The Tuscany Java Recursive Core: Architecture Update

5 June 2006

Agenda

- Spec Update
- Core Update
- Extending the Runtime

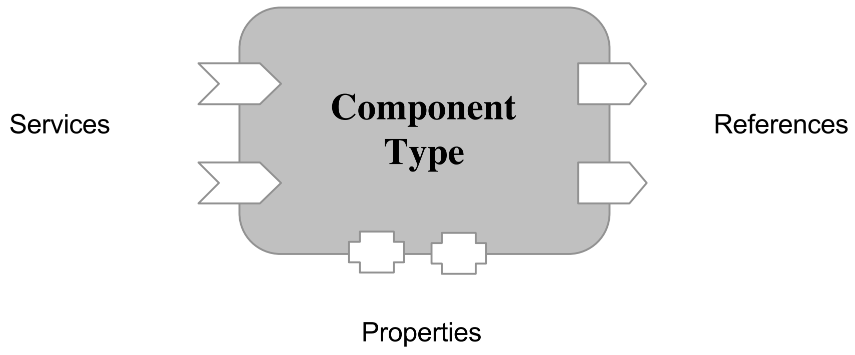
Agenda

- Spec Update
- Core Update
- Extending the Runtime

Simplify Composition

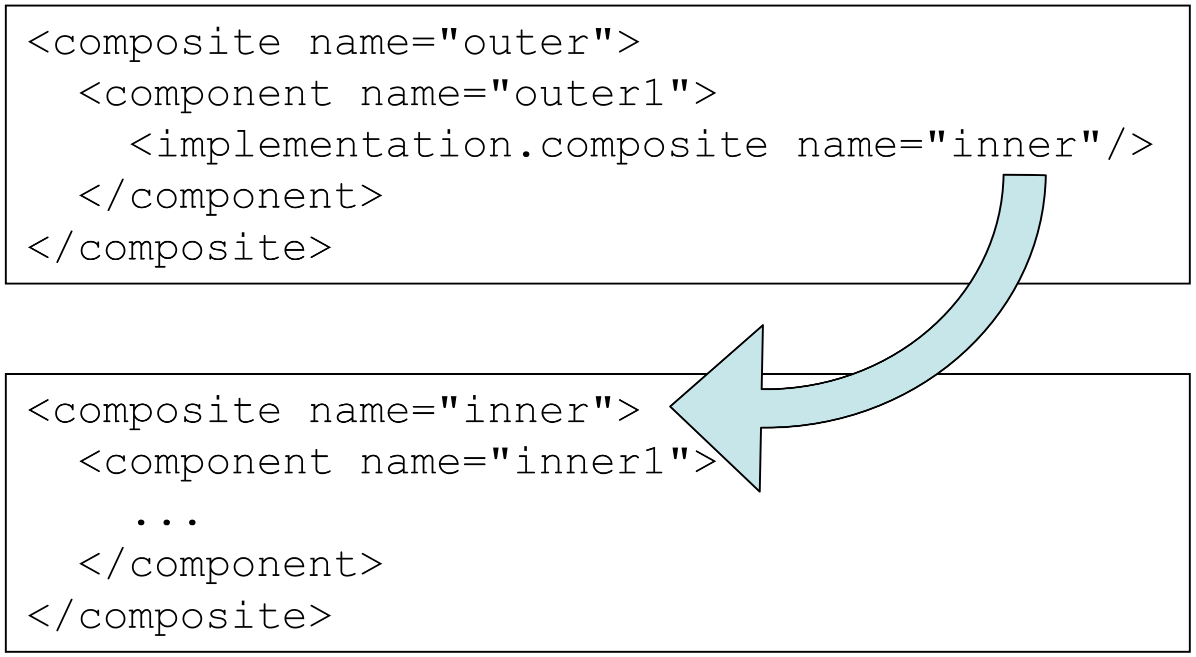
- Reduce the number of SCA concepts
  - Yet adds additional capabilities

- Modules become Composites
  - Composites are a compound Component Type
  - Entry Point eliminated, Service provides same function
  - External Service eliminated, Reference provides same function



Recursive Composition

- A Composite can become the implementation of a component
  - The component's services, references and properties are those of the composite that is used as its implementation.
- Nestable to any level



```
<composite name="outer">
  <component name="outer1">
    <implementation.composite name="inner"/>
  </component>
</composite>
```

```
<composite name="inner">
  <component name="inner1">
    ...
  </component>
</composite>
```

Complex Property Types

- Property types can now be any XML complex type
  - Before they were simple types represented by a string
- “type” attribute contains QName of complex type
- Body of <property> element contains default value

```
<property name=“fooProp" type=“foo:FooType">
  <foo:a>value</foo:a>
  <b>
    <b-1>inner value</b-1>
  </b>
</property>
```

Setting Component Property Values

- <v:> concept has been removed
- Value can be derived from content of <property> element:

```
<component name="comp”>
  <property name="prop">
    <foo:a>value</foo:a>
    <b><b-1>inner value</b-1></b>
  </property>
</component>
```

- Or, can be the result of an XPath expression

```
<property name="fooProp" type="FooType”>
    <foo:a>value</foo:a>
    <b><b-1>inner value</b-1></b>
</property>
<component name="comp”>
  <property name="prop” source="$fooProp/b”/>
</component>
```

Composite Inclusion

- Module Fragments have been removed
- Replaced by a SCA-specific include mechanism
  - XInclude was rejected due to complexity of XPointer
  - Alternative still being defined

- Work in progress on subsystem concept
  - Some form of open composite
  - Tied into deployment concepts

Packaging Structures

- No specification yet of a packaging format for a composite
- We need to do something in Tuscany
  - OSGi bundle format looks promising

Agenda

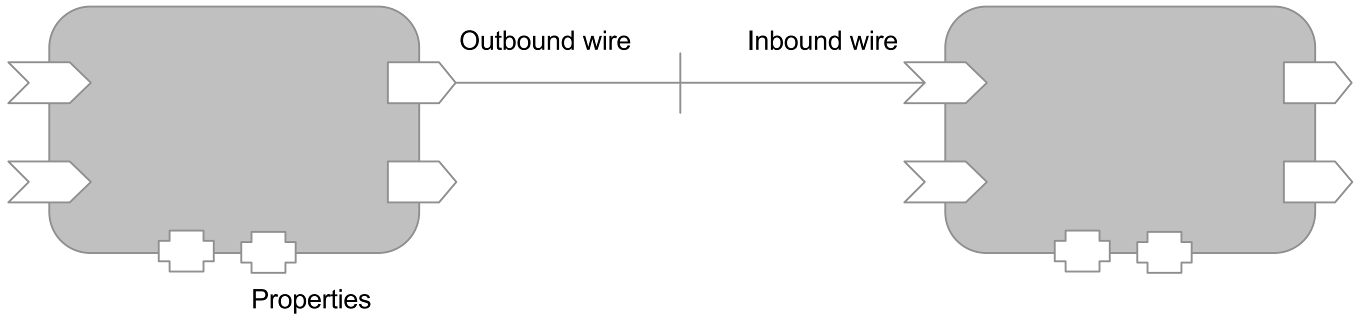
- Spec Update
- Core Update
  - Overview
  - Atomic Components
  - Wires
  - Composites
  - The Component Tree
  - Deployment
  - Loading
  - Building
  - Bootstrap
- Extending the Runtime

Overview

- SCA recursion as well experience with M1 led us to the conclusion that the existing core architecture needed substantial refactoring
  - Overall brittleness
    - The $10,000,000 question: Which classloader is active at any given point?
    - Component type loading
    - Model initialization and walking
    - Wiring and proxy generation
    - Deprecated methods and use throughout the runtime
  - Parts of the core were baroque
    - Artifact registration
    - Context factories, builders, invokers
    - Model decoration
    - Mixing of behavior and model
  - No clear separation of concerns
    - No SPI demarcation
    - Model bleeds into runtime
  - Core development cycle too long
    - Develop-build-test-deploy cycle was painful
  - Implementing recursion with the existing architecture would not simplify
  - Most importantly, things could be easier and simpler…

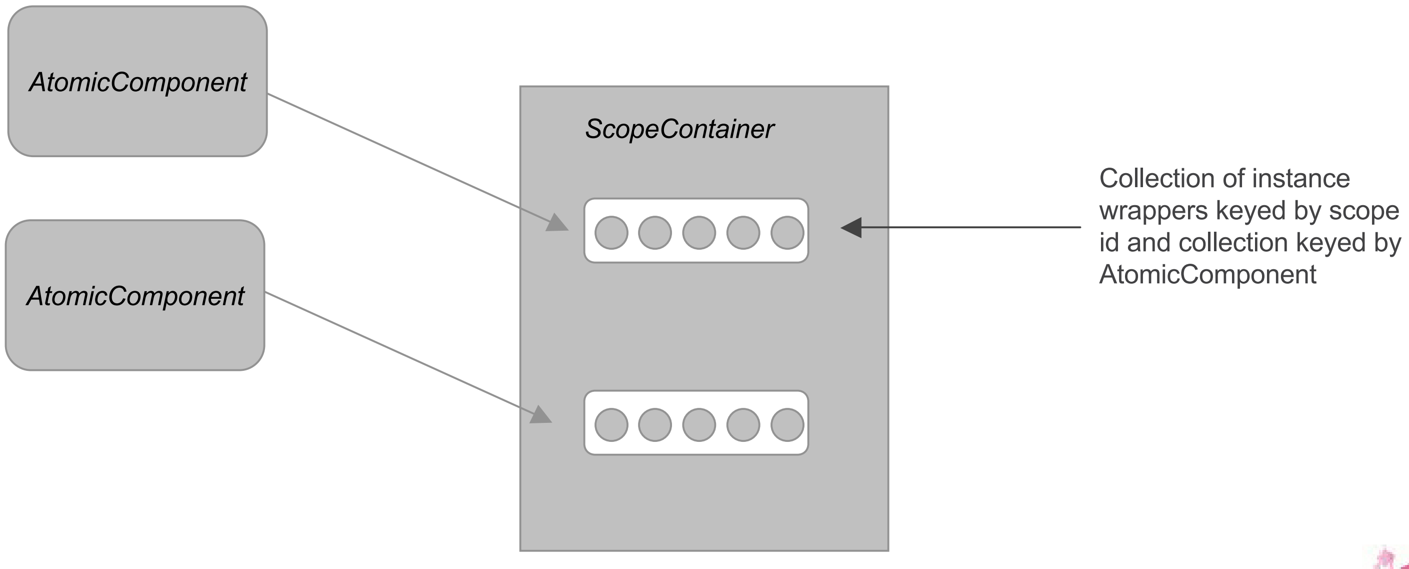
Atomic Component

- *AtomicComponent:* the most basic component form
  - Corresponds to the spec concept
    - Offers services, has references and properties
  - Implementations deal with the specifics of a type, e.g. Java, XSLT, etc.
  - Has implementation instances
  - Has inbound and outbound wires

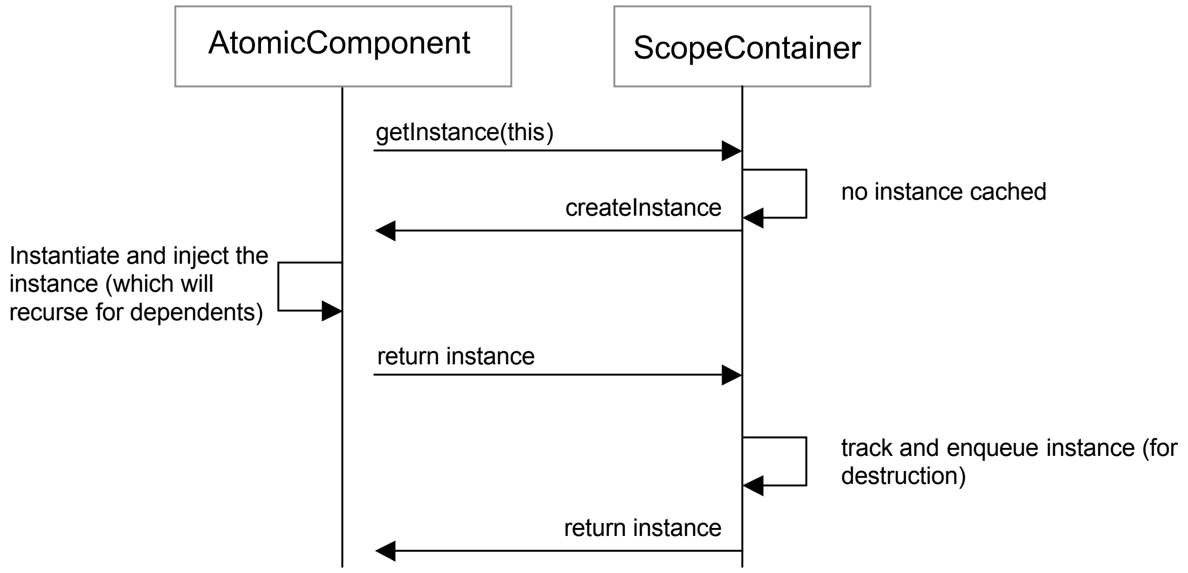


Atomic Component Instance Management

- Atomic components use a *ScopeContainer* to manage implementation instances
  - Module, HTTP Session, Request, Stateless
  - *ScopeContainers* track implementation instances by scope id and the AtomicComponent instance identity
  - Instances are stored in an *InstanceWrapper* which is specific to the component implementation type (e.g. PojoInstanceWrapper.java)

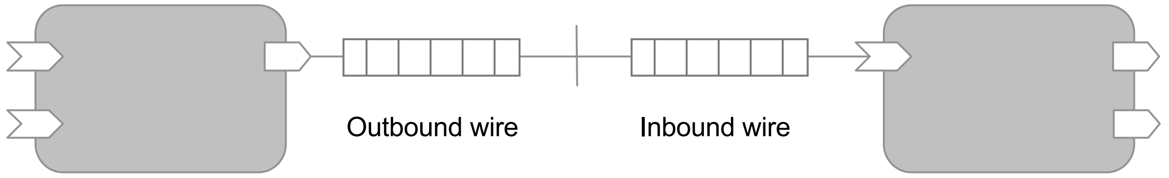

AtomicComponent
AtomicComponent
ScopeContainer
Collection of instance wrappers keyed by scope id and collection keyed by AtomicComponent

Component Implementation Instance Creation


AtomicComponent
ScopeContainer
getInstance(this)
no instance cached
createInstance
Instantiate and inject the instance (which will recurse for dependents)
return instance
track and enqueue instance (for destruction)
return instance

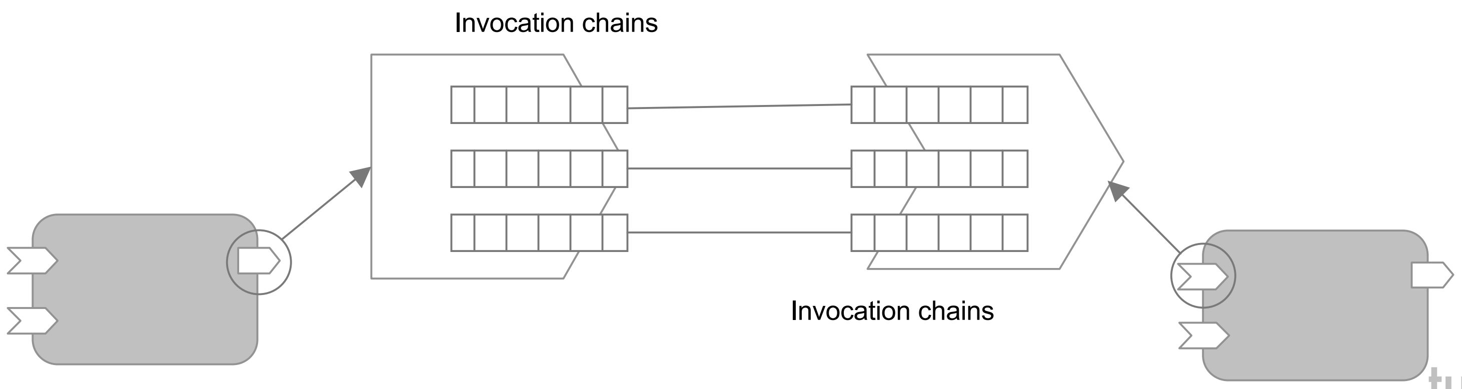
Wires: *RuntimeWire*

- Corresponds to the specification term *wire*
  - Responsible for flowing an invocation to a target
  - Components (atomic and composite) have 0..n wires
- Two sides
  - *InboundWire* - handles the source side of a wire, including policy
  - *OutboundWire* - handles the target side of a wire, including policy
  - The runtime connects inbound and outbound wires, performing optimizations if possible
  - Inbound and outbound wires may be associated with different service contracts
  - Different implementation types
    - “Standard” wires contain invocation chains that have *Interceptors* and *MessageHandler*s that perform some form of mediation (e.g. policy)
    - Other wire types exist that, for example, do not perform mediations



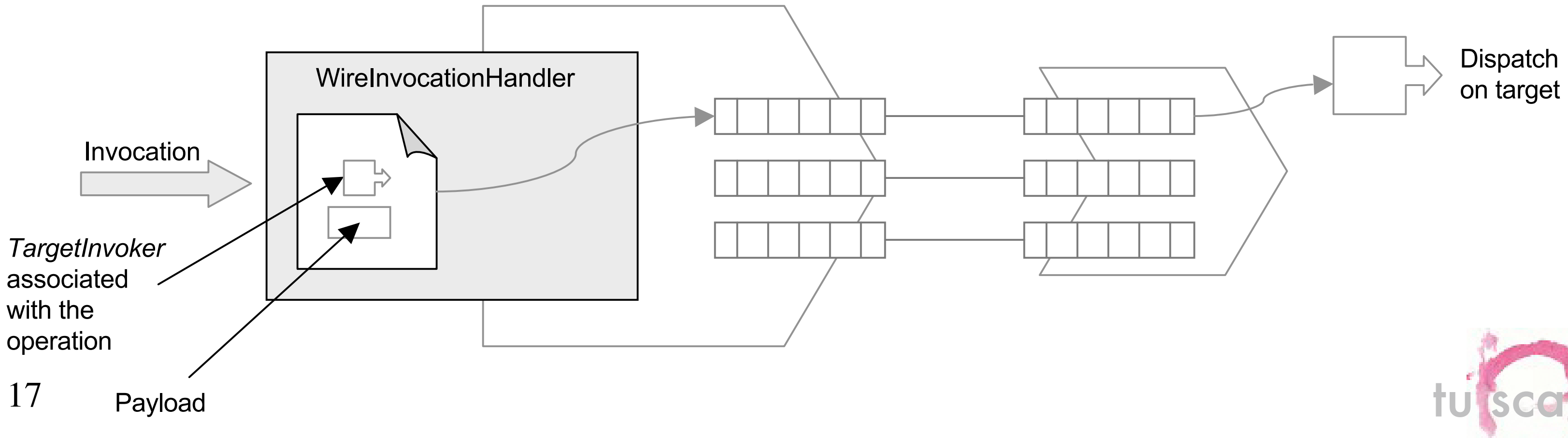
Invocation Chains

- A wire has an *InvocationChain* per service operation
  - An InvocationChain may have
    - *Interceptors* - “Around-style” mediation
    - One-way *MessageHandlers*
- Component implementation instances access a wire through a *WireInvocationHandler* associated with a reference
  - *WireInvocationHandlers* may (or may not depending on the component type) be fronted by a proxy
  - *WireInvocationHandlers* dispatch an invocation to the correct chain
- A wire has a *TargetInvoker* that is created from the target side AtomicComponent or Reference and is stored on the source wire invocation handler. The TargetInvoker is resposnible for dispatching the request to the target instance when the message hits the end of the target invocation chain.



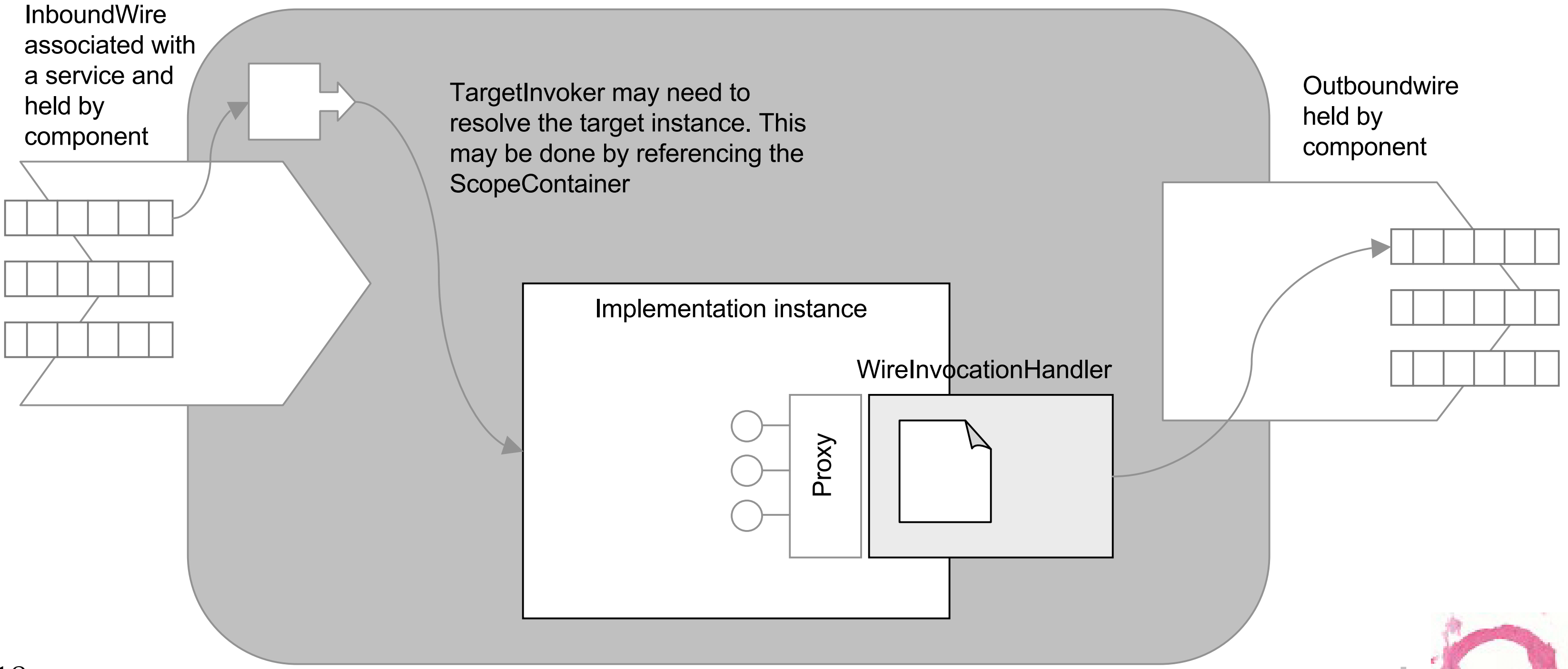
Invocation Overview

- An invocation is dispatched to the *WireInvocationHandler*
- The *WireInvocationHandler* looks up the correct *InvocationChain*
- It then creates a message, sets the payload, sets the *TargetInvoker,* and passes the message down the chain
- When the message reaches the end of the chain, the *TargetInvoker* is called, which in turn is responsible for dispatching to the target
- Having the TargetInvoker stored on the outbound side allows it to cache the target instance when the wire source has a scope of equal or lesser value than the target (e.g. request-->module)

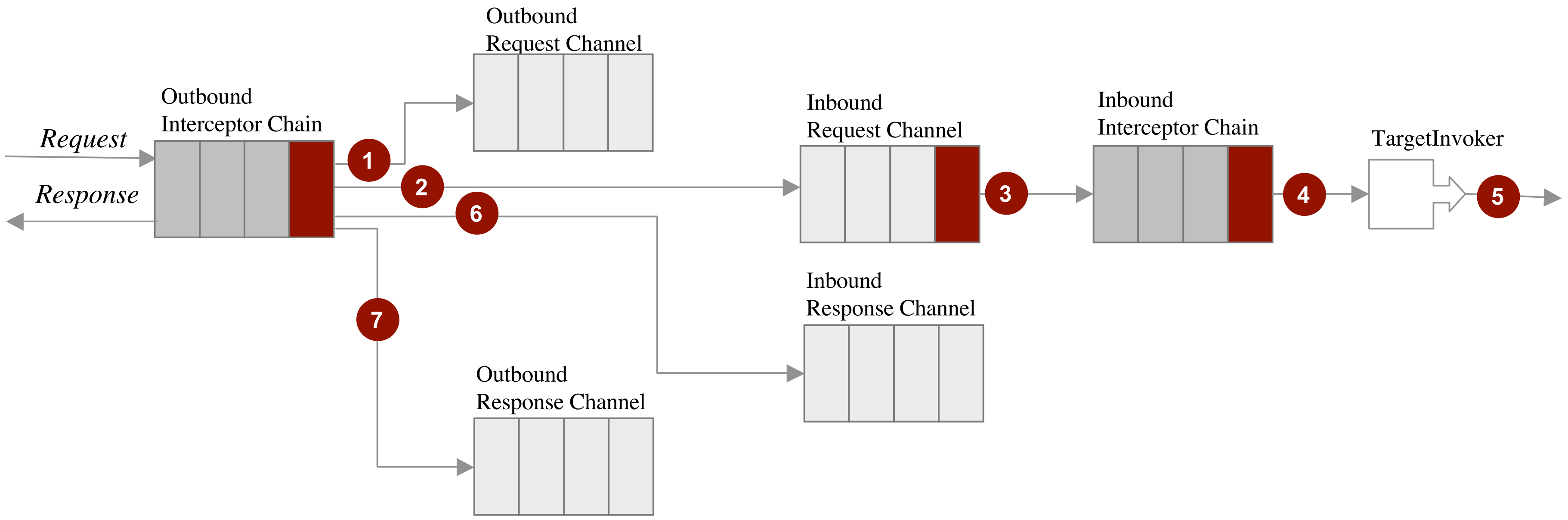


Wires and Implementation Instances

- The runtime provides components with *InboundWires* and *OutboundWires*
- *InvocationChains* are held in component wires and are therefore stateless
    - Allows for dynamic behavior such as introduction of new interceptors or re-wiring


InboundWire associated with a service and held by component
TargetInvoker may need to resolve the target instance. This may be done by referencing the ScopeContainer
Outboundwire held by component
Implementation instance
WireInvocationHandler
Proxy

Invocation Detail

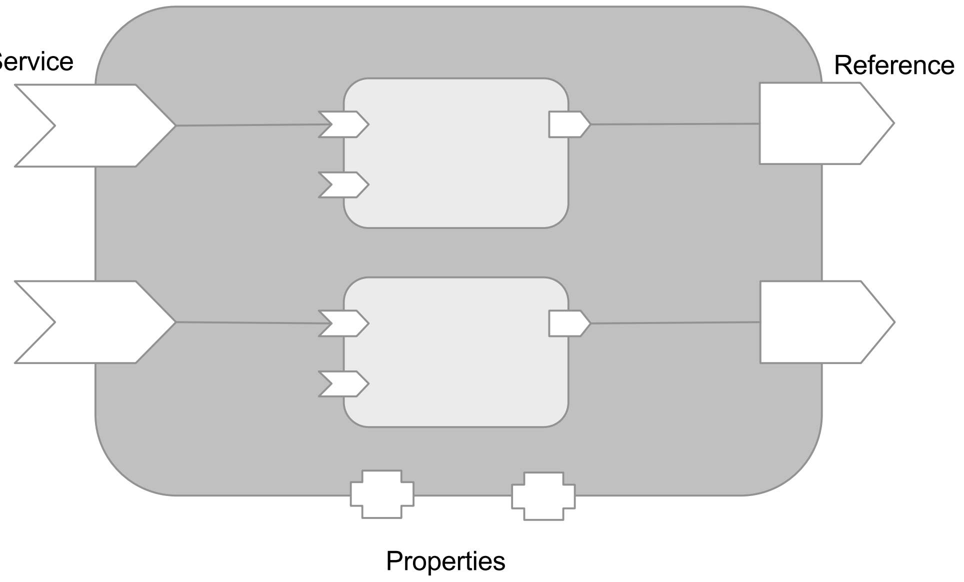


1. *RequestResponseInterceptor* (RRI) dispatches to the outbound request channel (OReqC)
2. RRI dispatches to the inbound request channel (IReqC)
3. *MessageDispatcher* dispatches to the inbound interceptor chain
4. *TargetInvokerInterceptor* pulls the *TargetInvoker* from the message and dispatches to it
5. *TargetInvoker* dispatches to the target (implementation instance or to some transport)
6. RRI dispatches to the inbound response channel (IRespC)
7. RRI dispatches to the outbound response channel (ORespC)
8. The response is returned back up the outbound channel to the client

Note the runtime may optimize invocations not to call interceptors or handlers and possibly dispatch directly to a target (or any combination thereof)

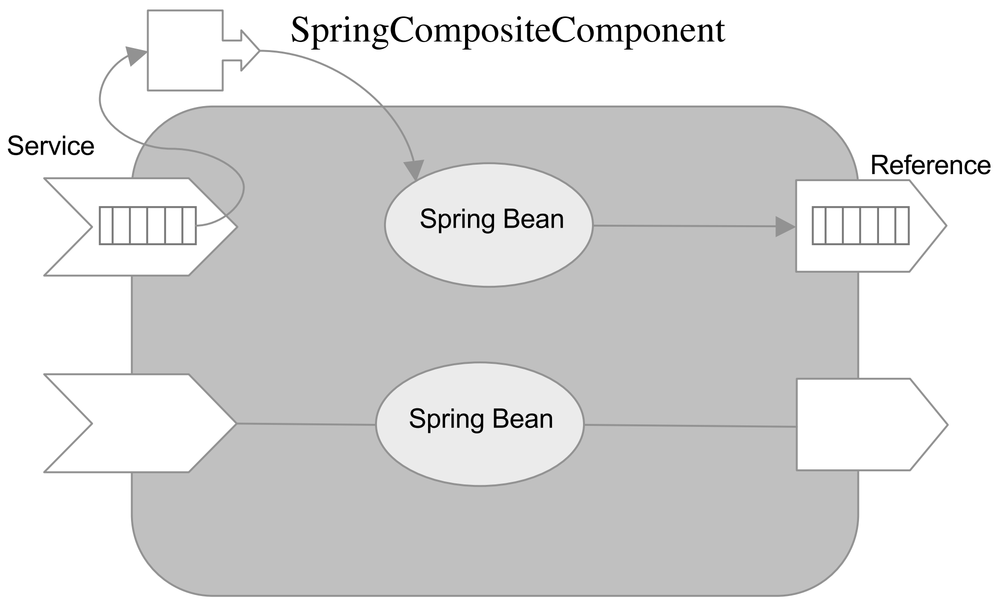
Composite Component

- *CompositeComponent:* a component which contains child components
    - Corresponds to the spec concept
        - Offers *services*, has *references* over bindings
        - Has properties
    - Has inbound and outbound wires associated with Services and References
- Implementations deal with the specifics of a type, e.g. Spring



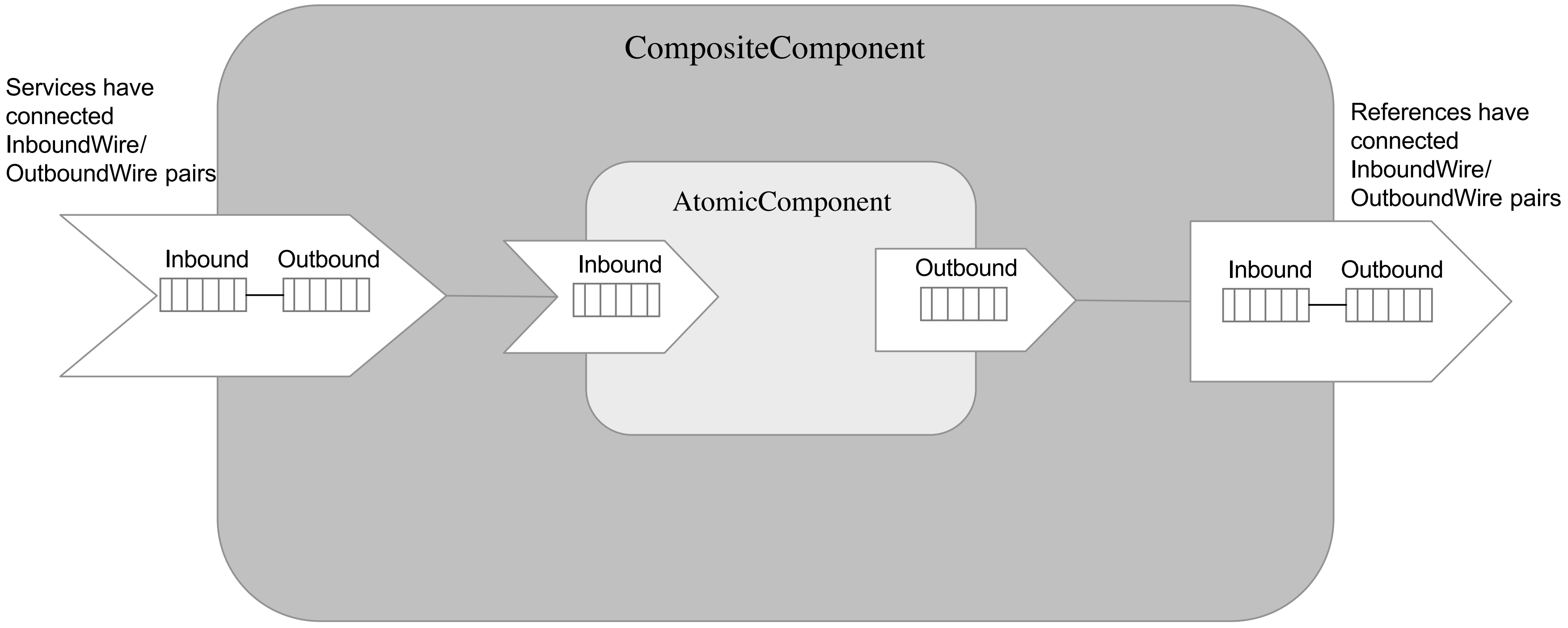
A Composite Example

- Expose a Spring Bean through a *service* with a SOAP/HTTP binding and using a *reference* bound over SOAP/JMS
- Spring *application.xml* does not need to be modified
  - Preserves Spring programming model
  - Allows existing Spring applications to be wired in an assembly
- Services and references are contributed by SCA bindings (e.g. Celtix)



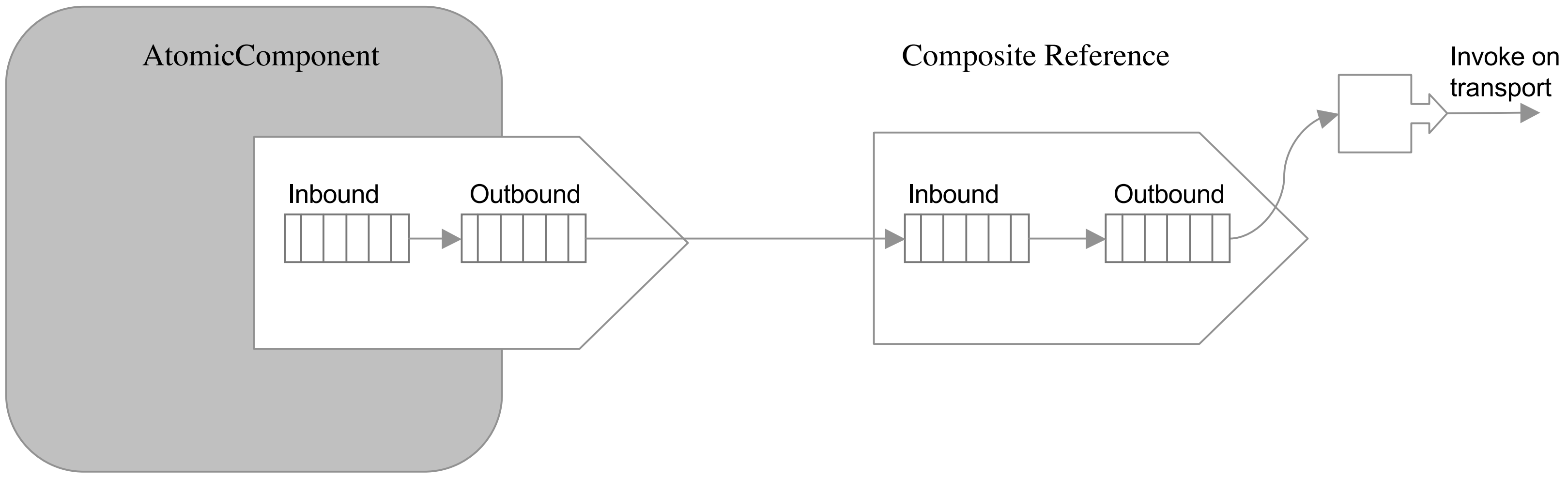
Composite Wiring

- Services and references are "twist-ties" for inbound and outbound wires


CompositeComponent
Services have connected InboundWire/ OutboundWire pairs
Inbound
Outbound
AtomicComponent
Inbound
Outbound
References have connected InboundWire/ OutboundWire pairs
Inbound
Outbound

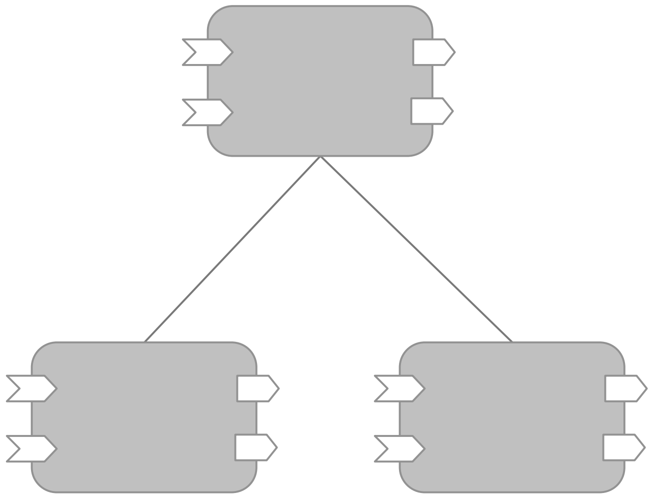
Composite Reference Invocations

- When a message reaches the end of the outbound reference wire, the TargetInvoker dispatches over a transport
  - In contrast to Component-->Component invocations, where the TargetInvoker would resolve the target implementation instance

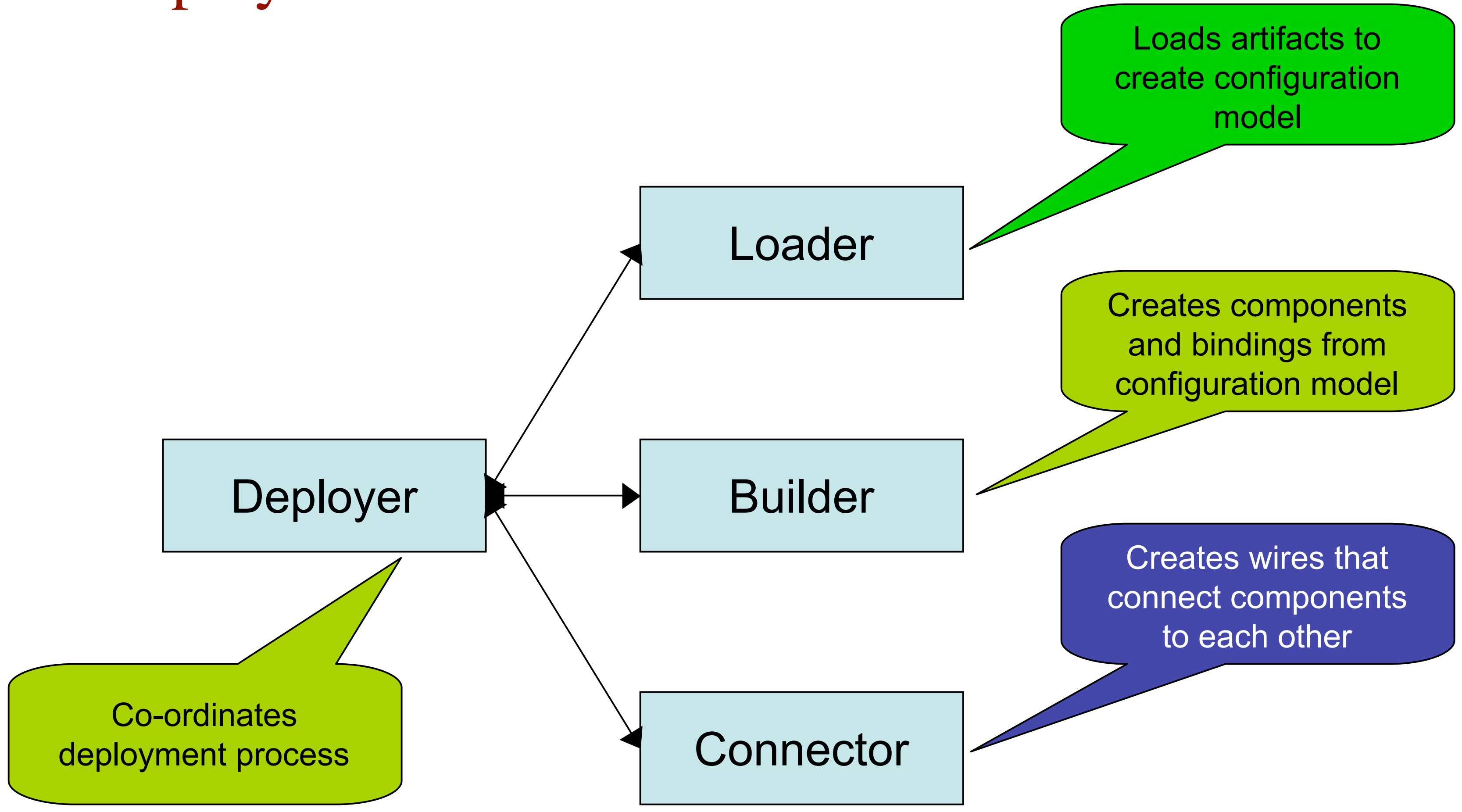


Component Tree

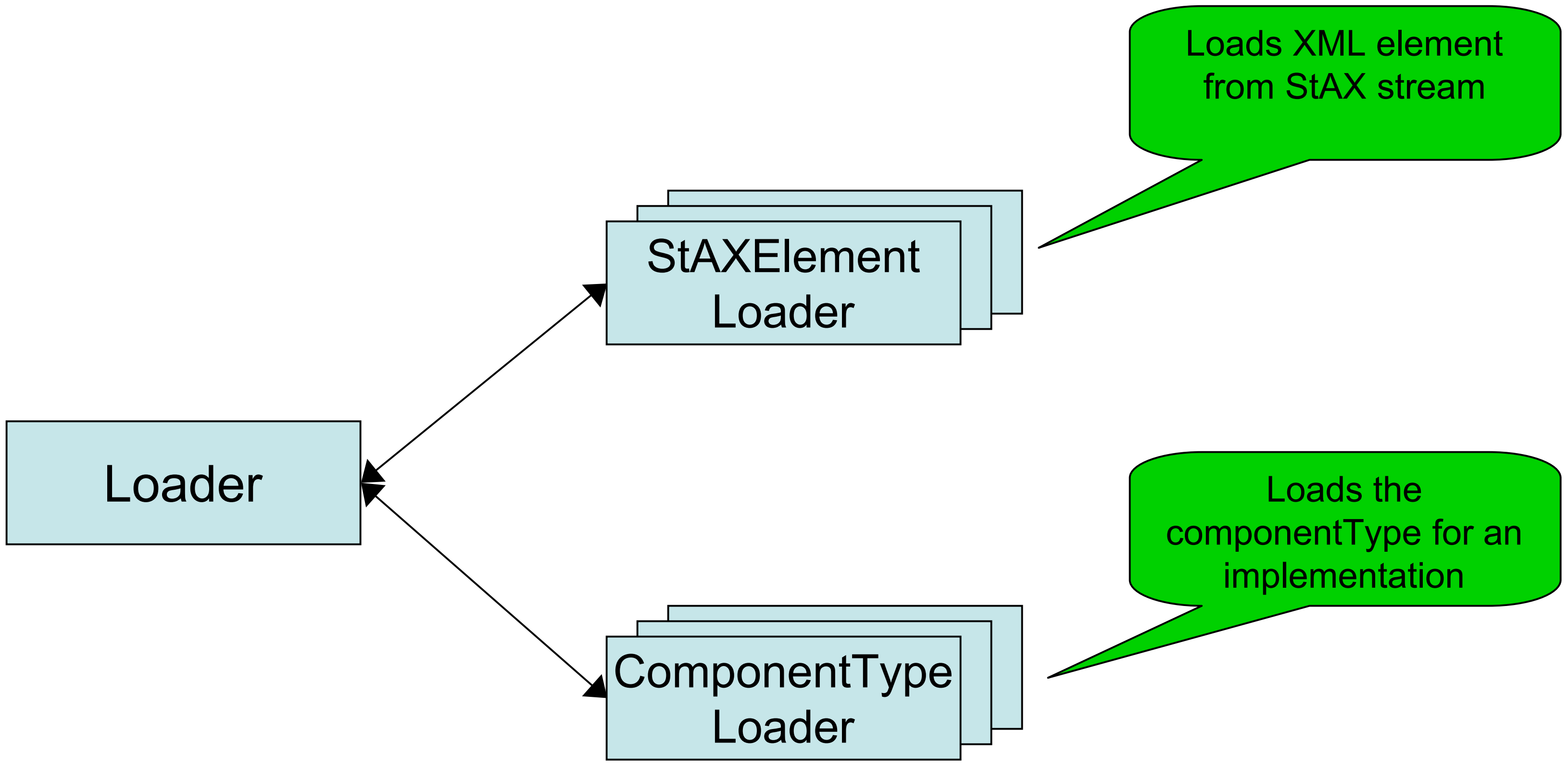
- Composites form a containment hierarchy
- The hierarchy may be arbitrarily deep
- The runtime itself, *DefaultRuntime*, is a composite
- There are two runtime hierarchies
  - The *application* hierarchy
    - End-user components
  - The *system* hierarchy
    - Runtime extensions



Deployment


Loads artifacts to create configuration model
Loader
Creates components and bindings from configuration model
Deployer
Builder
Creates wires that connect components to each other
Co-ordinates deployment process
Connector

Configuration Loading


Loads XML element from StAX stream
StAXElement Loader
Loader
Loads the componentType for an implementation
ComponentType Loader

Component Type Loading

- Loads the component type definition for a specific implementation
  - How it does this is implementation-specific
  - May load XML sidefile (location set by implementation)
  - May introspect an implementation artifact (e.g. Java annotations)
  - … or anything else
- Composite ComponentType Loader
  - Load SCDL from supplied URL
  - Extract and load SCDL from composite package (format TBD)
- POJO ComponentType Loader
  - Load SCDL from class-relative ".componentType" sidefile
  - Introspect Java annotations

Annotation Processing

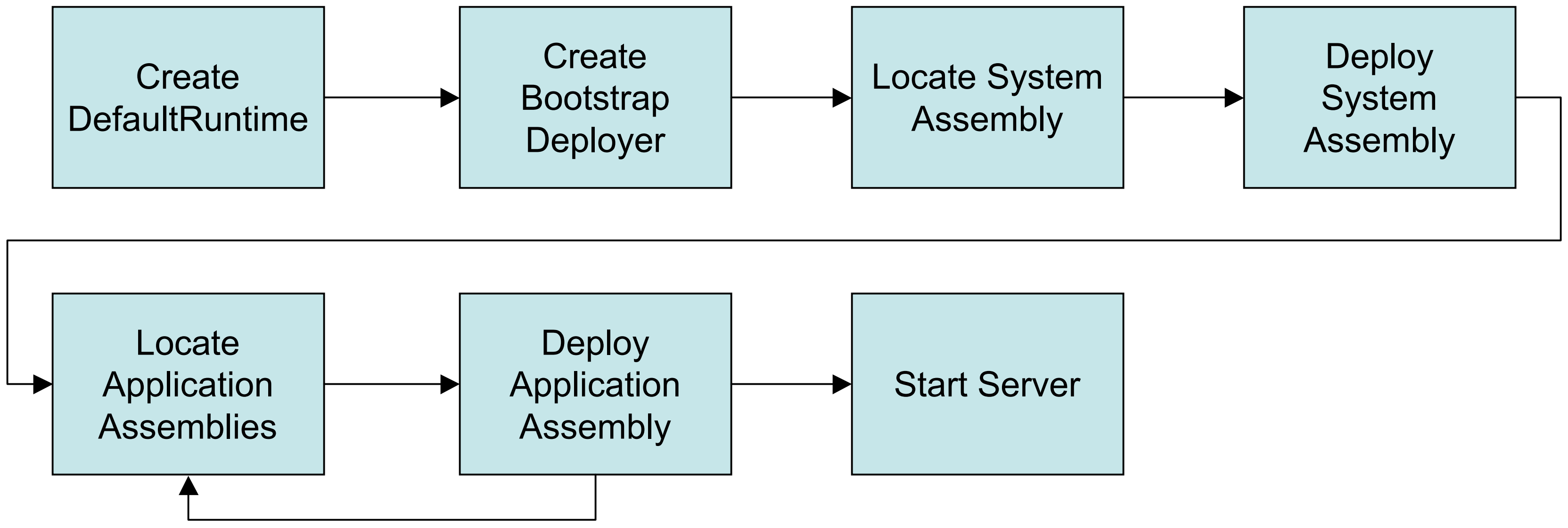
- Spec needs to clarify annotation processing algorithms
  - Proposal based on the email to our list 5/29
- Pluggable annotation processors
  - Ability to add new annotations
  - Extend componentType structure
    - @Init
    - @Autowire
  - Magic Property and Reference definitions
    - @SDOHelper
    - @Monitor

Building

- Builder creates a runtime component from the configuration model
  - Builder for each implementation type
  - Builder for each binding type (service or reference component)
- Runtime component manages:
  - Implementation instances
  - Inbound and Outbound wires
- Every implementation is likely to be different
  - Different artifacts, programming model, …
- Composite implementation recurses for contained components
  - Re-invokes the Builder for every child

Bootstrap

- Bootstrap process is controlled by Host environment
- Default process implemented in DefaultBootstrapper

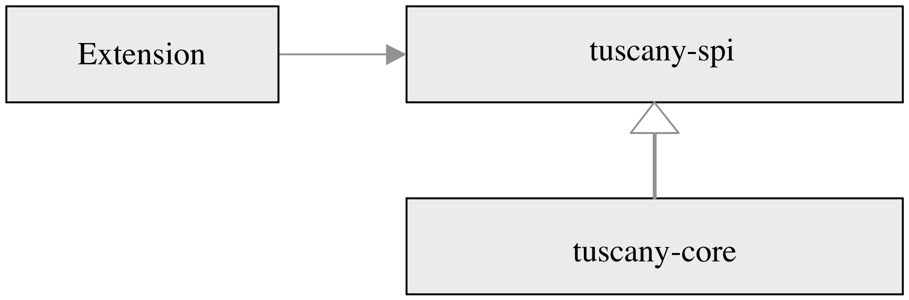

Create DefaultRuntime
Create Bootstrap Deployer
Locate System Assembly
Deploy System Assembly
Locate Application Assemblies
Deploy Application Assembly
Start Server

Agenda

- Spec Update
- Core Update
- Extending the Runtime

The SPI Package

- A separate SPI project has been created for extending the runtime
  - Extension code *must never* reference classes in *tuscany-core*
    - Besides being bad design, this type of code will break when deployed to hosts that enforce classloader isolation
    - There is a *tuscany-test* package for testcases to instantiate a few core implementation classes without directly referrencing them (we should look to eliminate it if possible)
- *tuscany-core* is an implementation of *tuscany-spi*



Extension Design Principals

- The core must be as simple as possible but no simpler
  - Simple but not simplistic
  - Don’t sacrifice performance for extension simplification
- Make the extension API progressive
  - High-level API for common cases
  - A lower-level API that allows extenders nearly complete access
- Make the runtime easier for application developers first, then extenders
- Make everything possible an extension
  - Limit the knowledge of the core
  - The core should know very little about extension points
  - People will want to extend the runtime in ways we haven’t thought of so make it as flexible as possible
- Extenders are Java developers and understand IoC
  - Extending the runtime should only require knowledge of J2SE and IoC

Base Extension Types

- Components
- Protocol Bindings
- Transport Bindings
- Data Bindings
- Policy
- Loaders
- Other types:
  - Scope contexts
  - Wires
  - Proxy generation
  - Anything people can think of and attach to the runtime…

Extension Package

- *tuscany-spi* contains an extension package with abstract classes for base extensions
  - *AtomicComponentExtension.java*
  - *CompositeComponentExtension.java*
  - *LoaderExtension.java*
  - *ReferenceExtension.java*
  - *ServiceExtension.java*

Creating an Atomic Component Implementation Type

1. Implement 1..n model POJOs
2. Implement *Loader* or extend *LoaderExtension*
   - Reads a StAX stream and creates appropriate model objects
3. Implement *AtomicComponent* or extend *AtomicComponentExtension*
   - Extension will be given *InboundWires* and *OutboundWires* corresponding to its services and references; it must decide how to inject those onto its implementation instances (e.g. create a proxy)
   - A *WireService* will be provided to the extension, which it can use to generate proxies and *WireInvocationHandlers*
   - The extension must implement `createTargetInvoker(..)` and instantiate *TargetInvokers* responsible for dispatching to target instances
4. Implement *ComponentBuilder* or extend *ComponentBuilderExtension*
   - Implements `build(..)` which returns the *AtomicComponent* implementation
5. Write a simple SCDL file and deploy into the system composite hierarchy

Creating a Composite Component Implementation Type

1. Implement 1..n model POJOs
2. Implement *Loader* or extend *LoaderExtension*
   - Reads a StAX stream and creates appropriate model objects
3. Implement *CompositeComponent* or extend *CompositeComponentExtension*
   - The extension must implement `createTargetInvoker(..)` and instantiate *TargetInvokers* responsible for dispatching to target child instances
4. Implement *ComponentBuilder* or extend *ComponentBuilderExtension*
   - Implements `build(..)` which returns the *AtomicComponent* implementation
5. Write a simple SCDL file and deploy into the system composite hierarchy

Creating a Service

1. Implement 1..n model POJOs
2. Implement *Loader* or extend *LoaderExtension*
   - Reads a StAX stream and creates appropriate model objects
3. Implement *Service* or extend *ServiceExtension*
   - Receives requests from a binding. This may involve interfacing with the Tuscany host environment through the *Host API*
4. Implement *BindingBuilder* or extend *BindingBuilderExtension*
   - Implements `build(..)` which returns the *Service* implementation
5. Write a simple SCDL file and deploy into the system composite hierarchy

Creating a Reference

1. Implement 1..n model POJOs
2. Implement *Loader* or extend *LoaderExtension*
   - Reads a StAX stream and creates appropriate model objects
3. Implement *Reference* or extend *ReferenceExtension*
   - Receives requests from a binding. This may involve interfacing with the Tuscany host environment through the *Host API*
   - The extension must also implement `createTargetInvoker(..)` and create *TargetInvokers* responsible for dispatching to the transport
4. Implement *BindingBuilder* or extend *BindingBuilderExtension*
   - Implements `build(..)` which returns the *Reference* implementation
5. Write a simple SCDL file and deploy into the system composite hierarchy

Creating a Generic Extension

1. Write a Java class
2. Write a simple SCDL file and deploy into the system composite hierarchy