HANDS-ON TRAINING DISTRIBUTED SYSTEMS

Business Informatics, BSc.

Prof. Dr. Martin Schmollinger

Reutlingen University
School of Informatics
Alteburgstraße 150
72762 Reutlingen
https://www.inf.reutlingen-university.de/home/

Version 0.1 21.10.2021

Table of Content

Introduction

The following programming exercises help students to improve their theoretical knowledge about distributed systems. Currently, we use the following technology stack fort he exercises:

- JDK 11 (or higher)
- Gradle
- Spring Boot
- Visual Studio Code (only as a recommendation)

The exercises belong to 3 different categories:

1. Networking and Concurrency

   Build distributed systems and concurrent programs by using the network and process/thread capabilities of your system.

2. Microservice with REST API

   Build REST services using a middleware (currently Spring Boot). Besides the implementations of an RESTful API, the service is able to store data in a database system using the Java Persistence API (JPA) for the object-relational mapping.

3. Microservices Communication Styles

   In this exercise, we build a distributed system consisting of several services. In order to realize the overall system, the services must communicate with each other. In this scenario, we want to use different communication styles, like message passing or remote procedure calls.

In general, it is sufficient to run your distributed systems on localhost. If you really want to distribute your components (e.g. client and server), you have use their real IP addresses and the firewall of the nodes must allow the access to the respective ports.

1 Networking and Concurrency

The Java API provides classes to create and use UDP, TCP and http connections. Further, we learned the basics of the Java Thread API. The following exercises will use both capabilities to develop distributed and concurrent systems.

1.1 Iterative Time Server and a Simple Client

Goal: Create a Server that can be ask for the current time using UDP. Develop a simple client to test the server. Both, client and server are single-threaded.

Classes: `java.net.InetAddress, java.net.DatagramPacket, java.net.DatagramSocket, java.net.SocketException, java.util.Date, java.text.DateFormat`

The time server works in an endless loop (e.g. `while(true) {}`) and can only be interrupted by CTRL-C in the terminal where it was started or by using an OS tool (e.g. the task manager on Windows). The server listens (blocking method receive) on a DatagramSocket on a certain port. If a client sends a time request containing a style information via its socket, the receive method returns and the payload of the DatagramPacket (the style) can be accessed by its byte array.

The client has the choice to receive the date in three styles:

- FULL: full style (e.g. Sunday, October 24, 2021)
- MEDIUM: medium style (e.g. Oct 24, 2021)
- SHORT: short style (e.g. 10/24/21)

An easy way to encode the style in the client's request is to use a String (e.g. "FULL") and to convert it to a byte array by the getBytes() method of class String. This byte array can be used as payload of the DatagramPacket that will be send by the client.

The server can decode the byte array to a String by using the respective constructor of class String.

The server can then prepare the return value in dependence of the received style. For each request the server creates a new Date object that represents the current time, converts it into the desired String format (style), encodes the String to a byte array and sends it back in a DatagramPacket to the client.

Example for encoding a Date object to a byte array in style FULL in the server:

```
DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.FULL);
String dateString = dateFormat.format(new Date());
byte[] dateBytes = dateString.getBytes();
```

1.2 Guess Game

Goal: Implement a Guess Game Client and Server using TCP.

Guess Game: the server picks a random number between 0 and 9. The client has 3 attempts to guess the picked number. The game ends if the client has guessed the picked number (client wins!) or if the client has not guessed the picked number after the third try (server wins!).

The communication during a game between client and server is handle by a TCP connection. This is created by the client starting a new game. The connection ends if the game is over. In this case the server sends a final message to the client before is closes the connection.

In part a) of the exercise, we develop an iterative server (single-threaded). In part b), we improve to a concurrent server, where each game will be handled by a worker thread.

Important classes: java.net.Socket, java.net.ServerSocket, java.io.InputStream, java.io.OutputStream, java.lang.Thread.

a) Iterative Server and console client

The server can be started without any parameter. In an endless loop, it waits on his server socket for a client connection. When the client connects the server does the following things:

- Prepare the game by picking a random number between 0 and 9.
- Send status SERVER_READY (0) back to the client.
- Wait for first number to be transmitted by the client.

A client can be started by passing the hostname and the port of the server using the command line parameters (args[0]=hostname, args[1]=port). With these wo arguments a socket for communication with the server can be created by the client. After creating the socket, the client sends a `START_GAME` (encoded with 0) to the server and waits for the server's confirmation (SERVER_READY).

After receiving `SERVER_READY`, the client continuously asks his user for his guess and transmits it to the server. The server checks the received number and returns one of the following server codes to the client:

Server Code	Encoding
`CORRECT_GUESS`	1
`INCORRECT_GUESS`	2
`GAME_OVER`	9

Dependent on the server code, the client may print out that the user has won the game, may ask his user for another number or may print out that the game is lost because of too many attempts,

Hints:

- Use the class java.util.Scanner together with the input stream System.in to implement the user input from the console. In order to allow gradle to read from standard input, please add the following to the file build.gradle:

```
run {

    standardInput = System.in

}
```

- The codes or numbers of type int that are passed from the client to the server and vice versa can be transmitted directly using the basic input and output stream of the socket. There is no need to use higher-level classes from package java.io. as long as there values are in the range from 0 to 255.

b) Concurrent Server

The iterative server from part a) has one big disadvantage! It is not possible to handle multiple games in parallel. As long as one game is running, accesses from other clients will be blocked and queued. Not until the previous game has been finished, the next can be started.

Try it out! Start several clients and look at their behavior. What can be observed?

In order to improve the situation, write a concurrent server. The main idea is to handle each game by a separate worker thread (class GuessGameThread). Hence, the main thread works as a dispatcher thread that listens to the server socket and creates a worker thread for each client, i. e. for each game. While creating and starting a worker thread the socket representing a concrete client is passed to the thread so that the game can be handled.

1.3 Concurrent Search

Goal: Realize a concurrent program that stores the N numbers from 0 to N-1 into a List and searches a certain number n. Distribute the work evenly among p threads. Measure the (worst-case) running times for N=50.000.000 and p=1, 2, 4, 8, 16. Calculate the Speed-up S=Ts/Tp for each combination, where Ts is the running time for one thread and Tp the running time for p threads.

The program can be started by passing the three parameters N, n and p. The main thread creates a list of size N. The list stores random Integer values from 0 to N-1 in the list. Duplicates are allowed.

Further the main thread creates a result list that stores all indexes where number n was found. Be sure to avoid race conditions by synchronizing the access to the result list. You can do this with the `synchronized` keyword/block or by using the Collections class:

```
Collection<Integer> results = Collections.synchronizedCollection(new ArrayList<>());
```

Calculate the ranges for each of the p threads and start them. After that the main thread waits until all search threads have finished their work (join).

We measure the time in the main thread from starting the threads until joining the last thread.

After joining with all search threads, the main thread prints the list of indexes to the screen.

Speedup with ...	1 Thread	2 Threads	4 Threads	8 Threads	16Threads
N=50.000.000	1	?	?	?	?

Hints:

- Use class Random to generate random numbers between 0 and N.
- Measure the running times using time stamps provided by class java.lang.System (currentTimeMillis()).
- Since each thread has to check his complete range, there is no special worst-case scenario.
- In some cases N % p is not 0. In these cases, we accept that thread p has a larger range of N / p + N % p.
- Each thread uses linear search to find n in its range of the list.
- Think about making several measures for a N-p-combination to use the mean as a base for calculating the speed-up.
- When you interpret the running times, it is good to know that an optimal speed-up would be p.

1.4 Web Crawler – A Multithreaded Client

Goal: As a white hat hacker, we want to warn companies if they publish e-mail addresses on their web site that can be extracted automatically. These email addresses can be easily used by black hat hackers for spam or phishing attacks.

Hence, we realize a web crawler for collecting e-mail addresses by using http connections. The idea is to start the web crawler with the initial URL of a certain company or organization.

a) The initial URL is passed to the web crawler using the command line parameters (args). The web crawler reads the web site of the url and searches for e-mail addresses. The email addresses are stored in a set. After finishing the search, the web crawler prints the set of email addresses to the screen. Use the package java.util.regex to find links and e-mail addresses in Strings using regular expressions.

Example for searching an email addresses in a string with a simple regex:

```
Pattern email = Pattern.compile("[a-zA-Z\\.]{2,20}@[a-zA-Z\\.-]{2,20}\\.[a-z]{2}");
Matcher matcher = email.matcher(webSite.toString());
while (matcher.find()) {
     System.out.println(matcher.group());//Print email addresses to screen
 }
```

b) Currently, the web crawler reads only the web site of one URL. Improve the program, so that it additionally searches for links in the current web site so that they can be followed next. Just search for well-formed href attributes in the HTML files that can be expressed by regular expressions.

Store each link in a queue. The crawler terminates if the queue is empty, otherwise it pulls the next URL from the queue an starts the search for email addresses again. Avoid visiting the same web site twice (Remember lecture algorithms & data structures! This is breadth-first search (BFS)!).

Example for searching a links in a string with a simple regex:

```
Pattern link = Pattern.compile("href=\"([a-zA-Z\\./-:0-9]{5,25})\"");
matcher = link.matcher(webSite.toString());
while (matcher.find()) {
   String match = matcher.group(1);
…
}
```

c) The whole search should be executed by a thread that is spawned from the main thread of the application. While the search thread does its work, the main thread listens to user input. The user may interrupt the search thread. The program may terminate in two ways. Either the user interrupts the search, or the search thread has finished his work. In both cases before exiting, the main thread should print all collected emails to the screen.

For this approach, you're free to realize a simple console-based client or a multithreaded JavaFX application (remember FP in your early semesters? ;-) with a text field for URL input and a start and stop button.

1.5 More ideas

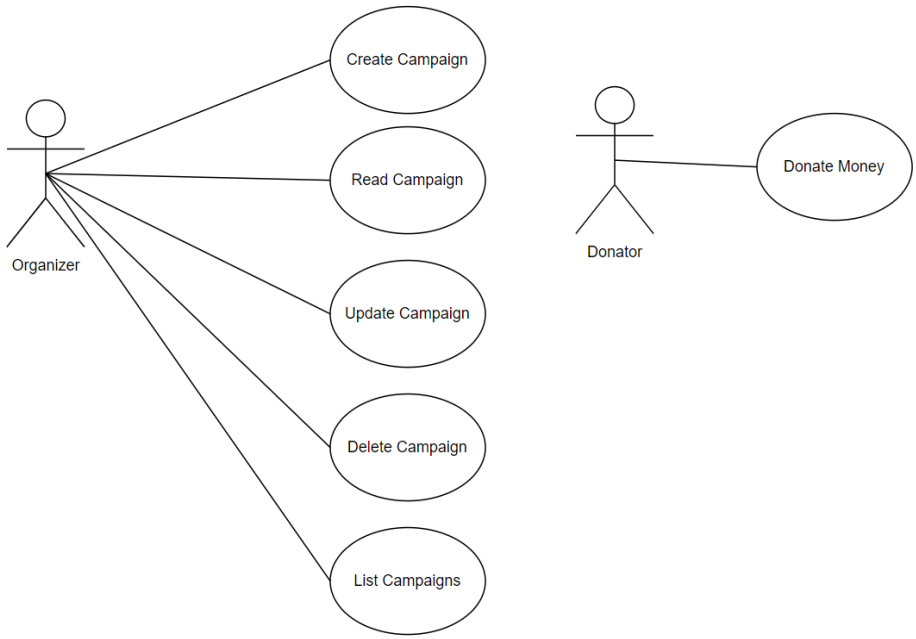
Pool of Idea: o Iterative vs. concurrent TCP server. How much requests are necessary until a denial of service (DoS)?

2 Microservice with REST API

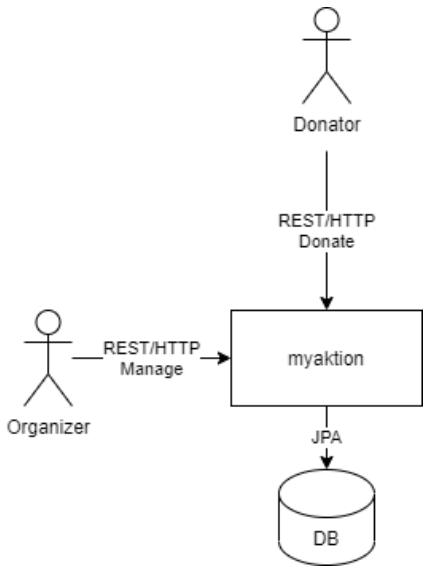
In this part of the hands-on training, we want to learn how to develop a REST service with Spring Boot. The service myaktion supports fund raisers in their work to collect donations for campaigns online.

Use Cases

- A fund raiser is the organizer of a campaign.
- An organizer can create, read, update and delete campaigns.
- An organizer has only access to its own campaigns.
- Each user may be a donator.
- A donator can donate to arbitrary campaigns.



We will develop the service step-by-step:

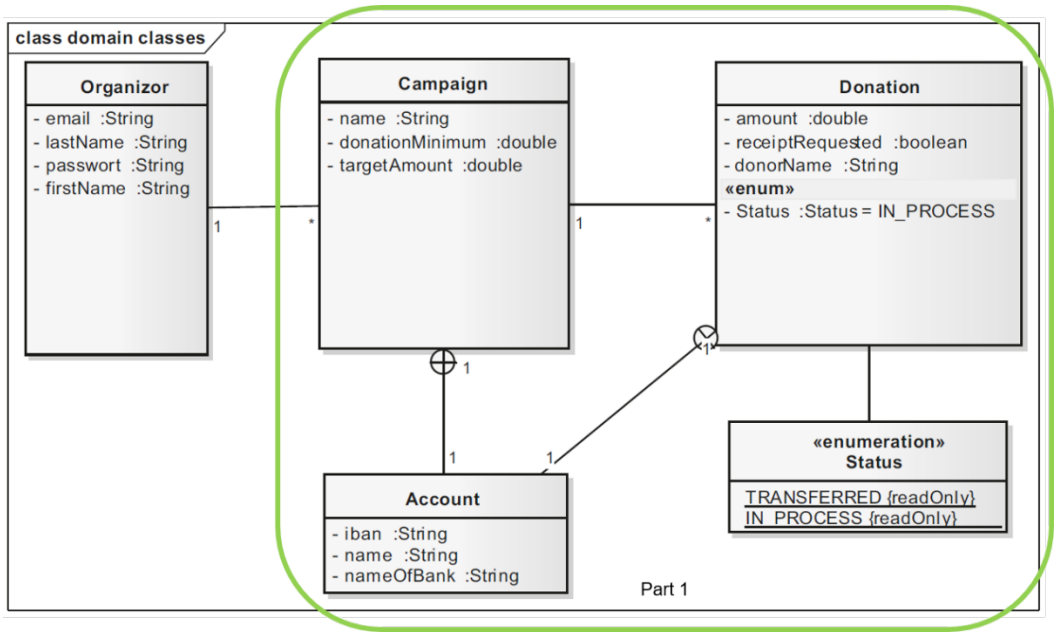


2.1 Domain Model

Goal: Create a new Spring Boot project myaktion-boot with Spring Initializr. Develop the domain model of the service according to the descriptions below.

Create a new Spring Boot project with the dependencies web, data-jpa and h2. Configure the h2 database as shown in the lecture in the application.properties file. Realize the following domain model (only the entities in the green frame!). The associations must be implemented as follows:

- Campaign <-> Donation (bidirectional, One-to-Many/Many-to-One, Cascade: ALL, Fetch-Type: EAGER).
- Campaign -> Account (embedded).
- Donation -> Account (embedded)
- Account itself is Embeddable.
- Status can be realized as an inner enum of class Donation, and is therefore not a JPA Entity.



Attention:

Embedding the entity Account into entity Campaign leads the a name conflict (Attribute name exists in both classes). Use the annotations @AttributeOverrides and @AttributeOverride to solve the conflict.

See: https://docs.jboss.org/hibernate/jpa/2.2/api/javax/persistence/AttributeOverrides.html.

2.2 JPA Repositories, Service classes and Sample Data

Goal: Complete the domain layer by adding JPA repository interfaces and add the service layer to the project. Fill the database with sample data.

Develop the two JPA repository interfaces for classes Campaign and Donation. Add them to package `domain`.

Create a new package `services` and add the service classes CampaignService.java and DonationService.java.

These two classes need a first bunch of methods for creating and retrieving data from the database.

- CampaignService:

  ```
  public Campaign addCampaign(Campaign campaign)
  public List<Campaign> getCampaigns()
  ```

- DonationService:

  ```
  public Donation addDonation(Donation donation, Long campaignId)
  ```

  This method must throw a custom `CampaignNotFoundException` (subclass of `RuntimeException`) if the campaignId is not valid (i. e. we do not find a campaign with this Id in the database). Create and use a separate package `exceptions` for exeception classes.

After finishing the steps above, we want to write sample data to our database using the methods of our service classes.

1) Create a new campaign object, add a new `Donation` object and save the `Campaign` object to the database by using your service class (`CampaignService`, `addCampaign`). Use h2 console to check that the data was stored as expected.
2) Create another `Donation` object for the campaign and save it to the database using the respective service class (`DonationService`, `addDonation`). Use h2 console to check that the data was stored as expected.
3) Read all campaigns from the database (`CampaignService`, `getCampaigns`) and log them to the screen (`System.out`).

Hints:

- Inject (`@Autowired`) the service classes into the class annotated with `@SpringBootApplication`.
- Inject the repositories into the service classes.
- Use a `CommandLineRunner` bean in the SpringBootApplication class that uses the injected service classes to write and read the sample data to and from the database. Implement the desired behavior as described above.

2.3 Add Logging

Goal: Add a Logger to the service classes and the SpringBootApplication class. Use the logger to make the service more verbose.

Set the logging level (TRACE, DEBUG, INFO, WARN, ERROR) in the applications.properties file for the whole package (here `io.ds`):

```
#logging
logging.level.io.ds=INFO
```

Add several logging messages to your classes with different logging levels. E.g.:

Class	Log level	Message
Your SpringBootApplication class	INFO	„Adding Sample data to DB“
	DEBUG	“Add campaign to DB”, “Add donation to campaign with id”
	DEBUG	“Read all campaigns”
CampaignService	TRACE	Log detailed Informations about the Campaign object (toString).
DonationService	TRACE	Log detailed Informations about the Donation object (toString).

Hint: Change the logging level in application.properties and verify the behavior.

2.4 REST API

Goal: Realize a REST API (Type 2) for the service

Implement the following REST API. So far, only a few of the resource accesses can be realized with our service classes. We have to adapt and complete the service classes to the need of the controller implementations.

http verb	URI	Descripion	RestController
GET	/campaigns	Returns the list of campaigns	CampaignController
POST	/campaigns	Receives the data of a campaign in the request body and adds it to the database.	CampaignController
GET	/campaigns/{id}	Returns campaign with Id id	CampaignController
PUT	/campaigns/{id}	Updates campaign id with data from the request body.	CampaignController
DELETE	/campaigns/{id}	Deletes campaign with Id id	CampaignController
POST	/campaigns/{id}/donations	Adds a donation to the campaign with Id id.	DonationController

Hints:

- Use `@JsonIgnore` for attribute `campaign` in class `Donation` to avoid endless cyclic JSON serialization!
- Use @JsonProperty(access=Access.READ_ONLY) for fields that are not allowed to set by REST clients (e.g. IDs).
- Test your implementation using curl and swagger-ui (add corresponding dependency to your project!)
- Add the following line to file application.properties to activate formatted JSON output: `spring.jackson.serialization.indent-output= true`
- Add a file README.md (markdown format) to the root folder of your project and document the API and the curl commands.

2.5 Bean Validation

Goal: The Bean Validation (BV) API provides annotations to define value contraints for the properties of JPA entities. By using BV, we want to guarantee that the data stored in the DB is valid and consistent. JPA checks the constraints automatically.

Add the following dependency to your gradle.build file:

```
implementation('org.springframework.boot:spring-boot-starter-validation')
```

Realize the contraints defined in the following tables. The following annotations are needed for the implementation and belong to `package javax.validation.contraints`:

`@NotNull`, `@Size`, `@DecimalMin`, `@Pattern`, `@Valid` (for associations and embedded classes)

More details can be found in the documentation:

https://docs.jboss.org/hibernate/beanvalidation/spec/2.0/api/

Contraints for class *Campaign*

Attributname	**Bedingung**	**Benutzermeldung**
Name	Min. 4 characters und max. 30 characters	Length of campaign name must be at least 4 and at most 30.
Donation minimum	>=1	The amount of the donation must be at least 1.
Target amount	>=10	The target amount of the campaign must be at most 10 Euro.

Contraints for class *Donation*

Attributname	**Bedingung**	**Benutzermeldung**
Donor name	Min. 5 characters und max. 40 characters.	Length of donor name must be at least 5 and at most 40.
amount	>=1	The amount of the donation must be at least 1
State	Not null	-
Quittung	Nicht null	-

Constraints for class *Account*

Attributname	**Bedingung**	**Benutzermeldung**
Name	Min. 5 and max. 60 characters	Length of Name of account owner must be at least 5 and at most 60.
Name of Bank	Min.4 Zeichen und max. 40 Zeichen	Length of name of bank must be at least 4 and at most 40.
IBAN	Two big letters, two digits and then 12 to 30 characters.	An IBAN starts with to big letters followed by two digits and 12 to 30 alphanumeric characters.

Test your contraints by trying to write invalid data into your database!

2.6 Controller Advices

Goal: Add controller advices to your project. Provide internal errors of the service with appropriate semantics for the client.

Realize a controller advice class for two Exceptions:

- ConstraintViolationException
  Something is wrong with the data of the request, In this case, we return an HTTP status code 400 (BAD REQUEST),

- CampaignNotFoundException
  This is a custom Exception that has to be developed first. It is thrown in case there is no campaign with the respective id stored in our database. In this case, we return an HTTP status code 404 (NOT FOUND).

In both cases the message of the exception should be embedded in response body of the request.

2.7 Extra Mile: Refactor REST API

Goal: Refactor REST API; avoid eager loading.

Currently, we use eager loading between the entities Campaign and Donation. Loading all Campaigns always means to load the whole database! That's not efficient!

We want to refactor the API as follows:

- Remove the fetch type in the association between Campaign and Donation (default is LAZY).
- Add `@JsonIgnore` to list of donations in class Campaign.
- Add an attribute `amountDonatedSoFar` to class Campaign. Set this attribute transient (@TRANSIENT), that means it will not be stored in the database. Instead, the idea is calculated and set the before transmitting the Campaign.
- Add a new method getAmountDonatedSoFar to interface CampaignRepository. Use @Query to add the SQL-Statement for calculating the amount.
- Change the Implementation of the REST API as follows (red).

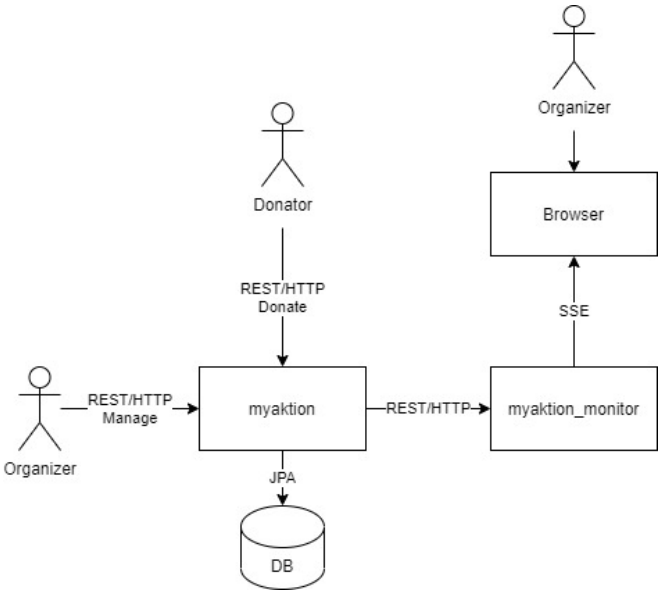
http verb	URI	Descritpion	RestController
GET	/campaigns	Returns the list of campaigns. Contains no donations! Only the amount donated so far is transmitted!	CampaignController
POST	/campaigns	Receives the data of a campaign in the request body and adds it to the database.	CampaignController
GET	/campaigns/{id}	Returns campaign with Id id. Contains no donations! Only the amount donated so far is transmitted!	CampaignController
PUT	/campaigns/{id}	Updates campaign id with data from the request body.	CampaignController
DELETE	/campaigns/{id}	Deletes campaign with Id id	CampaignController
POST	/campaigns/{id}/donations	Adds a donation to the campaign with Id id.	DonationController
GET	/campaigns/{id}/donations	Returns the list of donations of campaign with Id id.	DonationController

Test the API and update your documentation in README.md!

3 Microservice Communication Styles

In the last part of our hands-on training, we focus on the implementation of different communication styles in a microservices architecture.

3.1 Monitoring Service



Develop a second service with Spring Boot called “myaktion-monitor”. This service provides a simple web page showing all new donations. In this sense, the web page realizes a “Live Ticker” for donations. As soon as our service myaktion receives a new donation via its REST API, it also forwards the information to the new service myaktion-monitor.

We want to realize the communication between the two services in a synchronous, blocking manner. Therefore, the new monitoring service must offer a REST API for transmitting a single donation (POST) that can be used by service myaktion. After receiving a donation from service myaktion, the new monitoring service informs the connected browsers using Server-Sent-Events (SSE).

It is sufficient to send a reduced donation object (Data Transfer Object - DTO) containing the following informations:

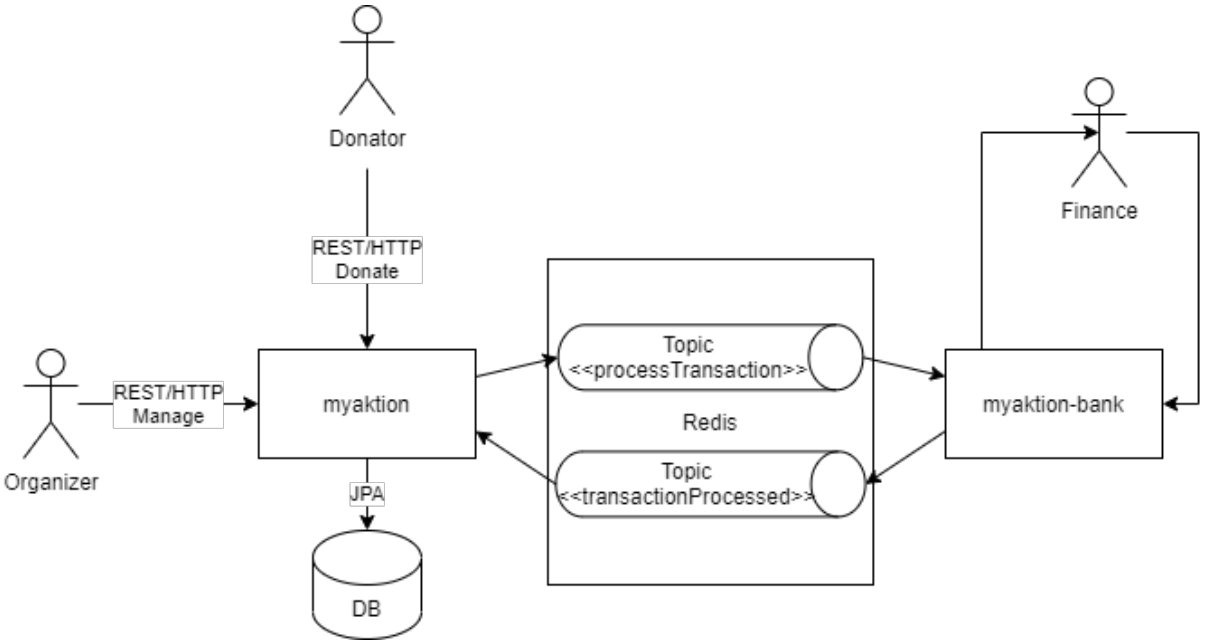
- Donor name
- Amount
- Id of campaign

The new service myaktion-monitor uses port 8081. This can be defined by the following entry in the file `application.properties`:

```
server.port=8081
```

You will find a more detailed step-by-step manual in the slides.

3.2 Bank Transaction Service



Develop a third service “myaktion-bank” that provides a user of the finance department with a bank transaction for each donation. Again, as soon as service myaktion receives a donation, it forwards the related bank transaction to the new service. The bank transaction has to be executed by the user in a separate external bank system. Therefore, the execution may last from minutes to several days. When the transaction was executed, the users informs myaktion-bank, and myaktion-bank informs myaktion. Then service myaktion changes the state of the corresponding donation in the database.

As a consequence of this scenario, we implement the communication in an asynchronous, non-blocking manner. We want to use Redis a message broker. We define one topic for each communication direction.

In the exercise, we simulate the bank transaction by going to sleep for 5 seconds.

You will find a more detailed step-by-step manual in the slides.