# Qi4j - Dependency Injection

Rickard Öberg

# Agenda

# Agenda

:: What is Dependency Injection?

# Agenda

:: What is Dependency Injection?
:: Why use Dependency Injection?

# Agenda

:: What is Dependency Injection?
:: Why use Dependency Injection?
:: Domain-oriented injection

# Agenda

:: What is Dependency Injection?
:: Why use Dependency Injection?
:: Domain-oriented injection
:: DI in Qi4j

# Agenda

:: What is Dependency Injection?
:: Why use Dependency Injection?
:: Domain-oriented injection
:: DI in Qi4j
:: Dependency scopes in Qi4j

# What is Dependency Injection?

# What is Dependency Injection?

:: How does code access other objects?

# What is Dependency Injection?

:: How does code access other objects?
:: Instantiation

# What is Dependency Injection?

:: How does code access other objects?
:: Instantiation
:: Service locator

# What is Dependency Injection?

:: How does code access other objects?
:: Instantiation
:: Service locator
:: Allow user to “inject” dependencies through constructors or methods

# Why use Dependency Injection?

# Why use Dependency Injection?

:: Put assumptions in one place

# Why use Dependency Injection?

:: Put assumptions in one place
  :: Makes reuse easier

# Why use Dependency Injection?

:: Put assumptions in one place
  :: Makes reuse easier
  :: Makes testing easier

# Why use Dependency Injection?

- Put assumptions in one place
  - Makes reuse easier
  - Makes testing easier
  - Makes code clearer

# Why use Dependency Injection?

- Put assumptions in one place
  - Makes reuse easier
  - Makes testing easier
  - Makes code clearer
  - Reduces coupling on infrastructure

# Why use Dependency Injection?

- Put assumptions in one place
  - Makes reuse easier
  - Makes testing easier
  - Makes code clearer
  - Reduces coupling on infrastructure
- Makes dependencies more explicit

# Domain-oriented injection

# Domain-oriented injection

:: “When all you have is Object everything looks like a dependency”

# Domain-oriented injection

:: "When all you have is Object everything looks like a dependency"
:: Injected objects usually have pattern roles

# Domain-oriented injection

:: "When all you have is Object everything looks like a dependency"
:: Injected objects usually have pattern roles
  :: Adapter

# Domain-oriented injection

- “When all you have is Object everything looks like a dependency”
- Injected objects usually have pattern roles
  - Adapter
  - Decorator

# Domain-oriented injection

- “When all you have is Object everything looks like a dependency”
- Injected objects usually have pattern roles
  - Adapter
  - Decorator
  - Service

# Domain-oriented injection

- "When all you have is Object everything looks like a dependency"
- Injected objects usually have pattern roles
  - Adapter
  - Decorator
  - Service
- Not all objects can fulfill all roles

# Domain-oriented injection

- "When all you have is Object everything looks like a dependency"
- Injected objects usually have pattern roles
  - Adapter
  - Decorator
  - Service
- Not all objects can fulfill all roles
- Let injection be domain-oriented

# Domain-oriented injection

- "When all you have is Object everything looks like a dependency"
- Injected objects usually have pattern roles
  - Adapter
  - Decorator
  - Service
- Not all objects can fulfill all roles
- Let injection be domain-oriented
  - @Inject -> @Service

# DI in Qi4j

# DI in Qi4j

:: Uses annotations to declare dependencies

# DI in Qi4j

:: Uses annotations to declare dependencies
:: Each annotation has @DependencyScope

# DI in Qi4j

:: Uses annotations to declare dependencies
:: Each annotation has @DependencyScope
  :: You can create your own scopes

# DI in Qi4j

:: Uses annotations to declare dependencies
:: Each annotation has @DependencyScope
  :: You can create your own scopes
:: Each scope has its own resolver

# DI in Qi4j

- Uses annotations to declare dependencies
- Each annotation has @DependencyScope
  - You can create your own scopes
- Each scope has its own resolver
- DI is done in two phases

# DI in Qi4j

:: Uses annotations to declare dependencies
:: Each annotation has @DependencyScope
  :: You can create your own scopes
:: Each scope has its own resolver
:: DI is done in two phases
  :: Resolution - once

# DI in Qi4j

- Uses annotations to declare dependencies
- Each annotation has @DependencyScope
  - You can create your own scopes
- Each scope has its own resolver
- DI is done in two phases
  - Resolution - once
  - Injection - many times

# DI in Qi4j

:: Uses annotations to declare dependencies
:: Each annotation has @DependencyScope
  :: You can create your own scopes
:: Each scope has its own resolver
:: DI is done in two phases
  :: Resolution - once
  :: Injection - many times
  :: Fast!

# DI in Qi4j

:: Uses annotations to declare dependencies
:: Each annotation has @DependencyScope
  :: You can create your own scopes
:: Each scope has its own resolver
:: DI is done in two phases
  :: Resolution - once
  :: Injection - many times
  :: Fast!
:: Some scopes allow optional injection

# DI in Qi4j

:: Uses annotations to declare dependencies
:: Each annotation has @DependencyScope
  :: You can create your own scopes
:: Each scope has its own resolver
:: DI is done in two phases
  :: Resolution - once
  :: Injection - many times
  :: Fast!
:: Some scopes allow optional injection
  :: @Service(optional=true)

# DI in Qi4j

# DI in Qi4j

- Injection can be done on constructor parameters, fields and method parameters (in that order)

# DI in Qi4j

- Injection can be done on constructor parameters, fields and method parameters (in that order)
- Injection is set per-parameter, so one constructor or method can have multiple injections

# DI in Qi4j

- Injection can be done on constructor parameters, fields and method parameters (in that order)
- Injection is set per-parameter, so one constructor or method can have multiple injections
- All parameters must be injected

# DI in Qi4j

- Injection can be done on constructor parameters, fields and method parameters (in that order)
- Injection is set per-parameter, so one constructor or method can have multiple injections
- All parameters must be injected
- Any access modifier can be used

# DI in Qi4j

- Injection can be done on constructor parameters, fields and method parameters (in that order)
- Injection is set per-parameter, so one constructor or method can have multiple injections
- All parameters must be injected
- Any access modifier can be used
  - private, protected, package protected, public

# DI in Qi4j

# DI in Qi4j

- Injection can be done both Fragments and regular Objects

# DI in Qi4j

- Injection can be done both Fragments and regular Objects
- Use to let Objects gain references to Qi4j constructs easily

# DI in Qi4j

- Injection can be done both Fragments and regular Objects
- Use to let Objects gain references to Qi4j constructs easily
  - UI's

# DI in Qi4j

- Injection can be done both Fragments and regular Objects
- Use to let Objects gain references to Qi4j constructs easily
  - UI's
  - Legacy adapters

# DI Example

```
@AppliesTo( Transactional.class )
public class TransactionConcern
    implements InvocationHandler
{
    @Invocation Transactional transactional;
    TransactionManager tm;
    InvocationHandler next;

    public TransactionConcern(@Service TransactionManager tm)
    {
      this.tm = tm;
    }

    public void setNext(@ConcernFor InvocationHandler next)
    {
       this.next = next;
    }

    public Object invoke( Object proxy, Method method, Object[] args )
        throws Throwable
    {

        switch( transactional.value() )
        {

        case REQUIRED:
        {
...
```

# DI matching

# DI matching

:: Many factors are used to match annotations to actual injections

# DI matching

- Many factors are used to match annotations to actual injections
  - Type

# DI matching

- Many factors are used to match annotations to actual injections
  - Type
  - Name

# DI matching

- Many factors are used to match annotations to actual injections
  - Type
  - Name
    - @Service(name=”MyService”)

# DI matching

- Many factors are used to match annotations to actual injections
  - Type
  - Name
    - @Service(name=”MyService”)
  - Generics

# DI matching

- Many factors are used to match annotations to actual injections
  - Type
  - Name
    - @Service(name=”MyService”)
  - Generics
    - Iterable<MyService>

# DI matching

- Many factors are used to match annotations to actual injections
  - Type
  - Name
    - @Service(name="MyService")
  - Generics
    - Iterable<MyService>
      - Iterable.iterator() may have dynamic value

# DI matching

- Many factors are used to match annotations to actual injections
  - Type
  - Name
    - @Service(name="MyService")
  - Generics
    - Iterable<MyService>
      - Iterable.iterator() may have dynamic value
    - Iterable<Person>

# @Service

# @Service

:: Use @Service to declare use of services

# @Service

:: Use @Service to declare use of services
:: Services are usually stateless singletons

# @Service

:: Use @Service to declare use of services
:: Services are usually stateless singletons
:: Use @Service Iterable<MyService> to allow services to come and go

# @Service

:: Use @Service to declare use of services
:: Services are usually stateless singletons
:: Use @Service Iterable<MyService> to allow services to come and go
:: May have names

# @Service

- Use @Service to declare use of services
- Services are usually stateless singletons
- Use @Service Iterable<MyService> to allow services to come and go
- May have names
  - @Service(name="SomeService")

# @Service

:: Use @Service to declare use of services
:: Services are usually stateless singletons
:: Use @Service Iterable<MyService> to allow services to come and go
:: May have names
  :: @Service(name="SomeService")
:: May be optional (default is false)

# @Service

- Use @Service to declare use of services
- Services are usually stateless singletons
- Use @Service Iterable<MyService> to allow services to come and go
- May have names
  - @Service(name="SomeService")
- May be optional (default is false)
  - @Service(optional=true)

# Service Example

```
public class FooMixin
  implements Foo
{
    @Service MyService service;
    @Service Iterable<MyService> services;
    @Service(name="bar") MyService service;
    @Service(optional=true) MyService service;

    void init(@Service MyService service, @Service SomeService otherService)
    {
...
```

# @PropertyField

# @PropertyField

:: Use @PropertyField on Mixin fields to declare properties

# @PropertyField

:: Use @PropertyField on Mixin fields to declare properties
:: Property values are set on CompositeBuilder

# @PropertyField

:: Use @PropertyField on Mixin fields to declare properties
:: Property values are set on CompositeBuilder
:: Values are injected after Mixin constructor has executed

# @PropertyField

:: Use @PropertyField on Mixin fields to declare properties
:: Property values are set on CompositeBuilder
:: Values are injected after Mixin constructor has executed
:: May have names

# @PropertyField

:: Use @PropertyField on Mixin fields to declare properties
:: Property values are set on CompositeBuilder
:: Values are injected after Mixin constructor has executed
:: May have names
    :: If not, then name of field is used

# @PropertyField Example

```
public final class HelloWorldSettingsMixin
    implements HelloWorldSettings
{
    @PropertyField String phrase = "";
    @PropertyField String name = "";

    public String getPhrase()
    {
        return phrase;
    }

    public void setPhrase( String aPhrase )
    {
        phrase = aPhrase;
    }

    public String getName()
    {
        return name;
    }

    public void setName( String aName )
    {
        name = aName;
    }
}
```

# @PropertyField Example

```
// Use Composite interface
CompositeBuilder<HelloWorldComposite> builder;
builder = cbf.newCompositeBuilder( HelloWorldComposite.class );
builder.propertiesOfComposite().setPhrase( "Hello");
builder.propertiesOfComposite().setName( "World");
helloWorld = builder.newInstance();

// Use individual domain interfaces
CompositeBuilder<HelloWorldComposite> builder;
builder = cbf.newCompositeBuilder( HelloWorldComposite.class );
builder.propertiesFor( HelloWorldState.class ).setPhrase( "Hello" );
builder.propertiesFor( HelloWorldState.class ).setName( "World" );
helloWorld = builder.newInstance();


// Use PropertyValue
CompositeBuilder<HelloWorldComposite> builder;
builder = cbf.newCompositeBuilder( HelloWorldComposite.class );

// Set value using name
builder.properties(HelloWorldState.class, PropertyValue.property( "phrase", "Hello"));

// Create property using static import of PropertyValue
builder.properties(HelloWorldState.class, property( "name", "World"));
helloWorld = builder.newInstance();
```

# @PropertyParameter

# @PropertyParameter

:: Use @PropertyParameter on Mixin constructor or method parameters to declare properties

# @PropertyParameter

:: Use @PropertyParameter on Mixin constructor or method parameters to declare properties
:: Property values are set on CompositeBuilder

# @PropertyParameter

:: Use @PropertyParameter on Mixin constructor or method parameters to declare properties
:: Property values are set on CompositeBuilder
:: Values are injection after Mixin constructor has executed

# @PropertyParameter

:: Use @PropertyParameter on Mixin constructor or method parameters to declare properties
:: Property values are set on CompositeBuilder
:: Values are injection after Mixin constructor has executed
:: Must have names

# @PropertyParameter Example

```
public final class HelloWorldSettingsMixin
    implements HelloWorldSettings
{
    String phrase = "";
    String name = "";

    public HelloWorldSettingsMixin(@PropertyParameter("phrase") String phrase,
                                   @PropertyParameter("name") String name)
    {
      this.phrase = phrase;
      this.name = name;
    }

    public String getPhrase()
    {
        return phrase;
    }

    public void setPhrase( String aPhrase )
    {
        phrase = aPhrase;
    }

    public String getName()
    {
        return name;
    }

    public void setName( String aName )
    {
        name = aName;
    }
}
```

# @Adapt

# @Adapt

:: Use @Adapt to declare dependency on object to be used as source in Adapter design pattern

# @Adapt

:: Use @Adapt to declare dependency on object to be used as source in Adapter design pattern
:: Adapted object is set on CompositeBuilder

# @Adapt

:: Use @Adapt to declare dependency on object to be used as source in Adapter design pattern
:: Adapted object is set on CompositeBuilder
:: May be optional

# @Adapt Example

```
public final class MyServiceAdapterMixin
    implements MyService
{
    @Adapt OtherService service;

    public String doThings()
    {
        return service.otherMethod();
    }
}

OtherService otherService = ...;
CompositeBuilder<HelloWorldComposite> builder;
builder = cbf.newCompositeBuilder( MyComposite.class );
builder.adapt(otherService);
myService = builder.newInstance();

myService.doThings();
```

# @Decorate

# @Decorate

:: Use @Decorate to declare dependency on object to be used as source in Decorator design pattern

# @Decorate

- Use @Decorate to declare dependency on object to be used as source in Decorator design pattern
- Decorated object is set on CompositeBuilder

# @Decorate

- Use @Decorate to declare dependency on object to be used as source in Decorator design pattern
- Decorated object is set on CompositeBuilder
- Composite must implement interface of decorated object

# @Decorate Example

```
public final class MyServiceDecoratorMixin
    implements MyService
{
    @Decorate MyService service;

    public String doThings()
    {
        return "Service sez:" + service.doThings();
    }
}

MyService otherService = ...;
CompositeBuilder<HelloWorldComposite> builder;
builder = cbf.newCompositeBuilder( MyComposite.class );
builder.decorate(otherService);
myService = builder.newInstance();

myService.doThings();
```

# @Structure

# @Structure

- Use @Structure to declare dependency on structural resources

# @Structure

- Use @Structure to declare dependency on structural resources
- Provides access to the CompositeBuilderFactory, Module, Layer, and Application which the Fragment exists in

# @Structure Example

```
public final class HasDisplayIconMixin
    implements HasDisplayIcon
{
    @Structure CompositeBuilderFactory factory;
    @Structure Layer layer;

    public DisplayIcon getDisplayIcon()
    {
        if (layer.getUsages().size() > 0)
          return null; // Don't create if we're not in the UI layer
        CompositeBuilder<MyServicePanelComposite> builder;
        builder = factory.newCompositeBuilder( DisplayIcon.class );
        return builder.newInstance();
    }
}
```

# @Invocation

# @Invocation

:: Use @Invocation to declare dependency on invocation-specific objects

# @Invocation

- Use @Invocation to declare dependency on invocation-specific objects
  - Method

# @Invocation

- Use @Invocation to declare dependency on invocation-specific objects
  - Method
  - AnnotatedElement

# @Invocation

- Use @Invocation to declare dependency on invocation-specific objects
  - Method
  - AnnotatedElement
  - InvocationContext

# @Invocation

- Use @Invocation to declare dependency on invocation-specific objects
  - Method
  - AnnotatedElement
  - InvocationContext
  - Specific annotations

# @Invocation

- Use @Invocation to declare dependency on invocation-specific objects
  - Method
  - AnnotatedElement
  - InvocationContext
  - Specific annotations
- Only legal in Modifiers

# @Invocation Example

```
@AppliesTo( Transactional.class )
public class TransactionConcern
    implements InvocationHandler
{
    @Service TransactionManager tm;
    @Invocation Transactional transactional;
    @ConcernFor InvocationHandler next;

    public Object invoke( Object proxy, Method method, Object[] args )
        throws Throwable
    {

        switch( transactional.value() )
        {

        case REQUIRED:
        {
...
```

# @ThisCompositeAs

# @ThisCompositeAs

:: Use @ThisCompositeAs to declare dependency on other Mixins in the same Composite

# @ThisCompositeAs

- Use @ThisCompositeAs to declare dependency on other Mixins in the same Composite
- Type must be extended by Composite interface

# @ThisCompositeAs

- Use @ThisCompositeAs to declare dependency on other Mixins in the same Composite
- Type must be extended by Composite interface
  - Or, some Mixin must be declared which can handle it (=private Mixins)

# @ThisCompositeAs

```
public class HelloWorldBehaviourMixin
    implements HelloWorldBehaviour
{
    @ThisCompositeAs HelloWorldState state;

    public String say()
    {
        return state.getPhrase() + " " + state.getName();
    }
}
```

# Private Mixins Example

```
public class HelloWorldBehaviourMixin
    implements HelloWorldBehaviour
{
    @ThisCompositeAs HelloWorldBehaviourMixin.HelloWorldState state;

    public String say()
    {
        return state.getPhrase() + " " + state.getName();
    }

    public interface HelloWorldState
    {
       void setPhrase(String phrase);
       String getPhrase();
       void setName(String name);
       String getName();
    }
}
```

# @ConcernFor

# @ConcernFor

:: Use @ConcernFor in Concerns to have the next Concern or Mixin in the invocation chain injected

# @ConcernFor

- Use @ConcernFor in Concerns to have the next Concern or Mixin in the invocation chain injected
- Type must be an interface of the Composite which is also implemented by the Concern itself

# @ConcernFor

- Use @ConcernFor in Concerns to have the next Concern or Mixin in the invocation chain injected
- Type must be an interface of the Composite which is also implemented by the Concern itself
  - Can be InvocationHandler for Generic Concerns

# @ConcernFor

- Use @ConcernFor in Concerns to have the next Concern or Mixin in the invocation chain injected
- Type must be an interface of the Composite which is also implemented by the Concern itself
  - Can be InvocationHandler for Generic Concerns
- Concerns MUST have one-and-only-one @ConcernFor declaration

# @ConcernFor Example

```
public abstract class HelloWorldStateConcern
    implements HelloWorldComposite
{
    @ConcernFor HelloWorldComposite next;

    public void setPhrase( String phrase )
        throws IllegalArgumentException
    {
        if( phrase == null )
        {
            throw new IllegalArgumentException( "Phrase may not be null" );
        }

        next.setPhrase( phrase );
    }

    public void setName( String name )
        throws IllegalArgumentException
    {
        if( name == null )
        {
            throw new IllegalArgumentException( "Name may not be null" );
        }

        next.setName( name );
    }
}
```

# @SideEffectFor

# @SideEffectFor

:: Use @SideEffectFor in SideEffects to have the a reference to the result of the invocation injected

# @SideEffectFor

- Use @SideEffectFor in SideEffects to have the a reference to the result of the invocation injected
- Type must be an interface of the Composite which is also implemented by the SideEffect itself

# @SideEffectFor

- Use @SideEffectFor in SideEffects to have the a reference to the result of the invocation injected
- Type must be an interface of the Composite which is also implemented by the SideEffect itself
  - Can be InvocationHandler for Generic SideEffects

# @SideEffectFor

- Use @SideEffectFor in SideEffects to have the a reference to the result of the invocation injected
- Type must be an interface of the Composite which is also implemented by the SideEffect itself
  - Can be InvocationHandler for Generic SideEffects
- SideEffects MUST have one-and-only-one @SideEffectFor declaration

# @SideEffectFor

- Use @SideEffectFor in SideEffects to have the a reference to the result of the invocation injected
- Type must be an interface of the Composite which is also implemented by the SideEffect itself
  - Can be InvocationHandler for Generic SideEffects
- SideEffects MUST have one-and-only-one @SideEffectFor declaration
- Not required to actually call the injected reference

# @SideEffectFor Example

```
@AppliesTo( CountCalls.class )
public class CountCallsSideEffect
    implements InvocationHandler
{
    private @SideEffectFor InvocationHandler next;
    private @ThisCompositeAs Counter counter;

    public Object invoke( Object proxy, Method method, Object[] args ) throws Throwable
    {
        counter.increment();
        return next.invoke( proxy, method, args );
    }
}
```

# @Entity

# @Entity

:: Use @Entity in Fragments to inject references to individual Entities, Entity collections or QueryBuilders for Entities

# @Entity

- Use @Entity in Fragments to inject references to individual Entities, Entity collections or QueryBuilders for Entities
- Type must be a Composite which extends EntityComposite

# @Entity

- Use @Entity in Fragments to inject references to individual Entities, Entity collections or QueryBuilders for Entities
- Type must be a Composite which extends EntityComposite
- Type may use generics for Iterable or QueryBuilders

# @Entity

- Use @Entity in Fragments to inject references to individual Entities, Entity collections or QueryBuilders for Entities
- Type must be a Composite which extends EntityComposite
- Type may use generics for Iterable or QueryBuilders
  - @Entity Iterable<PersonComposite> iter;

# @Entity

- Use @Entity in Fragments to inject references to individual Entities, Entity collections or QueryBuilders for Entities
- Type must be a Composite which extends EntityComposite
- Type may use generics for Iterable or QueryBuilders
  - @Entity Iterable<PersonComposite> iter;
  - @Entity QueryBuilder<PersonComposite> qb;

# @Entity Example

```
public class TimelineMixin
    implements Timeline
{
    QueryBuilder<EventComposite> events;

    public TimelineMixin( @Entity QueryBuilder<EventComposite> timelineEvents,
                          @ThisCompositeAs Period period )
    {
        Period p = timelineEvents.parameter( Period.class );
        this.events = timelineEvents.where( ge( p.getStart(), period.getStart() ) )
                      .where( le( p.getEnd(), period.getEnd() ) )
                      .orderBy( p.getStart() );
    }

    public QueryBuilder<EventComposite> getEvents()
    {
        return events;
    }
}
```

# Questions?