Instructions for HOL1320

A Step-by-Step Guide from Traditional
Java EE to Reactive Microservice Design

Ondrej Mihalyi

1. Introduction

In this session, presenters and attendees explore together how to migrate parts of an existing Java EE application step-by-step in order to decrease its response time, increase throughput, and make it more flexible and robust. The presentation shows you by example how to apply reactive design to a traditional codebase, split it into several microservices, and deploy them to a cloud environment. Finally it evaluates the performance and flexibility gains.

1.1. Resources

All the tools and resources you'll need are preinstalled in this virtual machine.

In the directory `workspace` (`/home/javaone/workspace`), accessible from the desktop shortcut:

- WorkProject - directory with an initial project which we'll modify
- ExampleProject - an example how the project could end up after applyig all the steps of this lab
- payara-server - the Payara Server install location
- payara-micro.jar - Payara Micro executable

Other tools we'll be using:

- Netbeans IDE, accessible from the desktop shortcut
- Docker Community Edition, accessible from the terminal with command `docker`

Netbeans IDE also contains Payara Server plugin and Payara Server is already configured and ready to use.

1.2. The Work Project

The Work project in the WorkProject directory is an example Java EE application called Cargo Tracker. It's based on the original cargo tracker[1] application (cargo tracker on github[2]). The description of the original application can be found in the readme.txt in the cargo-tracker directory.

The initial application in the cargo-tracker directory is opened as a project in Netbeans IDE (the Projects View).

In this project, we start with the original monolithic application, we focus one particular usecase of searching for routes for delivering cargo and we improve its responsiveness using asynchronous approach in multiple steps.

1.3. The Example Project

This project is an example of how our Work project should finally look like.

The project consists of 2 git repositories:

- ReactiveWay-cargotracker[3] — updates to the original CargoTracker application
- ReactiveWay-cargotracker extensions[4] — additional modules required by the updated CargoTracker application, and information about all the changes and how to run the application

The project's git repositories contain a branch for every step of this lab.

2. Running the Work Project

To run the demo from the above branches, you need to install Payara Server version 4.1.1.171[5] or newer and a standalone Derby database, which is available either in the Payara Server distribution or in a JDK installation.

[1] https://cargotracker.java.net/
[2] https://github.com/javaee/cargotracker
[3] https://github.com/OndrejM-demonstrations/ReactiveWay-cargotracker
[4] https://github.com/OndrejM-demonstrations/ReactiveWay-cargotracker-ext
[5] http://www.payara.fish/downloads

Deploy the application:

1. Start Derby DB on localhost and the default port 1527
   - in Netbeans, open **Services** View, **Databases**, right click on **Java DB** and select **Start Server**
2. Start Payara Server
   - in Netbeans, open **Services** View, **Servers**, **Payara Server**, right click and select **Start** or **Start in debug mode**
3. Deploy the application with context root `cargo-tracker`
   - in Netbeans, right clikc on the **cargo-tracker** project and select either **Run** or **Debug**
4. Verify that the application is running by openning the URL http://localhost:8080/cargo-tracker/ in the browser (should open automatically when the application is started from Netbeans)

Then navigate to the "routing" page we focus on:

1. Select **Administration interface**
2. Select a record in the "Not routed" table
3. The routing page opens with suggested routes

If you have troubles running the application, make sure that the Derby database is running and restart the application.

3. Part 1: Introduce reactive behavior in the monolith

Each of the steps to transform the application is in a separate branch in the Example project[6]:

1. `master` - the original source code of the Cargo Tracker project with some general improvements
2. Asynchronous API and chaining callbacks:

[6] https://github.com/OndrejM-demonstrations/ReactiveWay-cargotracker

a. `01_async_api_01_jaxrs_client` - enhancement of the REST client accessing the pathfinder microservice - uses async API, but the request still waits for results to update GUI.

b. `02_chaining_01_completablefuture` - CompletableFuture is used to chain executions when computation is completed asynchronously

3. Messaging:

a. `03_messages_01_websocket` - added web sockets to update the UI asynchronously and make it more responsive. Web page is loaded immediately and data is pushed later when ready $\rightarrow$ page is lot more responsive. We still have some blocking calls in the pipeline, therefore page still takes unnecessary time to load initially, or the application waits too long before sending updated to the page using the websocket.

b. `03_messages_02_event_bus` - turned synchronous request-response call to the PathFinder component over REST API into asynchronous message passing communication. Each computed item is sent immediately as a message, without any delay. DirectCompletionStream builds upon CompletableFuture to provide means to chain callbacks over a stream of incomming messages, which is not supported by CompletableFuture itself. Incoming messages are turned into websocket messages and sent to the page, therefore the computed data can be displayed immediatelly without waiting for all data.

c. `03_messages_03_jaxrs_sync` - refactoring of the PathFinder module so that it supports both the asynchronous message communication method as well as the original REST API. This is to show the difference between both approaches in the same code base

4. Executing blocking code on a separate thread pool

a. `04_separate_thread_pools_01_for_DB_calls` - Blocking DB calls in ItinerarySelection.java are executing using a separate managed executor service, to avoid blocking the main executor service and listener thread pools, which are meant for non-blocking fast processing and should reserve small amount of threads to decrease unnecessary context switching

5. Context propagation

a. `05_context_propagation_01_jaxrs_async_request` - propagation of JAX-RS request context so that the response from the PathFinder REST API can be built and completed in asynchronous callbacks in different threads if needed

b. `05_context_propagation_02_tx` - propagation of JTA transactions to the threads that execute callbacks

   i. JTA transactions must not be container managed because they need to outlive the method call that started them

   ii. JTA transactions must not be started within an EJB, because EJBs throw exception when such a transaction is not finished before its method is left

   iii. TransactionManage is used to suspend a transaction before an asynchronous call and resume it in a callback

4. Part 2: Introduce reactive microservices architecture

In this part, we will separate a module of the monolith into a standalone microservice, running with Payara Micro. We will then look at the ways how to extend the reactive concepts to the architecture of microservices, beyond a single monolith.

The starting point is the branch `10_monolith_before_splitting`, which already contains the previous reacive improvements in the original monolithic application.

4.1. Introduce a microservice

The branch `11_separate_microservice` in both repositories.

2 new maven modules:

- Pathfinder service (WAR) - a separate microservice providing GraphTraversalService service as both a REST resource and via Payara CDI event bus
- Pathfinder API (JAR) - common code reused in both the monolithic application and the Pathfinder micro service

4.2. Run the project

1. run `mvn clean install` in the root of this repository (for the top-level maven module)
2. deploy the monolithic cargo-tracker application to Payara Server as before
3. run Pathfinder micro service with Payara Micro — go to the directory `pathfinder/target` and execute: `java -jar /home/javaone/workspace/payara-micro.jar --autobindhttp --deploy pathfinder.war` (alternatively copy payara-micro.jar to the target directory or use the Payara Micro maven plugin[7])

The `--autobindhttp` argument to Payara Micro instructs the service to bind the HTTP listener to an available port. Since the monolithic application already occupies the port 8080, therefore the Pathfinder service will probably bind to the port 8081. We can find out the port from the console output. We can check that the application is running with the following URL: http://localhost:8081/pathfinder/rest/graph-traversal/shortest-path?origin=CNHKG&destination=AUMEL

The port number is not important and can even vary. The monolith communicates with the service using the CDI even bus messages and doesn't use the REST endpoint.

4.3. Decouple microservices

In this step, the microservices share the API code. To enable that the service API can evolve without redeploying its clients, we need to avoid the shared code.

This is done in the branch `11_separate_microservice_02_decoupled_api` in both repositories.

Since the API consists of serializable class, we can decouple the API by copying the API classes into the client so that they are still available in both services, but maitained separately. We need to ensure that the `serialVersionUID` remains equal and that the future contract changes are compatible with the standard serialization mechanism, or introduce a custom serialization.

[7] https://docs.payara.fish/documentation/ecosystem/maven-plugin.html

4.4. Introduce JCache for caching and process synchronization

The branch `12_load_balancing_01_jcache` introduces JCache API (JSR 107).

JCache can be used for caching of results to optimize repetitive processing. But if the cache is distributed, it also provides distributed locks, which we will use to synchronize message observers so that at most one of them processes the message.

4.5. Run the microservices

Deploy and run the main application in a usual way on Payara Server.

Run the Pathfinder microservice with Payara Micro:

```
java -jar payara-micro.jar --deploy pathfinder.war --autobindhttp
```

Try if the route cargo page is working, check the logs of the main application and the Pathfinder service.

Build a standalone executable JAR with the Pathfinder microservice:

```
java -jar payara-micro.jar --deploy pathfinder.war --autobindhttp --outputuberjar pathfinder-standalone.jar
```

Run a second instance of the Pathfinder microservice, now using the standalone JAR:

```
java -jar pathfinder-standalone.jar
```

Request the route cargo page in 2 or more different windows at the same time and observer that the requests are load-balanced to one or the other Pathfinder instance.

5. Part 3: Deploying microservices with Docker

In this part, we will deploy the monolith and the microservice as connected docker containers and implement some microservice patterns.

5.1. Build Docker Image of Payara Server with additional configuration

We need to enable Hazelcast in Payara Server, therefore we'll build a custom Payara Server image. The instructions to use the stock `payara/server-full` image and its Dockerfile can be found in the Docker Hub[8].

To see the example, checkout the branch `13_deploy_to_docker_01_simple` in both repositories.

Go into the `cargo-tracker/docker` directory and run the following command:

```
docker build -t reactivems/payara-server .
```

5.2. Running the main application in Docker

Leave the `13_deploy_to_docker_01_simple` branch checked out.

Rebuild the cargo-tracker main application with `mvn install`.

Run the application inside Docker with the following command, with PATH_TO_THE_GITHUB_REPO substituted by the path to parent folder of the cargo-tracker project:

```
docker run -p 8080:8080 -v 'PATH_TO_THE_GITHUB_REPO/cargo-tracker/
target/autodeploy':/opt/payara41/deployments reactivems/payara-server
 bin/asadmin start-domain -v
```

Test that the application is running at the URL: localhost:8080/cargo-tracker[9]

If the application isn't running, try building the application again to deploy it.

5.3. Running the Pathfinder service in Docker

Run the application inside Docker with the following command, with PATH_TO_THE_GITHUB_REPO substituted by the path to parent folder of the pathfinder project:

[8] https://hub.docker.com/r/payara/server-full/
[9] http://localhost:8080/cargo-tracker/

```
docker run -p 8081:8080  -v 'PATH_TO_THE_GITHUB_REPO/pathfinder/
target':/opt/payara/deployments payara/micro java -jar /opt/payara/
payara-micro.jar --deploy /opt/payara/deployments/pathfinder.war
```

Test that the service is running and exposes a REST resource at the URL: http://localhost:8081/pathfinder/rest/graph-traversal/shortest-path?origin=CNHKG&destination=AUMEL

5.4. Running multiple Pathfinder services in Docker

Run additional services with the same docker command, but with modified port mapping. It's not necessary to map the HTTP port:

```
docker run -v 'PATH_TO_THE_GITHUB_REPO/pathfinder/target':/opt/payara/
deployments payara/micro java -jar /opt/payara/payara-micro.jar --
deploy /opt/payara/deployments/pathfinder.war
```