Building a right-sized, do-anything runtime using OSGi technologies

a case study
(sort of)

Erin Schnabel
schnabel@us.ibm.com
@ebullientworks

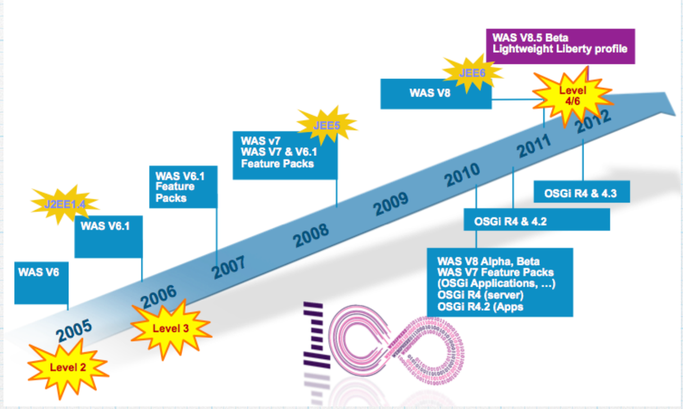
<background>

Some notes on motivation

* The full profile of WebSphere application server is awesome in its capabilities
* It is also well-known that the full profile is not well-suited for development
* We did and do listen... and were presented with a challenge: "Create a light-weight profile of WebSphere that starts in under 2 seconds... [but] Don't break any eggs" — Ian Robinson

History

* WebSphere Application Server (the full profile) has been around forever.
  * Big codebase
  * Big customer base
  * Big workloads
  * ... Big inhibitors to massive change



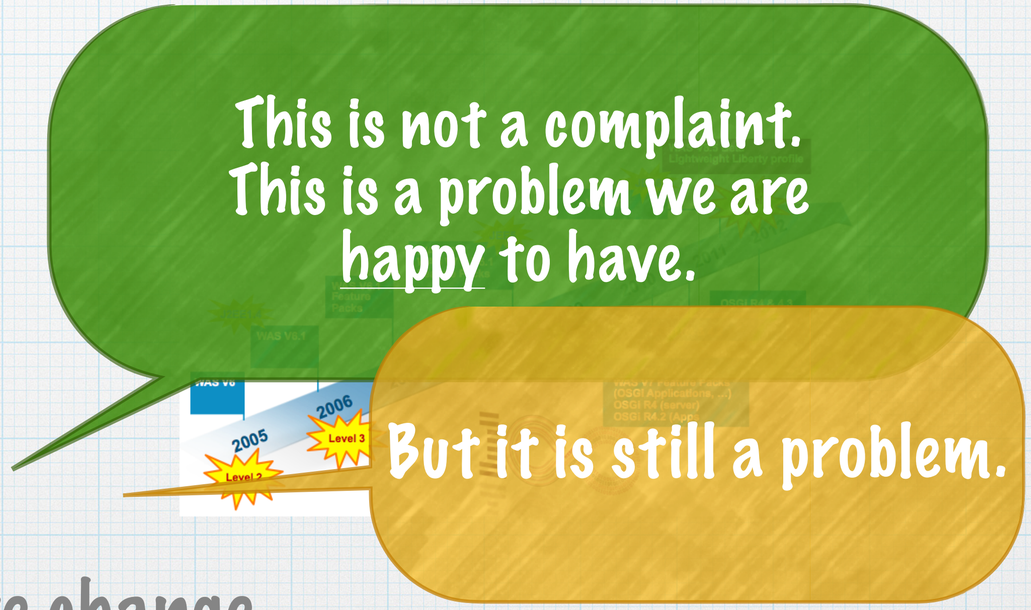
History

* WebSphere Application Server (the full profile) has been around forever.
  * Big codebase
  * Big customer base
  * Big workloads
  * ... Big inhibitors to massive change

This is not a complaint.
This is a problem we are happy to have.

History

* WebSphere Application Server (the full profile) has been around forever.
  * Big codebase
  * Big customer base
  * Big workloads
  * … Big inhibitors to massive change



Code that has been around forever...

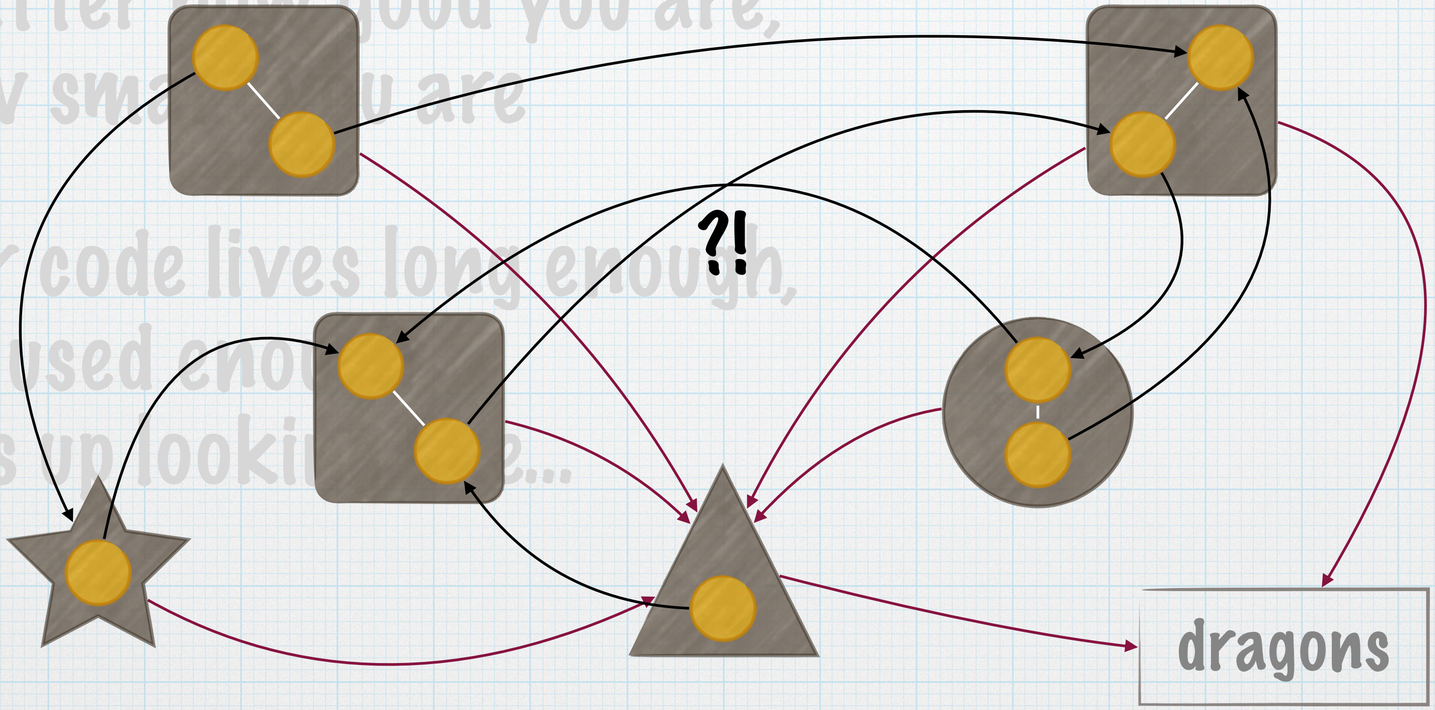
* Doesn't matter how good you are, or how smart you are

Code that has been around forever...

* Doesn't matter how good you are,
  or how smart you are
* If your code lives long enough,
  and is used enough,
  it ends up looking like...

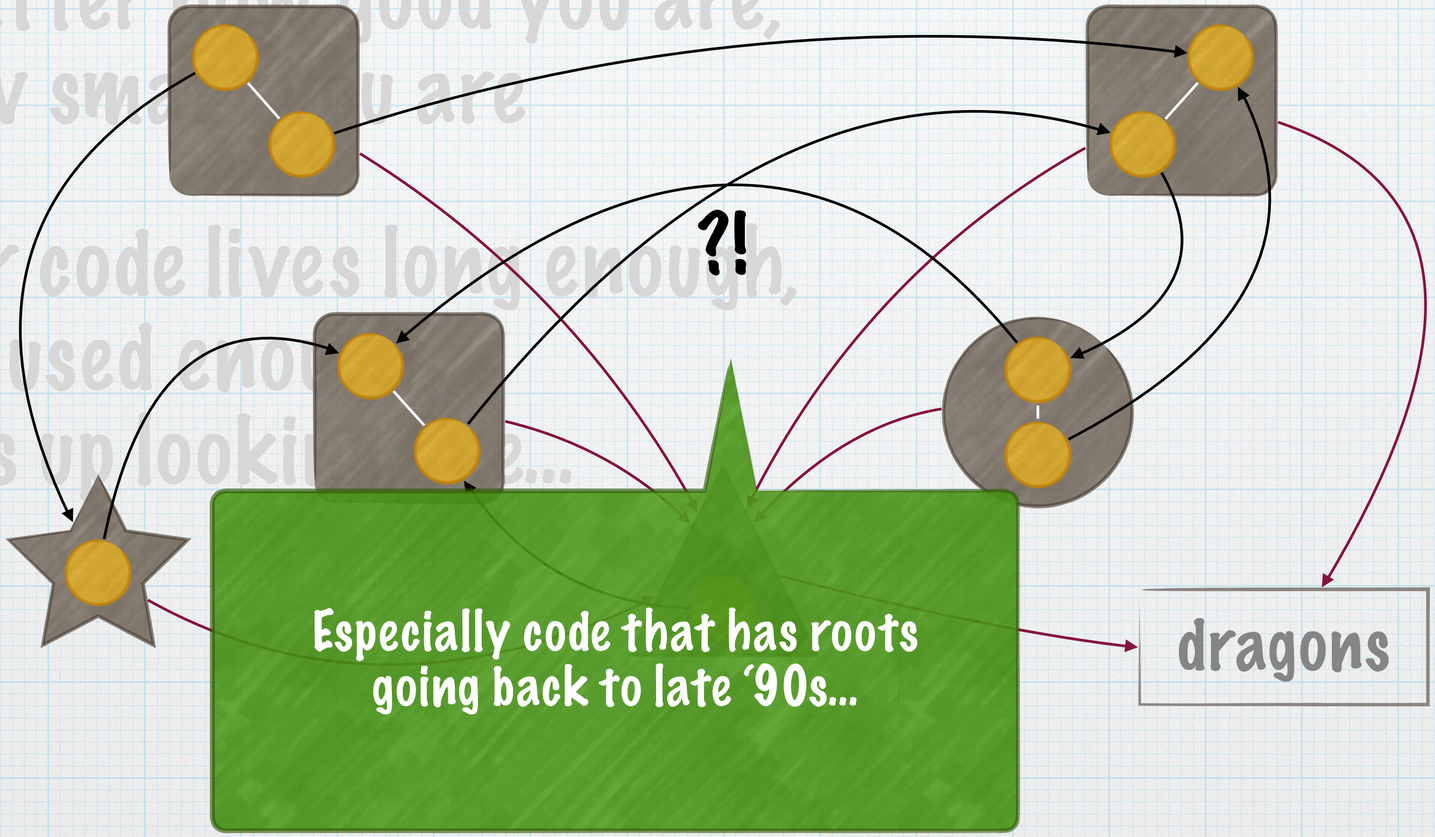
Code that has been around forever...

* No matter how good you are, or how smart you are
* If your code lives long enough, and is used enough, it ends up looking like...



Code that has been around forever...

* No matter how good you are, or how sm[illegible] you are
* If your code lives long enough, and is used enou[illegible] it ends up looki[illegible]...



OSGi and WAS: The first pass...

* OSGi was included in WAS v6.1, in 2006
* Went from lots of arbitrary jars to a few bundles
  * Achieved some modularity enforced by OSGi
* We did not use or expose OSGi services
  * Compatibility constraints: WAS is the bottom of the stack
  * Assumptions about resource initialization and availability
  * Entrenched dependencies between some core elements

</background>

<cleanSlate>

This is the version of the story you won't have heard before...

What if...

If we could start over, what would we want?

* Developer-friendly
  * Simple
  * Dynamic
  * Light-weight
  * Composable / Flexible
  * Extensible

What if...

If we could start over, what would we want?

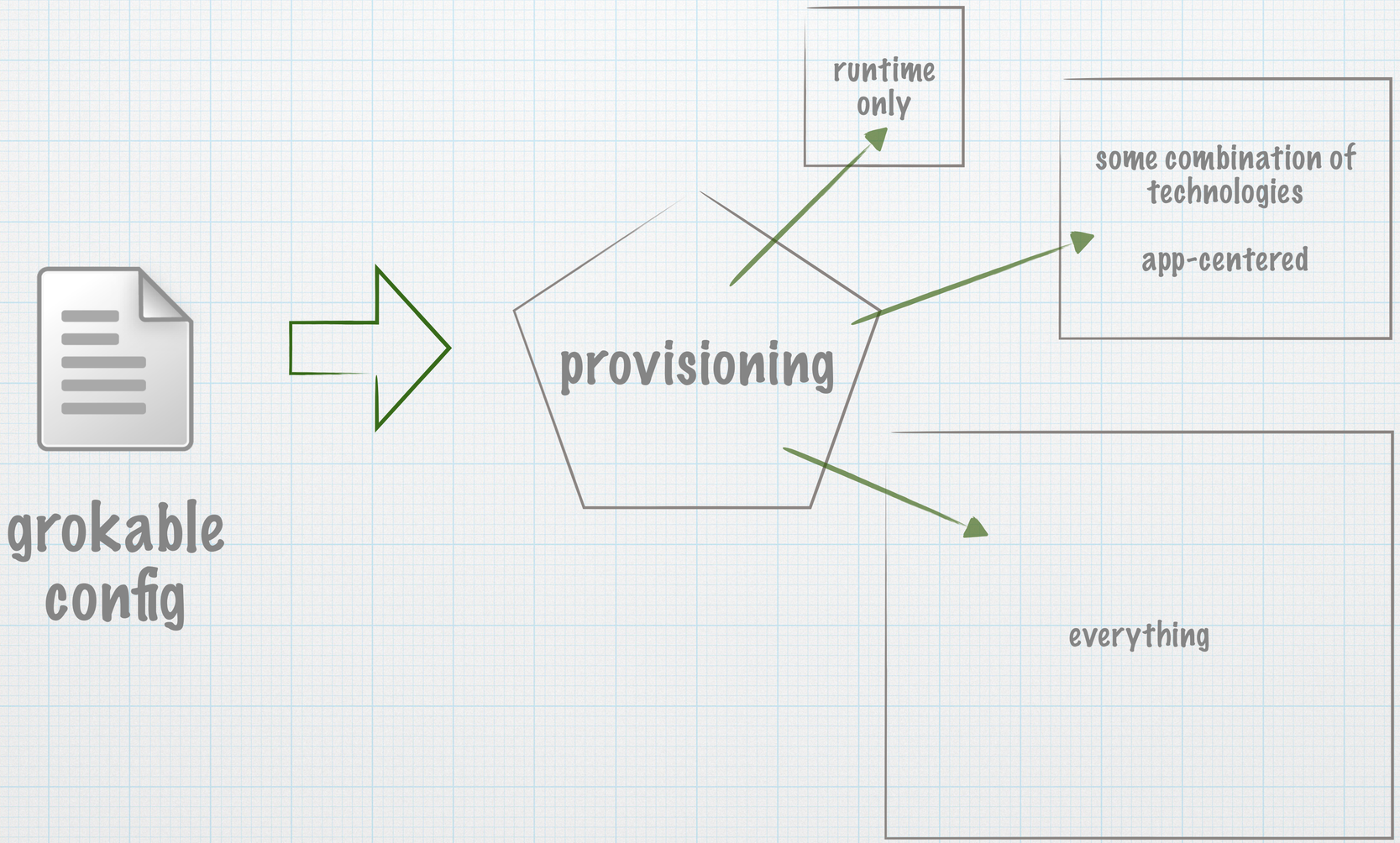
* Developer-friendly
  * Simple
  * Dynamic
  * Light-weight
  * Composable / Flexible
  * Extensible

human usable configuration

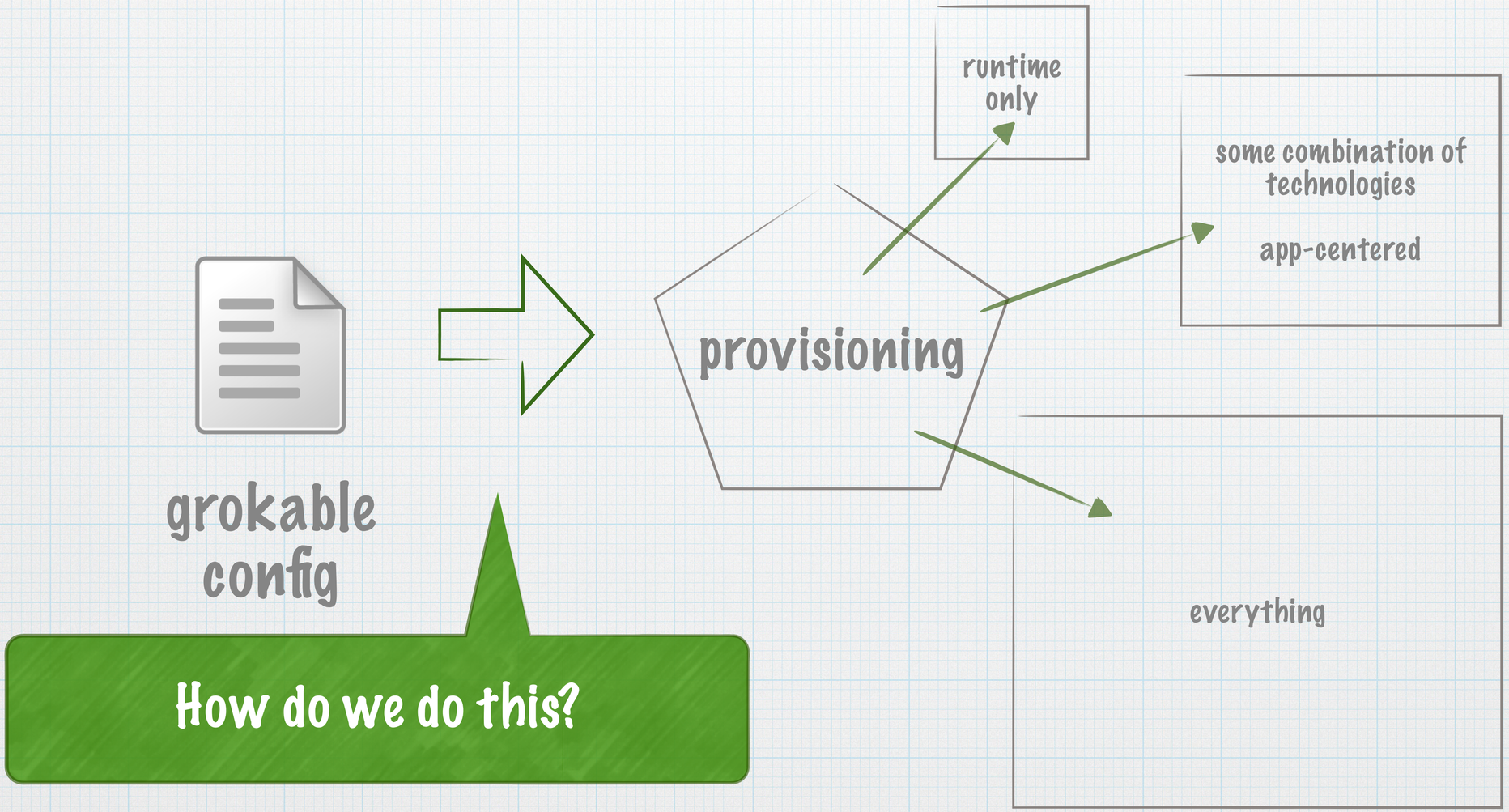
selectable content

clear API/SPI
runtime/app isolation

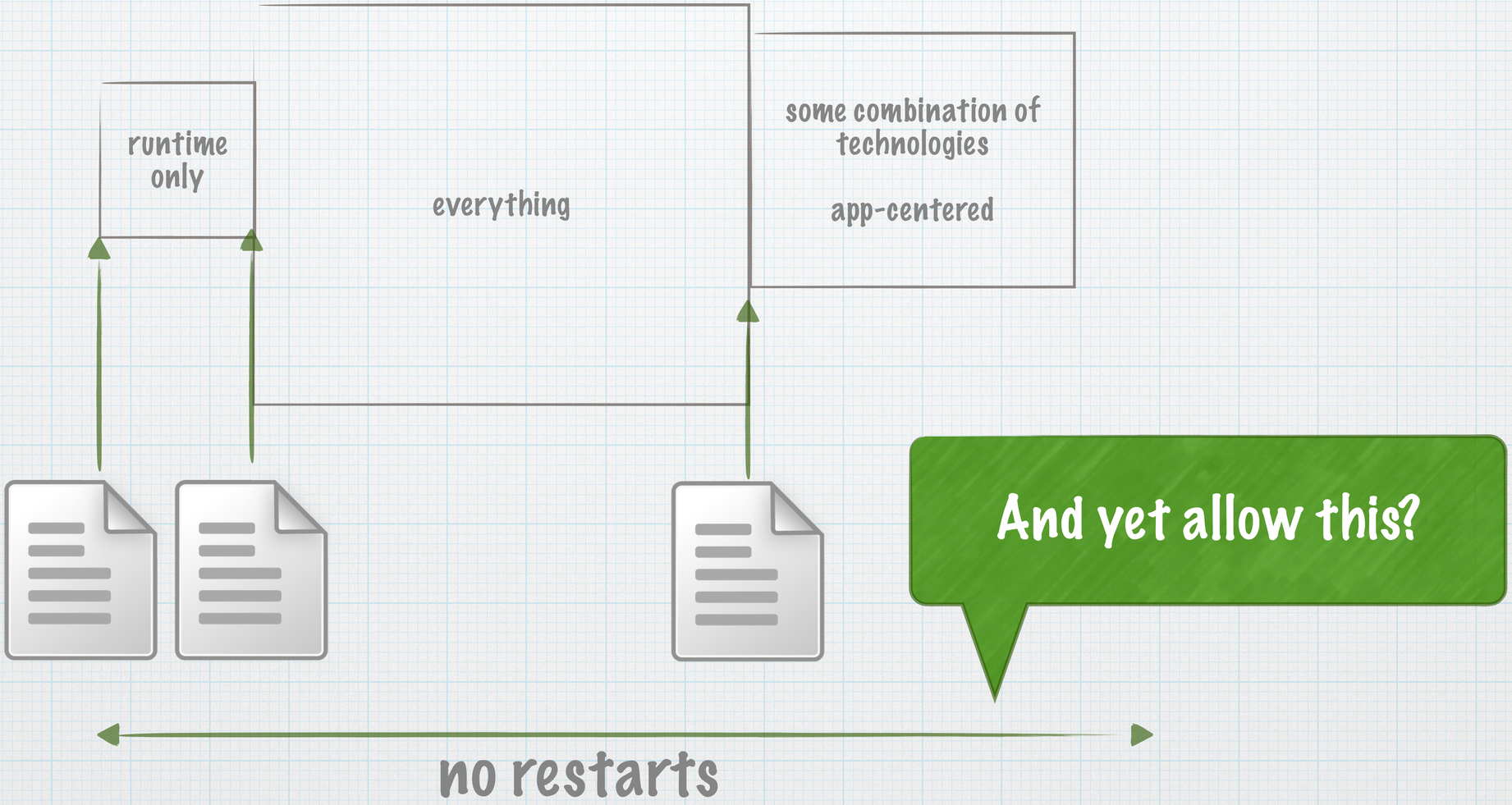
What if...



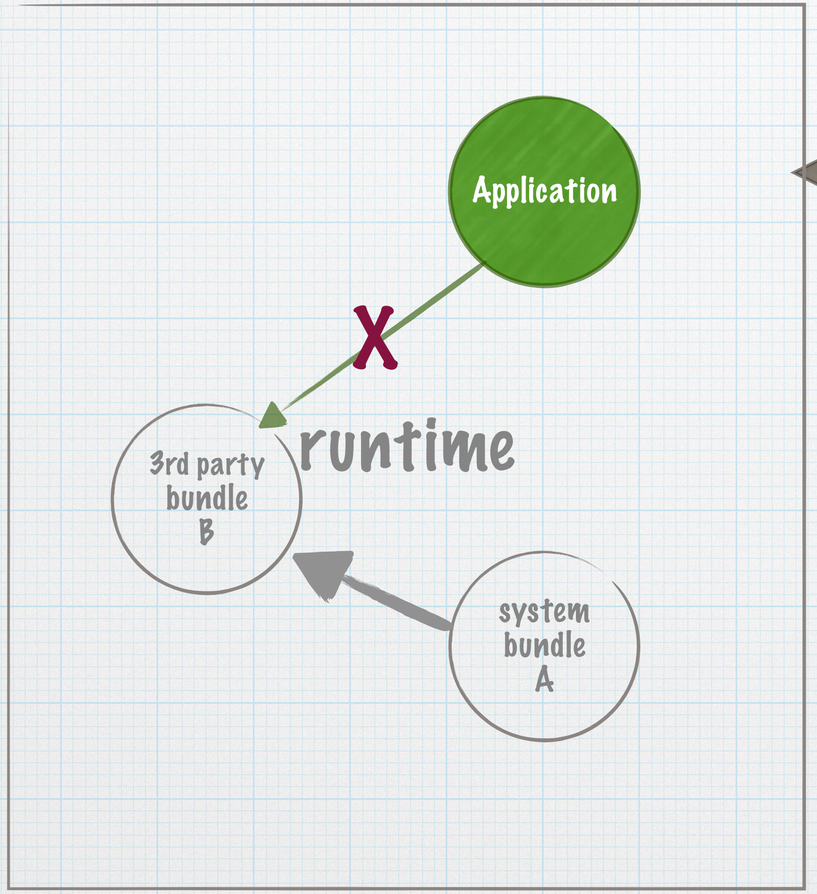
What if...



What if...



What if...



And for crying out loud, can we prevent THIS?!

Building a kernel from scratch

* OSGi-based for all the reasons
* First-class use of OSGi services
  * Must react to configuration changes
  * Runtime composition on-the-fly

Configuration

* Settled on XML for configuration format
  * Ubiquitous
  * Expressive
* BUT, for simplicity:
  * single file
  * usable defaults

```
<server description="simple">
  <featureManager>
    <feature>jsp-2.2</feature>
  </featureManager>

  <httpEndpoint id="defaultHttpEndpoint"
        httpPort="9080" httpsPort="9443" />
</server>
```

Configuration

* Composable system requires composable configuration:
  * Individual components own their config
  * No centralized repository
  * No externally defined global config model

Configuration

* Composable system requires composable configuration:
  * Individual components own their config

Configuration Admin and Metatype #FTW!

Configuration Admin

We rolled our own (sorry)

* Parse and merge user configuration and bundle-provided defaults
* Resolve variables
* Provide configuration to consumers as required by the spec (mostly)

Metatype

Equinox impl + extensions

* Uniform validation of user input
* Define configuration and constraints in one place, it gets used everywhere else.
  * We favor metatype.xml for this reason
* Custom namespace for additional types and validators
  * ibm:type — duration, location, password
  * pid/reference
  * unique, final, variable, etc.

http://www-01.ibm.com/support/knowledgecenter/was_beta_liberty/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/rwlp_extensions_osgi_metatype.html

Metatype

Equinox impl + extensions

* Uniform validation of user input
* Configuration constraints in one place, use else.
  * We favor metatype.xml for this reason
* Custom namespace for additional types and validators
  * ibm:type — duration, location, password
  * pid/reference
  * unique, final, variable, etc.

human readable:
1h30m converted to unit of choice

used by developer tools to help prompt for the right kind of path: file vs. url

http://www-01.ibm.com/support/knowledgecenter/was_beta_liberty/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/rwlp_extensions_osgi_metatype.html

Metatype

Equinox impl + extensions

* Uniform validation of user input
* Define configuration and constraints once, use anywhere else.
  * We favor metatype.xml for this reason
* Custom namespace for additional types and validators
  * ibm:type — duration, location, password
  * pid/reference
  * unique, final, variable, etc.

type="String"
ibm:type="password"

The value is a "SerializedProtectedString", which is not a String.

Developer tools display encoding options: xor or aes, etc.

http://www-01.ibm.com/support/knowledgecenter/was_beta_liberty/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/rwlp_extensions_osgi_metatype.html

Metatype

Equinox impl + extensions

* Uniform validation of user input
* Define configuration and constraints in one place, it gets used everywhere else.
  * We favor metatype.xml for this reason
* Custom namespace for additional types and validators
  * ibm:type — duration, location, password
  * pid/reference
  * unique, final, variable, etc.

This is some crazy stuff.

ibm:type="pid"
ibm:reference="specific.service.pid"

Allows nested configuration elements
to define service relationships

#awesome

http://www-01.ibm.com/support/knowledgecenter/was_beta_liberty/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/rwlp_extensions_osgi_metatype.html

Provisioning

- Two phases of provisioning:
  - Bootstrap the kernel to get configuration
  - Add or remove features based on configuration update
    - Features as in Subsystem features
      (*.esa files, metadata, etc.)
    - Adding or removing features
      installs or uninstalls bundles, which
      adds or removes configurations, which
      triggers the creation or removal of services!

Provisioning

* Two phases of provisioning:
  * Bootstrap the kernel to get configuration
  * Add or remove features based on configuration update
    * Features as in Subsystem features (*.esa files, metadata, etc.)
    * Adding or removing features installs or uninstalls bundles, which adds or removes configurations, which triggers the creation or removal of services!

Dynamically respond to configuration changes at any time without requiring a restart.

#really

Using OSGi Services...

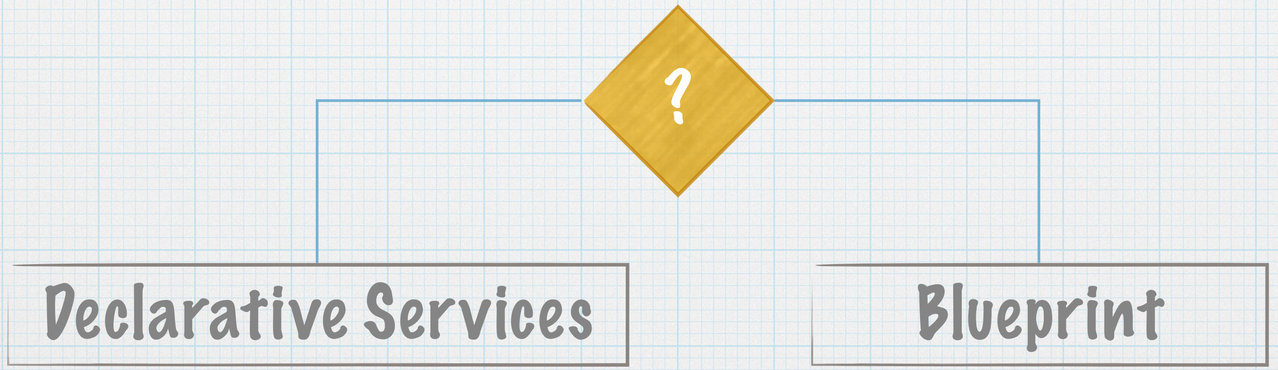
* But who in their right mind wants to manage OSGi services themselves??

Using OSGi Services...

* But who in their right mind wants to manage OSGi services themselves??
  * Exactly. NOBODY.

Using OSGi Services...

* But who in their right mind wants to manage OSGi services themselves??
  * Exactly. NOBODY.



yes, there are others.
We focused on these two.

Declarative Services

- We chose DS for two main reasons:
  - Timing: Blueprint and Aries were just getting started
  - Integration with Configuration Admin and Metatype!
    - Config injected as one unit
      - activate/modified/updated methods
    - Service instance creation based on metatype-declared factory pid
    - DS target filters can be set via configuration

DS is AWESOME!

* DS is a central part of the Liberty runtime
* CA + M + DS = "magic"
  We do insane things with config-derived target filters
* Our runtime would not be what it is without DS in the middle of it
* BUT..

Service dynamics can hurt!

* Service dynamics are a huge hurdle for "new" developers
* DI and IoC can turn even experienced brains inside out if they aren't prepared.
  Thankfully, they do seem to recover.
* Utilities created to "help" can have unintended consequences.
  Especially if cut and paste are involved.
* There is definitely a "better way" to do things with DS..

Let DS do it. Really.

* DS is excellent at managing service dynamics.
* DS is excellent at managing non-trivial service dependencies
* It is very unlikely that you will be able to do better— just let DS do it. That means:
  * Don't register services inside a component
  * Don't manage references inside a component

Isolation

* We mean this in a good way.
* Liberty runtime serves two masters:
  * Typical Application Server paradigm
    (apps strictly separated from runtime) — API
  * Platform extender paradigm
    (the "app" is the runtime) — SPI
* Persistent problem:
  how to allow apps or extensions to use their own versions of libraries that don't conflict with the runtime!?

Subsystems, Resolver Hooks, and Regions... (oh my!)

* Features must explicitly declare API and SPI packages (IBM-* metadata in the feature manifest)
* Isolation between API/SPI, apps/extensions/runtime is enforced in a few ways:
  * Subsystems (the Aries impl) for OSGi Applications (API)
  * Resolver hooks and/or Eclipse Regions for isolation between runtime, extensions (SPI), and containers (API).

</cleanSlate>

Of course, we didn't really get a clean slate.
Application compatibility had to be preserved.

But that still gave us a LOT of room...

Dealing with our legacy

* We did start over with our kernel
* Used the new base to re-group...
  * Lots of code still common with full profile
  * Wrap/Shim: New face on old code
  * Patch: tweak and replace bits where necessary

Thank you!

* Questions?