

# Web Infrastructure

RES, Lecture 5

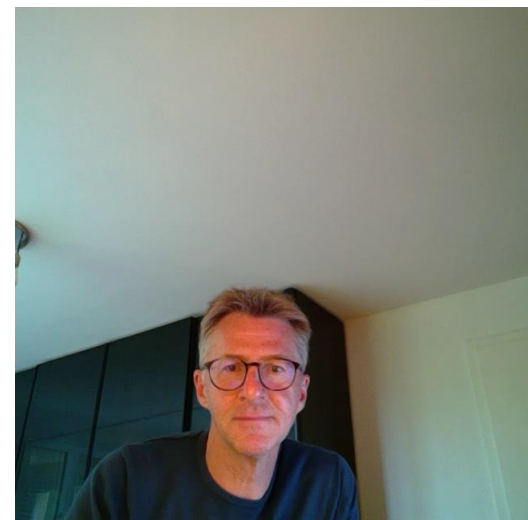
---

Olivier Liechti  
Juergen Ehrensberger



HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD

[www.heig-vd.ch](http://www.heig-vd.ch)



# Systemic Qualities



# The Notion of Systemic Quality

---

- How do we specify the **requirements** of a system?
  - We always build systems to deliver some sort of "**service**" (web, messaging, access control, customer management, etc.).
  - We first have to specify "**what**" the system should do. In other words, we have to specify **functional requirements**.
  - We also have to specify "**how**" the system should behave, i.e. what qualities it should exhibit. These are the **non-functional requirements**.
- Non-functional requirements characterize **systemic qualities, or "-ilities"**:
  - There are lots of different systemic qualities. Depending on the system, some are more important than others.
  - Choices made when defining the system architecture have a large impact on the systemic qualities.
  - Your life as an architect will be to deal with **trade-offs** in addressing conflicting systemic qualities.



## **System**

Vehicle

### **Functional requirements**

Move people around

### **Non-functional requirements**

Performance

Capacity

Reliability

Cost

Aesthetics

Ease of use





Different systemic qualities often create **opposing forces**.

Defining the “right” architecture for a system means finding the right **balance** between these forces.



# Some Systemic Qualities...

---

- **Response time**
  - Measures the time required to present a result to the user
  - Important for the end-user
- **Throughput**
  - Measures the number of requests that can be processed in a given time frame
  - Important for the service provider
- **Scalability**
  - Measures how easy/costly it is to adapt the system in order to handle additional load
  - Ideally: linear scalability. "2 x more users => 2 x more servers"
- **Availability**
  - Measures the percentage of time during which the system can be used
  - 99% availability = average unavailability of 3.65 jours per year, i.e 1 hour and 41 minutes per week.
- **Security**
- **Manageability**
  - Measures the ease of operating the system: how easy is it to monitor it, to detect issues, to upgrade, etc.



# Manageability

---



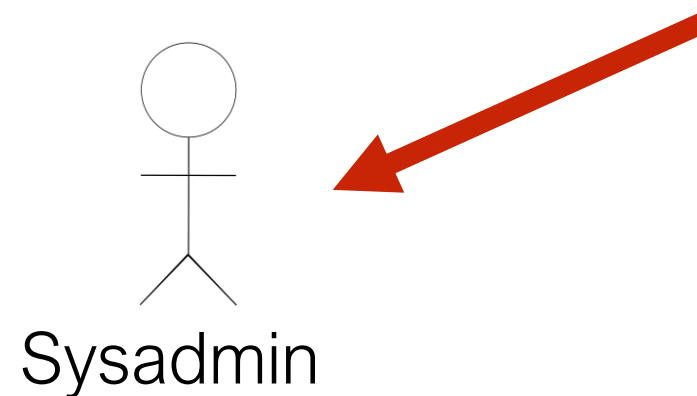
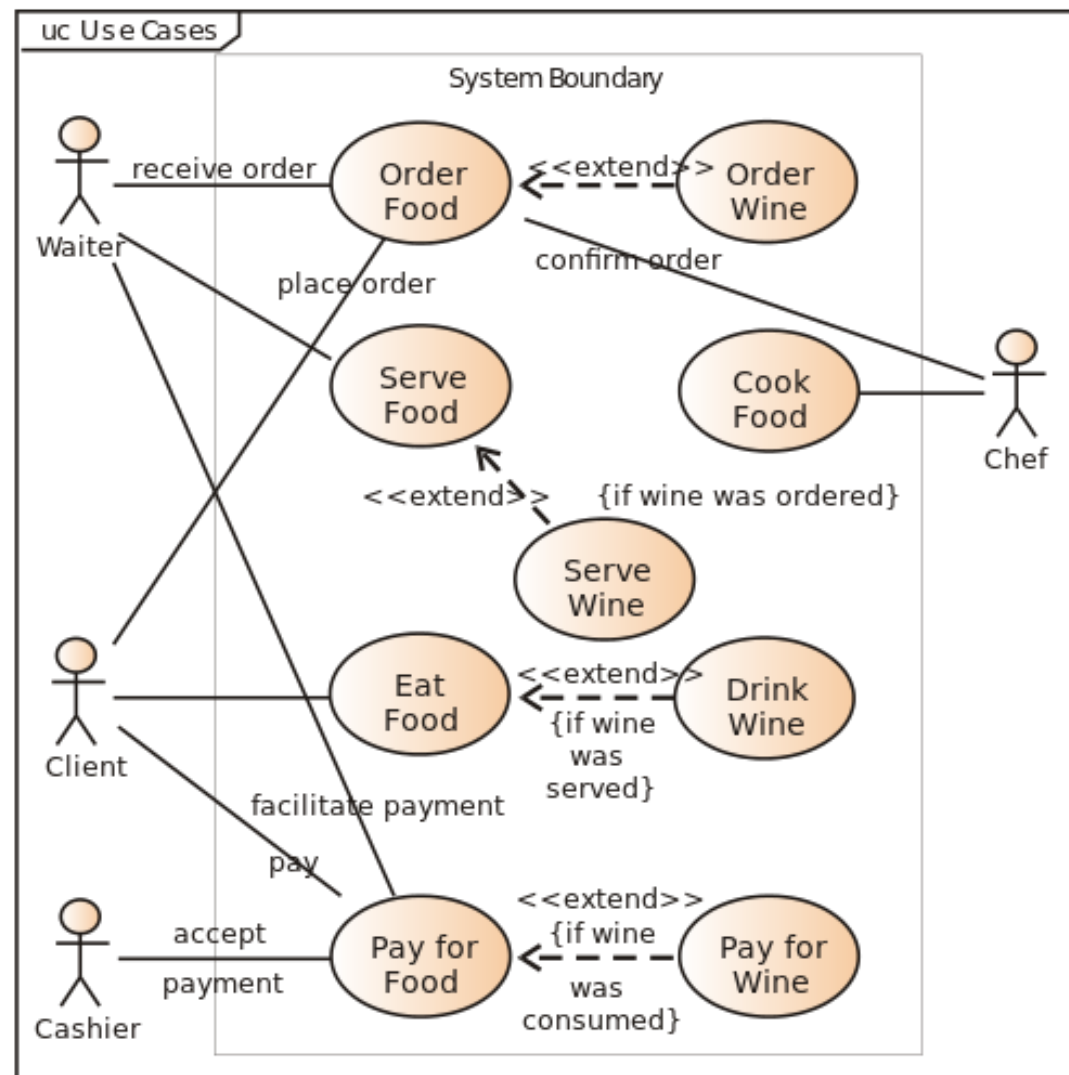
# Manageability

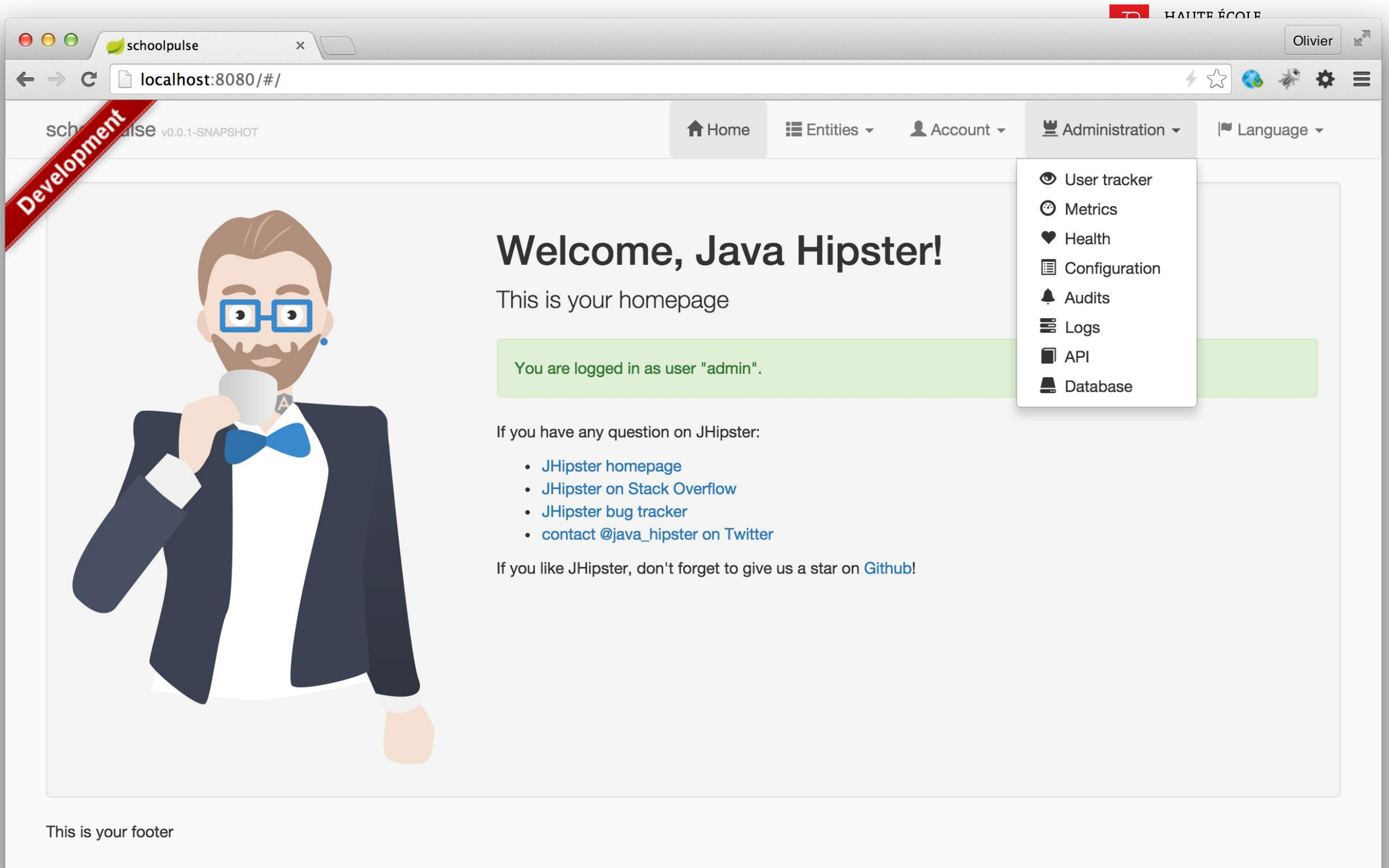
---





# Manageability





# Manageability

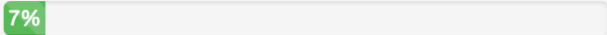
## Application Metrics

Refresh

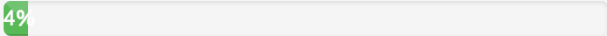
### JVM Metrics

#### Memory

Total Memory (259M / 3,818M)



Heap Memory (141M / 3,818M)

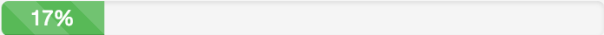


Non-Heap Memory (118M / 121M)

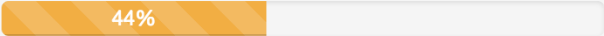


#### Threads (Total: 48)

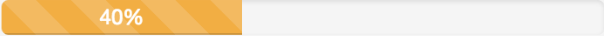
Runnable 8



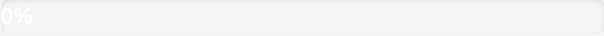
Timed waiting (21)



Waiting (19)



Blocked (0)



#### Garbage collections

Mark Sweep count	5
Mark Sweep time	644ms
Scavenge count	33
Scavenge time	146ms

## Health checks

Refresh

Service name	Status	Details
Database	UP	
Email	DOWN	
Disk space	UP	

## HTTP requests (events per second)

Active requests: 1 - Total requests: 146

Code	Count	Mean	Average (1 min)	Average (5 min)	Average (15 min)
Ok	145	0.15	0.03	0.07	0.08
Not found	0	0.00	0.00	0.00	0.00
Server Error	0	0.00	0.00	0.00	0.00



# Performance et scalabilité du service HTTP





# Optimiser les performances du service HTTP

- > Deux approches complémentaires
  - Scaling Up (scalabilité verticale)
  - Scaling Out (scalabilité horizontale)

Scale Up:  
"Dopez vos performances!"



- > Scale Up
  - On vise à améliorer les **performances d'un composant**, en procédant à des optimisations et en ajoutant des ressources (CPU, mémoire, bande passante, etc.).



Scale Out:  
"L'union fait la force!"



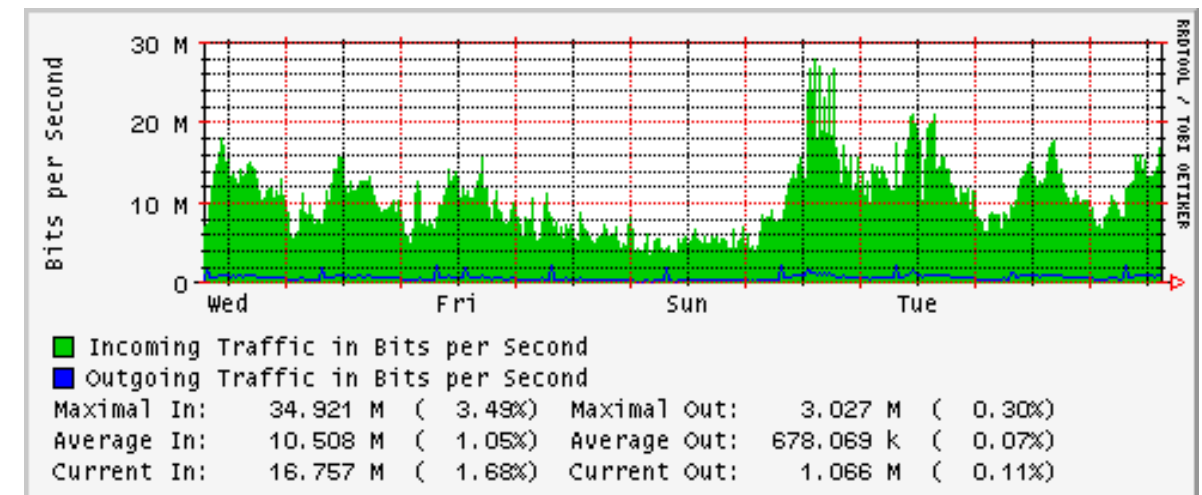
- > Scale Out
  - On vise à améliorer les **performances du système global**, en répartissant la charge entre plusieurs composants équivalents.



# Optimiser les performances du service HTTP

## > Références

- Sander Temme, expert performance
- Articles et présentations:  
<http://people.apache.org/~sctemme>

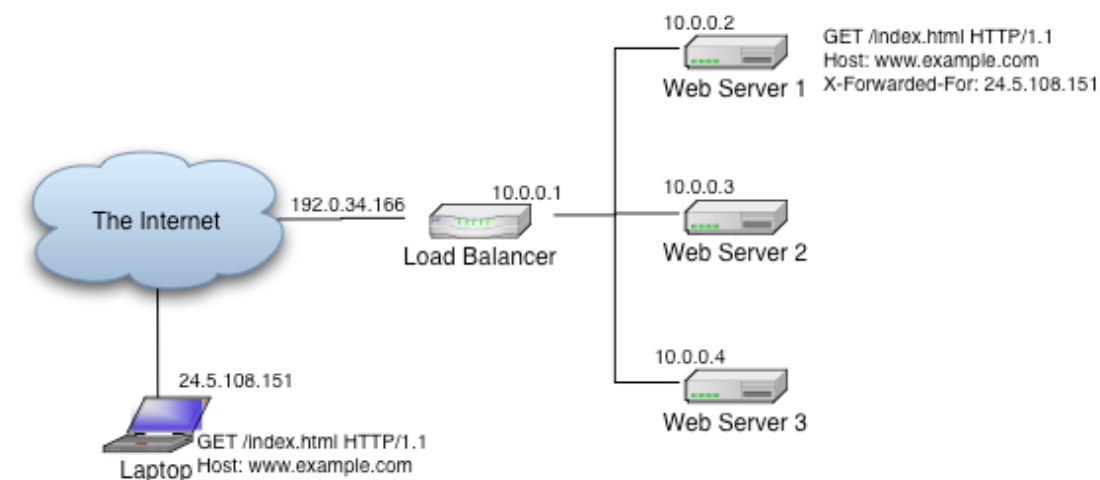


## > Scale Up

- Avant d'optimiser, mesurer!
- Choisir le bon modèle de (multi-process vs. multi-thread), en fonction de la plateforme (système d'exploitation)

## > Scale Out

- Utiliser Apache en mode reverse proxy et exploiter les modules de caching.



# Caching

- **Qu'est-ce qu'un "cache"**

- Une zone mémoire où on peut stocker des données, que l'on pourrait (ré-)utiliser dans le futur.
- "Pourquoi recalculer quelque chose que j'ai déjà calculé il y a 5 minutes?"
- "Si je dois aller acheter une bouteille d'eau, autant acheter directement une harasse pour avoir une réserve."

- **Quels sont les problèmes liés à l'utilisation d'un cache?**

- "Fraîcheur" des données: est-ce que ce que je lis dans le cache est encore une copie fidèle de l'original?
- Quelle politique définit la gestion du cache (ajouter, supprimer des données, etc.)
- Synchronisation: que se passe-t-il quand j'utilise plusieurs caches (par exemple dans un cluster)?



# Caching et HTTP

- **Exemple:**

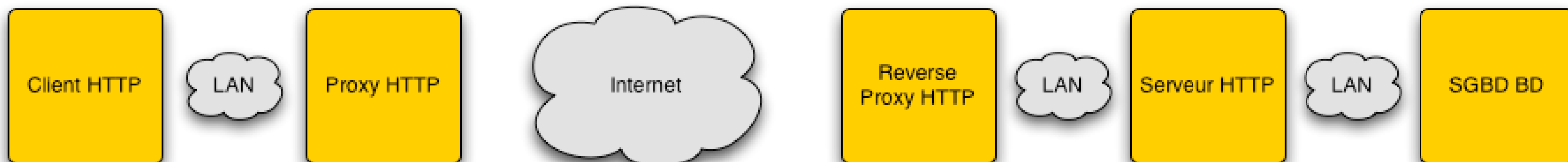
- On considère une application web, qui permet d'accéder à des ressources stockées dans une base de données.

- **Sans cache:**

- Chaque requête fait "un long voyage" (du client à la base de données)
- Plusieurs réseaux sont traversés.
  - Plusieurs composants doivent fournir un travail.

- **Question: où pourrait-on gérer un cache, pour pouvoir réutiliser toute ou partie des données liées à la requête?**

Requête



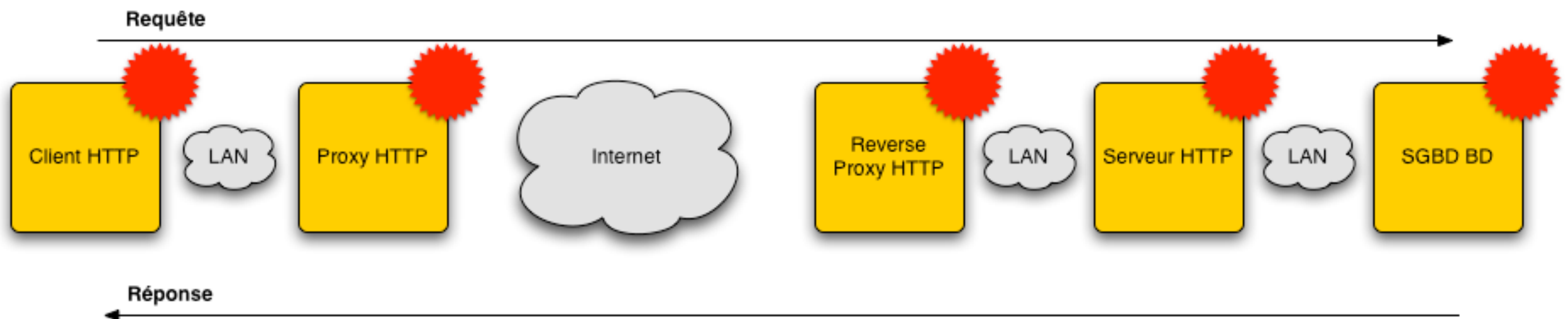
Réponse





# Réponse: à tous les niveaux!

- **Dans le navigateur:**
  - On ne va pas télécharger le logo de CNN tous les jours!
- **Dans un proxy du côté client:**
  - On ne va pas télécharger l'article du jour pour chaque utilisateur!
- **Dans un proxy du côté serveur:**
  - On ne va pas recalculer la page d'accueil pour chaque visiteur!
  - Dans le serveur http (niveau de l'application web):
    - On ne va pas tout recalculer, on ne va pas chaque fois aller jusqu'à la DB.
  - Dans la base de données (on ne va pas aller jusqu'au disque)



# Caching: que dit la spécification HTTP?

---

- Une section entière est consacrée à ce sujet:
  - <http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13>
  - 26 pages!
- **Modèle d'expiration (13.2)**
- **Modèle de validation (13.3)**
- Pertinence de la mise en cache des réponses (13.4)
- Construction de réponses grâce aux caches (13.5)



# Expiration et validation

---

Caching would be useless if it did not significantly improve performance.

The goal of caching in HTTP/1.1 is to eliminate the need to send requests in many cases, and to eliminate the need to send full responses in many other cases.

The former reduces the number of network round-trips required for many operations; we use an "**expiration**" mechanism for this purpose (see section 13.2). The latter reduces network bandwidth requirements; we use a "**validation**" mechanism for this purpose (see section 13.3).



- **Expiration**

- Quand on reçoit une réponse, on veut déterminer combien de temps elle va rester "valide".
- Pendant ce temps, il sera possible de récupérer la représentation de la ressource dans le cache.

- **Validation**

- Après expiration, on n'est pas sûr que le contenu du cache soit encore une représentation "fraîche" de la ressource.
- On peut alors procéder en 2 temps:
  - On envoie une première requête au serveur pour demander si la ressource a changé.
  - Si oui, c'est seulement dans un deuxième temps qu'on récupère la nouvelle représentation de la ressource.





- **Expiration**

- Quand on reçoit une réponse, on veut déterminer combien de temps elle va rester "valide".
- Pendant ce temps, il sera possible de récupérer la représentation de la ressource dans le cache.

- **Validation**

- Après expiration, on n'est pas sûr que le contenu du cache soit encore une représentation "fraîche" de la ressource.
- On peut alors procéder en 2 temps:
  - On envoie une première requête au serveur pour demander si la ressource a changé.
  - Si oui, c'est seulement dans un deuxième temps qu'on récupère la nouvelle représentation de la ressource.



# Disponibilité du service HTTP



# Disponibilité du service HTTP

---

- **Objectif**

- Garantir que le service HTTP de mon entreprise va être utilisable "la plus grande partie du temps".

- **Causes pour lesquelles mon service pourrait être inutilisable**

- Causes prévisibles: mises à jour et déploiement d'une nouvelle version (du système d'exploitation, du serveur HTTP, de l'application web, etc.)
- Causes imprévisibles: pannes matérielles, pannes de réseau, bugs dans le serveur HTTP, surcharge, attaque (e.g. denial of service), catastrophe naturelle, etc.

- **Mesurer la disponibilité**

- $A = \text{MTBF} / (\text{MTBF} + \text{MTTR})$
- MTBF: Mean Time Between Failure (temps moyen entre 2 pannes)
- MTTR: Mean Time To Repair (temps moyen pour résoudre un problème)



# Disponibilité du service HTTP

---

- **Taux de disponibilité: 98%**
  - indisponibilité de 7.3 jours par an, soit 3 heures 22 minutes par semaine
- **Taux de disponibilité: 99%**
  - indisponibilité de 3.65 jours par an, soit 1 heure 41 minutes par semaine
- **Taux de disponibilité: 99.9%**
  - indisponibilité de 8 heures 45 min par an, soit 10 min 5 sec par semaine
- **Taux de disponibilité: 99.99%**
  - indisponibilité de 52.5 minutes par an, soit 1 minute par semaine
- **Taux de disponibilité: 99.999%**
  - indisponibilité de 5.25 minutes par an, soit 6 secondes par semaine
- **Taux de disponibilité: 99.9999%**
  - indisponibilité de 31.5 secondes par an, soit 0.6 secondes par semaine
  - indisponibilité de 6 minutes en 11.4 années!





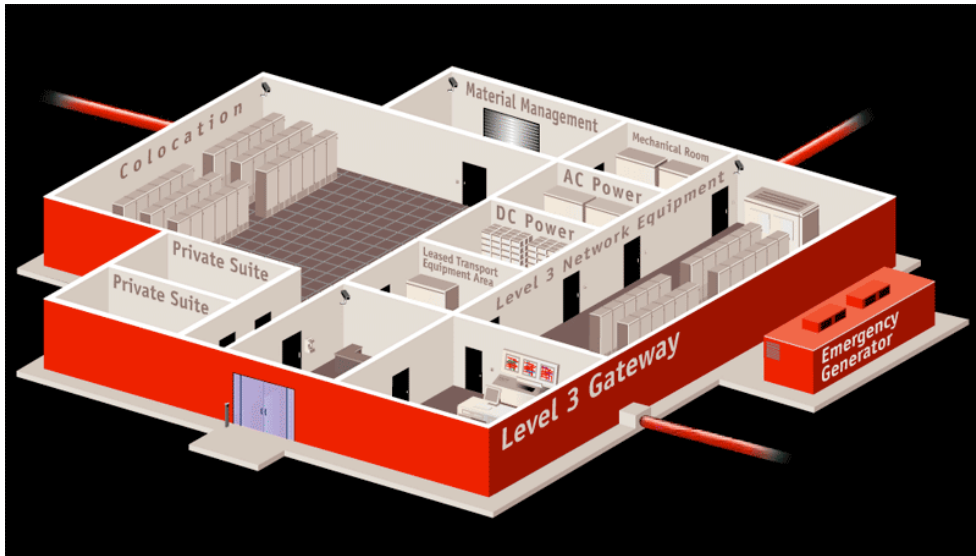
# Comment augmenter la disponibilité?

---

- **La disponibilité du système dépend de la qualité et de la fiabilité des composants du système:**
  - Les spécifications des composants matériels indiquent souvent une valeur de MTBF.
  - Il sera difficile d'assurer la disponibilité d'un service si le logiciel est "buggé".
- **La disponibilité dépend également de la charge supportée par le système:**
  - Un système qui se comporte bien à charge normale peut exhiber des problèmes quand un seuil est atteint (contentions).
- **Ajouter de la redondance à l'intérieur du système, à différents niveaux, permet d'augmenter la disponibilité:**
  - Si un composant tombe en panne, on peut "basculer" sur un composant équivalent de rechange.



# Composants qui peuvent être redondants



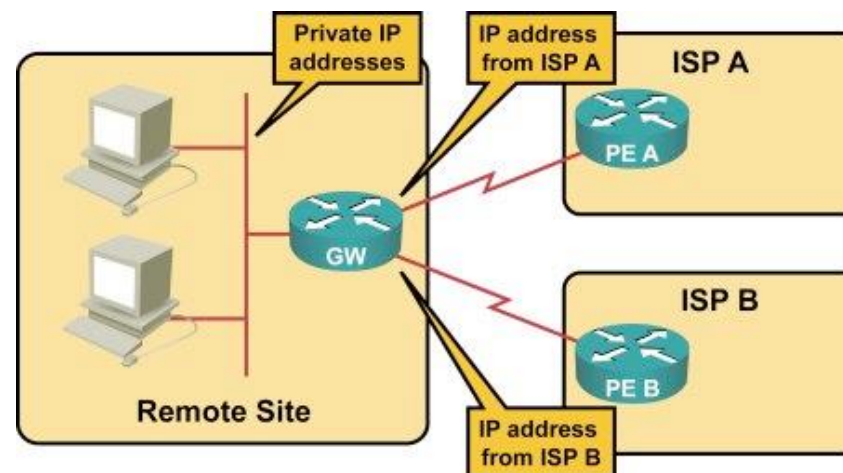
*Centre de calcul*



*Switch*



*Disques*



*Accès Internet*



*Serveurs*



*Alimentations électriques*



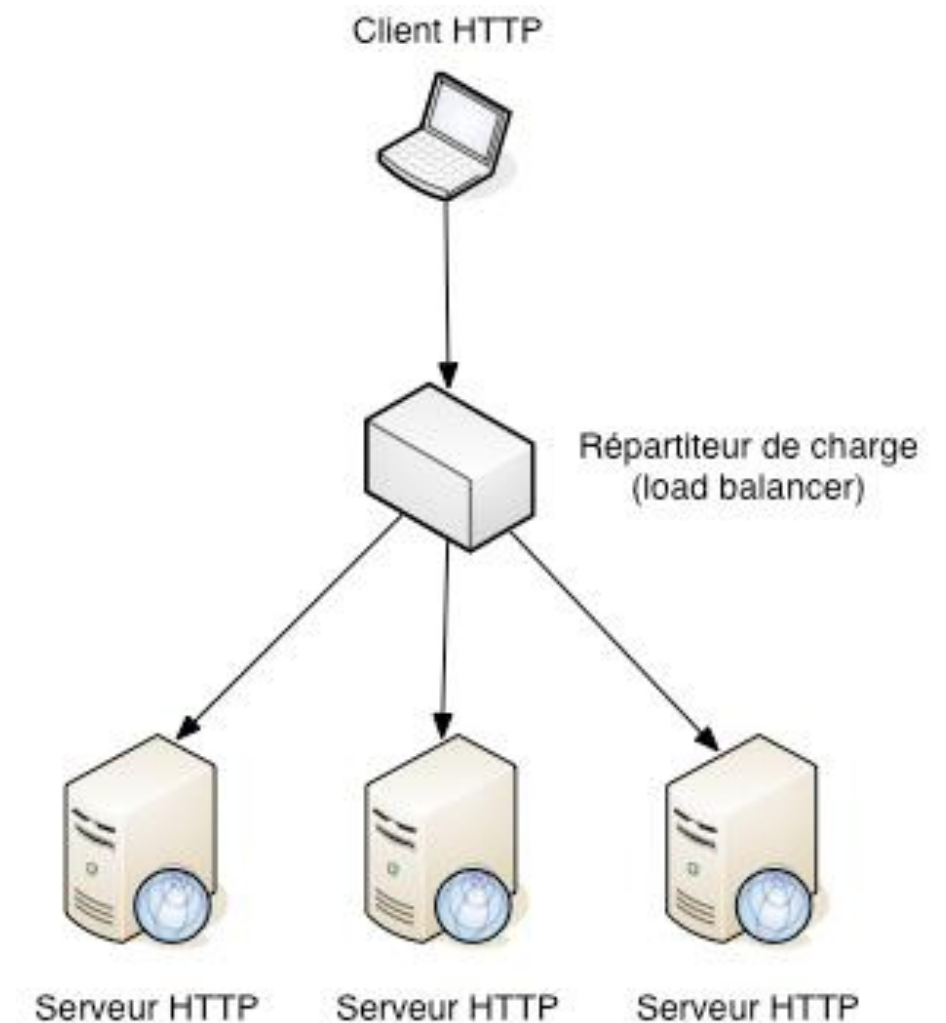
# Service redondant avec le serveur Apache (httpd)

## > Architecture

- au moins 2 serveurs physiques (ou deux machines virtuelles).
- au moins 2 serveurs httpd configurés de manière identique.
- 1 répartiteur de charge.

## > Fonctionnement

- Les requêtes HTTP sont envoyées vers le répartiteur de charge (Virtual IP address).
- Le répartiteur de charge distribue les requêtes entre les différents serveurs.
- Une politique définit les modalités de cette distribution!



# Comment réaliser le répartiteur de charge?

## > Solution matérielle

- Des entreprises fournissent des équipements spécialisés pour la répartition de charge.
- Ils peuvent être configurés de manière flexible.
- Ils peuvent être eux-mêmes déployés de manière à être hautement disponibles.
- Exemple: **BIG-IP de F5**. Voir: <http://www.f5.com/solutions/resources/deployment-guides/>

## > Solution logicielle

- La fonction de répartition de charge peut aussi être assurée par un logiciel installé sur un serveur "standard".
- Exemple: **Apache mod\_proxy\_balancer + mod\_proxy**
  - [http://httpd.apache.org/docs/2.2/mod/mod\\_proxy\\_balancer.html](http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html)
  - [http://httpd.apache.org/docs/2.2/mod/mod\\_proxy.html#proxypass](http://httpd.apache.org/docs/2.2/mod/mod_proxy.html#proxypass)





# Affinité de session (sticky sessions) (1)

## > Situation:

- On a besoin de haute disponibilité pour une application.
- On installe un **cluster** de serveurs apache.
- Un load balancer distribue les requêtes entre les instances du cluster.
- L'application gère une **session** (e.g. shopping cart).

## > Problème:

- Que se passe-t-il si les requêtes d'un utilisateur arrivent vers des instances différentes du cluster?

## > Solution:

- Le load balancer doit "**reconnaître**" des requêtes qui appartiennent à la même session.
- Toutes ces requêtes sont systématiquement envoyées à la même instance.



# Affinité de session (sticky sessions) (2)

## > Question:

- Comment le load balancer peut-il reconnaître les requêtes d'une même session?

## > Solution 1:

- En utilisant un cookie spécial

## > Solution 2:

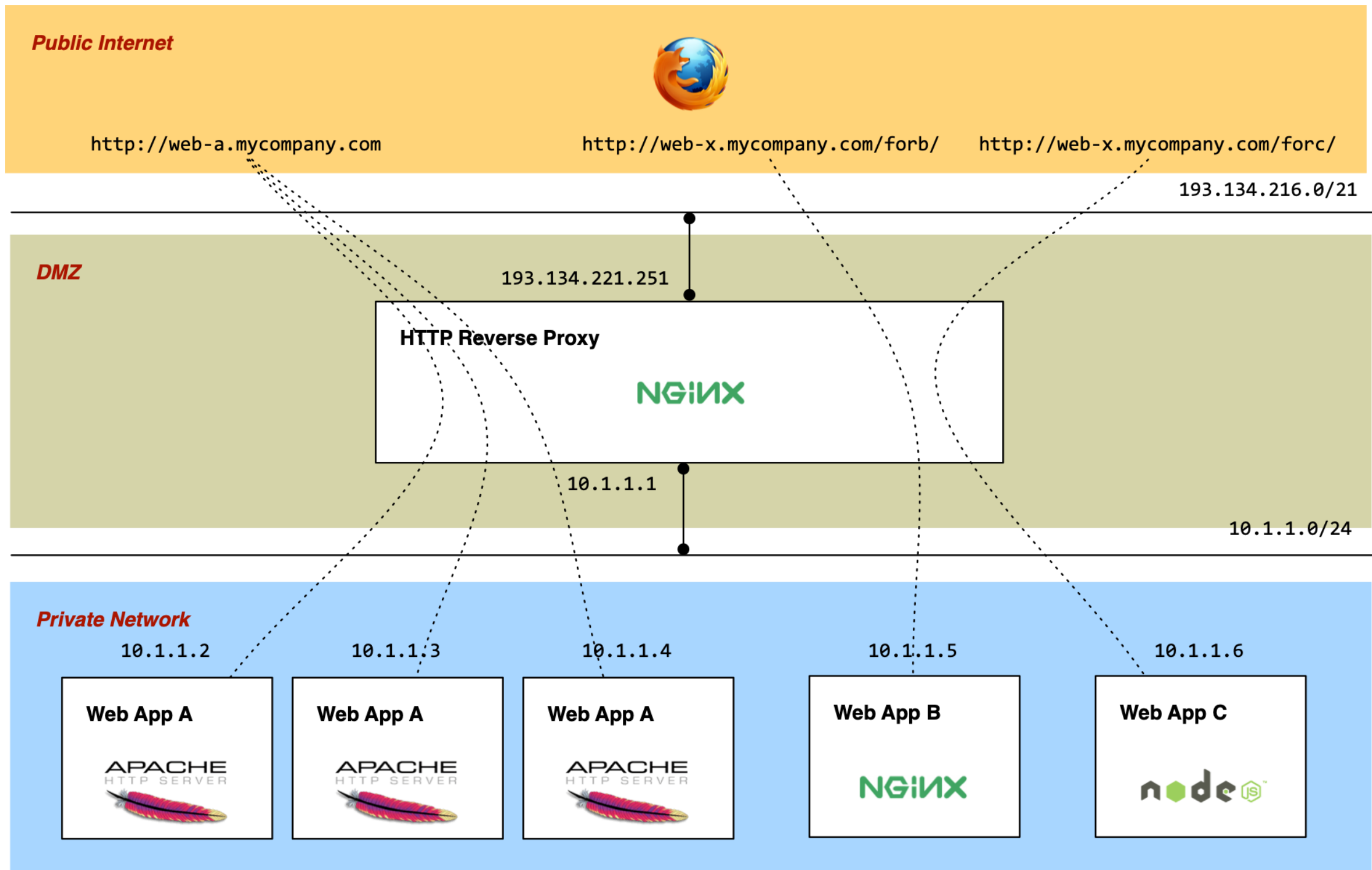
- En réécrivant l'URL et en ajoutant un ID de session

## > Solution 3:

- Certains produits ont des méthodes plus sophistiquées.







# The Role of the Reverse Proxy

- An HTTP proxy that is “close” to the server
- Forwarding requests to the “appropriate” server
- A reverse proxy can also be configured as a load balancer. In this case, it distributes HTTP requests between several “equivalent” servers.
- **Sticky sessions!**

```
ProxyRequests Off
<Proxy *>
Order deny,allow
Allow from all
</Proxy>

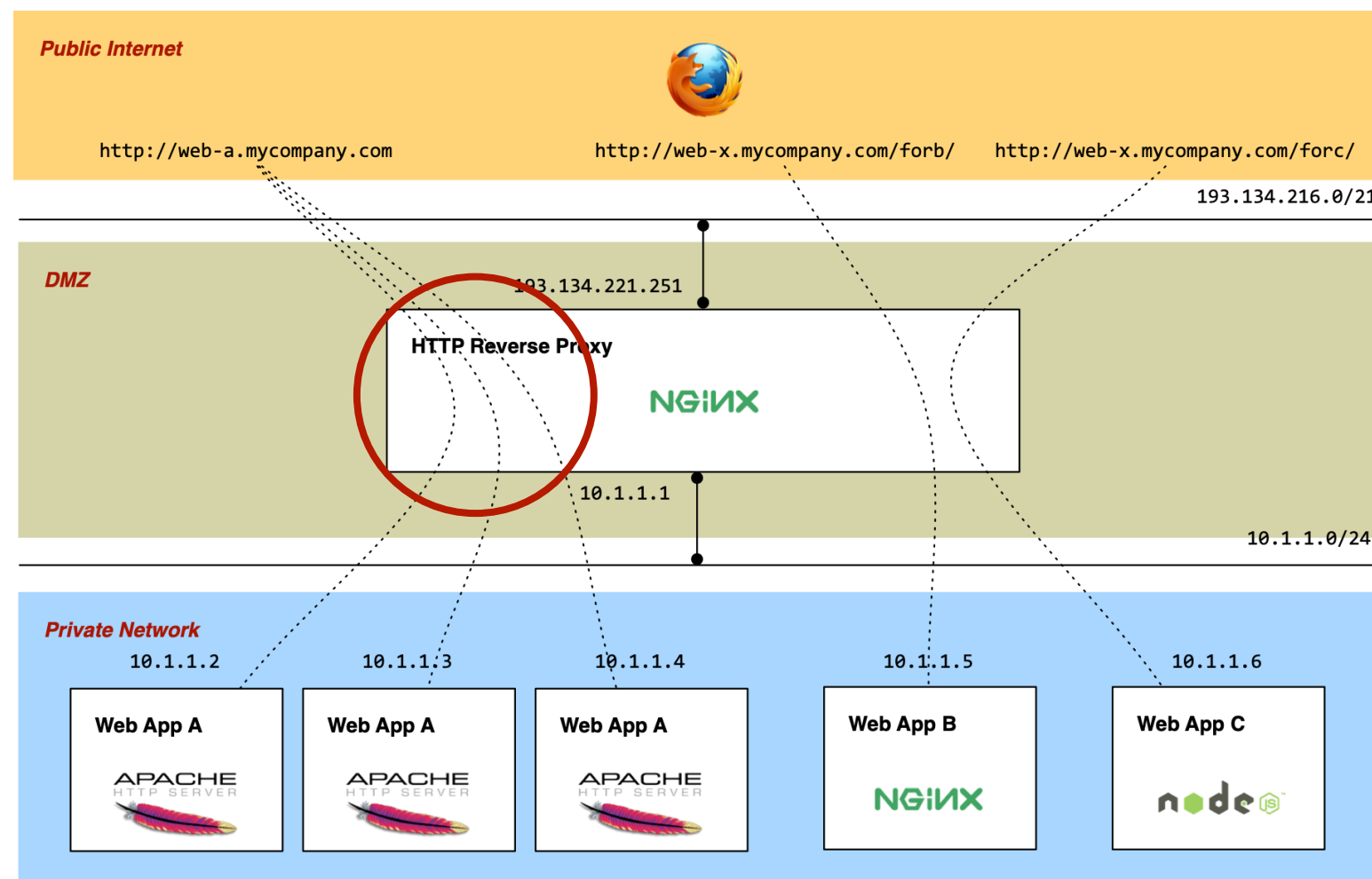
ProxyPass /foo http://192.168.1.2:8080/bar
ProxyPassReverse /foo http://192.168.1.2:8080/bar
```

```
Header add Set-Cookie "ROUTEID=.%{BALANCER_WORKER_ROUTE}e; path=/"
env=BALANCER_ROUTE_CHANGED
<Proxy balancer://mycluster>
BalancerMember http://192.168.1.50:80 route=1
BalancerMember http://192.168.1.51:80 route=2
ProxySet stickysession=ROUTEID
</Proxy>
ProxyPass /test balancer://mycluster
```



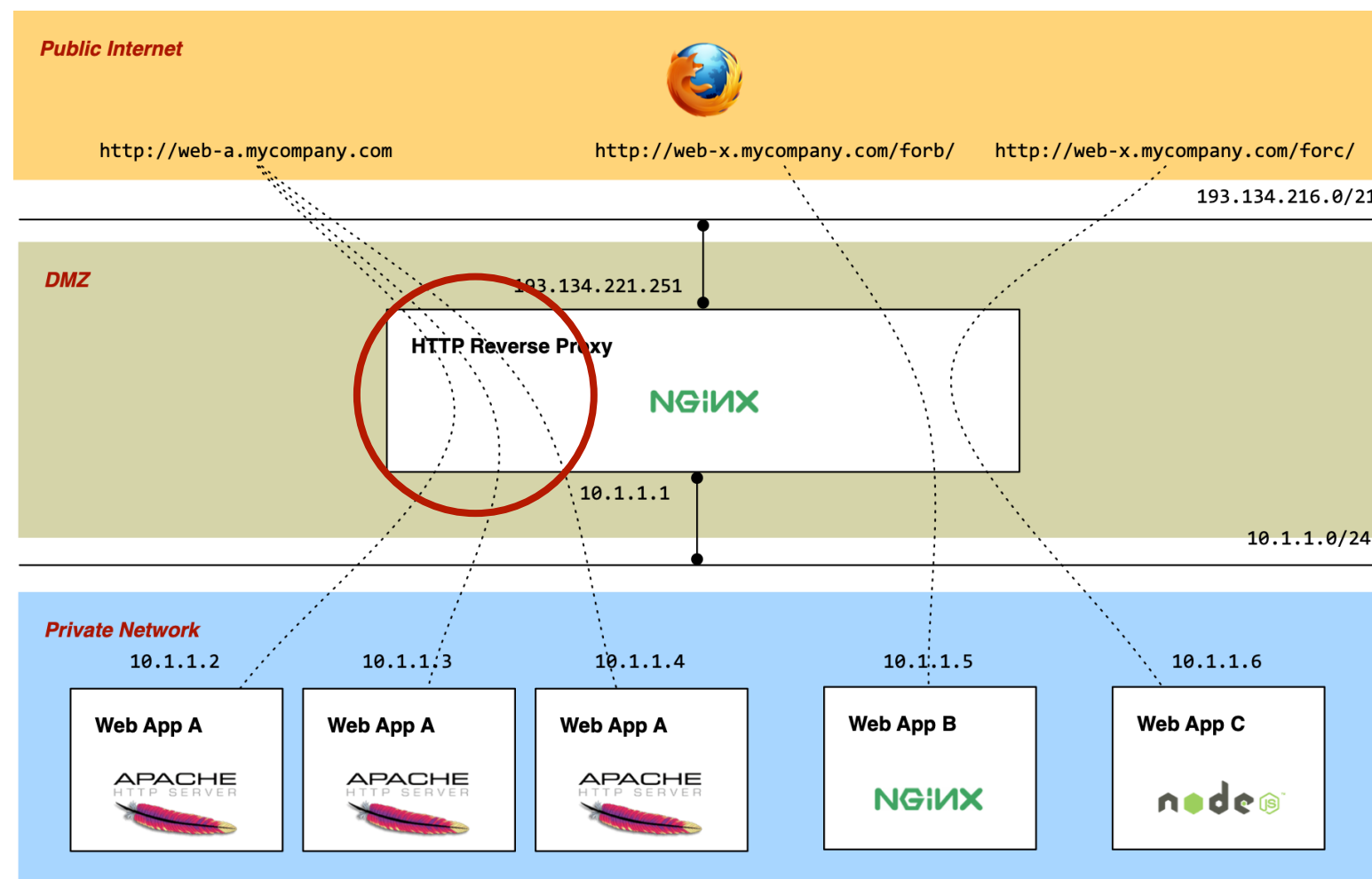
# Availability

- What happens if a server (or server component) fails?

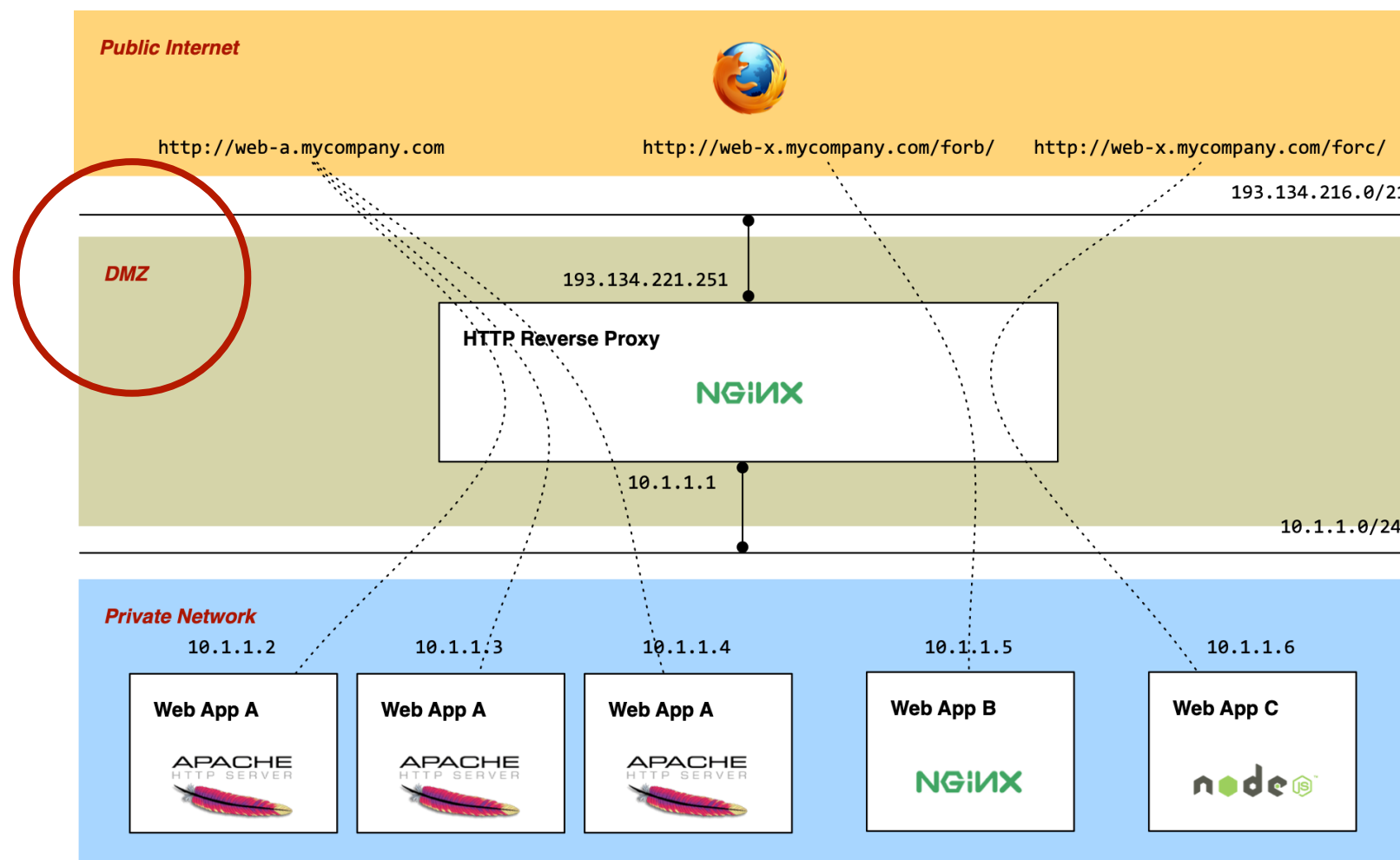


# Scalability

- Ability to evolve in order to sustain a bigger load, in quick and economical manner.
- Horizontal vs Vertical Scalability
- Elasticity



# Security



# Virtual Hosts





# Virtual Hosts & the Host Header

```
> telnet 84.16.80.79 80
```

```
GET /index.html HTTP/1.1
```

```
Host: www.wasabi-tech.com
```

```
> telnet 84.16.80.79 80
```

```
GET /index.html HTTP/1.1
```

```
Host: www.otherdomain.com
```



End of chapter

