Tedega Documentation

Release 0.1.0

Torsten Irländer

Dez. 10, 2017

Inhaltsverzeichnis

KAPITEL 1

Tedega

Tedega ist ein Python basiertes Spielland für Mircoservices. Es bietet Bibliotheken und Werkzeuge für den Bau und Betrieb von verteilten Anwendungen auf der Basis von Microservices.

Dieses Paket ist das Meta-Paket des Tedega Projekt. Es enthält die allgemeine Dokumentation, Scripte um eine Anwendung auf der Basis von Tedega zu bootstrappen und installiert alle notwendigen Abhängigkeiten in der letzten stabilen Version.

- Freie Software: MIT Lizenz
- Dokumentation: https://tedega.readthedocs.io.
- Projekt Status: Alpha, Konzeption

1.1 Funktionen

- Umfangreiches Logging mit Fluentd

1.2 Installation

1.2.1 Stabiler Release

Um Tediga zu installieren, führe folgendes Kommando in deinem Terminal aus:

```
$ pip install tedega
```

Das ist die bevorzugte Methode um Tediga zu installieren, da sie immer die letzte stabile Version installiert.

Wenn du pip nicht installiert hast, dann hilft die Python Installation Anleitung.

1.2.2 Entwicklungsversion

Die Quellen von Tediga können vom Github repo geladen werden.

Du kannst das öffentliche Repository klonen und dann instalieren:

```
$ git clone git://github.com/toirl/tedega
$ cd tedega
$ python setup.py install
```

KAPITEL 2

Architektur

Dieser Abschnitt gibt einen allgemeinen Überblick über den Aufbau und die Architektur von Tedega. Dabei wird beschrieben, wie mit Hilfe der verschiedenen Komponenten eine Anwendung bestehend aus verschiedenen Microservices erstellt und betrieben werden können.

Detailierte Informationen finden sich in der Dokumentation der jeweiligen Komponente.

Über die Modellierung von Microservices gibt es viel Literatur. Auf die Quellen, die den Aufbau von Tediga maßgeblich beeinflusst haben möchte ich gerne verweisen: Viele Ideen und Ansätze zum Aufbau von Microservices stammen das dem Buch "Building Microservices" von Sam Newmann *[Newmann2015]*. Weiter sind einige Ideen (Insbesonderer zur deployment) durch das Buch "The DevOps Handbook" *[DOP2016]* inspiriert. Hilfreich bei der Modellierung von gut abgegrenzen Diensten war "Implementing Domain-Driven Design" *[Ver2013]*.

Zunächst wird der Aufbau eines einzelnen *View* beschrieben. Dann wird beleuchtet wie Microservices innerhalb der *Anwendung* interagieren. Neben dem grundsätzlichen Aufbau werden auch die *Sicherheit* betrachtet und *Design Prinzipien* beschrieben, die bei der Entwicklung von Tediga berücksicht wurden.

2.1 Microservice

Ein Microservice ist in einem Schichtmodell ähnlich des MVC Modell organisiert. Allerdings wird das Schichtmodell um eine weitere Schicht erweitert, die speziell die Speicherung der Daten behandelt. Tedega nutzt für die Implementation eines Microservices somit ein MVCS Schichtmodell (Model, View, Controller, Store). Jede der Schichten ist als eigenständige Komponente (Bibliothek) implementiert, die die Aufgaben der jeweiligen Schicht übernimmt.

Ziel dieser Architektur ist eine möglichst klare Abgrenzung und Trennung von Verantwortlichkeiten zwischen den Schichten. Diese Trennung wird durch die Implementation in unterschiedlichen Bibliotheken unterstrichen. Damit folgt die Architektur grundsätzlich der Idee des Single Responsibility Prinzip (SRP). Weiter ergibt sich die eine loose Kopplung zwischen den Schichten und dadurch die Flexibilität Microservices in unterschiedlichen Versionen einer Komponente zu betreiben, was das Risiko bei Updates minimiert.

Als Ergebnis erhalten wir Komponenten, die über mehrere Microservices wiederverwendet werden können, und vermeiden so die Duplizierung von Code (DRY). Dadurch kann man sich bei der Implementation eines Microservice voll auf den Kern konzentrieren: Der Implementation der Geschäftslogik innerhalb der *Domain*.

Das Herzstück eines Microservice ist die *Domain* Komponente. Sie ist der individuelle Teil eines Microservice und bestimmt wie ein Microservice funktioniert. Sie implementiert das Datenmodell (Modell) und alle Details der Geschäftslogik (Controller). Details zur Speicherung oder die Behandlung von Requests werden nicht behandelt, sondern werden an jeweils anderen Komponente delegiert.

Die *View* Komponente dient als Einstieg in den Microservice und behandelt sämtliche Aspekte der Behandlung von HTTP Anfragen. Sie macht die durch die *Domain* definierter Controller über eine REST-API öffentlich verfügbar. Sämtliche Zugriffe erfolgen ausschließlich über die durch die View definierten REST-API.

Die *Storage* Komponente abstrahiert die Speicherung (Store) von Daten und erlaubt der *Domain* die Daten aus dem Model zu speichern.

Die *Share* Komponente stellt allgemeine Funktionalität für die übrigen Komponenten zur Verfügung.

2.2 Anwendung

Eine Anwendung setzt sich in seiner Gesamtfunktion aus verschiedenen Microservices zusammen. Jeder Microservice übernimmt einen klar abgrenzte Teilfunktion.

In einer Anwendung für ein Versandhaus könnte ein Service die Kundendaten verwalten, und anderer den Lagerbestand, das Abrechnungssystem oder den Warenkorb. Die Abgrenzung von den einzelnen Services ist eine nicht triviale Aufgabe und Bedarf viel Erfahrung, Überlegungen und Klärung im Vorfeld. Sehr hilfreich bei dem Ermitteln von diesen Grenzen sind Methoden aus dem *Domain Driven Desigen (DDD)* die unter anderem auch in *[Ver2013]* beschrieben sind.

In der Grafik sind drei Services zu sehen. Jeder Service ist weitgehend unabhängig von anderen Services. Ein Service speichert seine Daten in seiner eigenen Datenbank und enthält sämtliche Geschäftslogik. Jeder der Services bietet über eine REST-API seine Dienste an.

2.2.1 Inter Service Kommunikation

Wir haben gesehen, dass jeder Service möglichst unabhängig von anderen Diensten sein soll. Dadurch ergibt sich in einer verteilten Anwendung naturgemäß sehr schnell der Bedarf, dass Informationen zwischen den Services ausgetauscht werden müssen.

Der wahrscheinlich häufigste Grund für den Austausch von Daten ist, dass ein Service die notwendigen Daten, die er zur Bearbeitung einer Anfrage benötigt, nicht vollständig selber speichert und diese von einem anderen Dienst abgefragt werden müssen. Ein anderer Grund kann sein dass andere Dienste in Folge einer Änderung an den Daten benachrichtigt werden müssen, damit diese eigene Aktionen ausführen.

Inter Service Kommunikation bezeichnet den Austausch von Daten zwischen den einzelnen Microservices innerhalb der Anwendung. Das können Benachrichtigungen über Ereignisse sein, oder das Laden von weiteren Informationen und Daten aus anderen Quellen.

Bemerkung: Eine weitere häufig anzufindene und vielleicht naheliegende Möglichkeit zur Umsetzung dieser Kommunikation ist ein zentraler Service, der die Koordination zwischen den verschiedenen Services übernimmt.

Allderdings verletzt diese zentrale Instanz gleich in mehreren Punkten das Prinzip der loosen Kooplung und hohen Zusammenhalt: Erstens wird durch eine zentrale koordinierende Instanz eine starke Kopplung zwischen den Services eingeführt. Zweitens wird zusammenhängende Logik über mehrere Services verteilt. Daher wird dieser Ansatz in Tediga nicht weiter berücksichtigt.

Tediga sieht für die Kommunikation zwei verschiedene Arten vor:

1. Direkte Kommunikation zwischen den Microservices. Diese findet ausschließlich per HTTP über die jeweilige öffentliche REST-API der Services statt. Ein Service agiert dabei wie ein gewöhnlicher Client.
2. Indirekte Kommunikation über eine Message-Queue. Diese wird verwendet, um anderen Services zu benachrichtigen. Dabei schreibt ein Service alle Dinge, über die er andere Services informieren möchte in die Queue. Die anderen Dienste lesen diese Nachrichten und entscheiden selbständig, ob Sie selber tätig werden müssen. Ein Beispiel: Der Nutzer-Service des Versandhaus löscht einen Nutzer und schreibt diese Aktion in die Message Queue. Der Warenkorb liest diese Nachricht und löscht daraufhin hin den zu dem Nutzer gehörenden Warenkorb.

Als Message-Queue wird die Software RabbitMQ verwendet.

2.2.2 Logging

Um den Betrieb der Anwendung zu überwachen benötigen wir einen Mechanismus zum Protokollieren von verschiedenen Metriken unserer Dienste. Diese Informationen helfen uns zu beurteilen ob unsere Anwendung gut funktioniert. Sie ermöglichen uns frühzeitig Engpässe zu erkennen, zu sehen dass ein Dienst ausgefallen ist, oder ob Fehler auftreten, und in welcher Form die Anwendung genutzt wird.

In einer monolithischen Anwendung liegen all diese Informationen auf einem System vor. Das macht die Analyse der Informationen überschaubar. In einer verteilten Anwendung ist das aber ungleich schwieriger. Hier entstehen diese Informationen auf vielen unterschiedlichen Systemen, und steht vor der Herausforderung diese Informationen in ihrer Gesamtheit auszuwerten, um Rückschlüsse über die Anwendung zu erhalten.

Ich halte das Protokollieren von verschiedenen Metriken als ein Element von zentraler Bedeutung für einen reibungslosen Betrieb. Aus diesem Grund sieht Tediga einen Mechanismus für die Protokollierung vor, der die Informationen zentral in einer einheitlichen Form erfasst und verschiedenen Werkzeugen zur Analyse und Auswertung zur Verfügung stellt.

Tedega nutzt für die die zentrale Erfassung von Logs Fluentd. Dieser sammelt alle zu Logs in einer einheitlichen Form ein, und speichert diese nach Bedarf in verschieden Backends. Von dort können die Logs Sie dann mit Werkzeugen wie *Elasticsearch* oder *Hadop* analysiert werden. Tediga stellt den Anwendungen Funktionen zum Protokollieren zur Verfügung, um sicher zu stellen, dass die Daten in einer einheitlichen Form geloggt werden, was eine Voraussetzung für spätere Auswertungen ist.

Weitere Informationen zum Logging finden Sie in *Logging*.

2.3 Sicherheit

Die folgenden Betrachtungen beschränken sich auf die Frage wie ein einzelner Microservice gegen nicht autorisierte Zugriffe geschützt werden kann.

Tediga verwendet zur Autorisierung ein Jason Web Token welches im Header einer Anfrage enthalten sein muss:

```
Authorization: Bearer <token>
```

Ohne gültiges JWT wird eine Anfrage nur dann autorisiert, wenn der Service für die entsprechende Anfrage keine Autorisierung erfordert.

Die Autorisierung von Anfragen wird an zentraler Stelle durch die *View* Komponente durchgeführt. Die Überprüfung findet für jede Anfrage einmalig beim Eingang in die View statt. Die Überprüfung der Autorisierung wird in zwei Schritten und an zwei Stellen durchgeführt:

1. Zunächst überprüft die View ganz grundlegende Dinge wie das Format, die Integrität des Tokens, oder ob dieses noch gültig ist. Sobald eine dieser ersten Überprüfungen fehlschlägt, wird die Anfrage abgewiesen.
2. Danach findet eine spezifische Autorisierung statt. Sie findet im Kontext der jeweiligen Domain und Funktion statt. Hierfür definiert die *Domain* eine spezielle Funktion, die alle Details der Autorisierung implemetiert. Diese Funktion wird bei der Registrierung der jeweiligen Methoden der Controller mit der Funktion *config_service_endpoint* als Parameter übergeben. Im Bild ist das die Funktion *check_authorisation*. Sie nimmt als Parameter das JWT entgegen auf dessen Basis die Überprüfung durchgeführt werden kann.

Nur wenn beide Überprüfungen erfolgreich sind, wird die Anfrage weiter bearbeitet. Eine erfolgreich überprüfte Anfrage wird nicht erneut überprüft. Alle weiteren Zugriff innerhalb des Service gelten als implizit autorisiert.

Unterabfragen an einen einen anderen Service, müssen erneut autorisiert werden. Hierzu sendet der Service bei der Anfrage das JWT zur Autorisierung einfach weiter.

2.4 Design Prinzipien

Tedega wurde vor dem Hintergrund der folgenden Prinzipien im Design umgesetzt. Diese Prinzipien finden sich sowohl in einem einzelnen Microservice, als auch in der Anwendung im Gesamten.

1. **API first.** Die API ist das wichtigste User Interface und die zentralen Schnittstelle für Konsumenten, und Entwickler unserer Dienste. Eine sauber definierte API ist die Voraussetzung für alle folgenden Prinzipien. Aus diesem Grund hat die Definition einer API eine hohe Bedeutung. Tedega verwendet zur Dokumentation der öffentliche API die Open API Spezifikation und Swagger
2. **KISS.** Keep it simple and stupid. Wir wollen Dinge so einfach wie möglich halten und nicht unnötig verkomplizieren. Die Funktion einer Komponente oder eines Service soll für ein breites Publikum einfach zu verstehen und anwendbar sein. Hierfür bevorzugen wir etablierte und weit verbreitete Technologien und Konzepte, um das Verständnis durch die verfügbare Dokumentation und Informationen zu vereinfachen.
3. **Loose Kopplung und hoher Zusammenhalt.** Tedega versucht zusammenhängende und gleichartige Funktionalität in Komponenten zu organisieren und diese Komponenten möglichst voneinander zu entkoppeln indem Abhängigkeiten vermieden werden (Single Responsibility Pronzip (SRP)). Das fördert das Verständnis der Funktion und vermeidet unerwünschte Seiteneffekte bei Änderungen einer Komponente.
4. **DRY.** Don't Repeat yourself. Tedega setzt bei der Implementation eines Service soweit möglich auf wiederverwendbare Komponenten und gemeinsam genutzte Bibliotheken. Das vermeidet Redundanzen durch Code-Duplizierung und reduziert so den Aufwand für die Wartung. DRY darf und wird verletzt werden, wenn sich der Code dadurch zu sehr verkompliziert und damit das höher eingestufte KISS Prinzip verletzen würde. Die potenziell entstehende Kopplung der Bibliotheken wird dabei bewusst in Kauf genommen, da der erwartete Vorteil bei der Wartung die Nachteile einer Kopplung überwiegen[1].

2.5 Beispiel

Im folgenden Beispiel wird ein einfacher Microservice implementiert. Dieser Service stellt unter dem Pfad */pings* eine einzige Methode zur Verfügung, die ohne weitere Parameter per einfache GET Anfrage aufgerufen werden kann.

Das Beispiel auf Github lässt sich wie folgt ausprobieren. Der Service lässt sich dann auf http://localhost:5000/ui ausprobieren:

```
git clone https://github.com/tedega/examples
cd examples
```

[1] Das gilt besonders vor dem Hintergrund des frühen Entwicklungsstadiums von Tedega und dem Umstand das die Entwicklung derzeit eine One-Man-Show ist.

```
python setup.py develop
python tedega_examples/app.py
```

Jeder Aufruf dieser Adresse wird mit der aktuellen Zeit in eine Tabelle in der Datenbank geschrieben. Der Server beantwortet jede Anfrage mit einer JSON-Datenstruktur, die das Datum der Ersten und Letzten Anfrage enthält, sowie die Anzahl aller bisherigen Anfragen und einen feste Zeichenkette.

Wichtig: Ein Microservice muss immer als Python Paket implementiert werden. Der hier beschriebene Code ist also nur ein Teil eines solchen Pakets. Informationen darüber was mindestens in einem solchen Paket enthalten sein muss finden sich im Python Packaging Tutorial. Weiterführende Informationen zur Paketierung finden sich im Python Packaging User Guide

---

Dieses Beispiel beinhaltet alle wichtigen Funktionen aus den verschiedenen Komponenten, die benötigt werden um einen Microservice zu bauen.

2.5.1 API

```
swagger: "2.0"
info:
  version: "1.0.0"
  title: Tedega Example
  description: Example of the bare minimum Swagger spec
paths:
  /pings:
    get:
      responses:
        200:
          description: Returns "Pong" on each Request.
          schema:
             $ref: '#/definitions/Pong'
definitions:
  Pong:
    type: object
    properties:
      data:
        type: string
        description: Value
        example: Pong
```

2.5.2 Service

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from tedega_share import (
    init_logger,
    get_logger,
    monitor_connectivity,
    monitor_system
)

from tedega_view import (
    create_application,
```

```
    config_view_endpoint
)

from tedega_storage.rdbms import (
    BaseItem,
    RDBMSStorageBase,
    init_storage,
    get_storage
)

########################################################################
#                                Model                                 #
########################################################################


class Ping(BaseItem, RDBMSStorageBase):
    __tablename__ = "pings"


########################################################################
#                              Controller                              #
########################################################################


@config_view_endpoint(path="/pings", method="GET", auth=None)
def ping():
    data = {}
    log = get_logger()
    with get_storage() as storage:

        factory = Ping.get_factory(storage)
        item = factory.create()
        storage.create(item)

        items = storage.read(Ping)
        data["total"] = len(items)
        data["data"] = [item.get_values() for item in items]
        log.info("Let's log something")
    return data


def build_app(servicename):
    # Define things we want to happen of application creation. We want:
    # 1. Initialise out fluent logger.
    # 2. Initialise the storage.
    # 3. Start the monitoring of out service to the "outside".
    # 4. Start the monitoring of the system every 10sec (CPU, RAM,DISK).
    run_on_init = [(init_logger, servicename),
                   (init_storage, None),
                   (monitor_connectivity, [("www.google.com", 80)]),
                   (monitor_system, 10)]
    application = create_application(servicename, run_on_init=run_on_init)
    return application

if __name__ == "__main__":
    application = build_app("tedega_examples")
    application.run()
```

KAPITEL 3

Komponenten

Wie in im Abschnitt *Architektur* beschrieben, besteht eine Microservice aus verschiedenen Komponenten. In diesem Abschnitt sollen diese Komponenten näher in ihrer Funktion und Aufgaben beschrieben werden. Dabei werden insbesondere die Schnittstellen beschrieben, die für die Implementation einer Domain genutzt werden können bzw. müssen.

3.1 Share

3.1.1 Logging

Tediga provides logging capabilities to applications to ensure that the data is logged in a consistent manner, which is a prerequisite for later evaluation.

Tedega uses Fluentd Logs for centralized logging. This collects all logs into a uniform form, and stores them in different backends as needed. From there, the logs can then be analyzed using tools such as Elasticsearch or Hadop.

The following categories are logged:

- **Requests**. All requests to the application are logged. The Includes the url, method (GET, POST, PUT ...) and possible parameters. They are marked in the category *REQUEST*.
- **Processing time**. Every request to a service will have the response time logged in milliseconds that a service needs to answer a request. The time adds up to the necessary steps for the answering be carried out. So also possible queries to others services. The category for the processing time is *PROCTIME*
- **Status response**. Each request logs the HTTP status of it answer. The category for the status is *RETURNCODE*
- **Reachability**. Periodically, each service is will check its connecivity to other hosts and services. Accessibility messages have the category *PING*.
- **Utilization RAM, CPU, memory**. We pick up at regular intervals information about the load of RAM, load and storage space in order to detect bottlenecks early. The corresponding category is *SYSTEM*.
- **More information**. In addition to the information described above, of course also any other information can be logged as needed. These should then be categorized with *CUSTOM*.

So that the messages are systematically evaluated in a central location all messages must be in a predefined format. Each Log message has a tag which Fluentd uses to route log messages can be used:

```
<HOST>. <SERVICE> [. <CONTAINER>]
```

The actual log message is saved in JSON format:

```
{
    "type": "INFO",
    "category": null,
    "correlation_id": null,
    "extra_key_depending_on_category": "value"
}
```

The following table provides information about the most important tags in log messages.

Section	Description
HOST	Name of the computer.
CONTAINER	Name Containers.
SERVICE	Name of the service
CATEGORY	Type of log message.
CORRELATION_ID	When a request first encounters a service, it generates a unique UUID, which is used in all subsequent queries to track related messages across different services. This information is optional because not all messages are generated in services or require such a UUID.
LEVEL	Indicates whether the message is an ERROR, a WARNING, an INFO, or a DEBUG output. The default for a message is INFO.

class tedega_share.logger.**Logger**(*logger*, *service*)

Wrapper around the Python logger to ensure a specific log format.

debug(*message*, *category=None*, *correlation_id=None*)

Write a debug message.

error(*message*, *category=None*, *correlation_id=None*)

Write a error message.

info(*message*, *category=None*, *correlation_id=None*)

Write a info message.

warning(*message*, *category=None*, *correlation_id=None*)

Write a warning message.

tedega_share.logger.**init_logger**(*service*, *host='fluentd'*, *port=24224*)

Will initialise a global *Logger* instance to log to fluentd.

Tag String used as tag for fluentd log routing.

Host Host where the fluentd is listening

Port Port where the fluentd is listening

tedega_share.logger.**get_logger**()

Will return the global *Logger* instance. Will raise an exception if the Logger is not initialised.

Rückgabe Logger instance

tedega_share.logger.**log_proctime**(*func*)

Decorator to log the processing time of the decorated method.

tedega_share.logger.**monitor_connectivity**(*hosts*, *interval=60*)
Continually check and log the connection to the list of given hosts in the given intervall in seconds.

Hosts List of tuples (hostname, port)

Interval Check will be executed every X seconds

3.2 View

tedega_view.registry.**config_view_endpoint**(*path*, *method*, *auth*)

tedega_view.server.**create_application**(*modulname*, *swagger_file='swagger.yaml'*, *run_on_init=None*)
Will create a connexion application for the given domain modul in *modulname*. The method will register all endpoints of the modul and make them available with the definden REST-API given in the *swagger_file*.

The funtion can optionally run a list of functions when the application is created. This can be used to start background processes like monitoring. The callable are give as a list of tuples. The first element in the tuple is the callable and the second element are the arguments used to call the callable.

Modulname String of the name of the domain modul/package.

Swagger_file Name of the Swagger config relativ to the given modul/package.

Run_on_init List of callable which are called after the application has been created.

Rückgabe Connexion application.

3.3 Storage

tedega_storage.rdbms.**init_storage**()

tedega_storage.rdbms.**get_storage**()

tedega_storage.rdbms.**scoped_session**()

class tedega_storage.rdbms.storage.**Storage**(*engine=None*)
Docstring for Storage.

KAPITEL 4

History

4.1 0.1.0 (2017-11-27)

- First release on PyPI.

KAPITEL 5

Literaturverweise

KAPITEL 6

Indices and tables

- search

Literaturverzeichnis

[Newmann2015] Sam Newman; Buildin Microservices; O'Reilly 2015

[DOP2016] Gene Kim, John Willis, Patrick Debois; The DevOps Handbook; IT Revolution Press 2016

[Ver2013] Vaughn Vernon; Implementing Domain-driven Design; Addison-Wesley 2013

Python-Modulindex

t

Stichwortverzeichnis

R

S

T

V

W