NanoContainer
Scripted Dependency Injection
by Paul Hammant and Aslak Hellesoy, one of its lead developers.
The Presenter(s)

- Paul has been active with Avalon (the first IoC framework) years before Dependency Injection was coined. He’s written many (stupid) IoC OSS tools over six years.

- Aslak is famous for XDoclet, Middlegen, Damage Control, QDox and eXtreme ideas like Ashcroft.
Quick facts and givens

- We started PicoContainer and NanoContainer in 2003
- NanoContainer essentially extends PicoContainer (think Russian dolls)
- We recommend Constructor Dependency Injection (CDI) over Setter (SDI)
- You should already know why singletons in large enterprise apps are bad
- You might also be tired of writing XML to anyone’s specification
- PicoContainer has no meta-data, NanoContainer is ‘open’ to multiple meta implementations
Firstly:
IoC Crash Course

Containers control
Components
Container Instantiated
Phase 1: Registration
Circle registered (needs Star)
Star registered
public class Circle

    public Circle(Star star) {
    
    }

}

public class Star {

}
Phase 2: Instantiation
Star instantiated

Star instantiated
Circle instantiated, referring to Star
Container often disposed of
IoC Crash Course

Container hierarchies:
Scoping of dependencies
Instantiation
IoC Recap

- Containers instantiate registered components
- Component dependencies are satisfied by their container
- Components shouldn’t reference their containers
- Life-cycle (start/stop) and configuration are also controlled by the container
- Hierarchies blah blah
Introducing NanoContainer via (simplified) code examples..
MutablePicoContainer pico = new DefaultPicoContainer();
pico.register(Map.class, HashMap.class);
pico.register(NeedsMap.class);
Object nm = pico.getInstance(NeedsMap.class);
NanoContainer - in code

NanoContainer nano
    = new DefaultNanoContainer();
    nano.register("java.util.Map.class",
                  "java.util.HashMap.class");
    nano.register("my.NeedsMap.class");
    Object nm
        = nano.getInstance("my.NeedsMap.class");
<container>
  <component key="java.util.Map"
    class="java.util.HashMap"/>
  <component class="my.NeedsMap"/>
</container>
Groovy NanoContainer

container {
    component(key: "java.util.Map",
              class: "java.util.HashMap")
    component(class: "my.NeedsMap")
}

Groovy is a language though:

```groovy
container {
    component(key: java.util.Map,
               class: java.util.HashMap)
    component(class: my.NeedsMap)
}
```

* can reference classes in classpath or classloader...
Comps from diff classloaders

container {
    classLoader {
        classPathElement(path: "foo.jar")
        component(key: java.util.Map, class: "FooMap")
    }
    component(class: my.NeedsMap)
}
container {
  classLoader {
    classPathElement(path: "foo.jar") {
      grant(new SocketPermission ("google.com:80"))
    }
    component(class: "FooMap")
  }
  component(class: "my.NeedsMap")
}
Containers hierarchies

container {
    container {
        component(class:my.NeedsMap)
    }
    component(class:FooMap)
    component(class:CantRequireNeedsMap)
}

Implementation hiding

```java
container(class: ImplHidingContainer) {
    component(key: Map, class: FooMap)
    component(
        class: NeedsMapButCantCastToFooMap)
}
```
NanoContainer Recap

- Java, XML, Groovy
- Unshown were JavaScript, Jython, Beanshell
- Implementation hiding
- Container Hierarchies
- Standalone or embedded (all markups/langs)

XML is like the Groovy one, but hurts your eyes some more cos of the <angle brackets/> you can image the others.. .. they're less declarative
A different view of the classloader hierarchy...
NanoContainer Booting

- NanoContainer-Boot started via a main method.
NanoContainer Booting

- Boot creates a tree of two classloaders, ensures it is not part of that tree.
NanoContainer Booting

- Boot hands control to NanoContainer, effectively taking itself out of the picture
NanoContainer Booting

- NanoContainer (and Groovy etc) create more of the classloader hierarchy, keeping itself out of the resulting classloader tree.
NanoContainer Booting

- From the point of view of the component, NanoContainer, Groovy (etc) and Boot are invisible.
Related NanoContainer Components

- NanoWar - some bindings to web frameworks (Struts, Webwork plus ‘NanoWeb’ our own tiny one)
- Hibernate bindings (2 & 3)
- Remoting, etc.

Yup, all this in development since 2003
What Next?
In-lined Web-apps

```java
container {
    component(BizBean)
    webContainer(host: "*", port: 80) {
        context("foo/") {
            servlet(“bar.xyz”,
                class: BarServletNeedsBizBean)
        }
    }
}
```

No web.xml obviously
Tiering

tier(name: "persistence") {
  exposedComponent(PersistenceFoo)
  component(FooSerializer)
  component(FooRepicator)
  tier(name: "biz") {
    component(BizBeanUsingFoo)
    component(BizBean2)
  }
}
Publishing

publication {
  soapPublisher(host: "*", port: 8080)
  container {
    component(my.ZipCodeServiceImpl) {
      publish(name: "zipCodeService")
    }
    component(NeedsZipServices)
  }
}
Subscription

subscription {
    soapSubscriber(host: "beatle", port: 8080)
    container {
        remoteComponent {
            (key: my.ZipCodeService.me,
            name: "zipCodeService")
        component(AlsoNeedsZipServices)
    }
}
Spring vs Pico/Nano

- Could use Pico and Nano for some usages Spring is used for.
- Spring is more comprehensive for J2EE though (we’re chasing it).
- You can use Pico and Nano with Spring.
- Pico/Nano has an audience amongst specialist embedders.