Advanced Software Engineering (LAB)

Stefano Forti

name.surname@di.unipi.it

Department of Computer Science, University of Pisa


Q&A

Agenda (intro. $(A|B|C)$)

intro

Group Process Evaluation Form						
Each team member should complete this form independently. Compute and report team average for each entry. Do aggregation anonymously.						
Goals						
Goals are unclear or poorly understood, resulting in little commitment to them.	1	2	3	4	5	Goals are clear, understood, and have the full commitment of team members.
Openness						
Members are guarded or cautious in discussions.	1	2	3	4	5	Members express thoughts, feelings, and ideas freely.
Mutual Trust						
Members are suspicious of one another's motives.	1	2	3	4	5	Members trust one another and do not fear ridicule or reprisal.
Attitudes Toward Difference						
Members smooth over differences and suppress or avoid conflict.	1	2	3	4	5	Members feel free to voice differences and work through them.
Support						
Members are reluctant to ask for or give help.	1	2	3	4	5	Members are comfortable giving and receiving help.
Participation						
Discussion is generally dominated by a few members.	1	2	3	4	5	All members are involved in discussion.
Decision-making						
Decisions are made by only a few members.	1	2	3	4	5	All members are involved in decision-making.
Flexibility						
The group is locked into established rules and procedures that members find difficult to change.	1	2	3	4	5	Members readily change procedures in response to new situations.
Use of Member Resources						
Individuals' abilities, knowledge and experience is not well utilized.	1	2	3	4	5	Each member's abilities, knowledge, and experience are fully utilized.

$A \equiv$ group members compile the assessment sheet. The leader averages them anonymously.

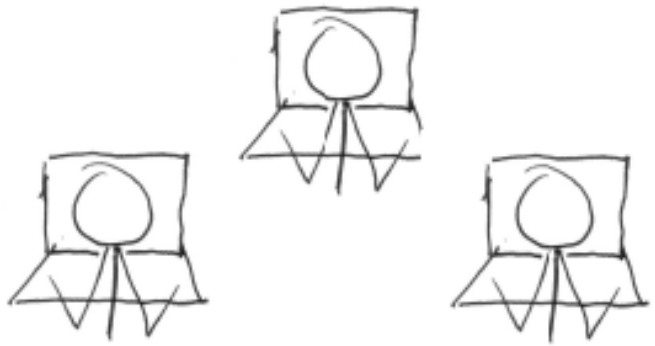
$B \equiv$ in turns, group showcase their project to the TA for evaluation

$C \equiv$ everybody works at splitting the monolith app

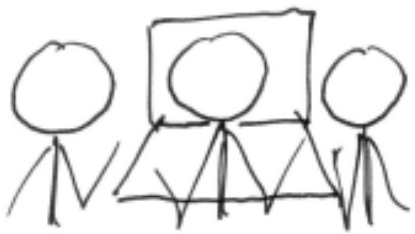
Is Teamwork Important?

- Most real-world software can't be developed by one person in a reasonable amount of time.
- So, teams are needed.
- The problem is... if the team doesn't work well *together* then the project will fail.
- It is not the team leader's responsibility to make the team work well, it is the entire team's responsibility to make the team work well.
- Succeed together or fail together.
- Do matter how good a programmer you are, most companies will not hire you, if you can't work well in a team.

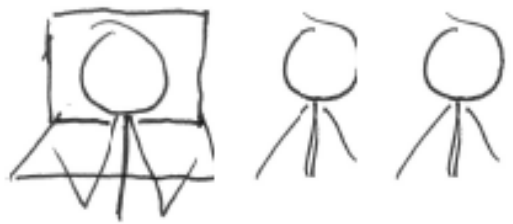
Types of Teams



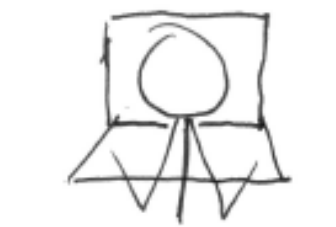
Divide And Conqueror



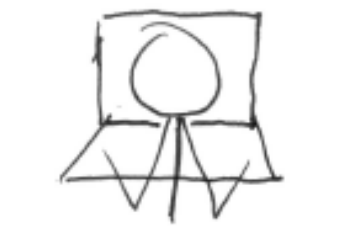
Trio



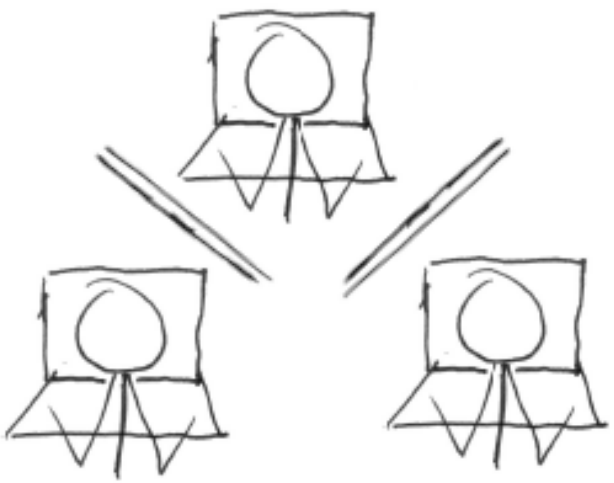
Relay



Drop Outs



One Man Band



Every Man for Himself

Characteristics of Good Teams

- Commitment
- Participation
- Communication
- Trust
- Respect
- Support
- Effective decision making
- Fun

How to?

- Commitment
  - Work hard: Don’t expect others to do what you won’t do yourself
  - Want the project to succeed
- Participation
  - Include everyone in discussions
  - Ask people’s opinion
  - Don’t be shy
  - Understand why it is being done this way
  - Don’t zone out
  - Who is doing what? Clearly assign tasks and/or roles. Rotate them.
- Communication
  - Regular discussions - meeting, email, skype, call, messaging
  - Explain why things were done a certain way
  - Discuss, don’t argue
  - Keep to the point
  - Give you opinion, provided it is constructive
  - Be responsive, e.g. “Can’t do it now, will have it done by 9.”
- Respect
  - Be on time
  - Listen to others and consider their opinion
- Trust
  - Do what you said you will do, when you said you will do it.
- Support
  - Help each other
  - Offer to help
  - Allow space for each other to work
- Effective decision making
  - Be aware of when a decision is needed (and when it isn’t)
  - Sometimes a poor decision is better than no decision
  - Have reasons for your decisions
  - Write down your decisions in an email
- Fun
  - Meetings over coffee
  - Have a laugh
  - Rewards when something if finished or working

Meetings

- What is the objective of the meeting (goals)?
- What topics do you need to talk about (agenda)?
- How long will each item take?
- What preparation is needed?
- When & where

- Hold the meeting
- Write down the findings (during the meeting)
- Write down the action points (during the meeting)
- Review the agenda & objectives, is everything covered?
- Email the minutes

Resolving Arguments

- Kick for touch
- Cool off
- Think through your and the other person's point of view
- Circle back with a facilitator
- Disentangle the argument
  - state the points of agreement
  - discuss the points of disagreement
  - get to the core reasons of why

We are in a World where BeepBeep is getting used by millions of people.
New features, new bugs, an ever enlarging database.

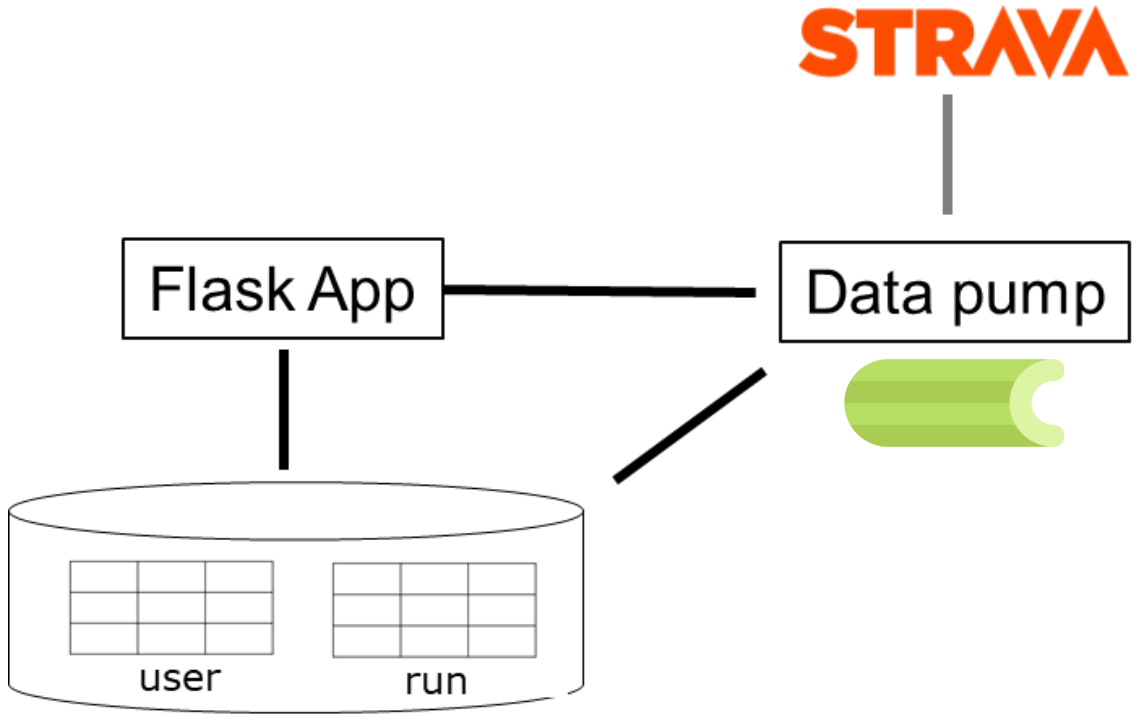
- How to fetch data for all of them with one server?
- Why does the Celery worker need to be in the Flask app to run?
- Why don’t we create a Strava microservice?

Trying to devise a perfect design based on several microservices on day one is a recipe for disaster.

But we're in a lab!

And we can make time go faster (maybe)

The Monolith



- Currently, celery worker takes care of background tasks.
- They get work from Redis and interact with the db.
- We create a Data Service that wraps database calls.

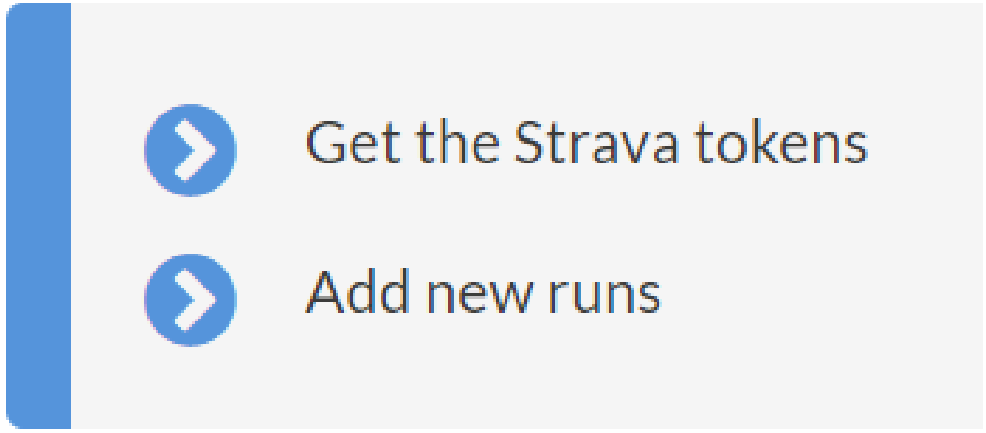
Split it!

- Problem - As the app needs to scale, we have to run background tasks on separate servers.
- Solution – We dedicate a couple of servers to this.
- But – Celery workers need to include the whole Flask app to behave properly.
- Problem – How to reflect changes in the «main» app?

- What can we do?

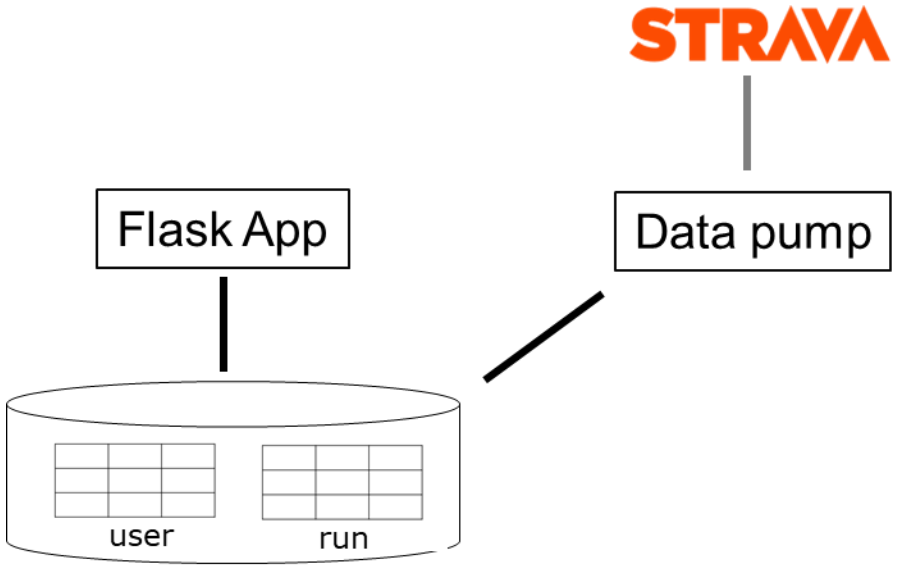
Step 1 – Look for interactions

- The Celery worker interacts with the main app to



- These are DB queries.

- The Celery worker code could be entirely independent and just interacts with the database directly.

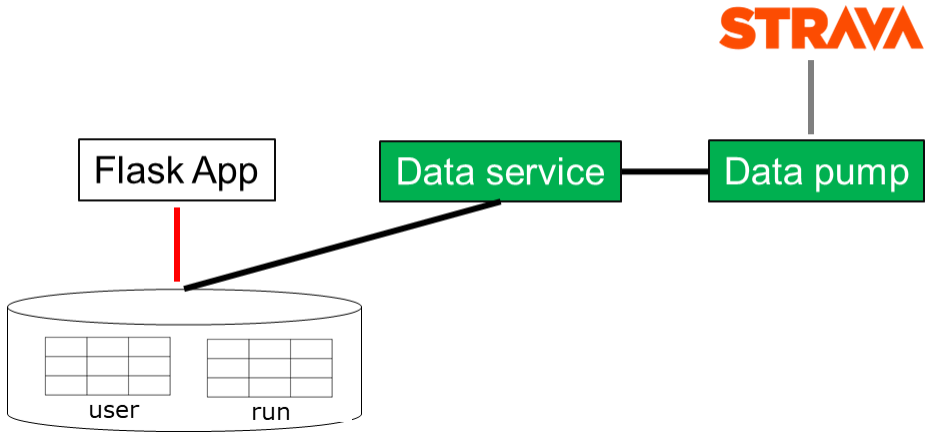


Step 2 – Make design decisions

- Solution - Direct database calls seem a simple solution.
- But - All components share the same database.
- Problem - Every time something changes in it, they all get impacted.

Let's call this new microservice the **Data Service**.

- Solution – Wrap the DB and expose a RESTful API that gives to the different services just the info they need to do their jobs.
- We identified a new micro-service in our architecture!

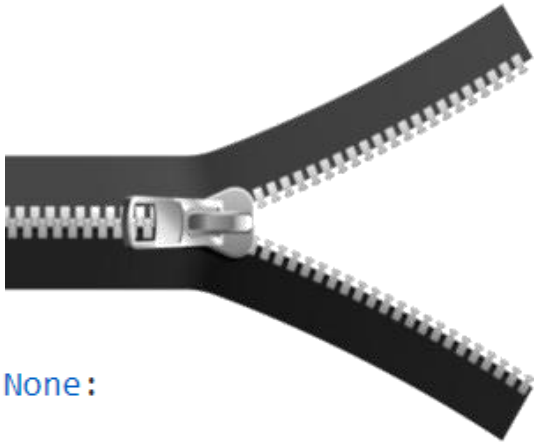


Step 3 – Do the actual splitting

https://www.youtube.com/watch?v=iFIhxPIuNIo

Step 3.1 – Look for seams

- Look for all DB sessions in your code and understand what piece of information they need from the DB.
- Say you found this one →
- It requires to access all users in the DB and will use their Strava tokens.
- How can we do this?



```
with app.app_context():
    q = db.session.query(User)
    for user in q:
        if user.strava_token is None:
            continue
        print('Fetching Strava for %s' % user.email)
        runs_fetched[user.id] = fetch_runs(user)
```

Step 3.2 – Define an API

- For the seam to be unstitched you need to define a suitable RESTful API between the Flask App and the new service.
- You can use tools like OpenAPI to accomplish this task.

OpenAPI

- OpenAPI is a simple description language that lists HTTP endpoints, their usage, and the structure of input/output data.
- For instance, if we want to create a Data Service with a GET endpoint to retrieve all user IDs we would have the following **`api.yaml`** file →

```
swagger: “2.0"
info:
  title: BeepBeep Data Service
  description: returns info about BeepBeep
  license:
    name: APLv2
    url: https://www.apache.org/licenses/LICENSE-2.0.html
  version: 0.1.0
basePath: /api
paths:
    /users:
      get:
        operationId: getUsers
        description: Returns a list of users
        produces:
        - application/json
        responses:
          '200':
            description: All users indexed by their id
            schema:
                type: array
                items:
                    type: integer
```

https://swagger.io/docs/specification/2-0/basic-structure/

Step 3.3 – Implement the API with Flakon

- Flakon has a special **`Blueprint`** class called **`SwaggerBlueprint`**, which takes a Swagger spec file and provides an **`@api.operation`** decorator that is similar to **`@api.route`**.
- It takes an **`operationId`** name instead of a route--so the Blueprint can link the view to the right route explicitly.

```
from flakon import SwaggerBlueprint

api = SwaggerBlueprint('swagger', spec='api.yml')

@api.operation('getUsers')
def get_users():
# .. do the work ..
```

Step 4 – Switch to REST

```
def fetch_all_runs():
    users = requests.get(DATASERVICE + '/users').json()['users']
    runs_fetched = {}

    for user in users:
        strava_token = user.get('strava_token')
        email = user['email']

        if strava_token is None:
            continue

        print('Fetching Strava for %s' % email)
        runs_fetched[user['id']] = fetch_runs(user)

    return runs_fetched
```

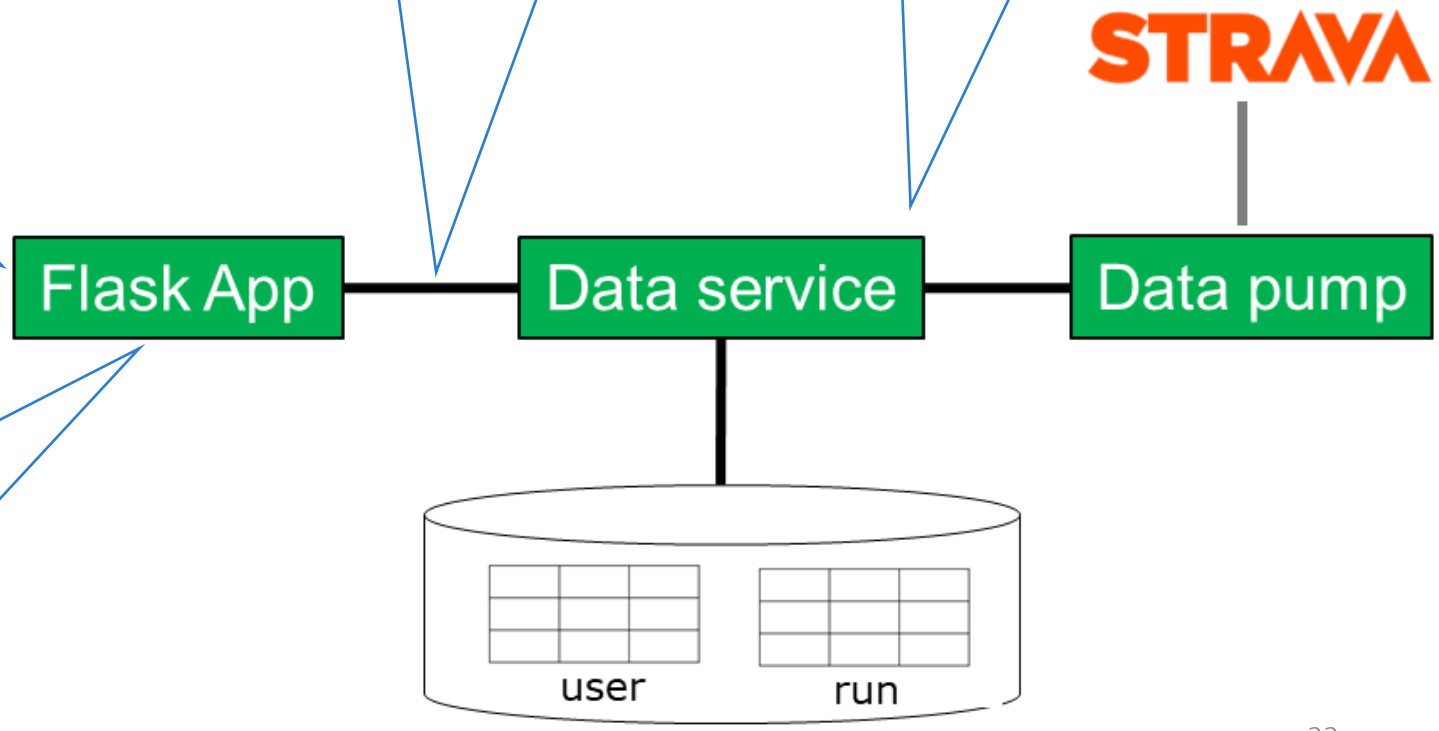
- Use the **`requests`** module to change DB sessions into HTTP requests between services (see Lab 2 for reference).
- Test it! :D

Ta-daaaa!

In **team**, repeat Steps 1-4 to get the whole new microservice architecture!

*We give you **a first possible splitting** of Data Service and Data Pump (Moodle).*

Work on your monolith. Keep a DB with login credentials in the Flask App (`Credential`) so to handle authentication there.



Is there anything else you could split? How?