

Rapid Dynamic Modular Application Development with Impala

The simple approach to dynamic modules with Spring

Phil Zoio

Creator of Impala

philzoio@realsolve.co.uk

SPEAKER INFO

- ▶ **Java developer, 10+ years experience**
- ▶ **Active open source developer and blogger**
- ▶ **Creator of Impala**

Today's Talk

- ▶ **What is Impala, and why does it exist?**
 - ▶ What problems does it solve?
- ▶ **Applying Impala**
 - ▶ Web application development
 - ▶ Integration testing
- ▶ **Impala and OSGi**
 - ▶ Comparison with OSGi-based alternatives
- ▶ **Roadmap and summary**



What is Impala?



Source: www.animalpicturesociety.com



Source: <http://commons.wikimedia.org/>



What is Impala?

Impala is an **open source dynamic module framework** for Java-based web applications, based on Spring.

Focus on **simplicity** and **productivity**.



What real-world problems does it solve?

How do you ...

- ▶ stop your application from mushrooming in complexity as it grows large?
- ▶ maintain high developer productivity, even as your application grows large?
- ▶ make it easy to write integration tests which are fast to run, no matter how large the application?
- ▶ make it easy to configure complex applications for different environments and requirements?

Impala is built on Spring

▶ **Great facilities for bean creation**

- ▶ dependency injection keeps code cleaner
- ▶ “wiring” of beans external to code
- ▶ beans managed within an `ApplicationContext`

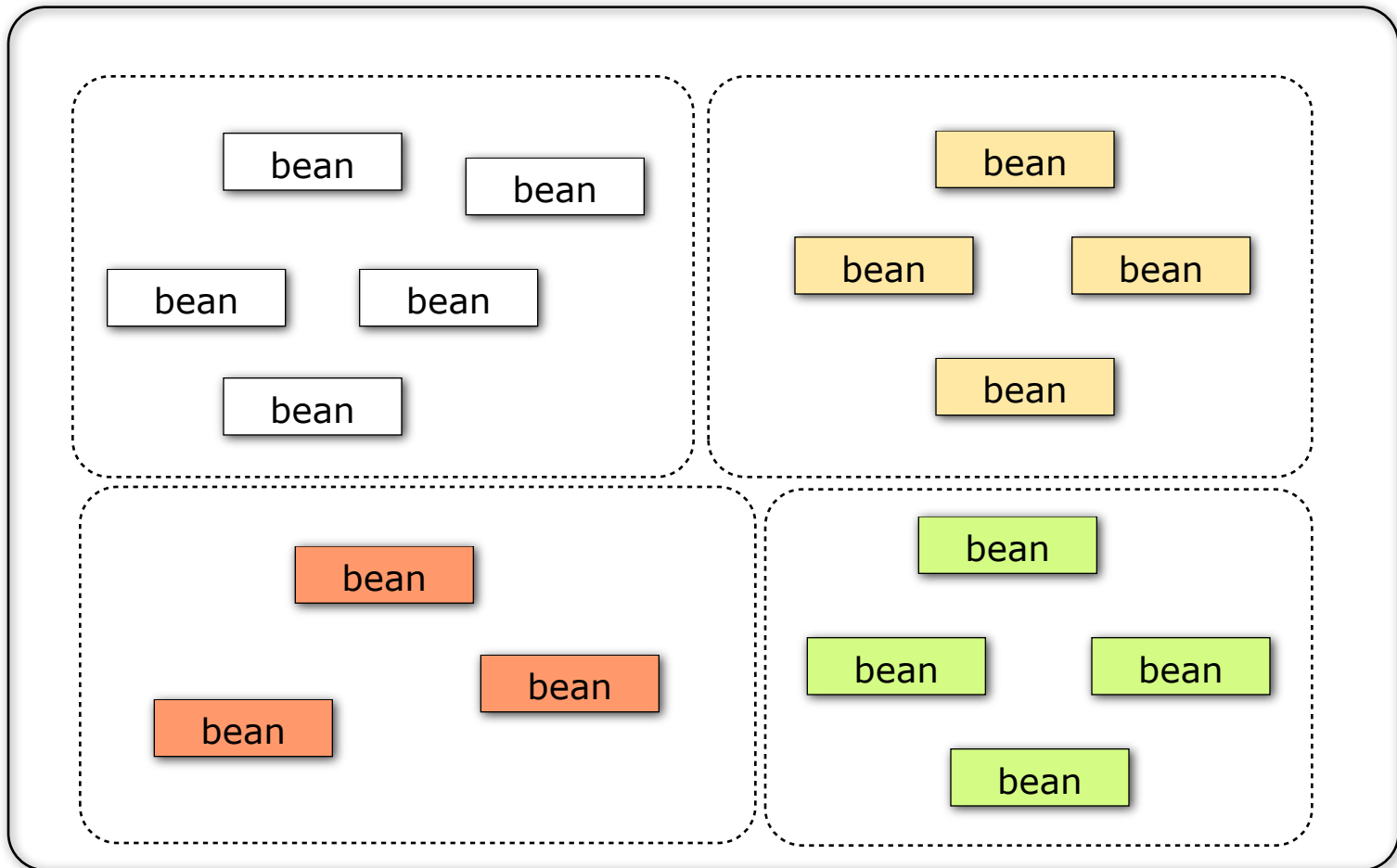
BUT managing wirings can get tricky

- ▶ Single bean address space within `ApplicationContext`
- ▶ Beans splittable by file
- ▶ No other grouping abstraction



ApplicationContext beans

Beans in ApplicationContext share address space and class loader



The typical Spring project ...

Is large ...

- ▶ Thousands of classes
- ▶ Numerous functional areas
- ▶ Hundreds (even thousands) of bean definitions

Becomes increasingly difficult to

- ▶ Tailor application wirings for different environments
- ▶ Keep integration tests simple to write and fast to run
- ▶ Maintain fast build/deploy/test cycles
- ▶ Keep code base free of incidental complexity
- ▶ Maintain development productivity

The solution to these issues is not ...

- ▶ Autowiring
- ▶ XML namespaces
- ▶ Annotation-based dependency injection
- ▶ Class path component scanning
- ▶ Code generation

The solution is **dynamic modules**

Spring's ApplicationContext

- ▶ Well defined life cycle
- ▶ Good class loader management
- ▶ Plenty of programmatic hooks
- ▶ Can be arranged in hierarchy

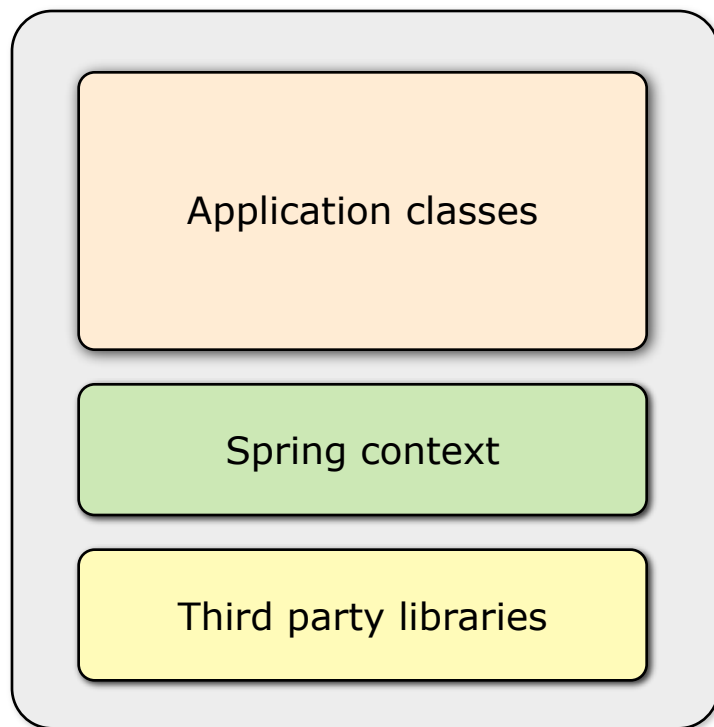
Makes ideal choice as unit of modularity

Both for Impala, and for Spring OSGi projects

Architecture comparison

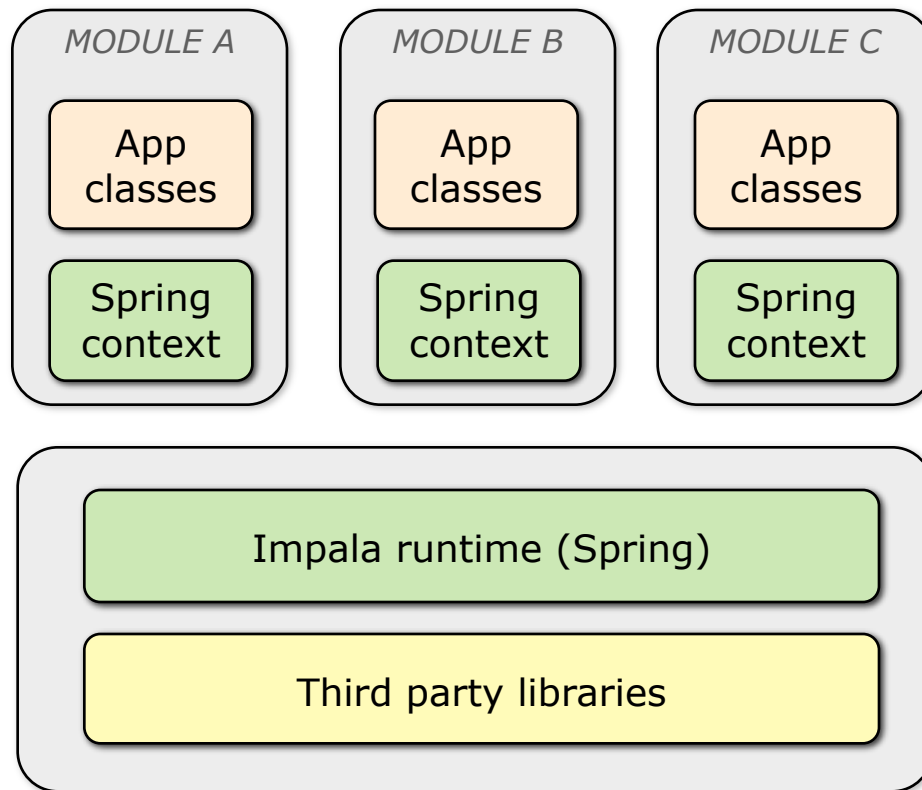
Traditional Java application

Single class loader for application



Impala application

Class loader and spring context per module



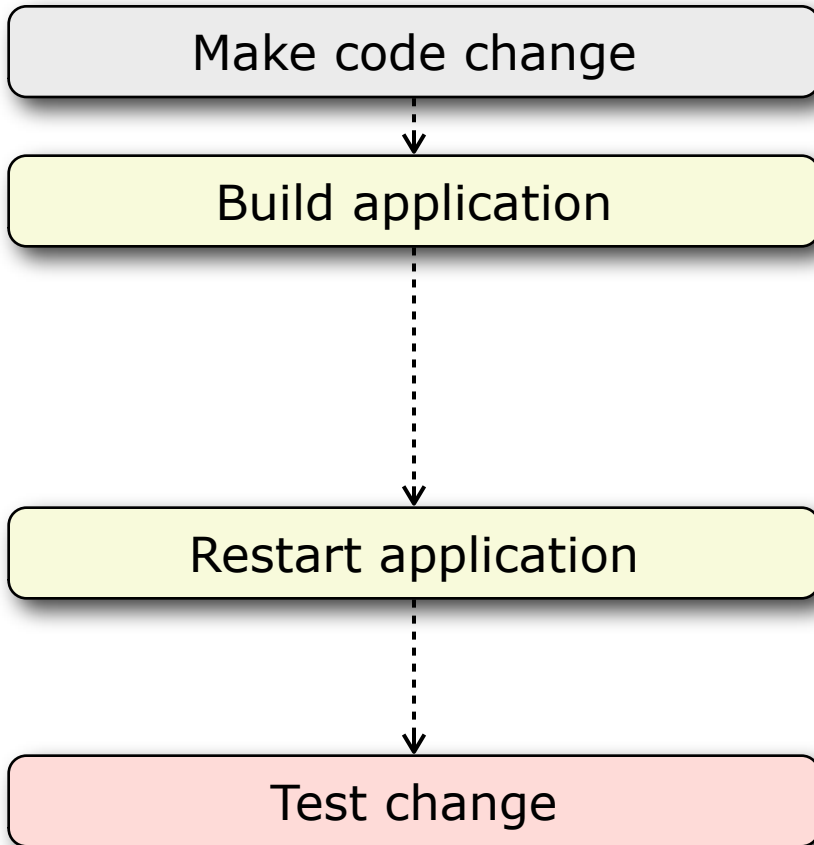


DEMO

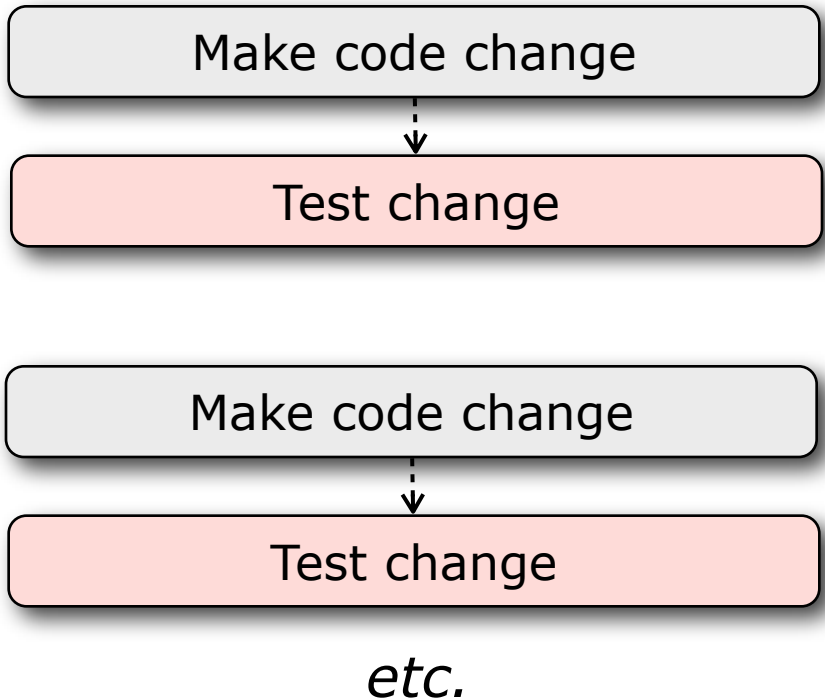


Productive Build/deploy/test cycle

Traditional application



Impala application



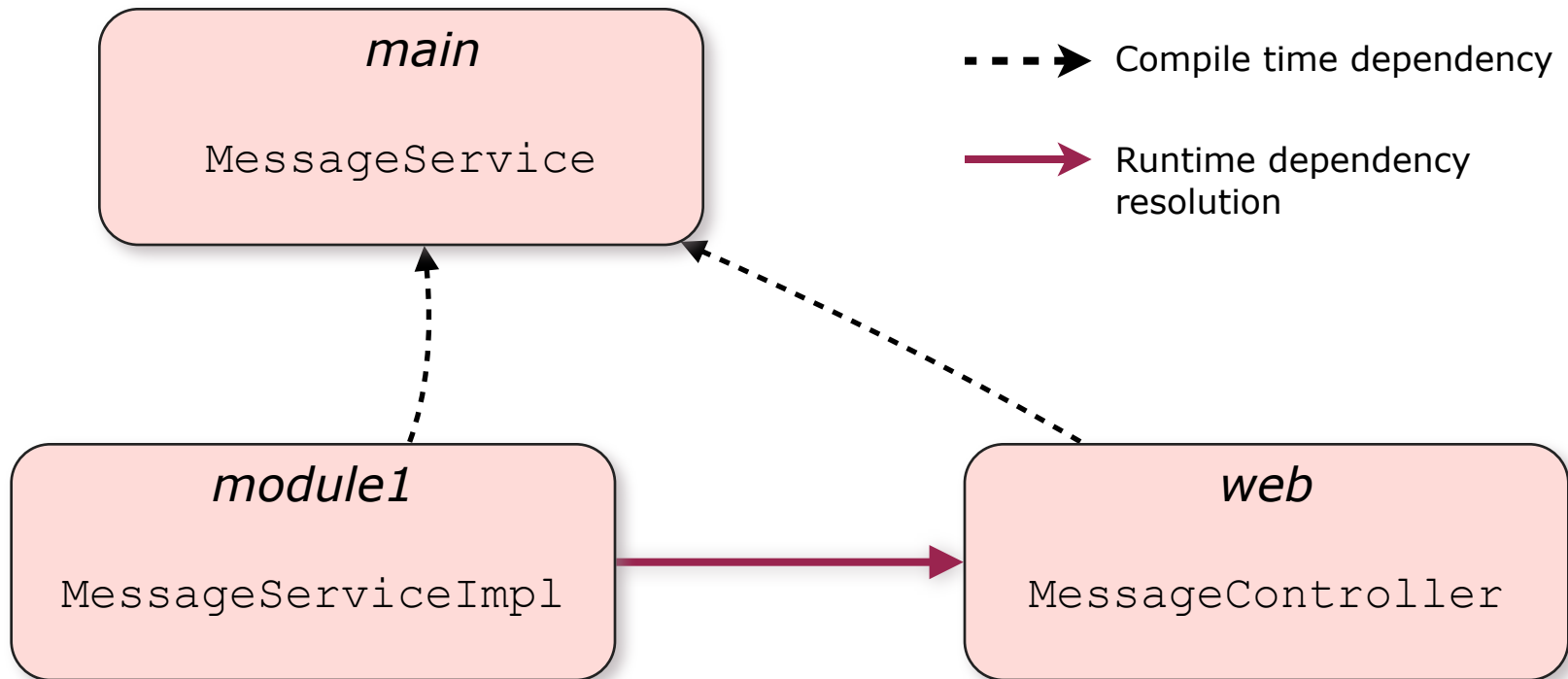


Impala “Hello World” explained



Impala "Hello World"

The solution - traditional static approach

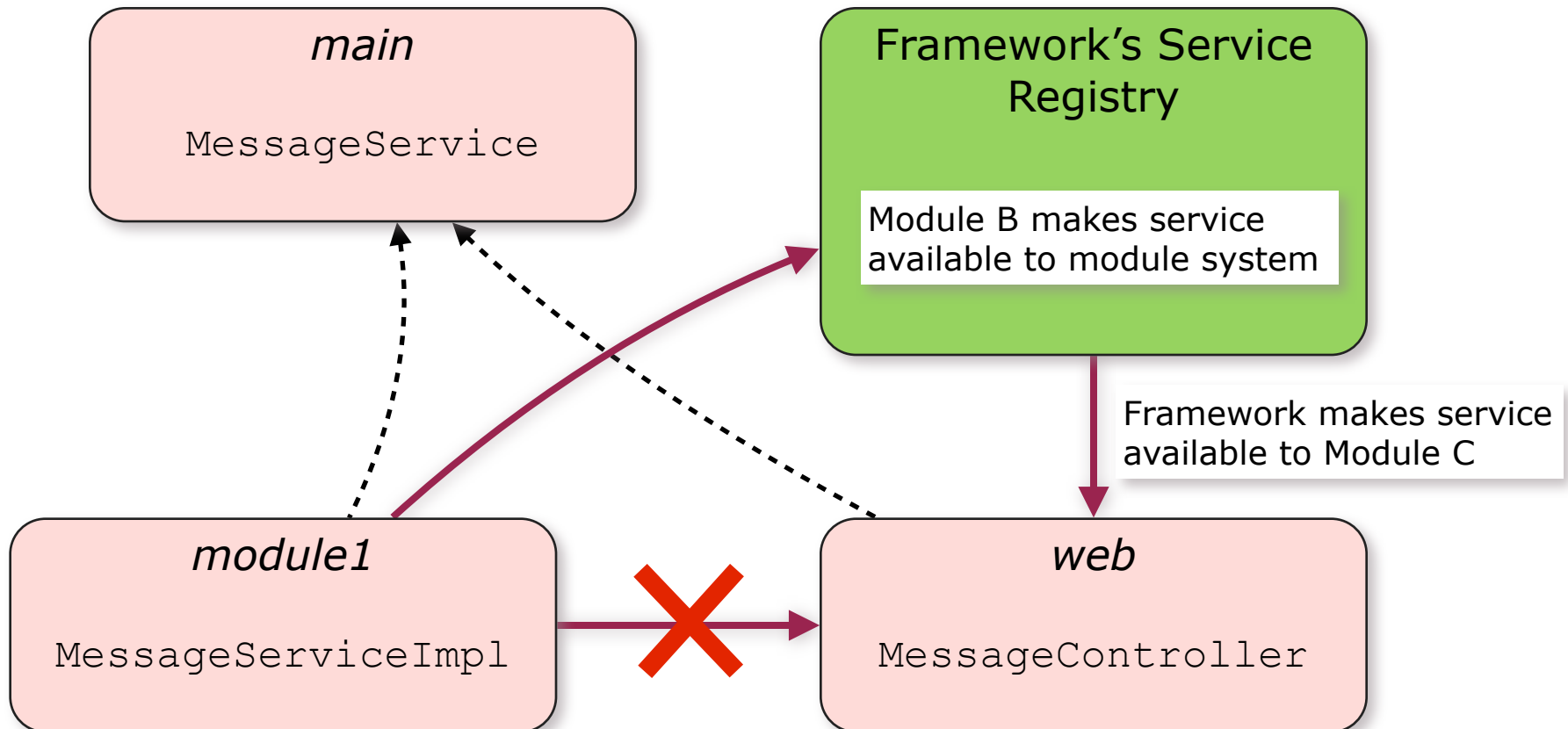


Module C gets dependency directly from module B.



Impala "Hello World"

The solution with Impala



Benefits of dynamic modules

By breaking the direct dependency between service and client:

- ▶ Implementation or client can be **swapped or reloaded** dynamically
- ▶ Implementation changes **transparent to client**
- ▶ **Decoupling** reduces overall system complexity as application grows



Applying Impala - Integration Testing

Integration testing

- ▶ Good suite of integration tests is critical
- ▶ Integration tests can be painful to write in vanilla Spring applications

Impala makes integration tests easier to write and faster to run

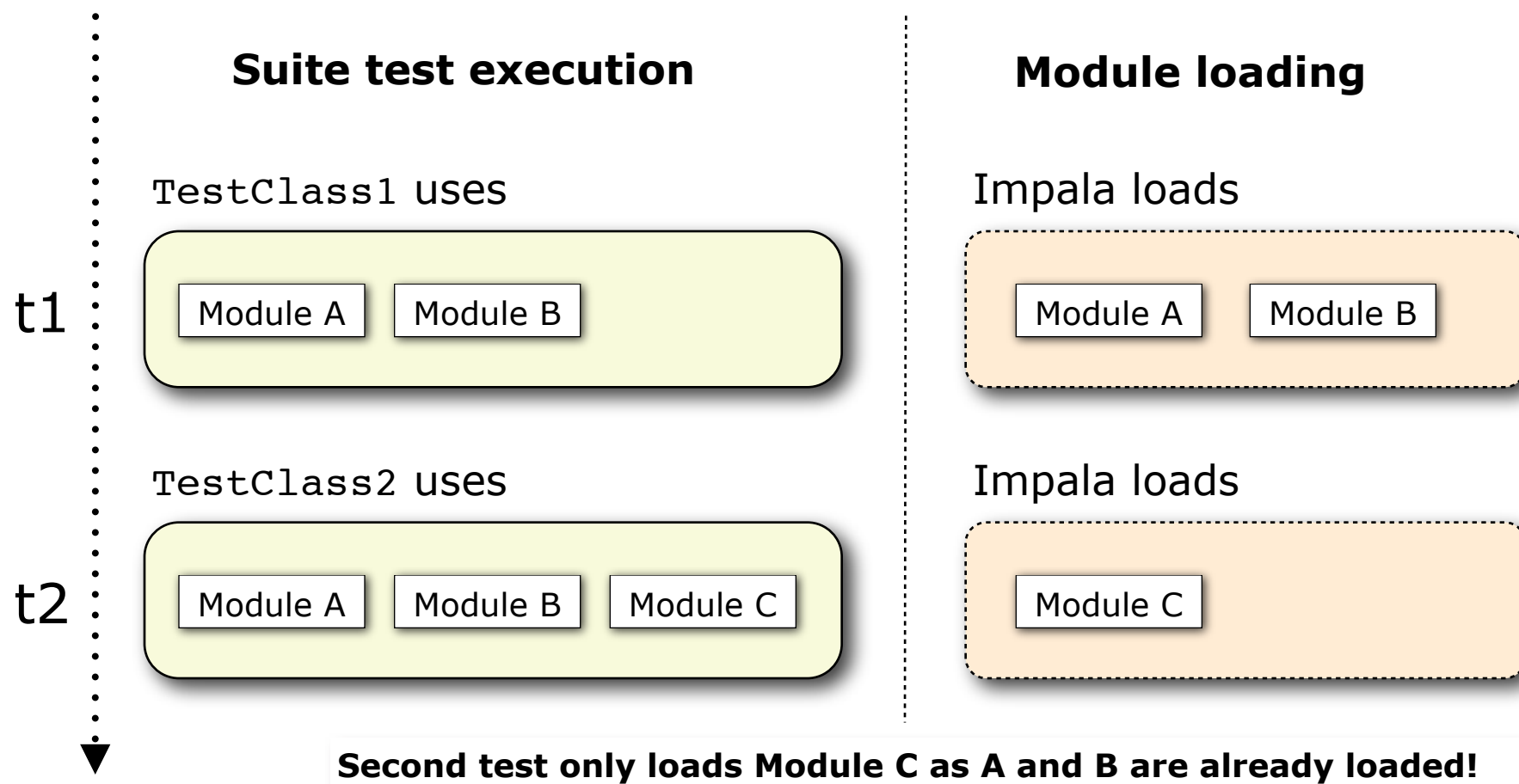
- ▶ dependencies can be expressed through modules
- ▶ incremental module loading as suite executes
- ▶ interactive test runner



DEMO



Incremental test module loading



With Impala it is much easier to ...

- ▶ Tailor application wirings for different environments
- ▶ Keep integration tests simple to write and fast to run
- ▶ Maintain fast build/deploy/test cycles
- ▶ Keep code base free of incidental complexity
- ▶ Maintain development productivity

Even for large applications!



Impala and OSGi

The SpringSource approach

The Spring OSGi marriage

- ▶ Spring Dynamic Modules
- ▶ SpringSource dm Server

Re-architecting the enterprise Java landscape

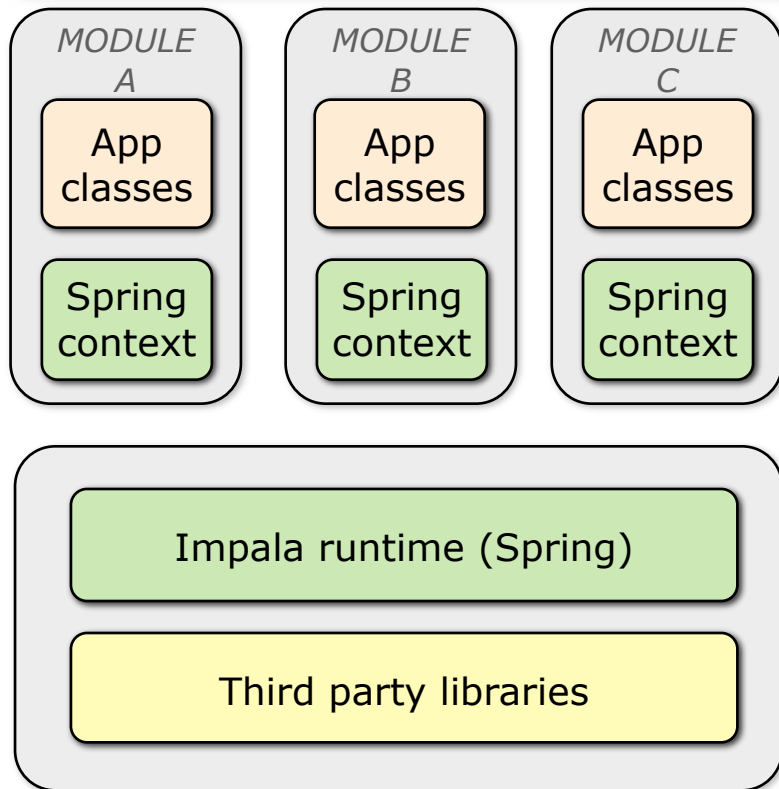
- ▶ Raft of new standards (Blueprint, OSGi EEG)
- ▶ Will developers be able to keep up?



Architecture comparison

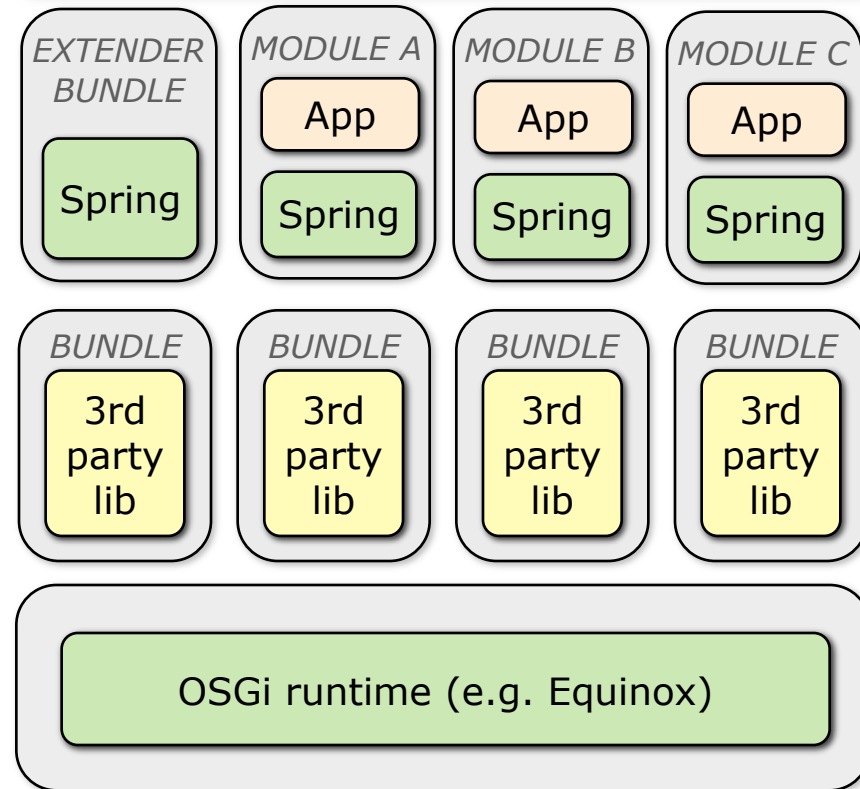
Impala

Impala loads application modules



OSGi-based (Spring DM)

Extender loads Spring for app bundles



Pros and cons of OSGi

PROS

- ▶ Third party library versioning
- ▶ Very precise control over module boundaries
- ▶ The most widely adopted modularity standard

CONS

- ▶ Extra work to make sure library jars are OSGi compliant
- ▶ Compatibility issues with existing libraries (e.g. TCCL use)
- ▶ Requires significant effort to learn and apply
- ▶ Integration testing is difficult

Reasons for preferring Impala

- ▶ Simpler, more familiar environment
- ▶ Self contained - no reliance on third party
 - ▶ runtime infrastructure
 - ▶ build support
 - ▶ tooling
- ▶ Works with third party libraries from public Maven repo
- ▶ Easier to write integration tests
- ▶ More lightweight

However, prefer OSGi alternatives if ...

- ▶ You're already committed to the OSGi platform
- ▶ You need modularity and versioning for third party libraries
- ▶ Standards compliance is important for your project or organization
- ▶ You want big vendor support for your application



Roadmap and Summary

Roadmap

Focus on 1.0 final release

- ▶ current release 1.0 RC1
- ▶ feature complete
- ▶ final release planned for end of 2009

Post 1.0 Roadmap

Definite/Probable

- ▶ more pre-canned web framework integration
- ▶ better Maven support
- ▶ administration console application
- ▶ JUnit4 interactive test runner
- ▶ support for IntelliJ/Netbeans-based apps
- ▶ Support for modules in Scala/Groovy

Post 1.0 Roadmap

Potential

- ▶ more complete OSGi support
- ▶ Grails-style plugin system
- ▶ support for modular Grails applications
- ▶ new build system (possibly based on Gradle)
- ▶ TestNG interactive test runner
- ▶ Portlet support
- ▶ Eclipse plugin

Parting thoughts

- ▶ modularity is critical to **managing complexity** and **maintaining productivity** for large applications
- ▶ Impala offers the **benefits of modularity** while remaining **more simple than you'd expect**
- ▶ Impala provides **practical solutions** to real problems

Why not help?

- ▶ try Impala out
- ▶ use it in your next project
- ▶ contribute to Impala, and help it reach its potential

Further information

Home page and issue tracker

<http://www.impalaframework.org/>

<http://code.google.com/p/impala/>

Project blog

<http://impalablog.blogspot.com/>

User mailing list

http://groups.google.com/group/impala_group/



Questions

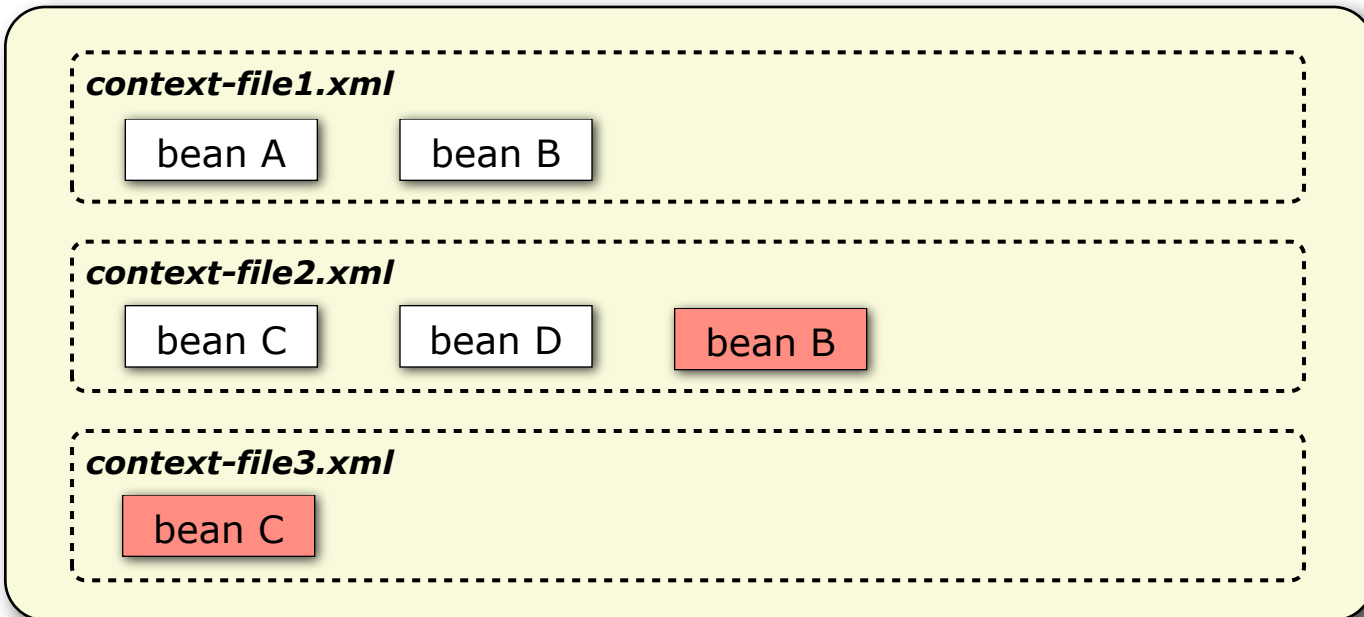


Extra Slides

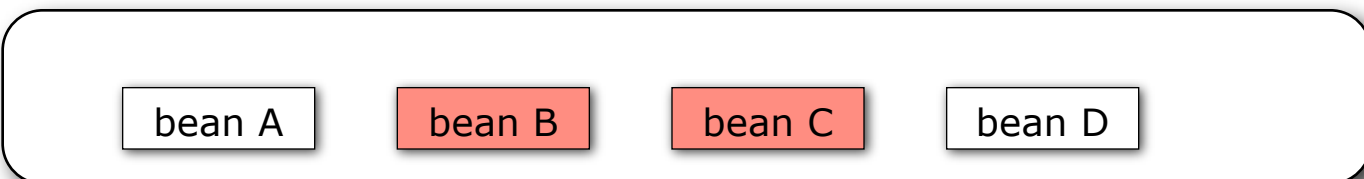


Spring's ApplicationContext

Bean definitions - static representation



Translates to runtime representation





Some questions

Q: How do you configure the functionality which you deliver in a web application to meet the needs of specific environments?

A: With difficulty, probably with various commented out sections in the *web.xml*, or with wildcards.

```
<context-param>
  <param-name>contextConfigLocation
  </param-name>
  <param-value>
    context-jdbc.xml
    context-domain.xml
    context-service.xml
    context-billing.xml
    context-security.xml
    ...
    context-notification.xml
    etc.
    plugin-context-*.xml
  </param-value>
</context-param>
```

Comment this out if
notification not needed

Some questions

Q: How do you write robust, fast-loading integration tests which only load the functionality you need?

A: With difficulty. One technique is to add dummy bean definitions to "mock out" unneeded ones.

```
String[] locations = {  
    "applicationContext-jdbc.xml",  
    "applicationContext-domain.xml",  
    "applicationContext-service.xml",  
    "applicationContext-billing.xml",  
    "applicationContext-security.xml",  
    "applicationContext-messaging-mock.xml",  
    "applicationContext-notification-mock.xml"  
};
```

Mocks for unneeded
bean definitions

```
ApplicationContext context =  
new ClassPathXmlApplicationContext(locations);
```

Some questions

Q: Maximize productivity by dynamically reload only portions of the application when needed?

A: Simply not possible. Any code change requires a full reload.

Remember what Spring is good at ...

- ▶ Keeping the code clean
(through dependency injection)
- ▶ AOP and infrastructure support
(transactions, remoting, etc)
- ▶ Some great APIs



Module reload example

1. Module metadata loaded

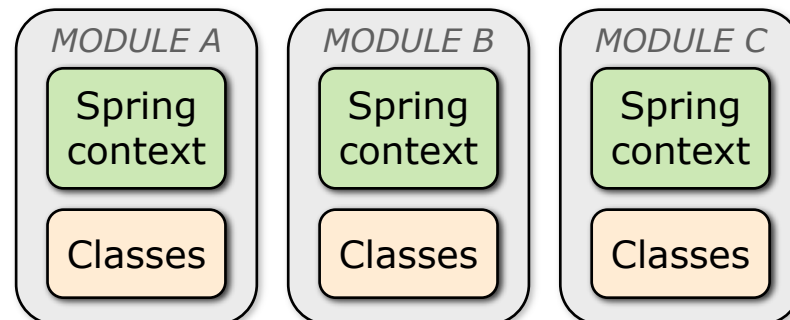
module.properties
moduledefinitions.xml
ModuleDefinitionSource

2. Module operations
determined

Unload D C B A
Load A B C

3. Modules (un)loaded in order

- ▶ class loader for module created
- ▶ application context loaded
- ▶ services exported to and imported from service registry as required





Applying Impala - Web Application Setup

Impala Web Application Setup

Familiar WAR layout

- ▶ Third party libraries in */WEB-INF/lib*
- ▶ Application modules in */WEB-INF/modules*

Configuration files

- ▶ *module.properties* for within-module configuration
- ▶ *moduledefinitions.xml* for composing the application
- ▶ *impala.properties* for configuring Impala runtime
- ▶ `ImpalaContextLoaderListener` in *web.xml*

Impala setup

module.properties

- ▶ Specifies properties for module
- ▶ Found per module on module class path

```
parent=example  
type=web_root  
#config-locations=web-context.xml
```



Impala setup

moduledefinitions.xml

- Specify collection of modules for application, plus overrides

```
<root>  
  <names>  
    main  
    module1  
    web  
  </names>  
</root>
```



Simply name modules
to load




Impala setup

impala.properties

- Sets built-in configuration options for Impala runtime

```
application.version=1.0-SNAPSHOT
auto.reload.modules=true
auto.reload.monitoring.type=stagingDirectory
use.touch.file=true
session.module.protection=true
partitioned.servlet.context=true
expose.mx4j.adaptor=true
jmx.adaptor.port=8003
extra.locations=META-INF/impala-example-extension.xml
```



Plug in your own extensions
via Spring beans



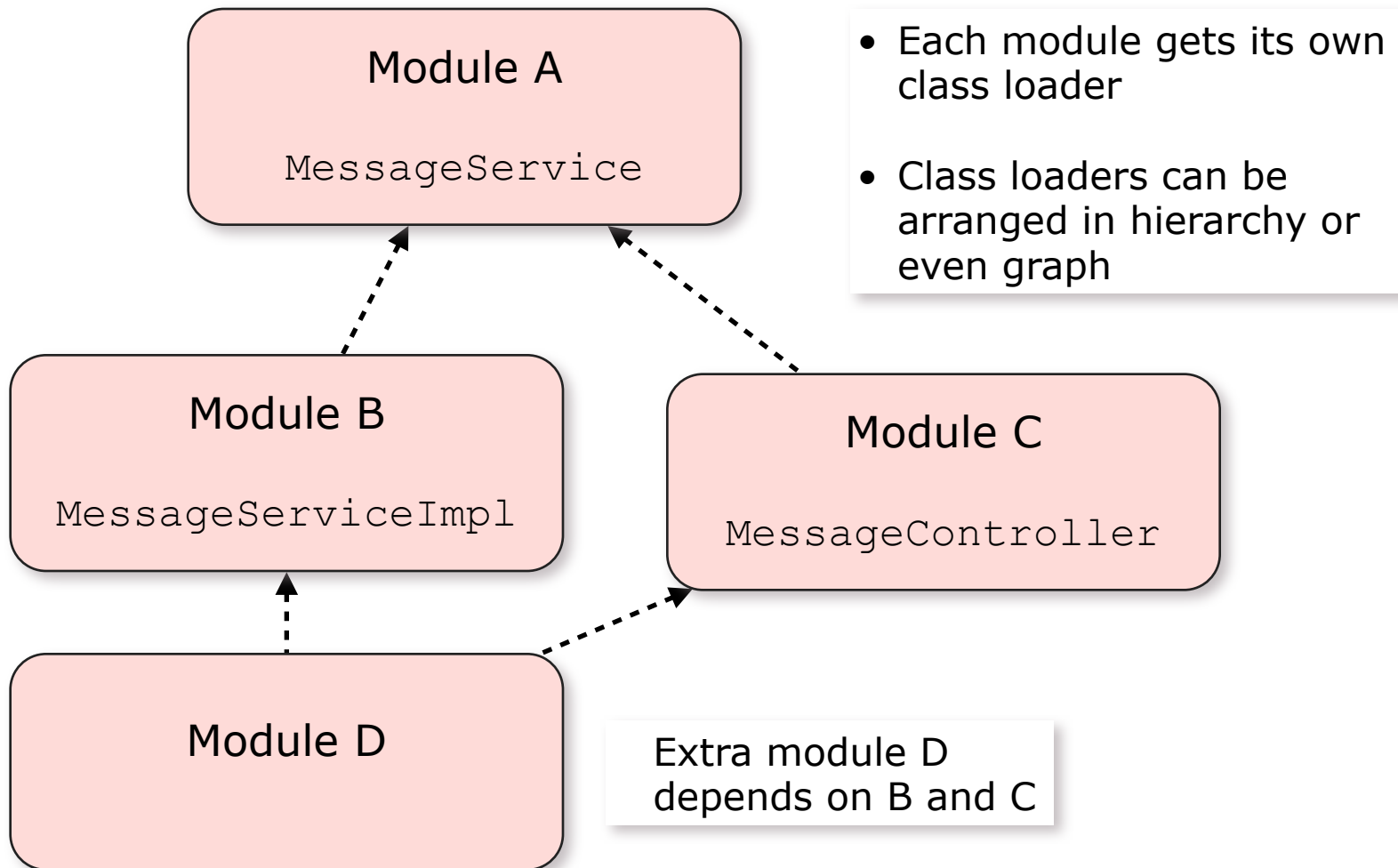
Impala “Under the Hood”

Key elements

- ▶ **Dynamic class loaders**
- ▶ **Dynamic services**
 - ▶ Shared service registry
 - ▶ Client proxies
- ▶ **Module life cycle management**
- ▶ **Pluggability**



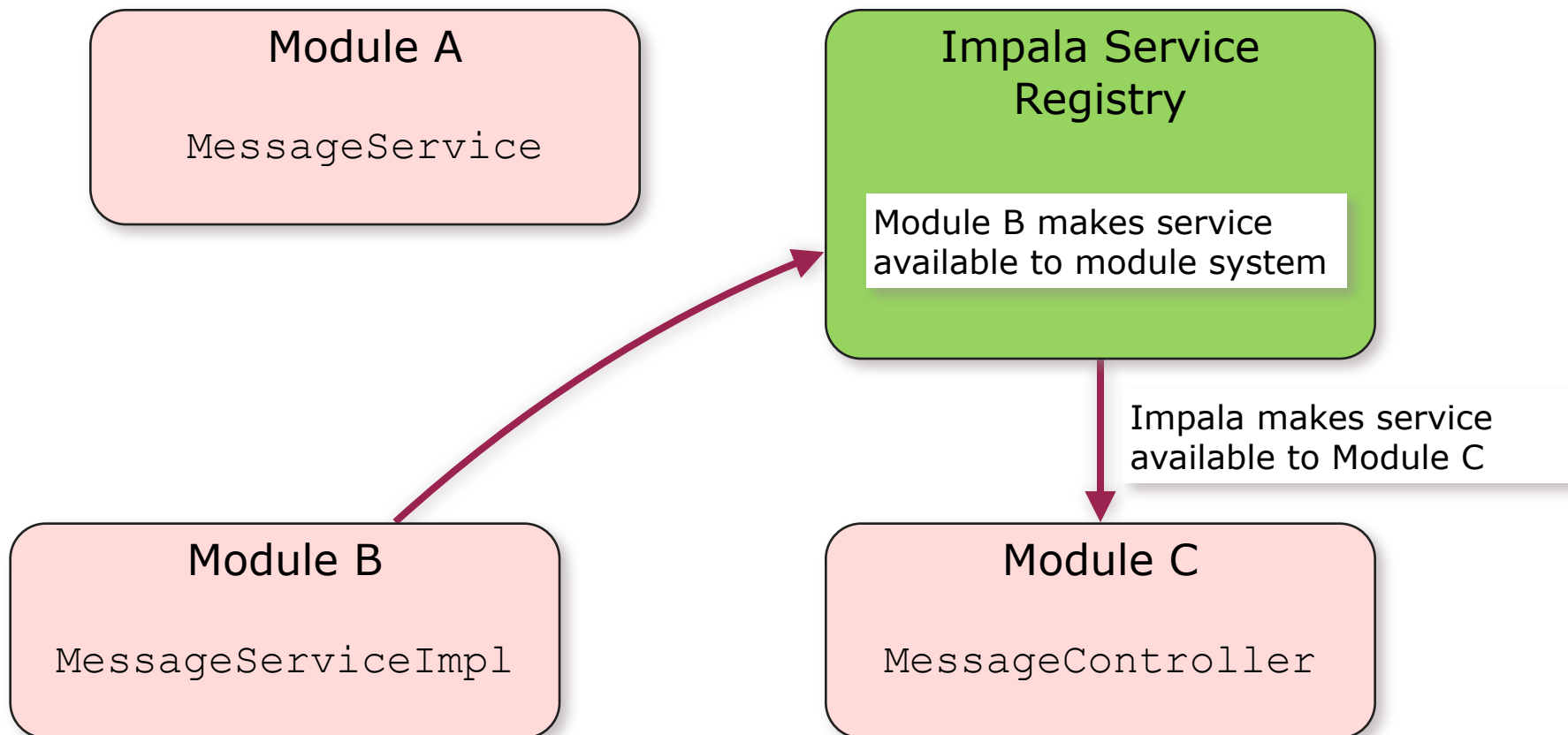
Dynamic class loaders





Dynamic services (1)

Shared service registry



Dynamic services (2)

Provider exports services to registry

- ▶ export by name, type or with attributes

Fragment from implementation
module application context

```
<!-- Export the message service by name to service registry -->  
  
<service:export beanName="messageService"/>  
  
<bean id="messageService" class="...MessageServiceImpl">  
    <property name = "message" value = "Hello World!"/>  
</bean>
```



Dynamic services (3)

Client service references managed via proxies

- ▶ client module can be loaded before service
- ▶ service module can be reloaded without affecting clients

```
<service:import  
  id="messageService"  
  proxyTypes="com.application.main.MessageService"/>
```

Fragment from client
module application
context

```
<bean name="/message.htm" class="...MessageController">  
  <property name = "messageService" ref = "messageService"/>  
</bean>
```



Module reload example

1. Module metadata loaded

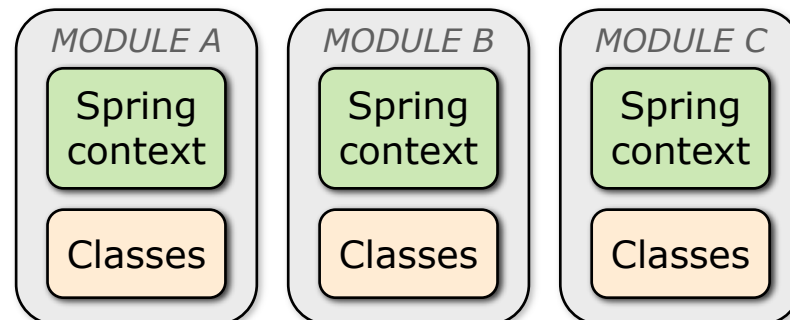
module.properties
moduledefinitions.xml
ModuleDefinitionSource

2. Module operations
determined

Unload D C B A
Load A B C

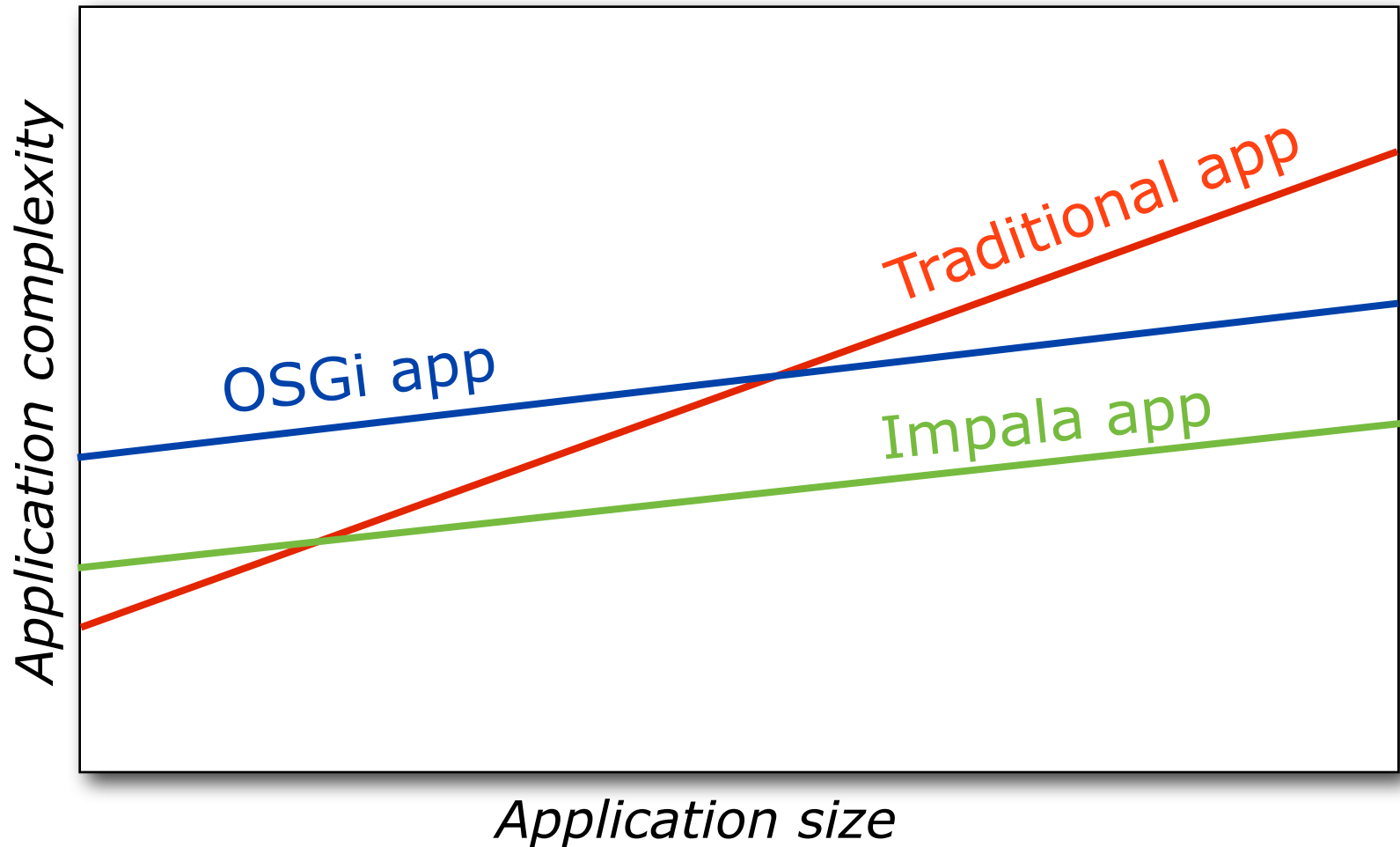
3. Modules (un)loaded in order

- ▶ class loader for module created
- ▶ application context loaded
- ▶ services exported to and imported from service registry as required





Complexity growth compared



The Impala Offering

- ▶ A modular application project structure
- ▶ Abstractions for representing and manipulating modules
- ▶ Dynamic class loaders for selective on-the-fly updates
- ▶ Transparent management of service dynamics
- ▶ Easier to write and faster to execute integration tests
- ▶ Genuinely multi-module web applications
- ▶ An Apache 2.0 open source license