redhat.

Wildfly Swarm

Rightsize Your Services

Stuart Douglas
Principal Software Engineer

Topics

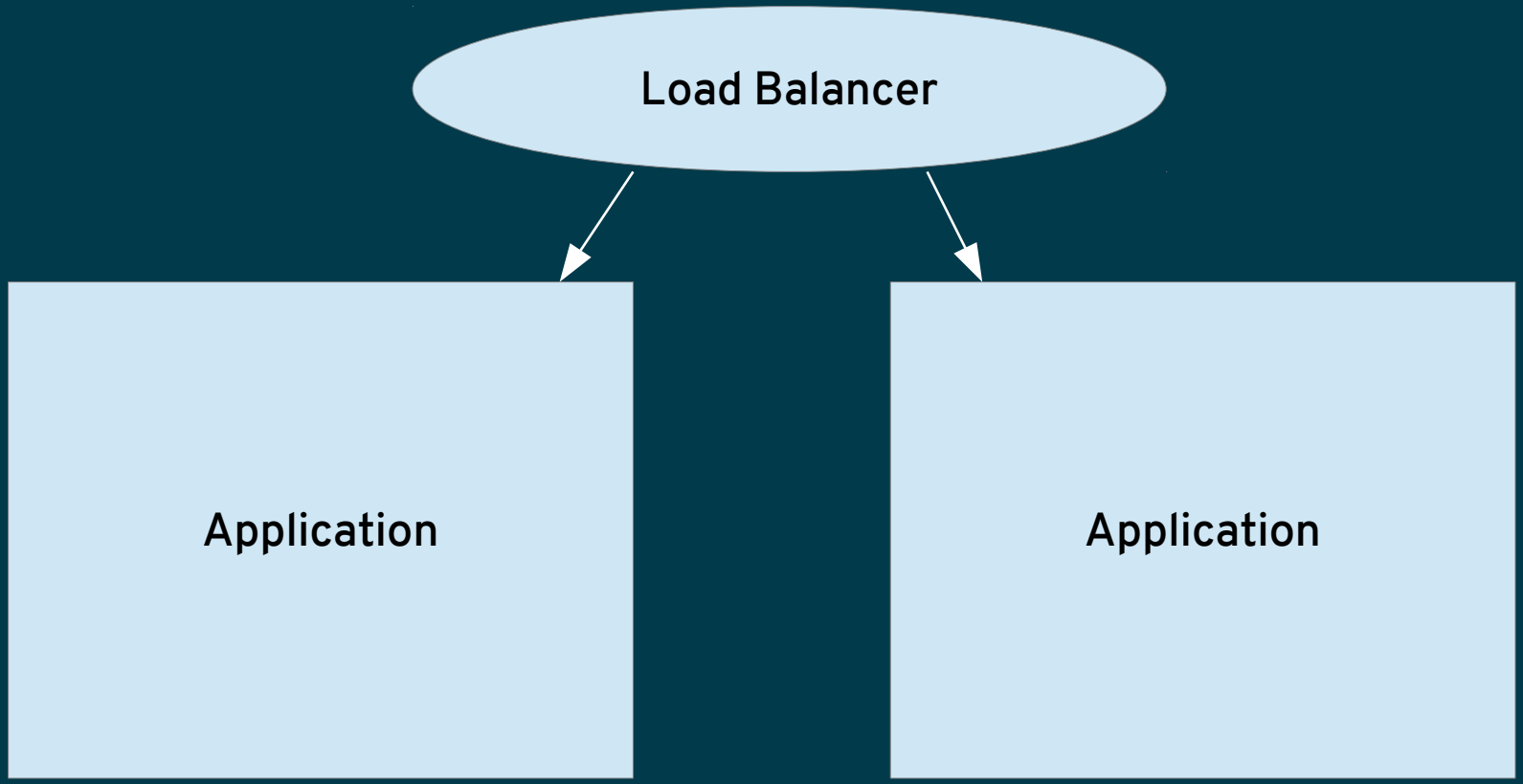
- Some background
- What and Why?
- Core concepts
- Usage
- Demo

What are Microservices anyway?

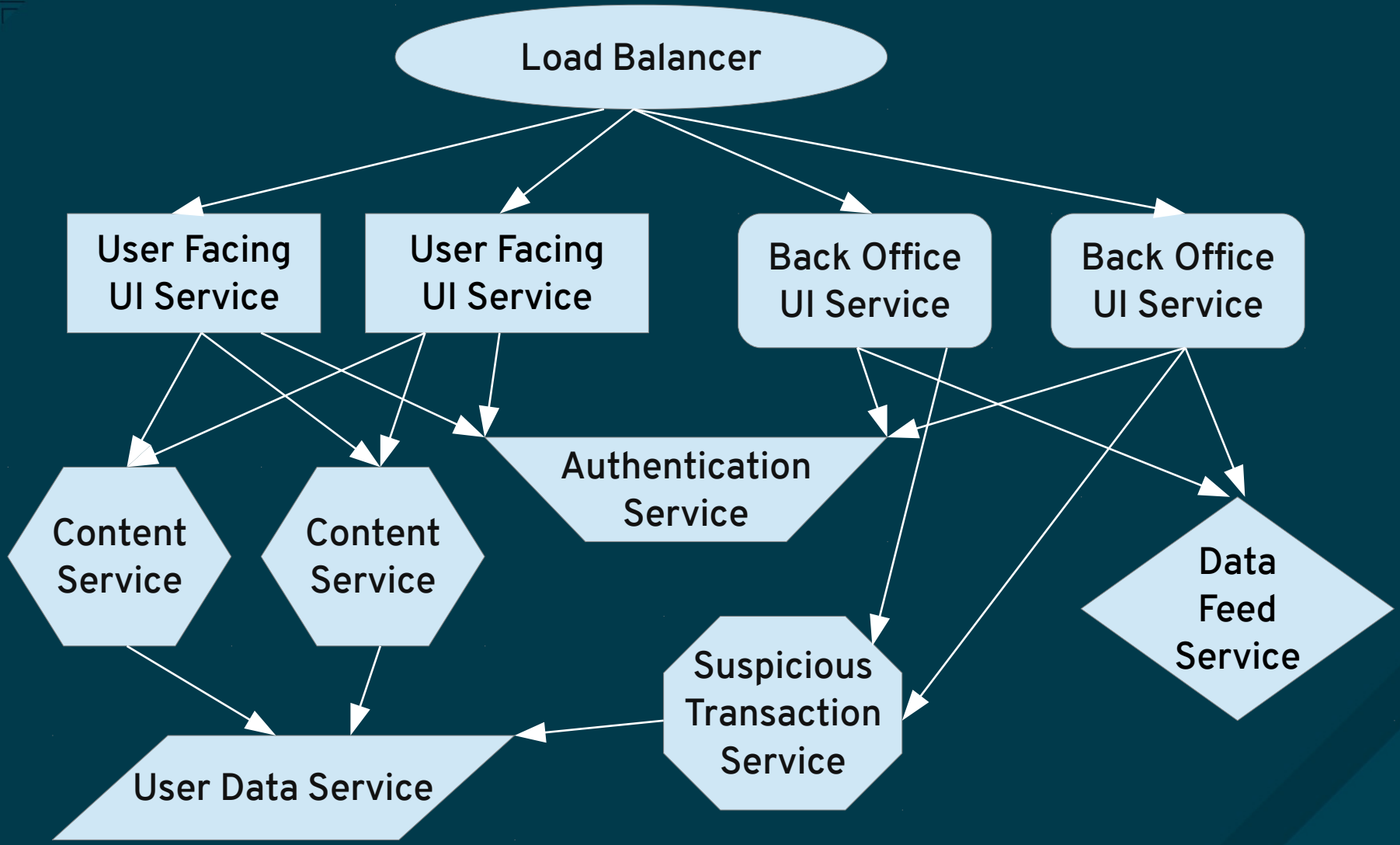
"In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently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

- Martin Fowler, ThoughtWorks

Monolithic Architecture



Microservice Architecture



Why Microservices

- Allow you to take advantage of recent innovations:
  - Linux Containers (Docker)
  - Continuous Integration
  - DevOps
  - The cloud
- Allow updates to be deployed faster, as each service can be updated individually

Wildfly Swarm

- Open Source project sponsored by Red Hat
- Based on the Wildfly Application Server
- Allows you to create microservices using standard Java EE API's using a mature runtime stack
- Allows you to pick the functionality that you require, and leave everything else

Wildfly Swarm Core Concepts

- Packages everything into a ‘fat jar’ application
- Includes just the bits of Wildfly that your application requires
- Composed of many ‘Fractions’ which map to specific server features
- You decide on the fractions you need, and only the required functionality will be bundled
- Provides additional features, such as Netflix open source components

Installation

WILDFLY

- Unzip Wildfly to server
- Configure Wildfly (add datasource etc)
- Install deployments into server

WILDFLY SWARM

- Copy jar file to server

Configuration

WILDFLY

- Configuration stored in XML file
- Managed via web console or CLI
  - System Administrators (not developers) most likely to be responsible for maintaining the server installation and configuration

WILDFLY SWARM

- Convention over configuration approach
  - Configuration can be programatic, XML based, or a mix
- Developers can choose exactly which config options to expose, and how to configure them
- Maps well to a DevOps approach

DevOps

- Wildfly Swarm has been designed to be dev ops friendly
- Simple installation
- Developers can choose which config options to support
- In general only config options relevant to the app will be exposed
- Single jar approach works well with Linux containers (Docker)

A word of caution

- Microservices are not a magic bullet
- You need to evaluate if they are right for your team
- Services may be simpler, but there is more complexity in how they interact and are managed
- Monolithic architecture is still a perfectly valid choice

Using Wildfly Swarm

Swarmifying a JavaEE application

```
<build>
  <plugins>
    <plugin>
      <groupId>org.wildfly.swarm</groupId>
      <artifactId>wildfly-swarm-plugin</artifactId>
      <executions>
        <execution>
          <id>package</id>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Build it

```
mvn package
```

Run it

```
mvn wildfly-swarm:run                        (or)
java -jar target/myapp-swarm.jar
```

Convention over configuration

- Based around sensible defaults
- Appropriate Fractions will be selected based on your applications dependencies
- A default configuration will be provided
- All this can be customized

Fractions

- Fractions provide a specific piece of functionality
  - May map directly do a Wildfly subsystem (e.g. JAX-RS)
  - Or bring in something else entirely (e.g. Keycloak, Ribbon)
- Fractions can provide deployments
  - Topology deployment gives access to the topology
- Fractions can alter user deployments
  - Keycloak fraction can be used to secure a deployment
- Fractions can be auto detected by the Swarm plugin based on the contents of your project, or can be explicitly enabled and configured

Fractions can be explicitly added

Adding a fraction to the pom will cause the plugin to include its functionality

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>jpa</artifactId>
</dependency>
```

The main() method

- The main() method is the entry point to a Swarm app
- If it is not provided Swarm uses a default main() method
- Can be used to configure the container and the deplyment

Take control with the main() method

```
public static void main(String[] args) throws Exception {

     Container container = new Container();

     JAXRSArchive deployment =
            ShrinkWrap.create(JAXRSArchive.class, "myapp.war");
     deployment.addClass(MyResource.class);
     deployment.addClass(NotFoundExceptionMapper.class);
     deployment.addAllDependencies();
     container.start().deploy(deployment);
```

```
<configuration>
  <mainClass>org.wildfly.swarm.examples.jaxrs.shrinkwrap.Main</mainClass>
</configuration>
```

Programatic configuration

As configuration is programatic the developer chooses what to expose.

The example configures the web worker pool from the command line

```
public static void main(String[] args) throws Exception {
    int webThreads = Integer.parseInt(args[0]);
    Container container = new Container();

    container.fraction(new IOFraction().worker("default", worker -> {
        worker.taskMaxThreads(webThreads);
    }));
    container.start();
}
```

Programatic configuration

Adding a data source

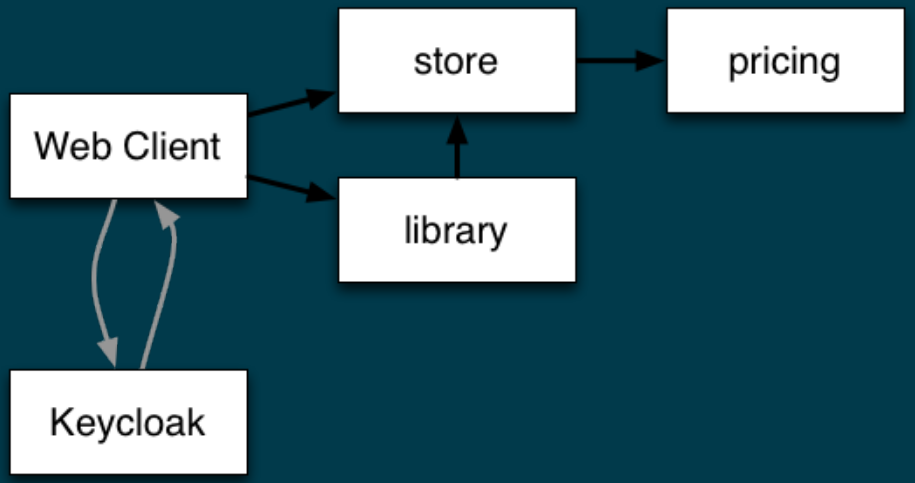
```
public static void main(String[] args) throws Exception {
    Container container = new Container();

    container.fraction(new DatasourcesFraction()
                    .jdbcDriver("h2", (d) -> {
                        d.driverClassName("org.h2.Driver");
                       d.xaDatasourceClass(
                            "org.h2.jdbcx.JdbcDataSource");
                        d.driverModuleName("com.h2database.h2");
                    })
                    .dataSource("MyDS", (ds) -> {
                        ds.driverName("h2");
                        ds.connectionUrl("jdbc:h2:mem:test");
                        ds.userName("sa");
                        ds.password("sa");
                    })
    );
```

Demo

Booker Demo

Simple micro services book store example

Booker is a book store that is comprised of the following services:



Keycloak

- Identity management server
- Swarm provides a Keycloak fraction to make integration easy
- Booker uses OAuth to authenticate users against Keycloak

Pricing Service

- JAX-RS endpoint that provides pricing for books
- Secured via Keycloak
- Discovered via the topology fraction

Pricing Service

Standard JAX-RS endpoint, charges logged in users $9 and other users $10

```
@Path("/")
public class PricingResource {
     @GET
     @Path("/book/{id}")
     @Produces(MediaType.APPLICATION_JSON)
     public Integer search(@PathParam("id") String id,
          @Context SecurityContext c) {
         KeycloakPrincipal p = (KeycloakPrincipal) c.getUserPrincipal();
        if ( p != null && p.getKeycloakSecurityContext() != null ) {
            return 9;
        }
        return 10;
    }
}
```

Pricing Service

Main method that sets up the service, and advertises it under “pricing”

```
public static void main(String... args) throws Exception {

    Container container = new Container();
    container.fraction(ContainerUtils.loggingFraction());

    JAXRSArchive deployment = ShrinkWrap.create(JAXRSArchive.class);
    deployment.addPackage(Main.class.getPackage());
    deployment.as(RibbonArchive.class).advertise("pricing");
    deployment.as(Secured.class);
    ContainerUtils.addExternalKeycloakJson(deployment);

    container.start();
    container.deploy(deployment);

}
```

Store Service

- In memory store of Project Gutenburg books
- Pricing service is invoked via Ribbon
- Uses JAX-RS async processing to efficiently handle pricing requests

Store Service

Using Ribbon to access the pricing service and expose it over CDI

```
@ResourceGroup(name="pricing")
public interface PricingService {

    @TemplateName("get")
    @Http(
            method = Http.HttpMethod.GET,
            uri = "/book/{id}"
    )
    RibbonRequest<ByteBuf> get(@Var("id") String id);
}

public class ServicesFactory {
    @Produces
    @ApplicationScoped
    public static PricingService getInstance() {
        return SecuredRibbon.from(PricingService.class);
    }
}
```

Store Service

Use JAX-RS async processing to service request

```
@Inject
private PricingService pricingService;

@GET
@Path("/book")
@Produces(MediaType.APPLICATION_JSON)
public void get(@Suspended final AsyncResponse asyncResponse,
    @QueryParam("id") String id) {
    Book book = this.store.get(id);
    Observable<ByteBuf> obs = pricingService.get(id).observe();
    obs.subscribe(
            (result) -> {
                int price =
Integer.parseInt(result.toString(Charset.defaultCharset()));
                book.setPrice(price);
                asyncResponse.resume(book);
            },
  ...
```

Library Service

- Uses JPA to store a record of purchased books
- Uses the store service to retrieve book details

Library Service

JPA and Datasources are configured when creating the Container

```
public static void main(String... args) throws Exception {
    Container container = new Container();
    container.fraction(new JPAFraction()
            .inhibitDefaultDatasource()
            .defaultDatasource("LibraryDS"));

    container.fraction(new DatasourcesFraction()
          ...
            .dataSource(new DataSource("LibraryDS")
.connectionUrl("jdbc:h2:./library;DB_CLOSE_ON_EXIT=TRUE")
                    .driverName("h2")
                    .jndiName("java:/LibraryDS")
                    .userName("sa")
                    .password( "sa" )));
```

Web Front End

- Mostly static deployment (react.js based)
- Uses the topology fraction to proxy backend services
- Topology information is provided to the front end by the topology webapp

Web Front End

Proxies backend services using the topology fraction

```
public static void main(String... args) throws Exception {
    Container container = new Container();
    container.fraction(ContainerUtils.loggingFraction());

    System.setProperty(TopologyProperties.CONTEXT_PATH, "/topology-
webapp");
    TopologyWebAppFraction topologyWebAppFraction = new
TopologyWebAppFraction();
    topologyWebAppFraction.proxyService("store", "/store-proxy");
    topologyWebAppFraction.proxyService("pricing", "/pricing-proxy");
    topologyWebAppFraction.proxyService("library", "/library-proxy");
    container.fraction(topologyWebAppFraction);
    container.start();

    WARArchive war = ShrinkWrap.create(WARArchive.class);
    war.staticContent();
    war.addAllDependencies();
    war.addModule("org.wildfly.swarm.container");
    container.deploy(war);
```

Questions?