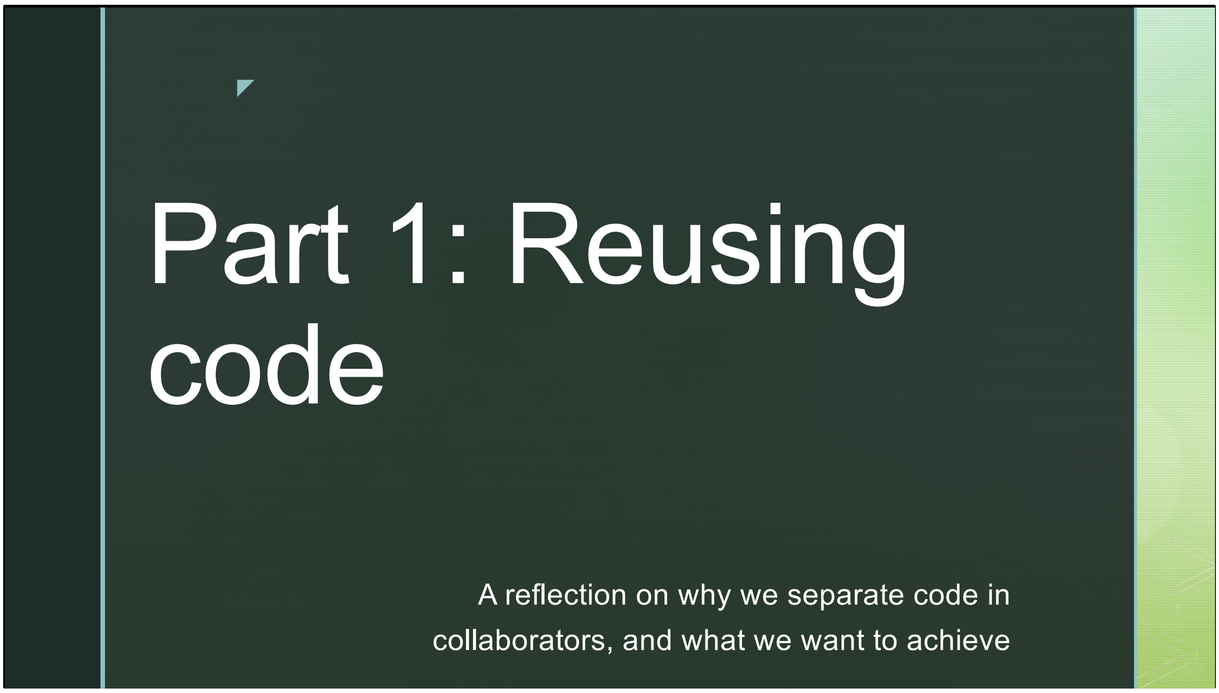
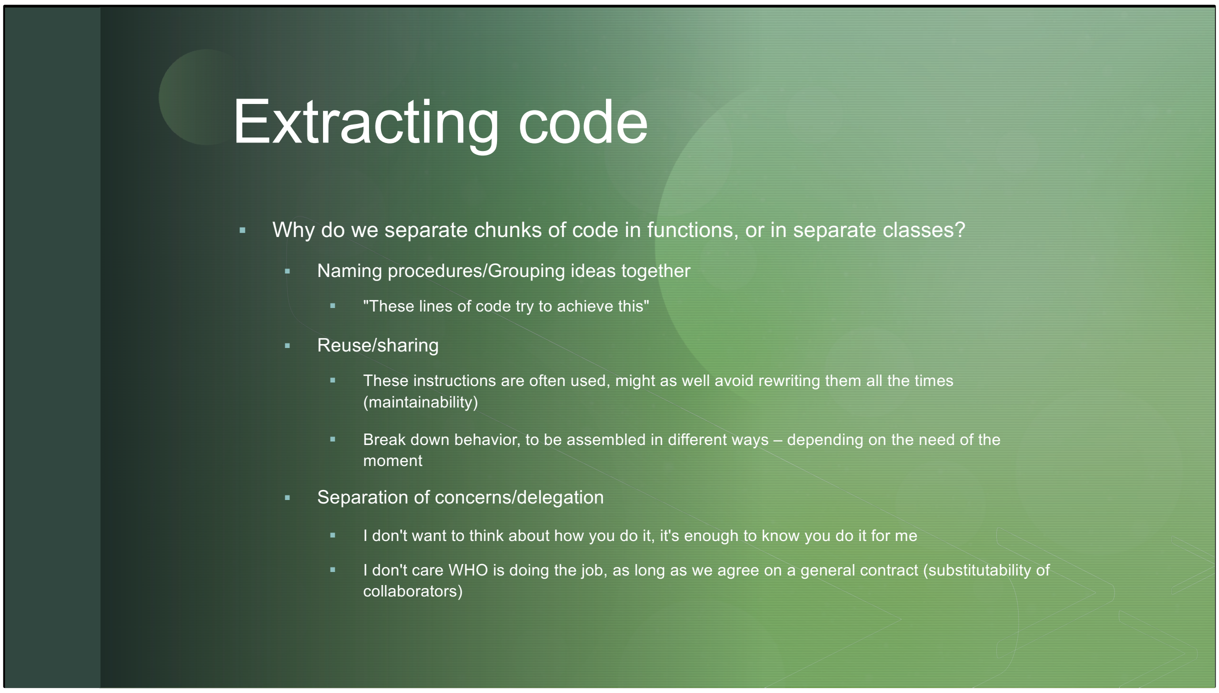
From Statics to DI



Part 1: Reusing code

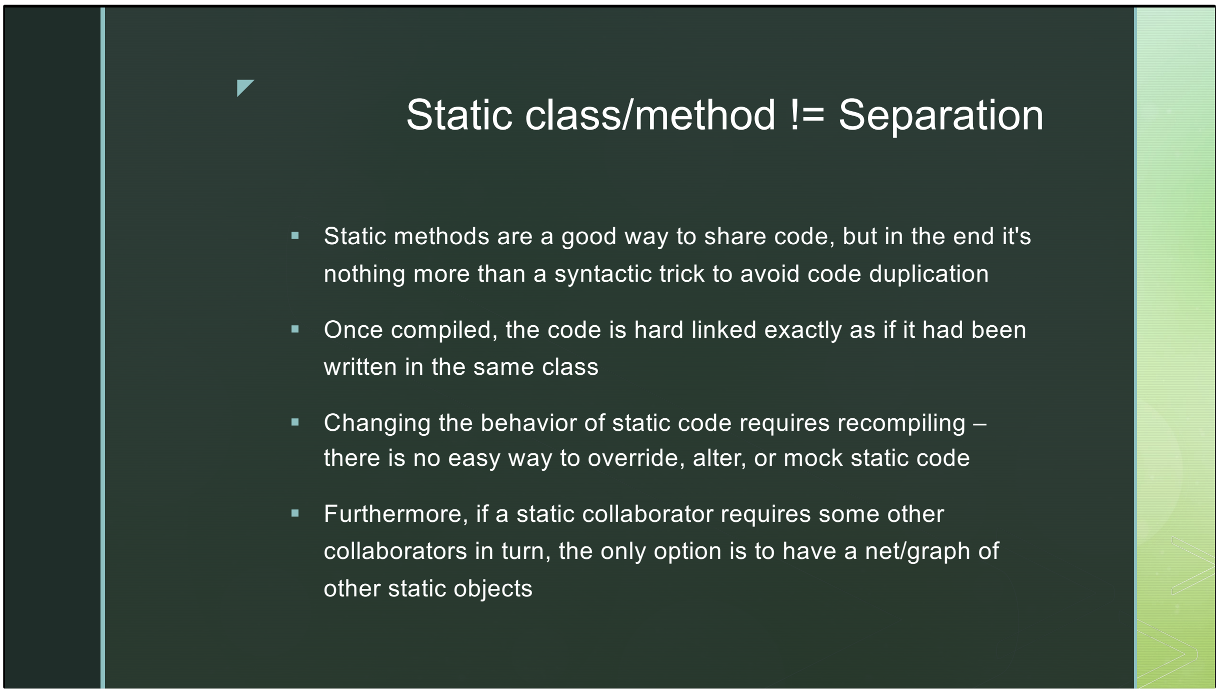
A reflection on why we separate code in collaborators, and what we want to achieve



Extracting code

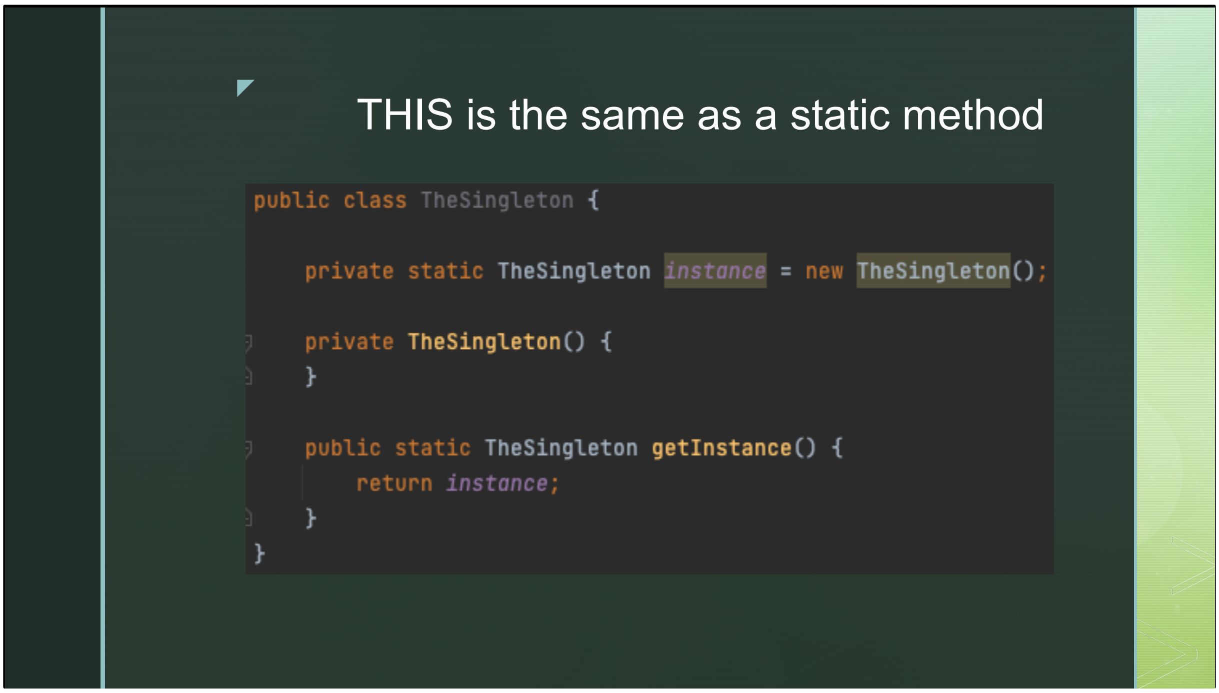
- Why do we separate chunks of code in functions, or in separate classes?
  - Naming procedures/Grouping ideas together
    - "These lines of code try to achieve this"
  - Reuse/sharing
    - These instructions are often used, might as well avoid rewriting them all the times (maintainability)
    - Break down behavior, to be assembled in different ways – depending on the need of the moment
  - Separation of concerns/delegation
    - I don't want to think about how you do it, it's enough to know you do it for me
    - I don't care WHO is doing the job, as long as we agree on a general contract (substitutability of collaborators)

As software grows beyond a couple of lines of code, we tend to group chunks of it together, be it by separating them with simple comments or encapsulating them first in a separate method, then possibly in a separate class, then abstracting away the implementation by defining a contract (interface)



Static class/method != Separation

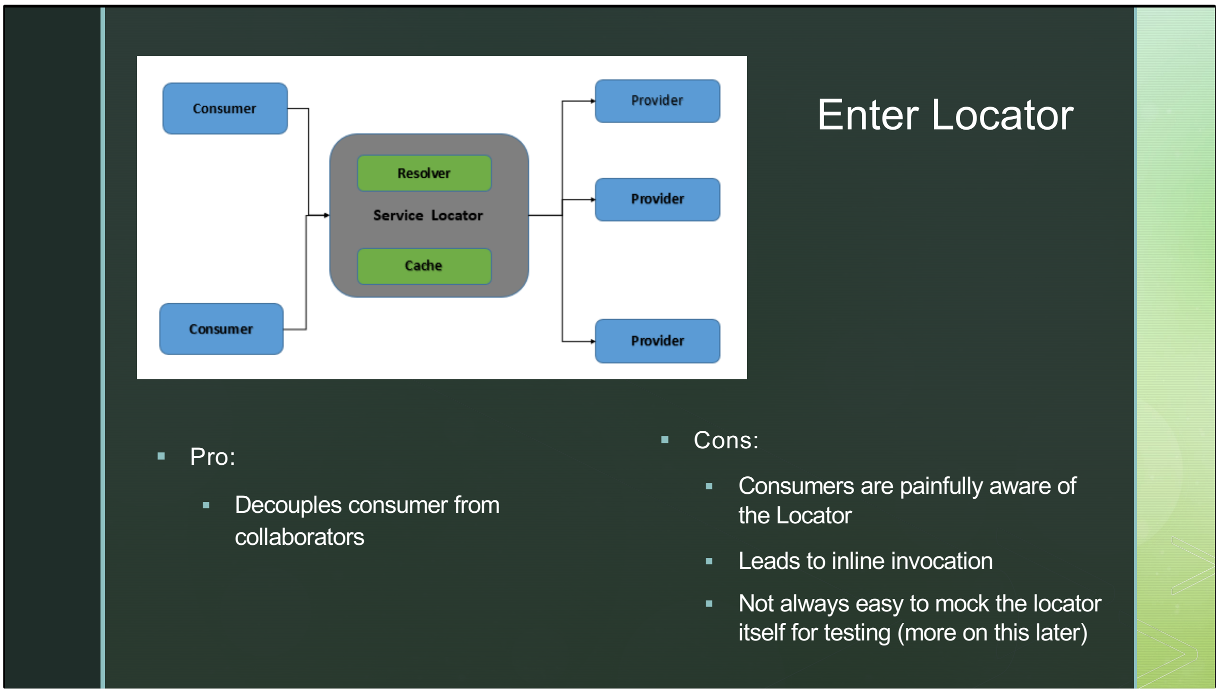
- Static methods are a good way to share code, but in the end it's nothing more than a syntactic trick to avoid code duplication
- Once compiled, the code is hard linked exactly as if it had been written in the same class
- Changing the behavior of static code requires recompiling – there is no easy way to override, alter, or mock static code
- Furthermore, if a static collaborator requires some other collaborators in turn, the only option is to have a net/graph of other static objects



THIS is the same as a static method

```java
public class TheSingleton {

    private static TheSingleton instance = new TheSingleton();

    private TheSingleton() {
    }

    public static TheSingleton getInstance() {
        return instance;
    }
}
```

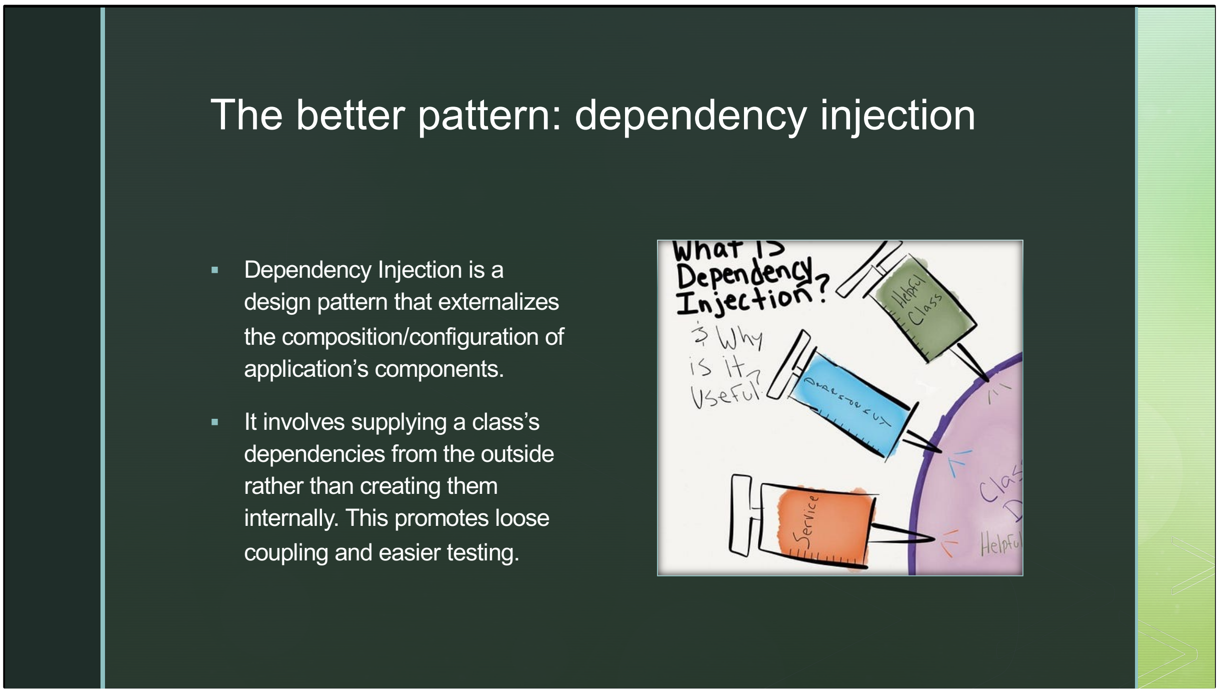
The fact that it is a separate class should not fool you. The only way to build this class is through the static "getInstance()" method – which means it cannot be replaced. https://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/



Enter Locator

- Pro:
  - Decouples consumer from collaborators
- Cons:
  - Consumers are painfully aware of the Locator
  - Leads to inline invocation
  - Not always easy to mock the locator itself for testing (more on this later)

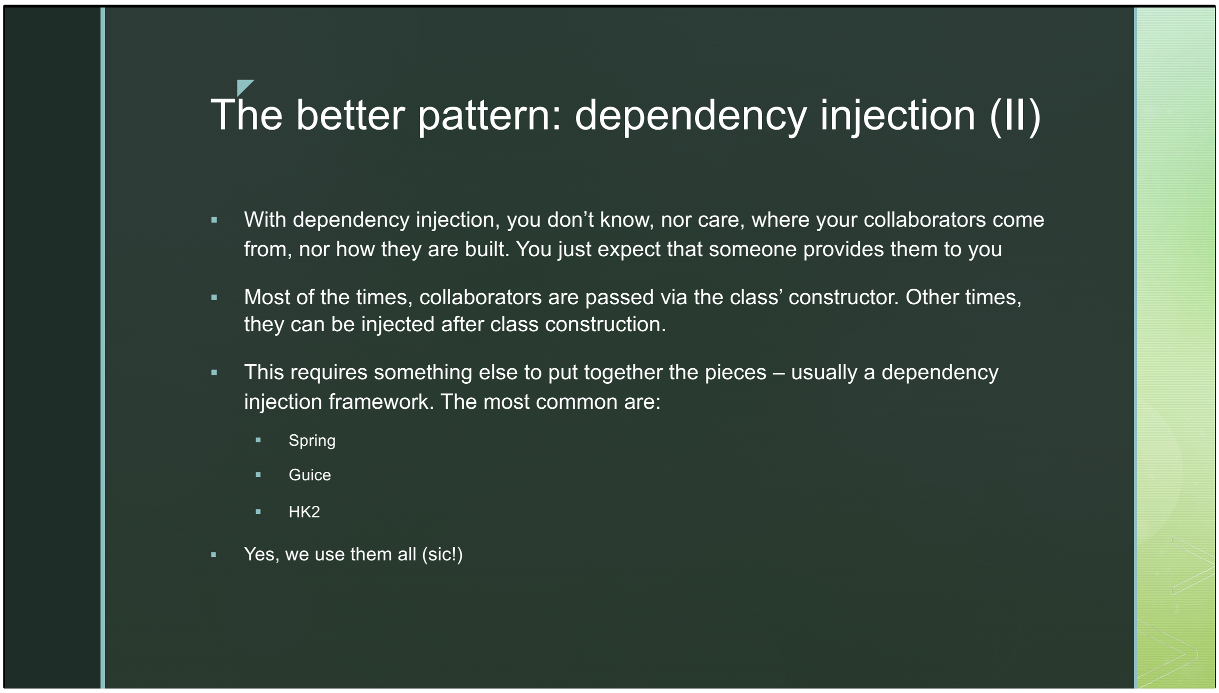
Sometimes referred to as an Anti-Pattern – because of its limitation and initial lure, which seems to solve the coupling problem in the short term
Actually an excellent solution to begin moving old coupled code towards a dependency inversion-based solution, as long as it is used as a stepping stone and not as a final solution



The better pattern: dependency injection

- Dependency Injection is a design pattern that externalizes the composition/configuration of application's components.
- It involves supplying a class's dependencies from the outside rather than creating them internally. This promotes loose coupling and easier testing.

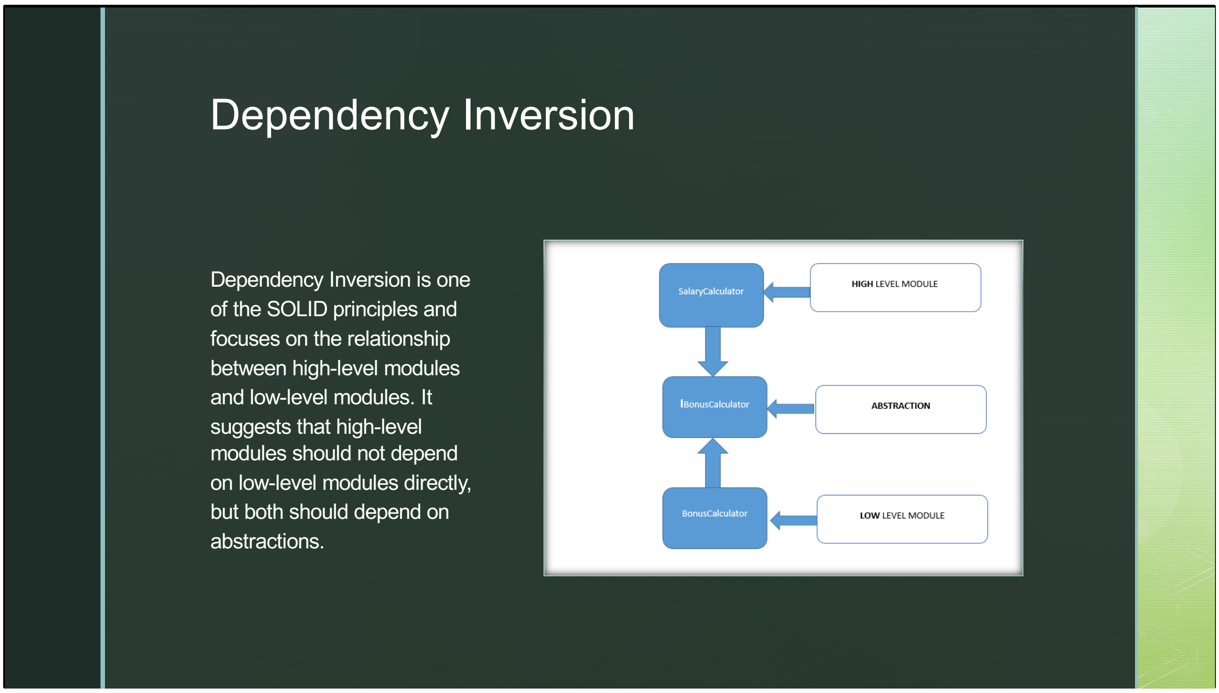
https://blog.stackademic.com/understanding-the-difference-between-dependency-inversion-and-dependency-injection-in-c-c9934ee7f6f5
https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/
https://medium.com/@mena.meseha/dependency-injection-complete-guide-14b5ee4e47eb



The better pattern: dependency injection (II)

- With dependency injection, you don’t know, nor care, where your collaborators come from, nor how they are built. You just expect that someone provides them to you
- Most of the times, collaborators are passed via the class’ constructor. Other times, they can be injected after class construction.
- This requires something else to put together the pieces – usually a dependency injection framework. The most common are:
  - Spring
  - Guice
  - HK2
- Yes, we use them all (sic!)

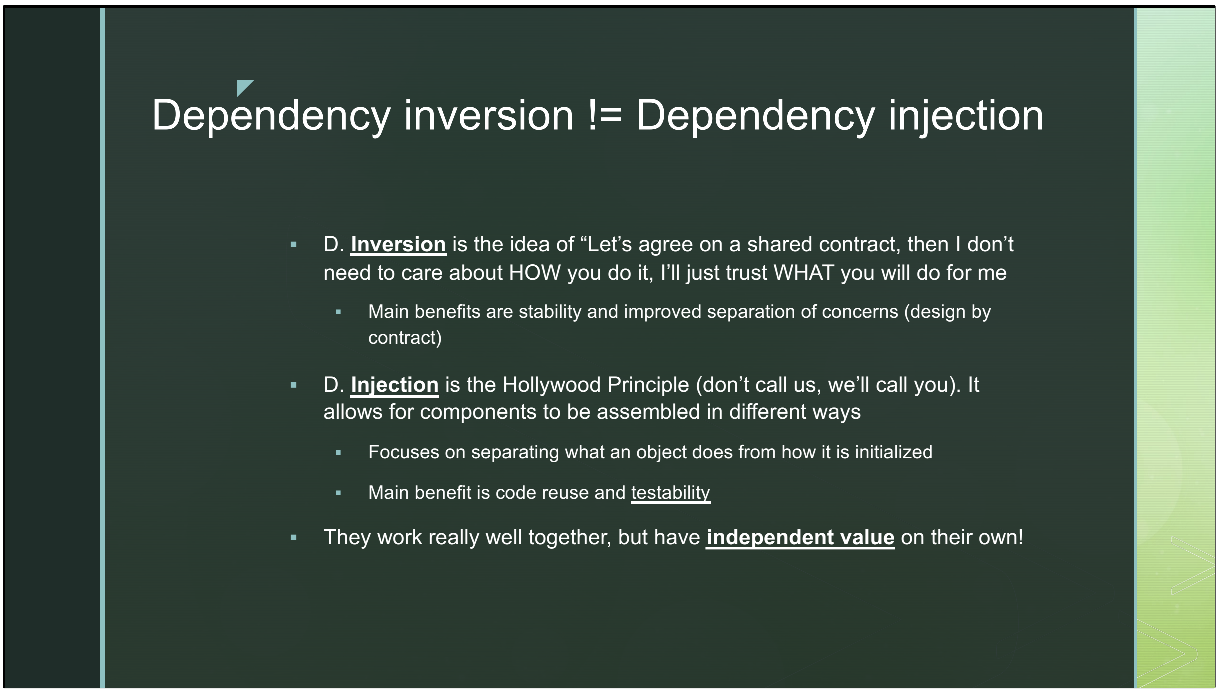
https://blog.stackademic.com/understanding-the-difference-between-dependency-inversion-and-dependency-injection-in-c-c9934ee7f6f5
https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/
https://medium.com/@mena.meseha/dependency-injection-complete-guide-14b5ee4e47eb



Dependency Inversion

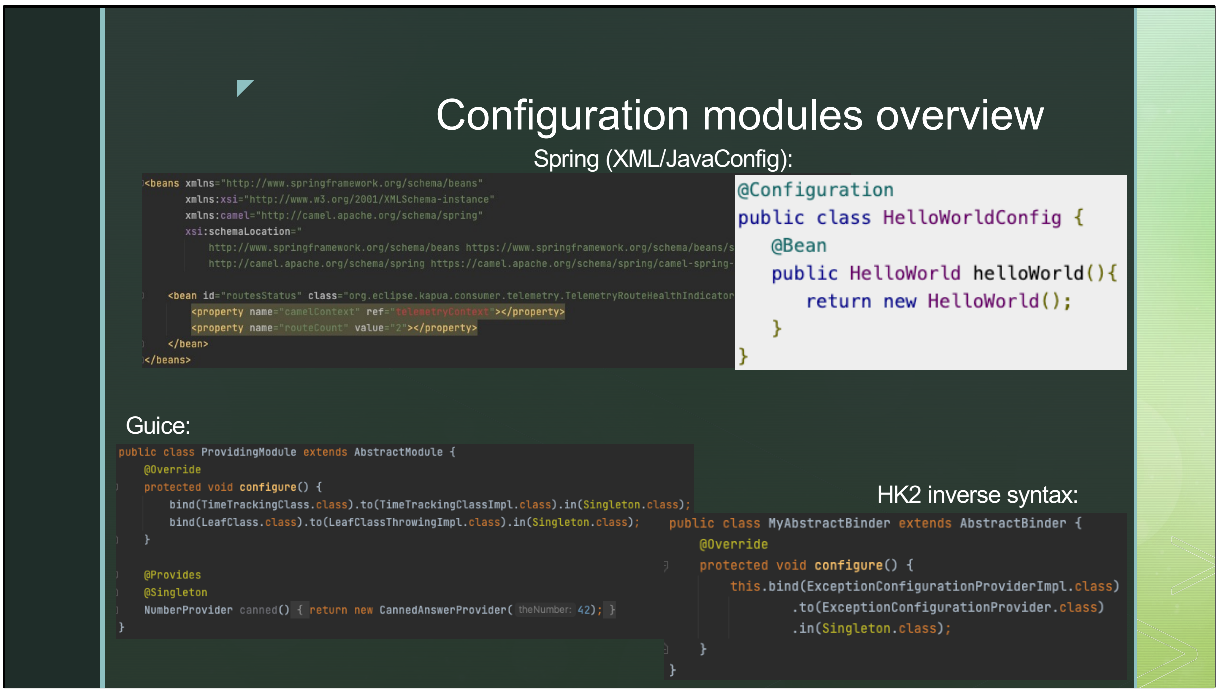
Dependency Inversion is one of the SOLID principles and focuses on the relationship between high-level modules and low-level modules. It suggests that high-level modules should not depend on low-level modules directly, but both should depend on abstractions.

It promotes component's interchangeably



Dependency inversion != Dependency injection

- D. **Inversion** is the idea of “Let’s agree on a shared contract, then I don’t need to care about HOW you do it, I’ll just trust WHAT you will do for me
  - Main benefits are stability and improved separation of concerns (design by contract)
- D. **Injection** is the Hollywood Principle (don’t call us, we’ll call you). It allows for components to be assembled in different ways
  - Focuses on separating what an object does from how it is initialized
  - Main benefit is code reuse and testability
- They work really well together, but have **independent value** on their own!



Configuration modules overview

Spring (XML/JavaConfig):

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans https://www.springframework.org/schema/beans/s
         http://camel.apache.org/schema/spring https://camel.apache.org/schema/spring/camel-spring-

    <bean id="routesStatus" class="org.eclipse.kapua.consumer.telemetry.TelemetryRouteHealthIndicator
        <property name="camelContext" ref="telemetryContext"></property>
        <property name="routeCount" value="2"></property>
    </bean>
</beans>
```

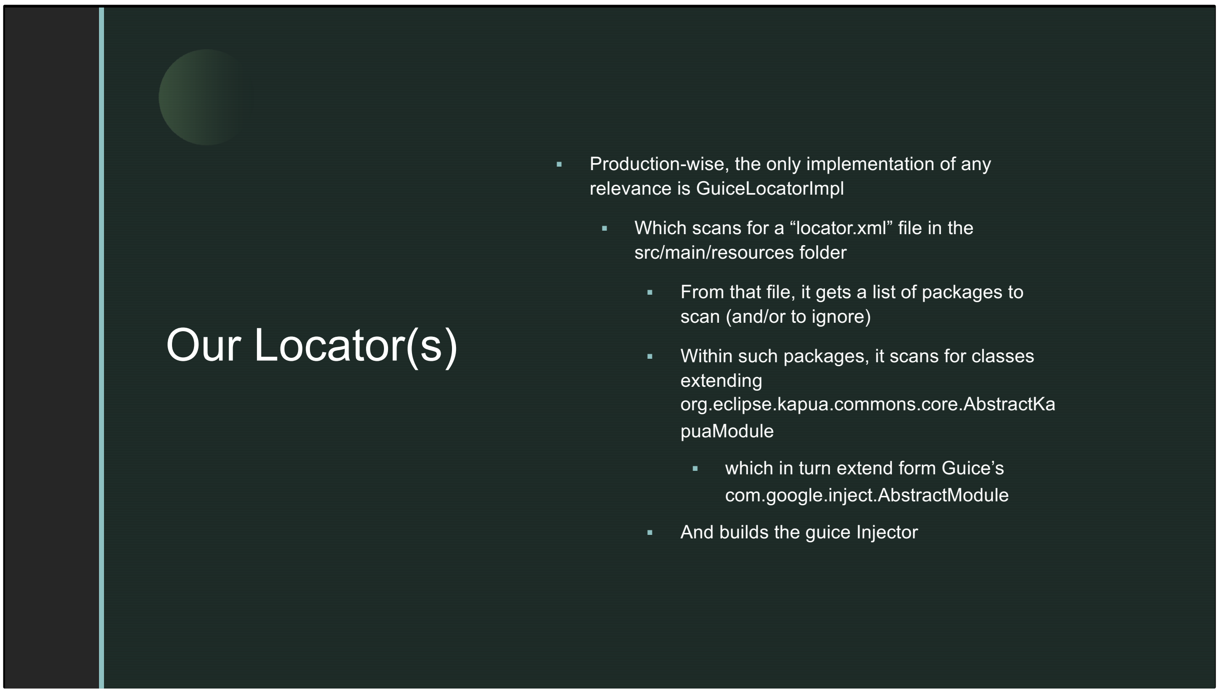
```java
@Configuration
public class HelloWorldConfig {
   @Bean
   public HelloWorld helloWorld(){
      return new HelloWorld();
   }
}
```

Guice:

```java
public class ProvidingModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(TimeTrackingClass.class).to(TimeTrackingClassImpl.class).in(Singleton.class);
        bind(LeafClass.class).to(LeafClassThrowingImpl.class).in(Singleton.class);
    }

    @Provides
    @Singleton
    NumberProvider canned() { return new CannedAnswerProvider( theNumber: 42); }
}
```

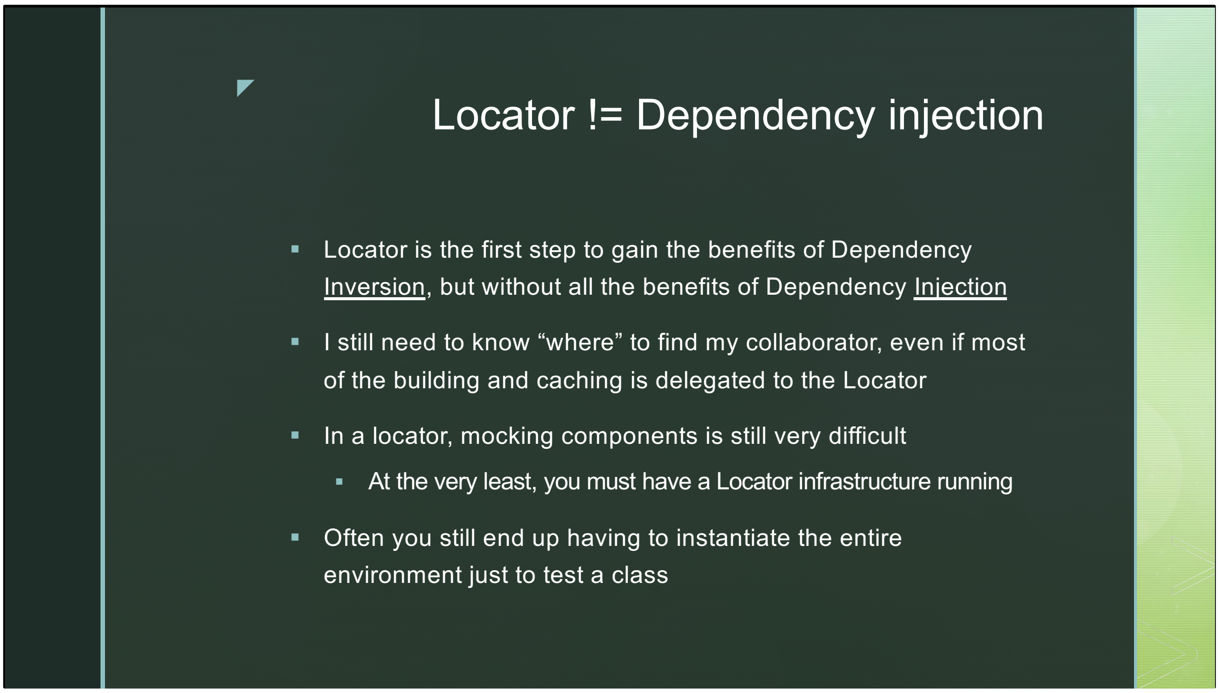
HK2 inverse syntax:

```java
public class MyAbstractBinder extends AbstractBinder {
    @Override
    protected void configure() {
        this.bind(ExceptionConfigurationProviderImpl.class)
                .to(ExceptionConfigurationProvider.class)
                .in(Singleton.class);
    }
}
```



Our Locator(s)

- Production-wise, the only implementation of any relevance is GuiceLocatorImpl
  - Which scans for a “locator.xml” file in the src/main/resources folder
    - From that file, it gets a list of packages to scan (and/or to ignore)
    - Within such packages, it scans for classes extending org.eclipse.kapua.commons.core.AbstractKapuaModule
      - which in turn extend form Guice’s com.google.inject.AbstractModule
    - And builds the guice Injector

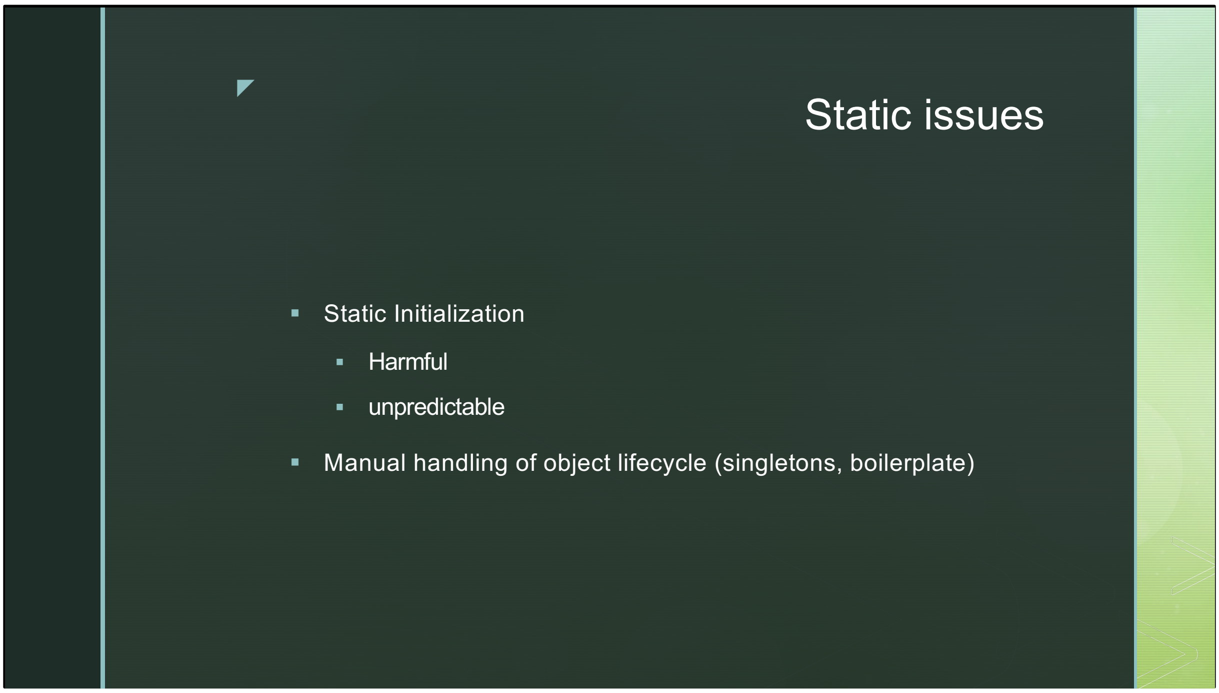


Locator != Dependency injection

- Locator is the first step to gain the benefits of Dependency Inversion, but without all the benefits of Dependency Injection
- I still need to know “where” to find my collaborator, even if most of the building and caching is delegated to the Locator
- In a locator, mocking components is still very difficult
  - At the very least, you must have a Locator infrastructure running
- Often you still end up having to instantiate the entire environment just to test a class

Part 2: reasons for change

With a brief digression on testing

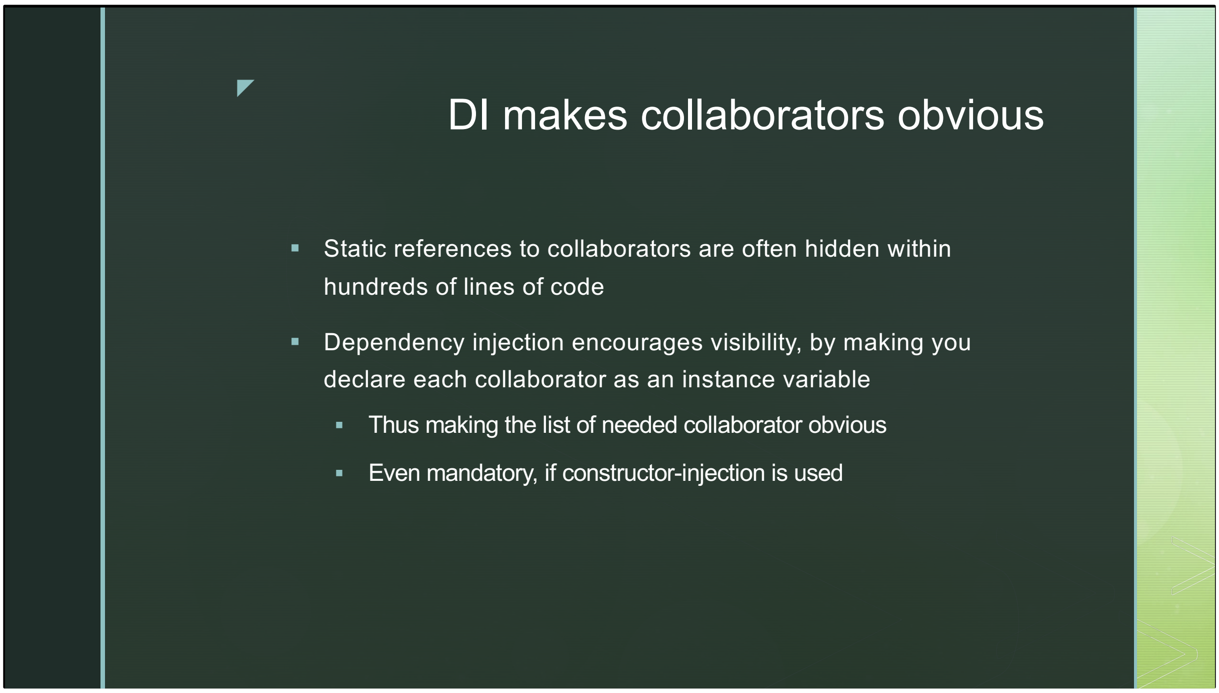


Static issues

- Static Initialization
  - Harmful
  - unpredictable
- Manual handling of object lifecycle (singletons, boilerplate)

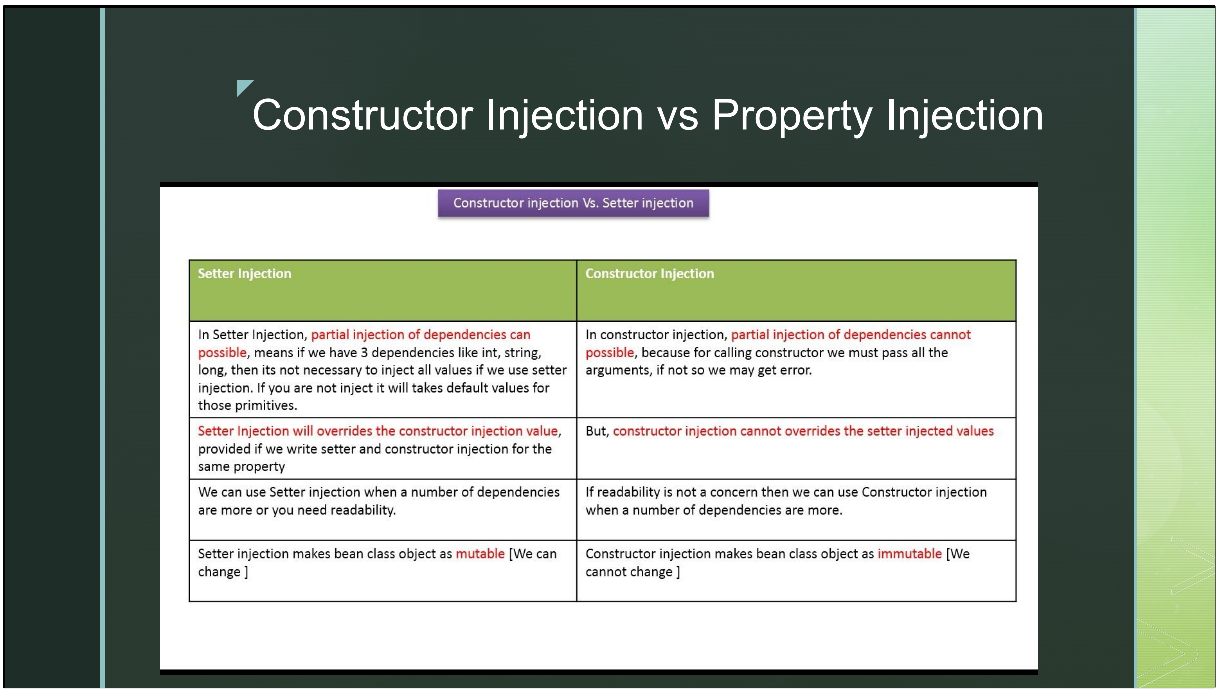
https://medium.com/att-israel/should-you-avoid-using-static-ae4b58ca1de5

https://pangin.pro/posts/computation-in-static-initializer



DI makes collaborators obvious

- Static references to collaborators are often hidden within hundreds of lines of code
- Dependency injection encourages visibility, by making you declare each collaborator as an instance variable
  - Thus making the list of needed collaborator obvious
  - Even mandatory, if constructor-injection is used



Constructor Injection vs Property Injection

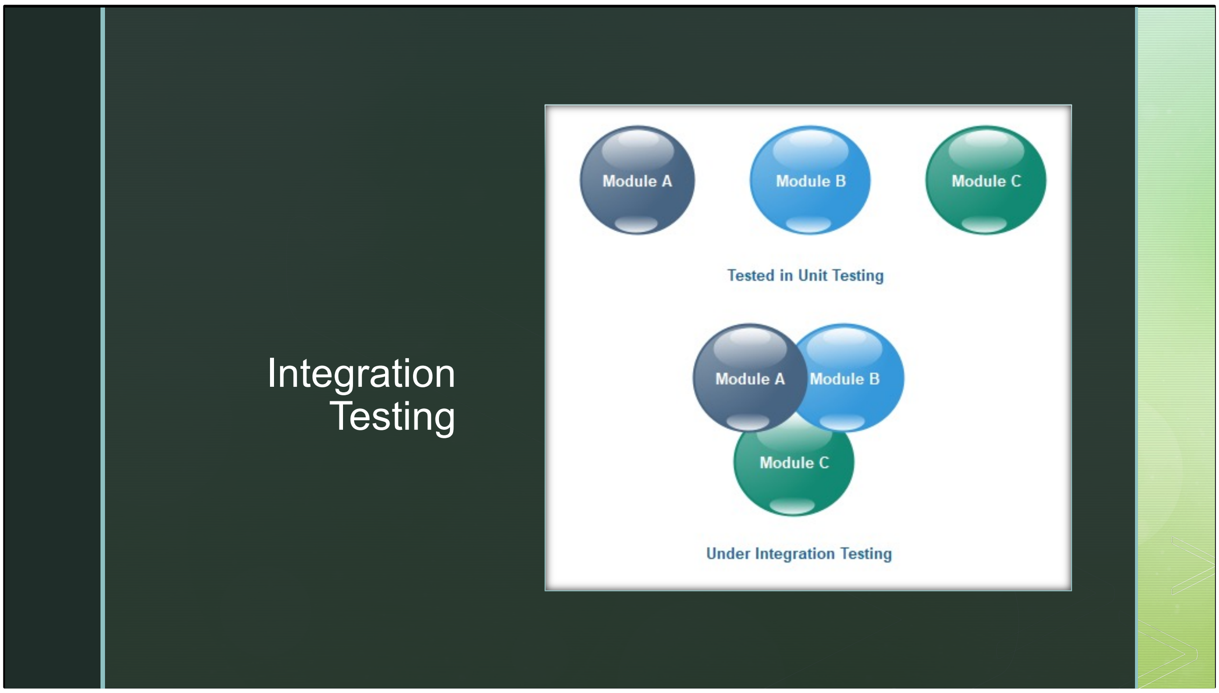
Constructor injection Vs. Setter injection

Setter Injection	Constructor Injection
In Setter Injection, partial injection of dependencies can possible, means if we have 3 dependencies like int, string, long, then its not necessary to inject all values if we use setter injection. If you are not inject it will takes default values for those primitives.	In constructor injection, partial injection of dependencies cannot possible, because for calling constructor we must pass all the arguments, if not so we may get error.
Setter Injection will overrides the constructor injection value, provided if we write setter and constructor injection for the same property	But, constructor injection cannot overrides the setter injected values
We can use Setter injection when a number of dependencies are more or you need readability.	If readability is not a concern then we can use Constructor injection when a number of dependencies are more.
Setter injection makes bean class object as mutable [We can change]	Constructor injection makes bean class object as immutable [We cannot change]

https://ramj2ee.blogspot.com/2018/07/what-is-difference-between-constructor_9.html

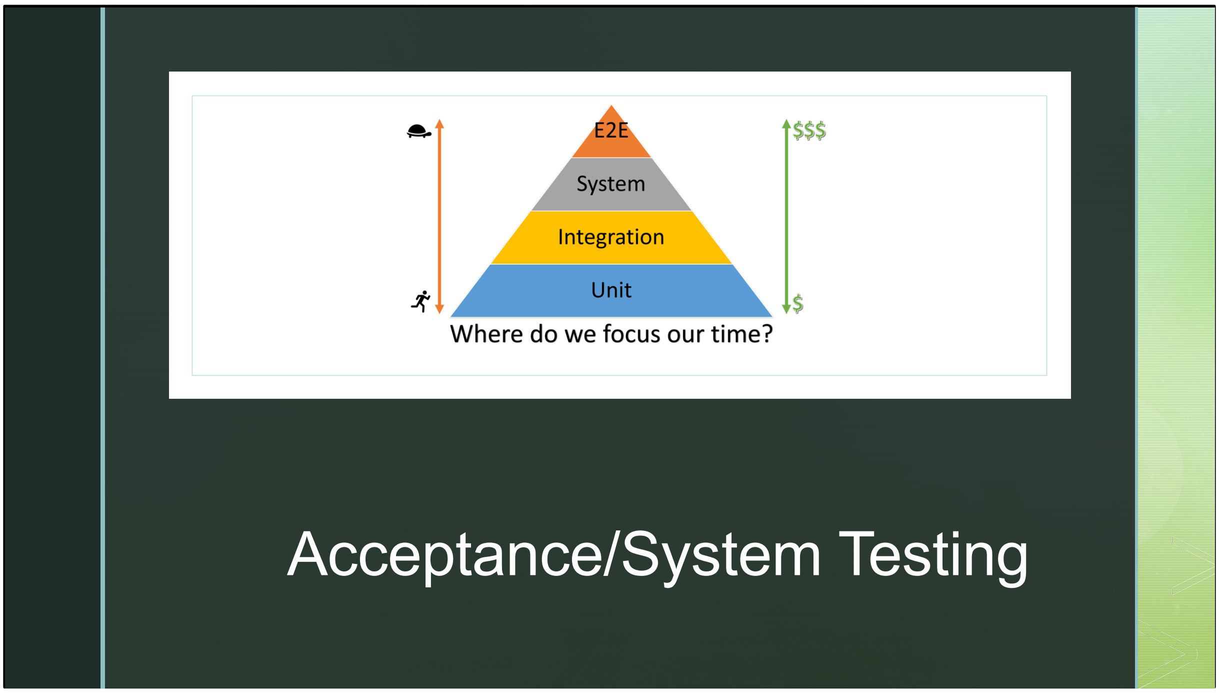
Unit Testing

Testing in isolation. Each component is tested in a void, to see if it reacts correctly to external stimuli. ALL external collaborators are mocked, or closely manipulated to behave in piloted ways.



Integration Testing

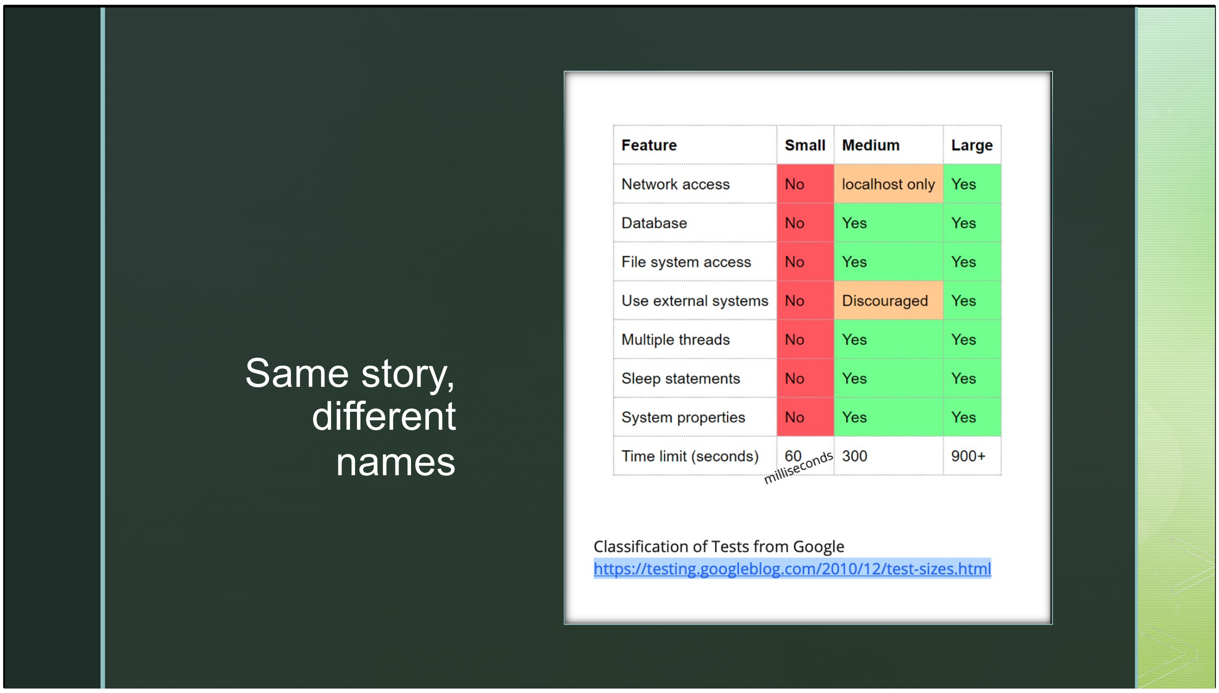
Two or more components are put together in order to see how they behave together. The focus here is in the interaction between components - not the specific internals of each component.



Where do we focus our time?

Acceptance/System Testing

We test the system in a close-to-production environment, with all components working together. Testing should focus on the overall behaviour of the system – testing all the paths is pretty much impossible at this point.

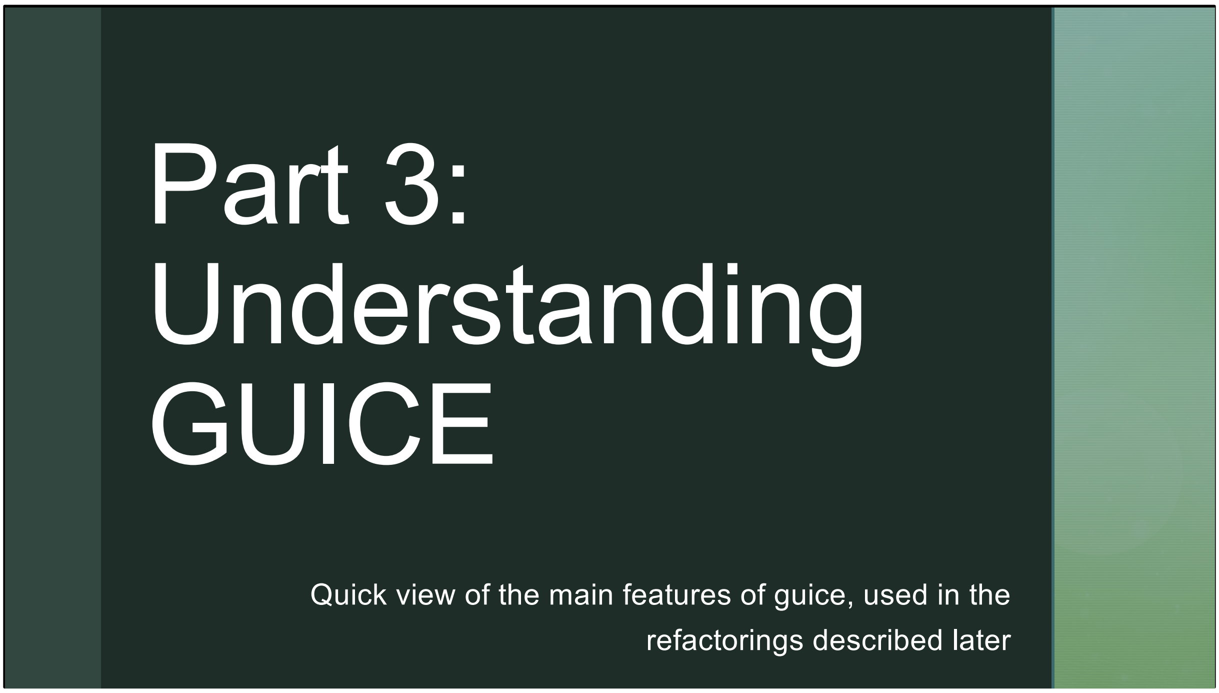


Same story, different names

Feature	Small	Medium	Large
Network access	No	localhost only	Yes
Database	No	Yes	Yes
File system access	No	Yes	Yes
Use external systems	No	Discouraged	Yes
Multiple threads	No	Yes	Yes
Sleep statements	No	Yes	Yes
System properties	No	Yes	Yes
Time limit (seconds)	60 milliseconds	300	900+

Classification of Tests from Google
https://testing.googleblog.com/2010/12/test-sizes.html

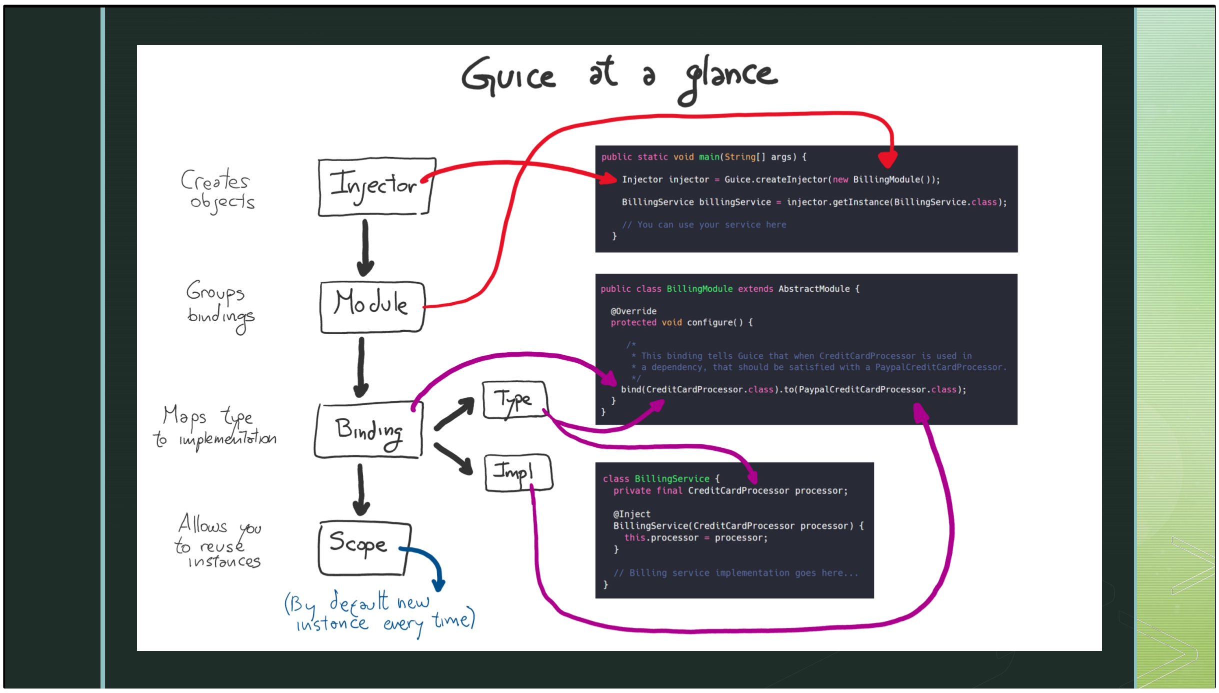
https://testing.googleblog.com/2010/12/test-sizes.html



Part 3: Understanding GUICE

Quick view of the main features of guice, used in the refactorings described later

https://www.youtube.com/watch?v=hBVJbzAagfs



Guice at a glance

Creates objects
Injector
Groups bindings
Module
Maps type to implementation
Binding
Type
Impl
Allows you to reuse instances
Scope
(By default new instance every time)

```java
public static void main(String[] args) {

    Injector injector = Guice.createInjector(new BillingModule());

    BillingService billingService = injector.getInstance(BillingService.class);

    // You can use your service here
  }
```

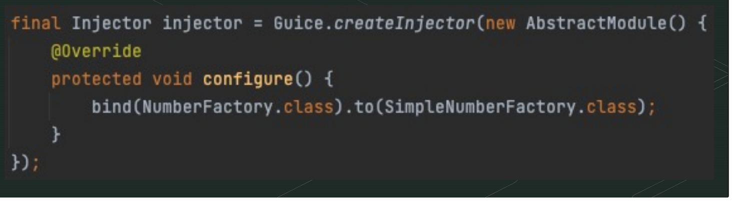
```java
public class BillingModule extends AbstractModule {

  @Override
  protected void configure() {

     /*
      * This binding tells Guice that when CreditCardProcessor is used in
      * a dependency, that should be satisfied with a PaypalCreditCardProcessor.
      */
    bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);
  }
}
```

```java
class BillingService {
  private final CreditCardProcessor processor;

  @Inject
  BillingService(CreditCardProcessor processor) {
    this.processor = processor;
  }

  // Billing service implementation goes here...
}
```

https://jivimberg.io/blog/2019/02/08/guice-at-a-glance/

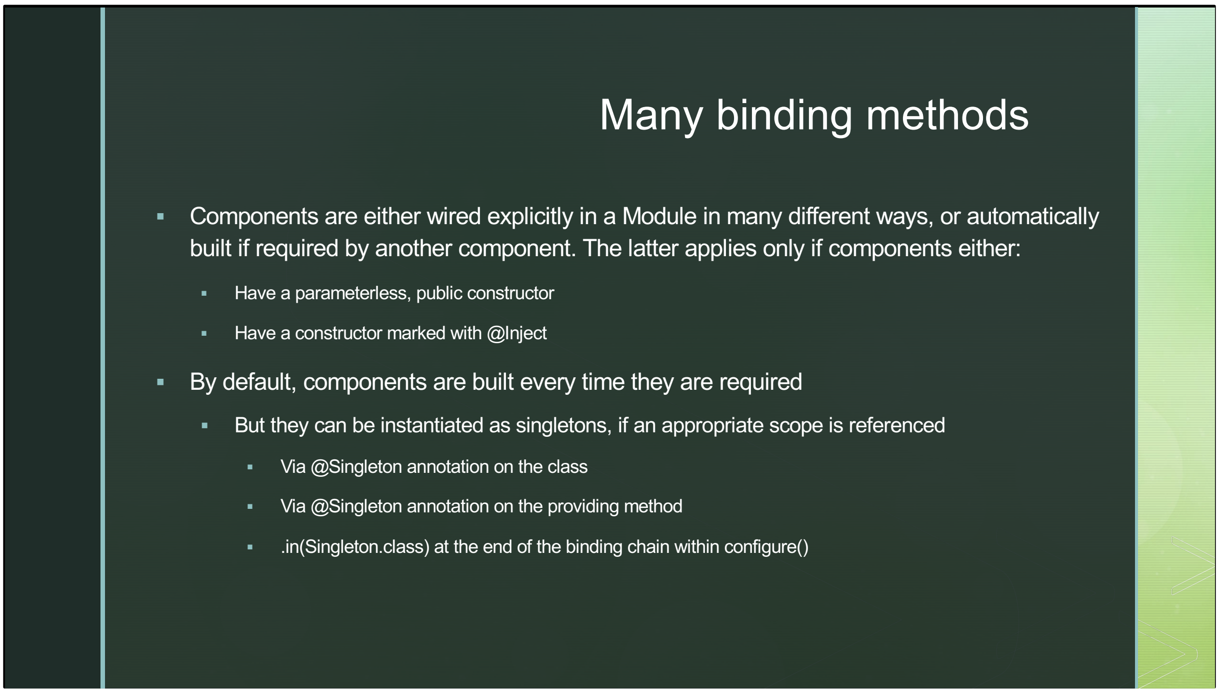
Guice Bindings

- Guice can be seen as a huge key-value map
  - The key is the combination of a type (be that a class or interface) and zero or more annotations
  - The value is a provider that returns an instance
    - It can be a singleton instance
    - Or a new instance for every request (default)
  - Most of the times, the provider is implicit



```java
final Injector injector = Guice.createInjector(new AbstractModule() {
    @Override
    protected void configure() {
        bind(NumberFactory.class).to(SimpleNumberFactory.class);
    }
});
```

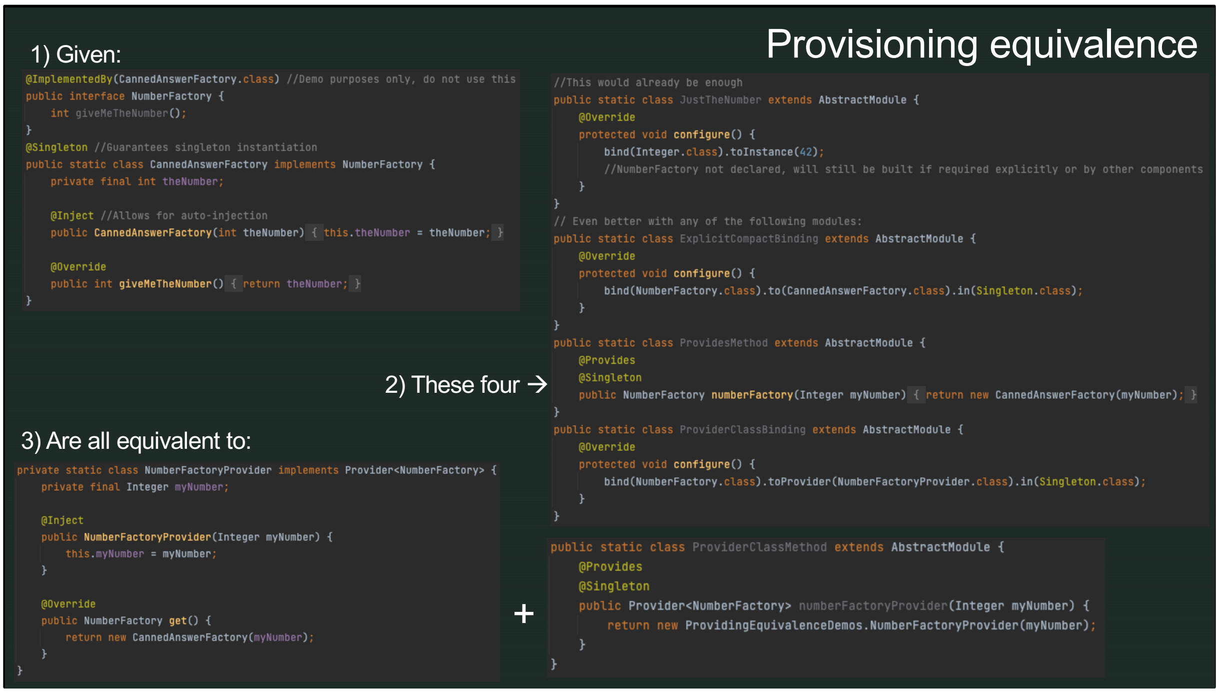
https://github.com/google/guice/wiki/MentalModel



Many binding methods

- Components are either wired explicitly in a Module in many different ways, or automatically built if required by another component. The latter applies only if components either:
  - Have a parameterless, public constructor
  - Have a constructor marked with @Inject
- By default, components are built every time they are required
  - But they can be instantiated as singletons, if an appropriate scope is referenced
    - Via @Singleton annotation on the class
    - Via @Singleton annotation on the providing method
    - .in(Singleton.class) at the end of the binding chain within configure()

https://github.dev/dseurotech/understanding-guice



Provisioning equivalence

1) Given:

```java
@ImplementedBy(CannedAnswerFactory.class) //Demo purposes only, do not use this
public interface NumberFactory {
    int giveMeTheNumber();
}
@Singleton //Guarantees singleton instantiation
public static class CannedAnswerFactory implements NumberFactory {
    private final int theNumber;

    @Inject //Allows for auto-injection
    public CannedAnswerFactory(int theNumber) { this.theNumber = theNumber; }

    @Override
    public int giveMeTheNumber() { return theNumber; }
}
```

2) These four →

```java
//This would already be enough
public static class JustTheNumber extends AbstractModule {
    @Override
    protected void configure() {
        bind(Integer.class).toInstance(42);
        //NumberFactory not declared, will still be built if required explicitly or by other components
    }
}
// Even better with any of the following modules:
public static class ExplicitCompactBinding extends AbstractModule {
    @Override
    protected void configure() {
        bind(NumberFactory.class).to(CannedAnswerFactory.class).in(Singleton.class);
    }
}
public static class ProvidesMethod extends AbstractModule {
    @Provides
    @Singleton
    public NumberFactory numberFactory(Integer myNumber) { return new CannedAnswerFactory(myNumber); }
}
public static class ProviderClassBinding extends AbstractModule {
    @Override
    protected void configure() {
        bind(NumberFactory.class).toProvider(NumberFactoryProvider.class).in(Singleton.class);
    }
}
```

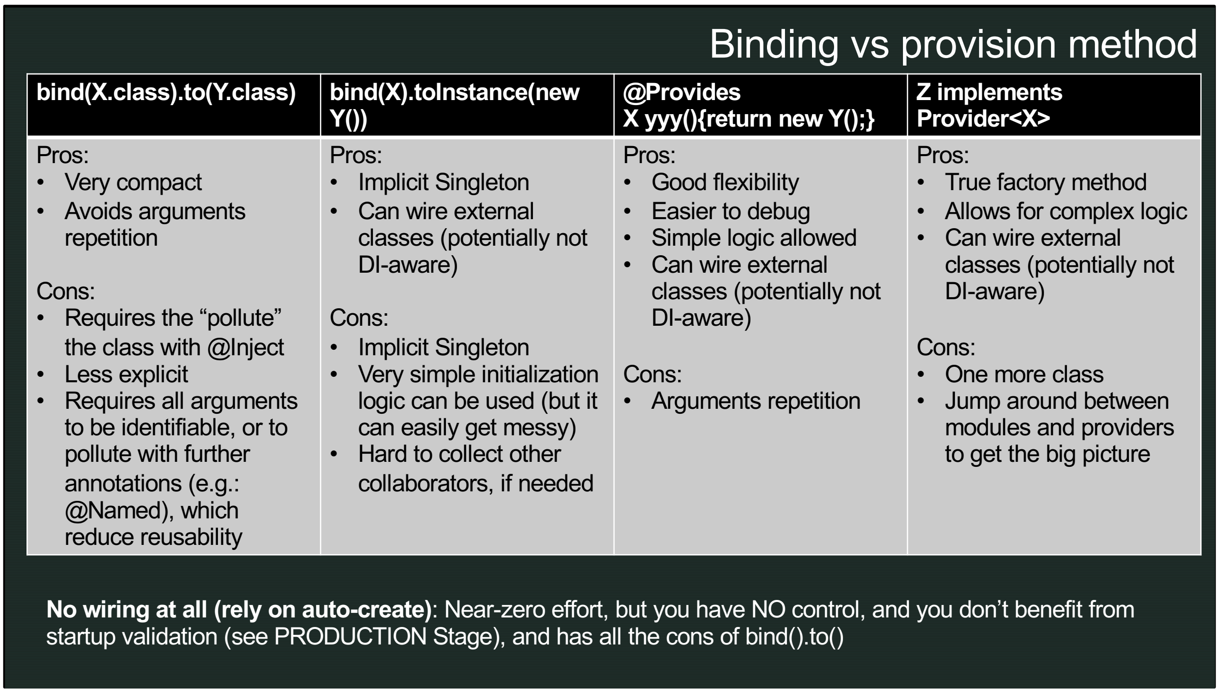
3) Are all equivalent to:

```java
private static class NumberFactoryProvider implements Provider<NumberFactory> {
    private final Integer myNumber;

    @Inject
    public NumberFactoryProvider(Integer myNumber) {
        this.myNumber = myNumber;
    }

    @Override
    public NumberFactory get() {
        return new CannedAnswerFactory(myNumber);
    }
}
```

+

```java
public static class ProviderClassMethod extends AbstractModule {
    @Provides
    @Singleton
    public Provider<NumberFactory> numberFactoryProvider(Integer myNumber) {
        return new ProvidingEquivalenceDemos.NumberFactoryProvider(myNumber);
    }
}
```

While implicit declaration (just rely on @Inject and do not declare the wiring) might seem desirable, it reduces visibility, and makes debugging the injection phase impossible.

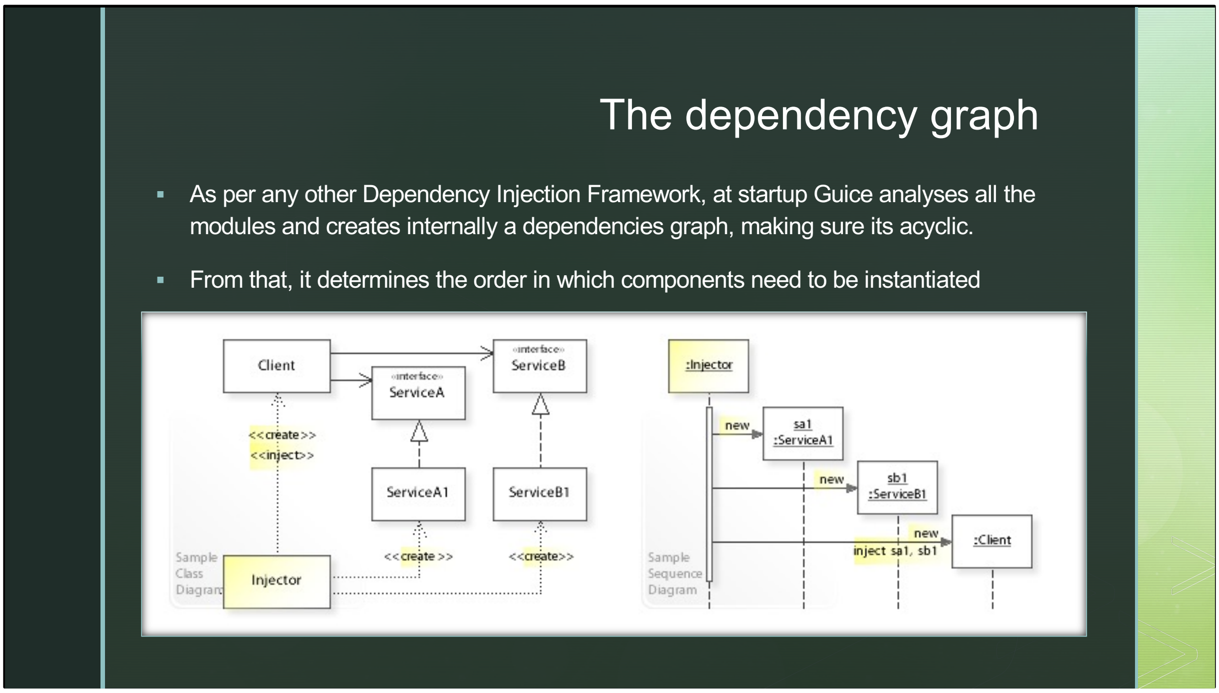


Binding vs provision method

bind(X.class).to(Y.class)	bind(X).toInstance(new Y())	@Provides X yyy(){return new Y();}	Z implements Provider<X>
Pros: • Very compact • Avoids arguments repetition Cons: • Requires the “pollute” the class with @Inject • Less explicit • Requires all arguments to be identifiable, or to pollute with further annotations (e.g.: @Named), which reduce reusability	Pros: • Implicit Singleton • Can wire external classes (potentially not DI-aware) Cons: • Implicit Singleton • Very simple initialization logic can be used (but it can easily get messy) • Hard to collect other collaborators, if needed	Pros: • Good flexibility • Easier to debug • Simple logic allowed • Can wire external classes (potentially not DI-aware) Cons: • Arguments repetition	Pros: • True factory method • Allows for complex logic • Can wire external classes (potentially not DI-aware) Cons: • One more class • Jump around between modules and providers to get the big picture

No wiring at all (rely on auto-create): Near-zero effort, but you have NO control, and you don’t benefit from startup validation (see PRODUCTION Stage), and has all the cons of bind().to()

While implicit declaration (just rely on @Inject and do not declare the wiring) might seem desirable, it reduces visibility, and makes debugging the injection phase impossible.



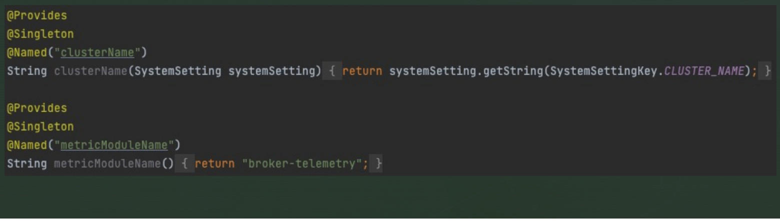
The dependency graph

- As per any other Dependency Injection Framework, at startup Guice analyses all the modules and creates internally a dependencies graph, making sure its acyclic.
- From that, it determines the order in which components need to be instantiated

https://commons.wikimedia.org/wiki/File:W3sDesign_Dependency_Injection_Design_Pattern_UML.jpg

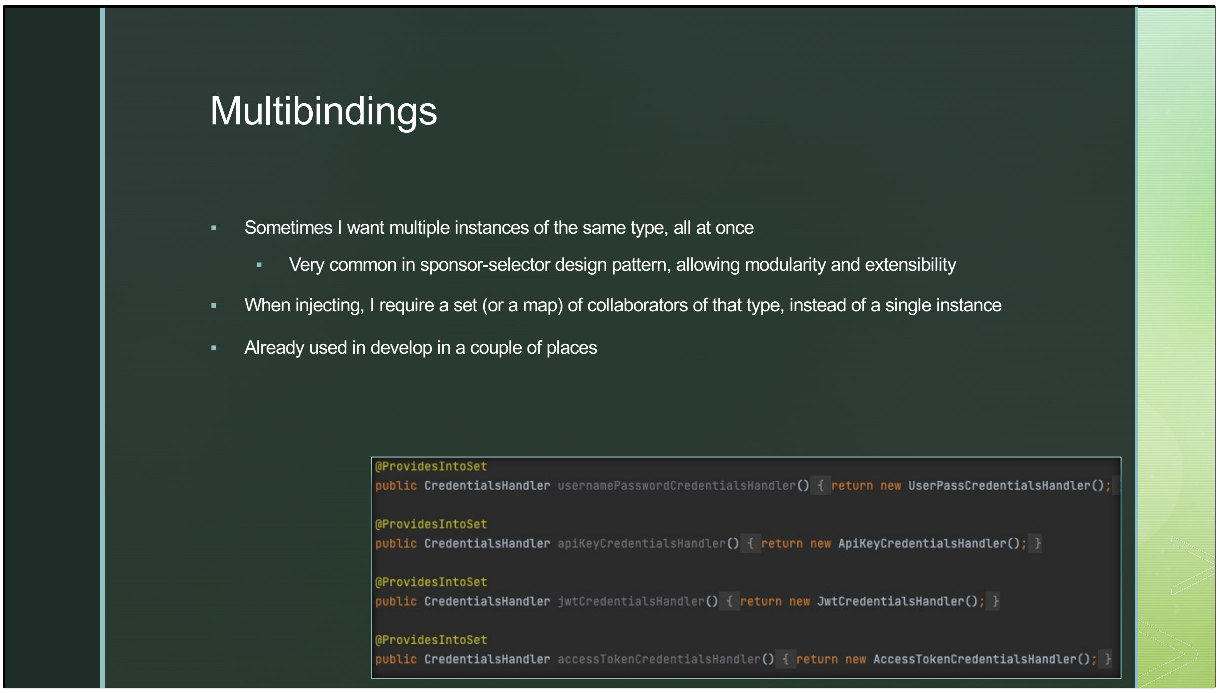
Binding Annotations

- The vast majority of bindings associate an type (key) to a the provider of a concrete class.
- When another component requires an instance of that type as a collaborator, the provider is called and the instance passed to the component
- Normally you can’t bind a Key to multiple implementations
- If I need multiple instances of the same type (e.g.: multiple Strings, each representing a configuration parameter), they can be distinguished using specific annotation, @Named being the most common one



```
@Provides
@Singleton
@Named("clusterName")
String clusterName(SystemSetting systemSetting) { return systemSetting.getString(SystemSettingKey.CLUSTER_NAME); }

@Provides
@Singleton
@Named("metricModuleName")
String metricModuleName() { return "broker-telemetry"; }
```

https://github.com/google/guice/wiki/BindingAnnotations

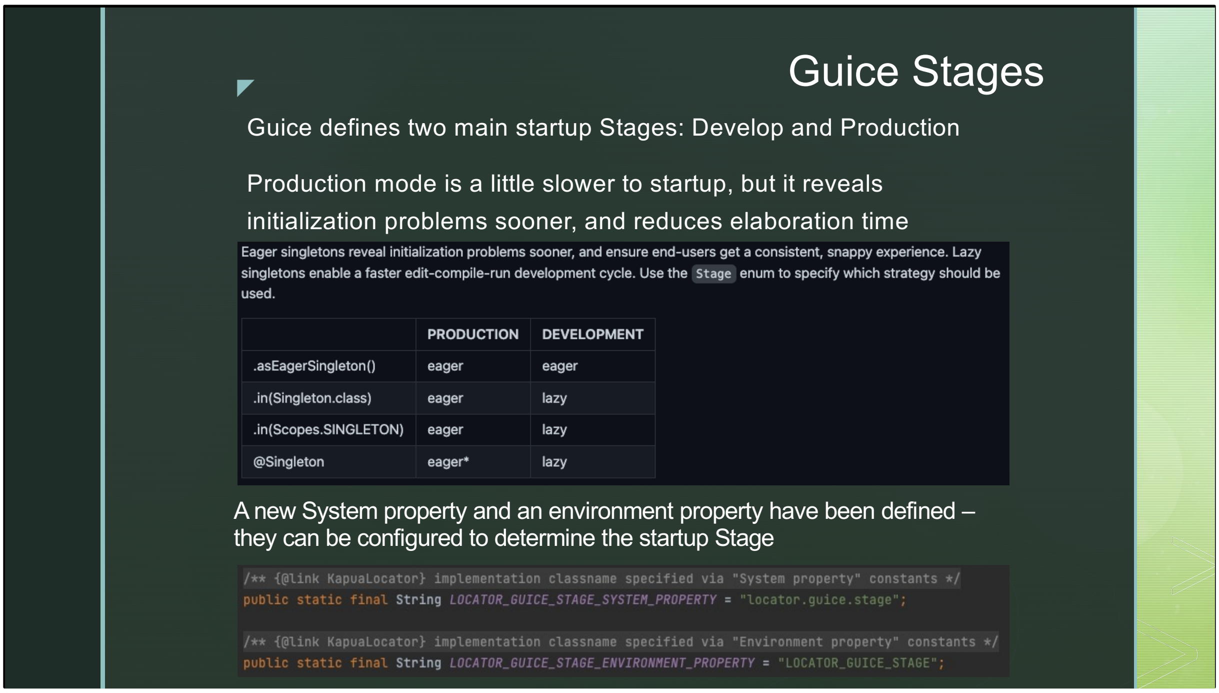


Multibindings

- Sometimes I want multiple instances of the same type, all at once
  - Very common in sponsor-selector design pattern, allowing modularity and extensibility
- When injecting, I require a set (or a map) of collaborators of that type, instead of a single instance
- Already used in develop in a couple of places

```
@ProvidesIntoSet
public CredentialsHandler usernamePasswordCredentialsHandler() { return new UserPassCredentialsHandler();

@ProvidesIntoSet
public CredentialsHandler apiKeyCredentialsHandler() { return new ApiKeyCredentialsHandler(); }

@ProvidesIntoSet
public CredentialsHandler jwtCredentialsHandler() { return new JwtCredentialsHandler(); }

@ProvidesIntoSet
public CredentialsHandler accessTokenCredentialsHandler() { return new AccessTokenCredentialsHandler(); }
```

https://github.com/google/guice/wiki/Multibindings



Guice Stages

Guice defines two main startup Stages: Develop and Production

Production mode is a little slower to startup, but it reveals initialization problems sooner, and reduces elaboration time

Eager singletons reveal initialization problems sooner, and ensure end-users get a consistent, snappy experience. Lazy singletons enable a faster edit-compile-run development cycle. Use the `Stage` enum to specify which strategy should be used.

	PRODUCTION	DEVELOPMENT
.asEagerSingleton()	eager	eager
.in(Singleton.class)	eager	lazy
.in(Scopes.SINGLETON)	eager	lazy
@Singleton	eager*	lazy

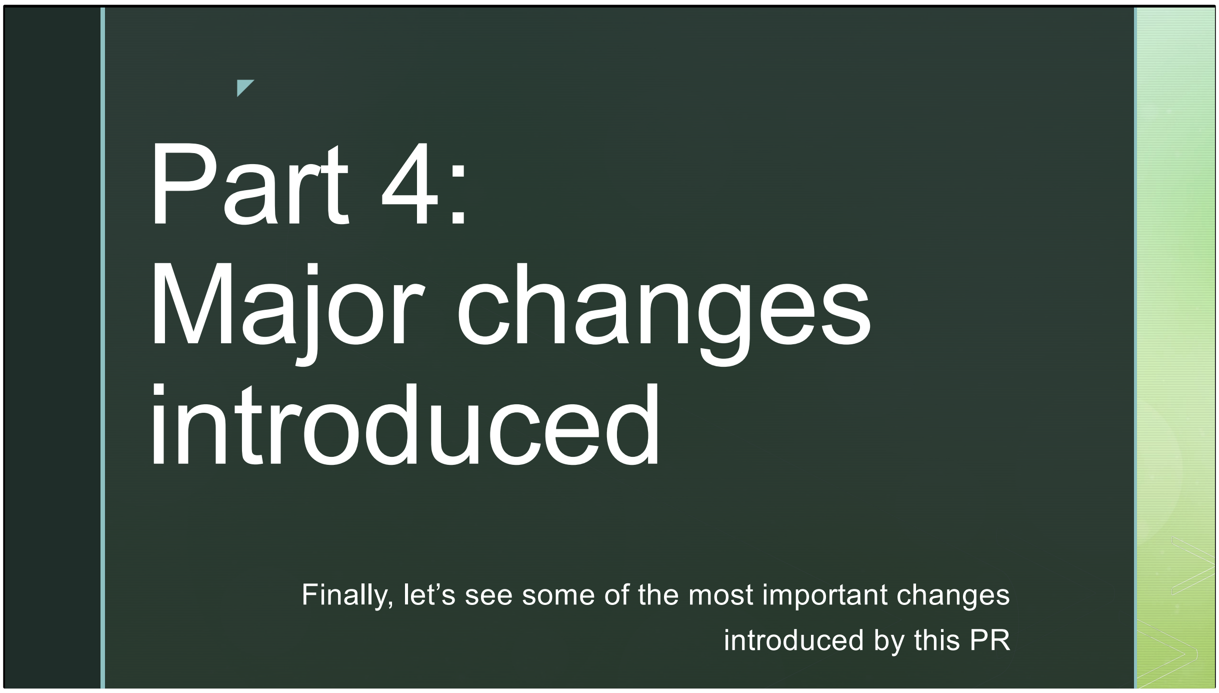
A new System property and an environment property have been defined – they can be configured to determine the startup Stage

```
/** {@link KapuaLocator} implementation classname specified via "System property" constants */
public static final String LOCATOR_GUICE_STAGE_SYSTEM_PROPERTY = "locator.guice.stage";

/** {@link KapuaLocator} implementation classname specified via "Environment property" constants */
public static final String LOCATOR_GUICE_STAGE_ENVIRONMENT_PROPERTY = "LOCATOR_GUICE_STAGE";
```

https://github.com/google/guice/wiki/Scopes#eager-singletons
https://www.technowizardry.net/2022/05/best-practices-for-working-with-google-guice/

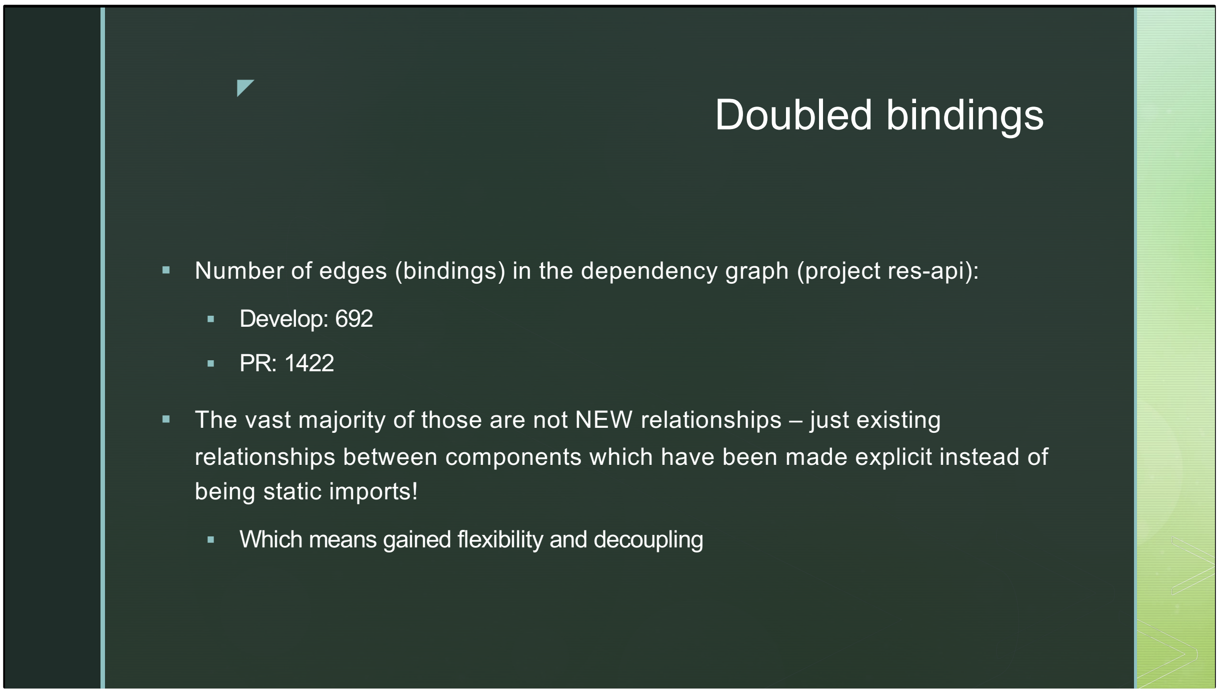
All of Kapua’s components now start with Stage.PRODUCTION

Most of EC components do, too.



Part 4: Major changes introduced

Finally, let’s see some of the most important changes introduced by this PR



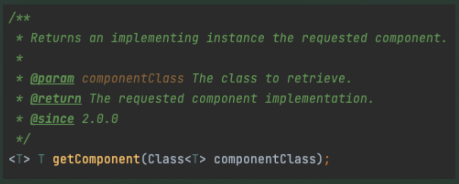
Doubled bindings

- Number of edges (bindings) in the dependency graph (project res-api):
  - Develop: 692
  - PR: 1422
- The vast majority of those are not NEW relationships – just existing relationships between components which have been made explicit instead of being static imports!
  - Which means gained flexibility and decoupling

https://lindbakk.com/blog/utility-and-helper-classes-a-code-smell

Originally marked as: //TODO: FIXME: promote from static utility to injectable collaborator

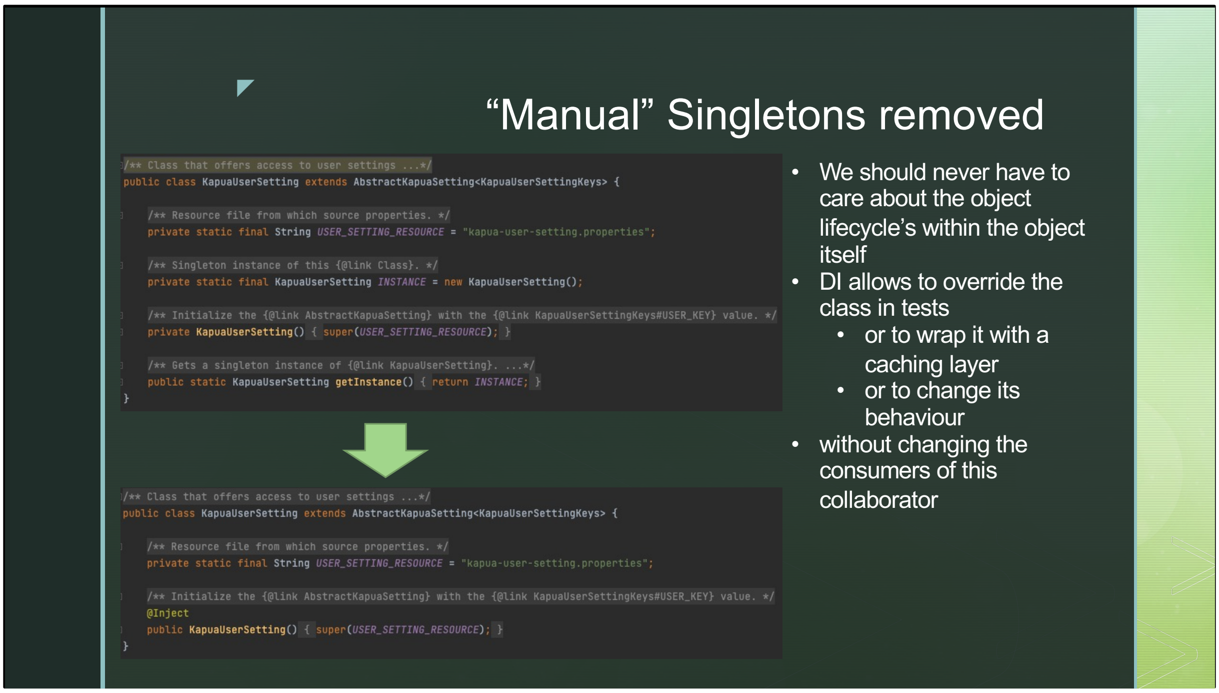
GetComponent trumps getService and getFactory



```
/**
 * Returns an implementing instance the requested component.
 *
 * @param componentClass The class to retrieve.
 * @return The requested component implementation.
 * @since 2.0.0
 */
<T> T getComponent(Class<T> componentClass);
```

Introduced a new method called “getComponent”, which is capable of returning ANY wired component

- Because not all collaborators are Services or Factories
- Because not all collaborators are Kapua’s classes
- Works with both implementations and interfaces
- Can de-facto replace getService and getFactory



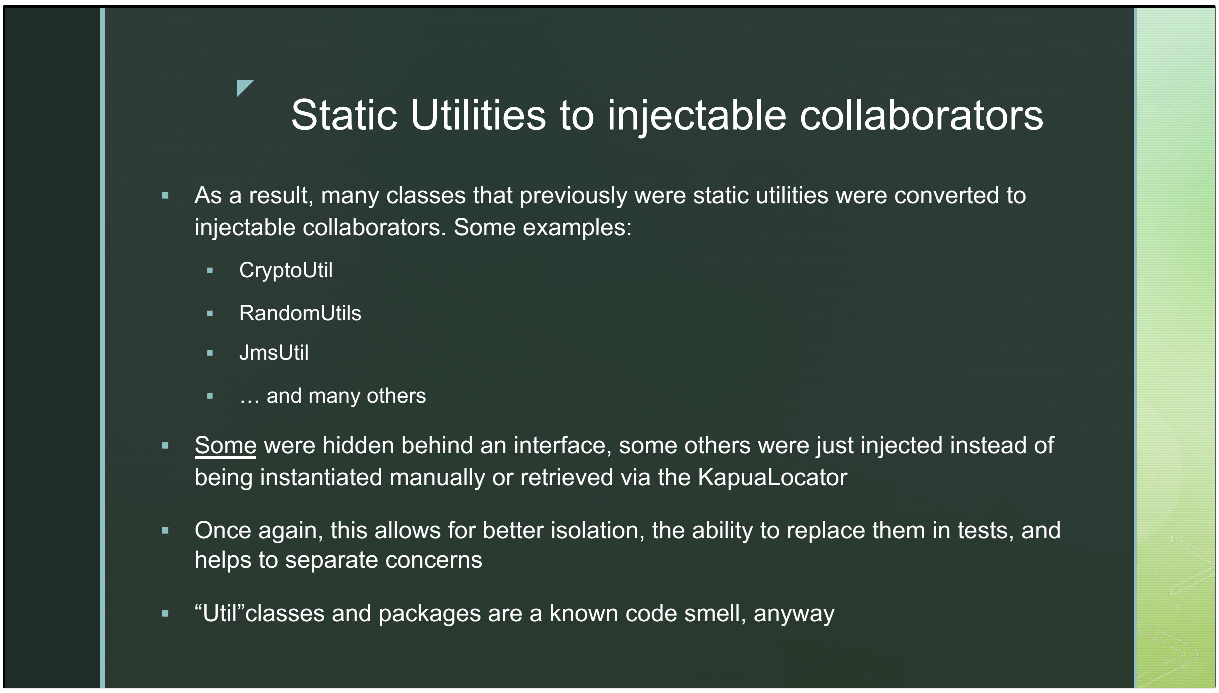
“Manual” Singletons removed

```java
/** Class that offers access to user settings ...*/
public class KapuaUserSetting extends AbstractKapuaSetting<KapuaUserSettingKeys> {

    /** Resource file from which source properties. */
    private static final String USER_SETTING_RESOURCE = "kapua-user-setting.properties";

    /** Singleton instance of this {@link Class}. */
    private static final KapuaUserSetting INSTANCE = new KapuaUserSetting();

    /** Initialize the {@link AbstractKapuaSetting} with the {@link KapuaUserSettingKeys#USER_KEY} value. */
    private KapuaUserSetting() { super(USER_SETTING_RESOURCE); }

    /** Gets a singleton instance of {@link KapuaUserSetting}. ...*/
    public static KapuaUserSetting getInstance() { return INSTANCE; }
}
```

```java
/** Class that offers access to user settings ...*/
public class KapuaUserSetting extends AbstractKapuaSetting<KapuaUserSettingKeys> {

    /** Resource file from which source properties. */
    private static final String USER_SETTING_RESOURCE = "kapua-user-setting.properties";

    /** Initialize the {@link AbstractKapuaSetting} with the {@link KapuaUserSettingKeys#USER_KEY} value. */
    @Inject
    public KapuaUserSetting() { super(USER_SETTING_RESOURCE); }
}
```

- We should never have to care about the object lifecycle’s within the object itself
- DI allows to override the class in tests
  - or to wrap it with a caching layer
  - or to change its behaviour
- without changing the consumers of this collaborator

Originally marked as: //TODO: FIXME: singletons should not be handled manually, we have DI for that



Static Utilities to injectable collaborators

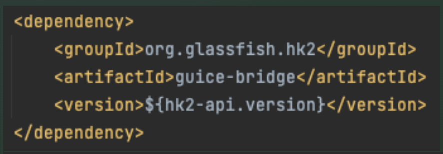
- As a result, many classes that previously were static utilities were converted to injectable collaborators. Some examples:
  - CryptoUtil
  - RandomUtils
  - JmsUtil
  - … and many others
- Some were hidden behind an interface, some others were just injected instead of being instantiated manually or retrieved via the KapuaLocator
- Once again, this allows for better isolation, the ability to replace them in tests, and helps to separate concerns
- “Util”classes and packages are a known code smell, anyway

https://lindbakk.com/blog/utility-and-helper-classes-a-code-smell

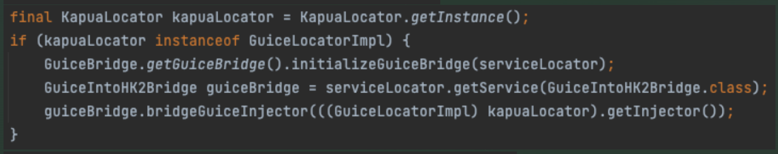
Originally marked as: //TODO: FIXME: promote from static utility to injectable collaborator

Guice-to-hk2 bridge

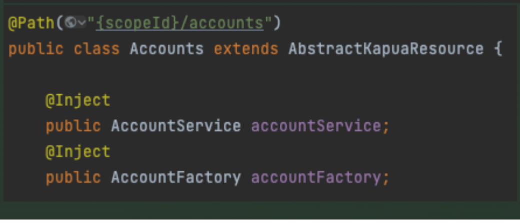
As the name suggests, bridges between Guice wiring and hk2's one, allowing components defined in the former to be used in the latter



```
<dependency>
    <groupId>org.glassfish.hk2</groupId>
    <artifactId>guice-bridge</artifactId>
    <version>${hk2-api.version}</version>
</dependency>
```



```
final KapuaLocator kapuaLocator = KapuaLocator.getInstance();
if (kapuaLocator instanceof GuiceLocatorImpl) {
    GuiceBridge.getGuiceBridge().initializeGuiceBridge(serviceLocator);
    GuiceIntoHK2Bridge guiceBridge = serviceLocator.getService(GuiceIntoHK2Bridge.class);
    guiceBridge.bridgeGuiceInjector(((GuiceLocatorImpl) kapuaLocator).getInjector());
}
```

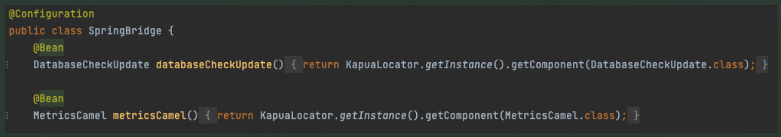


```
@Path("{scopeId}/accounts")
public class Accounts extends AbstractKapuaResource {

    @Inject
    public AccountService accountService;
    @Inject
    public AccountFactory accountFactory;
```

This allows wiring of kapua's components even where normally we would not have control, as in jersey resources (for example)

Guice-to-spring bridge (manual)

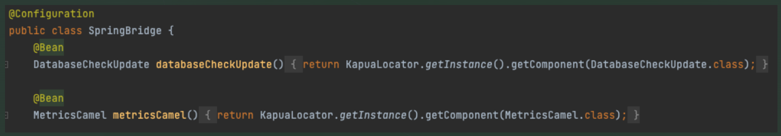
- In the past, spring components (@Beans) were configured in isolation, and used to consume kapua’s collaborators either via the Locator, or via singleton’s access.
- Now the two wiring’s graphs have been connected directly, allowing Guice-defined bindings to be used as spring’s @Beans
- Unfortunately, at the moment this require manual bridging



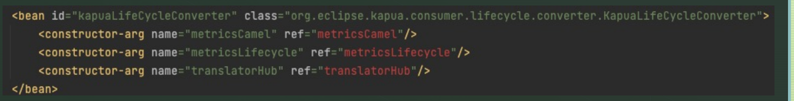
```
@Configuration
public class SpringBridge {
    @Bean
    DatabaseCheckUpdate databaseCheckUpdate() { return KapuaLocator.getInstance().getComponent(DatabaseCheckUpdate.class); }

    @Bean
    MetricsCamel metricsCamel() { return KapuaLocator.getInstance().getComponent(MetricsCamel.class); }
```

Guice-to-spring bridge (manual) II

Notice that the name of the method annotated with @Bean is also the id of the bean, to be referenced in the xml configuration:

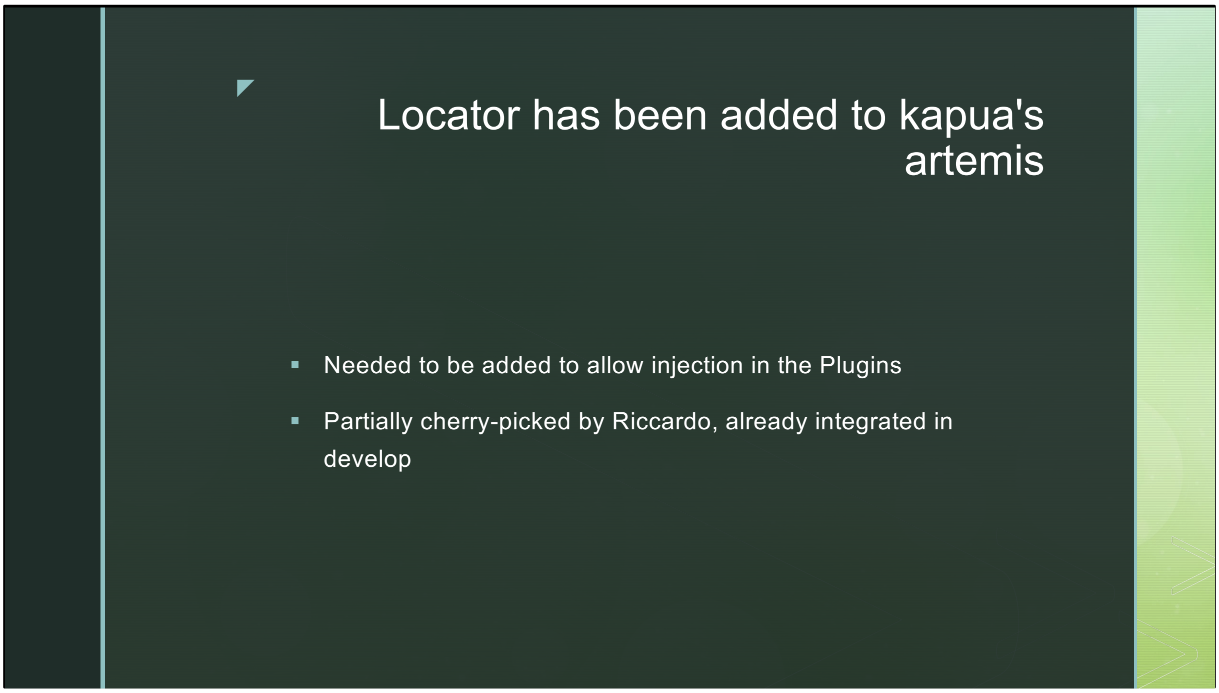


```java
@Configuration
public class SpringBridge {
    @Bean
    DatabaseCheckUpdate databaseCheckUpdate() { return KapuaLocator.getInstance().getComponent(DatabaseCheckUpdate.class); }

    @Bean
    MetricsCamel metricsCamel() { return KapuaLocator.getInstance().getComponent(MetricsCamel.class); }
```



```xml
<bean id="kapuaLifeCycleConverter" class="org.eclipse.kapua.consumer.lifecycle.converter.KapuaLifeCycleConverter">
    <constructor-arg name="metricsCamel" ref="metricsCamel"/>
    <constructor-arg name="metricsLifecycle" ref="metricsLifecycle"/>
    <constructor-arg name="translatorHub" ref="translatorHub"/>
</bean>
```

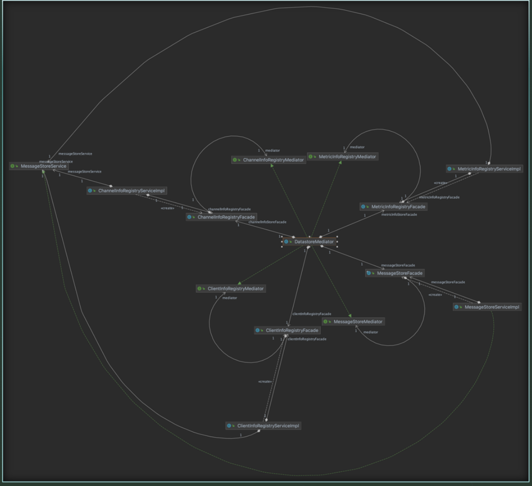
Side note: Xml configuration should really be removed in favour of java config for spring components, but that’s for another refactoring



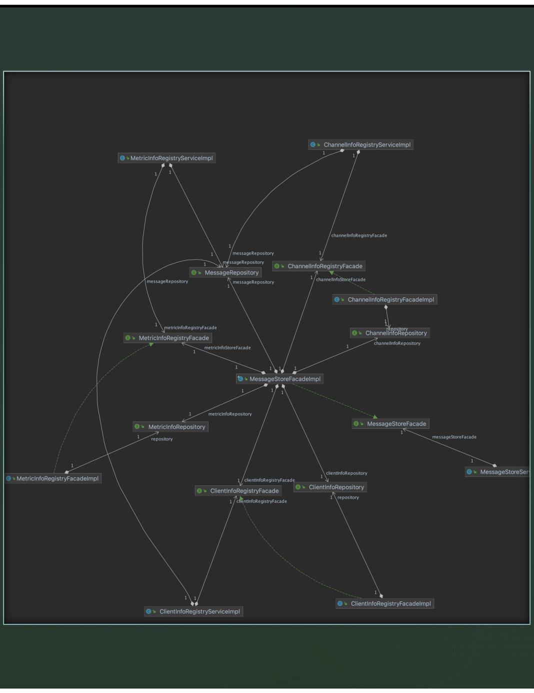
Locator has been added to kapua's artemis

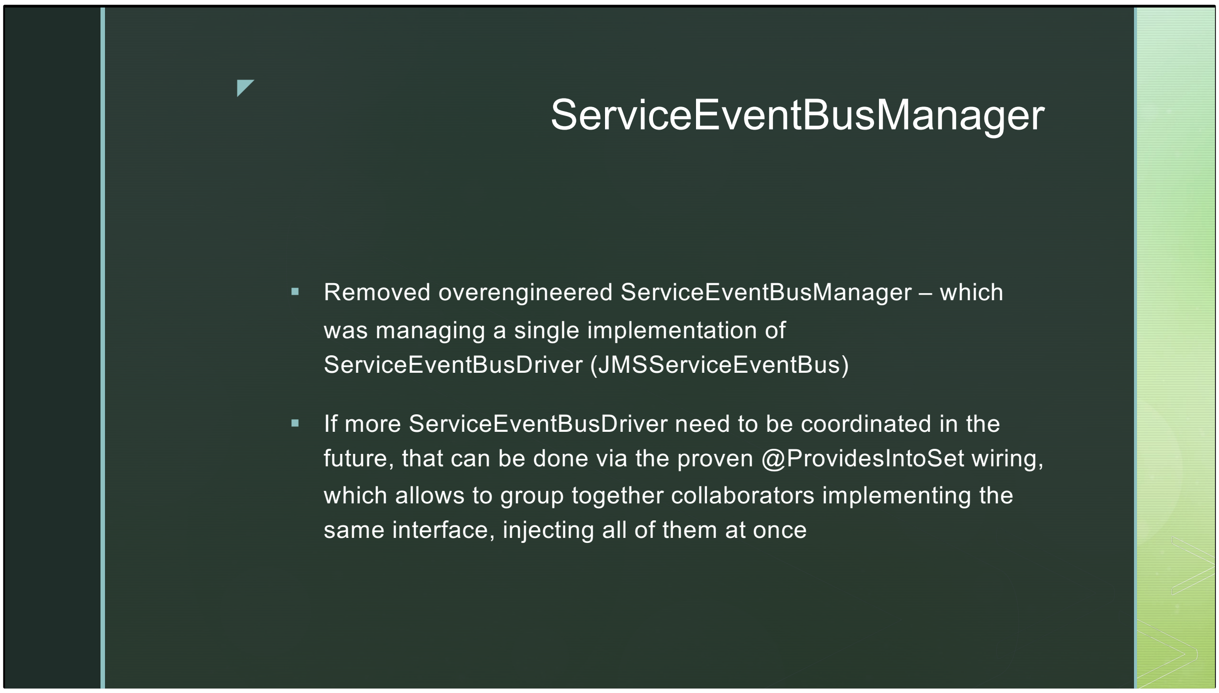
- Needed to be added to allow injection in the Plugins
- Partially cherry-picked by Riccardo, already integrated in develop

Unwrapping elasticsearch services (before)



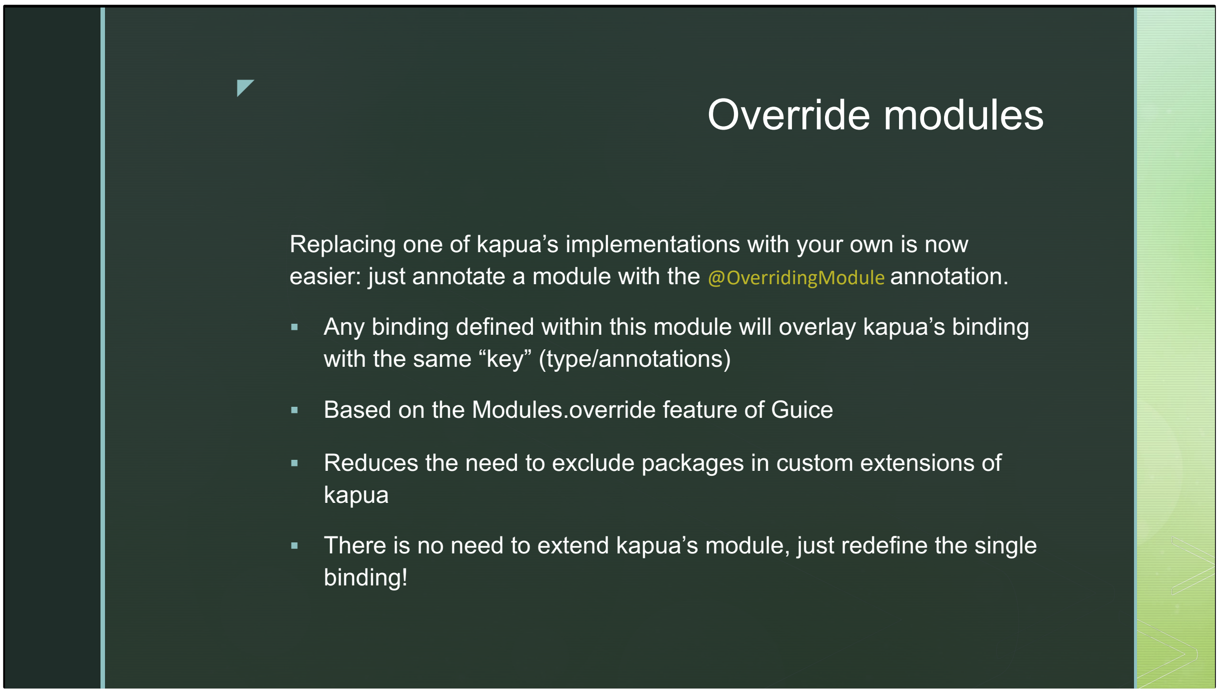
Unwrapping elasticsearch services (after)





ServiceEventBusManager

- Removed overengineered ServiceEventBusManager – which was managing a single implementation of ServiceEventBusDriver (JMSServiceEventBus)
- If more ServiceEventBusDriver need to be coordinated in the future, that can be done via the proven @ProvidesIntoSet wiring, which allows to group together collaborators implementing the same interface, injecting all of them at once

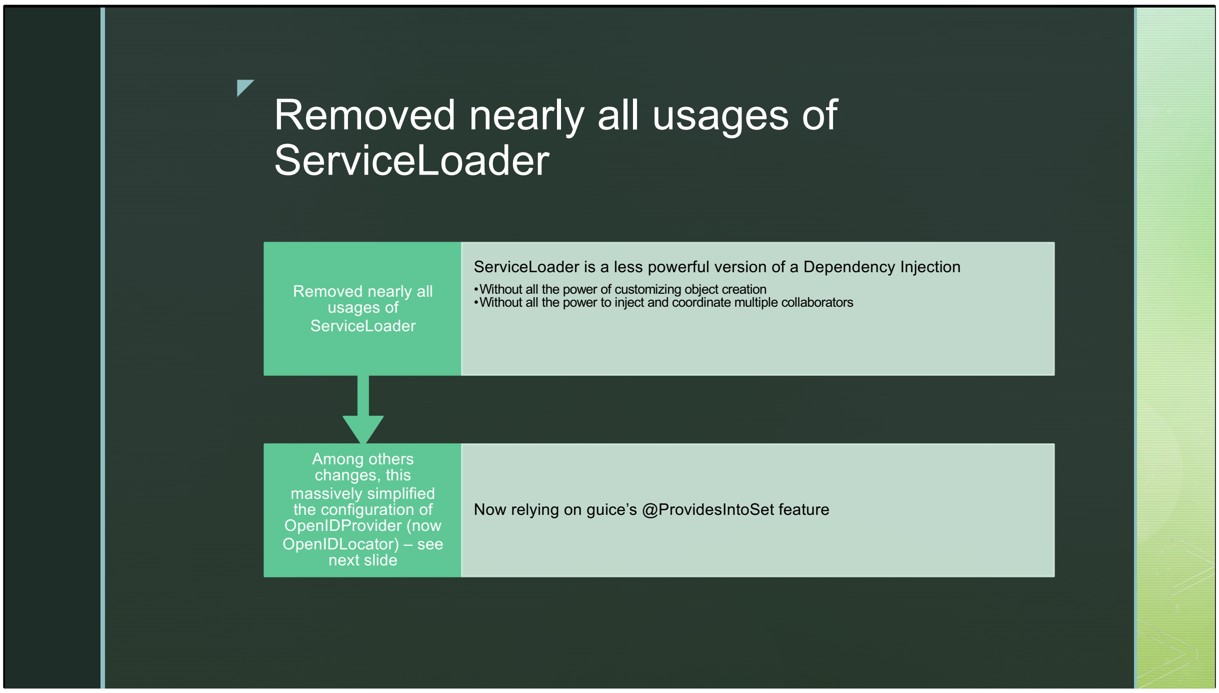


Override modules

Replacing one of kapua's implementations with your own is now easier: just annotate a module with the @OverridingModule annotation.

- Any binding defined within this module will overlay kapua's binding with the same "key" (type/annotations)
- Based on the Modules.override feature of Guice
- Reduces the need to exclude packages in custom extensions of kapua
- There is no need to extend kapua's module, just redefine the single binding!

https://google.github.io/guice/api-docs/5.1.0/javadoc/com/google/inject/util/Modules.html#override(com.google.inject.Module...)



Removed nearly all usages of ServiceLoader

Removed nearly all usages of ServiceLoader

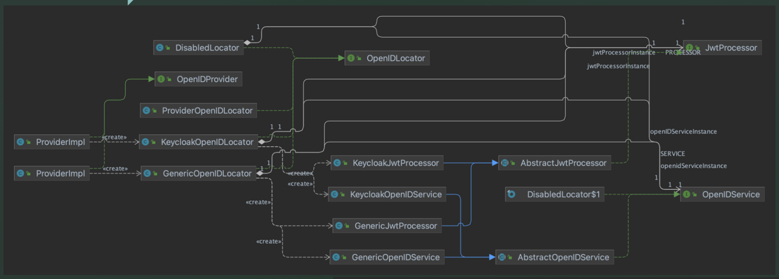
ServiceLoader is a less powerful version of a Dependency Injection

- Without all the power of customizing object creation
- Without all the power to inject and coordinate multiple collaborators

Among others changes, this massively simplified the configuration of OpenIDProvider (now OpenIDLocator) – see next slide

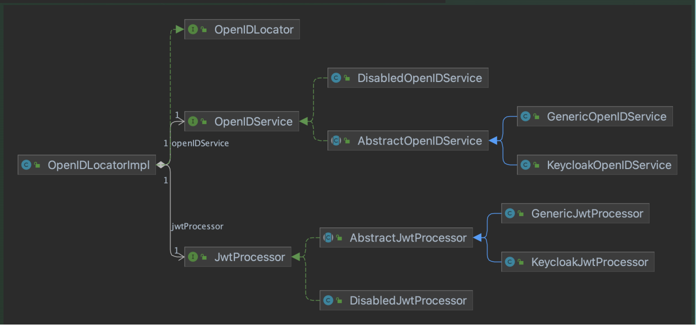
Now relying on guice's @ProvidesIntoSet feature

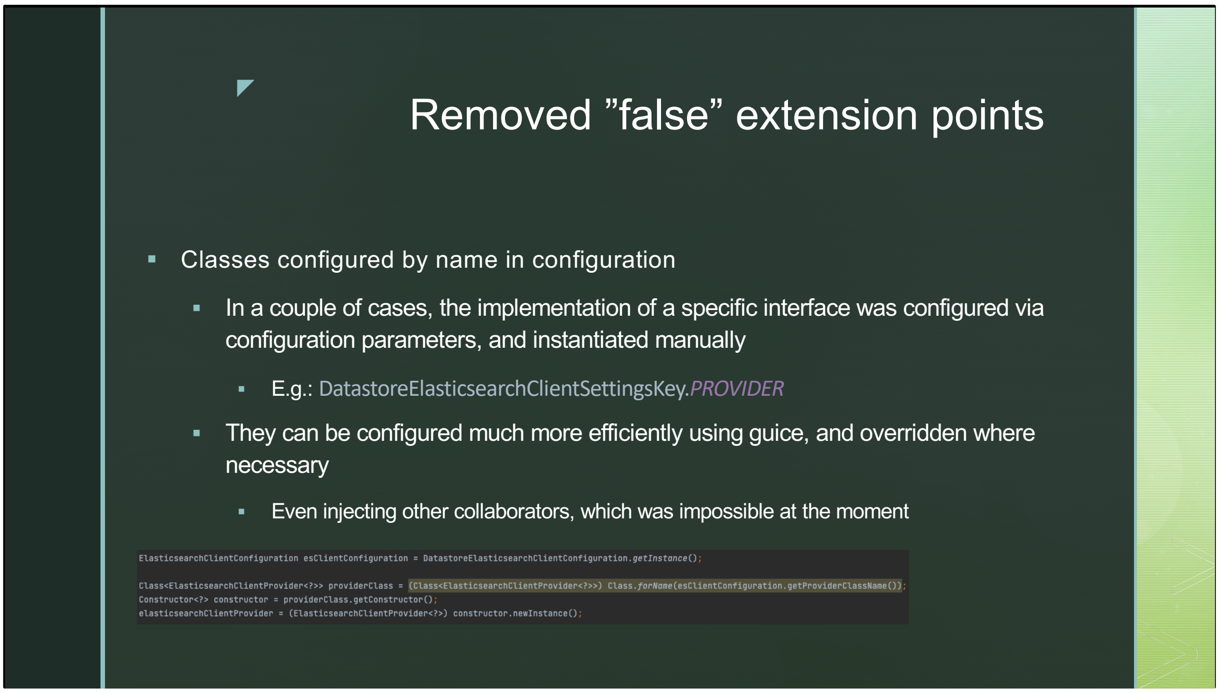
OpenIdProvider Refactoring



Before

After

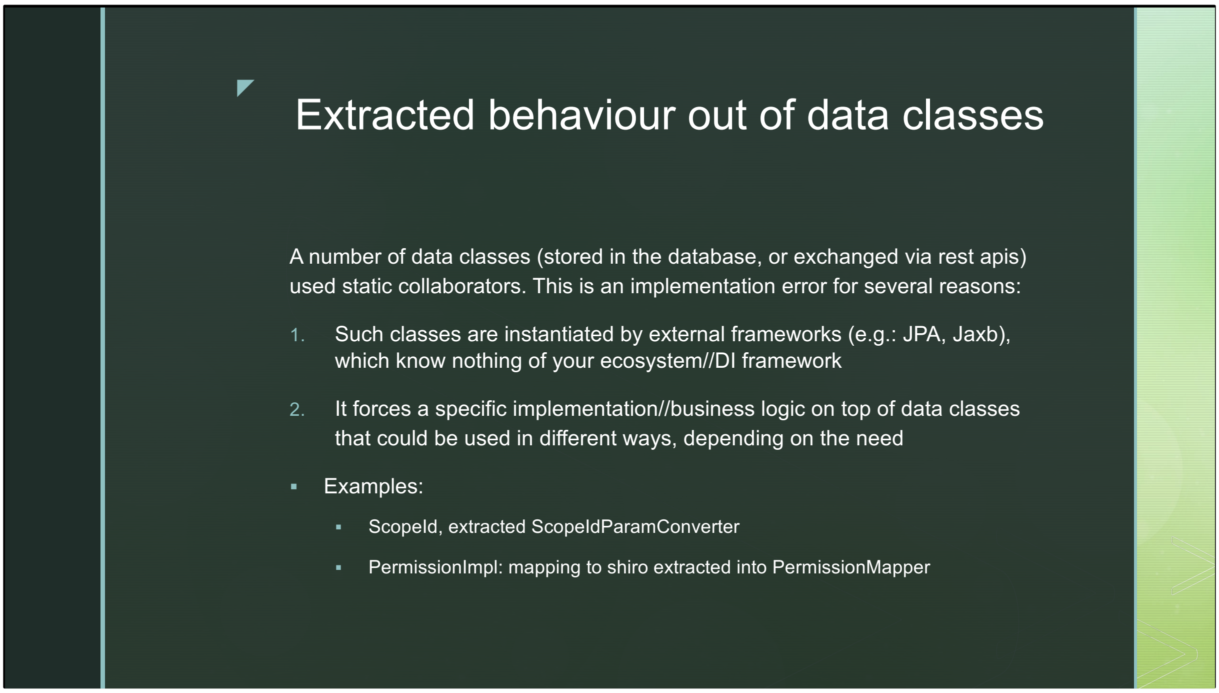




Removed ”false” extension points

- Classes configured by name in configuration
  - In a couple of cases, the implementation of a specific interface was configured via configuration parameters, and instantiated manually
    - E.g.: DatastoreElasticsearchClientSettingsKey.*PROVIDER*
  - They can be configured much more efficiently using guice, and overridden where necessary
    - Even injecting other collaborators, which was impossible at the moment

```
ElasticsearchClientConfiguration esClientConfiguration = DatastoreElasticsearchClientConfiguration.getInstance();

Class<ElasticsearchClientProvider<?>> providerClass = (Class<ElasticsearchClientProvider<?>>) Class.forName(esClientConfiguration.getProviderClassName());
Constructor<?> constructor = providerClass.getConstructor();
elasticsearchClientProvider = (ElasticsearchClientProvider<?>) constructor.newInstance();
```



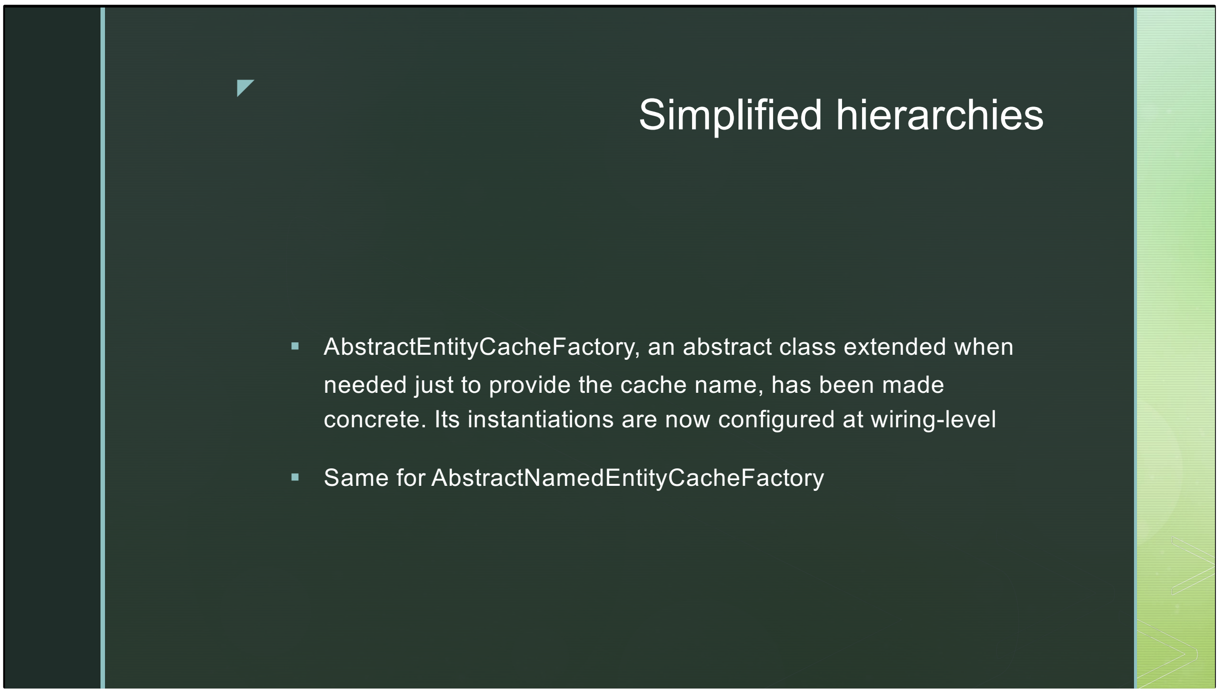
Extracted behaviour out of data classes

A number of data classes (stored in the database, or exchanged via rest apis) used static collaborators. This is an implementation error for several reasons:

1. Such classes are instantiated by external frameworks (e.g.: JPA, Jaxb), which know nothing of your ecosystem//DI framework
2. It forces a specific implementation//business logic on top of data classes that could be used in different ways, depending on the need

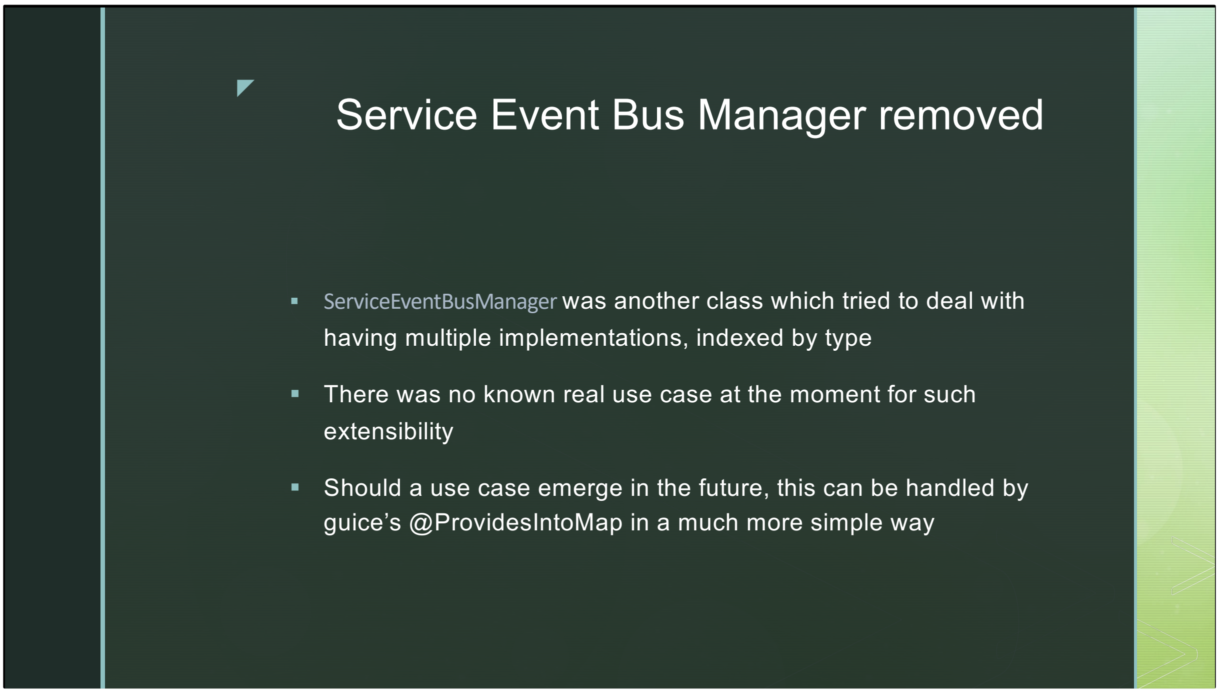
- Examples:
  - ScopeId, extracted ScopeIdParamConverter
  - PermissionImpl: mapping to shiro extracted into PermissionMapper

//TODO: FIXME: REMOVE: A collaborator in a data class? Behaviour should not be part of a data class!



Simplified hierarchies

- AbstractEntityCacheFactory, an abstract class extended when needed just to provide the cache name, has been made concrete. Its instantiations are now configured at wiring-level
- Same for AbstractNamedEntityCacheFactory

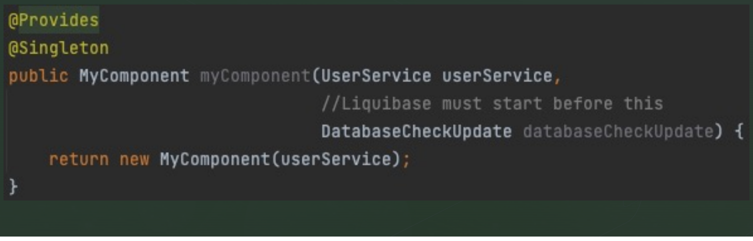


Service Event Bus Manager removed

- ServiceEventBusManager was another class which tried to deal with having multiple implementations, indexed by type
- There was no known real use case at the moment for such extensibility
- Should a use case emerge in the future, this can be handled by guice’s @ProvidesIntoMap in a much more simple way

Standardized db initialization

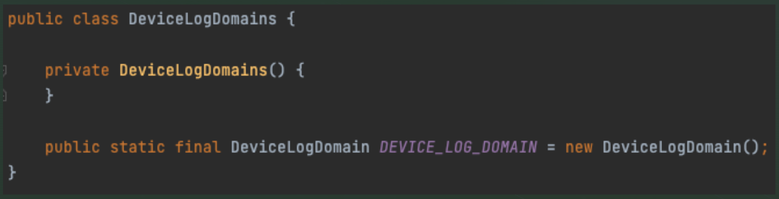
- There were many different ways to inizialize the database driver and to execute Liquibase
- They have been aligned, using appropriate wiring techniques
- If your class initialization depends on the database being initialized (sic! e.g.: because it needs Liquibase to seed some values, like sys users), just declare the bean as a dependency (even if not used):



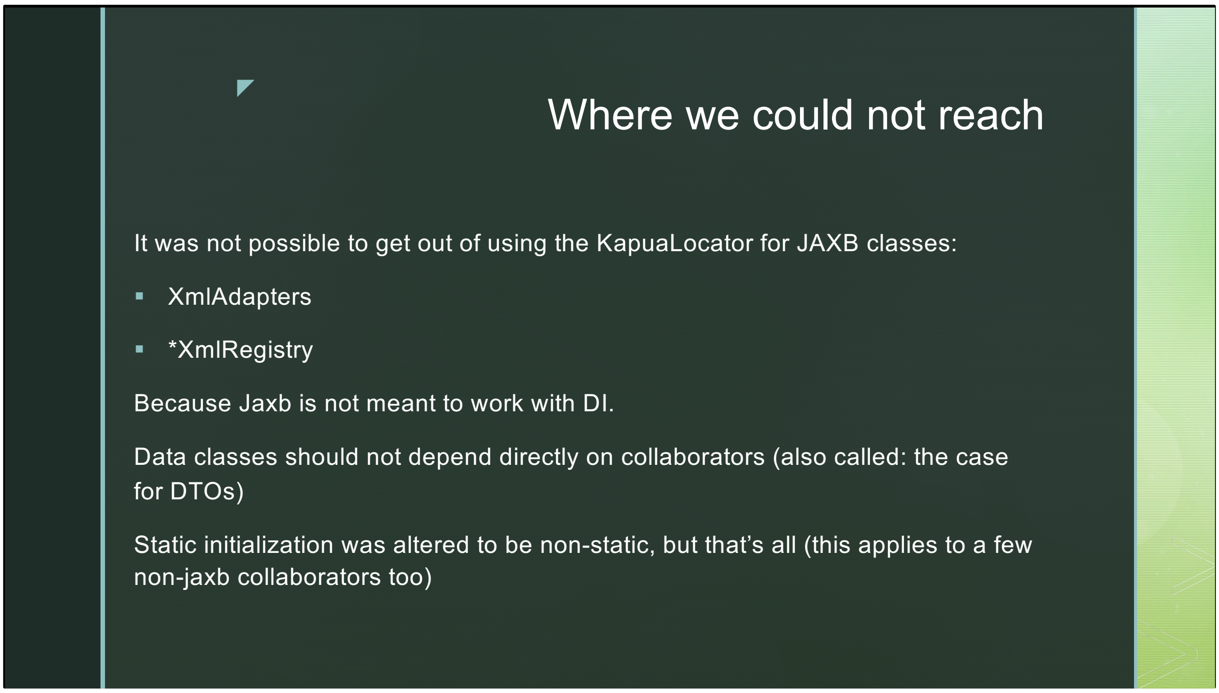
```
@Provides
@Singleton
public MyComponent myComponent(UserService userService,
                               //Liquibase must start before this
                               DatabaseCheckUpdate databaseCheckUpdate) {
    return new MyComponent(userService);
}
```

*Domains classes removed

- Just static references to new *Domain()



```
public class DeviceLogDomains {

    private DeviceLogDomains() {
    }

    public static final DeviceLogDomain DEVICE_LOG_DOMAIN = new DeviceLogDomain();
}
```



Where we could not reach

It was not possible to get out of using the KapuaLocator for JAXB classes:

- XmlAdapters
- *XmlRegistry

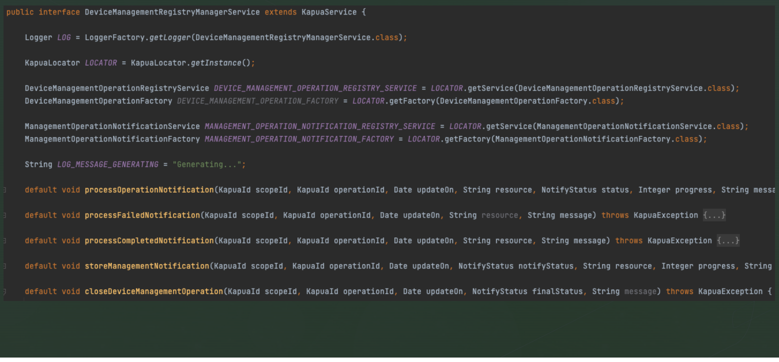
Because Jaxb is not meant to work with DI.

Data classes should not depend directly on collaborators (also called: the case for DTOs)

Static initialization was altered to be non-static, but that’s all (this applies to a few non-jaxb collaborators too)

DeviceManagementRegistryManagerService

- Interface with a lot of default methods, relying on interface fields
- Effectively the same as a fully-static class

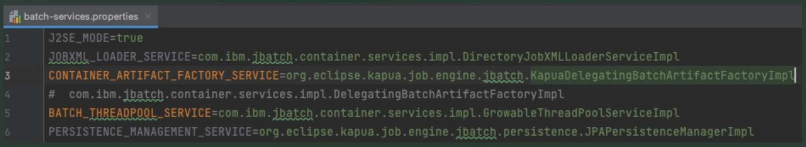


```
public interface DeviceManagementRegistryManagerService extends KapuaService {

    Logger LOG = LoggerFactory.getLogger(DeviceManagementRegistryManagerService.class);

    KapuaLocator LOCATOR = KapuaLocator.getInstance();

    DeviceManagementOperationRegistryService DEVICE_MANAGEMENT_OPERATION_REGISTRY_SERVICE = LOCATOR.getService(DeviceManagementOperationRegistryService.class);
    DeviceManagementOperationFactory DEVICE_MANAGEMENT_OPERATION_FACTORY = LOCATOR.getFactory(DeviceManagementOperationFactory.class);

    ManagementOperationNotificationService MANAGEMENT_OPERATION_NOTIFICATION_REGISTRY_SERVICE = LOCATOR.getService(ManagementOperationNotificationService.class);
    ManagementOperationNotificationFactory MANAGEMENT_OPERATION_NOTIFICATION_FACTORY = LOCATOR.getFactory(ManagementOperationNotificationFactory.class);

    String LOG_MESSAGE_GENERATING = "Generating...";

    default void processOperationNotification(KapuaId scopeId, KapuaId operationId, Date updateOn, String resource, NotifyStatus status, Integer progress, String messa

    default void processFailedNotification(KapuaId scopeId, KapuaId operationId, Date updateOn, String resource, String message) throws KapuaException {...}

    default void processCompletedNotification(KapuaId scopeId, KapuaId operationId, Date updateOn, String resource, String message) throws KapuaException {...}

    default void storeManagementNotification(KapuaId scopeId, KapuaId operationId, Date updateOn, NotifyStatus notifyStatus, String resource, Integer progress, String

    default void closeDeviceManagementOperation(KapuaId scopeId, KapuaId operationId, Date updateOn, NotifyStatus finalStatus, String message) throws KapuaException {
```

TODO: add why change

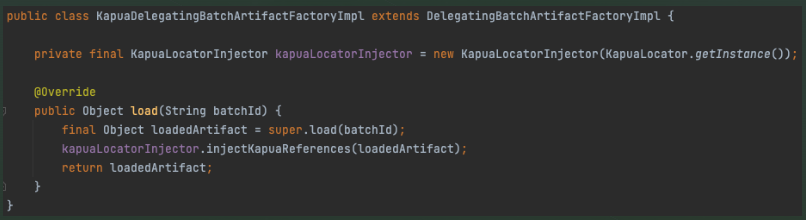
DI with Jbatch

Created a custom Factory Service, configured in batch-services.properties



batch-services.properties

```
J2SE_MODE=true
JOBXML_LOADER_SERVICE=com.ibm.jbatch.container.services.impl.DirectoryJobXMLLoaderServiceImpl
CONTAINER_ARTIFACT_FACTORY_SERVICE=org.eclipse.kapua.job.engine.jbatch.KapuaDelegatingBatchArtifactFactoryImpl
#  com.ibm.jbatch.container.services.impl.DelegatingBatchArtifactFactoryImpl
BATCH_THREADPOOL_SERVICE=com.ibm.jbatch.container.services.impl.GrowableThreadPoolServiceImpl
PERSISTENCE_MANAGEMENT_SERVICE=org.eclipse.kapua.job.engine.jbatch.persistence.JPAPersistenceManagerImpl
```

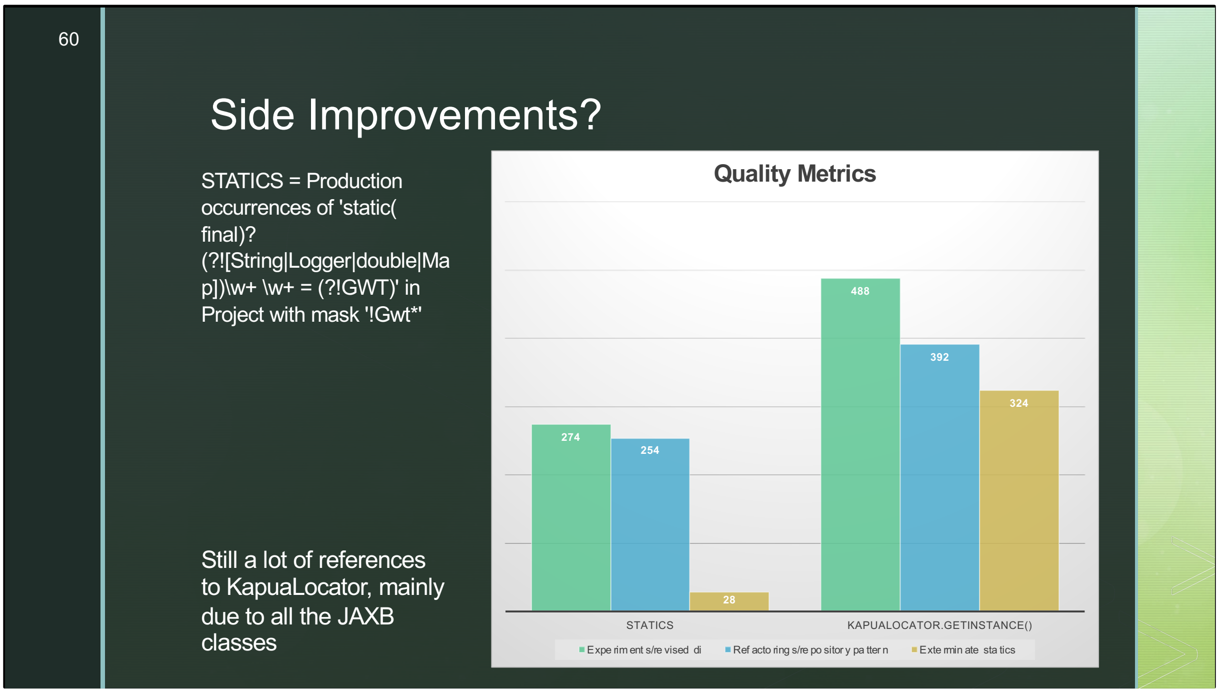


```java
public class KapuaDelegatingBatchArtifactFactoryImpl extends DelegatingBatchArtifactFactoryImpl {

    private final KapuaLocatorInjector kapuaLocatorInjector = new KapuaLocatorInjector(KapuaLocator.getInstance());

    @Override
    public Object load(String batchId) {
        final Object loadedArtifact = super.load(batchId);
        kapuaLocatorInjector.injectKapuaReferences(loadedArtifact);
        return loadedArtifact;
    }
}
```

KapuaLocatorInjector scans for fields marked with @Inject and sets their value, allowing (rudimentary) dependency injection in objects created by JBatch

Part 5: Conclusions

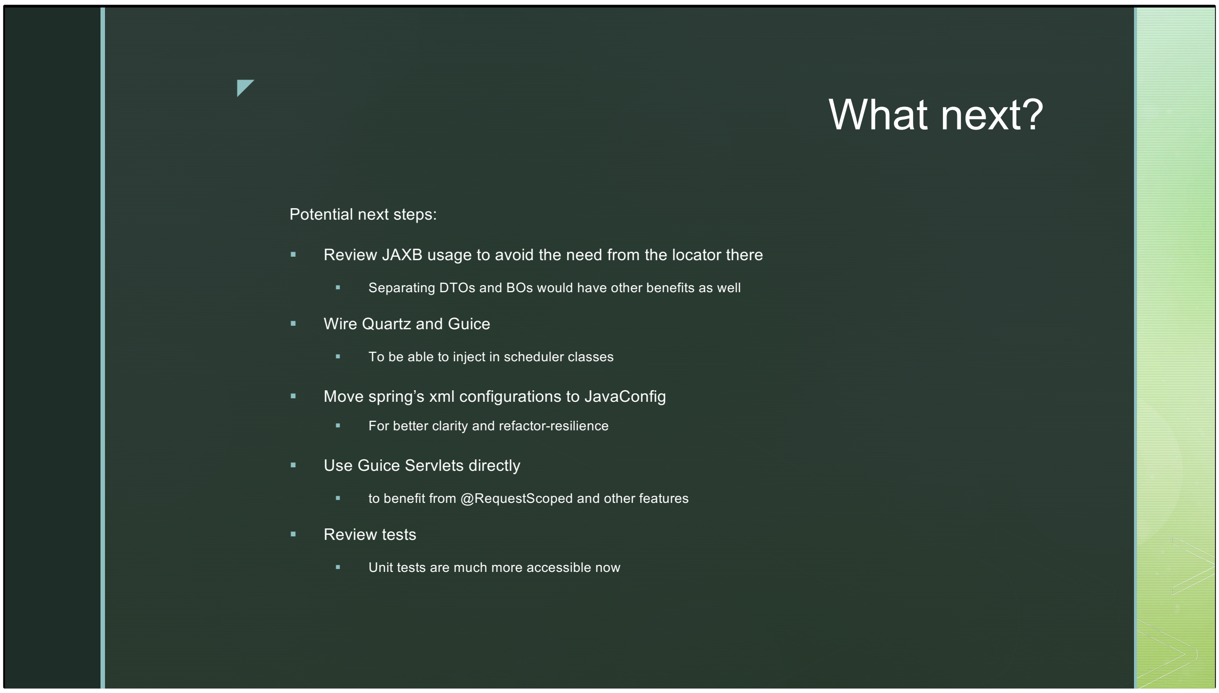
So, where are we?



Side Improvements?

STATICS = Production occurrences of 'static(final)? (?![String|Logger|double|Map])\w+ \w+ = (?!GWT)' in Project with mask '!Gwt*'

Still a lot of references to KapuaLocator, mainly due to all the JAXB classes



What next?

Potential next steps:

- Review JAXB usage to avoid the need from the locator there
  - Separating DTOs and BOs would have other benefits as well
- Wire Quartz and Guice
  - To be able to inject in scheduler classes
- Move spring’s xml configurations to JavaConfig
  - For better clarity and refactor-resilience
- Use Guice Servlets directly
  - to benefit from @RequestScoped and other features
- Review tests
  - Unit tests are much more accessible now