisitada.
<b>POST</b>	Pede para o servidor <i>aceitar</i> a informação do corpo anexada na solicitação, e a entrega para aquilo que consta na URL solicitada. É como um GET com mais calorias... um GET com informação extra enviada com a solicitação.
<b>HEAD</b>	Pede apenas a parte do <i>header</i> daquilo que o GET vai retornar. É como um GET sem corpo na resposta. Informa a URL requisitada sem, de fato, retornar a coisa.
<b>TRACE</b>	Solicita um loopback da mensagem de solicitação, para que o cliente veja o que está sendo recebido do outro lado, para teste ou troubleshooting.
<b>PUT</b>	Diz para <i>colocar</i> a informação anexada (o corpo) na URL requisitada.
<b>DELETE</b>	Diz para <i>apagar</i> a coisa (recurso/arquivo) na URL requisitada.
<b>OPTIONS</b>	Solicita uma <i>lista</i> dos métodos HTTP para os quais a coisa na URL requisitada pode responder.
<b>CONNECT</b>	Diz para <i>conectar</i> no caso de tunneling.

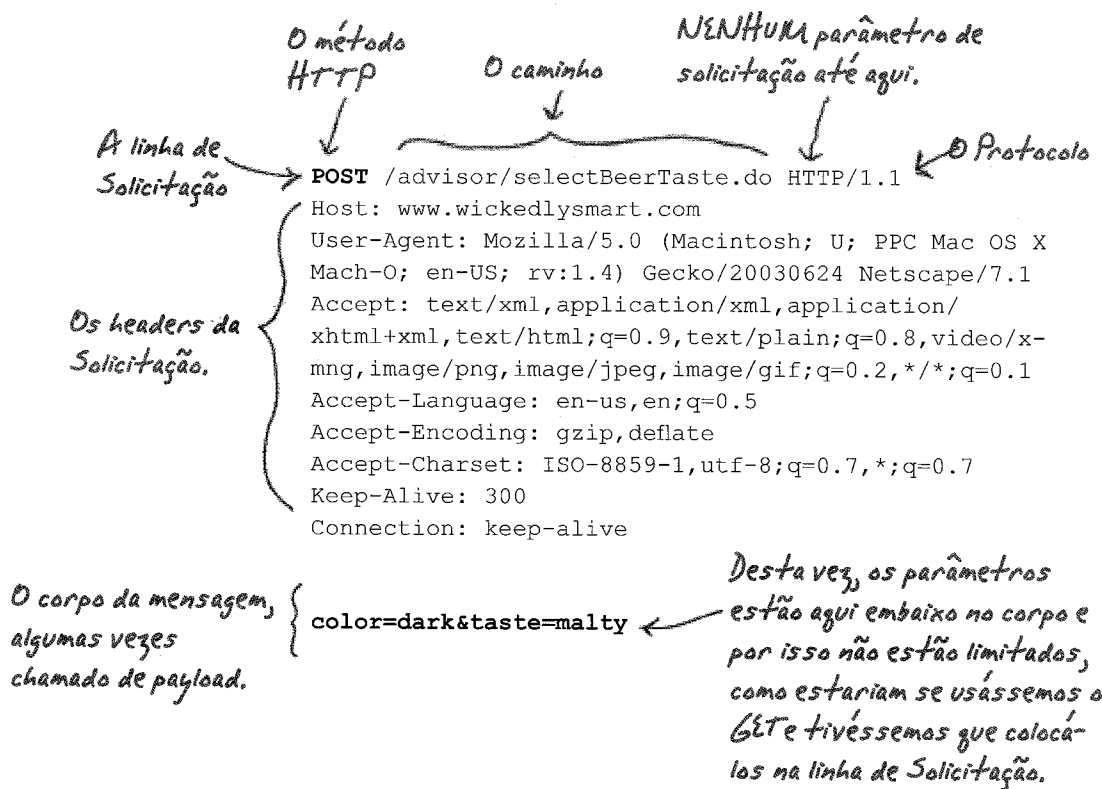
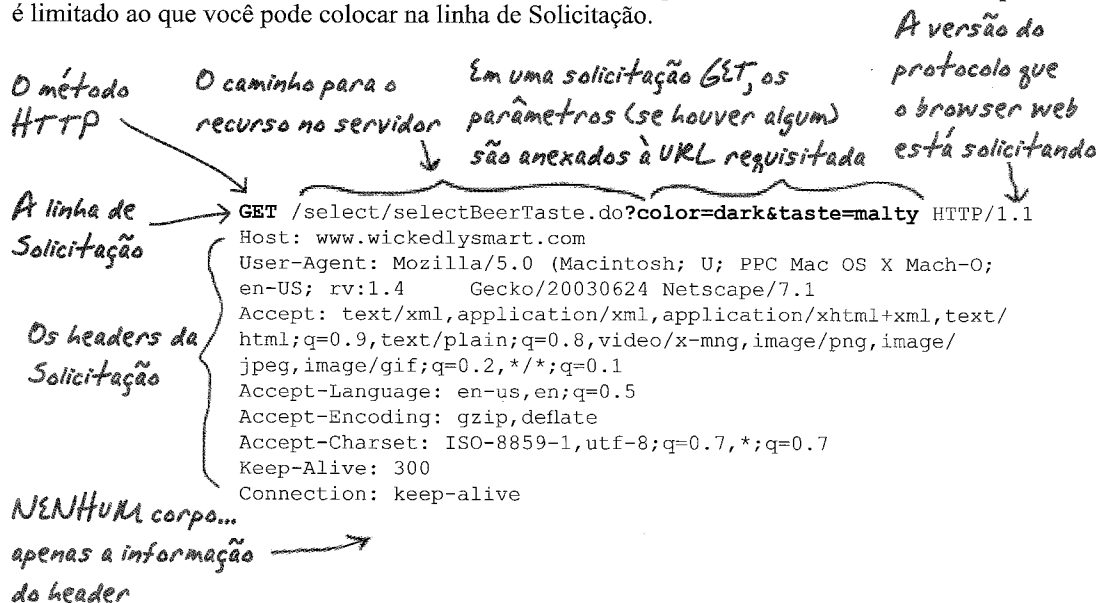
Exemplo de uma resposta para uma solicitação HTTP OPTIONS:

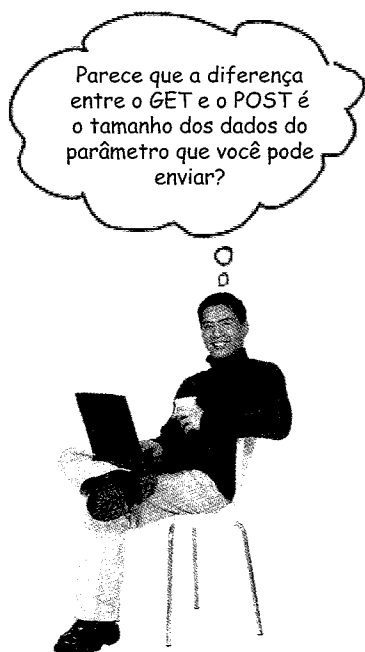


```
HTTP/1.1 200 OK
Server: Apache-
Coyote/1.1
Date: Thu, 20 Apr
2004 16:20:00 GMT
Allow: OPTIONS,
TRACE, GET, HEAD,
POST
Content-Length: 0
```

## A diferença entre GET e POST

O **POST** tem um corpo. Essa é a dica. Ambos podem enviar parâmetros, mas com o GET, o parâmetro é limitado ao que você pode colocar na linha de Solicitação.





## Não, não se trata só do tamanho

Nós falamos de outros problemas do GET no capítulo um, lembra? Quando você usa o GET, os dados do parâmetro aparecem na barra de endereços do browser, logo após a URL (e separados por um "?"). Imagine uma situação em que você não quisesse que os parâmetros fossem exibidos.



Então, segurança pode ser outro problema.

Outro problema poderia ser se você precisasse ou quisesse que os visitantes fizessem um bookmark da página. As solicitações GET aceitam bookmark; já as POST, não. Talvez seja realmente importante que sua página, digamos, permita aos usuários especificar critérios de busca. Os usuários podem querer retornar uma semana depois e fazer a mesma busca novamente, quando existirão novos dados no servidor.

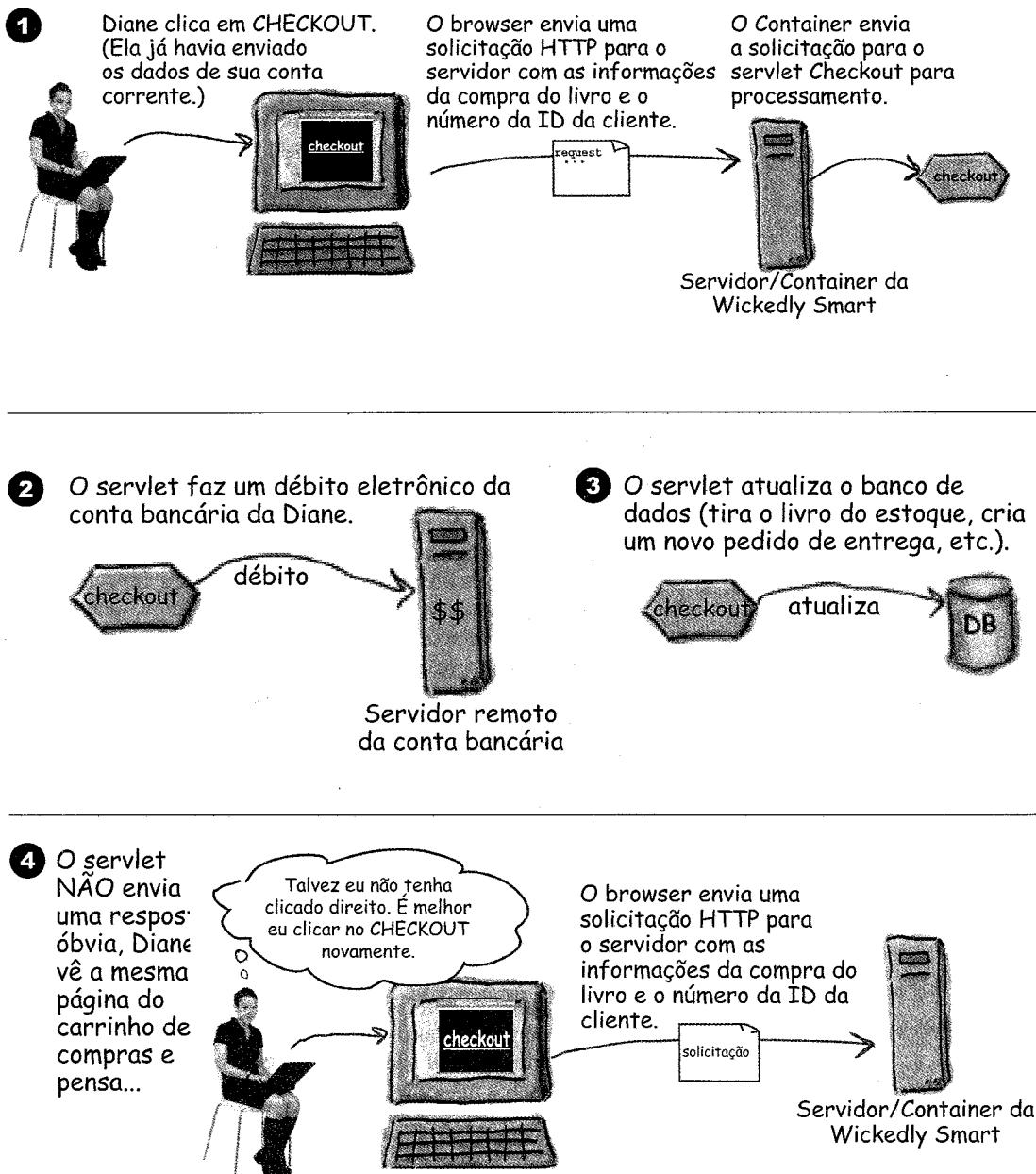
Mas *além* de tamanho, segurança e bookmark, existe outra diferença crucial entre o GET

e o POST: a maneira como *devem* ser usados. O GET deve ser usado para *obter* coisas. Ponto. Simplesmente receber. Claro, você deve usar os parâmetros para ajudar a descobrir o que enviar de volta, mas a questão é: você não está fazendo nenhuma mudança no servidor! O POST deve ser usado para *enviar dados para serem processados*. Isto pode ser tão simples como pesquisar parâmetros usados para descobrir o que será enviado de volta, assim como no GET. Mas quando você pensa em POST, você pensa em *atualização*. Você pensa: usar os dados do corpo do POST para *mudar alguma coisa no servidor*.

E isso traz um outro problema... se a solicitação é *idempotente*. Se *não* for, você pode estar diante de um problema que aquele pequeno comprimido azul não resolverá. Se você não está familiarizado com a maneira com que o termo "idempotente" é usado no mundo web, continue lendo...

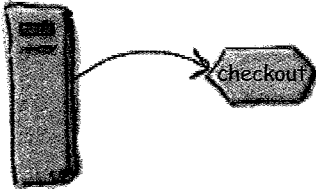
## A história da solicitação não-idempotente

**Diane tem uma necessidade.** Ela está tentando desesperadamente comprar o “Use a Cabeça Tricô” na livraria on-line Wickedly Smart que, sem que ela saiba, ainda está na fase de testes. Ela tem pouco dinheiro – apenas o suficiente para comprar *um* livro. Ela pensou em comprar direto do Amazon ou do O’Reilly.com, mas decidiu que queria uma cópia *autografada*, disponível apenas no Wickedly Smart. Decisão que ela futuramente viria a se arrepender...



## Nossa história continua...

- 5** O Container envia a solicitação para o servlet Checkout para processamento.

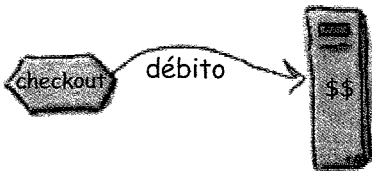


Servidor/Container da  
Wickedly Smart

- 6** O servlet não quer saber se Diane está comprando o mesmo livro novamente

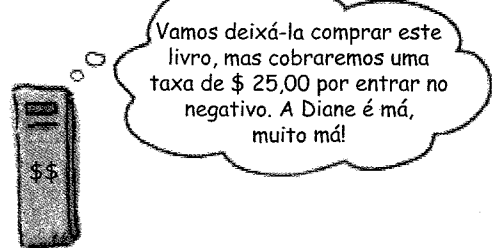


- 7** O servlet faz um débito eletrônico da conta bancária da Diane, pela segunda vez.



Servidor remoto da  
conta bancária

- 8** O banco de Diane aceita o débito, mas cobra uma tarifa por ultrapassar seu limite



Servidor remoto da  
conta bancária

- 9** Por fim, Diane acessa a página de Verificar Status do Pedido e percebe que tem DOIS pedidos para o livro de tricô...



- 10** Alô, é do banco? É que um programador idiota cometeu um erro e...





Aponte seu lápis

Quais dos métodos HTTP você acha que são (ou deveriam ser) idempotentes? (Baseado em seu prévio entendimento da palavra e/ou na história da compra duplicada de Diane que você acabou de ler.) As respostas estão no final desta página.

- ☐ GET
- ☐ POST
- ☐ PUT
- ☐ HEAD

(Deixamos o CONNECT de fora de propósito, visto que ele não faz parte do HttpServlet.)



## EXERCITE SUA MENTE

O que houve de errado com a transação da Diane?  
(E não foi apenas UMA coisa... provavelmente, o desenvolvedor terá que consertar diversos problemas.)

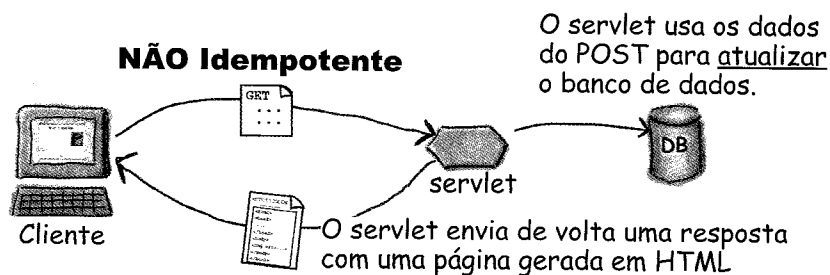
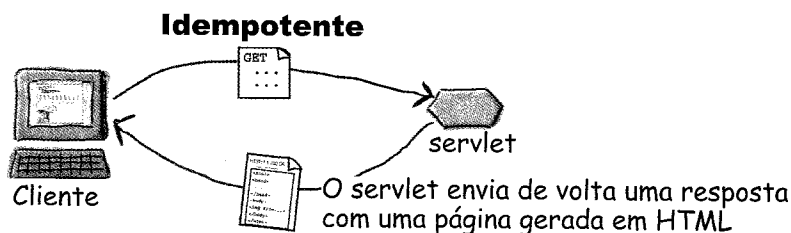
Quais seriam algumas formas do desenvolvedor reduzir riscos como esse?  
(Dica: talvez elas não sejam todas soluções de programação.)

deveria). O POST não é considerado idempotente pela especificação HTTP 1.1, que você POSSA escolher um método doGet() não-idempotente sozinho (mas não a especificação HTTP 1.1 declara GET, HEAD e PUT como idempotentes, ainda

Idempotência  
não é motivo de  
vergonha...



Ser idempotente é BOM. Significa que você pode fazer a mesma coisa repetidamente, sem os indesejáveis efeitos colaterais!



## O POST não é idempotente

O HTTP GET é usado apenas para *conseguir* coisas e não deve *mudar* nada no servidor. Então um GET é, por definição (e de acordo com a especificação HTTP), idempotente. Ele pode ser executado mais de uma vez, sem causar qualquer efeito colateral danoso.

O POST *não* é idempotente. Os dados submetidos no corpo de um POST podem ser destinados a uma transação que não pode ser desfeita. Portanto, tenha cuidado com a funcionalidade do seu doPost()!

O GET é idempotente. O POST, não. Cabe a você ter certeza de que a lógica da sua aplicação pode lidar com casos como o de Diane, em que o POST aparece mais de uma vez

O que vai me impedir de usar parâmetros no GET para atualizar o servidor?



O GET é sempre considerado idempotente no HTTP 1.1...

...mesmo que você encontre códigos no exame que usem parâmetros GET que causem efeitos colaterais! Ou seja, o **GET é idempotente de acordo com a especificação HTTP**. Mas não existe nada que possa impedi-lo de implementar um método doGet() não-idempotente no seu servlet. A solicitação GET dos clientes deve ser idempotente, ainda que a SUA manipulação dos dados cause um efeito negativo. Tenha sempre em mente a diferença entre o método HTTP GET e o método doGet() do seu servlet.

*Nota: há várias acepções para a palavra "idempotente". Nós estamos usando-a voltada para o HTTP/servlet para nos definir que a mesma solicitação pode ser feita duas vezes, sem nenhuma consequência negativa para o servidor. Nós \*não\* usamos "idempotente" para dizer que a mesma solicitação sempre retorna a mesma resposta e NEM para dizer que uma solicitação não tem NENHUM efeito colateral.*



## O que determina se o browser enviará uma solicitação GET ou POST?

### GET

Um hyperlink simples sempre significa GET

```
<A HREF="http://www.wickedlysmart.com/index.html/">click here</A>
```

### POST

Se você **DISSER** explicitamente `method=POST`, então, curiosamente, ele será **POST**

```
<form method="POST" action="SelectBeer.do">
  Select beer characteristics<p>
  <select name="color" size="1">
    <option>light
    <option>amber
    <option>brown
    <option>dark
  </select>
  <center>
    <input type="SUBMIT">
  </center>
</form>
```

Quando o usuário clica no botão "SUBMIT", os parâmetros são enviados no corpo da solicitação POST. Neste exemplo, há somente um parâmetro, chamado `color` e o valor é a <option> referente à cor da cerveja que o usuário selecionou (clara, âmbar, marrom ou escura).

## O que acontece se você **NÃO** disser `method="POST"` no seu <form>?

Desta vez, não há nenhum `method="POST"` aqui

```
<form action="SelectBeer.do">
  Select beer characteristics<p>
  <select name="color" size="1">
    <option>light
    <option>amber
    <option>brown
    <option>dark
  </select>
  <center>
    <input type="SUBMIT">
  </center>
</form>
```

AGORA, o que acontece com os parâmetros quando o usuário clica no botão SUBMIT se o formulário não possui um `method="POST"`?

## O POST NÃO é o padrão!

Se você não colocar **method="POST"** no seu formulário, o padrão será uma solicitação HTTP GET. Isso quer dizer que o browser envia os parâmetros no header da solicitação, mas este é o menor dos seus problemas. Em virtude de a solicitação chegar como GET, significa que você estará com um problemão no momento da execução, se você tiver no seu servlet apenas um `doPost()` e não um `doGet()`!

### Se você fizer isto:

↙ Nenhum "method=POST" no formulário HTML

```
<form action="SelectBeer.do">
```

### E depois isto:

```
public class BeerSelect extends HttpServlet {  
    public void doPost(HttpServletRequest request, HttpServletResponse response)  
        throws IOException, ServletException  
    {  
        // code here  
    }  
}
```

Nenhum método `doGet()` no servlet.

### Você terá isto:

**ERRO! Se o seu formulário HTML usa GET em vez de POST, você TEM que ter o `doGet()` na sua classe servlet. O método-padrão para formulários é GET.**

**P:** E se eu quiser suportar tanto GET como POST em um único servlet?

**R:** Os desenvolvedores que querem dar suporte a ambos os métodos geralmente colocam a lógica no `doGet()`, e fazem com que a implementação `doPost()` delegue poderes àquela `doGet()`:

```
doPost() method delegate to the  
doGet() method if necessary.  
public void doPost(...)  
    throws ... {  
    doGet(request, response);  
}
```

## Enviando e utilizando um único parâmetro

### Formulário HTML

```
<form method="POST" action="SelectBeer.do">
  Select beer characteristics<p>
  <select name="color" size="1">
    <option>light
    <option>amber
    <option>brown
    <option>dark
  </select>
  <center>
    <input type="SUBMIT">
  </center>
</form>
```

O browser enviará uma destas quatro opções na solicitação, associada com o nome color. Por exemplo, "color = amber".

### Solicitação HTTP POST

```
POST /advisor/SelectBeer.do HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4)
Gecko/20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

color=dark

Lembre-se, é o browser que gera esta solicitação; logo, você não precisa se preocupar em criá-la, mas é assim que ela se parece quando vem do servidor...

### Classe servlet

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
{
  String colorParam = request.getParameter("color");
  // mais códigos esclarecedores aqui...
}
```

(Neste exemplo, a String colorParam tem o valor "dark".)

Isto coincide com o nome no formulário

## Enviando e usando dois parâmetros

### Formulário HTML

```
<form method="POST" action="SelectBeerTaste.do">
  Select beer characteristics<p>
  COLOR:
  <select name="color" size="1">
    <option>light
    <option>amber
    <option>brown
    <option>dark
  </select>
  BODY:
  <select name="body" size="1">
    <option>light
    <option>medium
    <option>heavy
  </select>
  <center>
    <input type="SUBMIT">
  </center>
</form>
```

*O browser enviará uma destas quatro opções no corpo da solicitação para o parâmetro chamado color.*

*O browser enviará uma destas três opções na solicitação, associada com o nome body*

### Solicitação HTTP POST

```
POST /advisor/SelectBeerTaste.do HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4)
Gecko/20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

**color=dark&body=heavy**

*Agora, a solicitação POST possui ambos os parâmetros separados por um "&"*

### Classe servlet

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
{
  String colorParam = request.getParameter("color");
  String bodyParam = request.getParameter("body");
  // mais código aqui
}
```

*Agora, a variável String colorParam tem o valor "dark" e a bodyParam tem o valor "heavy".*



**Você pode ter vários valores para um único parâmetro! Isto significa que você precisará que o `getParameterValues()` retorne um array, e não um `getParameter()` que retorne uma `String`.**

### Veja isto!

*Alguns tipos de entrada de dados, como um grupo de checkboxes, podem ter mais de um valor. Isso quer dizer que um único parâmetro ("tamanhos", por exemplo) terá diversos valores, dependendo de quantos boxes o usuário assinalou. Um formulário em que o usuário possa selecionar diversos tamanhos de cerveja (indicando que ele está interessado em TODOS aqueles tamanhos) será algo como:*

```
<form method=POST
  action="SelectBeer.do">
  Select beer characteristics<p>
  Can Sizes: <p>
  <input type=checkbox name=sizes value="12oz"> 12 oz.<br>
  <input type=checkbox name=sizes value="16oz"> 16 oz.<br>
  <input type=checkbox name=sizes value="22oz"> 22 oz.<br>
  <br><br>

  <center>
    <input type="SUBMIT">
  </center>
</form>
```

*No seu código, você usará o método `getParameterValues()` que retorna um array:*

```
String one = request.getParameterValues("sizes")[0];

String [] sizes = request.getParameterValues("sizes");
```

*Se você quiser ver tudo no array, só por diversão ou teste, pode usar:*

```
String [] sizes = request.getParameterValues("sizes");
for(int x=0; x < sizes.length ; x++) {
    out.println("<br>sizes: " + sizes[x]);
}
```

*(Considere que "out" é um `PrintWriter` que você obteve da resposta)*

## Além dos parâmetros, o que mais eu posso obter de um objeto `Request`?

As interfaces `ServletRequest` e `HttpServletRequest` possuem uma tonelada de métodos que você pode chamar, mas você não precisa memorizar todos eles. Sozinho, você *realmente* deveria ver todas a API para `javax.servlet.ServletRequest` e `javax.servlet.http.HttpServletRequest`, mas aqui nós só veremos os métodos que você mais usará no seu trabalho (e que podem também aparecer no exame).

No mundo real, você estará com sorte (ou *sem* sorte, dependendo de sua perspectiva), se usar mais de 15% da API de solicitação. *Não se preocupe se não ficou claro para você como ou por que você usaria cada uma delas*; nós veremos mais detalhes de algumas delas (principalmente os cookies) mais adiante.

### A plataforma do cliente e a informação do browser

```
String client = request.getHeader("User-Agent");
```

### Os cookies associados a esta solicitação

```
Cookie[] cookies = request.getCookies();
```

### A sessão associada a este cliente

```
HttpSession session = request.getSession();
```

### O Método HTTP da solicitação

```
String theMethod = request.getMethod();
```

### Um stream de dados da solicitação

```
InputStream input = request.getInputStream();
```

#### Interface `ServletRequest` (`javax.servlet.ServletRequest`)

<code>&lt;&lt;interface&gt;&gt;</code> <b><code>ServletRequest</code></b>
<code>getAttribute(String)</code> <code>getLength()</code> <code>getInputStream()</code> <code>getLocalPort()</code> <code>getRemotePort()</code> <code>getServerPort()</code> <code>getParameter(String)</code> <code>getParameterValues(String)</code> <code>getParameterNames()</code> <i>// MUITOS outros métodos...</i>



#### Interface `HttpServletRequest` (`javax.servlet.http.HttpServletRequest`)

<code>&lt;&lt;interface&gt;&gt;</code> <b><code>HttpServletRequest</code></b>
<code>getContextPath()</code> <code>getCookies()</code> <code>getHeader(String)</code> <code>getIntHeader(String)</code> <code>getMethod()</code> <code>getQueryString()</code> <code>getSession()</code> <i>// MUITOS outros métodos...</i>

## Não existem Perguntas Idiotas

**P:** Por que algum dia eu iria *querer* obter uma `InputStream` da solicitação?

**R:** Com uma solicitação GET, não há nada além da informação header da solicitação. Em outras palavras, não há corpo com que se preocupar. MAS... com um HTTP POST, há informação de corpo. Na maioria das vezes, tudo o que interessa em relação ao corpo é retirar os valores dos parâmetros (por exemplo, "color=dark") usando o `request.getParameter()`, mas esses valores podem ser enormes. Se você quer analisar a fundo tudo o que chega com a solicitação, você pode usar o método `getInputStream()`. Com ele você pode, por exemplo, destrinchar todas as informações do header e processar byte a byte o payload (o corpo) da solicitação, copiando imediatamente para um arquivo no servidor, talvez.

**P:** Qual é a diferença entre `getHeader()` e `getIntHeader()`? Pelo que eu posso dizer, headers são *sempre Strings*! Até mesmo o método `getIntHeader()` leva uma `String` representando o nome do header; então, para que serve o *int*?

**R:** Os headers têm um *nome* (como "User-Agent" ou "Host") e um *valor* (como "Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/20030624 Netscape/7.1" ou "www.wickedlysmart.com"). Os valores retornados dos headers estão sempre no formato `String`, mas para alguns casos, a `String` representa um número. O header "Content-Length" retorna o número de bytes que compõe o corpo da mensagem. O header HTTP "Max-Forwards", por exemplo, retorna um valor inteiro, indicando quantos hops (saltos de roteadores) a solicitação pode fazer. (Você pode querer usar este header se estiver tentando fazer um trace da solicitação, que você suspeite estar presa em um loop em algum lugar.)

Você pode obter o valor do header "Max-Forwards" usando o `getHeader()`:

```
String forwards = request.getHeader("Max-Forwards");
int forwardsNum = Integer.parseInt(forwards);
```

E isso funciona perfeitamente. Mas se você *souber* o valor que o header deve assumir como `int`, você pode usar o `getIntHeader()` como um método de *conveniência* para poupá-lo da etapa de conversão da `String` para `int`:

```
int forwardsNum = request.getIntHeader("Max-Forwards");
```



### `getServerPort()`, `getLocalPort()` e `getRemotePort()` são confusos!

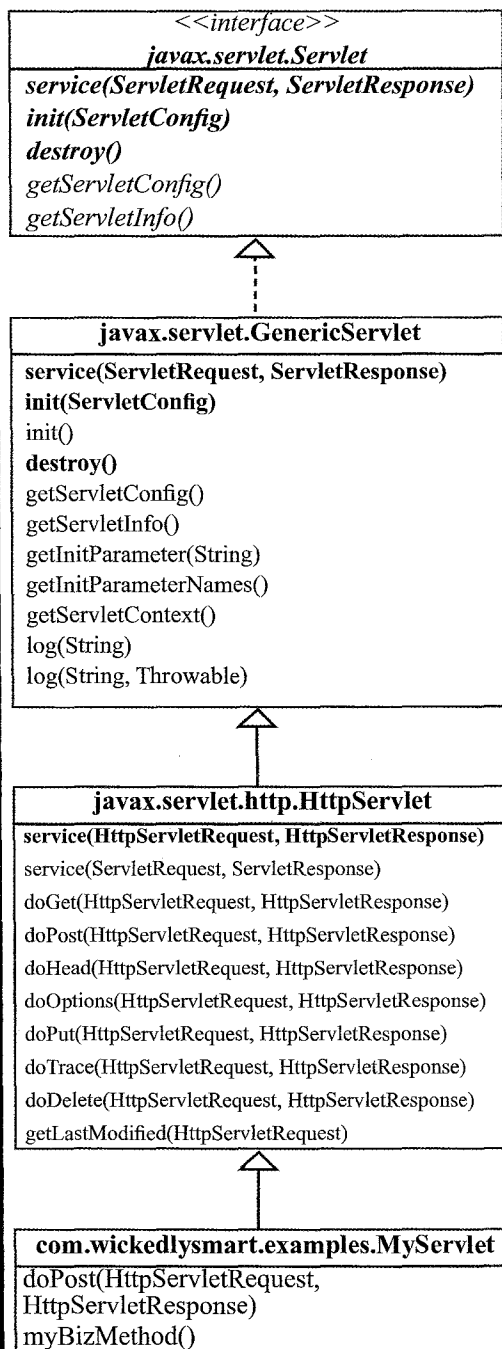
O `getServerPort()` deveria ser auto-explicativo... até que você se perguntasse para que serve então o `getLocalPort()`. Vamos começar pelo mais fácil: `getRemotePort()`. Primeiro você vai perguntar "remoto em relação a quem?" Neste caso, já que é o servidor quem solicita, remoto é o **CLIENTE**. O cliente é remoto na visão do servidor. Logo, `getRemotePort()` significa "obter a porta do **cliente**". Ou seja, o número da porta do cliente de onde partiu a solicitação. Lembre-se: se você for um **servlet**, **remoto** significa **cliente**.

A diferença entre `getLocalPort()` e `getServerPort()` é mais sutil: o `getServerPort()` diz "para qual porta a solicitação foi inicialmente **ENVIADA**?", enquanto que o `getLocalPort()` diz "em qual porta a solicitação **FOI PARAR**". Sim, tem uma diferença, porque embora as solicitações sejam **enviadas** para uma única porta (a qual o **servidor** está escutando), o servidor encontra uma porta local **diferente** para cada **thread**, para que a aplicação possa atender a vários clientes ao mesmo tempo.

## Revisão: Ciclo da vida do servlet e API

### Pontos de bala

- O Container inicializa um servlet carregando a classe, invocando o construtor-padrão do servlet e chamando o método `init()` do servlet.
- O método `init()` (que o desenvolvedor pode anular) é chamado apenas uma vez no ciclo de vida do servlet, e sempre antes do servlet atender a qualquer solicitação do cliente.
- O método `init()` dá ao servlet acesso para os objetos `ServletConfig` e `ServletContext`, que o servlet precisa para conseguir informações sobre a configuração do servlet e a aplicação web.
- O Container termina com a vida de um servlet chamando seu método `destroy()`.
- O servlet passa a maior parte da sua vida rodando um método `service()` para uma solicitação do cliente.
- Cada solicitação para um servlet roda em uma thread separada! Só existe apenas uma instância para qualquer classe servlet.
- Seu servlet quase sempre estenderá o `javax.servlet.http.HttpServlet`, do qual ele herda uma implementação do método `service()`, que traz um `HttpServletRequest` e um `HttpServletResponse`.
- O `HttpServlet` estende o `javax.servlet.GenericServlet` – uma classe abstrata que implementa a maioria dos métodos básicos do servlet.
- O `GenericServlet` implementa a interface `Servlet`.
- As classes servlet (exceto aquelas relacionadas aos JSPs) estão em um dos dois pacotes: `javax.servlet` ou `javax.servlet.http`.
- Você pode anular o método `init()` e deve anular pelo menos um método de serviço (`doGet()`, `doPost()`, etc).





## Revisão: HTTP e HttpServletRequest

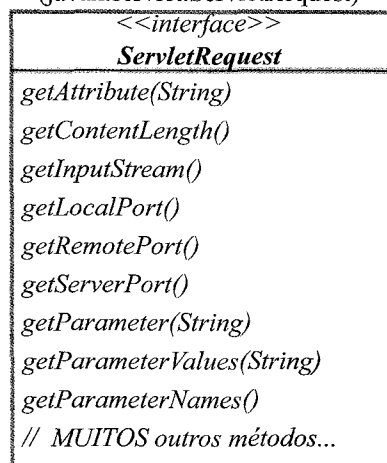
### Pontos de bala



- Os métodos `doGet()` e `doPost()` do `HttpServlet` levam um `HttpServletRequest` e um `HttpServletResponse`.
- O método `service()` determina se o `doGet()` ou o `doPost()` rodará, baseado no método HTTP (GET, POST, etc.) da solicitação HTTP.
- As solicitações POST têm um corpo; as solicitações GET, não, mas as solicitações GET podem ter parâmetros anexados à URL da solicitação (algumas vezes chamada “query string”).
- As solicitações GET são idempotentes por herança (de acordo com a especificação HTTP). Elas devem ser capazes de rodar várias vezes, sem causar nenhum efeito colateral no servidor. As solicitações GET não devem *mudar* nada no servidor. Mas você *pode* escrever um método `doGet()` não-idempotente e maldoso.
- O POST é não-idempotente por herança e cabe a você projetar e codificar sua aplicação, de forma que, se o cliente enviar uma solicitação duas vezes por engano, você possa cuidar disso.
- Se um formulário HTML não diz explicitamente “method=POST”, a solicitação é enviada como um GET e não como POST. Se você não possui um `doGet()` em seu servlet, a solicitação falhará.
- Você pode receber parâmetros da solicitação com o método `getParameter(“paramname”)`. O resultado é sempre uma `String`.
- Se você tem múltiplos valores de parâmetros para um determinado parâmetro, use o método `getParameterValues(“paramname”)` que retorna um array de `Strings`.
- Você pode obter *outras* coisas do objeto solicitação, como headers, cookies, uma sessão, a query string e um stream de dados.

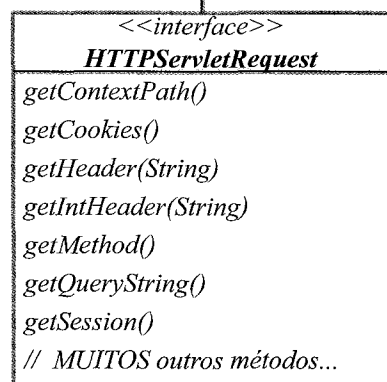
### Interface `ServletRequest`

(`javax.servlet.ServletRequest`)



### Interface `HttpServletRequest`

(`javax.servlet.http.  
HttpServletRequest`)



## Então, essa foi a Solicitação... vejamos agora a Resposta

A resposta é o que volta para o cliente. Aquilo que o browser recebe, analisa e retribui ao usuário. Tipicamente, você usa o objeto response para conseguir um stream de saída (geralmente um `Writer`), e você usa este stream para escrever o HTML (ou outro tipo de conteúdo) que retorna para o cliente. Contudo, o objeto response tem outros métodos além do I/O output. Veremos alguns deles com mais detalhes.

**interface `ServletResponse`**  
(`javax.servlet.  
ServletResponse`)

```
<<interface>>
ServletResponse
getBufferSize()
setContentTypes()
getOutputStream()
getWriter()
setContentLength()
//MUITOS outros métodos...
```

*Estes são alguns  
dos métodos mais  
geralmente usados.*

**interface `HttpServletResponse`**  
(`javax.servlet.http.  
HttpServletResponse`)

```
<<interface>>
HttpServletResponse
addCookie()
addHeader()
encodeURL()
sendError()
setStatus()
sendRedirect()
//MUITOS outros métodos...
```

*Às vezes você verá  
estes também...*

Na maioria das vezes, você usa a Resposta apenas para enviar dados de volta para o cliente. Você chama dois métodos na resposta: `setContentTypes()` e `getWriter()`. Depois disso, você estará fazendo simplesmente I/O para escrever o HTML (ou algo mais) no stream. Mas você também pode usar a resposta para configurar outros headers, enviar erros e adicionar cookies.

Espera um momento... Eu  
pensei que não íamos enviar  
HTML de um servlet,  
porque é horrível formatá-  
lo para o stream de saída...



## Usando a resposta para o I/O

Tudo bem, deveríamos estar usando JSPs em vez de enviar HTML de volta no stream de saída a partir de um servlet. Formatar um HTML para enfiá-lo no método `println()` do stream de saída é *penoso*.

Mas isso não significa que você nunca terá que trabalhar com um stream de saída do seu servlet.

Por que?

- 1) Seu provedor de hospedagem pode não suportar JSPs. Existem vários servidores e containers mais antigos por aí que suportam servlets, mas não JSPs, então, você fica “preso”.
- 2) Você não tem a opção de usar JSPs por algum outro motivo, como um gerente chato que não permite usar JSPs porque em 1998 seu cunhado lhe dissera que os JSPs eram ruins.
- 3) Quem disse que *HTML* era a única coisa que você poderia enviar de volta em uma resposta? Você pode devolver *outras* coisas em vez de HTML ao cliente. Algo para o qual um stream de saída faça sentido.

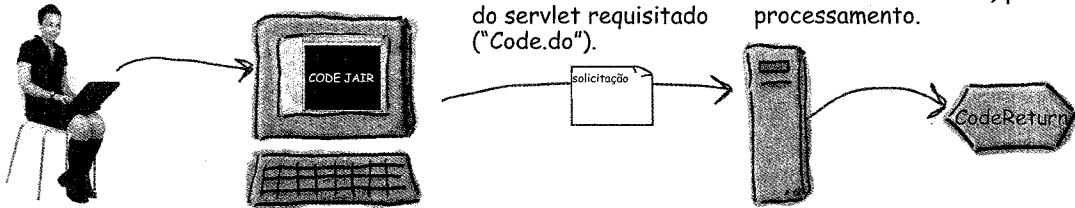
# Imagine que você queira enviar um JAR para o cliente...

Digamos que você tenha criado uma página para downloads onde o cliente pode baixar códigos a partir de arquivos JAR. Em vez de enviar de volta uma página HTML, a resposta contém os bytes representando o JAR. Você *lê* os bytes dos arquivos JAR e os *escreve* no stream de saída dos dados da resposta.

- 1 Diane está desesperada para fazer o download do JAR com um código do livro que ela está usando para aprender servlets e JSPs. Ela acessa o site do livro e clica no link "código jar", que se refere a um servlet chamado "Code.do".

O browser envia uma solicitação HTTP para o servidor com o nome do servlet requisitado ("Code.do").

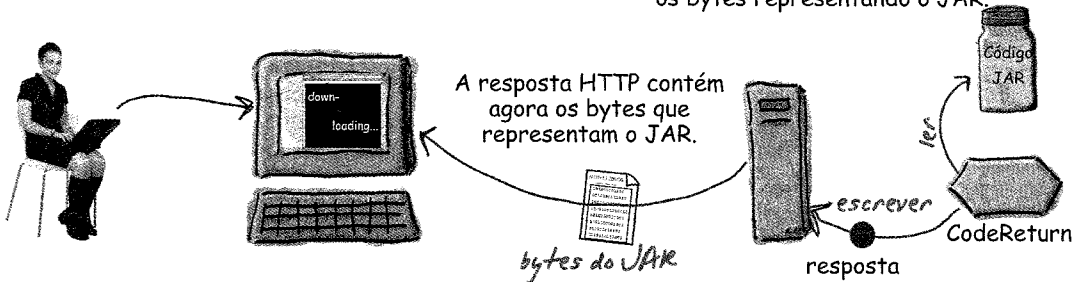
O Container envia a solicitação para o servlet CodeReturn (mapeado para o nome "Code.do" no DD) para processamento.



- 2 O JAR inicia o download na máquina do cliente. Diane está satisfeita.

O servlet CodeReturn recebe os bytes para o JAR, recebe da resposta um stream de saída e copia os bytes representando o JAR.

A resposta HTTP contém agora os bytes que representam o JAR.



## O código servlet que faz o download do JAR

// um monte de imports aqui

```
public class CodeReturn extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {

        response.setContentType("application/jar");

        ServletContext ctx = getServletContext();
        InputStream is = ctx.getResourceAsStream("/bookCode.jar");

        int read = 0;
        byte[] bytes = new byte[1024];

        OutputStream os = response.getOutputStream();
        while ((read = is.read(bytes)) != -1) {
            os.write(bytes, 0, read);
        }
        os.flush();
        os.close();
    }
}
```

*Nós queremos que o browser reconheça que isto é um JAR, não um HTML; então, configuramos o tipo de conteúdo para application/jar*

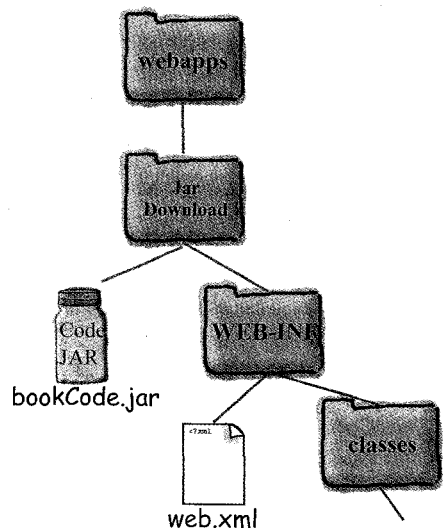
*Isto diz apenas "dê-me um stream de entrada para o recurso chamado bookCode.jar"*

*Esta é a parte fundamental, mas é apenas um I/O simples!! Nada especial, somente lê os bytes do JAR e os escreve para o stream de saída que nós conseguimos do objeto resposta.*

## Não existem Perguntas Idiotas

**P:** Onde estava localizado o arquivo JAR "bookCode.jar"? Em outras palavras, onde o método `getResourceAsStream()` PROCURA para localizar o arquivo? Como você lida com o caminho?

**R:** O `getResourceAsStream()` requer que você inicie com uma barra ("/"), que representa a raiz da sua aplicação. Já que a aplicação foi chamada de **JarDownload**, então a estrutura de diretórios é semelhante à figura. O diretório **JarDownload** está dentro de **webapps** (o diretório-pai para todos os outros diretórios da aplicação), e dentro do **JarDownload** nós colocamos o **WEB-INF** e o código JAR em si. Assim, o arquivo "bookCode.jar" está situado no nível raiz da aplicação **JarDownload**. (Não se preocupe, nós entraremos em maiores detalhes sobre a estrutura de diretórios de distribuição quando chegarmos no capítulo que trata da distribuição.)

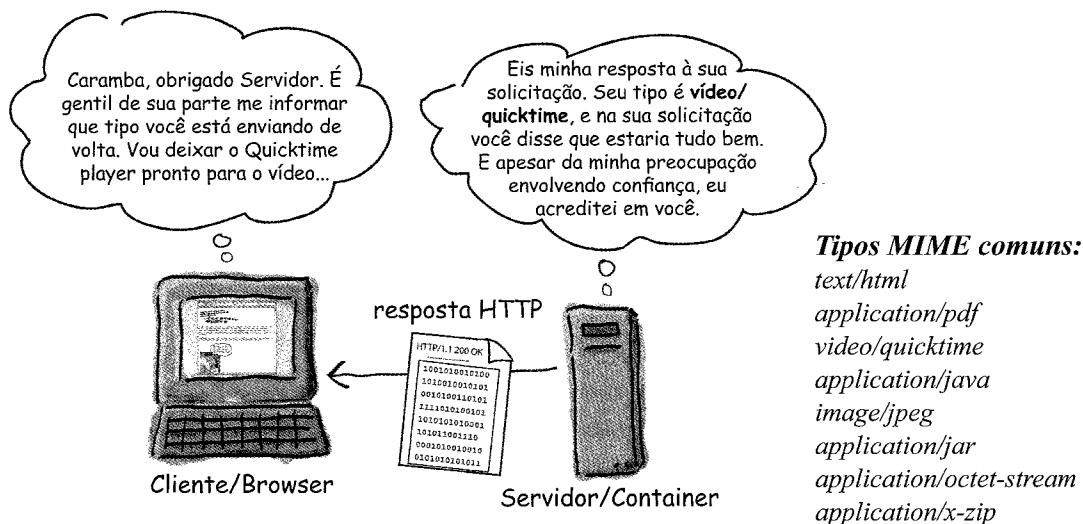


## Opa. Qual é o problema com o tipo do conteúdo?


Você deve estar surpreso com esta linha:

```
response.setContentType("application/jar");
```

Ou pelo menos *deveria*. Você tem que dizer ao browser o que você está devolvendo, para que ele possa **fazer a coisa certa**: abrir uma aplicação “assistente”, como um leitor PDF ou um player de vídeo, criar um HTML, salvar os bytes da resposta como um arquivo de download, etc. E já que você está aí se perguntando, sim, quando nós dizemos *tipo do conteúdo*, queremos dizer a mesma coisa que tipo MIME. O tipo do conteúdo é um header HTTP que *deve* ser incluído na resposta HTTP.



### Você não precisa memorizar um monte de tipos do conteúdo.

 **Relaxe** Você deve saber o que o `setContentType()` faz e como usá-lo, mas você não precisa saber nem mesmo os mais comuns tipos de conteúdo, exceto o `text/html`. O que você precisa saber sobre o `setContentType()` geralmente é de senso comum... por exemplo, não será bom para você mudar o tipo de conteúdo **DEPOIS** de ter escrito o que vai para o stream de saída da resposta. Óbvio. Isso quer dizer que você não pode configurar um tipo de conteúdo, escrever algo e, em seguida, mudar o tipo de conteúdo e escrever outra coisa. Pense um pouco: como o browser lida com isso? Ele só pode tratar um tipo de COISA da resposta de cada vez.

Para ter certeza de que tudo funciona bem, o que você deveria fazer (e em alguns casos é um requisito) é sempre chamar o `setContentType()` primeiro, **ANTES** de chamar o método que gera o stream de saída (`getWriter()` ou `getOutputStream()`). Isto garantirá que você não terá conflitos entre o tipo de conteúdo e o stream de saída de dados.

## Não existem Perguntas Idiotas

**P:** Por que você tem que definir o tipo de conteúdo? Os servidores não podem descobrir isso a partir da extensão do arquivo?

**R:** A maioria dos servidores *pode*, para conteúdo estático. No Apache, por exemplo, você pode configurar os tipos MIME mapeando a extensão do arquivo (.txt, .jar, etc.) para um tipo de conteúdo específico, e ele usará isso para configurar o tipo de conteúdo no header HTTP. Mas estamos falando sobre o que acontece dentro de um servlet quando NÃO HÁ nenhum arquivo! É você quem está enviando de volta a resposta; o Container não tem idéia do que você está enviando.

**P:** E com respeito ao último exemplo onde você lê um arquivo JAR? O Container não pode ver que você está lendo um JAR?

**R:** Não. Tudo o que fizemos do servlet foi ler os bytes de um arquivo (por acaso era um arquivo JAR), e escrever de volta esses dados para um stream de saída. O Container não faz idéia do que fazíamos quando líamos aqueles bytes. Pelo o que ele saiba, nós estávamos lendo a partir de um tipo de informação e escrevendo algo completamente diferente na resposta.

**P:** Como eu posso descobrir quais são os tipos comuns de conteúdo?

**R:** Faça uma busca no Google. Sério. Novos tipos MIME estão sendo adicionados todo o tempo, mas você pode encontrar facilmente listas na Web. Você também pode dar uma olhada nas suas preferências do browser por uma lista daqueles que foram configurados para seu browser, e também pode checar seus arquivos de configuração. Novamente, você não precisa se preocupar com isto para o exame e provavelmente não lhe causará muito estresse no mundo real também.

**P:** Espere um segundo... por que você precisaria usar um servlet para enviar de volta aquele arquivo JAR, quando você pode ter o servidor enviando-o como um recurso? Em outras palavras, por que não fazer com que o usuário clique em um link que vá para o JAR em vez do servlet? O servidor não pode ser configurado para enviar o JAR diretamente sem sequer PASSAR pelo servlet?

**R:** Sim. Boa pergunta. Você PODERIA configurar o servidor de forma que o usuário clique em um link HTML que o leve, digamos, ao arquivo JAR localizado no servidor (assim como qualquer outro recurso estático, como JPEGs e arquivos de texto), e o servidor simplesmente o enviaria na resposta.

Mas... estamos considerando que você tenha outras coisas que queira fazer no servlet ANTES de enviar de volta o stream. Você pode, por exemplo, precisar que a lógica no servlet que define *qual* arquivo JAR a enviar. Ou você pode estar devolvendo bytes que você esteja criando ali mesmo, na hora. Imagine um sistema em que você recebe parâmetros do input do usuário e os usa para gerar dinamicamente um som que você devolve. Um som que não existia antes. Ou seja, um arquivo de som que não se encontra em lugar algum no servidor. Você acaba de criá-lo e agora está enviando-o na resposta.

Então você está certo, talvez o nosso exemplo de enviar apenas um JAR localizado no servidor seja meio manjado, mas por favor... use sua imaginação e enfeite-o com tudo aquilo que você puder adicionar para torná-lo um servlet de *verdade*. Talvez seja algo tão simples como inserir um código no seu servlet que – junto com a devolução do JAR – escreva alguma informação no banco de dados sobre este usuário em particular. Ou talvez você tenha que checar para ver se ele está liberado para fazer o download deste JAR, baseado em alguma coisa que você tenha detectado no banco de dados.

## Você tem duas opções para saída: caracteres ou bytes

Isto é apenas um simples java.io, exceto pela interface ServletResponse que oferece apenas *duas* opções de streams para escolher: ServletOutputStream para bytes ou PrintWriter para dados em caracteres.

### ► **Printwriter**

Exemplo:

```
PrintWriter writer = response.getWriter();  
  
writer.println("some text and HTML");
```

Usado para:

Exibir dados de texto para um stream de caracteres. Embora você ainda possa exibir dados em caractere usando o OutputStream, é ele que você utilizará para tratar seus dados em caractere.

### ► **OutputStream**

Exemplo:

```
ServletOutputStream out = response.  
getOutputStream();  
  
out.write(aByteArray);
```

Usado para:

Escrever qualquer outra coisa!



### Você DEVE memorizar estes métodos

*Você tem que saber isso para o exame. É uma armadilha. Repare que para escrever para um ServletOutputStream você **write()**, porém, para escrever para um **PrintWriter** você... **println()**! É comum considerar que você escreve para um escritor, mas não. Se você já usa o java.io, já passou por isso. Se não, lembre-se:*

*println() para um **PrintWriter**  
write() para um **ServletOutputStream***

*E não se esqueça de que os nomes dos métodos para obter o stream ou o escritor perdem a primeira palavra, conforme segue:*

```
ServletOutputStream  
response.getOutputStream();
```

```
PrintWriter  
response.getWriter();
```

*Você precisa reconhecer nomes FALSOS, como:*

```
getPrintWriter()  
getResponseStream()  
getStream()  
getOutputStream();
```

*Estes NÃO existem!*

*Para Sua Informação: O **PrintWriter** na verdade empacota o **ServletOutputStream**. Ou seja, o **PrintWriter** tem uma referência para o **ServletOutputStream** e delega os chamados para ele. Apenas UM stream de saída de dados volta para o cliente, mas o **PrintWriter** o embeleza adicionando métodos de alto nível capazes de tratar os caracteres.*



## Você pode configurar headers de resposta, você pode adicionar headers de resposta

E você pode querer saber qual é a diferença. Mas pense nisso por um segundo e faça o exercício.

### Correlacione a chamada ao método com o seu comportamento

Trace uma linha do método `HttpResponse` para o seu comportamento. Nós fizemos o mais óbvio para você.

```
response.setHeader("foo", "bar");
```

```
response.addHeader("foo", "bar");
```

```
response.setIntHeader("foo", 42);
```

Adiciona um novo header e um novo valor na resposta, ou acrescenta um outro valor para um header existente.

Um método de conveniência que substitui o valor de um header existente por seu valor integral, ou acrescenta um novo header e um novo valor na resposta.

Se um header com este nome já existir na resposta, o valor é alterado por este. Ou então, acrescenta um novo header e um novo valor na resposta.

Bem fácil quando você os vê todos juntos.

Mas para a prova, você deve memorizá-los, pois se na próxima terça-feira o cara no final do corredor perguntar:

“Qual é o método de resposta que me permite acrescentar um valor para um header existente?”, você possa dizer, sem pestanejar: “É o `addHeader`, e ele possui duas Strings, para nome e valor.” Assim mesmo.

O `setHeader()` e o `addHeader()` acrescentarão um header e um valor na resposta, se o header (o primeiro argumento para o método) ainda não estiver lá. A diferença entre configurar e adicionar aparece quando o header *está* lá. Nesse caso:

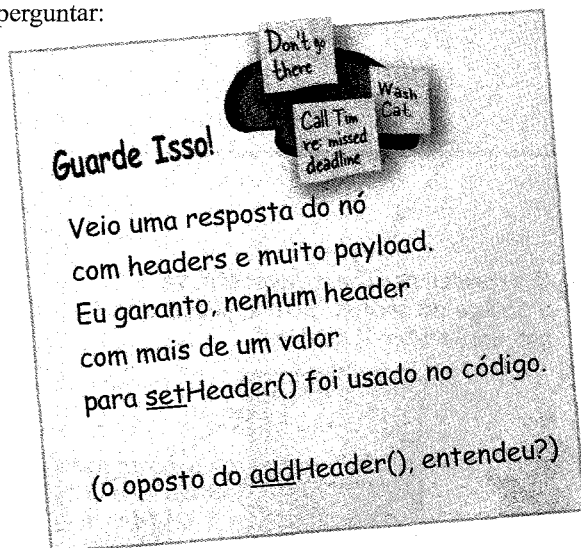
**`setHeader()` sobrescreve o valor existente**  
**`addHeader()` adiciona um novo valor**

Quando você chama o `setContentType` (“text/html”), você está configurando um header como se dissesse:

```
setHeader("content-type", "text/html");
```

Então, qual é a diferença? Nenhuma...

*considerando que você digite corretamente o header “content-type”.* O método `setHeader()` não vai reclamar se você escrever errado o nome dos headers – ele simplesmente acha que você está adicionando um outro tipo de header. Porém, algo dará errado lá na frente, já que agora você não configurou corretamente o tipo de conteúdo da resposta!

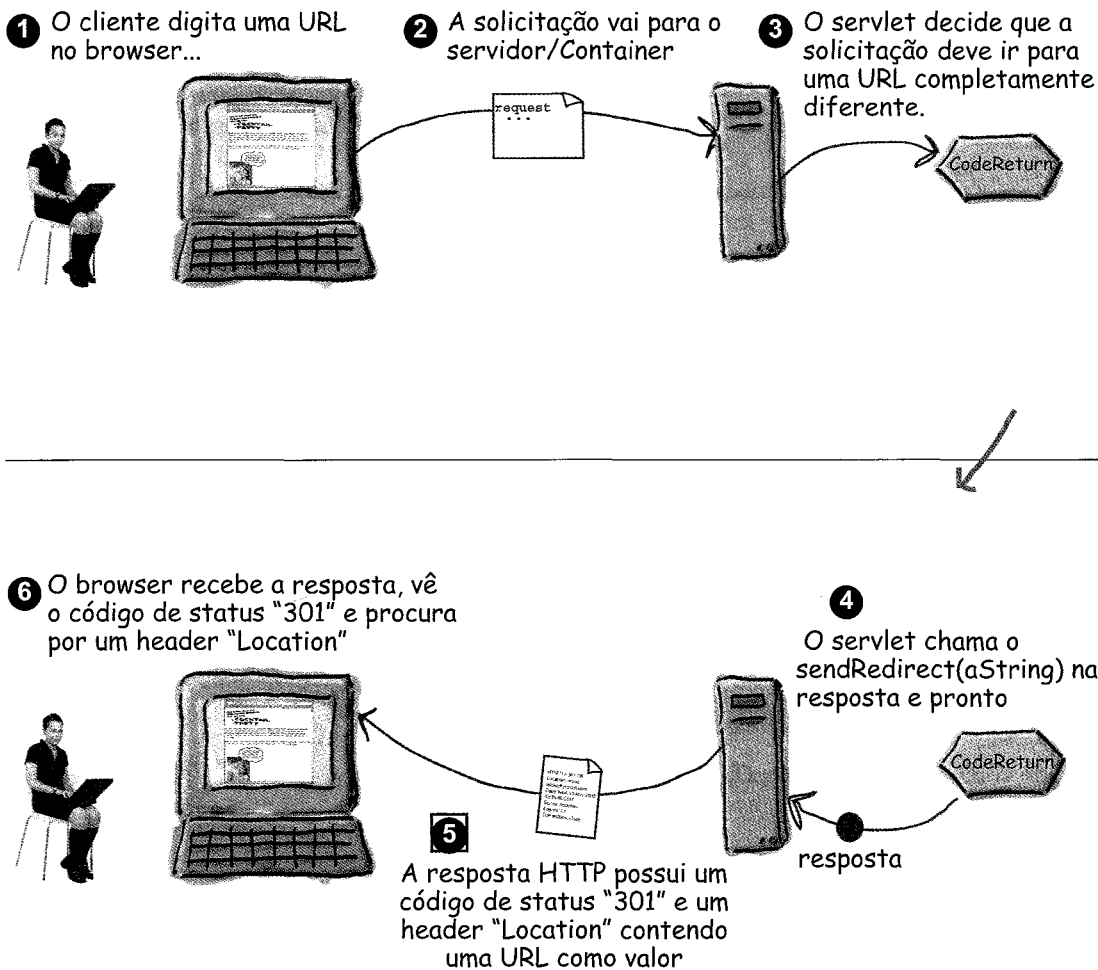


*(A primeira pessoa que nos enviar um arquivo mp3 recitando este poema, no ritmo certo e tudo mais, ganha uma camiseta especial.)*

# Mas algumas vezes você mesmo não quer lidar com a resposta...

Você pode decidir que algo diferente trate a resposta para a sua solicitação. Você pode ou *redirecionar* a solicitação para uma URL completamente diferente, ou *despachá-la* para algum outro componente da sua aplicação (geralmente um JSP).

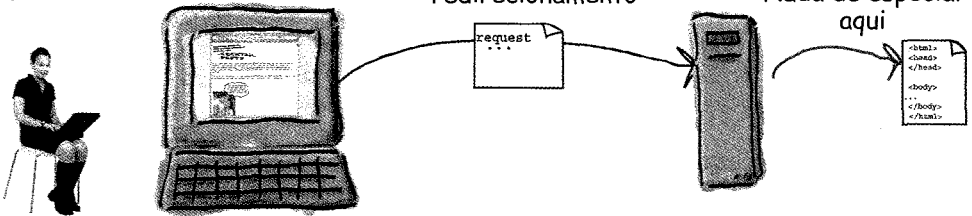
## Redirecionando



**7** O browser faz uma nova solicitação usando a URL que representava o valor do header "Location" na resposta anterior. O usuário pode notar que a URL na barra de endereços do seu browser mudou...

**8** Não há nada de exclusivo na solicitação, ainda que ela tenha sido ativada por um redirecionamento

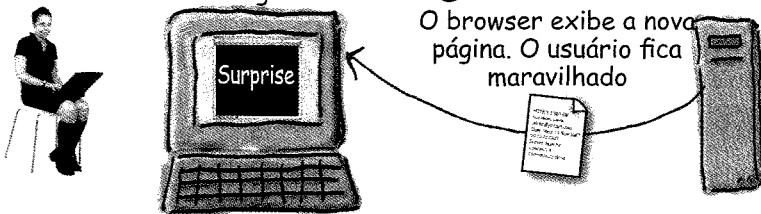
**9** O servidor recebe o que consta na URL solicitada. Nada de especial aqui



**11** A resposta HTTP é igual a qualquer outra resposta... exceto pelo fato de que ela não vem do local que o cliente digitou

Como eu vim parar aqui?

**10** O browser exibe a nova página. O usuário fica maravilhado



## O redirecionamento do Servlet faz o browser trabalhar

O redirecionamento deixa o servlet completamente aliviado. Após perceber que ele não poderá desempenhar o trabalho, o servlet simplesmente chama o método `sendRedirect()`:

```
if (worksForMe) {
    // trata a solicitação
} else {
    response.sendRedirect("http://www.oreilly.com");
}
```



*A URL que você quer que o browser use para a solicitação. É isso que o cliente vê*

## Usando URLs relativas no `sendRedirect()`

Você pode usar uma URL *relativa* como um argumento para o `sendRedirect()`, em vez de especificar o endereço completo “http://www...”. As URLs relativas vêm em dois sabores: com ou sem a barra (“/”) inicial.

Imagine que o cliente digitou primeiramente:

```
http://www.wickedlysmart.com/myApp/cool/bar.do
```

Quando a solicitação chega ao servlet chamado “bar.do”, ele chama o `sendRedirect()` com a URL relativa que NÃO inicia com a barra:

```
sendRedirect("foo/stuff.html");
```

O Container constrói a URL completa (ele precisa dela para o header “Location” que ele coloca na resposta HTTP), relativa à URL da solicitação inicial:

```
http://www.wickedlysmart.com/myApp/cool/foo/
stuff.html
```

*O Container sabe que a URL de solicitação original partiu do caminho myApp/cool; portanto, se você não usar a barra, esta parte do caminho é acrescentada ao começo de foo/stuff.html.*

Mas se o argumento para o `sendRedirect()` INICIAR com a barra:

```
sendRedirect("/foo/stuff.html");
```

O Container constrói a URL completa relativa ao container web em si, em vez de relativa à URL original da solicitação. Assim, a nova URL será:

```
http://www.wickedlysmart.com/foo/stuff.html
```



*“foo” é um aplicativo Web, separado do aplicativo web “myApp”.*

*A barra no começo significa relativa à raiz do container web.*



Veja isto!

**Você não pode  
fazer um  
sendRedirect()  
depois de escrever  
a resposta!**

*Isto provavelmente é óbvio, mas é a LEI e nós estamos apenas ratificando.*

*Se você procurar o `sendRedirect()` na API, você verá que ele envia uma `IllegalStateException` se você tentar chamá-lo depois "da resposta já ter sido criada".*

*Eles querem dizer com "criada" que a resposta foi **enviada**. Isso significa que os dados foram enviados para o stream.*

*Na prática, significa que **você não pode escrever na resposta e depois chamar o `sendRedirect()`**!*

*Mas um professor exigente diria que, tecnicamente, você pode escrever no fluxo de dados sem que haja o flush, e o `sendRedirect()` não geraria uma exceção.*

*Mas isto seria algo completamente estúpido e, por isso, não vamos tocar no assunto. (Se bem que... acabamos de falar...)*

*No seu servlet, decida, pelo amor de Deus! Ou trate a solicitação, ou faça um `sendRedirect()` para que OUTRA entidade a trate.*

*(Aliás, este papo de "uma vez enviada, já era" também se aplica à configuração de headers, cookies, códigos de status, o tipo de conteúdo e assim por diante...)*



**O `sendRedirect()`  
carrega um objeto  
String e NÃO uma  
URL!**

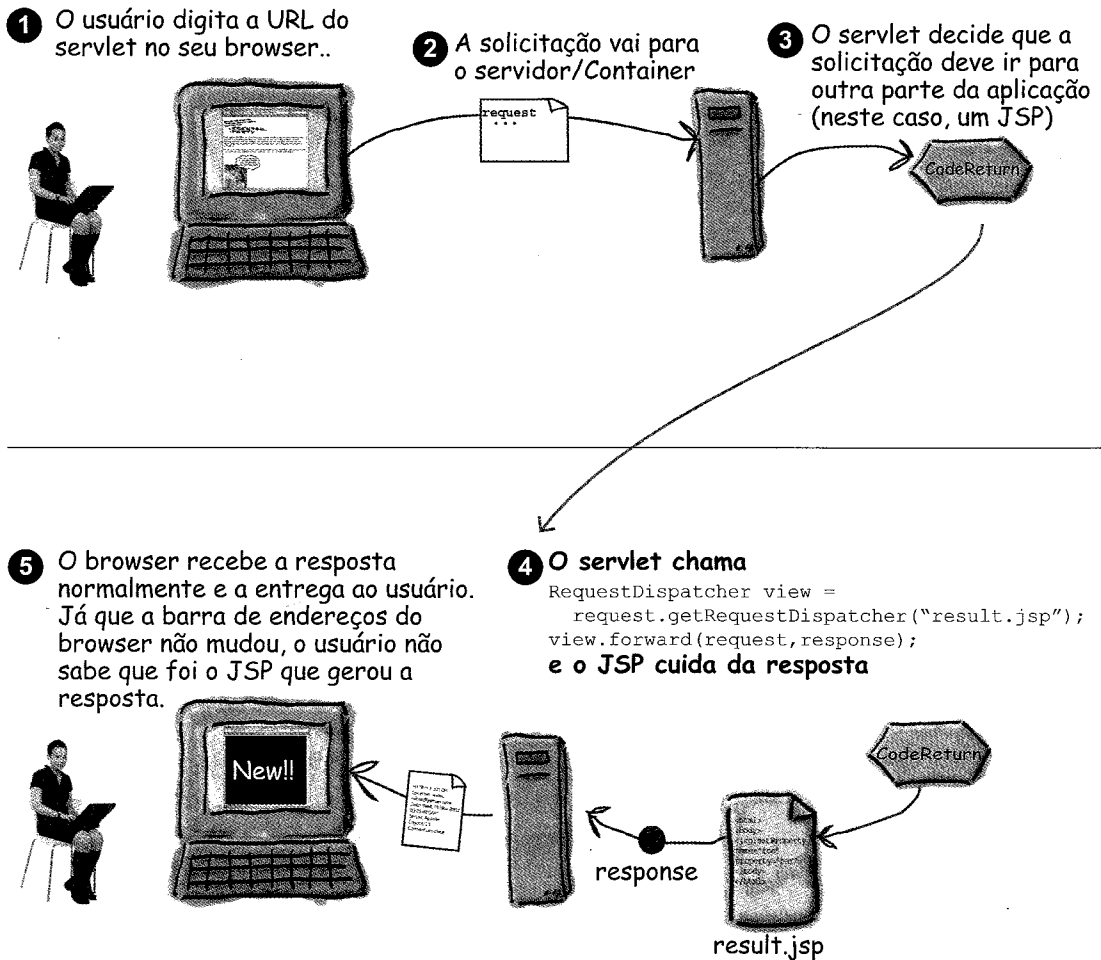
*Bem, ele carrega uma String que É uma URL. A questão é: o `sendRedirect()` NÃO carrega um objeto do tipo URL. Você passa para ele uma String que é uma URL completa ou relativa. Se o Container não puder transformar uma URL relativa numa URL completa, ele manda uma `IllegalStateException`. O que confunde é lembrar que ISTO é errado:  
`sendRedirect(URL nova("http://www.oreilly.com"))`*

*Não! Parece muito certo, mas está MUITO errado. O `sendRedirect()` carrega uma String e pronto!*

## O request dispatch acontece no lado do servidor

E esta é a grande diferença entre um redirecionamento e um request dispatch – o *redirecionamento* faz o *cliente* executar o trabalho, enquanto que o *request dispatch* faz com que outro componente no *servidor* execute o trabalho. Então, lembre-se: redirecionamento = *cliente*, request dispatch = *servidor*. Nós falaremos mais sobre o request dispatch mais adiante, porém, estas duas páginas deverão dar a você uma noção dos pontos mais importantes.

### O Request Dispatch



## Redirecionamento X Request Dispatch

Eu não tenho tempo para isto!  
Escuta, por que você não liga  
para o Barney? Talvez ELE  
tenha tempo para esta besteira.

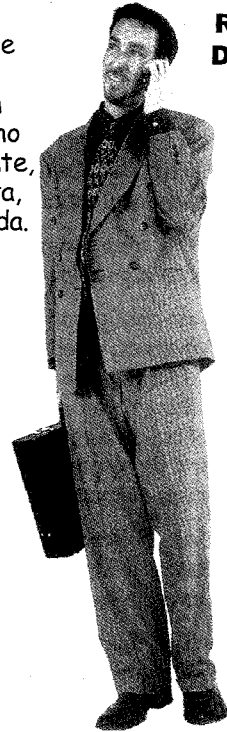
### Redirecionamento



Oi Kari, aqui é o Dan... Eu quero a sua ajuda  
com um cliente. Vou encaminhar para você os  
detalhes de como retornar para ele, mas eu  
preciso que você cuide disso agora.  
Sim, eu SEI que você também tem suas  
necessidades... sim, eu SEI como a View é  
importante para o MVC... não, eu não creio  
que consiga outro JSP como aquele... o  
quê? Não entendi? A ligação está ruim...  
desculpe... eu não consigo ouvir nada...  
perdendo pacotes...

### Request Dispatch

Quando um servlet faz o  
**request dispatch**, é como se  
pedíssemos que um colega  
de trabalho cuidasse de um  
cliente. O colega de trabalho  
acaba respondendo ao cliente,  
mas o cliente não se importa,  
desde que alguém o responda.  
O usuário nunca sabe que  
outra pessoa assumiu o  
controle, pois a URL no  
browser não muda.



Quando um servlet faz um  
**redirecionamento**, é como se  
pedíssemos que o cliente ligasse para  
outra pessoa. Neste caso, o cliente é  
o browser, e não o usuário. O browser  
faz a nova chamada em benefício  
do usuário, depois do servlet que  
foi originalmente requisitado dizer:  
"Desculpe, chame este cara aqui..."

O usuário vê a nova URL no browser.

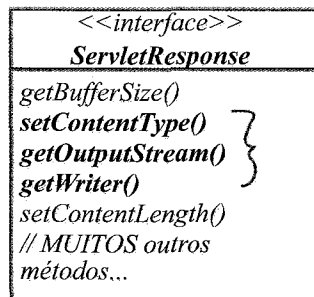
## Revisão: `HttpServletResponse`



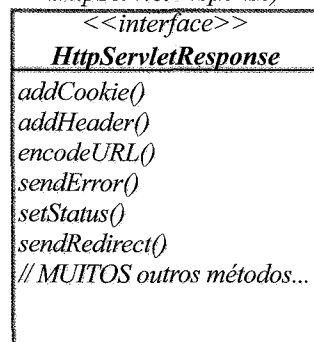
### Pontos de bala

- Você usa a Resposta para enviar dados de volta para o cliente.
- Os métodos mais comuns que você chamará no objeto response (`HttpServletResponse`) são o `setContentType()` e o `getWriter()`.
- Tome cuidado – muitos desenvolvedores acreditam que o método seja `getPrintWriter()`, mas é `getWriter()`.
- O método `getWriter()` permite fazer I/O por caractere para criar o HTML (ou algo mais) para o stream.
- Você também pode usar a resposta para configurar headers, enviar erros e adicionar cookies.
- No mundo real, você provavelmente usará um JSP para enviar a maioria das respostas HTML, mas você ainda poderá usar um stream de resposta para enviar dados binários (como um arquivo JAR, talvez) ao cliente.
- O método que você chama na sua resposta para receber um stream binário é `getOutputStream()`.
- O método `setContentType()` diz ao browser como tratar os dados vindos na resposta. Os tipos de conteúdos mais comuns são “text/html”, “application/pdf” e “image/jpeg”.
- Você não precisa memorizar os tipos de conteúdos (também conhecidos como tipos MIME).
- Você pode configurar headers de resposta usando o `addHeader()` ou o `setHeader()`. A diferença depende de o header já fazer parte da resposta. Se já fizer, o `setHeader()` irá substituir o valor, mas o `addHeader` acrescentará um valor adicional à resposta existente. Se o header ainda não fizer parte da resposta, então o `setHeader()` e o `addHeader()` comportam-se exatamente do mesmo modo.
- Se você não quiser responder a uma solicitação, você pode redirecionar a solicitação para outra URL. O browser encarregar-se-á de enviar a nova solicitação para a URL que você fornecer.
- Para redirecionar uma solicitação, utilize o `sendRedirect(aStringURL)` na resposta.
- Você não pode chamar o `sendRedirect()` após a resposta ter sido gerada! Ou seja, se você já escreveu alguma coisa para o stream, é muito tarde para fazer um redirecionamento.
- Um redirecionamento de uma solicitação é diferente de um *dispatch*. O *request dispatch* (mais detalhado em outro capítulo) acontece no servidor, enquanto que um redirecionamento ocorre no cliente. No caso do *dispatch*, a solicitação é entregue para outro componente no servidor, geralmente na mesma aplicação. Um redirecionamento simplesmente informa ao browser para ir para uma URL diferente.

**interface `ServletResponse`**  
(`javax.servlet.ServletResponse`)



**Interface `HttpServletResponse`**  
(`javax.servlet.http.  
HttpServletResponse`)







## Pausa para o café

### Teste Preparatório – Capítulo 4

---

1 Como o código do servlet de um método service (por exemplo, `doPost()`) obteria o valor do header “User-Agent” da solicitação? (Escolha todas as que se aplicam.)

- ☐ A. `String userAgent = request.getParameter("User-Agent");`
  - ☐ B. `String userAgent = request.getHeader("User-Agent");`
  - ☐ C. `String userAgent = request.getRequestHeader("Mozilla");`
  - ☐ D. `String userAgent = getServletContext().getInitParameter("User-Agent");`
- 

2 Quais são os métodos HTTP usados para mostrar ao cliente o que o servidor está recebendo? (Escolha todas as que se aplicam.)

- ☐ A. GET
  - ☐ B. PUT
  - ☐ C. TRACE
  - ☐ D. RETURN
  - ☐ E. OPTIONS
- 

3 Qual método do `HttpServletResponse` é usado para redirecionar uma solicitação HTTP para outra URL?

- ☐ A. `sendURL()`
- ☐ B. `redirectURL()`
- ☐ C. `redirectHttp()`
- ☐ D. `sendRedirect()`
- ☐ E. `getRequestDispatcher()`

4 Quais os métodos HTTP que NÃO são considerados idempotentes? (Escolha todas as que se aplicam.)

- ☐ A. GET
  - ☐ B. POST
  - ☐ C. HEAD
  - ☐ D. PUT
- 

5 Sendo `req` um `HttpServletRequest`, qual das alternativas recebe um stream de entrada de dados binários? (Escolha todas as que se aplicam.)

- ☐ A. `BinaryInputStream s = req.getInputStream();`
  - ☐ B. `ServletInputStream s = req.getInputStream();`
  - ☐ C. `BinaryInputStream s = req.getBinaryStream();`
  - ☐ D. `ServletInputStream s = req.getBinaryStream();`
- 

6 Como você configuraria um header chamado "CONTENT-LENGTH" no objeto `HttpServletResponse`? (Escolha todas as que se aplicam.)

- ☐ A. `response.setHeader("CONTENT-LENGTH", "1024");`
  - ☐ B. `response.setHeader("CONTENT-LENGTH", "1024");`
  - ☐ C. `response.setStatus(1024);`
  - ☐ D. `response.setHeader("CONTENT-LENGTH", 1024);`
- 

7 Escolha o trecho do código do servlet que recebe um stream binário para escrever uma imagem, ou outro tipo binário, no `HttpServletResponse`.

- ☐ A. `java.io.PrintWriter out = response.getWriter();`
- ☐ B. `ServletOutputStream out = response.getOutputStream();`
- ☐ C. `java.io.PrintWriter out =  
new PrintWriter(response.getWriter());`
- ☐ D. `ServletOutputStream out = response.getBinaryStream();`

8 Quais métodos são usados por um servlet para tratar os dados do formulário vindos de um cliente? (Escolha todas as que se aplicam.)

- ☐ A. `HttpServlet.doHead()`
  - ☐ B. `HttpServlet.doPost()`
  - ☐ C. `HttpServlet.doForm()`
  - ☐ D. `ServletRequest.doGet()`
  - ☐ E. `ServletRequest.doPost()`
  - ☐ F. `ServletRequest.doForm()`
- 

9 Quais dos seguintes métodos são declarados no `HttpServletRequest` ao contrário do `ServletRequest`? (Escolha todas as que se aplicam.)

- ☐ A. `getMethod()`
  - ☐ B. `getHeader()`
  - ☐ C. `getCookies()`
  - ☐ D. `getInputStream()`
  - ☐ E. `getParameterNames()`
- 

10 Como os desenvolvedores de servlet devem tratar o método `service()` do `HttpServlet` quando estenderem o `HttpServlet`? (Escolha todas as que se aplicam.)

- ☐ A. Eles devem anular o método `service()` na maioria dos casos.
- ☐ B. Eles devem chamar o método `service()` do `doGet()` OU `doPost()`.
- ☐ C. Eles devem chamar o método `service()` do método `init()`.
- ☐ D. Eles devem anular pelo menos um método `doXXX()` (como um `doPost()`).



## Pausa para o café

### Respostas - Capítulo 4

---

- 1 Como o código do servlet de um método service (por exemplo, `doPost()`) obteria o valor do header "User-Agent" da solicitação? (Escolha todas as que se aplicam.)

(API)

- ☐ A. `String userAgent = request.getParameter("User-Agent");`
- ☒ B. `String userAgent = request.getHeader("User-Agent");`
- ☐ C. `String userAgent = request.getRequestHeader("Mozilla");`
- ☐ D. `String userAgent = getServletContext().getInitParameter("User-Agent");`

— A opção B mostra a chamada correta ao método que passa o nome do header como um parâmetro `String`.

- 2 Quais são os métodos HTTP usados para mostrar ao cliente o que o servidor está recebendo? (Escolha todas as que se aplicam.)

(Cap. 4, métodos HTTP)

- ☐ A. GET
- ☐ B. PUT
- ☒ C. TRACE
- ☐ D. RETURN
- ☐ E. OPTIONS

— Este método é usado tipicamente para troubleshooting e não para produção.

- 3 Qual método do `HttpServletResponse` é usado para redirecionar uma solicitação HTTP para outra URL?

(API)

- ☐ A. `sendURL()`
- ☐ B. `redirectURL()`
- ☐ C. `redirectHttp()`
- ☒ D. `sendRedirect()`
- ☐ E. `getRequestDispatcher()`

— A opção D está correta e dos métodos listados, é o único que existe no `HttpServletResponse`.

- 4 Quais os métodos HTTP que NÃO são considerados idempotentes? (Escolha todas as que se aplicam.)

(Cap. 4, solicitações idempotentes)

- ☐ A. GET  
☒ B. POST  
☐ C. HEAD  
☐ D. PUT

— Por design, o POST deve conduzir as solicitações para atualizar o estado do servidor. Em geral, a mesma atualização não deve ser aplicada várias vezes.

- 5 Sendo req um HttpServletRequest, qual das alternativas recebe um stream de entrada de dados binários? (Escolha todas as que se aplicam.)

(API)

- ☐ A. `BinaryInputStream s = req.getInputStream();`  
☒ B. `ServletInputStream s = req.getInputStream();` — A opção B especifica o método e o tipo de retorno  
☐ C. `BinaryInputStream s = req.getBinaryStream();`  
☐ D. `ServletInputStream s = req.getBinaryStream();` corretos.

- 6 Como você configuraria um header chamado "CONTENT-LENGTH" no objeto HttpServletResponse? (Escolha todas as que se aplicam.)

(API)

- ☐ A. `response.setHeader("CONTENT-LENGTH", "1024");` — A opção B mostra a maneira correta de configurar um header HTTP com dois parâmetros Strings, um representando o nome do header, e o outro, o valor.  
☒ B. `response.setHeader("CONTENT-LENGTH", "1024");`  
☐ C. `response.setStatus(1024);`  
☐ D. `response.setHeader("CONTENT-LENGTH", 1024);`

- 7 Escolha o trecho do código do servlet que recebe um stream binário para escrever uma imagem, ou outro tipo binário, no HttpServletResponse.

(API)

- ☐ A. `java.io.PrintWriter out = response.getWriter();`  
☒ B. `ServletOutputStream out = response.getOutputStream();`  
☐ C. `java.io.PrintWriter out = new PrintWriter(response.getWriter());`  
☐ D. `ServletOutputStream out = response.getBinaryStream();`

— A opção A está incorreta, porque ela usa um PrintWriter orientado por caractere.

8 Quais métodos são usados por um servlet para tratar os dados do formulário vindos de um cliente? (Escolha todas as que se aplicam.)

(API)

- ☐ A. `HttpServlet.doHead()`
- ☒ B. `HttpServlet.doPost()`
- ☐ C. `HttpServlet.doForm()`
- ☐ D. `ServletRequest.doGet()`
- ☐ E. `ServletRequest.doPost()`
- ☐ F. `ServletRequest.doForm()`

*- As opções C e F estão erradas, porque estes métodos não existem.*

9 Quais dos seguintes métodos são declarados no `HttpServletRequest` ao contrário do `ServletRequest`? (Escolha todas as que se aplicam.)

(API)

- ☒ A. `getMethod()`
- ☒ B. `getHeader()`
- ☒ C. `getCookies()`
- ☐ D. `getInputStream()`
- ☐ E. `getParameterNames()`

*- As opções A, B e C referem-se aos componentes de uma solicitação HTTP.*

10 Como os desenvolvedores de servlet devem tratar o método `service()` do `HttpServlet` quando estenderem o `HttpServlet`? (Escolha todas as que se aplicam.)

(API)

- ☐ A. Eles devem anular o método `service()` na maioria dos casos.
- ☐ B. Eles devem chamar o método `service()` do `doGet()` ou `doPost()`.
- ☐ C. Eles devem chamar o método `service()` do método `init()`.
- ☒ D. Eles devem anular pelo menos um método `doxxx()` (como um `doPost()`).

*- A opção D está correta. Os desenvolvedores geralmente usam os métodos `doGet()` e `doPost()`.*