PUBLIC

2021-12-02

Developing Resilient Apps on SAP BTP

© 2021 SAP SE or an SAP affiliate company. All rights reserved.

Content

1 About this Guide. 4

2 What is Resilient Software Design. 5

2.1 Availability. 5

2.2 Failure Types that Affect the Availability. 6

2.3 Approaches Towards Stability. 6

The Determinism Approach. 6

The Resilience Approach. 7

2.4 Resilience Principles. 7

Isolation. 7

Redundancy. 7

Loose Coupling. 8

Fallback. 8

3 Applying Resilience Principles. 10

3.1 Failure Units. 10

Isolation. 10

Redundancy. 10

Recursion. 11

Supervisor. 12

Update. 12

3.2 Data Handling. 12

CAP Theorem. 13

Databases. 14

Cache. 14

Types of Data. 15

Transaction. 16

Update. 16

3.3 Control Circuit. 16

Metrics. 17

Logging. 17

Patterns for Error Detection. 17

Patterns for Error Recovery. 18

3.4 Message Driven. 18

Message-Handling Patterns. 19

Resilient Protocols. 19

3.5 Dependency Management. 19

Isolation of Dependencies. 20
Graceful Degradation. 20
Request Collapsing. 21
Service Discovery. 21
3.6 Rate Limiting. 21
Client Isolation. 22
Request Throttling. 22
Scaling. 23
Monitoring. 23

4 Resilience Patterns. 24
4.1 Failure Unit: Unit Isolation. 27
4.2 Data Handling: Temporary Replication. 32
4.3 Control Circuit. 37
Quarantine. 37
Supervisor. 40
Watchdog. 43
4.4 Dependency Management. 46
Circuit Breaker. 46
Retry. 48
Timeout. 50
4.5 Rate Limiting. 52
Shed Load. 52
Bounded Queue. 55

5 How to Make Your Apps Resilient. 59
5.1 Resilience Planning. 59
5.2 Resilience Maturity Monitoring. 60
5.3 Resilience Testing. 63

1 About this Guide

Develop resilient apps on top of SAP BTP.

Resilience is the main cloud quality besides security. This guide addresses developers, architects, and project teams in cloud development, and shows how to develop resilient apps on top of SAP BTP. Use it either to become familiar with the topic as a whole or to gain a deeper insight into single patterns or tools.

2 What is Resilient Software Design

Become familiar with the basic concepts of resilient software design.

> **i Note**
>
> This chapter is based on the article Eine kurze Einführung in Resilient Software Design by Uwe Friedrichsen .

System availability is essential for the value of software. If IT systems are not available or do not run correctly, customers are dissatisfied and not willing to spend money on them. Resilience, or resilient software design, is about handling failures that occur in complex system landscapes during runtime and ideally should not be noticed by the users. As opposed to traditional stability approaches, its goal is not to reduce the probability of failure occurrence, but to maximize the availability of systems and system landscapes. It accepts the unavoidability and unpredictability of failures and focuses on dealing with them as quickly as possible.

2.1 Availability

Availability is defined in the following equation: A : = MTTF / (MTTF + MTTR)

This formula contains the following elements:

Elements in the Availability Equation

Element	Definition
A	The availability of a system or a system landscape
MTTF (mean time to failure)	The average time that elapses from the start of a system operation until the occurrence of a failure
MTTR (mean time to recovery)	The average time that elapses from the occurrence of a failure to the recovery of the system operation

In the equation, the denominator (MTTF + MTTR) describes the overall time, while the numerator (MTTF) describes the time during which a system works properly. Therefore, the value for A can range from 0 for "not available at all" to 1 for "always available." Multiplied by 100, this value shows the system availability in percentage terms.

2.2 Failure Types that Affect the Availability

We distinguish between five different failure types that affect the system availability:

Failure Types that Affect the Availability

Failure Type	Definition
Crash failure	A system has ceased to respond permanently, but was working properly until the crash occurred.
Omission failure	A system does not react to (individual) requests, either because it has not received the request or because it is not sending a response.
Timing failure	The response time of a system exceeds a given time interval.
Response failure	The response by the system is incorrect.
Byzantine failure	A system provides random responses at random time intervals.

2.3 Approaches Towards Stability

There are two approaches towards the maximization of stability:

Approaches Towards Stability

Approach	Definition
Determinism approach	Maximizing the value of MTTF
Resilience approach	Minimizing the value of MTTR

2.3.1 The Determinism Approach

The traditional determinism approach is to increase the value of MTTF to such an extent that MTTR becomes insignificant, which is to delay the occurrence of failures for as long as possible. Determinism requires large investments at the infrastructure level, such as deploying redundant hardware, setting up high-availability clusters, and using network connections, as well as complex procedures for avoiding failures on the software level. Its idea is to anticipate possible failure sources and to take active measures to minimize the probability of their occurrence.

The determinism approach is reflected in the established standards for software quality. Its underlying assumption is that it is possible to continue the basic conditions for system operation deterministically. This approach is entirely valid as long as your system is relatively isolated and there are only a few factors that influence the system availability. The problem is that nowadays, we are dealing with highly complex distributed

system landscapes, and nearly every system in a company is a distributed system. In these systems, there are so many potential failure sources that it is impossible to predict and prevent all of them. There are no longer deterministically definable conditions, but probabilities, which requires a new approach towards stability.

2.3.2 The Resilience Approach

Resilient software design does not to try to avoid failures. It is assumed that they simply occur and cannot be prevented or predicted. Therefore, MTTF is accepted as a factor that cannot be influenced. Instead, the resilience approach tries to minimize MTTR, which is the time that elapses between the occurrence of a failure and its correction. Resilience is the ability of a system to handle unexpected situations without users noticing or to react with a graceful degradation of service.

2.4 Resilience Principles

Become familiar with the basic resilience principles, the questions you as architects should ask yourself, and the aspects you should consider if you want to design a resilient application.

2.4.1 Isolation

Isolation is the basic principle of resilience, on which almost all other resilience patterns build. Its idea is that a system should never break down entirely. To achieve this, divide your system in as many autonomous units as possible, and isolate them from each other. These units are called bulkheads, failure units, or units of migration, and isolate themselves upstream and downstream against failures from other units to prevent cascading failures.

2.4.2 Redundancy

Redundancy is a powerful pattern that can be used in a wide range of situation, but it requires careful implementation and a significant amount of preliminary conceptual work. To use redundancy, first clarify the scenario you want to address:

- Do you want to implement a failover? If one unit fails, do you want to switch entirely and transparently to another unit?
- Do you want to keep latency to a minimum? Do you want to use redundant units to reduce the probability of a slow response?
- Do you want to recognize response failures? Do you want to evaluate the responses of several redundant units that maybe run on different hardware and have been developed independently of each other to reduce the probability of erroneous responses?
- Do you want to implement load distribution? That is, to take requests that are too numerous for one unit to handle and distribute them across several units?

- Do you want to do something completely different?

Depending on your scenario, take additional aspects into consideration:

- Which routing strategy fits with your scenario if you want to use load balancing? Is a simple round robin sufficient or do you need a more complex strategy? If so, does your load balancer support this?
- Does your system make sure that the master is changed automatically if you are using the master-slave approach for failover and the master fails?
- If you want to keep latency to a minimum, do you want to use a fan out & quickest one wins approach? That is, to send your request to several redundant units and use the fastest response. In this case, how do you ensure that the units are not so overloaded with old and obsolete requests that they can no longer quickly answer new requests?
- Which aspects do you want to automate, and which ones do you want to handle manually? The general recommendation for resilience says that you should automate failure correction as completely as possible, however, the system administrator must be able to intervene the process, if necessary.
- How do you ensure that the administrator can always monitor the current system situation? Which unit is master and which one is slave? How many units are running? Where are they running?

2.4.3 Loose Coupling

Loose coupling is a basic principle for preventing cascading failures. One way to implement loose coupling is to use asynchronous event- or message-based communication, with which you can maximize the decoupling of individual units and minimize the probability of self-propagating latency failures. To use this principle, use patterns like timeout and circuit breaker to implement good latency monitoring. Libraries like Hystrix by Netflix are helpful for the implementation.

Another helpful pattern is idempotency. A modifying call is idempotent if repeated calls have no additional side effects. If you do not use idempotent calls, it is either not possible or very difficult to decide if you can send a call again, if responses are too slow. With idempotent calls, you can call your receiver as often as necessary until you receive the response that the call has been processed successfully. They enable the change from an exactly once communication approach to an at least one approach, which is much easier to implement and places significantly lower demands on the communication infrastructure.

2.4.4 Fallback

It is not sufficient to only recognize failures, but you also need a clear strategy of what to do when one has been detected. Ideally, users should not even notice that a failure has occurred. The best way to avoid new defects while trying to correct one, is to let fault correction run automatically.

The central question to ask yourself when it comes to fallback is: How should the system react when a failure is detected? Take various aspects into consideration:

- Should the user be notified about the error and asked to try it again later?
- Do you hold the user data in caches? Then you can take care of the read requests from the caches and handle write requests with a "Please try again later."
- Do you send requests to a queue that is processed by an asynchronous job as soon as the database is available again? Then you can tell notify the user that the request has been received and will be processed later.

- Should the system do something else?

3 Applying Resilience Principles

Identify and apply the right resilience patters for your service.

The resilience patterns are clustered into groups. To find the right patterns for your service, either check its requirements or identify the area of possible problems. If there are known issues, it is easier to determine the area and find the helping patterns.

3.1 Failure Units

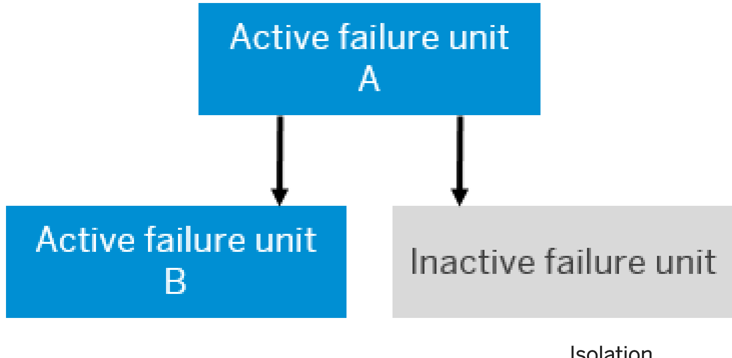
Learn about the different characteristics of failure units.

In monolithic software systems, it was easy to define the various system pieces. However, the cloud offers new boundary conditions for stability, and splitting the monolithic block into smaller pieces becomes increasingly accepted in the service development.

A failure unit has the following characteristics:

3.1.1 Isolation

Isolation is one of the most important resilience principles. Failure units are independent from each other, which means that if one failure unit stops working, others can still continue. Additionally, a failure works in isolation, which means that the execution does not depend on data or the availability of another failure unit.



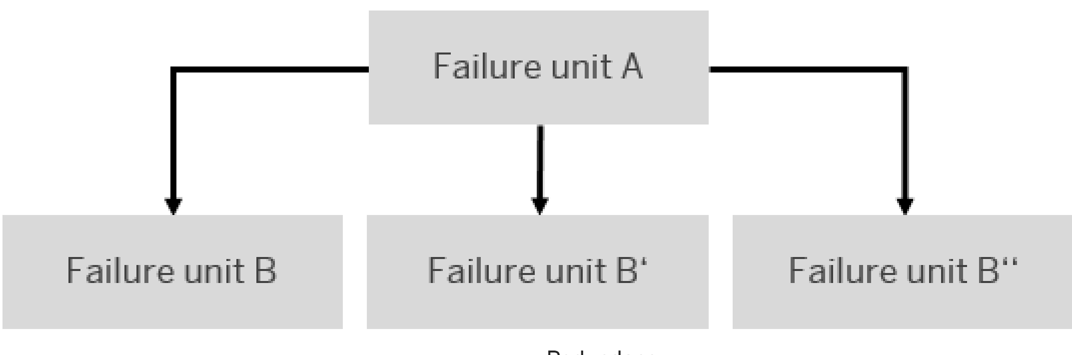
Isolation

3.1.2 Redundancy

Set up failure units in a redundant way to fully ensure against failures. There are different types of redundancy, including the following:

Types of Redundancy

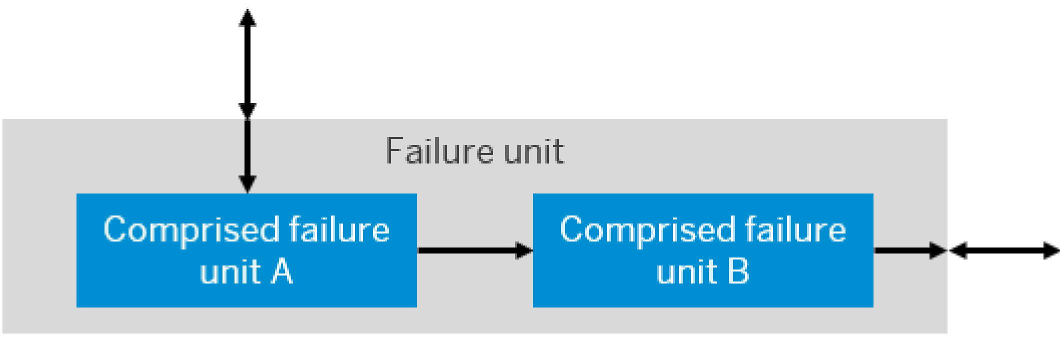
Type of Redundancy	Definition
Failover	If one unit fails, another one takes over transparently.
Load balancing	Failure units can be scaled to run additional requests. Load balancing is related to high availability.
Quorum voting	Multiple failure units can be used to calculate a correct answer. This mechanism is used to identify hardware damage. If one out of three identical devices provides a different answer, it is assumed to be "wrong" (damaged).



Redundancy

3.1.3 Recursion

Failure units can be implemented recursively. This means that a single failure unit is internally based on smaller failure units. One example of this recursion is a microservice that runs on a VM and uses a runtime that starts and stops multiple actors within. The microservice itself is a failure unit, and so is every single actor. Therefore, redundancy and failover kick in on different levels. The failure unit is a software design concept rather than a technology, and depends heavily on the software design.



Recursion

3.1.4 Supervisor

A supervisor is a component that controls other components and observes their status. These components build a failure unit, because they can fail, work independently, and are isolated. When a failure unit fails, the supervisor acts accordingly. There are different options, depending on the kind of problem:

- If the initial set of resources was insufficient, the supervisor puts a task in quarantine to provide additional resources for the execution.
- The supervisor limits the number of retries. If an action was retried several times, its execution should be handed over to a human being.
- If the supervisor notices that the execution has failed, it rolls back one or more transactions.

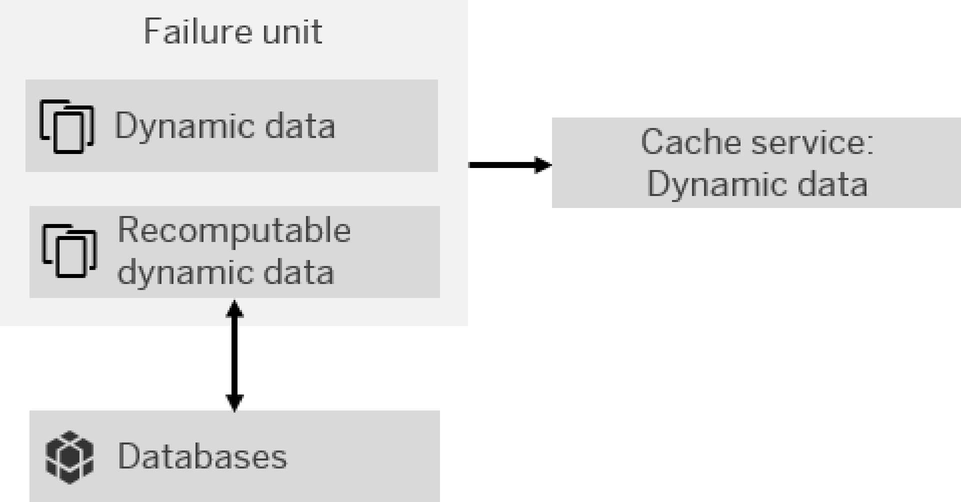
3.1.5 Update

Resilient system architecture implies that failure units can be updated without any planned downtime. This process is similar to the one during a failure, because if a failure crashes, a new instance must be started immediately. An update extends this behavior to start the unit using a new version of the software. A unit can start as a failover unit only if the new version is compatible. This means that the evolution of a failure unit must be done in a compatible way.

3.2 Data Handling

A service needs data to perform meaningful action. If multiple failure units are involved, data can be distributed across multiple failure units, which makes it more difficult for cloud services to handle it. Usually, microservices store data close to them and do not allow access from the outside, which is known as local persistency. Even within a microservice, data handling can be problematic, because this design implies that multiple microservices contain data replicas, which makes it difficult to ensure consistency. The failure unit owns the

data and must therefore control its handling. The following figure depicts the isolation principle for a failure unit:



Data Handling

Here, the databases indicate that there is persisted data that survives all outages, because it is not stored in the failure unit itself. Additionally, the databases include the technology to perform backups and restore functionality. The dynamic data is the data the failure unit holds in memory to compute the result. If the failure unit crashes, the data can be recomputed from the persisted data. Non recomputable data is lost without any way to retrieve it.

3.2.1 CAP Theorem

The CAP theorem plays a major role in all design decisions of a failure unit. It states that data storage can achieve only two qualities of the following three:

Data Storage Qualities

Data Storage Quality	Definition
Consistency	Currently, NoSQL storage technologies do not implement a common data quality like ACID (Atomicity, Consistency, Isolation, Durability), which is traditionally found in relational databases. The data quality of such storage technologies is called BASE (Basically Available, Soft state, Eventual consistency).
Availability	Availability is a cloud quality that usually cannot be compromised.

Data Storage Quality	Definition
Partition tolerance	As soon as the data becomes huge or is replicated for high availability, it is partitioned. The software system must now guarantee that the data is provided independently of the storage location.

With partitioned data, a service must choose either availability or consistency. The service must be available, so the technology compromises on consistency, which leads to increasing complexity on the consumer side. It is not guaranteed that the provided data is consistent, so consumers must prepare for an eventual consistency. For cloud services, it is important to understand the qualities that are provided by a given technology. The service must handle the data the way that suits its requirements best.

3.2.2 Databases

Some years ago, relational databases were widely used and adequately met the requirements regarding performance, data volume, and so on. However, cloud services require much more data and must handle many more requests than earlier systems. To fulfill these new requirements, a cloud service must use the appropriate technology. NoSQL databases have been developed to handle heavier workloads and more requests. NoSQL technology includes the following specific database types:

- Key-value store, such as Redis on SAP BTP
- Column-oriented store
- Document-oriented store, such as MongoDB on SAP BTP
- Graph database
- Relational database, such as PostgreSQL on SAP BTP

Best practices for developing cloud software recommend choosing the appropriate technology for the purpose. Each failure unit must be analyzed according to specific requirements and use the technology that best fits. There is no single technology that is suited for all use cases.

3.2.3 Cache

If functionality is split into failure units, the amount of communication between units increases. Such a split is unavoidable, and it is always a trade-off to find the right balance between using the appropriate failover mechanism and creating the additional network overhead. However, there are definitely less efficient ways to set up failure units. For example, services that do not only request values from another service but also actively poll to see if the data has changed in the called service might indicate that the failure unit domains are not split in the most efficient or practical way.

The use of caches can help reduce the network load. Caching for a failure unit is generally done either locally or using a central cache.

Types of Cache

Type of Cache	Definition
Local cache	Data is held in the failure unit, and the unit itself is responsible for the data life cycle, that is, when the data expires and should be removed from the cache. This is a critical issue, because a failure unit cannot generally determine when values become outdated. If you cache data locally, also consider using "smart" eviction strategies, such as defining specific events that update values and using retries in case of problems.
External cache	Using an external service to store data potentially improves the situation a lot. An external service can be used for the precomputed data if the computation is not performed again and again by the remote service or if the data, for example, a session, can be reused across multiple units to handle failures. There are multiple technologies in this area, such as in-memory databases (for example, Redis) or distributed caches (for example, Hazelcast).
Distributed cache	If entries in a cache have to be shared across multiple instances of a system, the data is transferred across the network. However, setting up a secure communication between nodes is difficult and can lead to eventual consistency. Also, the cache quality depends heavily on the quality of the network. If you want to use distributed caches, make sure to apply proper testing for the data synchronization to avoid split brain problems.

3.2.4 Types of Data

Categorization is an important area for data handling, because data is not always treated the same way. For example, static data can easily be cached, because changes are infrequent. However, data that is used in customer transactions is always new, because it is constantly changed by customer processes. These two types of data are treated differently, often even by using different technologies to handle each of the types. Data can be grouped in the following areas:

Types of Data

Type of Data	Definition
Static data	Data that changes infrequently.
Scratch data	Drafts of data.
Dynamic data	Data that changes often.

Type of Data	Definition
Recomputable data	Data that is generally processed and provides output.
Non recomputable data	Critical data, because failure leads to data loss.

3.2.5 Transaction

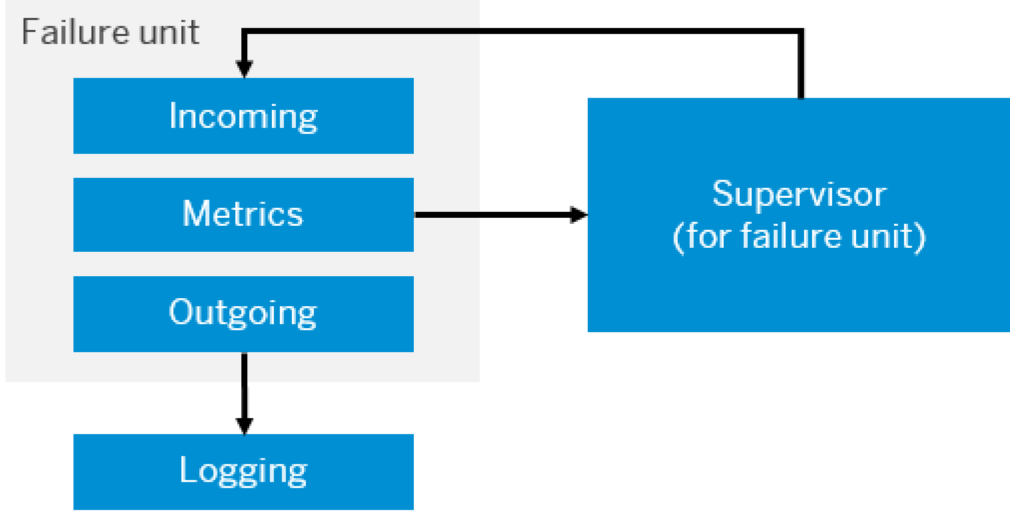
In addition to categorization, it is also important to define the life cycle of data across multiple failure units. If a unit fails, the data in a transaction is lost and in principle, the transaction must be rolled back and retried, which requires a lot of synchronization. Currently, the trend is more toward splitting transactions into smaller parts. If a failure or an error occurs, other failure units must compensate for the already applied changes, because, in general, these changes make only sense for the complete task that could not be executed.

3.2.6 Update

Persisted data must participate in the life-cycle management of the entire system. This means that updates to failure units may change the way the units interact with the data. Persisted data generally contains more data in a different structure than data which isn't persisted. Changes in data structures must be defined in a way that is compatible with the failure units. Incompatible changes can create transformation work that is difficult to accomplish without incurring downtime.

3.3 Control Circuit

A software system should run without any problems and serve its purpose. That is the overall goal, but reality shows that based on load, resource usage, and so on, it is very difficult for the software to react to all changes. The current state of the art is that there are human beings as operators, who monitor the system and intervene in case of problems. Sometimes, alerts are even thrown by the system to trigger specific actions, for example, the addition of new compute power, if more load is coming. To achieve high availability, it is required to automate as much as possible in this area, because the time of human interaction is simply too long to rely on or no downtimes. Therefore, it is required to apply a well-known pattern from engineering: a control circuit. The control circuit feeds back some of the output to the input of a system. For a software system, this means that the outputs of the system (more precisely, metrics) are put as input to the system to react accordingly. As already explained, it is important that this works automatically to reduce human interaction.



Control Circuit

3.3.1 Metrics

Metrics include all information of the software system and each particular failure unit. The metrics should cover the status of the unit to understand if it is working properly. The metrics contain very basic values such as CPU usage, memory consumption, and so on, but can also go beyond external communication.

The metrics are used for error detection. That is, the values are the output of the unit that is used as input to allow the unit to handle the behavior. Sometimes, it even makes sense to introduce a supervisor. A supervisor is a component in a software system that controls other units. Managing from within the system is not a good idea, which means that in the case of a crash of the unit, the unit cannot handle it, because it is simply not there.

3.3.2 Logging

Beside the metrics that are used for automatic recovery of the system, it is also required to log the behavior of the system to allow root cause analysis, identify unexpected behavior, and collect statistical data used later to correlate problems. Each failure unit has to define the logging strategy to allow implementation improvements.

3.3.3 Patterns for Error Detection

Fault-tolerant software development includes many patterns that are meaningful for the error detection. To do the control circuit, you have to identify the errors first. The patterns for error detection are connected to the resource management, incoming requests that lead to overload, and outgoing calls to external services.

Patterns for Error Detection

Pattern	Definition
Heartbeat	The components send the status information.
Watchdog	A supervisor component monitors to make sure that all child components are running correctly.
Circuit breaker	Failure units call external services and resources. If the circuit control identifies a problem with external dependencies, the failure unit should not increase the problem by overloading the external service.

3.3.4 Patterns for Error Recovery

After identifying the errors, the control circuit must trigger the actions automatically.

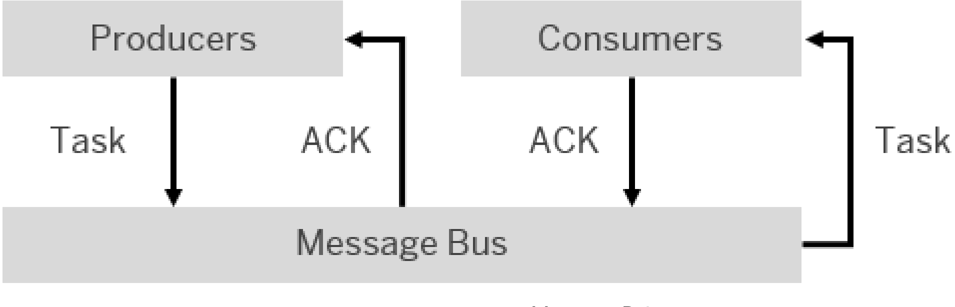
Patterns for Error Recovery

Pattern	Definition
Quarantine	If the control circuit detects that the same data repeatedly leads to the same problems, this data should be put into quarantine to handle it in isolation.
Rollback	If the control circuit detects that the task is running into roblems, an automatic rollback should be possible.
Return to recovery point	In case of an issue, there should be a clearly defined point to which to return to retrigger the execution.
Slow it down	If the control circuit detects that the incoming load is getting to high, the failure unit must slow down to avoid overload situations and exceptions.

3.4 Message Driven

Today's cloud software is based on many interacting services. As the communication between services is inherently unreliable, it is hard to maintain the connection between two services in conversational interaction. Request-response protocols give control on the client but pose difficulties when implementing a client to survive crashes both on itself and on the server side. Another issue is to discover endpoints of all participants in conversational interaction, because the endpoints must be hardcoded and can come and go during the lifetime of a system. Much of the complexity and predictability involved in such interaction can be mitigated by implementing an intermediate messaging layer that facilitates communication between two or more components.

It is very important to decide on the communication paradigm to be used. Asynchronous processing based on the Reactive Manifesto, provides loose coupling of services. Decoupling the services allows to replace a single service without changing the others. The message is the contract between both sides.



Message Driven

3.4.1 Message-Handling Patterns

There are multiple patterns that support the message-driven approach:

- Acknowledgement is used for reliable communication. Producer and consumer exchange a message that triggers a task. The acknowledgement should provide a response if the task is to be handled by the consumer.
- Idempotency means that clients can repeatedly make calls that produce the same result. The producer can send a message as often as needed without creating a negative impact to the consumer.

3.4.2 Resilient Protocols

Resilience at the technical protocol level depends on asynchronous communication and eventual consistency. The consistency is eventual, because there is no guarantee that messages are delivered immediately.

A resilient protocol is tolerant to:

- Message loss
- Message reordering
- Message duplication

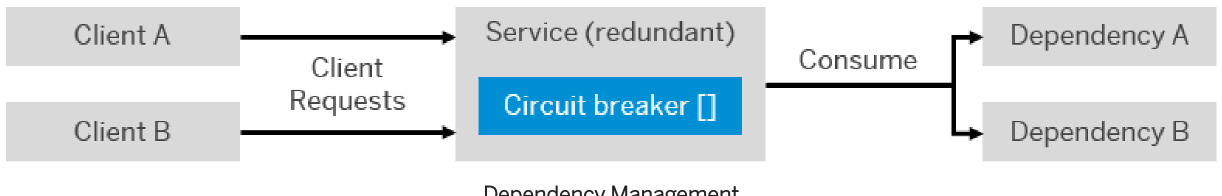
As explained, this means that the sender must ensure the resilience of the message transfer by duplication, that is, by resending messages that are idempotent.

3.5 Dependency Management

Failure units usually depend on other functionality. According to the principle of fault-tolerant software, external functionality is also provided by other failure units. The communication between these failure units is

an important question, but the failure unit itself should be able to handle the dependencies. A lack of dependency management could lead to multiple problems, including the follwing:

- A failure unit has a hard dependency to an external functionality and cannot work without it. If there is no fallback available, this is usually indicative of a poorly designed failure unit.
- A dependent failure unit is unable to handle the load, and requests to the dependent failure unit worsen the problem.



Dependency Management

3.5.1 Isolation of Dependencies

To identify problems at a very early stage, a failure unit should closely control outgoing traffic. Monitoring is used for this purpose, and usually includes measuring metrics such as response time distribution, maximum latency, failure rate, and so on.

The circuit breaker pattern is crucial in this context because every external request should be managed in a way that suspends calls ("open circuit") if there are repeated failures of dependency during a predefined time window. Some frameworks such as Hystrix implement such patterns.

After a predefined time in "open circuit" mode, the failure unit tries again to execute one or more invocations ("half-open circuit"). If these calls are successful, the circuit closes; otherwise, it is kept open for the duration of a new time window.

3.5.2 Graceful Degradation

The unavailability of a dependent failure unit must not result in problems of the unit in use. If the dependent unit cannot provide the required information, the calling unit should be implemented to handle such failures. In the best case, a failure unit does not have any mandatory dependencies, and uses only dependencies that increase its service quality. That is, the more dependencies that are up and running, the more functionality is provided to consumers by the failure unit. This pattern is called graceful degradation. The more external dependencies that are missing, the more the service quality decreases.

Throttling of Invocations Towards Dependencies

The failure unit should handle and manage both incoming and outgoing calls. Usually, a failure unit can handle the limits of the used functionality to avoid crashing any dependent units. If not, it can come to overload situations on the receiver side. The dependent failure unit uses rate limitations to protect itself, but because of isolation, units cannot rely on rate limitation only on the input. The rate limitation for outgoing calls can be implemented by the usual means, such as semaphores, dedicated fixed-size thread pools, and so on. Additionally, the failure unit cannot use all of its resources trying to call an external dependency. This is again a mandatory part of the control circuit inside the failure unit itself. To avoid too many calls to other failure units, a

unit can use a cache that simply holds data that is not required to travel via the network. Unfortunately, this brings with it all the inherent disadvantages of data handling because the data must be valid to ensure that the failure unit works properly. Eventual consistency comes into play in this situation.

3.5.3 Request Collapsing

Many failure units run into problems because the API used by the dependent failure unit provides an interface that is too fine-grained. In the end, the failure unit performs a high number of invocations to an external dependency, which places a huge load on the network.

To save network resources and resources on the dependent failure unit, merge individual requests into a bulk request.

3.5.4 Service Discovery

One important principle of resilient architecture is loose coupling. If a failure unit has a hard-coded dependency to another failure unit, both units are called tightly coupled; they cannot evolve independently. There are different ways to avoid such coupling at the protocol level, but there is yet another challenge for the initial discovery of an external dependency, because there must be another layer that allows the abstraction to decouple. Such a technology is usually called a service registry. All failure units that provide functionality that can be used by other services must register to this registry. The service registry allows the dynamic creation of service instances.

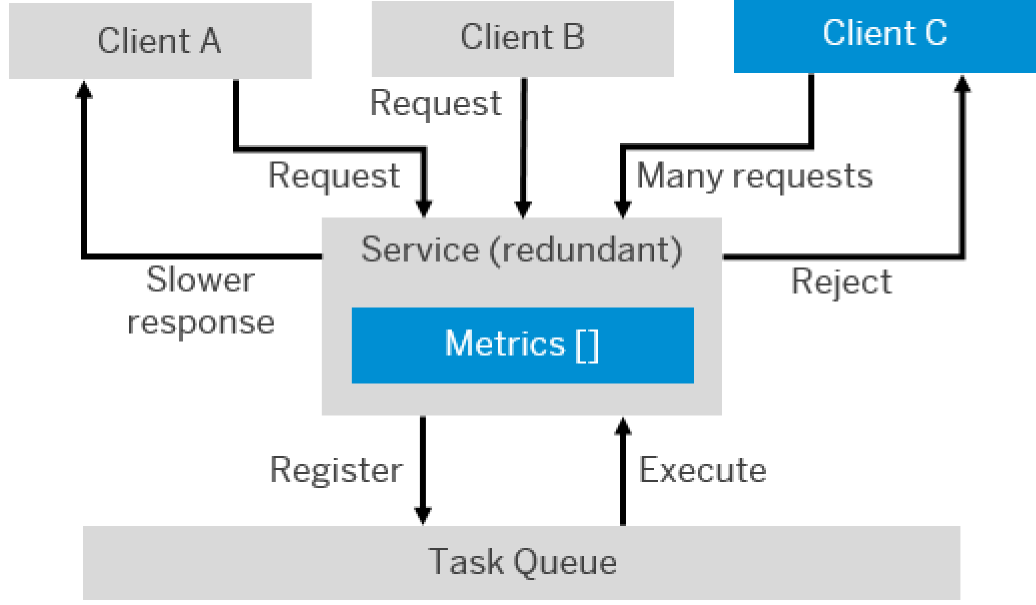
There are frameworks that implement such service discovery frameworks like Netflix Eureka or Hashicorp Consul.

Related Information

Netflix Eureka
Hashicorp Consul

3.6 Rate Limiting

Rate limiting is relevant to a failure unit for all incoming calls. The overall goal in limiting incoming traffic is to run the failure unit in well-defined boundary conditions. If these limitations are exceeded, the failure unit does not work in the defined environment, and the failure probability is very high. Therefore, incoming traffic must be controlled, which is not only a task for an external component such as a load balancer, but also for the failure unit itself.



3.6.1 Client Isolation

Each client that calls a failure unit can access only a dedicated set of resources within that unit, rather than the full capacity, because access to the full capacity would mean that one client could bring down the entire failure unit. Clients work with an isolated failure unit that follows the recursive character of all of them. A single client or subscriber to a failure unit should not be able to consume the entire capacity of a service.

Mechanisms to push back and limit the usage per client identity must be implemented. These mechanisms can be based on threads, processes, actors, and so on. Most importantly, the implementation technology should isolate requests and provide a fair scheduling for all clients.

3.6.2 Request Throttling

Another strategy for controlling the number of requests is to throttle them. The failure unit monitors clients by counting requests or measuring other important types of load.

There are multiple strategies for establishing limitations:

- Client quotas
- Prioritization of APIs and clients
- Defer/suspend operations (which should also inform the client)

Fault-tolerant architecture patterns such as slow-it-down or queue-of-resources support throttling.

3.6.3 Scaling

Scaling is another valid approach for handling limitations on incoming calls. Both vertical and horizontal scaling allow a greater number of requests. Scaling is mandatory for today's cloud services because a single node is often not sufficient to handle the load.

3.6.4 Monitoring

Like all other metrics, incoming requests must be measured to control and scale each failure unit. This is part of the control circuit of the failure unit. Request monitoring should cover scarce or limited resources, including memory, threads, disk space, storage, sockets, and file handlers. All critical resources must be protected.

4 Resilience Patterns

You can find definitions of principles for resilience and high availability in many places. Unfortunately, the principles are usually not described in sufficient detail to allow developers to apply them in their coding.

In software development, the usual solution for this situation is to define patterns that can be used during development. The software developingapproach changed after publishing these patterns because a developer can read the patterns and understand where to apply them. The patterns are defined in a composable way, which means that you can take one pattern and combine it with another. As a consequence, developers do not have to solve the same problems over and over again, but they can simply use well-defined patterns, check to see if a pattern is applicable, and then use it in their development.

Pattern Approach

This approach can also be applied to resilience by defining a set of composable building blocks, which are called patterns.

One of the first challenges is that the patterns differ in terms of granularity. Some patterns are well covered in known libraries and only require one line of code to use them:

Examples for Patterns Covered in Known Libraries

Pattern	Explanation
Timeout	Most libraries provide a parameter in the call method to define a timeframe after which execution is interrupted. Afterwards, information is provided indicating whether the timeout has occurred or the method was completed successfully. It is up to consumers of the library to handle this properly.
Retry	This is usually a wrapper around the call-method to define a retry strategy. This can be done linearly or with exponential backoff, but to call a method again and again is usually not difficult.

These patterns are part of the usual developer toolbox and are called micropatterns because they can be easily combined with others and used everywhere. This shows the composable aspect of resilience patterns, because retry and timeout are definitely mandatory patterns for other patterns such as unit isolation.

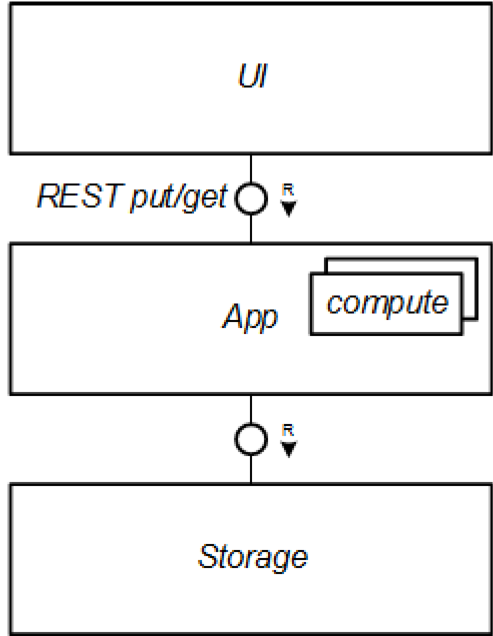
Definition of a Pattern

The general structure for defining a pattern is as follows:

Structure for Defining a Pattern

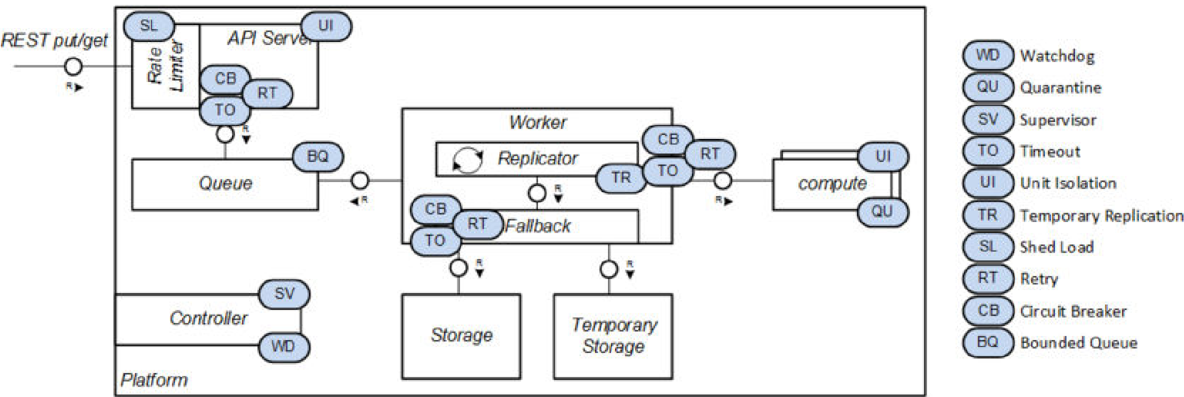
Category	Questions Answered
Action	What must be done to apply the pattern?
Application	When is the pattern
Principles	Which resilience patterns are used?
Patterns Used	Which other patterns are used?

The patterns are explained based on an old-style classic application (database for data storage, one process to handle requests). The application is kept very simple so that the focus is on the resilience pattern currently in scope.



Composition

Composition is a crucial concept for applying patterns. So far, the patterns have been explained as isolated guidelines, but the real power of patterns is that they can be applied to the architecture of a system, if they are applicable. The decision to apply a pattern depends on the use case. The following diagram shows an example of a system architecture that applies all of the patterns.



See the following expalation of why the patterns were applied:

Explanation of the Patterns Used

Pattern		Explanation
Unit isolation		API Server is separated because the acceptance of requests shall be handled independently of the execution of the requests.The application is focused on handling requests and putting these requests in a queue where worker nodes can then pick the requests up to handle them.
Separated compute		Compute is separated because computation requires many resources and can potentially crash. Therefore, it is separated from the worker node in order to handle it in an isolated way.
Temporary replication		A temporary storage is introduced to handle the downtime of the primary storage. The worker node can continue working even if the primary storage is down.
Dependency management		The patterns for dependency management (timeout, retry, circuit breaker) are used to handle the availability of the isolated units. These patterns are usually applied together.
	Timeout	A timeframe after which the execution is interrupted is defined.
	Retry	A retry strategy is defined.
	Circuit breaker	A circuit for each external dependency is introduced to control the behavior of calls.
Shed load		The API server protects itself by defining limits for the number of incoming requests.
Bounded queues		The requests are put in a queue to handle the load of the system by worker nodes picking up the number of requests that they can handle.

Pattern	Explanation
Supervisor	The supervisor is not implemented by the application itself. The platform defines a controller to check the availability of the individual nodes of the system.

Reference Application

Enterprise Sales and Procurement Model (ESPM) Cloud Native is a reference application based on microservices architecture principles, which showcases how resilience patterns can be implemented in cloud native applications. It shows the usage of the following patterns:

- Retry
- Timeout
- Circuit Breaker
- Bounded Queue
- Shed Load

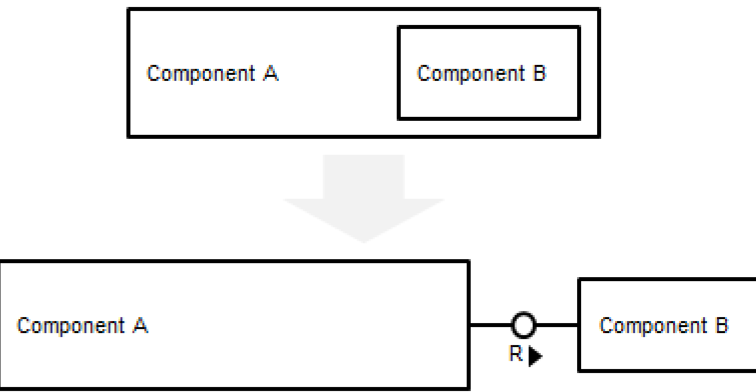
For a detailed example for the usage of resilience patterns in cloud native applications, see Enterprise Sales and Procurement Model (ESPM) Cloud Native .

4.1 Failure Unit: Unit Isolation

The pattern for unit isolation focuses on one of the core decisions of resilience, the design of the failure unit. A failure unit is the entity of an application that can fail without impacting the overall availability of the entire application. During the application design process, it is crucial to define a split of the functionality to avoid the monolithic architecture design. The overall problem for unit isolation is to find a good balance of the isolated entities. Nowadays, different methodologies such as domain-driven design are applied to define the cut of units, but at the moment, it looks more like an art than a science. It depends on many circumstances and boundary conditions. Therefore, it is a clear recommendation to assess multiple options of isolation and not to apply the pattern everywhere.

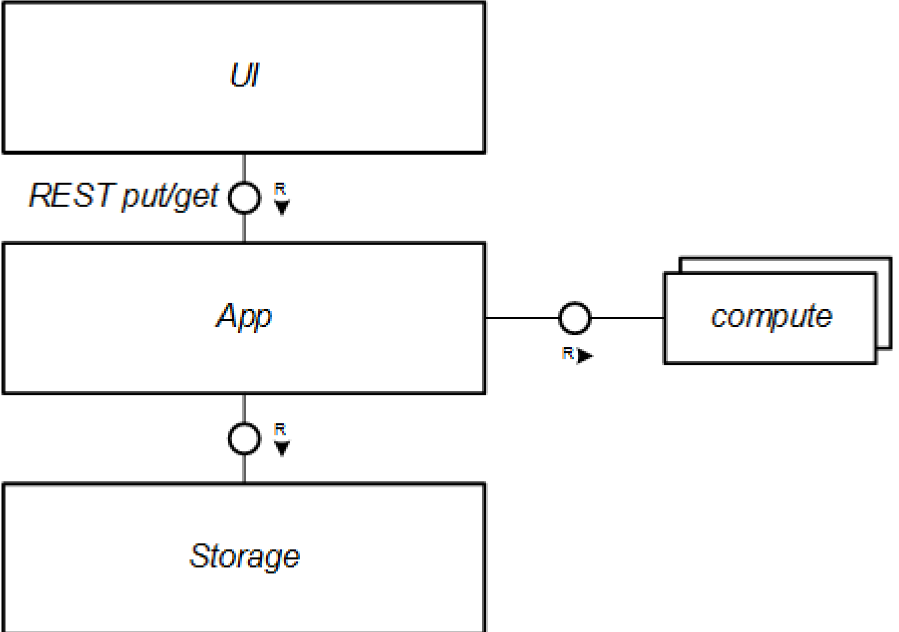
Action

The compute units are split during the failure unit design, because each run of a compute unit can crash and bring down the whole application. If the application is implemented as one monolith, one action can influence all others. As a consequence, the communication to compute unit is remote, and if the synchronous call crashes, the request on the caller side gets an exception. So, the caller can handle the error situation.



It makes a lot of sense that the patterns for retry and timeout are applied here, because the remote call is a good isolation for the compute unit, but it has to be handled in an appropriate way. Only the compute unit is a failure unit after applying the pattern, because nothing has changed for the overall application, which means that if the application is only running with one instance, the application does not have a fallback instance.

After applying the pattern, the application looks as follows:



Application

Separation of some functionality can only be done if:

- The task can be separated in an isolated way. This is usually the case for compute-intensive operations or non-mandatory information.
- The size of the data required for the computation is not so big that it overloads the network.
- An anti-pattern in this context is to simply pass a reference to an external storage that contains the huge data set. This is because passing such references implies coupling on data level. It can be done, however, if you isolate the compute functionality but couple on data level, this does only half of the job.

Resilience Principles

Of the four principles of resilience, the following are applied:

Resilience Principles

Resilience Principle	Definition
Isolation	The compute unit is isolated from the remaining parts of the application. There is a clear boundary between the units.
Decoupling	The isolated compute defines an interface, and this interface is used for decoupling from the remaining parts.
Redundancy	Unit isolation is usually applied to allow failover to a secondary runtime instance.

Patterns Used

The following patterns are used:

Patterns Used

Pattern	Definition
Retry	Each call to a compute unit must be wrapped by a retry logic.
Timeout	Each call must define a timeout. Otherwise, the remaining parts of the application could be blocked forever.

Implementation

The application contains business logic to handle orders:

```
private void process(OrderItem order) {

        Long productId = order.getProductId();

        Long customerId = order.getCustomerId();

        Product product = handler.productRepository.findOne(productId);

        Customer customer = handler.customerRepository.findOne(customerId);

        Double discount = DiscountCalculator.discountCalculator(productId,
customerId);

        Double price = product.getPrice();

        customer.setAmount(customer.getAmount() - price * discount);
```

```
        product.setQuantity(product.getQuantity() - 1);
        handler.customerRepository.save(customer);
        handler.productRepository.save(product);
        SuccessfulOrderItem successfulItem = new SuccessfulOrderItem();
        successfulItem.setProductId(order.getProductId());
        successfulItem.setCustomerId(order.getCustomerId());
        successfulItem.setPrice(price);
        successfulItem.setDiscount(discount);
        handler.successfulOrderRepository.save(successfulItem);
        handler.orderRepository.delete(order);
        }
```

All database operations are running in a transaction. The important statement is the call to the compute-intensive functionality.

The compute unit is now isolated, so that the functionality is separated from another process. The isolation of the compute unit is now done in such a way that the functionality is separated from another process. With modern development frameworks, it is not difficult to extract the logic to a separate process, but to handle the call appropriately.

The remote call is isolated to method calls:

```
private Double getDiscount(Long productId, Long customerId) {
        Double ret = 1.0;
        int retry = RETRY;
        while(retry > 0) {
        try {
        final ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<Double> future = executor.submit(new 
ExecuteRequest(customerId.toString(), productId.toString()));
        ret = future.get(30, TimeUnit.MILLISECONDS);
        retry = 0;
        } catch (TimeoutException e) {
        retry = triggerRetry(retry);
        } catch (InterruptedException e) {
        retry = triggerRetry(retry);
        } catch (ExecutionException e) {
        retry = triggerRetry(retry);
        }
```

```
        }
        return ret;
        }
        private class ExecuteRequest implements Callable<Double> {
        private String customerId;
        private String productId;
        public ExecuteRequest(String customerId, String productId) {
        this.customerId = customerId;
        this.productId = productId;
        }
        public Double call() throws Exception {
        Double ret;
        StringBuilder result = new StringBuilder();
        URL url = new URL("http://localhost:"+port+"/discount/value?
customerId="+customerId+"&productId="+productId);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        BufferedReader rd = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
        String line;
        while ((line = rd.readLine()) != null) {
        result.append(line);
        }
        ret = Double.valueOf(result.toString());
        rd.close();
        return ret;
        }
        }
```

The call is separated from a Java future to allow interruption of the call (see timeout parameter) and the entire call is wrapped into a retry loop to allow calls. It is also important to notice what happens in an exceptional case: The URL is changed because the call should go to another process. This implies that at least two instances of the isolated process for discount calculation are running.

4.2 Data Handling: Temporary Replication

The resilience pattern for a temporary replication solves the issue of inheriting the availability of a used storage technology. The underlying problem is that an application can never improve in terms of availability if the application is coupled to the storage. Each call to the application results in calls to the storage. If the storage is not available, the application cannot work. For this purpose, another storage technology might make sense.

Temporary replication focuses on one of the difficult problems of resilient applications because the data consistency must be handled. In monolithic software architectures, the server used a relational database and followed the ACID paradigm, but if the server load grows beyond a specific limit, the database can no longer handle the load. The load must be scaled out, and the CAP theorem becomes important here.

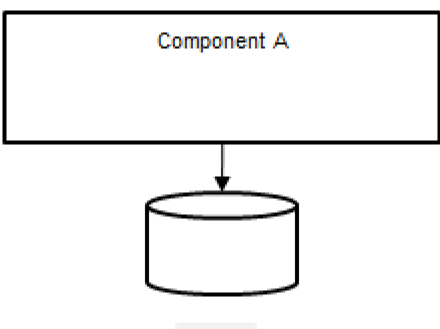
The CAP theorem defines that only two out of three qualities can be achieved:

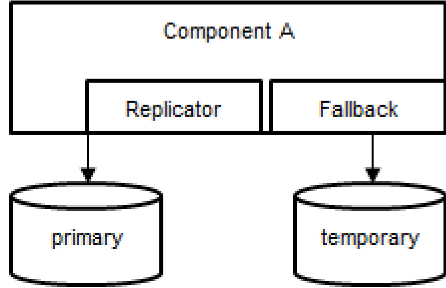
- Consistency
- Availability
- Partition tolerance

In a distributed system as is required to handle the load, there is partitioning. The distributed system must provide availability. This means that the consistency must be compromised. The term "eventual consistency" has gained prominence. It means that the different instances of the data storage can deliver different information, depending on the time when the data is requested. Consistency is achieved again only after a dedicated timeframe. Introducing a temporary storage to mitigate the downtime of the primary one is one way to achieve the availability of the overall service. However, it is problematic to make the data consistent afterwards. As a general recommendation, you should assess what qualities the system must provide under the given boundary conditions. Again, this assessment can only be done on a case-by-case basis.

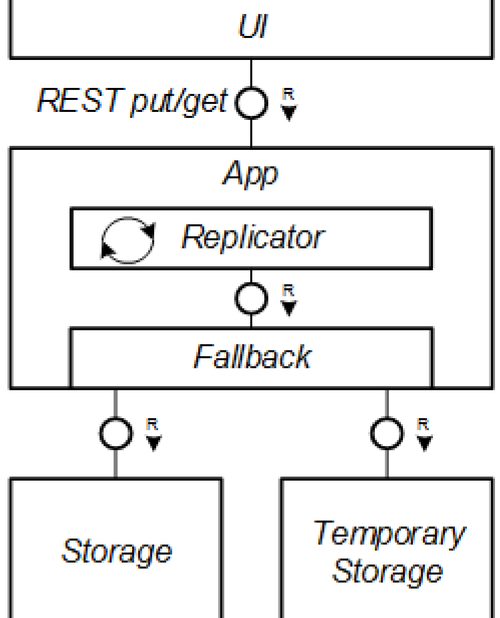
Action

There is a fallback storage to persist information and replicate later. For this purpose, a circuit breaker is used, and the fallback implementation redirects to the temporary storage. Once the primary storage is available again, the data from the temporary storage is transferred to the primary storage. This is done in an asynchronous loop, running periodically in the application.





The application is changed as follows:



Application

Using a secondary storage is an option, if:

- Persisting data is not required for later business logic steps (storage can be deferred).

- Temporary storage does not lose data (potentially it could be allowed to lose data; this is a business decision).

Resilience Principles

Of the four principles of resilience, the following are applied:

Resilience Principles

Resilience Principle	Definition
Fallback	If the primary storage is not available, a secondary storage is used as a fallback.
Redundancy	The secondary storage is a redundant place to store data.

Patterns Used

The following patterns are used:

Patterns Used

Pattern	Definition
Circuit breaker	If the connection to the primary storage is not available, the circuit is opened and closes only when the primary storage is available again.

Implementation

The implementation of the resilience pattern involves two major pieces:

- Fallback to temporary storage
- Replication to primary storage

Fallback

The call to the primary storage has to be wrapped to allow the fallback implementation. This is done via a dedicated instance of the call request.

```
@GetMapping(path="/add")

        public  @ResponseBody  String add(@RequestParam(value="productId") Long
productId,
```

```
        @RequestParam(value="customerId") Long customerId) {
        OrderItem order = new OrderItem();
        order.setCustomerId(customerId);
        order.setProductId(productId);
        SaveRequest request = new SaveRequest(order);
        request.execute();
        if ( request.isResponseFromFallback() )
        return "";
        return order.getId().toString();
        }
        private class SaveRequest extends HystrixCommand<String> {
        private OrderItem order;
        public SaveRequest(OrderItem order) {

super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("RequestGroup")).a
ndCommandPropertiesDefaults(

HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(1000)));
        this.order = order;
        }
        @Override
        protected String run() throws Exception {
        orderRepository.save(order);
        synchronized(list) {
        if ( list.contains(order))
        list.remove(order);
        }
        return order.getId().toString();
        }
        @Override
        protected String getFallback() {
        synchronized(list) {
        if ( !list.contains(order))
        list.add(order);
        }
```

```
        return "";
        }
        }
```

The Hystrix framework provides a good foundation to allow a fallback implementation and timeout behavior. The fallback implementation simply adds the order request to an in-memory list for later replication (this is not a reliable storage overall because if the process crashes, all data is lost).

Replication

The data in the secondary storage must also be replicated to the primary storage. A separate thread is used for this.

```
private List<OrderItem> list = new ArrayList<OrderItem>();
        public class OrderReplicator implements Runnable {
        @Override
        public void run() {
        while(true) {
        if ( list.size() > 0 ) {
        ArrayBlockingQueue<OrderItem> queue;
        synchronized(list) {
        queue = new ArrayBlockingQueue<OrderItem>(list.size(), true, list);
        }
        queue.forEach(order -> process(order));
        }
        try {
        Thread.sleep(500);
        } catch (InterruptedException e) {
        e.printStackTrace();
        }
        }
        }
        private void process(OrderItem order) {
        SaveRequest request = new SaveRequest(order);
        request.queue();
        }
```

```
        }
```

4.3 Control Circuit

Quarantine [page 37]
    Use the quarantine pattern to isolate the execution of a request in a more controlled environment.

Supervisor [page 40]
    Use the supervisor pattern to control the execution of failure units.

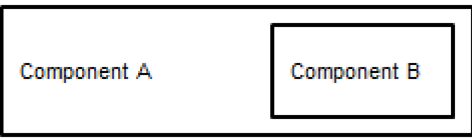
Watchdog [page 43]
    Use a watchdog component to monitor the behavior of another component.

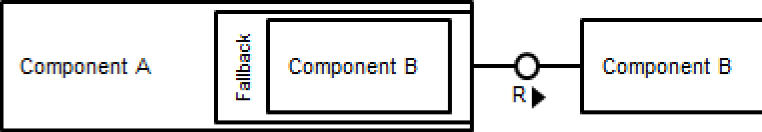
4.3.1 Quarantine

Use the quarantine pattern to isolate the execution of a request in a more controlled environment.

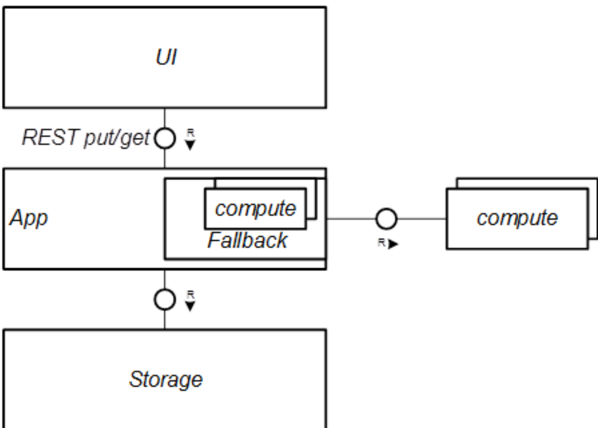
Action

The compute unit that can create problems should be isolated. Usual execution is done inside the usual boundary conditions. However, if a dedicated instance is identified, the instance for computation is delegated to a fallback implementation with dedicated resources.





The application looks like this after applying the pattern:



Application

Quarantine can only be done if:

- The unit of quarantine can be isolated.
- Usual execution does not cause the whole application to crash. If the unit to be quarantined causes a crash of the entire runtime, other strategies (supervisor) must be applied to handle the fallback.

Resilience Principles

Of the four principles of resilience, the following are applied:

Resilience Principles

Resilience Principle	Definition
Isolation	The compute unit is running with its own resources, nothing is shared.
Fallback	The implementation of the compute unit must be available in a fallback implementation to push the execution to it.
Loose coupling	The compute unit provides a clear interface to allow easy execution.

Patterns Used

The following patterns are used:

Patterns Used

Pattern	Definition
Unit isolation	The fallback implementation uses a separated unit to compute.

Implementation

Only the implementation of the command to trigger the computation is changed. The command uses the in-process execution to get the result. Only in case of an exception, the fallback to call an external service is used.

```
@Override
        protected Double run() throws Exception {
        return DiscountCalculator.discountCalculator(productId, customerId);
        }
        @Override
        protected Double getFallback() {
        logger.info("remote call for "+productId);
        StringBuilder result = new StringBuilder();
        try {
        URL url = new URL("http://localhost:"+port+"/discount/value?
customerId="+customerId+"&productId="+productId);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        rd = new BufferedReader(new InputStreamReader(conn.getInputStream()));
        String line;
        while ((line = rd.readLine()) != null) {
        result.append(line);
        }
        ret = Double.valueOf(result.toString());
        } catch (NumberFormatException e) {
        } catch (MalformedURLException e) {
        } catch (ProtocolException e) {
        } catch (IOException e) {
        } finally {
        if ( rd != null )
        try {
```

```
        rd.close();
        } catch (IOException e) {
        }
        }
        return ret;
        }
```

The call is separated from a Java future to allow the interruption of the call (see timeout parameter) and the entire call is wrapped into a retry loop to allow calls. It is also important to notice what happens in an exceptional case: The URL is changed because the call should go to another process. This implies that at least two instances of the isolated process for discount calculation are running.

Parent topic: Control Circuit [page 37]

Related Information

Supervisor [page 40]
Watchdog [page 43]

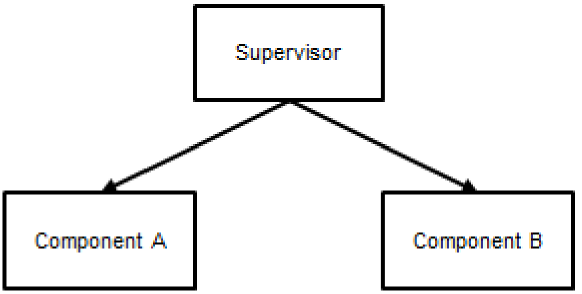
4.3.2 Supervisor

Use the supervisor pattern to control the execution of failure units.

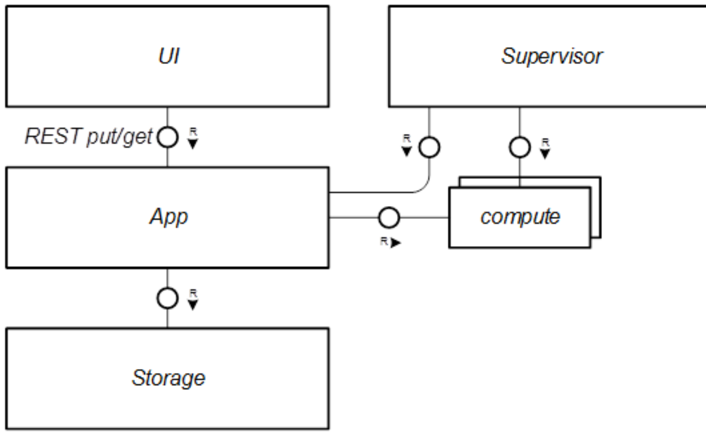
Sometimes, the unit of work might not be able recover on its own because the unit simply repeats the task over and over again. This could potentially lead to endless cycles. An external entity can fulfill the task of controlling the behavior and execution of the failure unit. If something unexpected happens, the supervisor becomes active and recovers the runtime by restarting, rescheduling the work, or any other appropriate action. Supervisor is a concept that was developed some time ago. The programming language Erlang, created in 1987, made it part of the core key words of the language. The idea is to define a language that is specifically intended to provide an environment for high-available systems. Other languages, such as Scala, have taken over the concepts, called actors, to provide exactly the same advantages.

Action

The application and all failure units are not impacted by the implementation of a supervisor. A supervisor is just an external component that controls the instances. The actions that are taken can include a complete restart or fine-tuning of the instances.



The application looks like this after applying the pattern:



Application

Supervisor is useful if:

- The control of the failure units cannot be done within the unit itself. Sometimes, the unit might work in a way that correct calculation of resource consumption is not possible. A supervisor is a good approach to interrupt such a task because the supervisor is not impacted by any work inside the failure unit.
- The failure unit can be separated properly (clear interface).
- External control is feasible because all mandatory metrics are available.

Resilience Principles

Of the four principles of resilience, the following are applied:

Resilience Principles

Resilience Principle	Definition
Isolation	The supervisor is an isolated component to control the other failure units.

Patterns Used

The following patterns are used:

Patterns Used

Pattern	Definition
Unit isolation	The failure unit implementation is isolated to allow partial
Watch dog	A watch dog retrieves information about the failure unit under supervision.

Implementation

The implementation of the supervisor can be rather simple:

```
public static void main(String[] args) {

        start();

        while(true) {

        while(!queue.isEmpty()) {

        logger.info(queue.poll());

        }

        watchdog(8081);

        watchdog(8082);

        }

        }
    The failure unit implementation is isolated to allow partial
                                            recovery.
```

First of all, the supervisor starts the failure units that are controlled. The start of the observed process is done as a command: `java -Dserver.port= -jar`

The Java functionality for handling external processes is used and handled in a special implementation class:

```
public class NodeObserver implements Runnable {

        public NodeObserver(int port, String executable,
ArrayBlockingQueue<String> queue) {

        ...

        }

        @Override

        public void run() {

        Process p;
```

```
        while(true) {
        try {
        p = Runtime.getRuntime().exec(new String[] {"java", "-
Dserver.port="+port, "-jar", executable});
        try {
        p.waitFor();
        } catch (InterruptedException e) {
        e.printStackTrace();
        }
        queue.put(port+" crashed -> restarting");
        } catch (IOException | InterruptedException e) {
        e.printStackTrace();
        }
        }
        }
```

The process is started and running. A restart is triggered only if the process crashes and a message is placed in a queue.

Parent topic: Control Circuit [page 37]

Related Information

Quarantine [page 37]
Watchdog [page 43]

4.3.3 Watchdog

Use a watchdog component to monitor the behavior of another component.

The watchdog resilience pattern focuses on the behavior of a second component. If some predefined metrics receive values that are beyond a specific threshold, the watchdog component can trigger appropriate actions. The pattern is connected to the supervisor pattern, but the supervisor specifies that the components under supervision are controlled by the supervisor from a lifecycle point of view. The supervisor can start or stop the supervised components. The watchdog only receives data from the other components.

Action

Metrics for a component are defined and externally accessible. Another component, the watchdog, can get the values.

Application

Watchdog can only be done if:

- Values for metrics are externalized.
- Specific thresholds and actions are defined that must be taken once the thresholds are reached.

Resilience Principles

Of the four principles of resilience, the following are applied:

Resilience Principles

Resilience Principles	Definition
Isolation	The watchdog component controls the isolated component.

Implementation

The supervisor component contains also some watchdog functionality:

```
public static void main(String[] args) {
        start();
        while(true) {
        while(!queue.isEmpty()) {
        logger.info(queue.poll());
        }
        watchdog(8081);
        watchdog(8082);
        try {
        Thread.sleep(500);
        } catch (InterruptedException e) {
        e.printStackTrace();
```

```
        }
        }
        }
        private static void watchdog(long port) {
        try {
        StringBuilder result = new StringBuilder();
        URL url = new URL("http://localhost:"+port+"/discount/requests");
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        BufferedReader rd = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
        String line;
        while ((line = rd.readLine()) != null) {
        result.append(line);
        }
        logger.info("Requests for "+port+": " + result.toString());
        rd.close();
        } catch (MalformedURLException e1) {
        } catch (ProtocolException e1) {
        } catch (IOException e1) {
        }
        }
```

The watchdog does a repeated call to see how many requests were done at the remote component. The functionality to return the number of requests is done via a REST endpoint:

```
@GetMapping(path = "/requests")
        public @ResponseBody Long requests() {
        return new Long(requests);
        }
```

Parent topic: Control Circuit [page 37]

Related Information

Quarantine [page 37]

Supervisor [page 40]

4.4 Dependency Management

Circuit Breaker [page 46]
Use the circuit breaker pattern to control future requests depending on the external request, the response time, and the answer.

Retry [page 48]
Use the retry pattern to retrigger the execution of a task.

Timeout [page 50]
Use the timeout pattern to define when a call is interrupted.

4.4.1 Circuit Breaker

Use the circuit breaker pattern to control future requests depending on the external request, the response time, and the answer.

The resilience pattern for circuit breaker is sometimes seen as the most important pattern. This is because the pattern tries to mitigate one of the shortcomings of distributed systems. Software systems are growing and usually do not handle a request without contacting another system. The problem with contacting an external system is that it is rather difficult to get information about the status of the external system. The external system might be overloaded and not responding or the response time might be fluctuating. The circuit breaker tackles these problems by introducing a kind of circuit for each external dependency. The external request is tracked, as well as the response time and the answer. If a problem is identified, the circuit on the caller side controls the behavior of the calls in future. For example: If the response time increases, the external system might be under heavy load, wich can be increased by each call so that the external system might crash. To avoid this crash, the circuit can be configured so that the calls are not done if the response time goes beyond a specific limit. The circuit opens. As the description already indicates, the configuration of all parameters for a circuit can become rather difficult. There is no ultimate set of configuration values that always works. It depends on the external system, the call itself, and even the network latency. So, the best approach is to introduce a circuit breaker, measure the usual behavior, and get a good understanding so that you can fine-tune the configuration settings.

Action

The external call to the remote system must be wrapped to allow the measurement of the external call including response time, error code, etc.

Application

Circuit breakers are applicable if:

- The call goes to a dedicated endpoint.

Resilience Principles

Of the four principles of resilience, the following are applied:

Resilience Principles

Resilience Principle	Definition
Isolation	The call can be isolated.

Implementation

The implementation of a circuit breaker is best explained with the example of a Hystrix command. Hystrix is an open source framework, especially designed for such remote call dependencies.

```
private class ExecuteRequest extends HystrixCommand<Double> {

        private String customerId;

        private String productId;

        public ExecuteRequest(String customerId, String productId) {

super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("RequestGroup")).a
ndCommandPropertiesDefaults(

HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(30)));

        this.customerId = customerId;

        this.productId = productId;

        }

        @Override

        protected Double run() throws Exception {

        ...

        }

        @Override

        protected Double getFallback() {

        ...
```

The request to the external system is isolated to a Hystrix command providing two methods. The method `run` handles the usual call, and the method `getFallback` provides the response, if the circuit is open, which means that the external system has a problem. To define the criteria to open the circuit, multiple configurations can be done in the constructor of the command. In the example, only the timeout is set.

Reference Application

For a detailed example for the usage of the circuit breaker pattern in cloud native applications, see Circuit Breaker .

Parent topic: Dependency Management [page 46]

Related Information

Retry [page 48]
Timeout [page 50]

4.4.2 Retry

Use the retry pattern to retrigger the execution of a task.

The resilience pattern for retry is not very complex, but highly recommended for allowing the successful execution of a task. The underlying assumption is that the execution of a task can succeed if another attempt is made. For attempts within the same process, the assumption might not be valid, but for remote calls, the probability is given that the call will not succeed because of network problems or problems in the external systems. There are multiple ways to handle a retry. The simplest way would be to just trigger the execution again after a fixed period of time. This can be varied so that exponential backoff is applied. That is, the time after the execution is retriggered is increased exponentially. Idempotency is a concept in this context, meaning that a call can be done multiple times without changing the status over and over again.

Action

The execution of a task is surrounded by a retry logic.

Application

Retry can only be done if:

- The first attempt to trigger a task does not have any side effects. If there are side effects, the second and all other subsequent execution attempts change the status of the system. The behavior could be unpredictable.

Resilience Principles

Of the four principles of resilience the following are applied:

Resilience Principles

Resilience Principle	Definition
Redundancy	The execution is triggered multiple times.

Implementation

In the example, the call to an external system is wrapped with a loop that triggers the retry of the call.

```
private Double getDiscount(Long productId, Long customerId) {

        Double ret = 1.0;

        int retry = RETRY;

        while(retry > 0) {

        try {

        final ExecutorService executor = Executors.newSingleThreadExecutor();

        Future<Double> future = executor.submit(new 
ExecuteRequest(customerId.toString(), productId.toString()));

        ret = future.get(30, TimeUnit.MILLISECONDS);

        retry = 0;

        } catch (TimeoutException e) {

        retry = triggerRetry(retry);

        } catch (InterruptedException e) {

        retry = triggerRetry(retry);

        } catch (ExecutionException e) {

        retry = triggerRetry(retry);

        }
```

```
        }
        return ret;
        }
        private int triggerRetry(int retry) {
        retry--;
        changePort();
        logger.info("Retry with "+(6-retry)+". attempt and port "+port);
        return retry;
        }
```

The retry is done after a fixed limit of retries.

Reference Application

For a detailed example for the usage of the retry pattern in cloud native applications, see Retry .

Parent topic: Dependency Management [page 46]

Related Information

Circuit Breaker [page 46]
Timeout [page 50]

4.4.3 Timeout

Use the timeout pattern to define when a call is interrupted.

The pattern for timeout solves the problem of blocking calls for external services. Usually, it is not possible to predict how long a call will take. It might take a very long time or even not return at all. If you want to provide a useful service for end users, this behavior is usually not acceptable. The timeout is mostly a setting that defines when the call is interrupted to handle the situation. The timeout is connected with other patterns such as retry or circuit breaker because all of these patterns define how calls to external systems are handled.

Action

The call to an external system is instrumented with a time frame. If the call time is longer, the call operation is interrupted.

Applicabtion

Timeout can only be used if:

- The execution time of a request is predictable. If the response time is completely erratic, the calling side cannot react appropriately (no definition of a time limit possible).

Resilience Principles

Of the four principles of resilience, the following are applied:

Resilience Principles

Resilience Principle	Definition
Loose coupling	The caller is decoupled from the execution in the remote system. If something happens on the remote side, the caller is no longer blocked.

Implementation

The execution of the call is wrapped by a thread that can be interrupted after a dedicated period of time.

```
 final ExecutorService executor = Executors.newSingleThreadExecutor();

        Future<Double> future = executor.submit(new
ExecuteRequest(customerId.toString(), productId.toString()));

        ret = future.get(30, TimeUnit.MILLISECONDS);
```

Reference Application

For a detailed example for the usage of the timeout pattern in cloud native applications, see Timeout .

Parent topic: Dependency Management [page 46]

Related Information

[Circuit Breaker [page 46]]
[Retry [page 48]]

4.5 Rate Limiting

Shed Load [page 52]
Use the shed load pattern to handle incoming requests to slow down the execution.
Bounded Queue [page 55]
Use the bounded queue pattern to handle the requests in a queue.

4.5.1 Shed Load

Use the shed load pattern to handle incoming requests to slow down the execution.

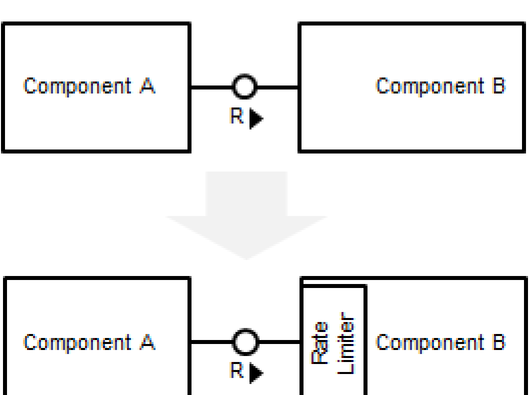
Control of the execution is required if too many requests are started. In this case, the process may consume too many resources, which causes it to crash. If the process crashes, this implies that all other requests can also no longer be served.

The rate limitation directs the problem of how many resources the system will consume to fulfill the tasks. In the past, systems were designed in such a way that they could handle the maximum load, that is, with enough capacity to handle even the peaks. In cloud engineering, the paradigm has changed to tailor the system so that it can scale automatically. To do this, the load situation must be known, which means that you need an upfront calculation of the created load and consumed resources. The pattern for shed load does nothing more than reject or slow down requests if the calculation indicates that the incoming load will overload the system.

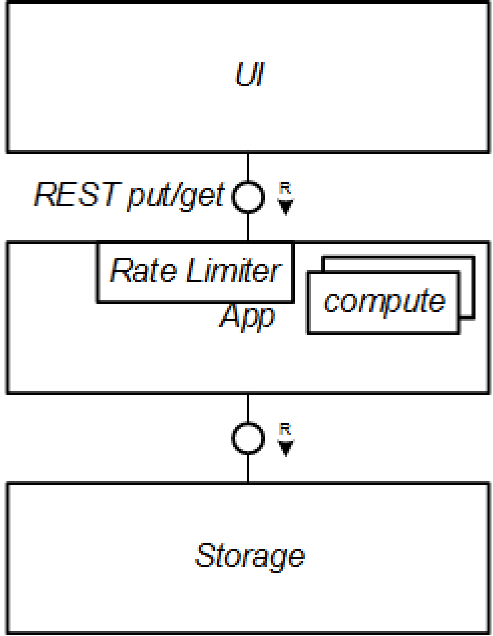
Defining a fixed rate limit is just the first step towards a system that can automatically tailor the load it can handle. In cloud engineering, this is called elasticity. Whether the system must provide elasticity or can survive with a fixed rate limit depends on the business scenario and the effort that must be put into the system. Usually, less automation means that more manual interaction and operations are required.

Action

The incoming requests are counted or analyzed as soon as possible to estimate the expected load of the runtime. Based on the available resources at runtime, a threshold is defined, and requests are rejected or postponed for later execution.



The application looks like this after applying the pattern:



Application

Shed load can only be done if:

- The execution time of a request can be calculated. If the execution time is not easy to calculate, it can become difficult to define a good threshold.

Resilience Principles

Of the four principles of resilience, the following are applied:

Resilience Principles

Resilience Principle	Definition
Isolation	The runtime is shielded from too many incoming requests.

Implementation

The application itself is not changed. The injection of the rate limiter must be done as early as possible. For HTTP requests, it is possible to define a servlet filter:

```
@Configuration
        public class FilterConfiguration {
        @Bean
        public FilterRegistrationBean DoSFilterBean() {
        final FilterRegistrationBean filterRegBean = new
FilterRegistrationBean();
        DoSFilter filter = new DoSFilter();
        filterRegBean.setFilter(filter);
        filterRegBean.addUrlPatterns("/");
        filterRegBean.setEnabled(true);
        filterRegBean.setAsyncSupported(Boolean.TRUE);
        filterRegBean.addInitParameter("maxRequestsPerSec", "10");
        return filterRegBean;
        }
        }
```

The filter implementation uses the way to inject the instance of the filter via a Spring configuration.

Reference Application

For a detailed example for the usage of the shed load pattern in cloud native applications, see Shed Load .

Parent topic: Rate Limiting [page 52]

Related Information

Bounded Queue [page 55]

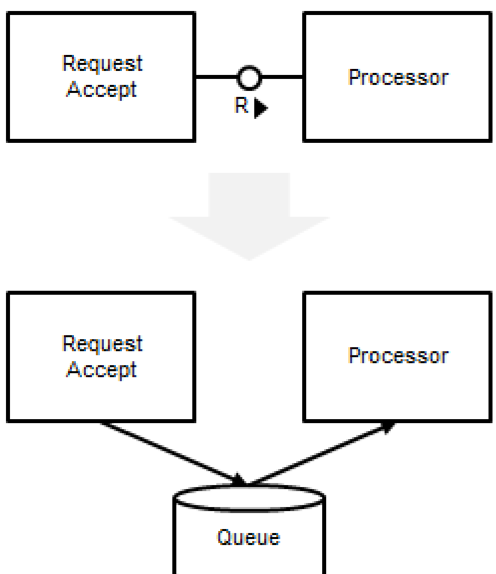
4.5.2 Bounded Queue

Use the bounded queue pattern to handle the requests in a queue.

The resilience pattern for bounded queue is based on the assumption that computing resources are not endless. It is not possible to consume infinite memory or CPU capacity to handle the requests. If the requests are created, the system must calculate how many requests can be served at once. This defines the length of the queue. The Introduction of a queue brings the implementation closer to an asynchronous processing paradigm because the queue holds the items. It is triggered only if the execution can be done.

Action

A queue is introduced to handle the requests. The behavior is changed from synchronous request processing to an asynchronous one.



Application

Bounded queue can only be used if:

- The data for a dedicated task can be separated in a task object.

Resilience Principles

Of the four principles of resilience, the following are applied:

Resilience Principles

Resilience Principle	Definition
Loose coupling	The request acceptance and the execution are decoupled.

Implementation

The processing of a request is directly done in the controller:

```
@GetMapping(path="/add")

        public @ResponseBody String add(@RequestParam(value="productId") Long 
productId,

        @RequestParam(value="customerId") Long customerId) {

        Product product = productRepository.findOne(productId);

        Customer customer = customerRepository.findOne(customerId);

        Double discount = DiscountCalculator.discountCalculator(productId, 
customerId);

        Double price = product.getPrice();

        customer.setAmount(customer.getAmount() - price * discount);

        product.setQuantity(product.getQuantity() - 1);

        customerRepository.save(customer);

        productRepository.save(product);

        SuccessfulOrderItem successfulItem = new SuccessfulOrderItem();

        successfulItem.setProductId(productId);

        successfulItem.setCustomerId(customerId);

        successfulItem.setPrice(price);

        successfulItem.setDiscount(discount);

        orderRepository.save(successfulItem);

        return successfulItem.getId().toString();

        }
```

Introducing a queue leads to two changes in the coding:

- Putting the processing request in a queue

- Taking tasks from the queue and process them

The introduction of a queue is done here:

```
@GetMapping(path="/add")
        public @ResponseBody String add(@RequestParam(value="productId") Long 
productId,
        @RequestParam(value="customerId") Long customerId) {
        OrderItem order = new OrderItem();
        order.setCustomerId(customerId);
        order.setProductId(productId);
        orderRepository.save(order);
        return order.getId().toString();
        }
```

The table for orders simply contains the orders for later execution. It is removed only if it is successful.

```
@Override
        public void run() {
        while(true) {
        if ( handler.orderRepository != null ) {
        Iterable<OrderItem> orders = handler.orderRepository.findAll();
        orders.forEach(order -> process(order));
        }
        }
        }
```

The processing itself is similar, but there is a loop that takes elements from the queue to control the execution.

> **i Note**
>
> A database table usually is not the first choice to implement a queue. There are multiple message brokers available that are suited for such use cases.

Reference Application

For a detailed example for the usage of the bounded queue pattern in cloud native applications, see Bounded Queue .

Parent topic: Rate Limiting [page 52]

Related Information

Shed Load [page 52]

5 How to Make Your Apps Resilient

5.1 Resilience Planning

Start to make your team familiar with the topic of resilience, resilience principles, and patterns.

Downtime Risk Analysis

We propose the following process to plan the resilience in your apps:

- Downtime Risk Analysis Workshop: Have a look at your application or service and identify the factors that could make it crash.
- What are the customer requirements?
- Break it down to the architecture per business process. Which services are involved and on which services do they rely?
- Per service: What would be the impact if if the service is not available or if services you rely on are not available?
- Sort risks according to impact and probability.
- For the top risks: Which resilience patterns or tools can you use to handle failure situations? Define your next steps.

Resilience Planning

Plan your architecture and backlog and assess your resilience readiness. Plan your architecture and backlog items to address the resilience topics identified. You can structure your plan by using the chapter Applying Resilience Principles [page 10].

This might include the following:

- What are your failure units?
- Which patterns do you want to apply to a service?
- Which frameworks do you want to use?
- Define metrics
- Application metrics which generally measure the state and performance of your application, for example successful and failed logins, errors, crashes, and so on
- Business metrics which generally measure the value of your application, for example how many sales were made in one day

- Check your resilience readiness by using resilience maturity monitoring

5.2 Resilience Maturity Monitoring

Design patterns can improve the architecture regarding resilience, but it is difficult to identify how resilient a system is. The resilience maturity model provides levels of quality for various areas of architecture patterns.

Failure Unit

Failure Unit Resilience

Quality Level	Architecture Pattern	Definition
1	Cyclic monoliths	Several strongly coupled monoliths with cyclic dependencies
2	Monolith	Only one component
3	Partly isolated	The principle of separation-of-concerns is applied to the service, but some parts of the service are shared. shared resources lead to tight coupling. For example: Multiple failure units that use one database/schema to synchronize the data.
4	Synchronously coupled	The different units of a service are decoupled based on different domains, but one failure unit of a service cannot work without the others.
5	Decoupled	All failure units of a service are decoupled, and if one failure unit fails, the dependent failure units can continue wokring.

Data Handling

Data Handling Resilience

Quality Level	Architecture Pattern	Definition
1	Data loss during failure	Information can be irrecoverably lost or inconsistent if the service fails
2	Single store	All data is persisted in one data store, no partitioning
3	Scaling	The persisted data is replicated to enable high availability. Data is also distributed or partitioned to enable scale-out.
4	Categorization	The data of the service is persisted in multiple storage technologies that best fit the requirements. However, coupled transactions by involving multiple technologies for one task are not allowed , because this kind of design violates the isolation principle.
5	Optimization	Service data is handled such that overload systems (caches) cannot occur and the data model follows the failure unit design.

Control Circuit

Control Circuit Resilience

Quality Level	Architecture Pattern	Definition
1	No data	No health data is collected, and there is no way to identify, even manually, what is broken.
2	Everything is manual	System operations are all triggered by human beings based on alerts.
3	Basic recovery	Low-level metrics are analyzed and used to trigger actions automatically. Recovery stays on the level of OS and VMs (CPU, JVM crashes, and so on).

Quality Level	Architecture Pattern	Definition
4	Service specific	The service defines some metrics and automatic recovery is triggered.
5	Self-healing	Self-healing: All metrics trigger automatic actions to heal the system.

Rate Limitation

Rate Limitation Resilience

Quality Level	Architecture Pattern	Definition
1	Singleton	Service with no limits and a single active node that cannot be scaled; there is no input validation and the service can be abused by a single misbehaving client.
2	No limits	The service does not check incoming traffic and does not control the load.
3	Maximum limit	The service defines an overall limit, and takes action (alerts or automatic actions) only when the load comes close to the limit.
4	Client limits	The service defines limits for a client to avoid blocking by one single "bad" client.
5	Throttling	The runtime handles the load dynamically by using different patterns such as throttling, slow-down, and queues.

Dependencies

Dependencies Resilience

Quality Level	Architecture Pattern	Definition
1	Dependency crash	Failure units break due to single-dependency misbehavior.

Quality Level	Architecture Pattern	Definition
2	Coupled	The failure units depend on external services.
3	Abstraction	The failure units use an abstraction for the external services to allow non-blocking behavior at runtime, but still need the functionality for proper execution (no degradation).
4	Deradation	The failure units continue working, even if an external dependency is not available.
5	Fallback	The failure unit continues to work, even without an external service, because fallback is implemented. There is no degradation of service.

5.3 Resilience Testing

Atomic Resilience Testing

Resilience testing on atomic (service) level ensures that the applied resilience patterns work as expected under the defined conditions. You should not implement a resilience pattern unless you have an idea of how to test it. How a resilience test can be executed depends heavily on the resilience pattern being tested. According to the test pyramid, the cheapest test is the automated unit test, so we should always strive towards implementing those tests. So, for instance, in the SAP Cloud Curriculum , you can find how to implement unit tests for the circuit-breaker pattern using Hystrix.

Application Performance Management and Availability Monitoring

The monitoring of cloud applications and their resources is essential both for cloud operations and for verifying system behavior. This continual monitoring supports resilience in the sense that the mean time-to-failure is reduced by detecting issues early and understanding the system state.

Application Performance Management (APM) is the discipline related to monitoring and management of application performance. It is difficult to make software that works well, fast, and reliably. This requires specialized, powerful tools such as Dynatrace, which offers a cohesive set of well-established, battle-tested technologies that enable you to do the following:

- End-to-end monitoring of application performance: from the end user's browser to the deepest back-end systems

- Automatic discovery of all components and systems involved in the processing of requests
- Infrastructure monitoring, that is, operative system, network, and virtualization
- Deep tracing of request processing, down to the single line of code, across all components
- Proactive and predictive anomaly detection
- Dash-boarding and reporting

Additionally, APM solutions can automatically detect outlying requests (that is, compared with usual baseline patterns) and proactively collect relevant information of the actual system status for later analysis and debugging by the dev/ops personnel (or dev and ops as separate entities).

Availability Monitoring allows you to monitor, analyze, alert, and report on the availability of your services. This method is of tremendous importance for operations as you are notified when a service or the whole system goes down. For the SAP BTP, this is provided by the Availability Service. With the Availability Service, you can monitor the health endpoints of your services. Health endpoints are simple HTTPS `GET` endpoints returning `200 OK`. In the body, they provide some JSON indicating the health status of the service. Health endpoints might be unsecured or secured by basic authentication, but no stronger than basic authentication. This is because many monitoring systems, including the Availability Service, can only deal with basic authentication. Leaving the health endpoints unprotected facilitates monitoring and the monitors in the Availability Service do not have to be changed each time the password changes. On the other hand, all information provided by the health endpoints are open to everybody and thus should not contain sensitive data.

Holistic Resilience Testing

Testing on unit level is not the end in a microservice world. Putting many microservices together results in a distributed system. There is a level above the single services: the system. The system can have its own behavior. To be able to test system behavior and understand its state, we need to have system monitoring in place. Holistic resilience testing aims at ensuring that applications are resilient under somehow realistic and close-to-production conditions. These tests apply to a scenario, an application, or a system level. For example, you might want to see how an application behaves if a needed service is not available.

Examples

End-to-End Resilience Tests for SAP BTP

In the SAP BTP, three different holistic scenarios are tested. These scenarios require significant effort as they need an isolated landscape that is similar to the productive one, and an artificial load into the landscape must be generated. This requires a lot of preparation. What is tested is the recovery time and the recovery point, that is, how much data is lost. The three scenarios are as follows:

Resilience Testing Scenarios for SAP BTP

Scenario	Definition
Network disruption	While the test system is under considerable load, the network is disrupted and then restored after a while. The measurement is against the time the system and the apps need to recover as well.
Upgrade behavior	Considerable load of business activities on the system while critical functions in the system are being updated or upgraded
Service dispruption	Individual service disruption while the system is under considerable functional load

These three scenarios are currently ready for SAP BTP Neo environment. The test can be triggered as necessary, for example, once per sprint.

Simian Army

The most prominent example for testing resilience on system level is Netflix's Chaos Monkey .

Chaos Monkey is a service that identifies groups of systems and randomly terminates one of the systems in a group. The service operates at a controlled time (does not run on weekends and holidays) and interval (only operates during business hours). In most cases, we have designed our applications to continue working when a peer goes offline, but in those special cases, we want to make sure there are people around to resolve and learn from any problems. With this in mind, Chaos Monkey only runs during business hours under the assumption that engineers will be alert and able to respond.

Important Disclaimers and Legal Information

Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.
About the icons:

- Links with the icon : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
  - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
  - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon : You are leaving the documentation for that particular SAP product or service and are entering a SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

Videos Hosted on External Platforms

Some videos may point to third-party video hosting platforms. SAP cannot guarantee the future availability of videos stored on these platforms. Furthermore, any advertisements or other content hosted on these platforms (for example, suggested videos or by navigating to other videos hosted on the same site), are not within the control or responsibility of SAP.

Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.
The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

Bias-Free Language

SAP supports a culture of diversity and inclusion. Whenever possible, we use unbiased language in our documentation to refer to people of all cultures, ethnicities, genders, and abilities.

www.sap.com/contactsap

© 2021 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company. The information contained herein may be changed without prior notice.

Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors. National product specifications may vary.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies.

Please see https://www.sap.com/about/legal/trademark.html for additional trademark information and notices.