Pelix/iPOPO
an OSGi framework for Python applications

Thomas Calmant$^{1,2}$  Joao Claudio Americo$^1$  Olivier Gattaz$^2$
Didier Donsez$^1$  Kiev Gama$^3$

$^1$ Université Joseph Fourier
Grenoble I
LIG/INRIA SARDES/ERODS Team
Grenoble, France

$^2$ isandlaTech
Grenoble, France

$^3$ Recife Center for Advanced Studies and Systems (CESAR)
Recife, Brazil
Outline

- Why Python?
- Pelix
- iPOPO
- Usage: COHORTE
- Limitations
- Conclusion and Perspectives
• Scripting Language
  • Released in 1990, designed by Guido Van Rossum
  • Interpreted, JIT (Just-In-Time), AOT (Ahead-of-Time)
  • Intends to be a bridge between Shell scripts and C
  • Developers: Sysadmins, Web, Scientists, …
  • Python Zen: description of good practices
• Large applications
  • Youtube, OpenERP, Shinken, Dice Battlelog, Zope, Blender, …
Why choose Python?

- **Pros**
  - Highly dynamic
  - Fast development
  - No compilation
  - Portable
  - Easy to maintain
  - Low-level methods access
  - Low imprint on system

- **Cons**
  - Slow
  - Errors detected at run time (no compilation)
Pelix (0.5.3 August 1\textsuperscript{st}, 2013)

- OSGi implementation in Python
  - Similar API
- Module loading, update, uninstall
- Services registry
- Same principles:
  - Bundle: Python module
  - Bundle Context: to access the framework
  - Service: instance of a Python object
  - ServiceRegistration / ServiceReference
  - LDAP filters on service properties
  - Bundle and service listeners
  - Extra functional properties handling
# Start a framework
framework = pelix.FrameworkFactory.get_framework(properties)
framework.start()

# Get its context
context = framework.get_bundle_context()

# Install and start bundles
required_bundles = ('bundle1', 'my.package.bundle')
for bundle_name in required_bundles:
    bid = context.install_bundle(bundle_name)
    context.get_bundle(bid).start()

# Register a service
foo = SpecificationImpl()
registration = context.register_service('my.specification', foo, 
                                      {'prop1': True, 'prop2': [0, 2]})
IPOPO (0.5.3 August 1\textsuperscript{st}, 2013)

- A Service-Oriented Component Model with Dependency Injection
  - Equivalent to Apache Felix iPOJO
  - Components are bound by service references: loose coupling
  - Handles dynamicity: late binding
- Installed as a Pelix bundle
- Allows the developer to focus on business logic
from pelix.ipopo.decorators import *
import pelix.ipopo.constants as constants

@ComponentFactory(name='MailCheckerFactory')
@Instantiate('MailChecker')
@Property('_name', constants.IPOPO_INSTANCE_NAME)
@Property('protocols', 'mail.protocols', ['imap', 'pop'])
@Requires('_accounts', 'demo.credentials.Account', aggregate=True)
@Requires('_logger', 'demo.utilities.Logger', optional=True)
@Provides(specifications='demo.mail.Checker')
class MailChecker(object):
    def __init__(self):
        # Declare injected members, for readability
        self._name = None
        self._accounts = None
        # ...
        self.thread = None

    @Validate
def validate(self, context):
        # Component validation
        if self._logger:
            self._logger.debug('%s started.', self._name)
        # Start internal logic
        # self.thread = threading.Thread(...)

    @Invalidate
def invalidate(self, context):
        if self._logger:
            self._logger.debug('%s stopped.', self._name)
        # Stop internal logic
        # kill(self.thread)
Usage: COHORTE

- A project from isandlaTech
- **COHORTE** provides a framework to build dynamic, distributed and mixed-language applications
- **COHORTE** isolates dynamically the error-prone components
- Communications done with **COHORTE** OSGi Remote Services
  - Based on JSON-RPC
  - Compatible OSGi and Pelix implementations
- Components are distributed in different processes and hosts
  - Low-level processes monitoring done in Python
  - High-level logic done in Python
  - Mixed iPOJO (Java) and iPOPO (Python) components for internal and business logic
Limitations

- Only one framework instance at a time in a process
- No implicit concurrency control on injected dependencies
- No module version control
Conclusion & Perspectives

- Fully-dynamic component model for Python
- Validated in a mixed Java-Python application for M2M
- Available at https://github.com/tcalmant/ipopo

- Perspectives
  - Artifacts, dependencies, repositories
  - Unify native component wrapping (CPU, GPGPU)
  - Target embedded platforms (M3 Cortex [Usenix 2012], ...)

11
Q & A

Thanks for your attention