Technische Hochschule Rosenheim

Module - Cloud Architectures (CA)

FWPM Computer Science

10 - Cloud Design Patterns (III)

Prof. Dr. Marcel Tilly

Faculty of Computer Science, Cloud Computing

Vers. 25.6.2025

Agenda for Today

Cloud design pattern

taken from https://pixabay.com

Asynchronous Request-Response

Decoupling the back-end processing from a front-end host, whereby the back-end processing must be asynchronous, but the front-end requires a clear response.

Asynchronous request-response

- In applications, a synchronous API call can lead to blocking behavior
    - The work performed by the back-end takes too long. In this case, it is not possible to wait for the work to complete before responding to the request. This situation is a potential problem with all synchronous *request-response* patterns.
- Some architectures solve this problem by using a message broker to separate request and response phases.
    - This separation is often achieved by using the *Queue-Based Load Leveling Pattern*. This separation can enable independent scaling of the client process and back-end API. However, this separation also introduces additional complexity if the client requests a success notification, as this step must be asynchronous.

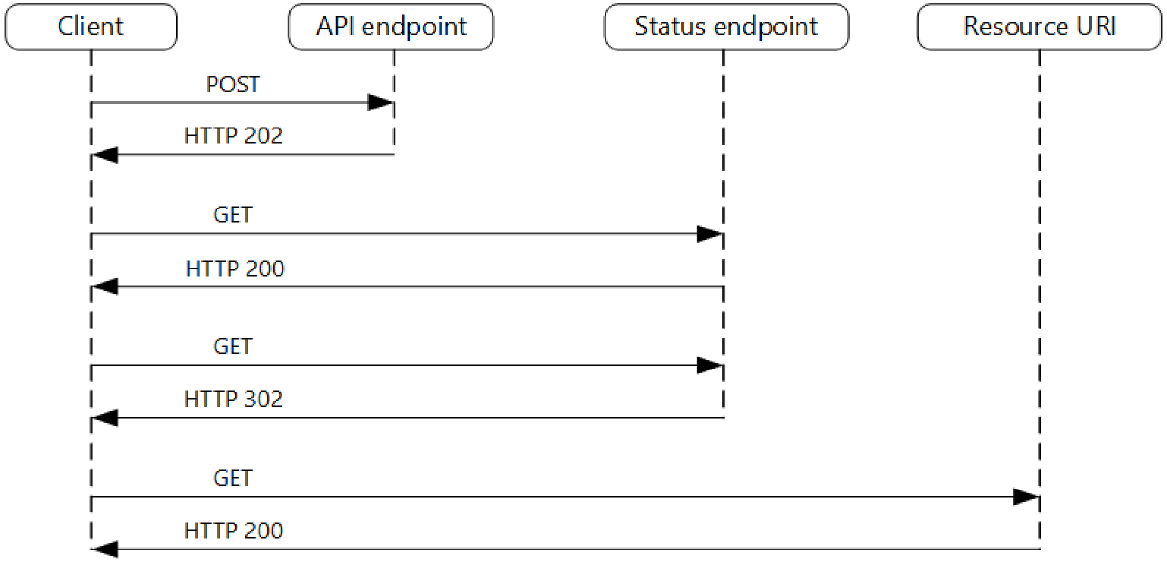
Asynchronous request-response

Solution

- The client application makes a synchronous call to the API and triggers a time-consuming process on the back end.
- The API responds synchronously as quickly as possible. It returns the status code "HTTP 202 (Accepted)", which confirms that the request has been received for processing.
- The response contains a location reference (UUID) that points to an endpoint (and process) that the client can query to check the result of the time-consuming operation.
- The API transfers the processing to another component, e.g. to a message queue.
- While the work is pending, the status endpoint returns "HTTP 202" and "In Progress" (**and an estimate**).
- After the work is complete, the status endpoint can either return a resource indicating completion or redirect to another resource URL. For example, if the asynchronous operation creates a new resource, the status endpoint would redirect to the URL for that resource.

Asynchronous request-response

Solution



Check HTTP Status Codes: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Asynchronous Request-Response

Consideration

- In some scenarios, the client may need to be given the option of canceling a time-consuming request. In this case, the back-end service must support a form of abort instruction.

Asynchronous request-response

Technische Hochschule Rosenheim

Use when ...

- Client-side code, e.g. browser applications, where it is difficult to provide callback endpoints (webhooks) or the use of time-intensive connections brings additional complexity.
- Service calls where only the HTTP protocol is available and the return service cannot trigger callbacks due to client-side firewall restrictions.
- Service calls that need to be integrated into legacy architectures that do not support modern callback technologies such as WebSockets or webhooks.

Asynchronous request-response

Do not use if …

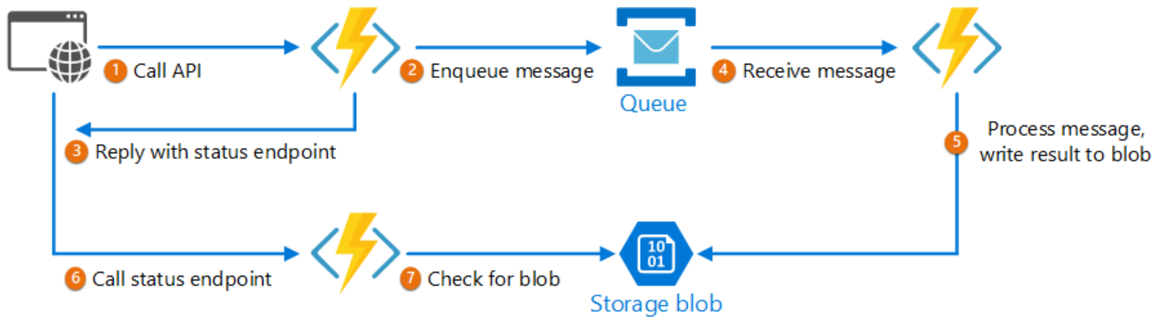
- Instead, you can use a service such as Event Grid or Pub/Sub that was created for asynchronous notifications.
- You can use server-side persistent network connections such as WebSockets or SignalR. These services can be used to notify the caller of the result.
- The network design allows you to open ports to receive asynchronous callbacks or webhooks.

Asynchronous Request-Response

Example

The solution contains three functions:

- the asynchronous API endpoint (2)
- the status endpoint (7)
- a back-end function that receives and executes work elements from the queue (5)



Asynchronous Request-Response

AsyncProcessingWorkAcceptor

The *AsyncProcessingWorkAcceptor* function implements an endpoint that accepts work items from a client application and inserts them into a queue for processing.

- The function generates a request ID and adds it to the queue message as metadata.
- The HTTP response contains a Location header that points to a status endpoint. The request ID is part of the URL path.

Asynchronous Request-Response

AsyncProcessingWorkAcceptor

```
public static class AsyncProcessingWorkAcceptor
{
    [FunctionName("AsyncProcessingWorkAcceptor")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Anonymous, "post", Route = null)] CustomerPOCO customer,
        [ServiceBus("outqueue", Connection = "ServiceBusConnectionAppSetting")] IAsyncCollector<Message> OutMessage, ILogger log)
    {
        if (String.IsNullOrEmpty(customer.id) || String.IsNullOrEmpty(customer.customername))
        {
            return new BadRequestResult();
        }
        string reqid = Guid.NewGuid().ToString();
        string rqs = $"http://{Environment.GetEnvironmentVariable("WEBSITE_HOSTNAME")}/api/RequestStatus/{reqid}";

        var messagePayload = JsonConvert.SerializeObject(customer);
        Message m = new Message(Encoding.UTF8.GetBytes(messagePayload));
        m.UserProperties["RequestGUID"] = reqid;
        m.UserProperties["RequestSubmittedAt"] = DateTime.Now;
        m.UserProperties["RequestStatusURL"] = rqs;

        await OutMessage.AddAsync(m);

        return (ActionResult) new AcceptedResult(rqs, $"Request Accepted for Processing{Environment.NewLine}ProxyStatus: {rqs}");
    }
}
```

Asynchronous Request-Response

AsyncProcessingBackgroundWorker

The *AsyncProcessingBackgroundWorker* function takes over the process from the queue, executes a few work steps based on the message payload and writes the result to the storage location.

```
public static class AsyncProcessingBackgroundWorker
{
    [FunctionName("AsyncProcessingBackgroundWorker")]
    public static void Run(
        [ServiceBusTrigger("outqueue", Connection = "ServiceBusConnectionAppSetting")]Message myQueueItem,
        [Blob("data", FileAccess.ReadWrite, Connection = "StorageConnectionAppSetting")] CloudBlobContainer inputBlob,
        ILogger log)
    {
        // Perform an actual action against the blob data source for the async readers to be able to check against.
        // This is where your actual service worker processing will be performed.

        var id = myQueueItem.UserProperties["RequestGUID"] as string;

        CloudBlockBlob cbb = inputBlob.GetBlockBlobReference($"{id}.blobdata");

        // Now write the results to blob storage.
        cbb.UploadFromByteArrayAsync(myQueueItem.Body, 0, myQueueItem.Body.Length);
    }
}
```

Asynchronous Request-Response

AsyncOperationStatusChecker

The *AsyncOperationStatusChecker* function implements the status endpoint. This function first checks whether the request has been completed.

- If the request has been completed, the function either returns the value or redirects the command directly to a URL.
- If the request is still pending, we should return a "202 Accepted" code with a self-referencing Location header.

Asynchronous Request-Response

AsyncOperationStatusChecker

```
public static class AsyncOperationStatusChecker
{
    [FunctionName("AsyncOperationStatusChecker")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route = "RequestStatus/{thisGUID}")] HttpRequest req,
        [Blob("data/{thisGuid}.blobdata", FileAccess.Read, Connection = "StorageConnectionAppSetting")] CloudBlockBlob inputBlob, string thisGUID,
        ILogger log)
    {
       // Check to see if the blob is present.
        if (await inputBlob.ExistsAsync())
        {
            // If it's present, depending on the value of the optional "OnComplete" parameter choose what to do.
            return (ActionResult)new OkObjectResult(await inputBlob.DownloadTextAsync());
        }
        else
        {
            // If it's NOT present, check the optional "OnPending" parameter.
            string rqs = $"http://{Environment.GetEnvironmentVariable("WEBSITE_HOSTNAME")}/api/RequestStatus/{thisGUID}";

            // Return an HTTP 202 status code.
            return (ActionResult)new AcceptedResult() { Location = rqs };
        }
    }
```

Scheduler Agent Supervisor Pattern

Coordinate distributed actions as a single process. If one of the actions is not successful, the errors should be handled transparently or the executed work should be undone as a whole.

Scheduler Agent Supervisor Pattern

Problem

- An application executes tasks that comprise a series of steps, some of which may call remote services or access remote resources. The individual steps can be independent of each other, but they are orchestrated by the application logic that implements the task.
- Whenever possible, the application should ensure that the task is executed to completion and resolve any errors that may occur when accessing remote services or resources.
- If the application detects a more long-term error for which no simple recovery is possible, it must be able to restore the system to a consistent state and ensure integrity for the entire operation.

Scheduler Agent Supervisor

The pattern "Scheduler-Agent-Supervisor" defines 3 actors.

- The **Scheduler** handles the steps for the task to be executed and orchestrates their execution.
  - These steps can be combined in a pipeline or a workflow.
  - The scheduler must ensure that the steps are executed in the correct order.
  - As each step is executed, the scheduler records the state of the workflow, such as "step not yet started", "step being executed" or "step completed".
  - The state information must also contain an upper limit for the time available to complete the step, known as the time to completion.
  - If a step requires access to a remote service or resource, the scheduler calls the corresponding agent and passes it the details of the work to be performed.

Scheduler Agent Supervisor

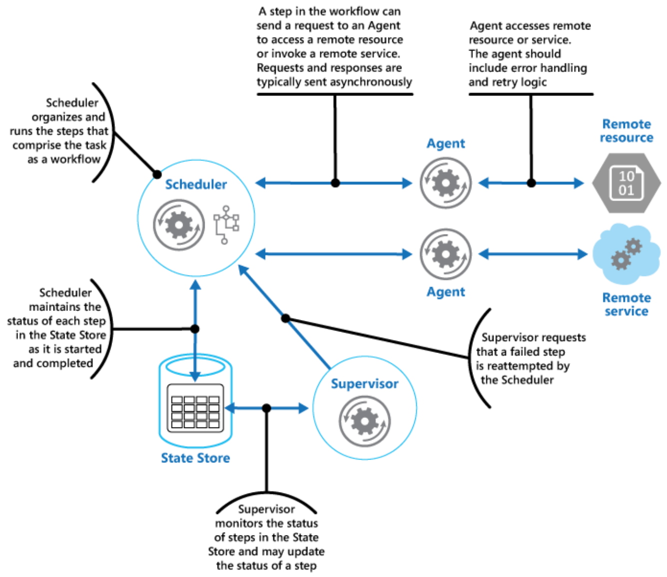
- The **Agent** includes logic that encapsulates a call to a remote service or access to a remote resource that is referenced by a step in a task.
    - Each agent encloses calls to a single service or resource and implements the appropriate error handling and retry logic (subject to a timeout constraint described later) (this is an implementation detail of the pattern).

Scheduler Agent Supervisor

- The **Supervisor** monitors the status of the steps in the task that is executed by the scheduler.
    - The supervisor is executed at regular intervals (the frequency depends on the system) and examines the status of the steps managed by the scheduler.
    - If the supervisor detects a timeout or an error, it prompts the responsible agent to restore the step or perform the appropriate cleanup action (this may involve changing the status for a step).
    - Note that the restore or cleanup actions are implemented by the scheduler and the agents. The supervisor only requests that these actions are executed.

Scheduler Agent Supervisor Pattern

Scheduler, Agent and Supervisor are logical components and their physical implementation depends on the technology used.


A step in the workflow can send a request to an Agent to access a remote resource or invoke a remote service. Requests and responses are typically sent asynchronously
Agent accesses remote resource or service. The agent should include error handling and retry logic
Scheduler organizes and runs the steps that comprise the task as a workflow
Scheduler
Agent
Agent
Remote resource
10 01
Remote service
Scheduler maintains the status of each step in the State Store as it is started and completed
Supervisor requests that a failed step is reattempted by the Scheduler
Supervisor
State Store
Supervisor monitors the status of steps in the State Store and may update the status of a step

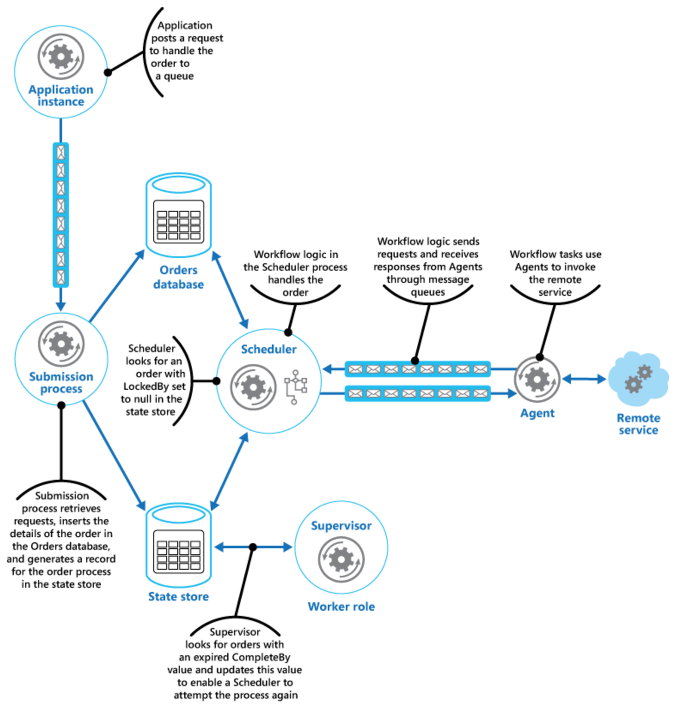
Scheduler Agent Supervisor Pattern

Considerations

The following points should be considered when deciding how to implement this pattern:

- This pattern can be difficult to implement and requires extensive testing of all possible failure modes of the system.
- The recovery/retry logic implemented by the scheduler is complex and depends on the state information in the state memory. It may also be necessary to record the information required to implement a *balancing transaction* in a persistent data store.
- It is important how often the supervisor is executed. The supervisor should be executed often enough to prevent unsuccessful steps from blocking an application for an extended period of time. At the same time, it should not be executed too often to avoid overhead.
- The steps processed by an agent can be executed several times. The logic for implementing these steps should be idempotent.

Scheduler Agent Supervisor Pattern


Application instance
Application posts a request to handle the order to a queue
Orders database
Workflow logic in the Scheduler process handles the order
Workflow logic sends requests and receives responses from Agents through message queues
Workflow tasks use Agents to invoke the remote service
Submission process
Scheduler looks for an order with LockedBy set to null in the state store
Scheduler
Agent
Remote service
Submission process retrieves requests, inserts the details of the order in the Orders database, and generates a record for the order process in the state store
State store
Supervisor
Worker role
Supervisor looks for orders with an expired CompleteBy value and updates this value to enable a Scheduler to attempt the process again

Monitoring Pattern

Functional checks in an application that external tools can access at regular intervals via available endpoints. This can help to check the proper execution of applications and services.

Monitoring Pattern

Problem

- It is a best practice and often a business requirement to monitor web applications and back-end services to ensure that they are available and functioning properly.
  - However, services running in the cloud are more difficult to monitor than on-premises services.
  - Lack of control over the hosting environment, and services usually depend on other services.
- There are many factors that affect cloud-based applications, such as network latency, performance and availability of the underlying compute and storage systems and the network bandwidth between them.
  - These factors can cause the service to fail in whole or in part. You must therefore check at regular intervals whether the service is working properly.

Monitoring Pattern

Solution

Implement integrity monitoring by sending requests to an endpoint of the application. The application must perform the necessary checks and return a status.

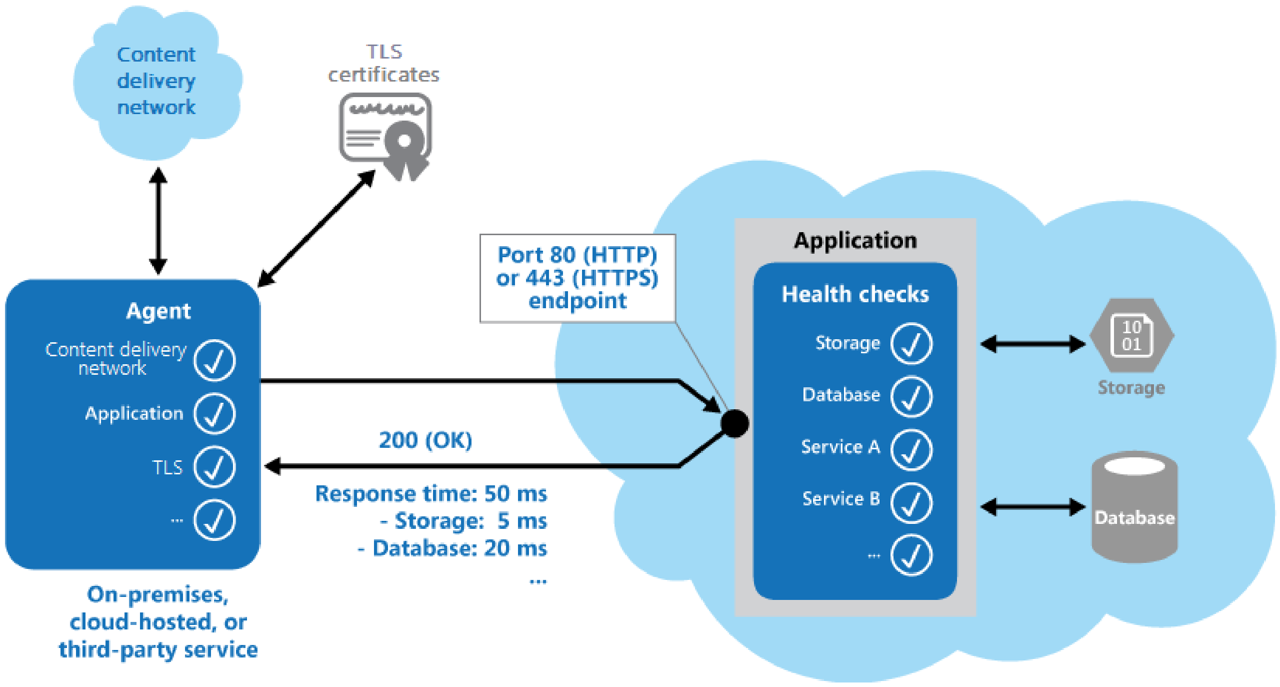
An integrity monitoring check typically combines two factors:

- The checks (if any) performed by the application or service in response to the request to verify the integrity of the endpoint.
- The analysis of the results by the tool or framework that performs the integrity check.

The response code indicates the status of the application and optionally the components or services used by it. The wait time or response time check is performed by the monitoring tool or framework.

Monitoring pattern

Solution



Task

Please note down typical tests that could be carried out in your application!

Go to: https://zumpad.zum.de/p/ca2021

Monitoring Pattern

Considerations 1/2

- **Validation of the response**: For example, is just the status code 200 (OK) sufficient to confirm that the application is working properly?
- **Endpoint**: Whether to use the same endpoint for monitoring that is used for general access, but selecting a specific path on the general access endpoint that is intended for integrity checking, e.g. "/HealthCheck/{GUID}/". In this way, some functional tests in the application can be performed by the monitoring tools, such as adding a new user registration, logging in and placing a test order, while checking that the general access endpoint is available.
- The type of information to be collected in the service in response to monitoring requests and how this information can be returned.

Monitoring Pattern

Considerations 2/2

- **Scope of information to be collected**: An excessive processing load during the check can slow down the application and thus have an impact on other users.
- Configure the security of the monitoring endpoints to protect against public access. This can expose the application to malicious attacks, compromise the application due to the disclosure of sensitive information or fall victim to Denial of Service (DoS) attacks.
- **Ensure that the monitoring agent is working properly**: One approach is to make an endpoint available that simply returns a value from the application configuration or a random value that can be used to test the agent.

Monitoring Pattern

Use of this pattern

This pattern is helpful:

- Monitoring websites and web applications for availability.
- Monitoring websites and web applications for proper operation.
- Monitoring of mid-level or shared services to detect and isolate an error that could disrupt other applications.

External Configuration Store

Move configuration information from the application deployment package to a central storage location.

- This can make it easier to manage and control configuration data and share configuration data across all applications and application instances.

Context

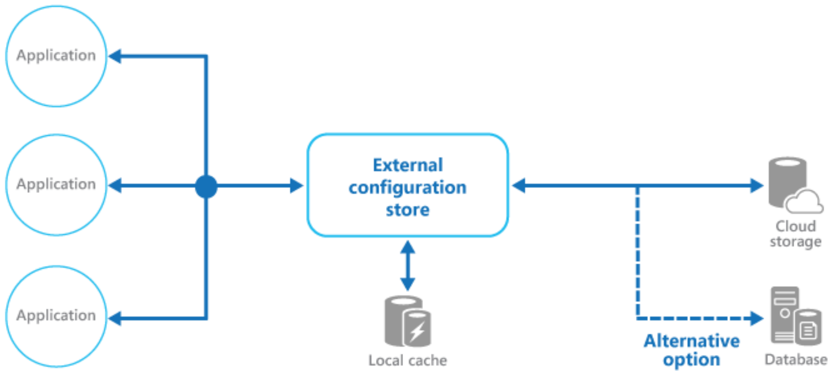
- Most application runtime environments contain configuration information stored in files that are provided with the application.
- In some cases, these files can be edited to change the behavior of the application after it has been deployed. However, after changes are made to the configuration, the application must be redeployed, often resulting in unacceptable downtime and other administrative overhead.
- Local configuration files also restrict configuration to a single application, but occasionally it would be useful to share configuration settings across multiple applications.
    - Examples: Database connections, UI design information, the URLs of endpoints
- It can be challenging to manage changes to configurations across multiple running instances of the application, especially in a cloud-hosted scenario.

Solution

- Save configuration information in an external memory
  - The memory has an interface for retrieving and changing the configuration settings
- The type of external storage depends on the hosting and runtime environment of the application.
  - In a cloud-hosted scenario, it is usually a cloud-based service
  - the service is abstracted from the actual storage
- The backup storage should have an interface that allows consistent, user-friendly access (REST).
  - The implementation may also need to authorize user access to protect configuration data
  - Management of multiple versions of the configuration (such as development, staging and production versions, each including multiple releases) should be possible

Caching

- Many integrated configuration systems read the data when the application is started and cache the data in memory to enable fast access and minimize the impact on application performance.
- Depending on the type of backup memory used and the latency of this memory, it may make sense to implement a caching mechanism in the external configuration memory.



Use of this pattern

Use when...

- ... for configuration settings that are shared between different applications and application instances, or where a default configuration needs to be enforced between multiple applications and application instances.
- ... as a method to simplify the management of multiple applications and optionally monitor the use of configuration settings by logging some or all access types to the configuration store.

Sample implementation can be found here:

- https://github.com/mspnp/cloud-design-patterns/tree/master/external-configuration-store

Azure Key Vault

Azure Key Vault is a tool for securely storing and accessing secrets.

Context

The following problems can be solved with Azure Key Vault:

- **Secret management**: Azure Key Vault enables secure storage and precise control of access to tokens, passwords, certificates, API keys, and other secrets.
- **Key management**: Azure Key Vault can also be used as a key management solution. Azure Key Vault makes it easy to create and manage the encryption keys used to encrypt your data.
- **Certificate management**: In addition, the Azure Key Vault service allows you to conveniently provision and manage public and private Secure Sockets Layer/Transport Layer Security (SSL/TLS) certificates for use with Azure and your internal connected resources.
- **Storing secrets**: The secrets and keys are encrypted and protected.

Why?

Centralize application secrets

- By centralizing the storage of application secrets in Azure Key Vault, the distribution can be controlled (see *External Configuration Store*).
- With a *Key Vault*, risks of unintentional disclosure of secrets can be reduced.
    - Application developers can store security information in the *Key Vault* and no longer store it in the application.
    - If security information no longer needs to be stored in applications, there is no need to include this information in the code.
    - **An example**: Suppose an application needs to connect to a database. Instead of storing the connection string in the app code, you can store it securely in the Key Vault.
    - Applications can securely access required information using URIs. These URIs allow applications to retrieve specific versions of a secret key.

Why?

Secure storage of secrets and keys

- Secrets and keys are protected with industry-standard algorithms, key lengths and hardware security modules (HSMs). The HSMs used fulfill the requirements of FIPS 140-2, Level 2 (Federal Information Processing Standards).
- A caller (user or application) can only access a *Key Vault* after proper authentication and authorization.
    - During authentication, the identity of the caller is determined.
    - Authorization, on the other hand, determines which operations the caller is allowed to perform.
- Authentication takes place via Azure Active Directory.
- Role-based access control (RBAC) can be used for authorization.
- Data in the *Key Vault* cannot be viewed or extracted by third parties.

Why?

Monitoring access and usage

- The *Key Vault* can be used to monitor how and when keys and secrets are accessed.
- Coupling with other services:
    - Archiving in a storage account
    - Streaming to an event hub
    - Sending the logs to Azure Monitor logs

Why?

Easier management of application secrets

- Eliminate the need for internal knowledge of hardware security models
- Replication of the contents of a Key Vault instance within a region and in a secondary region.
    - Data replication ensures high availability of information and failover can be triggered without administrator intervention.
- Provide standard Azure management options via the portal, Azure command line interface and PowerShell
- Automate certain tasks related to certificates you acquire from public certificate authorities (e.g. registration and renewal)

Create a key vault

- Create a Key Vault instance:
  - **Name**: Give your key vault a unique name.
  - **Resource Group**: RG in which the Key Vault will be created.
  - **Location**: Data center

```
$ az keyvault create --name "Infca-Vault" --resource-group "inf-ca"
  --location westeurope
```

Add a secret

A secret can be added to the *Key Vault* with a few additional steps.

Example:

- A password can then be used by an application.
- The password should be called `ExamplePassword` and contain the value `hVFkk965BuUv`.

```
$ az keyvault secret set --vault-name "Infca-Vault" --name "ExamplePassword"
   --value "hVFkk965BuUv"
```

Retrieve a secret

```
$ az keyvault secret show --name "ExamplePassword"
--vault-name "Infca-Vault"
```

or

`https://Infca-Vault.vault.azure.net/secrets/ExamplePassword`

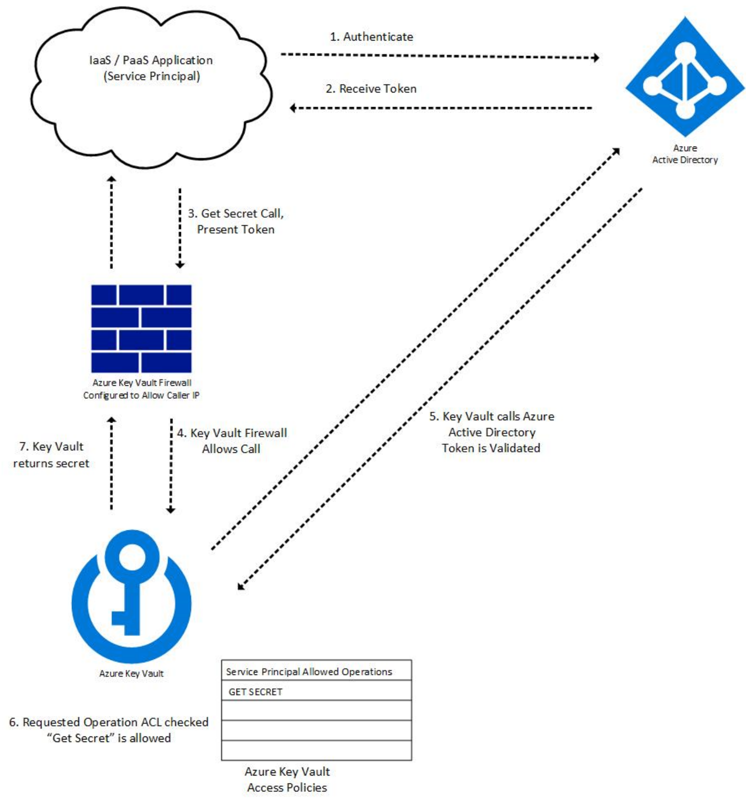
Best Practices (1/2)

- **Control access to your key vault**
  - Block access to your subscription, resource group and key vaults (RBAC).
  - Create access policies for each key vault.
  - Use the principal with the least privileges to grant access.
  - Enable the firewall and VNET service endpoints.
- **Use separate key vaults**: It is recommended to use one Key Vault per application and environment (development, pre-production and production).

Best Practices (2/2)

- **Backup**: Make sure you regularly backup your Key Vault when updating/deleting/creating items in a Key Vault.
- **Enable logging**: Enable logging for your Key Vault.
- **Enable recovery options**
    - Activate temporary deletion.
    - Activate delete protection if you want to protect yourself from forced deletion of the secret/key vault even after activating temporary deletion.

Key vault authentication

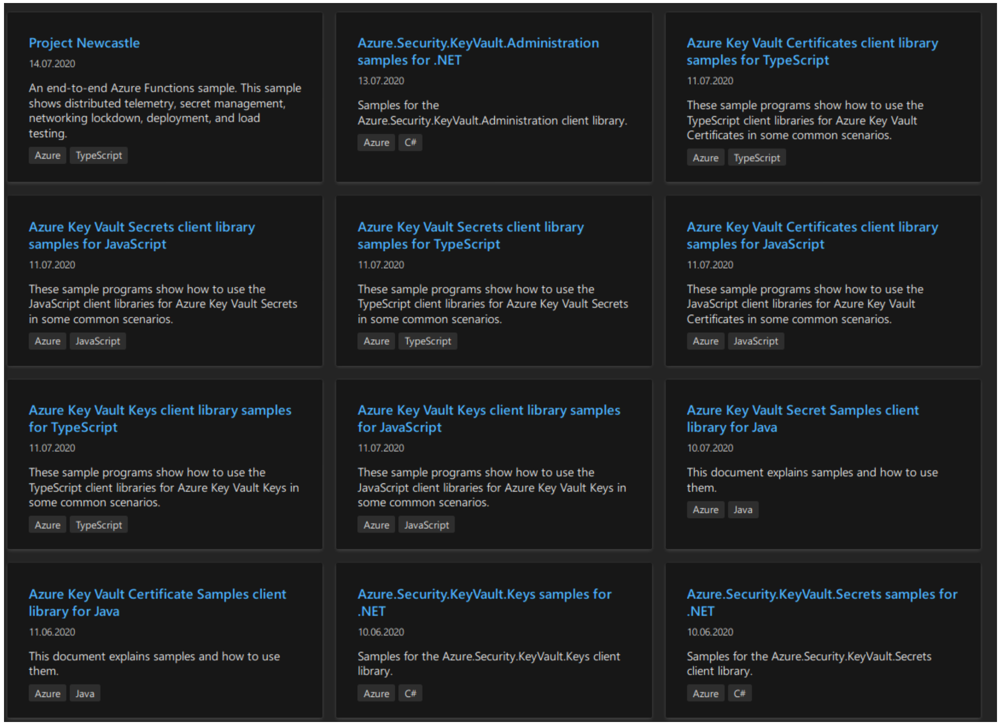


IaaS / PaaS Application (Service Principal)
1. Authenticate
2. Receive Token
Azure Active Directory
3. Get Secret Call, Present Token
Azure Key Vault Firewall Configured to Allow Caller IP
7. Key Vault returns secret
4. Key Vault Firewall Allows Call
5. Key Vault calls Azure Active Directory Token is Validated
Azure Key Vault
6. Requested Operation ACL checked "Get Secret" is allowed
Azure Key Vault Access Policies

Service Principal Allowed Operations
GET SECRET

Code samples

https://docs.microsoft.com/de-de/samples/browse/?expanded=azure&products=azure-key-vault



Project Newcastle
14.07.2020
An end-to-end Azure Functions sample. This sample shows distributed telemetry, secret management, networking lockdown, deployment, and load testing.
Azure | TypeScript

Azure.Security.KeyVault.Administration samples for .NET
13.07.2020
Samples for the Azure.Security.KeyVault.Administration client library.
Azure | C#

Azure Key Vault Certificates client library samples for TypeScript
11.07.2020
These sample programs show how to use the TypeScript client libraries for Azure Key Vault Certificates in some common scenarios.
Azure | TypeScript

Azure Key Vault Secrets client library samples for JavaScript
11.07.2020
These sample programs show how to use the JavaScript client libraries for Azure Key Vault Secrets in some common scenarios.
Azure | JavaScript

Azure Key Vault Secrets client library samples for TypeScript
11.07.2020
These sample programs show how to use the TypeScript client libraries for Azure Key Vault Secrets in some common scenarios.
Azure | TypeScript

Azure Key Vault Certificates client library samples for JavaScript
11.07.2020
These sample programs show how to use the JavaScript client libraries for Azure Key Vault Certificates in some common scenarios.
Azure | JavaScript

Azure Key Vault Keys client library samples for TypeScript
11.07.2020
These sample programs show how to use the TypeScript client libraries for Azure Key Vault Keys in some common scenarios.
Azure | TypeScript

Azure Key Vault Keys client library samples for JavaScript
11.07.2020
These sample programs show how to use the JavaScript client libraries for Azure Key Vault Keys in some common scenarios.
Azure | JavaScript

Azure Key Vault Secret Samples client library for Java
10.07.2020
This document explains samples and how to use them.
Azure | Java

Azure Key Vault Certificate Samples client library for Java
11.06.2020
This document explains samples and how to use them.
Azure | Java

Azure.Security.KeyVault.Keys samples for .NET
10.06.2020
Samples for the Azure.Security.KeyVault.Keys client library.
Azure | C#

Azure.Security.KeyVault.Secrets samples for .NET
10.06.2020
Samples for the Azure.Security.KeyVault.Secrets client library.
Azure | C#

Sharding

Divide a data store into a set of horizontal partitions or shards. This can improve scalability when storing and accessing large amounts of data.

Context

A data store hosted by a single server may be subject to the following restrictions:

- **Storage space**: A data store for a large-scale cloud application is expected to contain a huge amount of data, which could increase significantly over time. A server normally only offers a limited amount of data storage.
- **Computing resources**: A single server hosting the data store may not be able to provide the required compute power.
- **Network bandwidth**: It is possible that the amount of network traffic exceeds the capacity of the network used to connect to the server, resulting in request errors.
- **Geography**: It may be necessary to store data generated by certain users located in the same region as those users for legal, compliance or performance reasons, or to reduce the latency of data access.

Solution

Technische Hochschule Rosenheim

- Divide the data storage into horizontal partitions or **shards**.
- Each **shard** has the same schema, but has its own unique subset of the data.
- A shard is a standalone data store (it can contain the data for many entities of different types) that runs on a server that acts as a storage node.

Shards

This pattern has the following advantages:

- You can scale up the system horizontally by adding more shards that are executed on additional memory nodes.
- A system can use standard hardware for each storage node instead of specialized and expensive computers.
- You can reduce conflicts and improve performance by distributing the workload across multiple shards.
- In the cloud, shards can be physically located near the users accessing the data.

Shards

- A shard typically contains elements that fall within a certain range, which is determined by at least one attribute of the data.
  - These attributes form the shard key (also known as the partition key).
  - The shard key should be static. It should not be based on data that may change.

Physical Setup

- Sharding is used to arrange the data physically. When an application stores and retrieves data, the sharding logic directs the application to the appropriate shard.
  - This sharding logic can be implemented as part of the data access code in the application, or it can be implemented by the data storage system if it obviously supports sharding.
- The abstraction of the physical locations of the data in the sharding logic provides a high degree of control over which shards contain which data.
  - It also allows data to be migrated between shards without having to rework the business logic of an application if the data in the shards needs to be redistributed later (e.g. if the shards are unbalanced).
  - The disadvantage here is the additional data access effort required to determine the location of the individual data elements when they are retrieved.

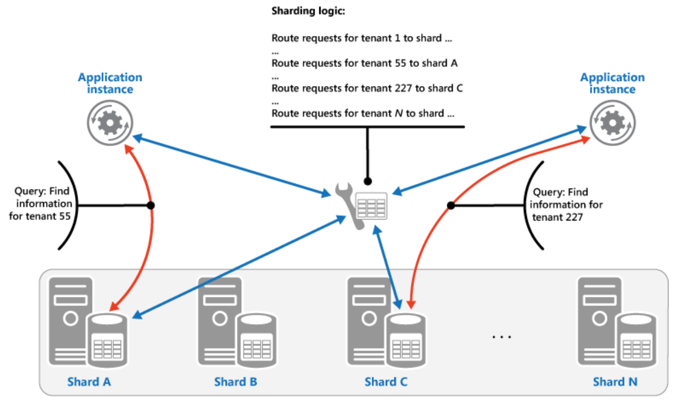
Access via shards

- To ensure optimal performance and scalability, it is important to split the data in such a way that it is suitable for the queries executed by the application.
- In many cases, it is unlikely that the sharding scheme will exactly match the requirements of each query.
    - For example, in a multi-instance system, an application may need to retrieve the client data via the client ID. However, it may also need to search for this data using another attribute, e.g. client name or location. To handle these situations, implement a sharding strategy with a shard key that supports the most common queries.
- If queries regularly retrieve data using a combination of attribute values, composite shard keys can be defined by linking attributes together.

Sharding strategies

Search strategy: In this strategy, the sharding logic implements a mapping that forwards a data request to the shard containing that data using the shard key.

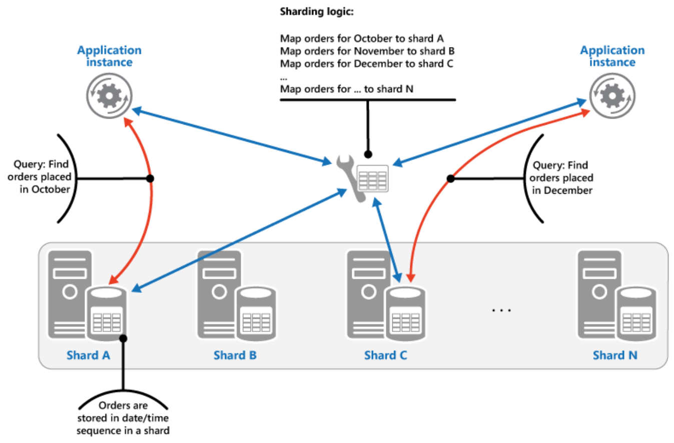
- In a multi-instance application, all of a client's data can be stored in a shard using the client ID as the shard key. Multiple clients can share the same shard, **but the data for a single client is not distributed across multiple shards**.


Sharding logic:
Route requests for tenant 1 to shard ...
...
Route requests for tenant 55 to shard A
...
Route requests for tenant 227 to shard C
...
Route requests for tenant N to shard ...
Application instance
Application instance
Query: Find information for tenant 55
Query: Find information for tenant 227
Shard A
Shard B
Shard C
...
Shard N

Sharding strategies

Sharding strategy: In this strategy, related elements are grouped in the same shard and arranged by shard key. The shard keys are consecutive.

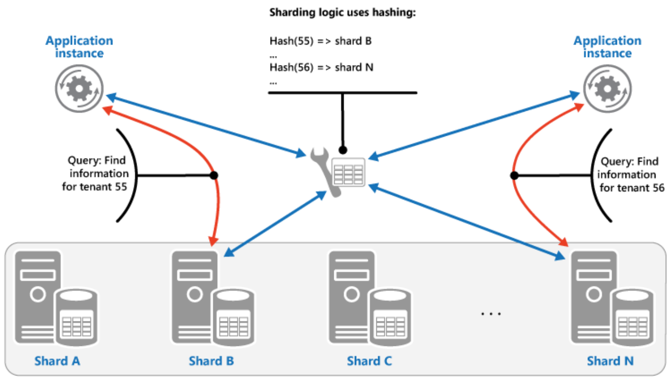
- For example, if an application regularly needs to find all orders placed in a particular month, this data can be retrieved more quickly if all orders for a month are stored in the same shard by date and time.


Sharding logic:
Map orders for October to shard A
Map orders for November to shard B
Map orders for December to shard C
...
Map orders for ... to shard N
Application instance
Application instance
Query: Find orders placed in October
Query: Find orders placed in December
Shard A
Shard B
Shard C
...
Shard N
Orders are stored in date/time sequence in a shard

Sharding strategies

Hash strategy: The aim of this strategy is to reduce the probability of so-called hotspots (shards that are disproportionately loaded). Distributes the data across the shards in a way that achieves a balance between the size of the individual shards and the average workload that occurs for the individual shards.

- The sharding logic calculates the shard for storing an element based on a hash of at least one attribute of the data. The selected hash function should distribute the data evenly across the shards, possibly by introducing a random element into the calculation.



Pros and cons

Search strategy

- Provides greater control over the configuration and use of shards
- Using virtual shards reduces the impact of redistributing data as new physical partitions can be added to balance workloads.
- The mapping between a virtual shard and the physical partitions that implement the shard can be changed without affecting the application code that uses a shard key to store and retrieve data.
- Searching for shard locations can cause additional overhead.

Pros and cons

Divisional strategy

- This strategy is easy to implement and works well for range queries as they can often retrieve multiple data elements from a single shard in a single operation.
- This strategy offers simpler data management. For example, if users are in the same region in the same shard, updates can be scheduled in the respective time zones based on the local workload and demand pattern.
- However, this strategy does not provide optimal balancing between shards.
- Rebalancing for shards is difficult and may not resolve the issue of uneven workloads if the majority of activity is for adjacent shard keys.

Pros and cons

Hash strategy

- This strategy offers a higher probability that data and workloads will be distributed more evenly.
- The routing of requests can be carried out directly via the hash function. It is not necessary to manage an assignment.
- The hash calculation may require additional effort.
- Redistribution to the shards is also difficult.

Example

- The following example in C# uses a series of SQL Server databases that act as shards.
- Each database contains a subset of the data used by an application.
- The application retrieves data distributed across shards using its own sharding logic (this is an example of a fan-out query).

Code

- The details of the data contained in each shard are returned by a method called `GetShards`.
- This method returns an enumerable list of ShardInformation objects, where the ShardInformation type contains an identifier for each shard and the SQL Server connection string that an application should use to connect to the shard (the connection strings are not shown in the code example).

```
private IEnumerable<ShardInformation> GetShards()
{
  // This retrieves the connection information from a shard store
  // (commonly a root database).
  return new[]
  {
    new ShardInformation
    {
      Id = 1, ConnectionString = ...
    },
    new ShardInformation
    {
      Id = 2, ConnectionString = ...
    }
  };
}
```

Code

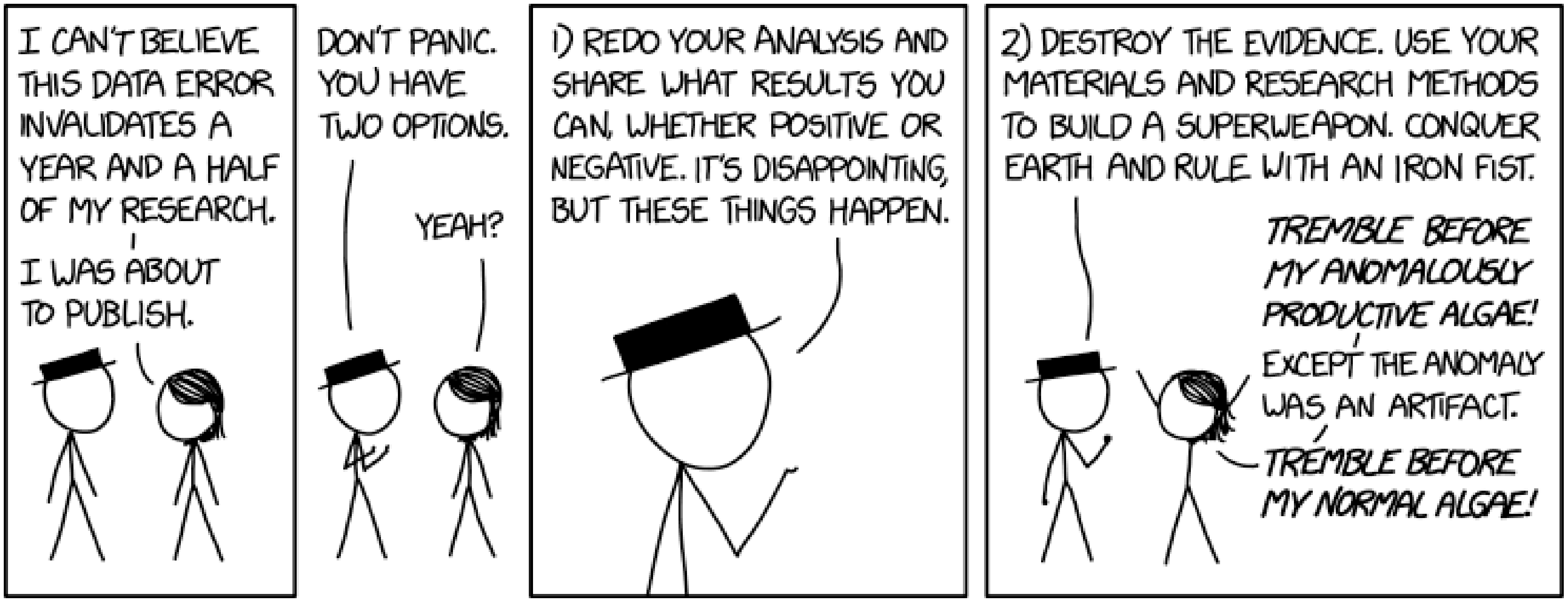
```
// Retrieve the shards as a ShardInformation[] instance.
var shards = GetShards();

var results = new ConcurrentBag<string>();

// Execute the query against each shard in the shard list.
// This list would typically be retrieved from configuration
// or from a root/master shard store.
Parallel.ForEach(shards, shard =>
{
  // NOTE: Transient fault handling isn't included,
  // but should be incorporated when used in a real world application.
  using (var con = new SqlConnection(shard.ConnectionString))
  {
    con.Open();
    var cmd = new SqlCommand("SELECT ... FROM ...", con);

    Trace.TraceInformation("Executing command against shard: {0}", shard.Id);

    var reader = cmd.ExecuteReader();
    // Read the results in to a thread-safe data structure.
    while (reader.Read())
    {
      results.Add(reader.GetString(0));
    }
  }
});

Trace.TraceInformation("Fanout query complete - Record Count: {0}",
                        results.Count);
```

Summary

Lessons Learned:

- Cloud design patterns
  - Asynchronous Request-Response
  - Scheduler Agent Supervisor Pattern
  - Monitoring
  - Configuration and Key Vault
  - Sharding

An almost final though …


I CAN'T BELIEVE THIS DATA ERROR INVALIDATES A YEAR AND A HALF OF MY RESEARCH.
I WAS ABOUT TO PUBLISH.
DON'T PANIC. YOU HAVE TWO OPTIONS.
YEAH?
1) REDO YOUR ANALYSIS AND SHARE WHAT RESULTS YOU CAN, WHETHER POSITIVE OR NEGATIVE. IT'S DISAPPOINTING, BUT THESE THINGS HAPPEN.
2) DESTROY THE EVIDENCE. USE YOUR MATERIALS AND RESEARCH METHODS TO BUILD A SUPERWEAPON. CONQUER EARTH AND RULE WITH AN IRON FIST.
TREMBLE BEFORE MY ANOMALOUSLY PRODUCTIVE ALGAE!
EXCEPT THE ANOMALY WAS AN ARTIFACT.
TREMBLE BEFORE MY NORMAL ALGAE!

Task

Exercise

- https://inf-git.fh-rosenheim.de/inf-ca/10_uebung

  Monitor your application:

  - `/ping` : Returns a sign of life
  - `/healthcheck`: Returns information about the status of your solution

Project

- Implement!!!