

Gestion d'erreurs et résilience

Plan et objectifs

- Validation de formulaire et messages d'erreur
- Gestion d'erreurs
- Journaux
- Patrons de résilience

SI ÇA PEUT CASSER

ÇA CASSERA

Validation de formulaire

Lorsqu'on demande à l'utilisateur de remplir un formulaire web, il faut faire attention aux erreurs de saisie et aux données invalides.

On peut **limiter les erreurs en ajoutant des règles de validation** aux formulaires web. Si les valeurs entrées par l'utilisateur ne sont pas conformes aux règles de validation, les données sont refusées jusqu'à ce que l'utilisateur les corrige.

Ceci évitera de traiter ou stocker des données incomplètes ou invalides.

Validation de formulaire - Exemple Angular

Angular propose un mécanisme de validation de formulaire simple.

Il est possible de spécifier une liste de *validateurs* (interface `Validator`) pour chacun des champs du formulaire. Il est aussi possible de configurer une liste de validateurs pour le formulaire dans son ensemble pour effectuer des validations qui concernent les valeurs de plusieurs champs (exemple: deux fois le même mot de passe).

Si la validation échoue, la fonction `validate` retourne une liste d'erreurs (utile pour afficher le message correspondant). Sinon, la fonction retourne `null`.

Si un seul des validateurs détecte une erreur, le formulaire est invalide.

```
interface Validator {  
    validate(control: AbstractControl<any, any>): ValidationErrors | null  
}
```

<https://angular.io/api/forms/Validators>

Bonnes pratiques - messages d'erreur

- Visibles et bien situés sur la page (le champ en erreur est évident)
- Courts
- Précis
- Utilisent un langage non technique (éviter les codes d'erreur ou trace d'appels)
- Guident l'utilisateur pour régler le problème

Exemple: <https://material.angular.io/components/input/examples#input-errors>

Validation de requêtes

Côté serveur, il est préférable de **valider les données reçues (corps HTTP, paramètres de requête, etc) aussitôt possible**. Ceci permet de détecter les erreurs rapidement et d'assurer aux composants suivants que les données qu'ils recevront seront valides.

Si les données sont invalides, le **serveur doit répondre avec un code d'erreur** approprié (typiquement dans la famille des 4xx).

Il est aussi possible d'ajouter un code et un message d'erreur spécifique à l'application dans le corps de la réponse.

Contrairement aux erreurs de la couche de présentation, ces erreurs seront interprétées par une autre service web ou par l'application frontend et **ne devraient pas être visibles des utilisateurs**.

Gestion d'erreurs inattendues

Jusqu'à maintenant nous avons traité des erreurs détectées lors de validations (frontend et backend).

Dans ces cas, le problème potentiel est prévu par l'application.

Il faut aussi considérer les erreurs *inattendues*, car elles surviendront.

Exemple d'erreurs *inattendues*:


- déréférencer une référence `null`
- atteinte d'une limite du système (nombre de descripteur de fichiers, mémoire)
- erreur sur un appel réseau (perte de connexion)
- bogue

Gestion d'erreurs

Les objectifs de la gestion d'erreurs sont:

- éviter que l'application tombe en panne suite à une ou plusieurs erreurs
- permettre aux utilisateurs de continuer à utiliser l'application web
- capter les erreurs pour pouvoir les corriger

L'important est de ne ***jamaïs ignorer*** ou ***cacher*** une erreur! Au minimum, l'ajouter au *journal* (log).

```
try {  
    this.operationQuiExplose();  
}  
catch (e) {  
      
}
```



Journaux

Si une erreur survient dans une application et que le code l'intercepte (catch), il est important, au minimum, de l'ajouter au journal (log).

Cela permet de répertorier les erreurs et de prendre action par la suite.

Dans le message ajouté au journal, il est utile d'ajouter un maximum de contexte pour permettre de comprendre pourquoi l'erreur s'est produite et, idéalement, pour pouvoir la reproduire.

Par exemple, si le code fait un appel d'API RESTful qui échoue à cause d'une erreur 400 (Bad Request), il est peut être intéressant de savoir:

- de quel appel il s'agit (e.g. POST /products)
- les paramètres de requête (e.g. fromId=12345)
- le corps de la requête (e.g. { "name": "abc", "price": "abc" })
- le code de réponse (e.g. 400 Bad Request)
- la réponse (e.g. { "error": "Invalid value for price" })

Journaux - bonnes pratiques

- format identique de log dans tous vos services
- ajouter les traces d'appel (*stacktraces*)
- *grepable* - information essentielle sur une seule ligne ou *parsable* - json
- ajouter l'id du thread (dans un serveur multi-threaded)
- ajouter l'id de corrélation de la requête (système multi-services)
- choisir le bon niveau de log (voir page suivante)
- rotation des journaux (pour éviter de remplir le disque)
- centraliser les journaux (pour faciliter le débogage)

Journaux - Niveaux

La sélection du bon niveau pour une entrée dans les journaux est très importante. En mode production, pour des raisons de performance, seul les niveaux supérieurs seront activés.

Donc si une erreur importante est inscrite avec un niveau trop bas, elle ne sera pas visible. À l'inverse, trop d'entrées sans importance dans le journal d'un serveur ne font qu'ajouter du bruit (et déclencher la rotation plus rapidement).

Niveau	Description
Trace / Debug	Information pour déboguer l'application en mode développement.
Info	Événements significatifs, mais normaux.
Warn	Information décrivant un état qui est potentiellement problématique.
Error	Erreur importante, mais qui n'empêche pas l'application de fonctionner.
Fatal	Erreur critique. L'application ne fonctionne plus ou en mode dégradé.

Journaux - Performance

Même si l'opération d'insérer une entrée dans les journaux est rapide, **la quantité d'opérations peut éventuellement avoir de gros impacts.**

Pour limiter ces impacts, **le système de gestion de journaux sera configuré pour n'inscrire que les entrées au-dessus d'un certain niveau** (typiquement INFO ou WARN).

Malgré que cela limite les écritures dans le fichier, certaines opérations seront quand même exécutées.

Par exemple, tous les objets temporaires créés lors de la génération de l'entrée peuvent mettre de la pression sur le ramasse-miettes (GC) de Java.

Une bonne bibliothèque de gestion de journaux et une bonne utilisation de l'API peut faire toute la différence.

Journaux - En situation de crise

Il est important d'utiliser les journaux judicieusement.

Selon la bibliothèque de journaux utilisée, il est possible de changer dynamiquement le niveau des entrées qui seront inscrites dans le journal. Il est aussi souvent possible de changer le niveau de certains *Loggers* en particulier (e.g. classe java), ce qui peut être extrêmement utile pour déboguer un problème en production sans avoir à redémarrer le serveur.

Par exemple, si on remarque des erreurs au niveau de l'API de produits et si les entrées (*log*) des niveaux ERROR, WARN et FATAL ne sont pas suffisantes, il est possible d'activer le niveau INFO pour certaines parties du code et ainsi avoir plus d'informations.

Si on activait les journaux au niveau INFO pour toute l'application, il est fort probable que ses performances soient diminuées.

Journaux - Performance

Comparer

```
logger.debug("Logging in user " + user.getName() + " with birthday  
" + user.getBirthdateCalendar());
```

et

```
logger.debug("Logging in user {} with birthday {}",  
user.getName(), user.getBirthdateCalendar());
```

Attention au PII

Les PIIs (**informations personnelles identifiables**) sont des informations qui permettent d'identifier un individu spécifique.

Il faut éviter de mettre ces informations dans les journaux, car les journaux peuvent aussi être la cible de cyberattaques... c'est aussi une question d'intimité.

```
[main] INFO LoginController - Login with username: alice and  
password: abcd
```

...

```
[main] INFO MessagesController - New message from [alice]:  
Bob, appelle moi au 555 444 7777.
```


Traces distribuées

Dans une architecture par services, il est souvent utile de pouvoir suivre une requête du client au travers des différents services.

De la même façon qu'il est possible d'insérer l'id d'un thread, il est aussi possible d'insérer un id de corrélation unique (GUID) qui sera généré à l'entrée du système et ajouté à chaque appels d'API subséquents (via les en-têtes HTTP). Si l'id de corrélation est ajouté à toutes les entrées de journaux, il sera possible de suivre l'évolution de la requête dans le temps à travers les journaux des différents serveurs.

Ce sera encore plus utile si combiné avec un système d'agrégation de journaux qui centralise l'ensemble des journaux dans un engin de recherche pour faciliter le suivi.

Suivi d'erreurs dans les applications frontend

Pour les applications frontend qui s'exécutent sur le navigateur de l'utilisateur, il est difficile de détecter les erreurs qui se produisent.

Pour se faire, il existe des solutions qui permettent de répertorier les erreurs survenues dans votre application frontend. Typiquement, il faut ajouter un gestionnaire d'erreur qui intercepte les erreurs non gérées dans votre application (ErrorHandler pour Angular) et qui les fait suivre à un système d'agrégation d'erreurs.

Ensuite, l'équipe de développement peut analyser les erreurs et les réparer rapidement.

La résilience

La **résilience** d'un système est sa capacité à opérer et à récupérer en présence d'erreurs.

Stratégies de résilience aux erreurs

Que ce soit un appel d'API provenant du frontend ou entre différents services du backend, les erreurs doivent être gérées.

Lorsque le serveur retourne un message d'erreur spécifique, il est parfois possible de prendre une action précise. Par exemple, sur un 403, il est possible de rediriger l'utilisateur vers la page d'authentification.

Mais dans la majorité des cas, le client ne peut pas automatiquement prendre action et peut seulement faire remonter l'erreur (dans la chaîne d'appels) ou afficher un message d'erreur à l'utilisateur (si l'erreur est reçue dans le frontend).

Il existe par contre un ensemble de stratégies standards simples pour gérer des erreurs qui permettent à l'application d'être plus résiliente.

Délai d'inactivité (Timeout)

Une bonne pratique à utiliser lorsqu'on fait un appel à un API, est de toujours spécifier un délai maximal au-delà duquel, le client agira comme si la requête avait échoué.

Ceci permet de libérer des ressources rapidement (important en backend) et de ne pas laisser l'utilisateur devant une page qui semble se charger indéfiniment.

Reprise (Retry)

Pour certains types d'erreurs qui peuvent être temporaires, la stratégie de reprise est efficace.

Par exemple, un client HTTP pourrait réessayer si

- le délai d'inactivité (timeout) est expiré (serveur inactif)
- sur un code 408 (request timeout), 429 (too many requests), 500 (internal server error), 502 (bad gateway), 503 (service not available), 504 (gateway timeout).

Dans les autres cas, il est souvent inutile de réessayer. Par exemple, sur un code 400 (bad request), il est peu probable que le serveur accepte la même requête la 2^e fois.

Intervalle exponentiel entre les tentatives

Dans le cas d'une erreur temporaire du côté serveur, il est possible que plusieurs clients effectuent une nouvelle tentative dans un intervalle de temps assez court.

Si les clients essaient plusieurs fois, la charge du serveur augmentera de façon dramatique. Si le serveur était déjà dans un état instable, il est possible que ce soit le coup de grâce. 🧟

Pour éviter ce genre de situation, **le client limite son nombre de reprises et injecte un délai entre chaque essai.** Le délai peut-être fixe (2 secondes), linéaire (2s, 4s, 6s) ou exponentiel (2s, 4s, 8s). Il est aussi possible d'ajouter une légère variation aléatoire pour distribuer la charge ($2s \pm \text{rand}(0, 1)$).

Une des avantages importants du mécanisme de reprise, est que le système récupère automatiquement d'une erreur ou d'une panne temporaire et, parfois, sans que l'utilisateur ne s'en rende compte.

Plan B - Fallback

Dans certains cas, il est aussi possible de gérer un cas d'erreur en utilisant une valeur par défaut ou en effectuant un appel différent sur un autre système.

Stratégies de résilience système

Nous avons vu des stratégies de résilience applicables aux erreurs qui peuvent survenir lorsque les composants d'un système communiquent entre eux.

Nous allons maintenant voir des stratégies applicables au niveau du système.

Redondance

Une stratégie simple consiste à intentionnellement créer de la redondance (plusieurs instances du même service) dans le système. Cette stratégie requiert l'utilisation d'un balanceur de charge qui distribue les requêtes entre les multiples instances d'un même service.

Par exemple, si, pour la charge sur notre système, nous avons besoin de 2 instances d'un service spécifique, nous allons en déployer 3.

De cette façon, si un serveur tombe en panne, l'application pourra continuer de fonctionner le temps que la panne soit réglée.

Identifier les limites

Les ressources d'une application sont souvent limitées (mémoire, CPU, disque, descripteurs de fichier, etc).

Pour éviter qu'un composant entre dans un état instable, il est préférable de réagir automatiquement avant que la limite du système ne soit atteinte.

Il est bon de définir un seuil à partir duquel il est préférable de faire échouer délibérément les requêtes (*graceful degradation*) et d'envoyer une alerte plutôt que de laisser le composant tomber en panne ou dans un état instable (*zombie*).

Attention aux valeurs par défaut des bibliothèques et cadriciels! Parfois les valeurs sont trop basses (goulots d'étranglement) ou trop hautes (ou aucune limite!)

Exemple - Heap Java

Il est important de bien définir la taille de la heap Java.

Pour une application serveur, on spécifie la taille initiale (Xms) égale à la taille maximale (Xmx).

- Ceci limite les GCs au départ et évite la réallocation de mémoire en cours de route.
- La limite maximale assure qu'il n'y aura pas de surprise à mesure que la charge augmente.

Attention aux objets alloués hors heap et aux autres ressources qui pourraient consommer de la mémoire sur le serveur.

On veut éviter l'utilisation de la mémoire swap à tout prix!

Nombre de requêtes simultanées

Selon le cadriciel utilisé, il est important de comprendre comment sont gérés les requêtes.

Est-ce que le cadriciel utilise de l'I/O bloquant ou non?

Quelle est la taille du pool de threads pour gérer les opérations?

Si on atteint la limite du nombre de connexions ou du nombre de threads, est-ce que le cadriciel place les requêtes en attente dans une file?

Quelle est la limite de cette file?

Une fois cette limite atteinte, il se passe quoi?

Exemple - Tomcat

Each incoming, non-asynchronous request requires a thread for the duration of that request.

*If more simultaneous requests are received than can be handled by the currently available request processing threads, **additional threads will be created up to the configured maximum (the value of the `maxThreads` attribute).***

*If still more simultaneous requests are received, Tomcat will accept new connections **until the current number of connections reaches `maxConnections`**. Connections are queued inside the server socket created by the Connector until a thread becomes available to process the connection.*

Once `maxConnections` has been reached the operating system will queue further connections. The size of the operating system provided connection queue may be controlled by the `acceptCount` attribute.

*If the operating system queue fills, **further connection requests may be refused or may time out.***

Exemple - ThreadPoolExecutor

Core and maximum pool sizes

A `ThreadPoolExecutor` will automatically adjust the pool size according to the bounds set by **`corePoolSize`** and **`maximumPoolSize`**.

When a new task is submitted in method `execute(Runnable)`,

- **if fewer than `corePoolSize` threads are running, a new thread is created to handle the request**, even if other worker threads are idle.
- **else if fewer than `maximumPoolSize` threads are running, a new thread will be created to handle the request only if the queue is full.**

By setting `corePoolSize` and `maximumPoolSize` the same, you create a fixed-size thread pool.

Exemple - ThreadPoolExecutor

Queuing

Any `BlockingQueue` may be used to transfer and hold submitted tasks. The use of this queue interacts with pool sizing:

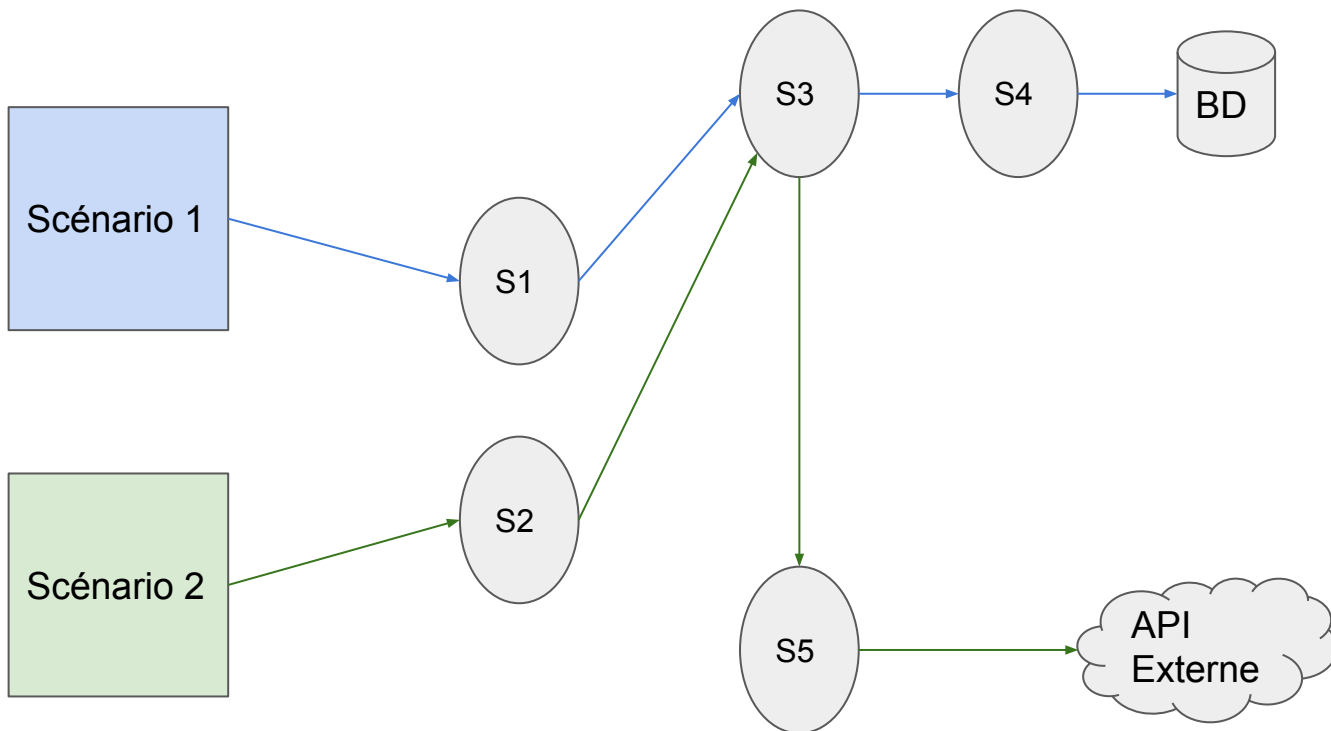
- If fewer than `corePoolSize` threads are running, the `Executor` always prefers adding a new thread rather than queuing.
- If `corePoolSize` or more threads are running, the `Executor` always prefers queuing a request rather than adding a new thread.
- If a request cannot be queued, a new thread is created unless this would exceed `maximumPoolSize`, in which case, the task will be rejected.

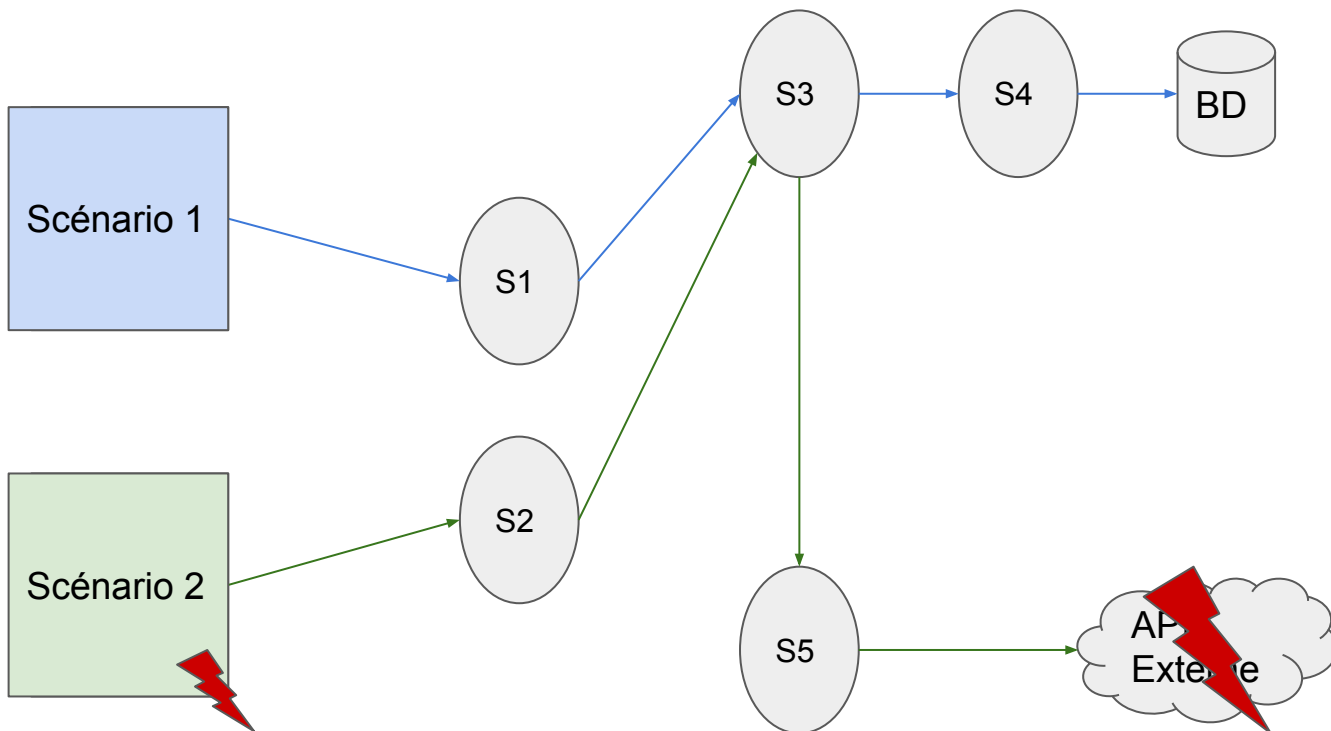
There are three general strategies for queuing:

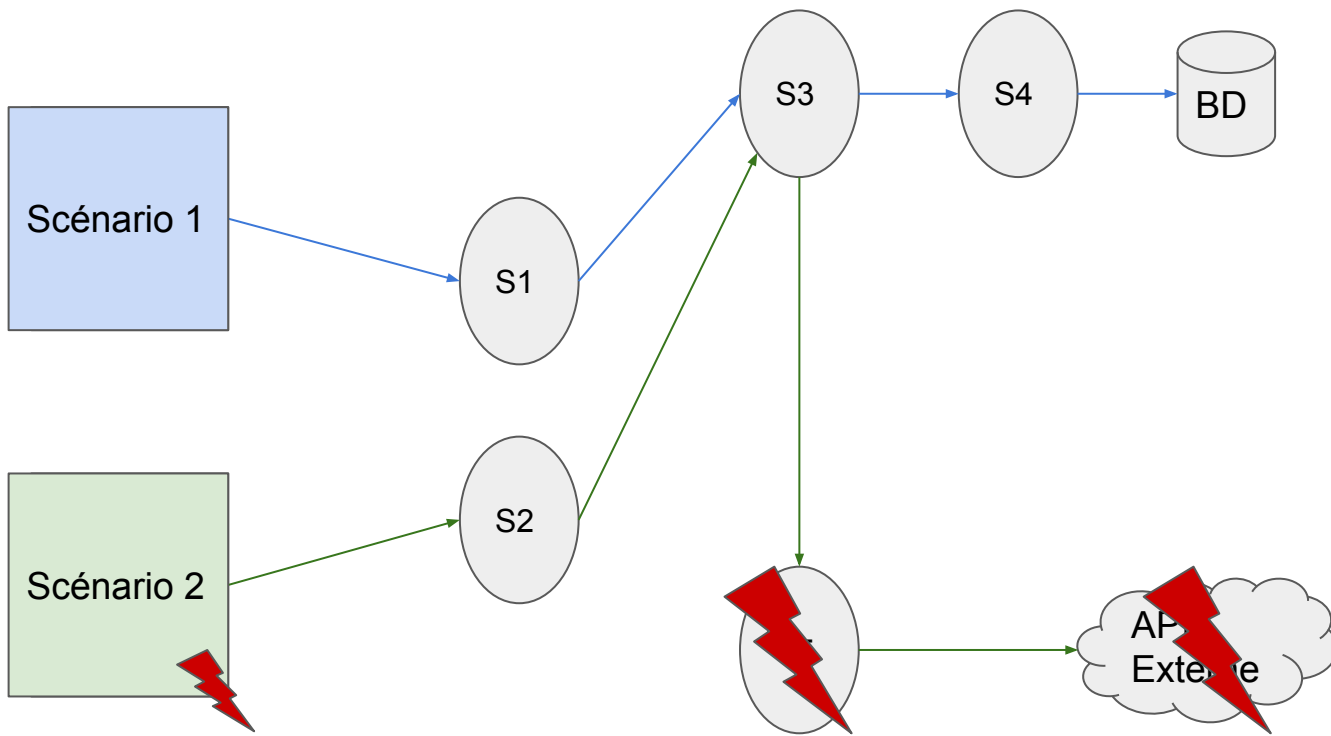
1. *Direct handoffs.* A good default choice for a work queue is a `SynchronousQueue` that hands off tasks to threads without otherwise holding them. ...
2. *Unbounded queues.* Using an unbounded queue (for example a `LinkedBlockingQueue` without a predefined capacity) will cause new tasks to wait in the queue when all `corePoolSize` threads are busy....
3. *Bounded queues.* A bounded queue (for example, an `ArrayBlockingQueue`) helps prevent resource exhaustion when used with finite `maximumPoolSizes`, but can be more difficult to tune and control....

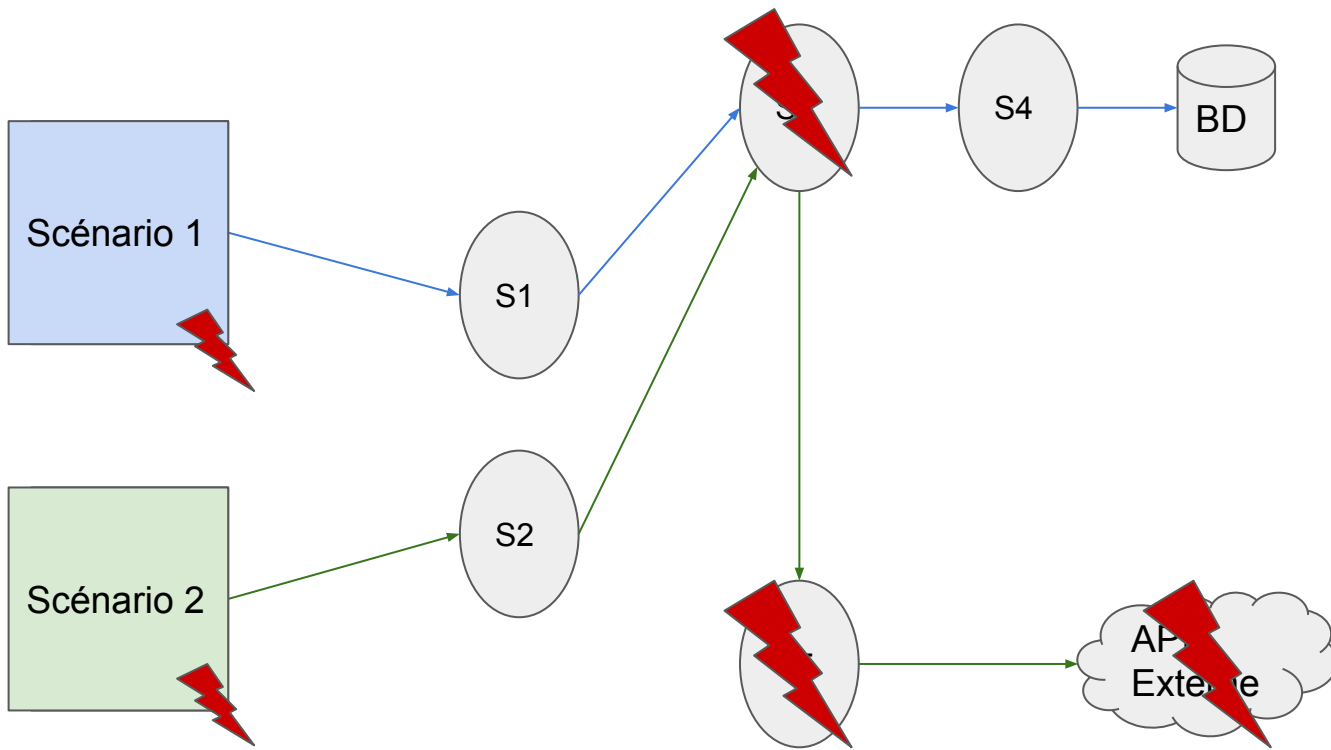
Défaillance en Cascade (Cascading Failure)

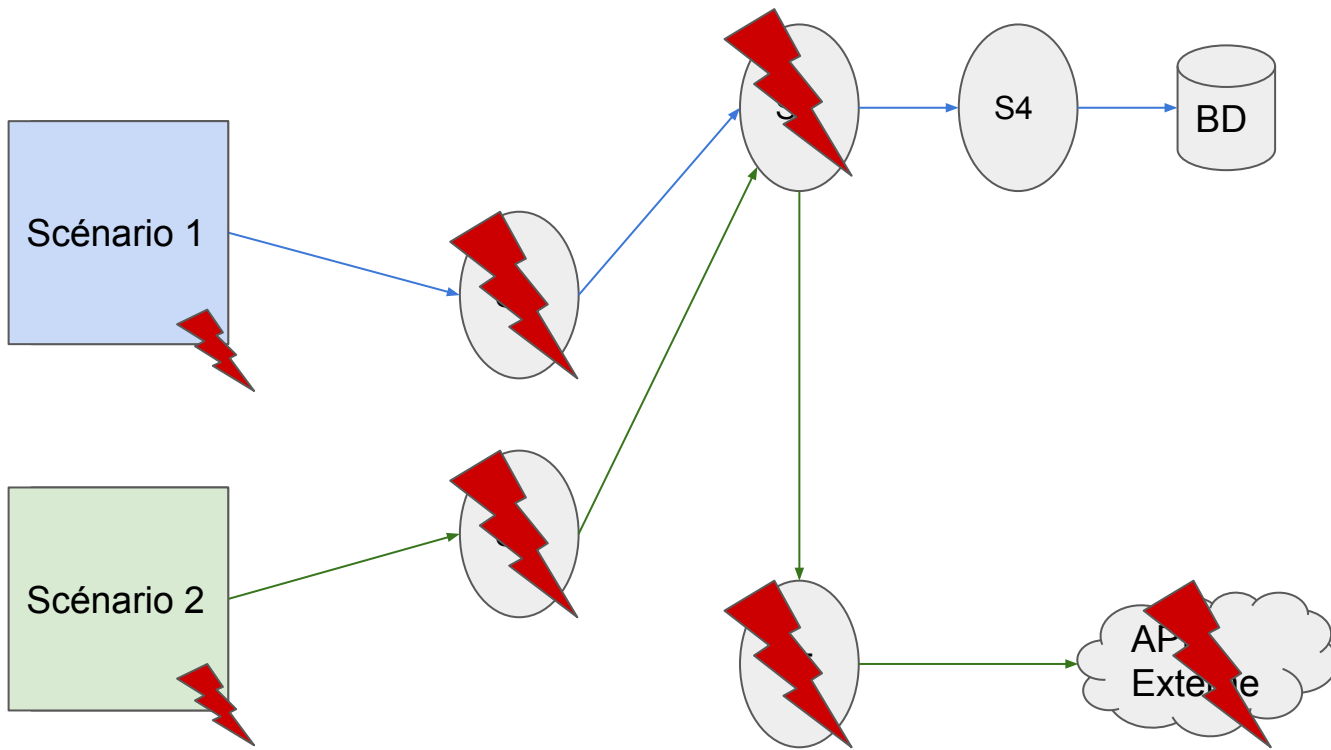
- Une défaillance en cascade survient lorsque la défaillance d'un composant du système entraîne la défaillance d'autres composants qui lui sont liés, aggravant ainsi la sévérité du problème.





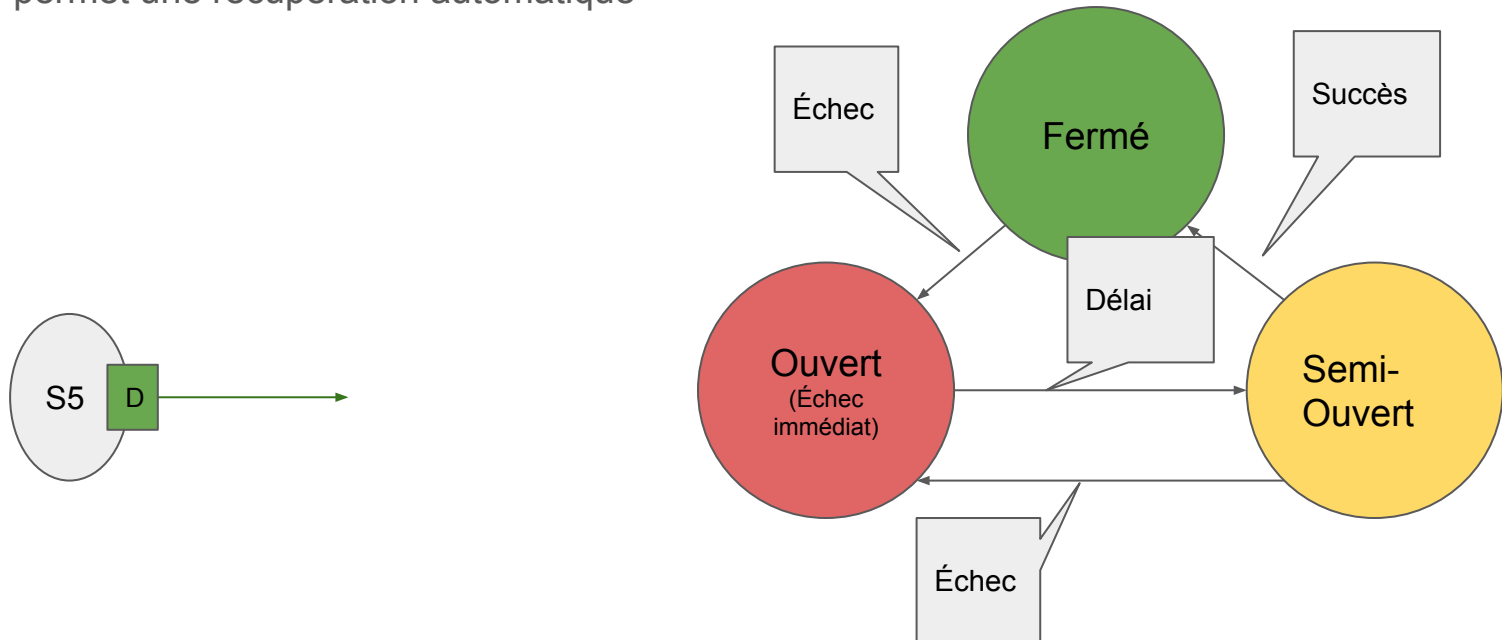


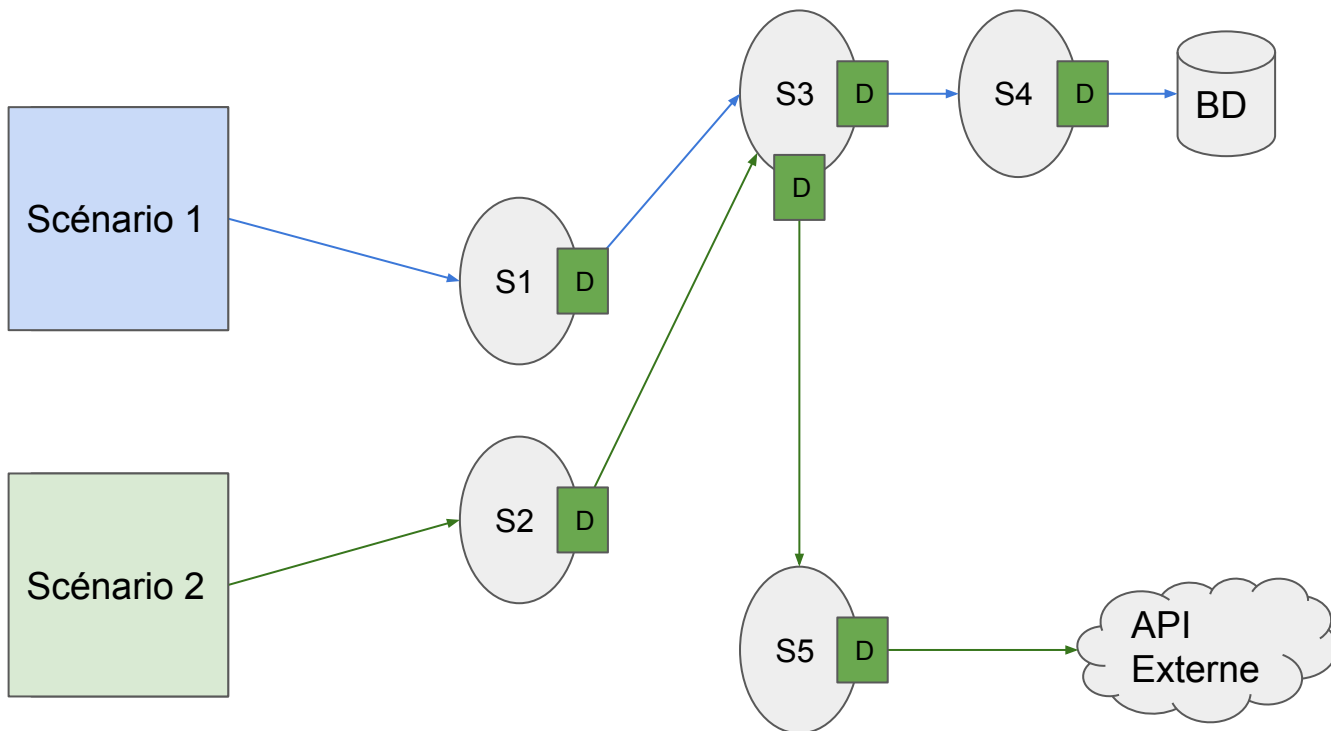


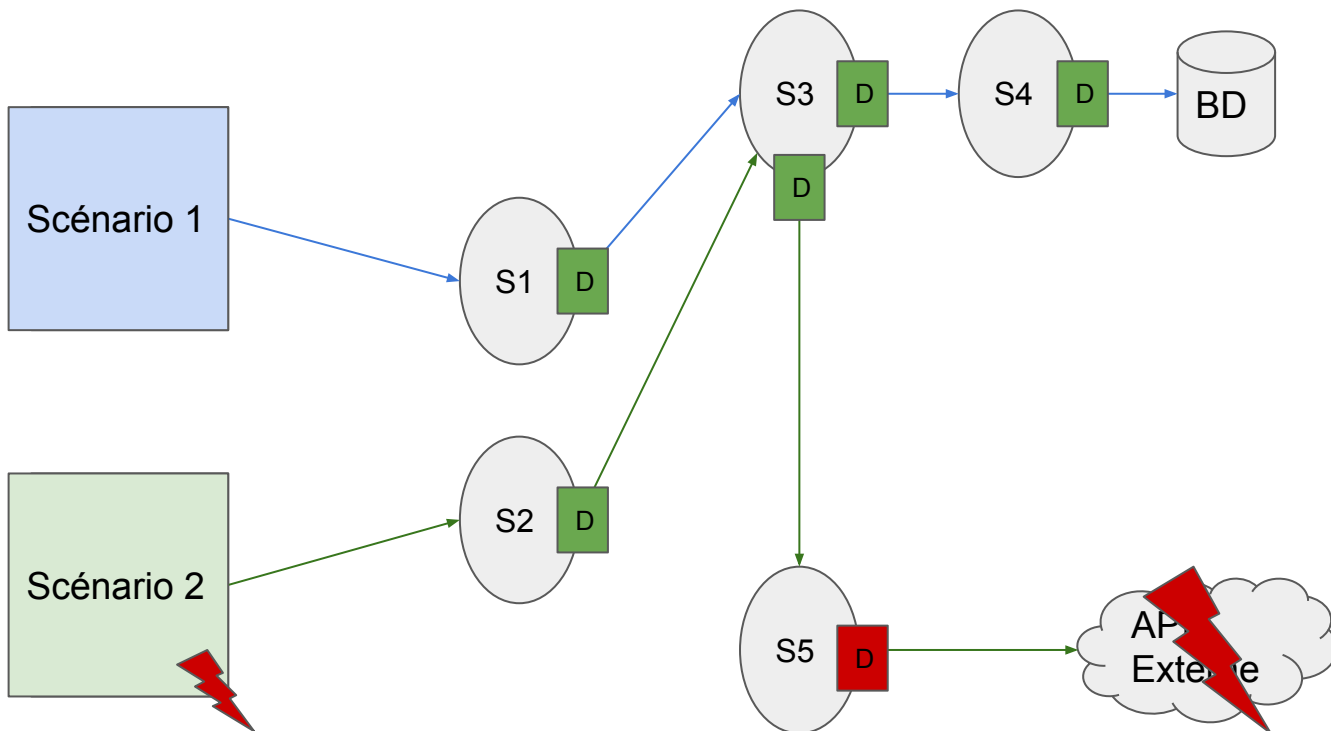


Solution: Disjoncteur (Circuit Breaker)

- Mécanisme utilisé pour limiter les impacts d'un problème et éviter une défaillance en cascade.
- Un outil simple qui protège un composant des échecs de ses dépendances (filtre).
 - limite les impacts de la défaillance
 - évite de surcharger un composant instable
 - permet une récupération automatique







Comment tester la résilience?



Chaos Engineering

Le chaos engineering est une discipline de test où l'on injecte des perturbations aléatoires dans un système et on valide que le système gère bien les erreurs.

L'idée vient de l'équipe d'ingénieurs de Netflix (2011).

Le projet consistait simplement à mettre hors service des serveurs au hasard en production(!) pour voir comment le système réagit et découvrir des problèmes. L'outil s'appelait le Chaos Monkey.

Il existe des variations qui injectent différents types de perturbation tels que de la latence réseau, un surplus d'utilisation de CPU ou simule un disque plein.



Simulation de désastre

Même si un système fait régulièrement des sauvegardes, les mécanismes de récupération sont rarement validés.

Si jamais un problème survient, ce n'est pas le moment de se rendre compte que la procédure de récupération ne fonctionne pas.

Il est donc important de tester la procédure de récupération (pour valider vos sauvegardes, mais aussi vos étapes de récupération) avant qu'un désastre arrive.

Typiquement, pour tester la récupération, on crée un environnement qui duplique l'environnement de production et on simule une perte de données, de réseau, etc.

Ceci demande généralement beaucoup de ressources et de temps.

IDENTIFIER L'ERREUR



TU DOIS

Post-mortem

Lorsqu'un problème survient dans un environnement de production et que ce problème a de graves conséquences (temps de panne, perte de données, etc), il est important de bien en analyser les causes pour éviter que le même problème survienne à nouveau.

Souvent, dans le feu de l'action, la première étape consiste à trouver une solution à très court terme (workaround) qui ne demande pas ou très peu de changements.

Une fois le problème sous contrôle, la deuxième étape consiste à trouver une solution à long terme (un vrai correctif) pour le problème en question.

Enfin, il convient de comprendre pourquoi ce problème est survenu, identifier la cause et de déterminer les actions à prendre pour la suite. Par exemple, si c'est une erreur humaine qui a déclenché la panne, est-il possible d'automatiser cette partie du processus?