We Are Reactive

QAWARE SOFTWARE ENGINEERING

Cloud Computing: Übung

Kapitel 2: Programmiermodelle für die Cloud (Reative Programming)

Florian Lautenschlager, Dr. Josef Adersberger

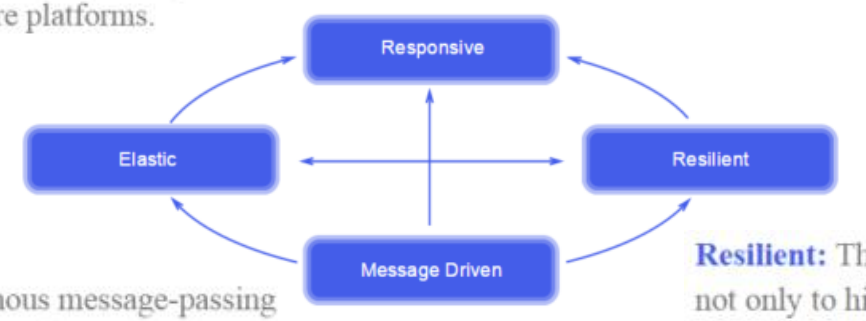
Das Reactive Manifesto

React to load

Elastic: The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

React to users

Responsive: The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.



React to events / messsages

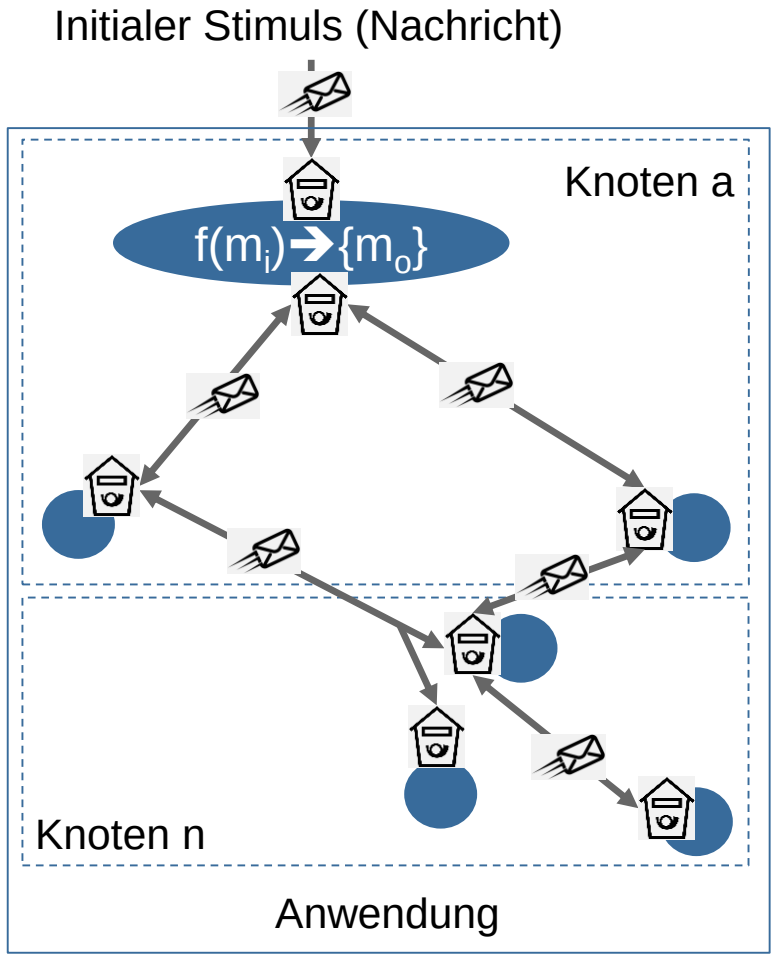
Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

Resilient: The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

React to failures

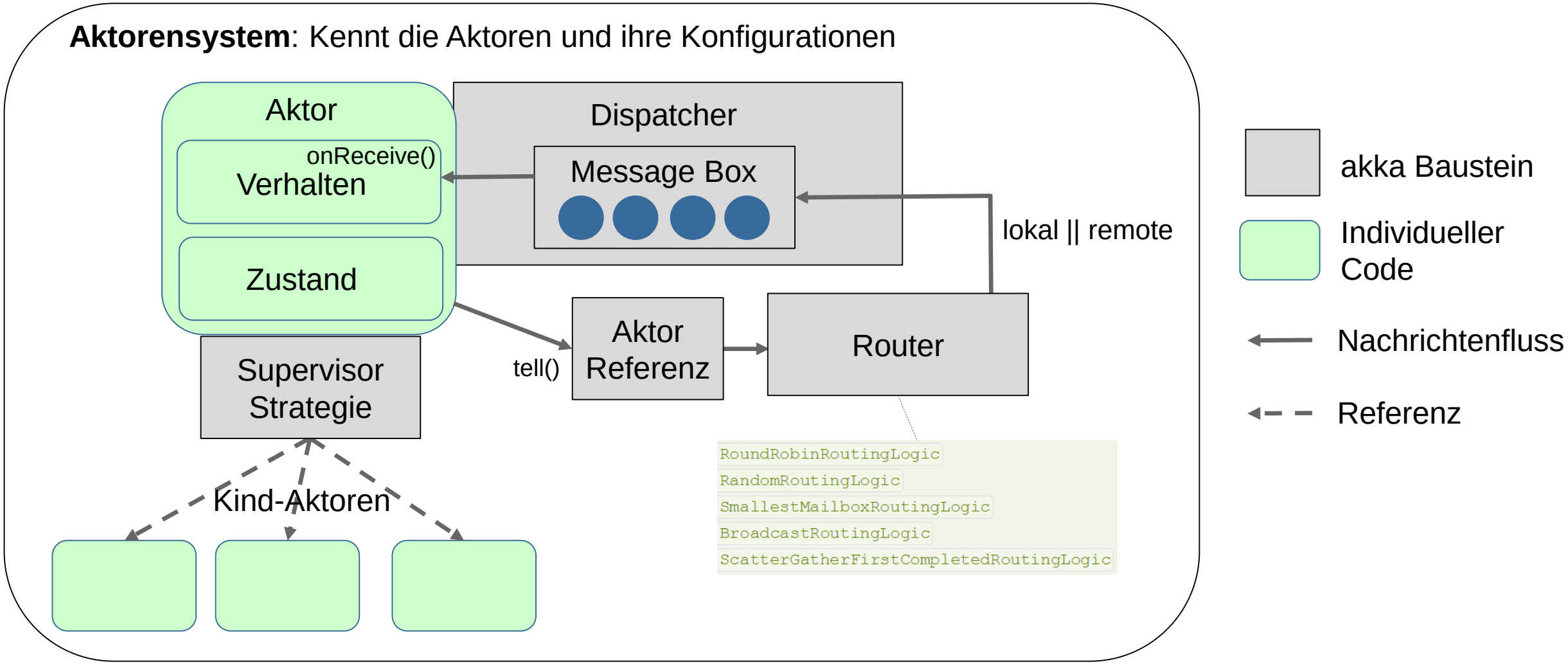
Functional Reactive Programming (FRP): Das Programmiermodell mit dem das Reactive Manifesto umgesetzt werden kann.

- **Dekomposition in Funktionen (auch Aktoren)**
  - funktionale Bausteine ohne gemeinsamen Zustand. Jede Funktion ändert nur ihren eigenen Zustand.
  - mit wiederaufsetzbarer / idempotenter Logik und abgetrennter Fehlerbehandlung (Supervisor)
- **Kommunikation zwischen den Funktionen über Nachrichten**
  - asynchron und nicht blockierend. Ein Funktion reagiert auf eine Antwort, wartet aber im Regelfall nicht auf sie.
  - Mailboxen vor jeder Funktion puffern Nachrichten (Queue mit n Producern und 1 Consumer)
  - Nachrichten sind das einzige Synchronisationsmittel / Mittel zum Austausch von Zustandsinformationen und sind unveränderbar
- **Elastischer Kommunikationskanal**
  - Effizient: Kanalportabilität (lokal, remote) und geringer Kanal-Overhead
  - Load Balancing möglich
  - Nachrichten werden zuverlässig zugestellt
  - Circuit-Breaker-Logik am Ausgangspunkt (Fail Fast & Reject)

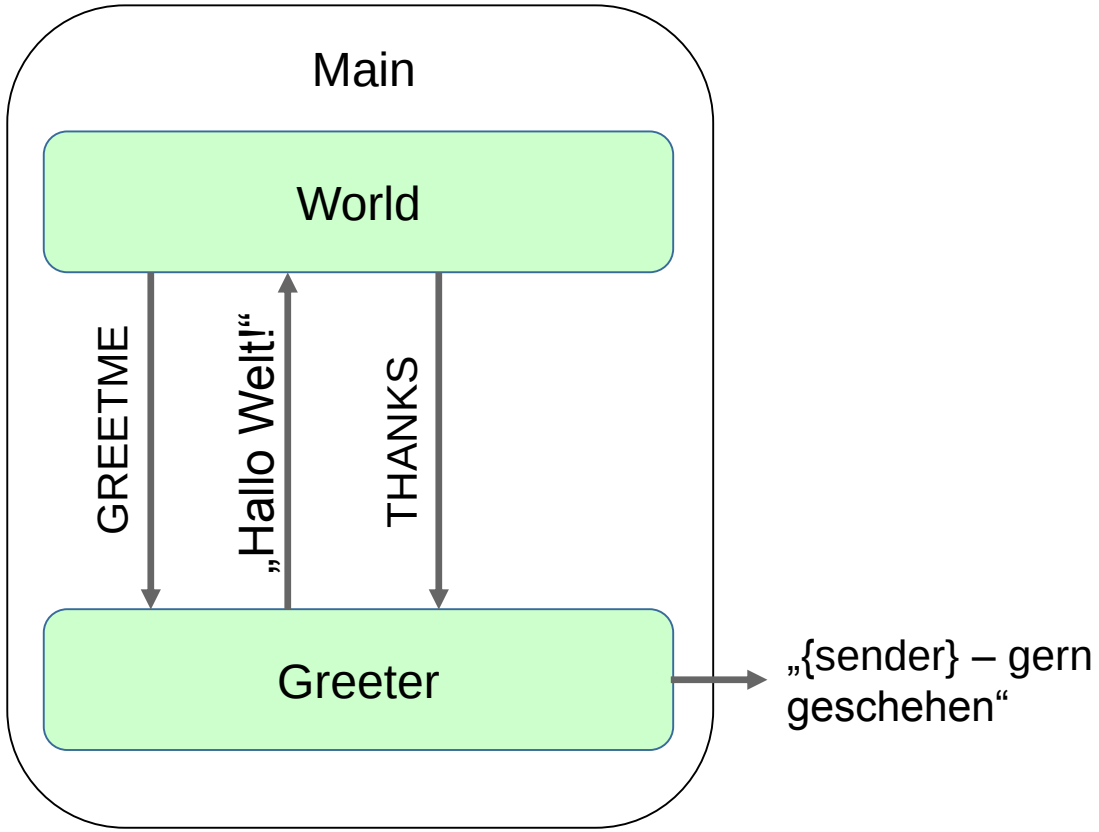


Die Grundkonzepte von akka

Konfiguration Aktor = Aktor-Klasse + Aktor-Name + Supervisor-Strategie + Dispatcher + Lokalität
Konfiguration Aktor Referenz = Aktor-Name + Router



Beispiel: Hello World



akka Hello World: Die Klasse *World*

```
public class World extends UntypedActor{

    @Override
    public void preStart() {
        ActorRef greeter = getContext().actorOf(Props.create(Greeter.class), "greeter");
        greeter.tell(Greeter.Msg.GREETME, getSelf());
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof String){
            System.out.println(message);
            getSender().tell(Greeter.Msg.THANKS, getSelf());
        } else {
            unhandled(message);
        }
    }

}
```

akka Hello World: Die Klasse *Greeter*

```
public class Greeter extends UntypedActor {

    public static enum Msg {
        GREETME,
        THANKS
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message == Msg.GREETME){
            getSender().tell("Hallo Welt!", getSelf());
        } else if (message == Msg.THANKS){
            System.out.println(getSender().toString() + " - gern geschehen");
        }
        else {
            unhandled(message);
        }
    }

}
```

akka Hello World: Die Klasse *Main*

```
public class Main {
    public static void main(String[] args) { akka.Main.main(new String[] {World.class.getName()}); }
}
```

Unsere Arbeiten an Zwitscher gehen nun weiter…

Zwitscher

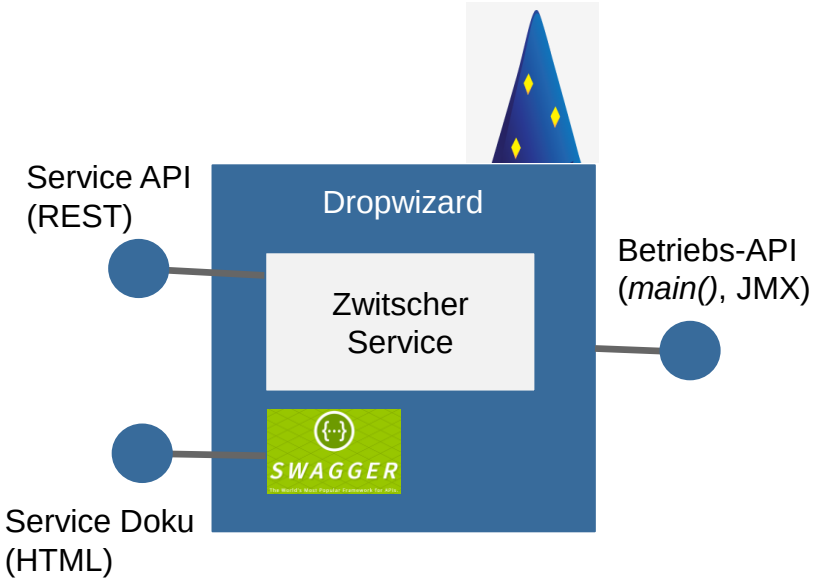
WIKIPEDIA
Die freie Enzyklopädie

FEEDZILLA

…

Schlüsselwörter

Was bisher geschah:



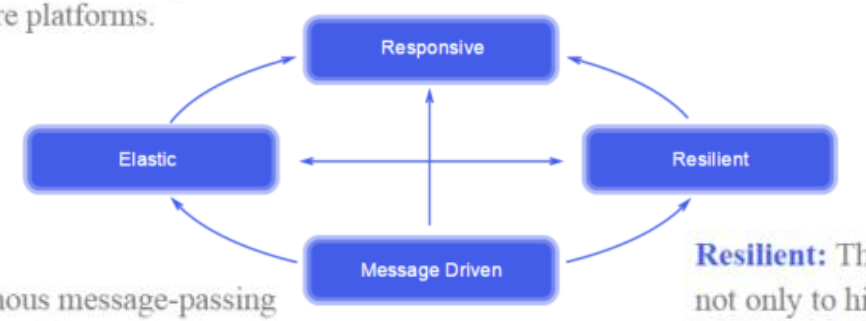
Das Reactive Manifesto

React to load

Elastic: The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

React to users

Responsive: The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.



React to events / messsages

Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

Resilient: The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

React to failures

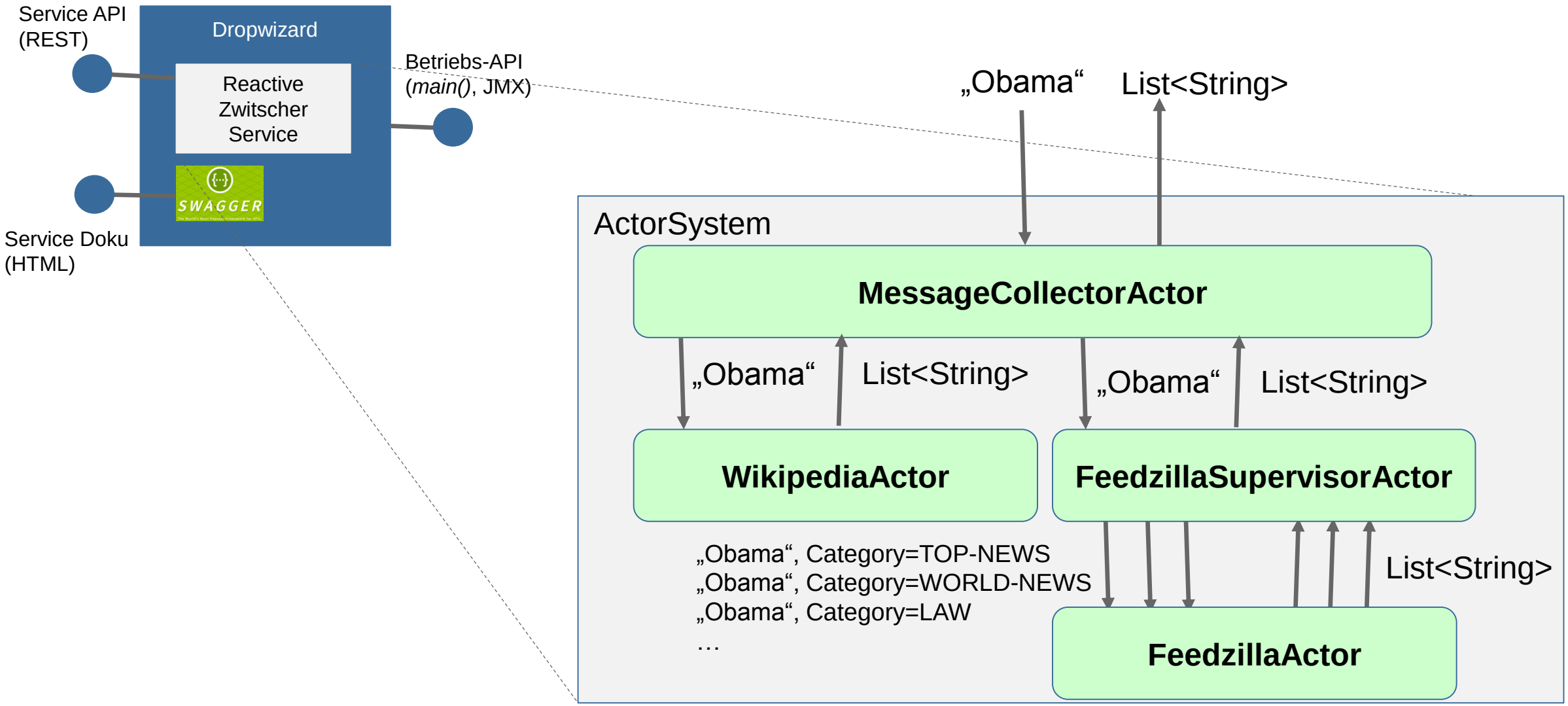
React to Users = React within 2 Seconds

Unser Problem: Der seqeuenzielle Zugriff auf Wikipedia und Feedzilla dauert zu lange.

```
Duration to collect Zwitschers: 2663 ms
```

```
Duration to collect Zwitschers (akka): 1079 ms
```

Das Ziel für heute:



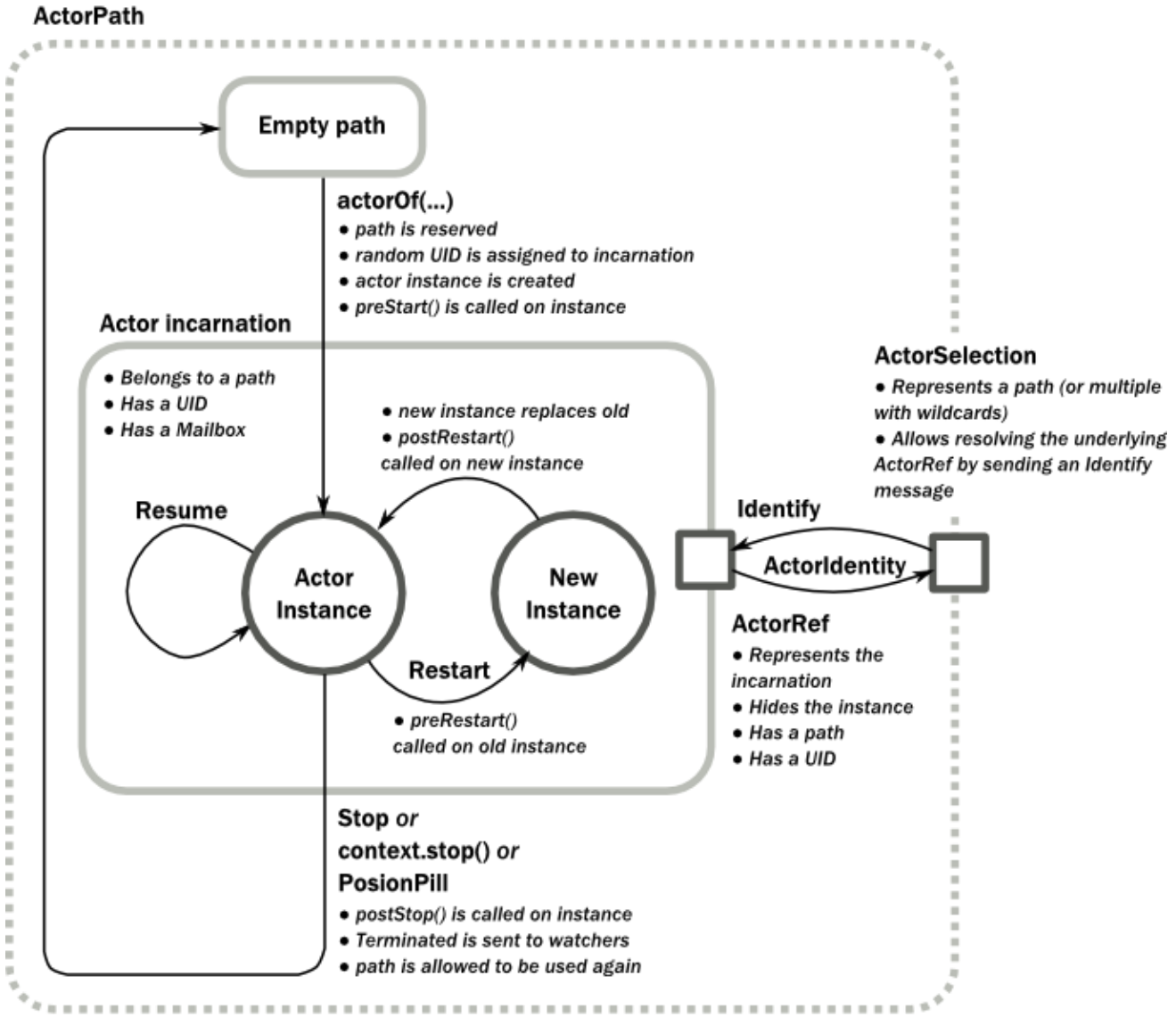
Der Quellcode in der Vorlage.


```
/**
 * Holt die News-Artikel in einer Kategorie auf Feedzilla
 * zu einem entsprechenden Suchbegriff.
 *
 * Es wird die REST-API von Newszilla genutzt:
 * <code>https://code.google.com/p/feedzilla-api/wiki/RestApi</code>
 *
 * Die Request-Struktur dafür sieht wie folgt aus:
 * <code>http://api.feedzilla.com/v1/categories/{category}/articles/search.json?q={term}</code>
 *
 * @param term Suchbegriff
 * @param category News-Kategorie in der gesucht werden soll
 * @return Liste der News-Titel entsprechend dem Suchbegriff.
 * Die Liste ist leer, wenn keine Artikel gefunden wurden.
 */
public List<String> getNewsFor(String term, NewsCategory category){
```

```
/**
 * Holt die Wikipedia-Artikeltitel zu einem Suchbegriff
 * Dabei wird ein Request nach folgendem Beispiel aufgebaut:
 * <code>http://en.wikipedia.org/w/api.php?action=opensearch&search={term}&limit=10&format=xml</code>
 *
 * @param term Suchbegriff
 * @return Liste der Artikeltitel entsprechend dem Suchbegriff.
 * Die Liste ist leer, wenn keine Artikel gefunden wurden.
 * Es werden maximal 25 Artikeltitel zurückgegeben.
 */
public List<String> getArticleTitlesFor(String term){
```

Bonusmaterial

Der Lebenszyklus eines akka Aktors



http://doc.akka.io/docs/akka/2.3.6/java/untyped-actors.html

API: *UntypedActor*

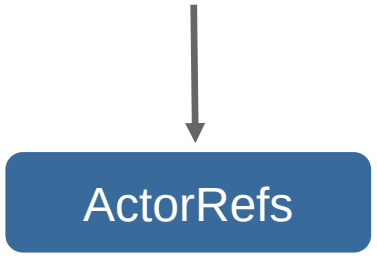
`getContext()` Returns this UntypedActor's UntypedActorContext The UntypedActorContext is not thread safe so do not expose it outside of the UntypedActor.
`getSelf()` Returns the ActorRef for this actor.
`getSender()` The reference sender Actor of the currently processed message.
`onReceive(java.lang.Object message)` To be implemented by concrete UntypedActor, this defines the behavior of the UntypedActor.
`postRestart(java.lang.Throwable reason)` User overridable callback: By default it calls `preStart()`.
`postStop()` User overridable callback.
`preRestart(java.lang.Throwable reason, scala.Option<java.lang.Object> message)` User overridable callback: '"By default it disposes of all children and then calls `postStop()`."' Is called on a crashed Actor right BEFORE it is restarted to allow clean up of resources before Actor is terminated.
`preStart()` User overridable callback.
`unhandled(java.lang.Object message)` Recommended convention is to call this method if the message isn't handled in `onReceive(java.lang.Object)` (e.g.

API: *ActorContext*

Actor suchen

Kind-Aktor erzeugen

actorFor(ActorPath path) Look-up an actor by path; if it does not exist, returns a reference to the dead-letter mailbox of the ActorSystem.
actorFor(scala.collection.Iterable<java.lang.String> path) Look-up an actor by applying the given path elements, starting from the current context, where ".." signifies the parent of an actor.
actorFor(java.lang.Iterable<java.lang.String> path) Java API: Look-up an actor by applying the given path elements, starting from the current context, where ".." signifies the parent of an actor.
actorFor(java.lang.String path) Look-up an actor by path represented as string.
actorOf(Props props) Create new actor as child of this context and give it an automatically generated name (currently similar to base64-encoded integer count, revers
actorOf(Props props, java.lang.String name) Create new actor as child of this context with the given name, which must not be null, empty or start with "$".



child(java.lang.String name) Get the child with the given name if it exists.
children() Returns all supervised children; this method returns a view (i.e.
dispatcher() Returns the dispatcher (MessageDispatcher) that is used for this Actor.
parent() Returns the supervising parent ActorRef.
props() Retrieve the Props which were used to create this actor.
receiveTimeout() Gets the current receive timeout.
self()
sender() Returns the sender 'ActorRef' of the current message.
setReceiveTimeout(scala.concurrent.duration.Duration timeout) Defines the inactivity timeout after which the sending of a ReceiveTimeout message is triggered.
system() The system that the actor belongs to.
unbecome() Reverts the Actor behavior to the previous one on the behavior stack.
unwatch(ActorRef subject) Unregisters this actor as Monitor for the provided ActorRef.
watch(ActorRef subject) Registers this actor as a Monitor for the provided ActorRef.
writeObject(java.io.ObjectOutputStream o) ActorContexts shouldn't be Serializable

API: *ActorRef*

`forward(java.lang.Object message, ActorContext context)` Forwards the message and passes the original sender actor as the sender.
`isTerminated()` Is the actor shut down? The contract is that if this method returns true, then it will never be false again.
`noSender()` Use this value as an argument to `tell(java.lang.Object, akka.actor.ActorRef)` if there is not actor to reply to
`path()` Returns the path for this actor (from this actor up to the root actor).
`tell(java.lang.Object msg, ActorRef sender)` Sends the specified message to the sender, i.e.

Das Aktorensystem

- Kommunikation von Außen mit dem Aktorensystem
  - Messaging:

Messages are sent to an Actor through one of the following methods.

- `tell` means “fire-and-forget”, e.g. send a message asynchronously and return immediately.
- `ask` sends a message asynchronously and returns a `Future` representing a possible reply.

akka.pattern.Patterns.ask

  - Inbox: Stellvertreter für einen Aktor. Sieht fast aus wie ein Aktor, kann aber bedenkenlos von Außen genutzt werden.