ConfigJSR and Friends

Mark Struberg,
RISE GmbH,
Apache Software Foundation,
INSO TU Wien

About me

- Mark Struberg
- 25++ years in the industry
- Apache Software Foundation member
- struberg [at] apache.org
- RISE GmbH employee
- TU-Wien / INSO researcher
- Committer / PMC for Apache OpenWebBeans, TomEE, Maven, OpenJPA, DeltaSpike, JBoss Arquillian, ...
- Java JCP Expert Group member and spec lead
- MicroProfile Spec Author
- Twitter: @struberg

A little bit of History

Evolution

- openwebbeans.properties (2008/9)
- Apache MyFaces CODI config (2010)
- Apache DeltaSpike Config (2011)
- Apache Tamaya (2014)
- MicroProfile Config (2016)
- ConfigJSR (2017)

The state of ConfigJSR

- Started as official JCP JSR-382
- Was planned as successor to mp-config
- Might be moved to Jakarta-EE
- But content will not change that much

Relation to microprofile-config

- We started from microprofile-config-1.2
- Some of the features introduced in ConfigJSR now got ported back to mp-config
  - We try to keep the diffs as minimal as possible
- Apache DeltaSpike also still exists and contains most of the features.

Proposed Timeline

- We are late…
- 3 options:
  - Graduate as JCP JSR
  - Graduate as JSR under JakartaEE
  - Merge again with mp-config
- In any case there **will** be a Standard for Config!

Resources

- ConfigJSR:
  - https://github.com/eclipse/ConfigJSR
  - https://struberg.net/public
- Microprofile-config:
  - https://github.com/eclipse/microprofile-config
- DeltaSpike-config:
  - https://github.com/apache/deltaspike
  - http://deltaspike.apache.org/documentation/configuration

Design Principles

Problems to solve

- Change configuration without rebuilding your app.
- Especially for reusable libraries!
- We don't like to rebuild WAR, EAR, etc
  - Changing WEB-INF/web.xml etc is a no-go!
- Adopting to different deployment scenarios without having to change your application or lib.
- Provide meaningful defaults.
- Allow Ops teams to tweak those.

Design Principles 1/2

- Easy to use.
- Straight and clear rules and design principles.

Strict separation of

- Application-side usage of configured values.
  - Using a configured value should be straight forward.
- Ops-side providing of configured values.
  - Integrating different config environments must have no impact on the usage.
- Easy extending via SPI allows for an eco system

Design Principles 2/2

- Configured Values strictly String/String based
  - String is the least common denominator
- Allow ‘override’ of configured values
- Pluggable Converters
- Namespacing similar to Java Package names

Using Configuration Programmatically

```
Config cfg = ConfigProvider.getConfig();
String url = cfg.getValue(“myapp.url”, String.class);
int port = cfg.getValue(“myapp.port”, int.class);
```

Using Configuration via Injection

```
@Inject
@ConfigProperty(name=”myapp.url”)
private String url;

@Inject
@ConfigProperty(name=”myapp.port”)
Private Provider<Integer> port;
```

Accessing a Config

- One `Config` per ‘Application’
- Automatic Discovery

```
Config cfg = ConfigProvider.getConfig();
@Inject Config cfg;
```

  - ConfigSources, Converter, etc get picked up via `java.util.ServiceLoader`

- ConfigBuilder

```
Config cfg = ConfigProviderResolver.instance()
    .getBuilder()
    .addDefaultSources()
    .withSources(new MyDbConfigSource())
    .withConverter(new DogConverter())
    .build();
```

  - Optionally register manual Config within Application

```
ConfigProviderResolver.instance().registerConfig(cfg, classLoader);
```

ConfigSource

Technical Approach

- Default configuration
- Plus optional plugin configuration
- Plus optional external configuration
- From a Database
- From Kubernetes
- From Zookeeper
- From a remote git repo
- Plus, plus, plus, ...
- -> lots of different `ConfigSources`

How does it work?

- Pick up information from various places (`ConfigSource`)
- Ordinal based (`config_ordinal`=..):
- higher number -> more important
- Extensible, register your own `ConfigSource`

Default ConfigSources

- System properties (ordinal=400)
- Environment properties (ordinal=300)
- /META-INF/javaconfig.properties (ordinal=100)
  - /META-INF/microprofile-config.properties (ordinal=100)
  - /META-INF/apache-deltaspike.properties (ordinal=100)

- NO JSON support out of the box!
- NO yaml support out of the box!

The ConfigSource SPI

```
public interface ConfigSource {
    int getOrdinal();
    String getName();

    String getValue(String propertyName);

    Map<String, String> getProperties();
    default boolean isScannable() { return true;}
}
```

Writing own ConfigSources

- Implement `javax.config.spi.ConfigSource`

```
public class org.acme.MyDbConfigSource implements
ConfigSource {
```

- Create a file META-INF/services/javax.config.spi.ConfigSource
- Add the fully qualified classname into this file

```
org.acme.MyDbConfigSource
```

ConfigSourceProvider

- Can pick up multiple dynamic ConfigSources
- Useful if you want to load custom file names
  - property files
  - JSON files
  - yaml files
- Again `java.util.ServiceLoader` based

Writing own Converter

- Implement `javax.config.spi.Converter`

```
public class DuckConverter implements Converter<Duck> {
    @Override
    public Duck convert(String value) {
        return new Duck(value);
    }
}
```

- Register Converter via `java.util.ServiceLoader` mechanism

Closing Resources

- Auto-discovered Config gets shut down with the Application if they implement `AutoCloseable`
  - `ConfigSource`
  - `Converter`
- Only use them in one Config!
  - If you want to share e.g. a DBConfigSource for all apps on your server, then simply use facading.

New Features

Enhancements over mp-config-1.0

- Pick up configuration changes at runtime
- Active notification on changes
- Support for aggressive caching
- Variable Replacement
- Guaranteed Consistency

ConfigAccessor - caching

```
ConfigAccessor<String> host =
    cfg.access(“doc.archive.host”, String.class)
        .cacheFor(Duration.ofHour(2))
        .withDefault(“localhost”)
        .build();
String h = host.getValue();
```

Config Change Notification

```
public class MyConfigSource implements ConfigSource {
  private Map<String, String> properties;
  private Consumer<Set<String>> onAttributeChange;
  ChangeSupport setAttributeChangeCallback(
      Consumer<Set<String>> callback) {
    this.onAttribeChange = callback;
    return ChangeSupport.SUPPORTED;
  }

  public static MyConfigDbWorker extends Thread {
    public void run() {
      do {
      Map<String, String> newProps = readAllPropsFromDb();
      Set<String> changed = diff(this.properties, newProps);
      if (!changed.isEmpty()) {
        this.properties = newProps;
        onAttributeChange.accept(changed);
      }
      } while (sleep5s())
    }..
```

Variable Replacement

```
ConfigAccessor<String> host =
    cfg.access(“doc.archive.search”, String.class)
        .evaluateVariables(true)
        .build();
host.getValue();

doc.archive.host=http://mydocarchive:8774
doc.archive.search=${doc.archive.host}/docs/search
```

ConfigSnapshot

```
Config cfg = ConfigProvider.getConfig();

private ConfigAccessor<String> hostAccessor
  = cfg.access(“doc.archive.host”,
String.class)...build();
private ConfigAccessor<Integer> portAccessor
  = cfg.access(“doc.archive.port”, Integer.class)...build();
…

public byte[] loadDoc(String docId) {
  ConfigSnapshot snap = cfg.snapshotFor(hostAccessor,
portAccessor);
  String host = urlAccessor.getValue(snap);
  Integer port = urlAccessor.getValue(snap);

  return loadDocsFromUrl(host, port, docId);
}
```

Best Practice

Tips!

- Check our TCK, it contains many samples!
- JavaDocs are up2date and well maintained.

Default Values

- Keep default values within your app!
  - META-INF/javaconfig.properties
  - META-INF/microprofile-config.properties

Questions?