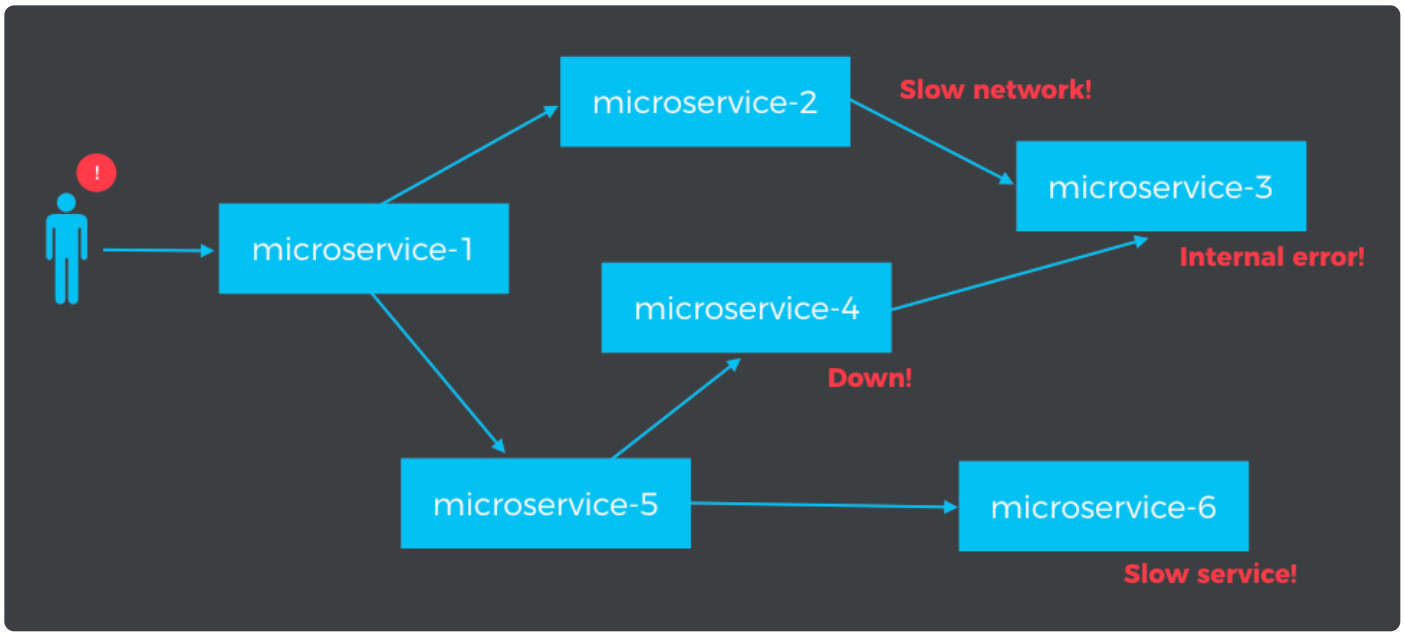
Alejandro Duarte
Feb 15, 2018 8:37:00 AM

Microservices: Fault Tolerance

In the previous article of this series, we developed two functional microservices: a REST web service, and a web UI for it. In this article, we'll add fault tolerance capabilities, by making the web UI resilient to failures in the REST web service.

Why do we need this?

Microservices depend on other services in order to fulfill their requirements. Since microservices are deployed independently, their instances may go up and down frequently. When the number of interactions between microservices increases, the more likely a failure in a service is to cause a severe impact in the system. This is illustrated in the following figure:



By introducing **fault tolerance** capabilities, an application is able to operate at a certain degree of satisfaction when failures appear. Failures in a system make a proportional effect in the operation of the application: The more severe a failure is, the more it affects the application. Without fault tolerance, a single failure in the system may cause a total breakdown.

How does it work?

Fault tolerance can be achieved with the help of a **circuit breaker**, a pattern that wraps requests to external services and detects when they are faulty. When a failure is detected, the circuit breaker “opens”, and instead of keep making requests to the unhealthy service, all subsequent requests immediately return an error. This reduces the load in the unhealthy service, and prevents resource exhaustion in the client.

Additionally to the circuit breaker, fallback mechanisms are used to **fail gracefully** when external services are not available. By incorporating fallback mechanisms, you increase resiliency in applications, which usually have views or

defined in the `biz-application`:

```
@FeignClient(name = "${biz-application.name:null}")
public interface CompanyService { ... }
```

We can implement a fallback class that we can use when the web service is down:

```
@Service
public class CompanyServiceFallback implements CompanyService {

    @Override
    public Resources<Company> findAll() {
        ... return cached or default data ..
    }

    @Override
    public Company add(Company company) {
        ... save in an alternative temporary database ...
    }

    @Override
    public Company update(Long id, Company company) {
        ... save in an alternative database ...
    }

    @Override
    public void delete(Long id) {
        ... mark as deleted in a temporary database ...
    }
}
```

You can find the fallback implementations for both the `admin-application` and the `news-application` on GitHub.

Implementing a fallback service

```
@FeignClient(value = "${biz-application.name:null}",
        url = "${biz-application.url:}",
            fallback = CompanyServiceFallback.class)
@Primary
public interface CompanyService { ... }
```

Use the `@Primary` annotation to declare the type of bean to inject by default in another bean (like beans of the type `VaadinUI`). You also have to enable Hystrix in a configuration file as follows (the default configuration on GitHub already has the required configuration):

```
feign.hystrix.enabled: true
```

Hystrix is a library to control the interaction between services to provide latency and fault tolerance.

Additionally, it makes sense to modify the UI to let the user know that something might not have worked as expected, or would take more time. In the case of the example application, we are simply showing an error notification:

```
public class VaadinUI extends UI {

    private GridCrud<Company> crud = new GridCrud<>(Company.class, new Vertical

    @Override
    protected void init(VaadinRequest request) {
        ...
        crud.setAddOperation(company -> Services.getCompanyService().add(compan
        ...
    }

    private Collection<Company> findAll() {
        Resources<Company> resources = Services.getCompanyService().findAll();
```

```
                crud.getGrid().setHeightByRows(companies.size());
            }
        } else {
            Notification.show("An error occurred. Please try again later.", Not
        }

        return companies;
    }
}
```

Real-life applications might show, for example, a list of companies that haven't been saved and an option to retry. For example, the `news-application`, which renders the latest tweets by the companies in the database, shows tweets from Vaadin, Pivotal, and Netflix when the web service is not available:

```
@Service
public class CompanyServiceFallback implements CompanyService {

    @Override
    public Resources<Company> findAll() {
        return new Resources<>(Arrays.asList(new Company("vaadin"), new Company
    }

}
```

You can try the fallback implementations by stopping all `biz-application` instances and using the application in the browser. You can see the instructions to run the example application in this article.

In the next article of this series, we will learn about UI composition of microservice with Vaadin Framework.

Learn more about Vaadin for Spring Applications