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 5.181[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 (if using Derby DB in Payara Server, execute `javadb/bin/startNetworkServer` or run the DB from Netbeans)
2. Start Payara Server with the default configuration `domain1` (either from Netbeans or from command line run `bin/asadmin start-domain -v`)
3. Deploy the application with context root `cargo-tracker` (either build & run the project from Netbeans and select Payara Server to deploy, or run from command line `</path/to/payara>/bin/asadmin deploy -force=true -contextroot cargo-tracker target/cargo-tracker.war`)
4. Verify that the application is running by opening the URL http://localhost:8080/cargo-tracker/ in the browser.

You may also verify that Payara Server console is available and the Cargo Tracker application is deployed - http://localhost:4848.

Navigate to the "routing" page we focus on:

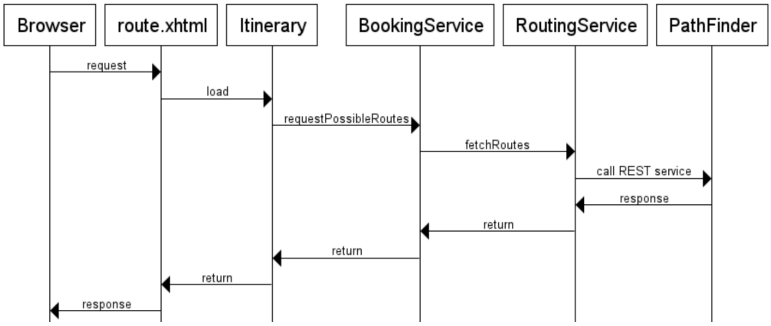
1. Select **Administration interface**
2. Select a record in the "Not routed" table
3. The routing page opens with suggested routes

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

Let’s start the exercises with the initial project in the Work Project. You can find the initial project also on the `master` branch of the Example project if you want to compare it with subsequent stages of the Example project.

The overview of the application components we will work with and they interact in the initial version with a traditional design:



3.1. 1. Enhance reactive REST client

Tasks:

- modify `ExternalRoutingService` to use reactive JAX-RS client API to access the remote `graphTraversalResource` service
- use the method `rx()` to turn the synchronous `Invocation.Builder` into a `RxInvoker`
- refactor the call to `get()` method on `Invocation.Builder` to call the service with the `get()` mthod on the `RxInvoker` instead

> You may use an instance of `GenericType` as an argument to define the type to convert the response to. Otherwise the type will be `Response` and it needs to be converted to the correct type in the handler

The new `get()` method returns `CompletionStage`, which we can return immediately or chain handlers before we return it to upper levels.

At this stage we're not going to change the definition of `fetchRoutesForSpecification` method, therefore we add a handler to the CompletionStage, wait for the stage to complete and return data as before.

Check the branch `01_jee8_async_api_01_jaxrs_client` in the Example Project for an example solution.

3.2. 2. Refactor method defition to cascade reactive calls

We now need to return the `CompletionStage` so that the modified method `fetchRoutesForSpecification` doesn't have to wait for getting the results from a remote call.

Tasks:

- change return type of `fetchRoutesForSpecification` to CompletionStage instead of List
- refactor all broken code so that `DefaultBookingService` receives a `CompletionStage`
- in `DefaultBookingService`, wait for the future to finish and return its result in the same way as we did in `ExternalRoutingService` before
- we need to repeat this for all components on the upper level

Check the branch `02_jee8_chaining_01_completablefuture` in the Example Project for an example solution.

3.3. 3. Introduce Web Sockets to update UI asynchronously

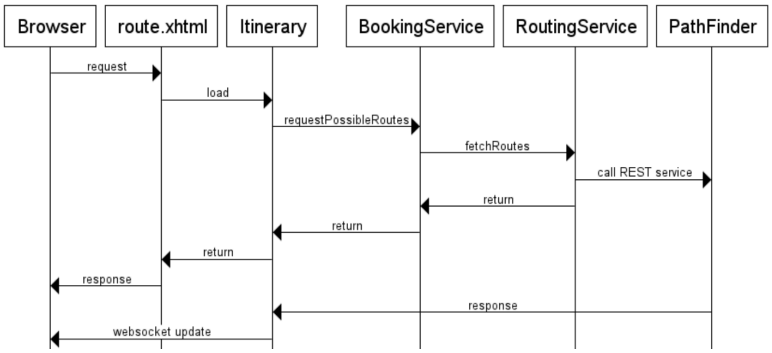
In this step, we complete the asynchronous chain up to the view bean. In the UI level, we turn CompletionStage into an update over a WebSocket.

This step requires a lot of tedious refactoring and boiler-plate code therefore you are encouraged to copy the solution from the branch `03_jee8_messages_01_websocket` in the Example Project.

The refactoring involves:

- propagating `CompletionStage` up to the `ItinerarySelection`
- change the page so that it accepts empty data
- initiate a WebSocket connection from the page
- send an event to update data on the page when the `CompletionStage` is completed

The current design:



3.4. 4. Introduce streaming

We've improved responsiveness a bit but the data are still displayed after all the data is available. We can improve it so that the page is immediately updated with partial results as soon as they are available.

Because REST protocol is limited to waiting for whole response, we'll refactor `GraphTraversalService` to use CDI events instead. We'll use CDI events to send both the request to the `GraphTraversalService` and also to send back pieces of the response from the service.

To speed up this exercise, you can start from the code in the branch `03_jee8_initial_messages_02_event_bus`, which already contains:

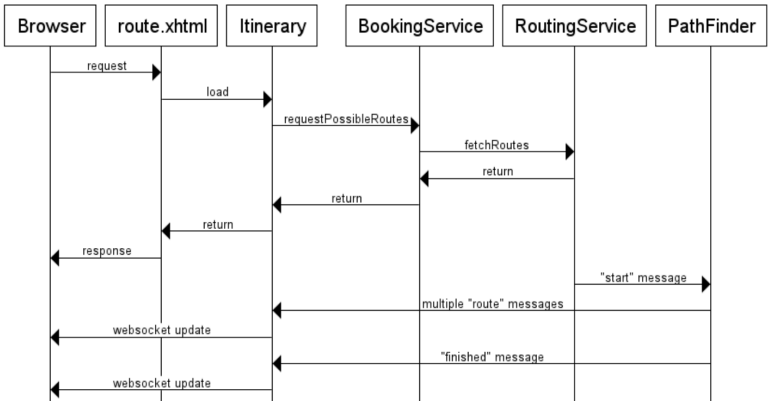
- dependency to RxJava, which provides a reactive API to simplify work with streams of data (dataflow)
- `GraphTraversalResource` - wraps sending and receiving CDI events on the client side. Needs to keep track of all the requests and match events to the corresponding requests (to their event emitters)
- `GraphTraversalRequest` and `GraphTraversalResponse` event types
- `route.xhtml` was modified to expect "error" message, "finished" message. Any other message will trigger update of the datatable

Tasks:

- modify `ExternalRoutingService` to use `GraphTraversalResource` to invoke the remote service instead of the REST client API

- modify the chain of invocations from `ExternalRoutingService` to `ItinerarySelection` so that `Flowable` is passed and `ItinerarySelection` finally adds handlers to it (doOnNext, doOnError, odOnComplete)
- call `subscribe()` on the `Flowable` in `ItinerarySelection` to trigger the execution

The current design:



3.5. Summary of all steps

For reference, each of the steps to transform the application is in a separate branch in the Example project[6].

During the lab we won't go through all of the steps. The rest of the steps are optional to improve different aspects of the application. They are not required to move to the steps in the next section.

1. `master` - the original source code of the Cargo Tracker project with some general improvements
2. Asynchronous API and chaining callbacks:
    a. `01_jee8_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. For Java EE 7 version, see `01_async_api_01_jaxrs_client`

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

   b. `02_chaining_01_completablefuture` - CompletableFuture is used to chain executions when computation is completed asynchronously. For Java EE 7 version, see `01_async_api_01_jaxrs_client`

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 → 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.

4.1. 1. Introduce a microservice

We want to separate the `GraphTraversalService` module into a standalone Pathfinder microservice. This will give several advantages:

- The code can evolve separately from the monolith
- We can run several instances of Pathfinder to scale (speed up) the application

Tasks:

- create a WAR maven project called `pathfinder`
- create a JAR maven project called `pathfinder-api` and add it as a dependency to the new WAR project and also to the existing monolith project
- move common GraphTraversalService classes to `pathfinder-api` (TransitEdge, TransitPath, GraphTraversalRequest, GraphTraversalResponse)

- move all other GraphTraversalService to `pathfinder` (GraphTraversalService, GraphDao)

Because the communication already goes through the CDI event bus, we don't need to let the monolith know how to route the events. We just need to make sure that Pathfinder joins the same cloud as the monolith.

4.2. 2. Run the microservice architecture

Start with the branch `11_separate_microservice` in both repositories.

There are 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

Tasks:

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 payara-micro.jar -clustermode domain -autobindhttp pathfinder.war` (alternatively 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

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

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. 3. Introduce JCache for caching and process synchronization

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.

Tasks:

- add JCache maven dependency to both projects: `javax.cache:cache-api:1.1.0`
- Inject `javax.cache.Cache<Long, String>` into `GraphTraversalResource`, give it a name with `@NamedCache` *

The branch `12_load_balancing_01_jcache` contains complete solution.

4.4. 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 observe 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

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

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/payara/deployments payara/server-full
```

Test that the application is running at the URL: localhost:8080/cargo-tracker[8]

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 (Note: It's not necessary to map the HTTP port because the application doesn't use it):

[8] http://localhost:8080/cargo-tracker/

```
docker run -v 'PATH_TO_THE_GITHUB_REPO/pathfinder/target':/opt/payara/
deployments payara/micro --deploy /opt/payara/deployments/pathfinder.war
 --clustermode domain:172.17.0.2:4900
```

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

Explanation

Payara Micro service can form a cluster together with Payara Server. We need to use `-clustermode` argument to instruct Payara Micro where to find the Payara Server Domain Server which we started in the previous step. You need to supply the IP of the network interface inside the docker container, which you can find out using one of the following ways:

- execute `docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' PAYARA_SERVER_CONTAINER_ID` - supply the container ID of the running Payara Server container you started in the previous step
- attach to the running Payara Server container with `docker exec -it PAYARA_SERVER_CONTAINER_ID /bin/sh`, execute `ip addr show eth0` and fing the IP address that follows `inet`

5.4. Running multiple Pathfinder services in Docker

Run additional services with the same docker command:

```
docker run -v 'PATH_TO_THE_GITHUB_REPO/pathfinder/target':/opt/payara/
deployments payara/micro --deploy /opt/payara/deployments/pathfinder.war
 --clustermode domain:172.17.0.2:4900
```