Software Engineering and Architecture
Micro Services

Olivier Liechti
HEIG-VD
olivier.liechti@heig-vd.ch

MSE | MASTER OF SCIENCE IN ENGINEERING

1. What is a “**micro-services**” architecture?

2. What does it mean to “**break the monolith**”?

3. In software engineering, is there a relationship between an architectural style and an organizational culture?

MSE | MASTER OF SCIENCE IN ENGINEERING

NAGANO 1998

Google!

Search the web using Google!

10 results | Google Search | I'm feeling lucky

Google has moved to www.google.com
Please update your links.

About Google!

Stanford Search | Linux Search

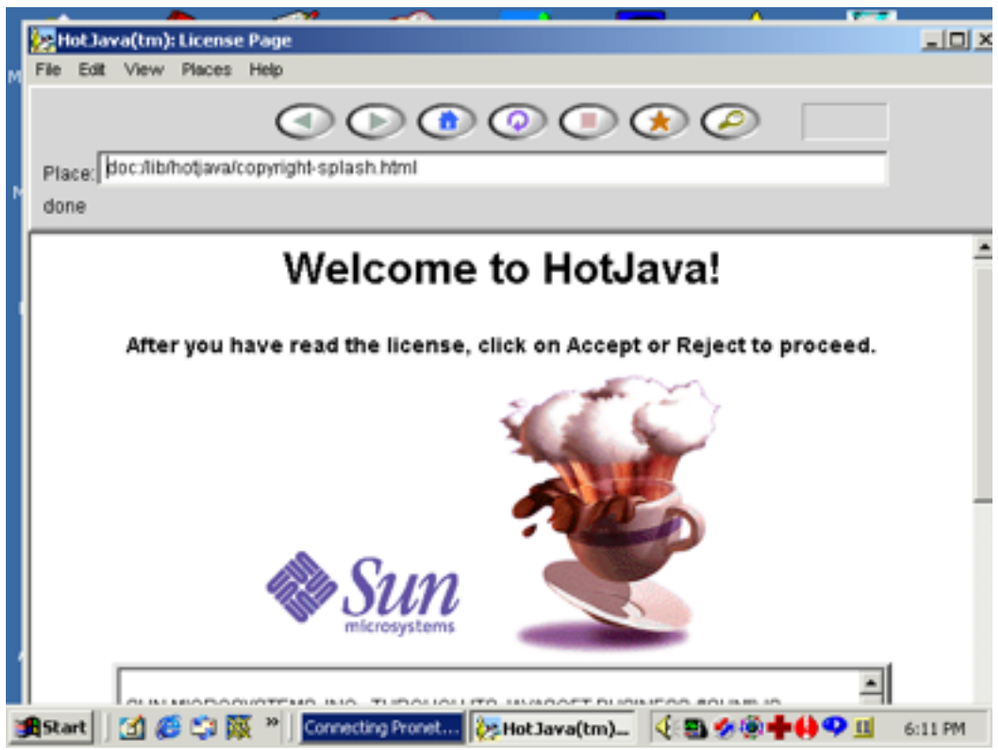
Get Google! updates monthly!

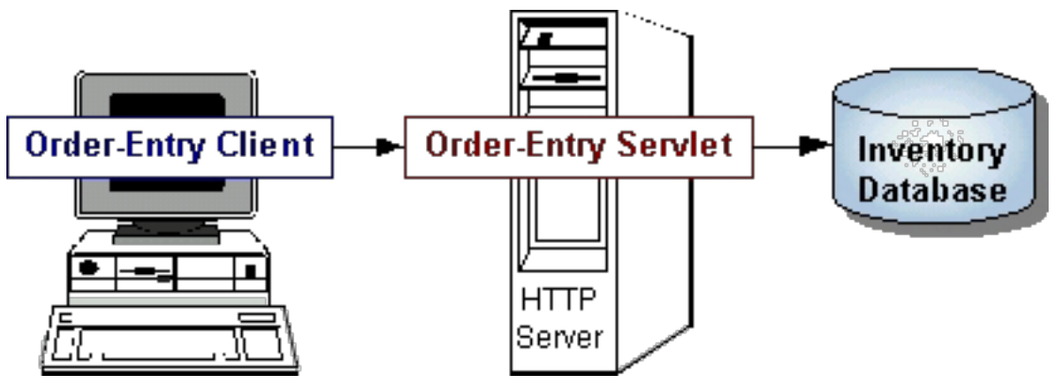
your e-mail | Subscribe | Archive

Copyright ©1997-8 Stanford University

What happened at the end of the 90's?

- With the **exponential growth of the WWW**, there was a big trend to move client-server (e.g. VB) and multi-tiered applications (e.g. CORBA) towards this new open technology platform.
- **Java** was the "hot" language at the time. **Server-side APIs** have been added to the JVM at the time of this transition towards the Web. This is what explains the real success of the Java platform!





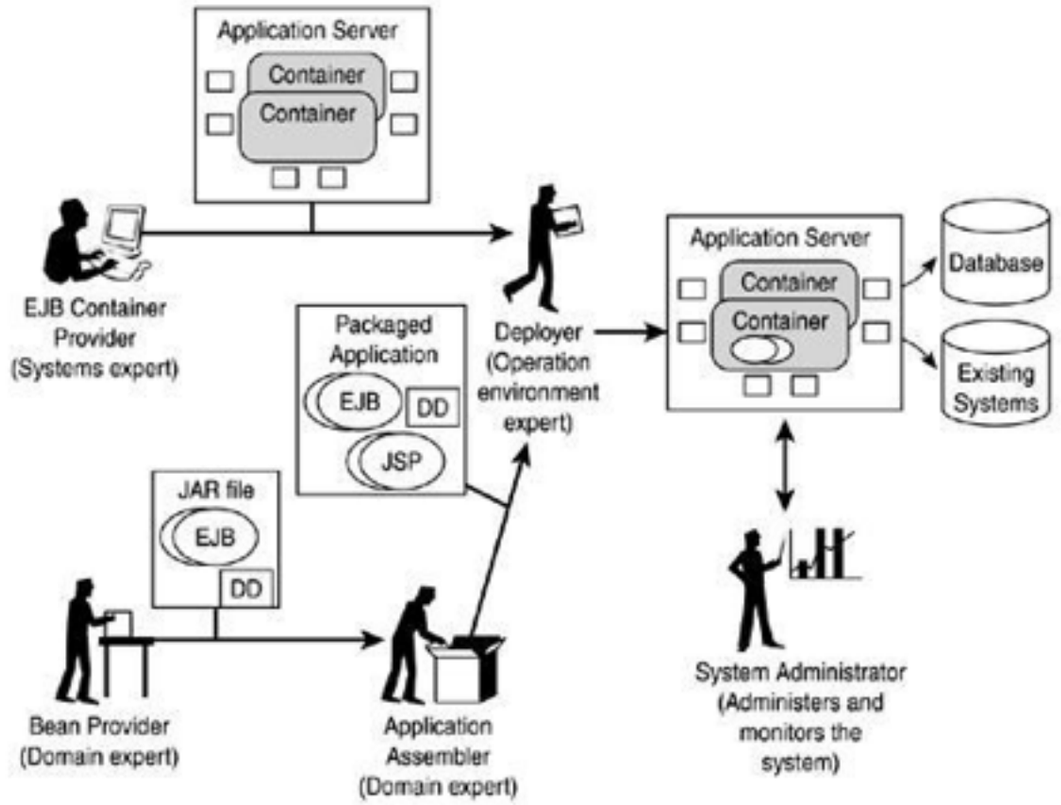
http://tldp.org/LDP/LG/issue45/gibbs/Linux_java.html

What happened at the end of the 90's?

- **Java Servlets** gave us the ability to write server-side code in Java. They are at the origin of a **hugely successful application platform**: Java Enterprise Edition (Java EE), formerly known as Java 2 Enterprise Edition (J2EE).
- Java EE is not only a bunch of API specifications:
  - It **has created a whole industry**. Think about the revenues generated by companies like IBM, Oracle (BEA before its acquisition by Oracle) and many, many others. Think about the jobs created. This is huge.
  - For a long time, it was the **richest open source ecosystem**. This is where a lot of innovation was happening.
  - The Java (EE) community has always been "**software engineering minded**". Agile practices and supporting tools are part of its "DNA".

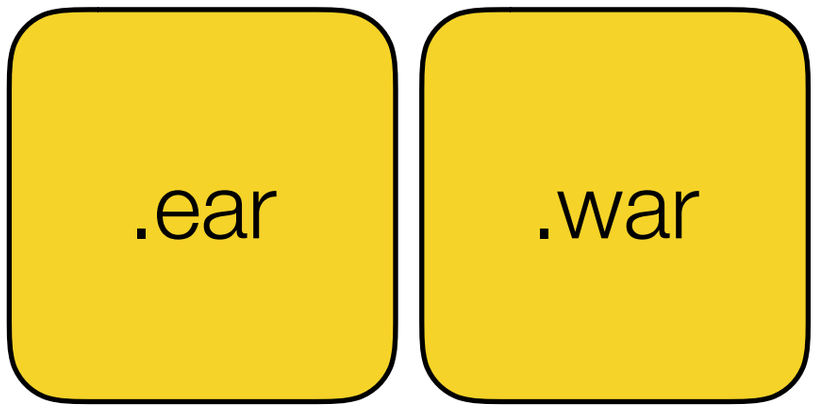
The J2EE model

- The initial J2EE specifications put a lot of emphasis on the “roles”:
  - Component developer
  - Application assembler
  - Deployer
  - System administrator
  - Container provider (e.g. IBM).
- ***Hum... 15 years later, does it still seem like a good idea?*** *How does it relate to the “one-team approach”?*



https://ejbvn.wordpress.com/category/week-1-enterprise-java-architecture/day-01-understanding-ejb-architecture/

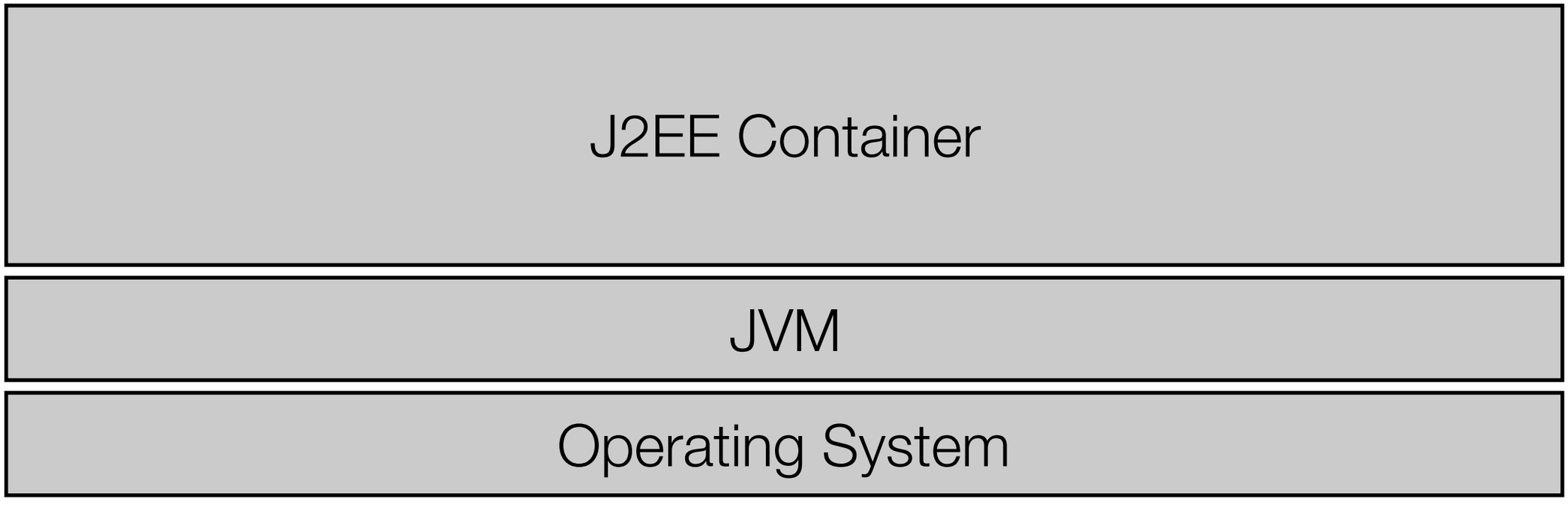
The J2EE architecture



Developers produce "**application archives**" (.ear or .war files) that are deployed in the container.
In practice, many enterprises package the entire application in a single archive.

This archive can become very large. This is what "**monoliths**" are all about.

In reality, J2EE does not force you to do that (the early vision was to package services in "EJB modules". RMI/IIOP was then foreseen as the remote communication protocol!

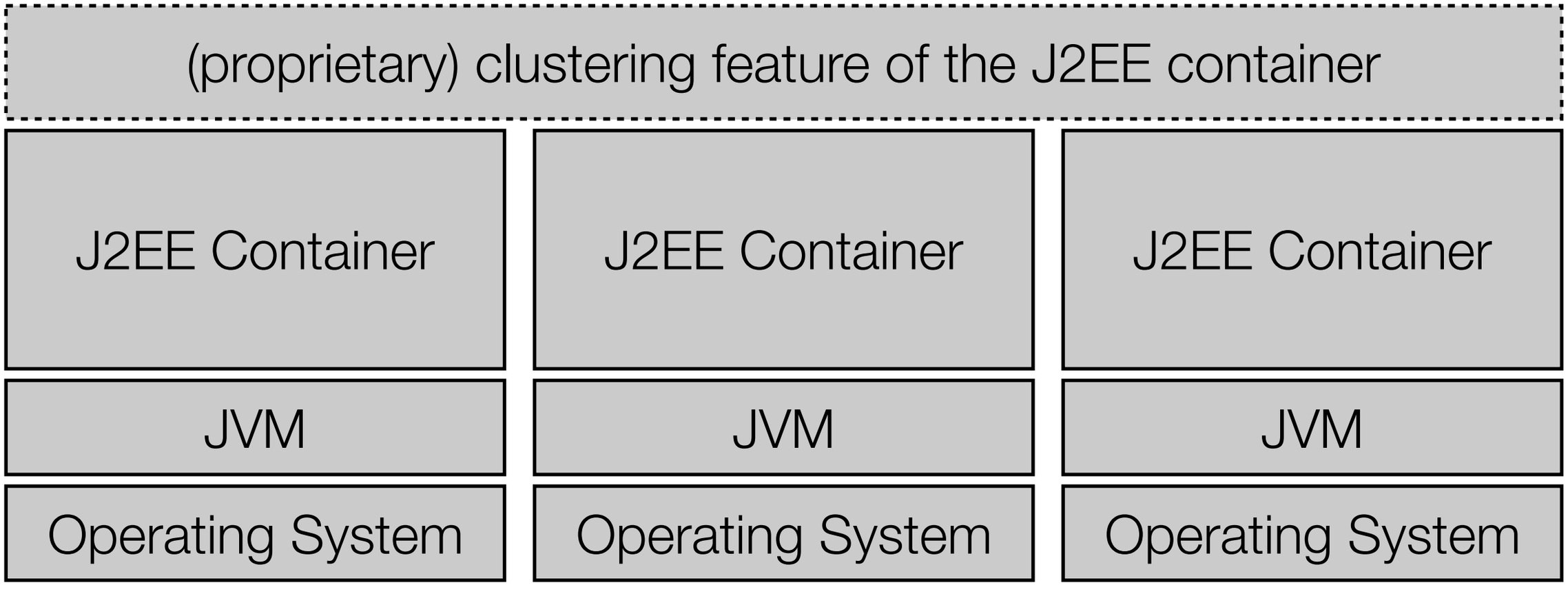


The J2EE architecture (distributed)

In the 2000's, there was a big push by commercial J2EE vendors (IBM, Sun, BEA, Oracle, JBoss, etc.) to extend the core J2EE specs, to deal with distributed system issues. How do we address scalability? fault-tolerance? location transparency?

The vision was to create a "virtual distributed platform" with J2EE. The idea was to be able to package and deploy an application on a group of machines and to "let the infrastructure deal with hard stuff".

Is that really different from the current hype around containers and micro-services?



Fast-forward: where are we now?

- Java and Java EE are still massively used in the market, but:
  - For a long time, developers have learned that you do not *have to* use every single API *offered* by Java EE. You can use a subset of these APIs be perfectly happy.
  - IMHO, when you do that, you still use Java EE. But you will often hear people say “I don’t use Java EE, I use Spring”. Spring is built on some of the Java EE APIs, so what they mean to say is “I don’t use a fully compliant Java EE container and I don’t use EJBs. Use Spring in Tomcat.”
- There is growing adoption of alternatives. Some of them come from Java ecosystem (e.g. “reactive” platforms: vert.x, play/akka/lagom, etc.). Others come from outside (e.g. Node.js).
- Of course, Microsoft .NET is also a widely used enterprise platform.

Is there a future for Java EE?

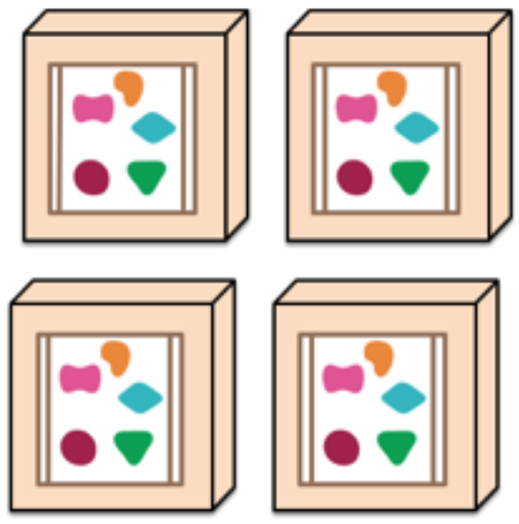
- **Let's have a look at the release dates of the umbrella specification:**
  - J2EE 1.2: December 1999
  - J2EE 1.3: September 2001 (~ + 2 years)
  - J2EE 1.4: November 2003 (~ + 2 years)
  - Java EE 5: May 2006 (~ + 2.5 years)
  - Java EE 6: December 2009 (~ + 3.5 years)
  - Java EE 7: June 2013 (~ + 3.5 years)
  - Java EE 8: ... 2017?? (at least 4 years...)
- Does it still make sense to rely on a model, where a *certified* application server gives us the guarantee that our app can use certain APIs (e.g. Servlets), at a certain version level (e.g. Servlets 2.3)?
- The value of this property was portability across application servers (e.g. moving from IBM WebSphere to BEA WebLogic). We now have other, better ways to manage dependencies on the standard API implementations. We cannot afford to wait several years before using the last versions of these standard APIs.

Is there a future for Java EE?

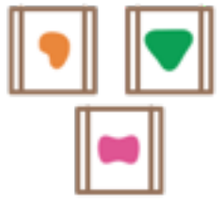
- IMHO, the **individual components** of the Java EE platform (Servlets, JAX-RS, etc.) will still be relevant for a very long time.
- Not only because there is a huge quantity of existing applications that will continue to evolve, but also because they are **powerful and robust**. The community is still very active.
- However, even companies who have invested in “complex, application-server centric infrastructures” are moving towards **more lightweight models**. Again, you will increasingly hear people say “We are now deploying in Tomcat (and moved from X or Y commercial server).
- The current hype around micro-services is further accelerating this trend. Frameworks like **Spring Boot** and **Dropwizard** are built on Java EE APIs, but they further “hide” the container. With them, you can create a standalone executable, which embeds its own container. This is container/cloud friendly.

A monolithic application puts all its functionality into a single process...

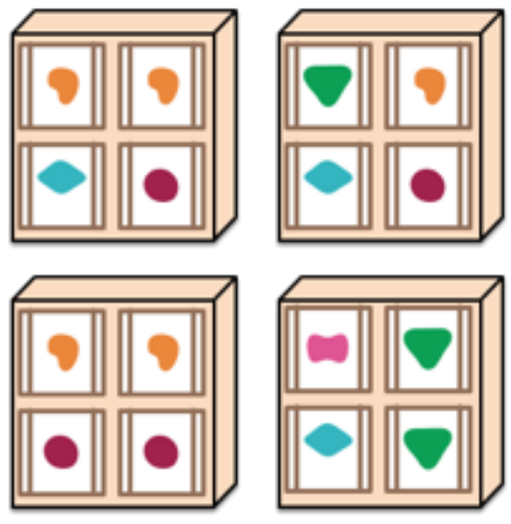
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.

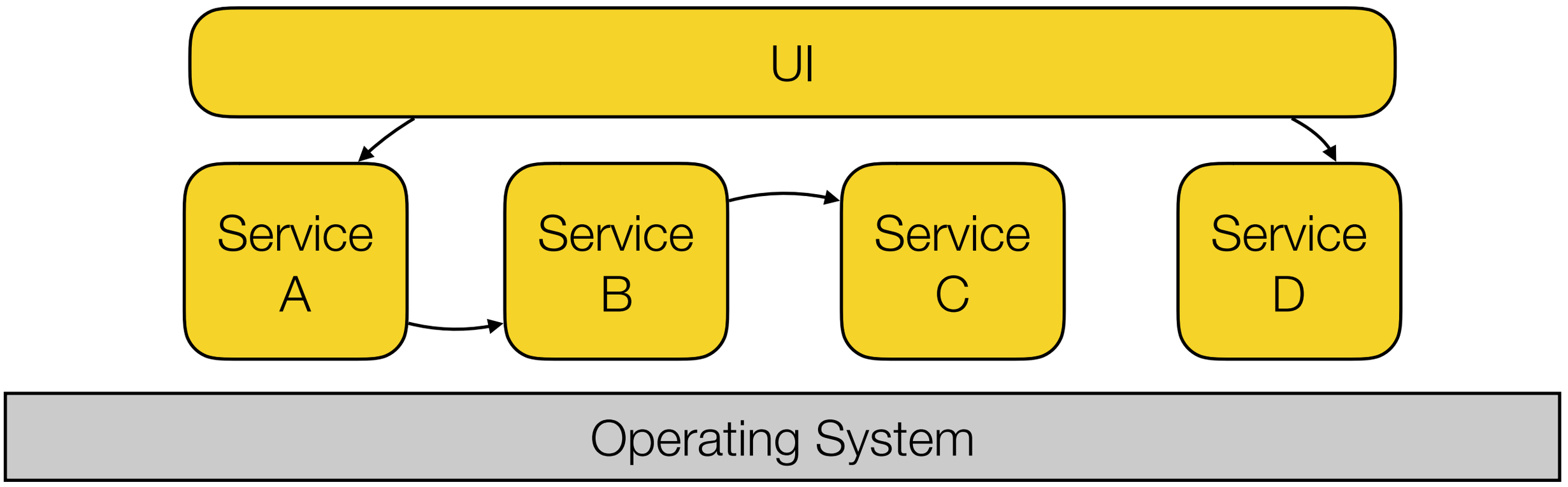


http://martinfowler.com/articles/microservices.html

Micro Services

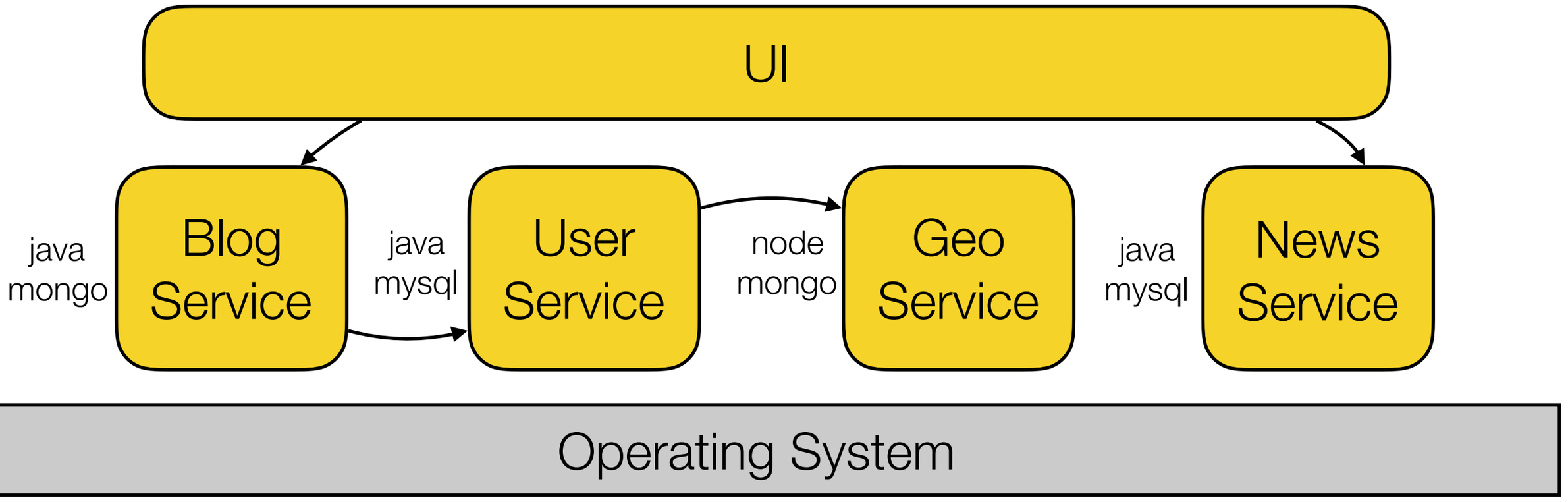
What is a micro service?

"In short, the **microservice architectural style** is an approach to developing a single application as a suite of small services, each **running in its own process** and communicating with **lightweight** mechanisms, often an HTTP resource **API**."



What is a micro service?

"These services are built around **business capabilities** and **independently deployable** by fully **automated deployment machinery**. There is a bare minimum of centralized management of these services, which **may** be written in different programming languages and use different data storage technologies."



What does it mean to “break the monolith”

- An example for a “monolith” is a **large** Java EE application:
  - You may have 15-20 developers working on the app.
  - You may have tens of modules/libraries, with hundreds or thousands of classes.
    - Doing a build with maven may take... minutes if not tens of minutes.
    - When you add a feature, even if it is a small, you have to take down the entire application to deploy it.
- There are many Java EE apps that are “self-contained” and that can still be maintained efficiently. You should not try to split them “just to the follow the trend”. Micro-service architectures have benefits, but they introduce a lot of complexity!

What does it mean to “break the monolith”

- “Breaking the monolith” means that you start to take some of the code that is in the complete application and to move it in independent modules.
- Instead of having one huge project, you now have 2, 3, ..., n smaller and more manageable projects.
- Every project can be built independently. Every project produces an executable service that can be deployed and “live its own life”.

Relationship between architecture and organization

- Agile and lean approaches:
  - See **organizational silos** and **hand-overs** as elements that slow down the development process and have a negative impact on quality.
  - Put an emphasis on **small**, **autonomous** and **responsible** teams.
  - When people use expressions such as “**whole-team approach**”, “**you-build-it-you-run-it**”, “**DevOps**”, it is always very much related to this idea.
  - Small teams that are **fully responsible** for “something” and given the liberty to organize themselves deliver better results than large organizations where responsibility is diffused (“somebody will take care of that... not my business”).

Relationship between architecture and organization

- Very often, a “**monolith**” is developed by an organization where:
  - Some developers work on the UI layer, some developers work on the business logic (without thinking about the UI), some developers work on the data access (without thinking about the business logic).
  - Some engineers take care of the application server infrastructure (without thinking about the apps).
  - Some engineers take care of the testing, once a new version has been built.
  - In other words, people often belong to groups based on their skills. As companies grow, the “Dev” and “Ops” organizations tend to move further apart.

Relationship between architecture and organization

- **Micro-service architectures** encourage “whole team” approaches:
  - A micro-service is a complete, independent component that delivers business value. To build it, you need skills across the whole stack.
  - You can give the entire responsibility of a micro-service to a single team. More and more, the team is not only responsible of the service during development, but during the entire life cycle (people sometimes talk about “products over projects”).

Micro Services with Spring Boot

What is Spring Boot?

- The **Spring Framework** is one of the most widely used frameworks built on top of Java EE technologies.
- It was created more than a decade ago, to provide a "**lightweight**" alternative to full-fledged application servers (based on the principle that you do not have to use EJBs to write robust enterprise apps).
- **Beyond the framework**, Spring is associated to a large and very active **open source community**. It is also associated to a commercial company (**SpringSource**, now part of **Pivotal**).
- Spring is now a **large collection of projects**, that can be combined to create a complete platform. Spring Boot is one of these projects.

What is Spring Boot?

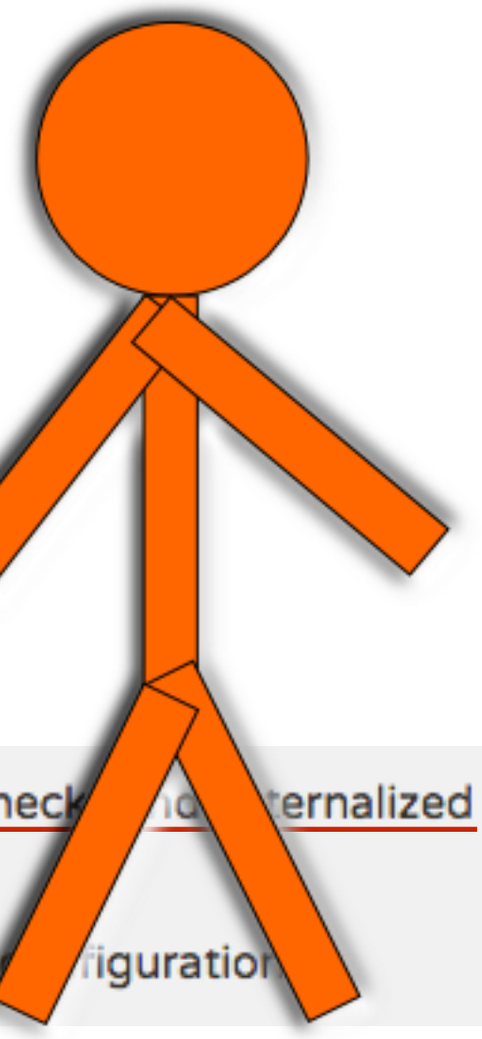
Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

Features

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' POMs to simplify your Maven configuration
- Automatically configure Spring whenever possible
- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

What is Spring Boot?

Thank You!!!



- Provide production-ready features such as metrics, health check[illegible]ternalized configuration
- Absolutely no code generation and no requirement for XML [illegible]figuration

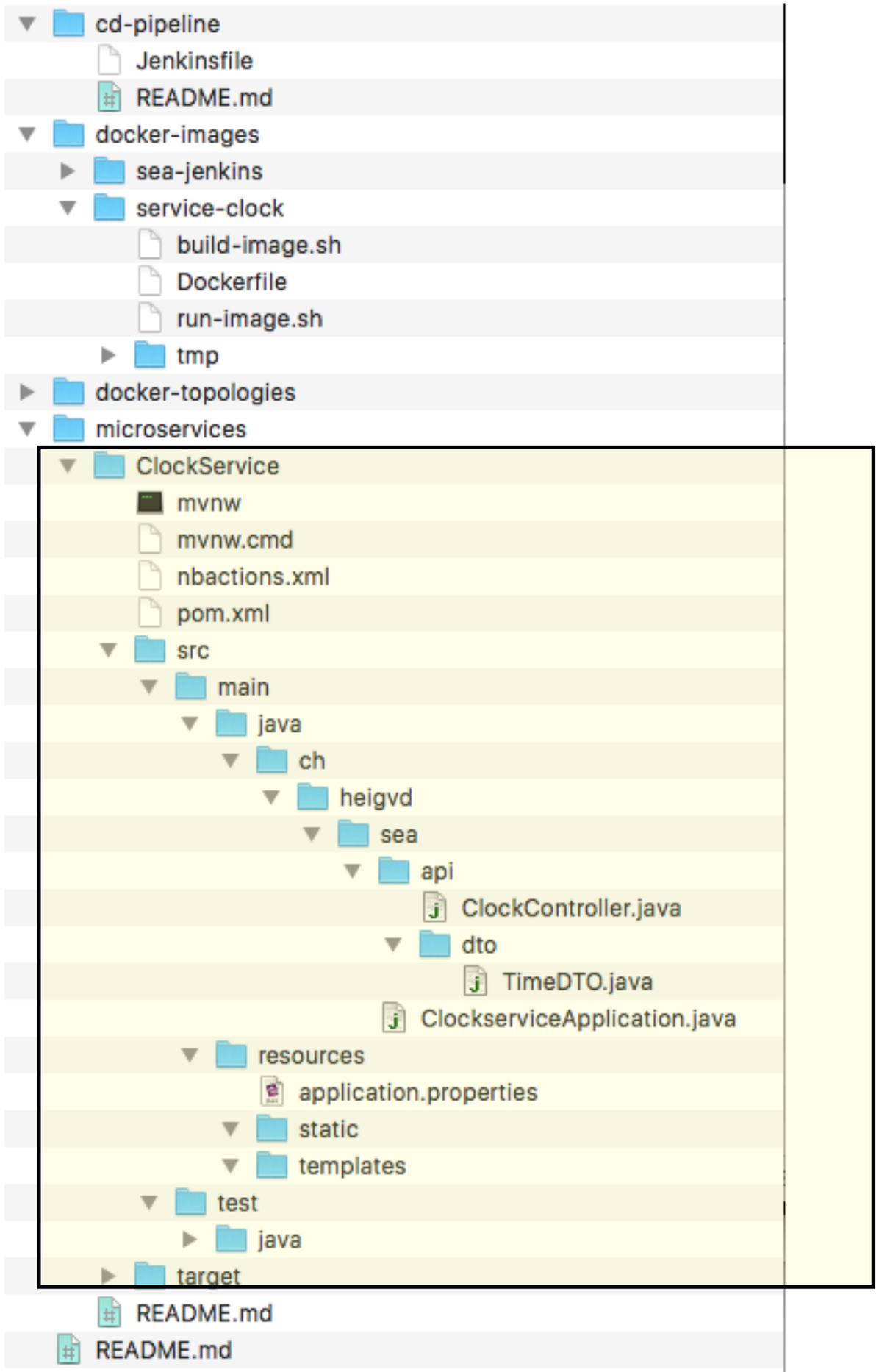
The System Administrator

Let's see what it means in practice!

- **Objective 1**: implement a "clock" micro-service with Spring Boot:
  - We will start from a standard skeleton and from a reference pom.xml file (maven), and add a couple of Java source files.
  - We will then build the project and produce an **executable .jar file**.
  - When we execute this .jar file (with java -jar xxx.jar), it will start the embedded server. We will then be able to invoke several RESTful endpoints:
    - **GET /clock**: our own endpoint, which should return a JSON object containing the current time.
    - **GET /metrics**, GET /health, GET /traces, etc: the endpoints provided by the Spring Boot Actuator module.

Let’s see what it means in practice!

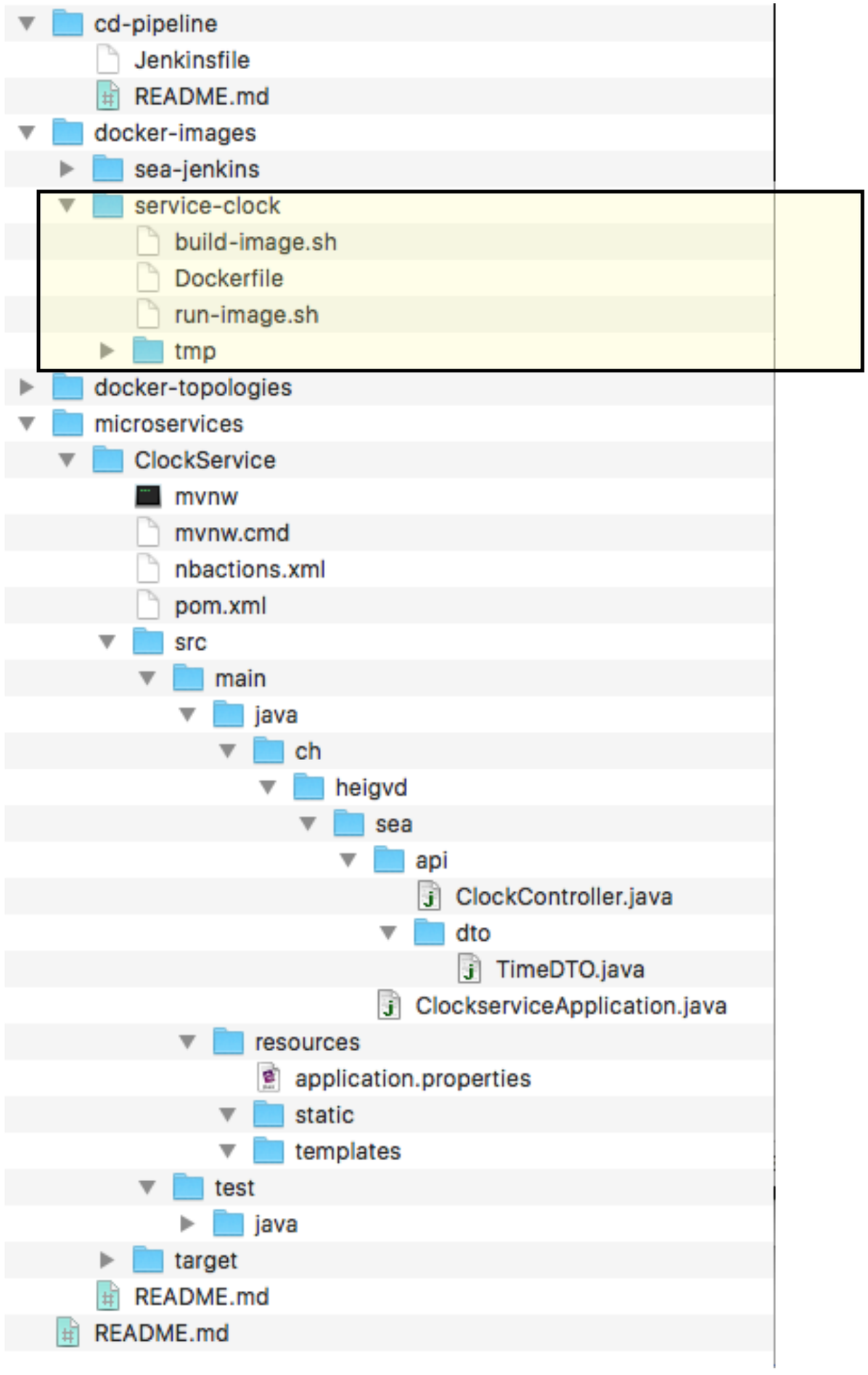
- **Objective 2**: package the clock micro-service in a Docker image:
  - When we have our code ready and have been able to produce a .jar file, we can create a Docker image.
  - To do this, we can use the **java:8** official Docker image as a starting point. If we use maven outside of the image (i.e. we build the executable .jar before building the Docker image), all we need to do in the Dockerfile is to COPY the .jar file, and invoke it in the ENTRYPOINT.
  - When you reach this step and if you experience a slow startup of the Spring Boot app, try to pass **-Djava.security.egd=file:/dev/./urandom** when you invoke the JVM.



This directory contains the sources of our clock micro-service.

When we are in the ClockService directory, we can do a **mvn clean install** to build the executable jar.

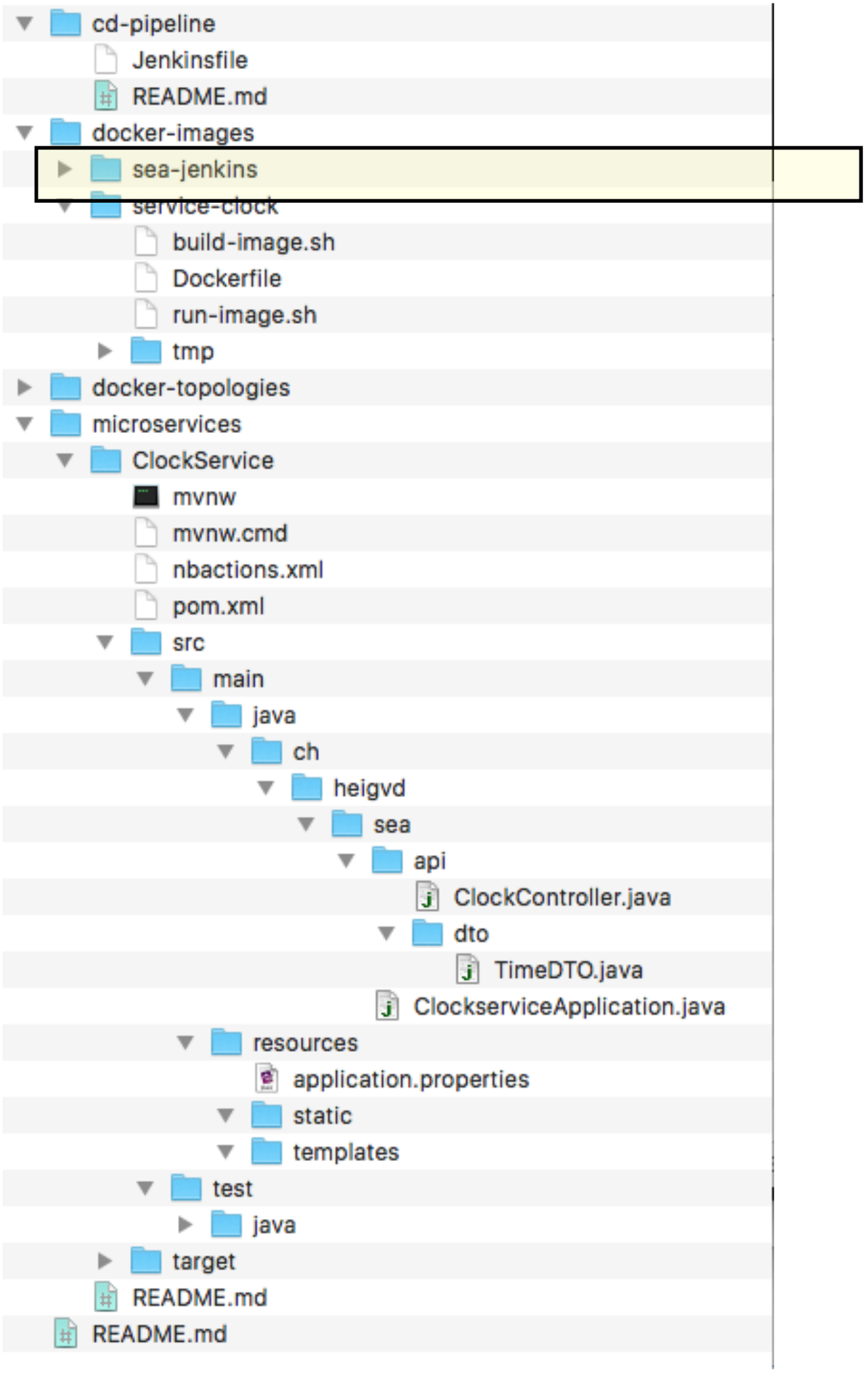
We can then run the app with **java -jar target/*xxx*.jar**.



This directory contains the Dockerfile for building our sea/service_clock image.

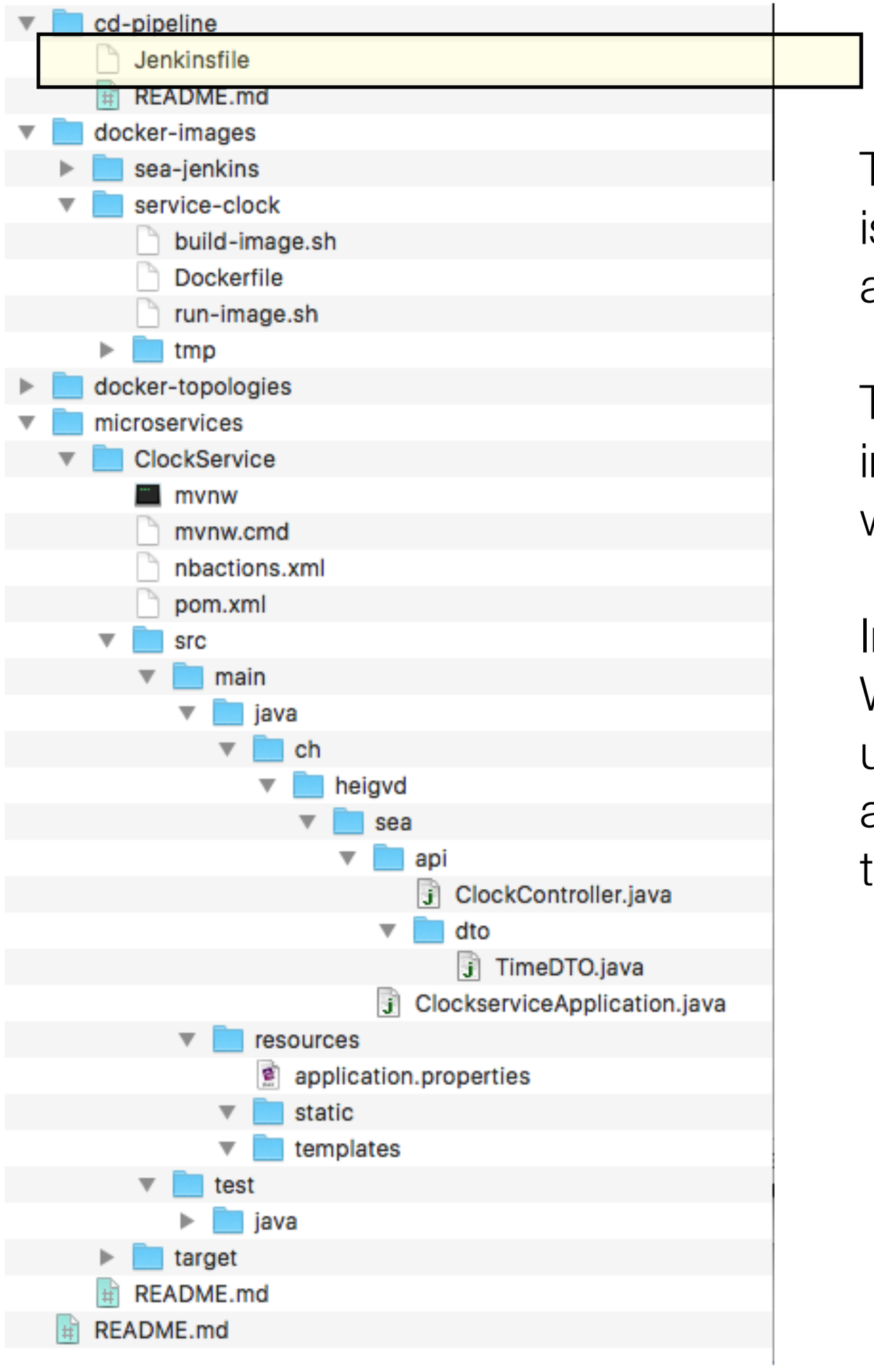
It is quite useful to create a build-image.sh and a run-image.sh to automate the corresponding tasks.

In the build-image.sh, one approach is to firstly do a mvn clean install of the Spring Boot application (I have done it “quick and dirty with relative links and hardcoded folder names...). When this is done, I copy the .jar in the tmp directory, which I can COPY from in my Dockerfile.



This directory contains the Dockerfile and the supporting files for building our customized Jenkins server.

More on this later.



This directory contains a Jenkins file, which is a groovy script that defines the stages and steps of our CD pipeline.

The Jenkinsfile is not used to create an image. It is dynamically fetched over git whenever the pipeline is executed.

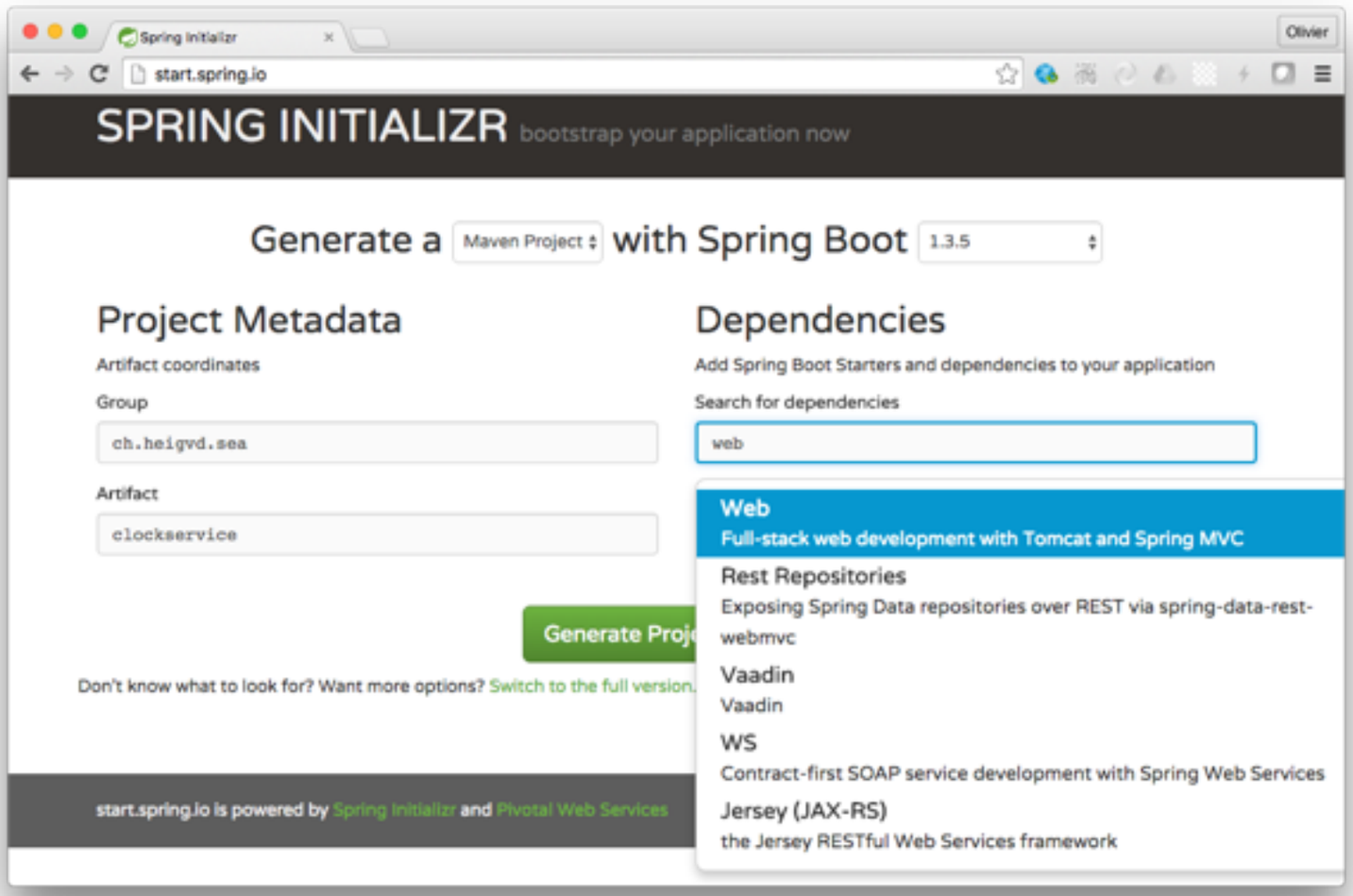
In other words, when I go to the Jenkins Web UI and trigger a new build, Jenkins will use git to retrieve Jenkinsfile (it could be in any git repo!). After that, it will have an up-to-date recipe to do the build.

Activity 1: build a micro-service

- **Step 1: follow this tutorial (outside of the repo)**
  - https://spring.io/guides/gs/actuator-service/
  - Use maven, forget about gradle (unless you know what it is).
  - Make sure that you understand what “Actuator” is all about.
- **Step 2: based on this experience, create a “clock” micro-service in the project repo.**
  - You should implement a /clock RESTful endpoint, which should return a JSON payload with the current time.
  - To get started, check next slide.

How to create the project skeleton

- There are different ways to do that, but here is one that works well:
  - Go to http://start.spring.io/
  - Fill out the **Group** and **Artifact** fields as you like
  - Select “Web” in the **dependencies**
  - Click on the “Generate Project” button and move the downloaded sources to your project folder.



Where did we leave our pipeline?

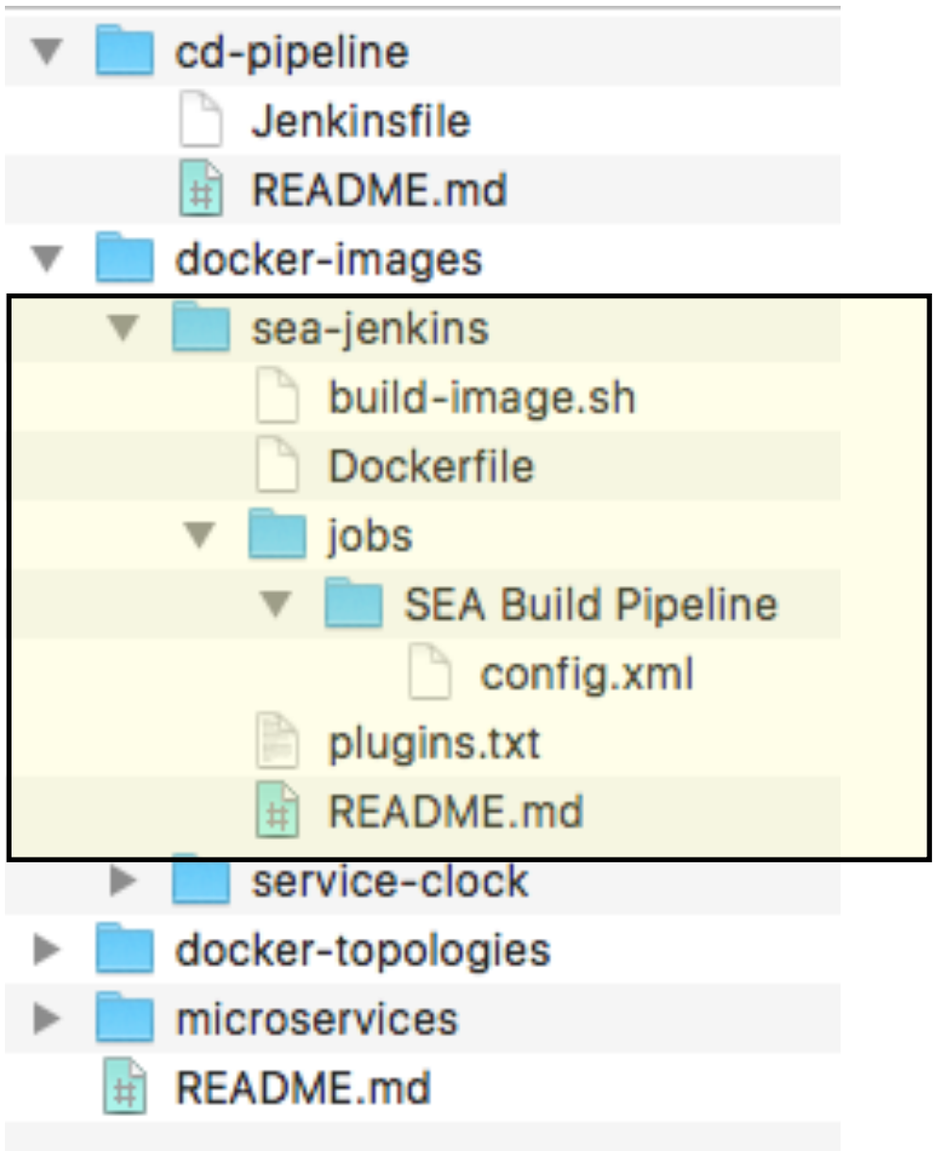
What you should have, at a minimum

- By now, you should be able to start a container from the **official Jenkins Docker image** (by the way, ‘latest’ is back to 1.x and does not point to 2.0 at the moment...).
- You should be able to connect to the Web UI and **do the configuration manually** (install plugins, configure jobs, etc.).
- You should be able to **log into the running container** (with **docker exec**) and explore its filesystem.

What you should have tried to do...

- Define your own customized Jenkins image, by creating a Dockerfile and using the **docker build** command:
  - Find a way to **automatically install a list of plugins**.
  - Find a way to automatically install a job definition, so that whenever a container is started, the user just has to click on a “build” button (and does not have to manually create a job, configure the git repo containing the sources, etc.).

Here is how I have done it



I have created a **Dockerfile**, which we will have a look at in the next slide.

I have created a **plugins.txt** file. It contains the list of plugins I want to install. One instruction of the Dockerfile references this file.

Note: I am running in a **risky setup**, because I have dependencies on the "**latest**" version of the plugins. This has already broken my image several times (the pipeline plugins are very dynamic and their dependencies change rapidly). To have a stable setup, you would need to change the plugin.txt and replace "latest" with **specific versions**.

The **jobs folder** contains one sub-folder for every job. The definition of the job is done in the **config.xml** file. How did I write that? I did the configuration via the Jenkins Web UI (manually), then connected to my running container and grabbed the XML file generated by Jenkins.

Here is how I have done it: Dockerfile

```
FROM jenkins:1.651.1

#
# Install the collection of pipeline plugins on top of the default jenkins installation
#
COPY plugins.txt /usr/share/jenkins/ref/
RUN /usr/local/bin/plugins.sh /usr/share/jenkins/ref/plugins.txt

#
# Install Docker
#
USER root
RUN curl -fsSL https://get.docker.com/ | sh
RUN usermod -aG docker jenkins
RUN curl -L https://github.com/docker/compose/releases/download/1.6.2/docker-compose-`uname -s`-
`uname -m` > /usr/local/bin/docker-compose
RUN chmod +x /usr/local/bin/docker-compose
RUN usermod -aG users jenkins
USER jenkins

#
# Install our job configuration
#
COPY jobs /usr/share/jenkins/ref/jobs/
USER jenkins
```

Here is how I have done it: plugins.txt

```
nodejs:latest
handlebars:latest
icon-shim:latest
scm-api:latest
git:latest
git-client:latest
git-server:latest
durable-task:latest
structs:latest
workflow-durable-task-step:latest
workflow-api:latest
workflow-support:latest
workflow-cps:latest
workflow-cps-global-lib:latest
workflow-job:latest
workflow-basic-steps:latest
workflow-aggregator:latest
workflow-scm-step:latest
workflow-step-api:latest
mapdb-api:latest
ace-editor:latest
jquery-detached:latest
script-security:latest
momentjs:latest
```

```
pipeline-rest-api:latest
pipeline-stage-view:latest
pipeline-build-step
workflow-multibranch
pipeline-stage-step
pipeline-input-step
branch-api
cloudbees-folder
workflow-multibranch
```

Here is how I have done it: config.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<flow-definition plugin="workflow-job@1.15">
  <actions/>
  <description></description>
  <keepDependencies>false</keepDependencies>
  <properties>
    <hudson.model.ParametersDefinitionProperty>
      <parameterDefinitions>
        <hudson.model.StringParameterDefinition>
          <name>MICROSERVICE_URL</name>
          <description></description>
          <defaultValue>http://192.168.99.100:9980</defaultValue>
        </hudson.model.StringParameterDefinition>
      </parameterDefinitions>
    </hudson.model.ParametersDefinitionProperty>
  </properties>
  <definition class="org.jenkinsci.plugins.workflow.cps.CpsScmFlowDefinition" plugin="workflow-cps@1.15">
    <scm class="hudson.plugins.git.GitSCM" plugin="git@2.4.2">
      <configVersion>2</configVersion>
      <userRemoteConfigs>
        <hudson.plugins.git.UserRemoteConfig>
          <url>https://github.com/wasadigi/Teaching-MSE-SEA-2016-MicroServices.git</url>
        </hudson.plugins.git.UserRemoteConfig>
      </userRemoteConfigs>
      <branches>
        <hudson.plugins.git.BranchSpec>
          <name>*/master</name>
        </hudson.plugins.git.BranchSpec>
      </branches>
      <doGenerateSubmoduleConfigurations>false</doGenerateSubmoduleConfigurations>
      <submoduleCfg class="list"/>
      <extensions/>
    </scm>
    <scriptPath>cd-pipeline/Jenkinsfile</scriptPath>
  </definition>
  <triggers/>
</flow-definition>
```

IMPORTANT: You will need to change this in order to use your own **Jenkinsfile**!

What’s next for our pipeline?

Use the pipeline and stage-view plugins

- In our first use of Jenkins, we have used the **“maven” project type**. We have seen that if we have maven project, we can easily ask Jenkins to build it.
- That’s fine, but we now want to **implement a real CD pipeline**, with several stages after the commit stage (validation via different types of tests, deployment in different environments, etc.).
- As we have seen last time, this **a big theme for Jenkins 2.0**. The latest version contains plugins “out-of-the-box”. **We can already use these plugins in Jenkins 1.x**.

Use the pipeline and stage-view plugins

- Our goal is to be able to describe the whole pipeline in some sort of script. We also want to manage this script in our version control system (i.e. git).
  - The **pipeline plugin** allows us to do just that.
  - When we use it, we can indicate in which repo it can find our script (it is called **Jenkinsfile**).
  - When we trigger a build, the pipeline plugin with retrieve the Jenkinsfile (using git) and it will execute it.
- In addition, it would be nice to have a graphical representation of our pipeline.
  - The stage-view plugin gives us that feature.
  - The green/red boxes that you have seen in this webcast are generated by the stage-view plugin: https://www.youtube.com/watch?v=mc2-GeQ0990

A dummy Jenkinsfile... for now

```
node {
    /*-----------------------------------------------------------------------*/
    stage 'Setup'
    /*-----------------------------------------------------------------------*/

     // We have received the IP of the Docker host from a Jenkins job parameter
    echo "We will contact our micro-service via: ${MICROSERVICE_URL}"

    /*-----------------------------------------------------------------------*/
    stage 'Commit'
    /*-----------------------------------------------------------------------*/

    echo "We don't know what to do in the commit stage..."

    /*-----------------------------------------------------------------------*/
    stage 'Validation'
    /*-----------------------------------------------------------------------*/

    echo "We don't know what to do in the validation stage..."

    /*-----------------------------------------------------------------------*/
    stage 'End'
    /*-----------------------------------------------------------------------*/

    echo "We have been through the entire pipeline"

}
```

Next steps (1)

- **The first goal is to reach this stage on your local setup:**
  - You should be able to build and run the Docker image for **your customized Jenkins server.** The plugins should be installed. Your job should be configured.
  - When you trigger a build, your Jenkins job should use the pipeline plugin to **retrieve the Jenkinsfile in your git repo**. It should run the dummy stages and display the results via the stage view plugin.
- **When you are done, you should look for the following information:**
  - Read this documentation: https://github.com/jenkinsci/pipeline-plugin/blob/master/TUTORIAL.md
  - How can add a step in the Jenkinsfile, so that I can fetch my project sources in my Github repo?
  - How can I add a step in the Jenkinsfile to trigger the maven build for the clock micro-service?
  - How can I add a step in the Jenkinsfile to build the Docker image for the clock micro-service?

Next steps (2)

- I am **completely lost**!!
- I haven’t been able to build my custom Jenkins image... I don’t know what I can do to install the plugins... It will already take me too much time to implement the Spring Boot app...
- There is no way I can do that, so should I just stop worrying about it and **do nothing**?

Next steps (3)

- **Here is what you can do to get back on track and up to speed:**
  - Check the original GitHub repo for the project. Have a look at the **`fb-springboot`** branch.
  - What you will find there is the target, with:
    - A Docker image for the customized Jenkins
    - An implementation of the clock micro-service
    - A Docker image for the clock micro-service
    - The “dummy” Jenkinsfile
- **So, what you should try to do is to build the pipeline by yourself. But at anytime, you can always peek at the result. It is also a good idea to start by play**