Lab 2: Python & GitHub Flow

Python: TicTacToe

GitHub Flow

GitHub flow is a lightweight, branch-based workflow.

Following the GitHub Flow

Create a Branch

Create a branch in your repository. A short descriptive branch name enables your collaborators to see the ongoing work at a glance.

By creating a branch you create a space to work without affecting the default branch and you give collaborators a chance to review your work.

Make Changes

On your branch make any desired changes to the repository. If you make a mistake you can revert your changes or push additional changes to fix the mistake.

Commit and push your changes to your branch. Each commit must have a message to help you and future contributors to understand what changes the commit contains.

Ideally each commit contains an isolated and complete change.
This makes it easy to revert your changes if you decide to take a different approach.
For example if you want to rename a variable and add some tests, put the variable renaming in one commit and the tests in another commit.

By committing and pushing your changes you back up your work to remote storage. This means that you can access your work from any device and your collaborators can see your work.

Create a Pull Request

Create a pull request to ask collaborators for feedback on your changes.
Pull requests review is so valuable that some repositories require an approving review before pull requests can be merged.
If you want early feedback or advice before you complete your changes you can mark your pull requests as a draft.

When creating a pull request a summary of the changes and what problem have been solved must be included.

Address Review Comments

Reviewers should leave questions, comments, and suggestion. Reviewers can comment on the whole pull requests or add comments to specific lines or files.

You can continue to commit and push changes in response to the reviews, your pull request will update automatically.

Merge your Pull Request

Once your pull request is approved, merge your pull request.
This will automatically merge your branch so that your changes appear on the default branch.
GitHub retains the history of comments and commits in the pull request to help future contributors understand your changes.

GitHub will tell you if your pull request has conflicts that must be resolved before merging.

Delete your Branch

After you merge your pull request and delete your branch.
This indicates that the work on the branch is complete and prevents you or others from accidentally using old branches.

Don't worry about loosing information. Your pull request and commit history will not be deleted.
You can always restore your deleted branch or revert your pull request if needed.

Lab 3: EIP with Apache Camel

Camel is an Open Source integration framework that empowers you to quickly and easily integrate various systems consuming or producing data.

Why Camel?

Based on Enterprise Integration Patterns.
Camel supports most of the Enterprise Integration Patterns, and newer integration patterns from microservice architectures to help you solve your integration problem by applying best practices out of the box.

Camel Routes

In Apache Camel, a route is a set of processing steps that are applied to a message as it travels from a source to a destination. A route typically consists of a series of processing steps that are connected in a linear sequence.

A Camel route is where the integration flow is defined.
For example, you can write a Camel route to specify how two systems can be integrated. You can also specify how the data can be manipulated, routed, or mediated between the systems.

The routes are typically defined using a simple, declarative syntax that is easy to read and understand.

Lab Exercise

- **World Songs Service:** offers a csv file containing the top 100 songs for each year
- **Top 20 Songs Service:** list the top 20 artists with most songs on billboard

You need to integrate the the World Songs Service with the Top 20 Songs Service through an Apache Camel Route, exploiting suitable EIPs.

Steps:

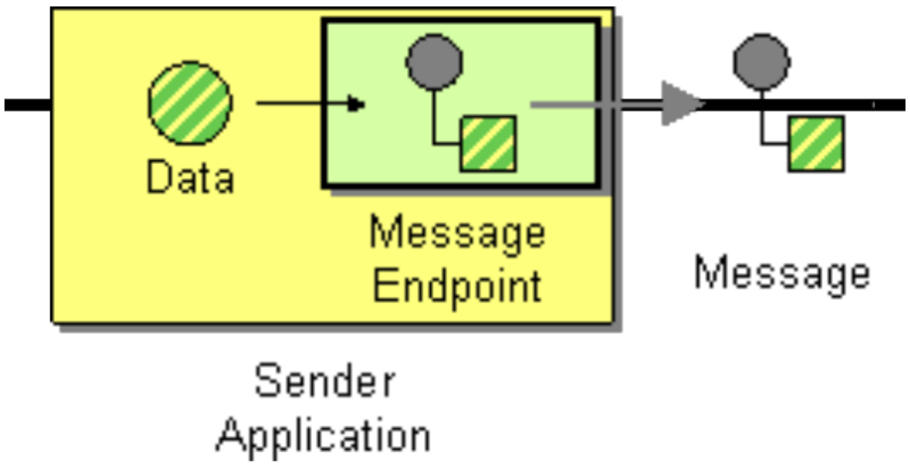
- download the skeleton project "music" from moodle
- use the given 14 cards to compose your Apache Camel Route
- code your route in the `configure()` method of the class `MyRouteBuilder`.

Solution

The cards are here displayed already in the right order and the code is commented step by step.

The idea (the flow of the data between the two services) is:
TODO.

Card 1



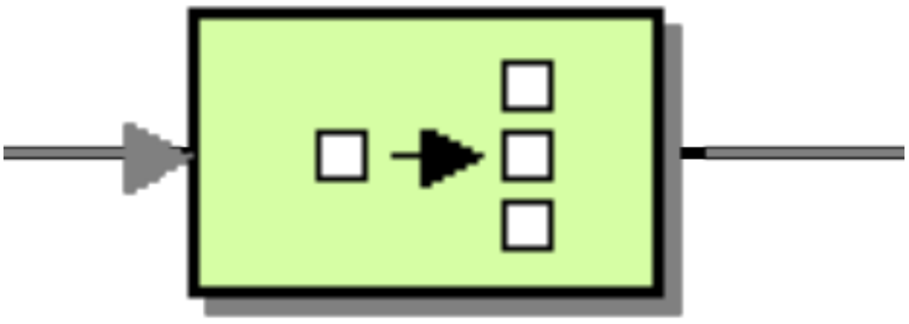
Every Camel route starts from an Endpoint as the input (source) to the route.
The From EIP is the input.

```
from("file:" + BASE_PATH + "?noop=true&idempotent=true")
```

What comes after `file:` is an URI: `file:directoryName[?options]`

- `?`: an optional query component preceded by a question mark (?), consisting of a query string of non-hierarchical data. Its syntax is not well defined, but by convention is most often a sequence of attribute–value pairs separated by a delimiter.
- `noop`: (consumer) if true, the file is not moved or deleted in any way. This option is good for read-only data. If `noop=true`, Camel will set `idempotent=true` as well, to avoid consuming the same files over and over again.
- `idempotent:` (filter) option to use the Idempotent Consumer EIP pattern to let Camel skip already processed files.

Card 2



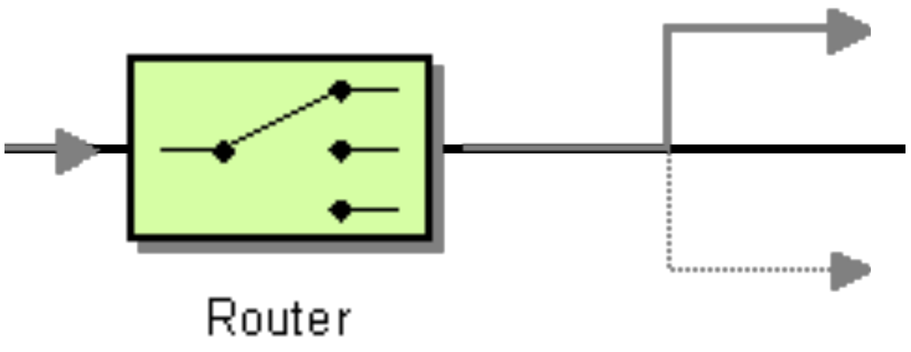
Splitter

How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?
Use a Splitter to break out the composite message into a series of individual messages, each containing data related to one item.

```
.split(body().tokenize("\n"));
```

- `body()`: split the message body
- `tokenize():` tokenize text documents using a specified delimiter pattern

Card 3



The Message Router from the EIP patterns allows you to consume from an input destination, evaluate some predicate then choose the right output destination.

In Camel the Message Router can be archived in different ways such as:

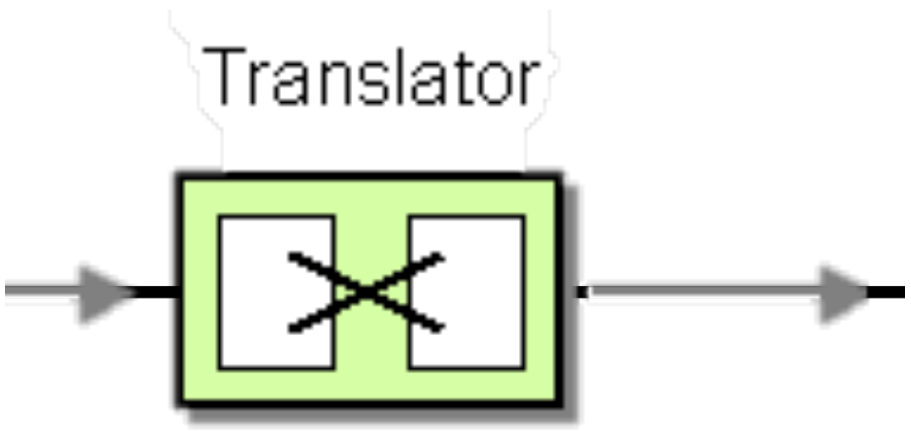
- **Content Based Router:** evaluate and choose the output destination.
- **Routing Slip and Dynamic Router EIPs:** can also be used for choosing which destination to route messages.

The Content Based Router is recommended to use when you have multiple predicates to evaluate where to send the message.
The Routing Slip and Dynamic Router are arguably more advanced where you do not use predicates to choose where to route the message, but use an expression to choose where the message should go.

```
.choice().when(simple(${exchangeProperty.CamelSplitIndex} > 0)) // skip
headers
```

- `choice().when()` : sets the when nodes
- `simple()` : The Camel Simple language is great to use with the Choice EIP when routing is based on the content of the message, such as checking message headers.

Card 4



The Message Translator can be done in different ways in Camel:

- Using Transform or Set Body in the DSL
- Calling a Processor or bean to perform the transformation
- Using template-based Components, with the template being the source for how the message is translated
- Messages can also be transformed using Data Format to marshal and unmarshal messages in different encodings

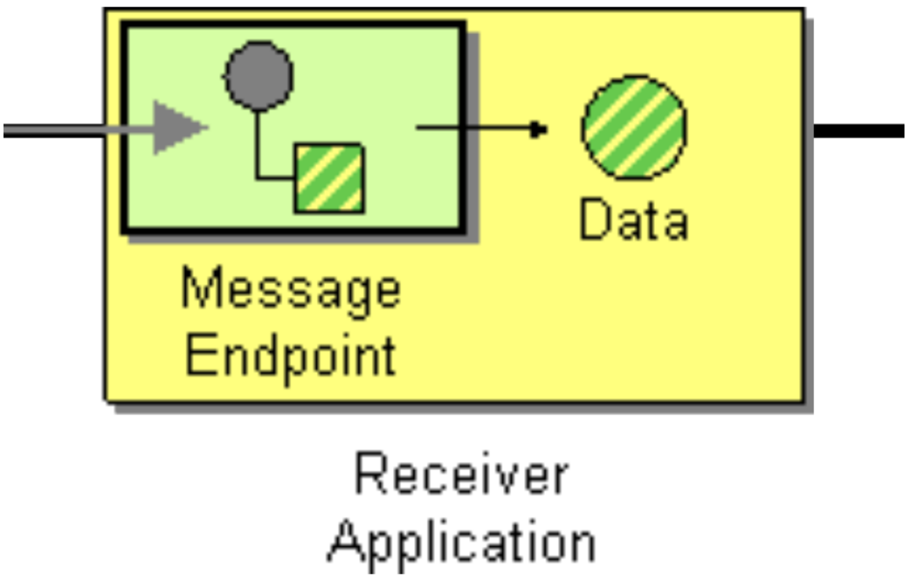
The Marshal and Unmarshal EIPs are used for Message Transformation.
So in other words the Marshal and Unmarshal EIPs are used with Data Formats.

- **Marshal** - Transforms the message body (such as Java object) into a binary or textual format, ready to be wired over the network.
- **Unmarshal** - Transforms data in some binary or textual format (such as received over the network) into a Java object; or some other representation according to the data format being used.

```
unmarshal().bindy(BindyType.csv, SongRecord.class)
```

- `bindy()` : The goal of this component is to allow the parsing/binding of non-structured data (or to be more precise non-XML data) to/from Java Beans that have binding mappings defined with annotations. Using Bindy, you can bind data from sources such as CSV records, Fixed-length records, or almost any other non-structured data to one or many Plain Old Java Object (POJO).

Card 5



```
.to("seda:aggregate")
```

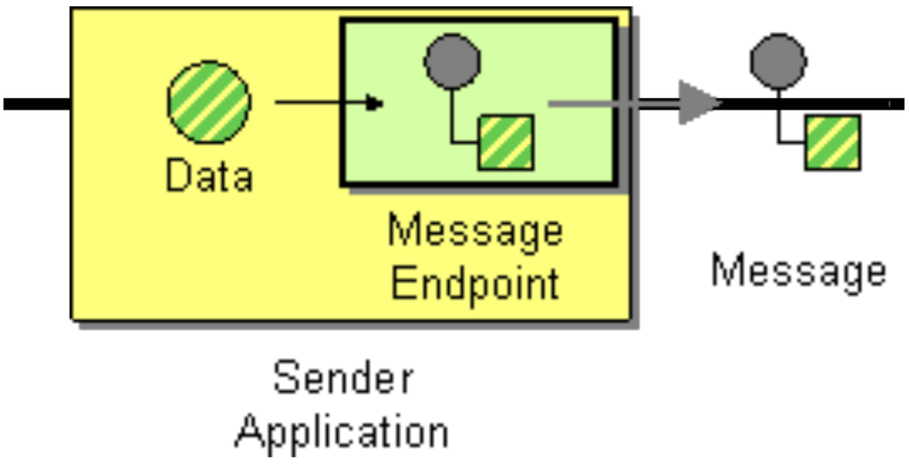
Connect an application to a messaging channel using a Message Endpoint, a client of the messaging system that the application can then use to send or receive messages.

- `seda`: The SEDA component provides asynchronous SEDA behavior, so that messages are exchanged on a blocking queue and consumers are invoked in a separate thread from the producer.
- `aggregate`: The Aggregator from the EIP patterns allows you to combine a number of messages together into a single message.

First Half

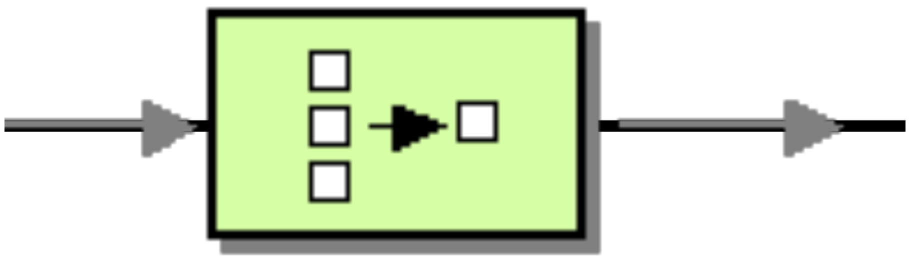
```
public void configure(){
        from("file:" + BASE_PATH + "?noop=true&idempotent=true")
        .split(body().tokenize("\n"))
        .choice().when(simple("${exchangeProperty.CamelSplitIndex} > 0"))
// skip headers
        .unmarshal().bindy(BindyType.csv, SongRecord.class)
        .to("seda:aggregate");
}
```

Second Half



```
from("seda:aggregate?concurrentConsumers=10")
```

```
.bean(MyAggregationStrategy.class, "setArtistHeader")
```


Aggregator

```
.aggregate(new MyAggregationStrategy()).header("artist")
```

```
.completionPredicate(header("CamelSplitComplete").isEqualTo(true))
```

```
.process(new MyProcessor());
```

Solution

```
public void configure(){
        from("file:" + BASE_PATH + "?noop=true&idempotent=true")
        .split(body().tokenize("\n"))
        .choice().when(simple("${exchangeProperty.CamelSplitIndex} > 0"))
// skip headers
        .unmarshal().bindy(BindyType.csv, SongRecord.class)
        .to("seda:aggregate");

        from("seda:aggregate?concurrentConsumers=10")
        .bean(MyAggregationStrategy.class, "setArtistHeader")
        .aggregate(new MyAggregationStrategy()).header("artist")
        .completionPredicate(header("CamelSplitComplete").isEqualTo(true))
        .process(new MyProcessor());
}
```

Lab 4: Docker Compose

What will be done:

- complete a multi-service application
- write Dockerfiles to create images to deploy your services
- use docker compose to run your multi-service application

The lab assignment is divided in three parts:

- part one
  - download the code of the microase app from moodle
  - complete the math service code to feature the requested operations and set a port in the .flaskenv file
  - run the math service through the provided docker file, after completing it with a suitable port
  - complete the gateway/urls.py file with suitable service name and ports
  - write Dockerfile for the gateway and string_rust services
- part two
  - write a docker-compose.yml file to run all three services
  - run the whole application through Docker Compose
- part three (bonus)
  - add a random service to feature operation related to randomness

TODO EVERYTHING

Lab 5: Kubernetes

TODO

Lab 6: Architectural Smells and Refactorings

The goal of this lab is learn to identify architectural smells and implement the corresponding refactorings in the microase application.

TODO, SERVE MACCHINA VIRTUALE

Lab 7: Static Security Analysis with Bandit

definitions:

- security: the ability to resist attacks
- attack surface: sum of all entry points that can be attacked
- defense-in-depth: layered security levels that adhere standards and best practices to secure a system

As attackers get more sophisticated over time, protecting a system is a never ending process.

Security at all Levels

Security is a **holistic property** of a system, it should in principle be guaranteed at all levels:

- **infrastructure security**, maintaining the security of all systems and networks that support services
- **application security**, securing individual application systems or related groups of systems
- **operational security**, secure operation and use of the organization's systems (e.g. users)

Terminology

- **asset:** something of value which has to be protected
- **attack:** an exploitation of a system's vulnerability. generally, this is from outside the system and is a deliberate attempt to cause some damage
- **control:** a protective measure that reduces a system's vulnerability. Encryption is an example of control that reduces a vulnerability of a weak access control system
- **exposure:** possible loss or harm to a computing system. This can be loss or damage to data, or can be a loss of time and effort if recovery is necessary after a security breach
- **threat:** circumstances that have potential to cause loss or harm. You can think of these as a system vulnerabilities that is subjected to an attack
- **vulnerability:** a weakness in a computer-based system that may be exploited to cause loss or harm

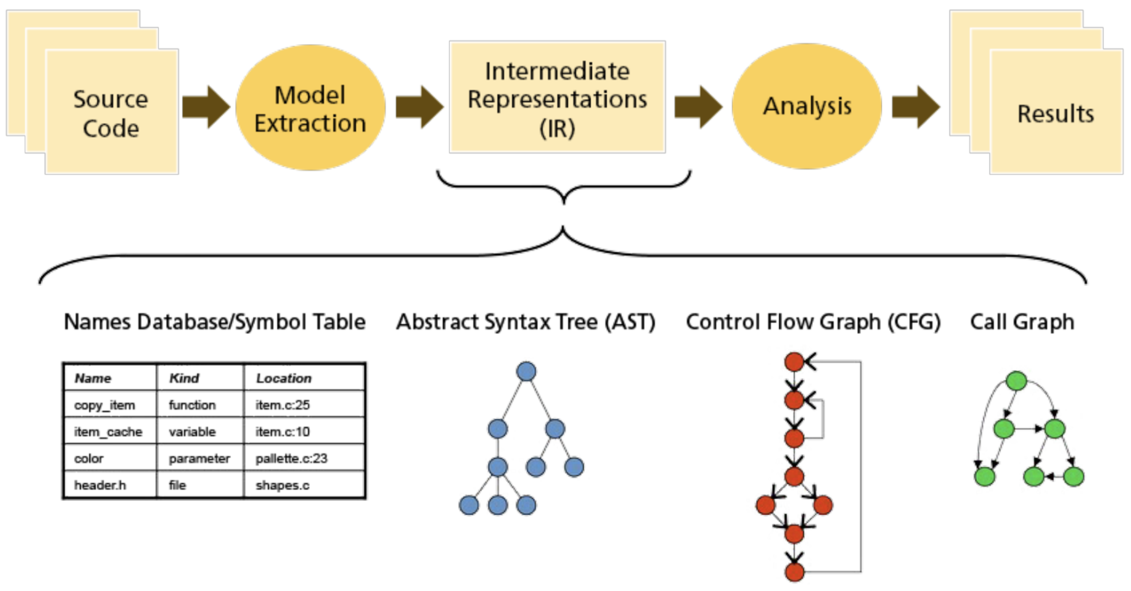
Security Assurance

Achieved by:

- **vulnerability avoidance:** that system is designed to avoid vulnerabilities
- **attack detection/elimination:** attacks are detected and neutralized
- **exposure limitation and recovery:** the system can recover quickly from a successful attack

Since **security is expensive**, organizations use a **risk-based approach** to support decision making and should have a set of security policies to decide upon with.

Vulnerability Avoidance with Static Analysis


Source Code
Model Extraction
Intermediate Representations (IR)
Analysis
Results
Names Database/Symbol Table
Abstract Syntax Tree (AST)
Control Flow Graph (CFG)
Call Graph

Name	Kind	Location
copy_item	function	item.c:25
item_cache	variable	item.c:10
color	parameter	pallette.c:23
header.h	file	shapes.c

Bandit is a static analysis tool designed to find common security issues in python code, by exploiting known patterns.

The lab exercise will consists to analyse a given application with bandit and a fix a simple security issue (e.g. hardcoded password)

Lab 8: API Vulnerabilities and Penetration Testing

Secure Coding Practices

- complexity of source code leads to more security vulnerabilities
- exponential increase of defects as number of lines of increases
- functional and security testing is utterly important

There are two types of testing:

- **static:** tools like bandit
- **dynamic:** what we see in this lab

API Based App

- client devices are getting more powerful
- logic moves from backend to frontend
- client consumes raw data and maintain/monitor user's status
- lots of parameters in http requests
- API disclose things about implementation

API exposes microservices to consumers and it is therefore important to make them secure and avoid known pitfalls.

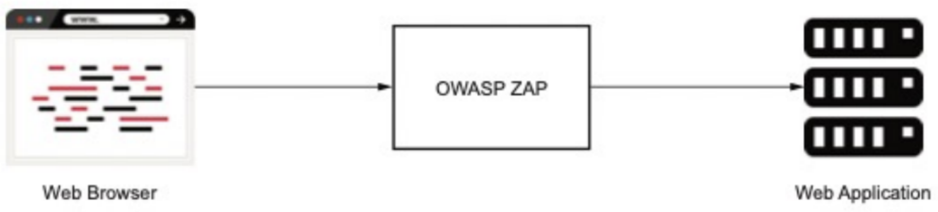
Top 10 API security vulnerabilities:

- broken object-level authorization
- broken authentication
- excessive data exposure
- lack of resources and rate limit
- broken function-level authorization
- mass assignment
- security misconfiguration
- injection
- improper asset management
- insufficient logging and monitoring

Dynamic Analysis

- It checks code while it executes
- it generates various types of input parameters to trigger as many execution flows as possible

OWASP ZAP is a tool for dynamic analysis that helps finding vulnerabilities in running web apps.
ZAP acts as a proxy between the client app and the server.



Passive vs Active Scanning

- **passive scanning:** harmless scan that looks for responses and checks them against known vulnerabilities
- **active scanning:** deliberately attempts to penetrate the system by using known techniques to find vulnerabilities, it modifies the system and its data.
    - penetration testing tests computer systems, networks or web apps for vulnerabilities that could be exploited
    - it is carried out as if the tester was an attacker with a goal of breaking into the system and either stealing data or carrying out some sort of denial-of-service

Passive finds less vulnerabilities but it is faster. Active finds more vulnerabilities but requires time and specific authorizations.

Penetration Testing 101

Penetration Testing follows three stages

- **explore:** the tester attempts to learn as much as he can about the system
- **attack:** the tester attempts to exploit the known or suspected vulnerabilities to prove they exists
- **report:** the tester reports back the results of their testing, including the vulnerabilities, how they have exploited them and how difficult the exploits were, and the severity of the exploitation

WebGoat is a deliberately insecure application maintained by the OWASP designed to teach web application security lessons.

TODO

Lab 9: CI/CD pipelines with Jenkins

TODO

Lab 10: BPMN with Camunda

Coreography vs Orchestration

- **Coreography:** service coreography permits to services to self-coordinate in a P2P fashion
    - **Orchestration:** service orchestration is the automated and centralized coordination of services

Microservice Development

We can design and implement a set of dedicated, autonomous tasks to do business tasks in our company domain, each will have its own business logic, data, and a team maintaining it.

Complexity in modern software systems lies in the collaboration among services (especially when they are a lot!)
Everything might easily "fail fast" when microservices do not manage to properly coordinate.

Camunda

Camunda is a framework supporting BPMN workflows and process automation.
It provides a RESTful API which allows you to use your language of choice.

Workflows are defined in BPMN which can be graphically modelled using the Camunda Modeler.

**Camunda has got 2 "usage patterns":

- **Pattern A:** aka endpoint-based integration
- **Pattern B:** aka queued-based integration

How does it work? (Pattern A)

After defining a BPMN, Camunda can directly call services via built-in connectors. It supports both RESTful and SOAP services in this way.

Scaling
However it can only allows scaling on process instances, not on microservices.

How does it work? (Pattern B)

TODO

Lab 11: Testing

Test Driven Development

TDD will not surely improve code quality, however it will make teams more agile: whenever you break a feature, you know it.

Test First Development: an automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.

Writing Tests

- It is time-consuming and can end up in tests that take too long to run
- it is the best approach to make a project grow at less expenses

Testing Microservices

- **functional testing:** test the functionality of the whole system (unit, feature, system, release)
- **user testing:** test usability by end-users
- **performance testing:** measure the microservice performance against varying workload
- **security testing:** remember OWASP Zap

Unit Tests

- In Flask projects, there usually are, alongside the views, some functions and classes, which can be unit-tested in isolation.
- In Python, calls to a class are mocked to achieve isolation
- **functional tests for a microservice project are all the tests that interact with the published API** by sending HTTP requests and asserting the HTTP responses
- **important to test:**
    - the application does what it is built for
    - a defect that was fixed is not happening anymore

Pattern: create an instance of the component in a test class and interact it with it by mock (or real) network calls.

TODO PYTEST

Load Test

The goal of a load test is to understand your service's bottlenecks under stress
Understanding your system limits will help you determining how you want to deploy it and if its design is future-proof in case the load increases.

Pattern: create an instance of the component and stress it by mocking different amount of workloads.

TODO LOCUST