Seedling

A Platform for Component-Oriented Applications

Todd V. Jonker

Seedling: A Platform for Component-Oriented Applications

Todd V. Jonker

Seedling 0.9.1

Publication date 2012-02-21
Copyright © 2001–2012 Todd V. Jonker

Chapter 1. Introduction

1.1. What is Seedling?

Seedling is a platform for component-oriented applications. By this we mean that a developer uses Seedling as a platform on which to build a complete application from smaller-scale reusable software components. The details beyond that depend greatly on the kind of application you want to build. Seedling's current built-in services are very limited, but the most powerful and important part is the configuration engine and component instantiation and retrieval services (the configuration trees and the actual Seedling component tree).

1.2. Platform Architecture

Seedling components are arranged in a hierarchical namespace, providing the platform's basic “branch/node” metaphor. Each component has a unique address within the tree by which it can be accessed from other nodes. A *configuration tree* specifies the structure of the tree, and the address, type and properties of each node. This is a simple model for organizing and visualizing an application. However, Seedling extends this model in two ways.

The first extension is the concept of *configuration layers*. Instead of allowing only a single tree to configure components, additional trees can be added in “layers”. “Higher” layers can alter the specifications provided by “lower” layers (for example, change the value of an individual component property, or change the type of a component altogether). Each layer can also add entirely new nodes to the application (for example, a new system-wide service, or a new menu item in a GUI).

The second extension to the basic hierarchical model is *scopes*: multiple independent hierarchies, each with its own instances of various nodes. Scopes are themselves arranged hierarchically; there is a single “global scope”, from which are derived any number of “local scopes” (which may themselves have subscopes). A scope's address space inherits the nodes housed in its parent scope, so each scope provides access to a unique set of scoped objects in addition to the global objects available to all scopes. This advanced feature can easily implement many common architectural forms, such as user profiles (where each scope houses component instances unique to one user), HTTP sessions (where each scope houses data specific to a browser session or request), and GUI frames (where each scope holds the widgets and logic of a single window).

Given this explanation, the core platform can be understood as a powerful model of component instantiation, defined by three key concepts:

- Hierarchical namespaces
- Layered configuration of components
- Namespace scope

1.3. Application Construction

A Seedling application (that is, an application built using the Seedling platform) is made up of one or more *modules*. Each module contributes new components to the tree, and customizes the configuration of existing components.

Each module has two aspects: a configuration layer that specifies the module's runtime components, and (optionally) a JAR file that provides any necessary classes that aren't provided by another module. A module can be deployed into an existing Seedling runtime system simply by copying it into a `modules` directory.

The core `runtime` module provides only the baseline configuration and namespace features, the heart of the Seedling platform. Because the runtime has nothing specific to any particular kind of application (client, server, Web, GUI, etc.), support in that dimension is provided by separate modules. You build an application by building one or more modules that run as layers on top of the runtime. Your modules will have code for custom components, and configuration files to install those components into the tree. You'll probably add at least one component to the `StartupNode` system so that your program will do something on launch. Look at `src/ticker` for a complete, if tiny, module.

A long-term goal of the project is to have modules available for many common application services. The project distribution includes support for compiling, testing and packaging Seedling modules.

1.4. About this Document

This document is under active development. Our intent is for it to become a complete guidebook and reference to the Seedling platform: its rationale, its architecture, its best practices, its innermost secrets. As it stands there are gaping holes, but don't let them frighten you. If there's some aspect of the project on which you'd like to gain clarity, just ask about it on the Seedling-user mailing list. We'll answer your question as soon as possible, and probably add a new chunk of text to this document so others will have the information at hand.

Tip

To join the Seedling-user mailing list, browse to the Seedling project page at SourceForge [1] and click on “Mailing Lists”, or just follow this link [2].

[1] http://sourceforge.net/projects/seedling/
[2] http://lists.sourceforge.net/lists/listinfo/seedling-user

Chapter 2. Component Basics

In order to truly understand Seedling, one must understand the problems that it is attempting to address. These problems involve many aspects of component-based application development.

This chapter begins by discussing the general nature of a component, the fundamental unit of composition within a Seedling application. We then examine issues that arise when one attempts to build an application from components. Finally, we derive some specific design goals that would solve (or at least reduce) the most common problems.

2.1. What is a Component?

The term *component* has been given many different definitions by many authors, but Seedling uses the word in a fairly specific manner. We define a component as a class or object that is designed for independent reuse and composition. Let's look at that definition in parts.

class or object: We will at times refer to both classes and objects (instances) as components. Most of the time, a “component class” will be specifically designed for use within a component framework; the specific benefits of components don't occur by accident. An object becomes a component when it is composed with other components within the framework. What makes a class a component is its design intent; what makes an object a component is the way it is used in a specific application.

independent reuse: The primary goal of component-based design is to enable reuse, and thus streamline application development. Component classes should be flexible and configurable, so that they can be applied to many different contexts. A class that cannot be used in multiple contexts is probably not useful as a component.

composition: A component is only part of an application, so that any useful program will involve connecting (“composing”) multiple components. A small application may only require a handful of components, but larger systems will make use of hundreds or even thousands of components. Although it is possible to write code to handle this composition process, the problems with doing so are numerous, so it is advisable to use a component framework to manage the complexity.

Note

Components are closely related to design patterns, but are generally at a higher level. A Visitor won't be a component, but Singletons, Facades, and Factories typically are. In fact, a good component framework solves many of the same problems addressed by such design patterns.

2.1.1. Components are Property-Driven

It's generally accepted that hard-coded values are to be avoided in source code. We should not assume that our HTTP server component must listen on port 80. We must not assume that our database connection factory utilizes the “production” catalog.

Components support good practice by providing *properties* to hold such values. A property is a storage location encapsulated within an object, along with programmatic read- and/or write-access via methods.

Note

Since the Java language does not directly support properties, Seedling uses the idioms established by the JavaBeans specification. The simplest properties involve pairs of “getter” and “setter” methods that meet certain naming conventions.

A well-designed reusable component is likely to have several properties that must be configured (assigned values) to make the component functional. This idiom allows us to avoid hard-coding values.

Binding Time

A related issue involves time: When do configured values become final? A Java constant becomes final at a very, very early point (when the line of code is written). A value read from a properties file when the program is launched becomes final at a relatively late point, since the file could be edited just before running the program. This concept is known as *binding time*, and one can find a broad range of possibilities between early binding and late binding. In general, earlier bindings increase efficiency, while later bindings increase flexibility.

2.1.2. Components are Loosely Coupled

A well-designed component will have low coupling, so that its dependencies are limited. This increases the reusability of the component, because it has less “overhead”, fewer requirements, and can therefore be used in a broader range of applications.

2.1.3. Components are Focussed

An object that does too many things can be hard to reuse, because in many cases the user will only want some subset of the features. Objects with a broad scope will usually require lots of configuration, again making it harder to reuse. A good component follows the maxim to “do one thing, and do it well.”

2.2. From Components to Applications

Many developers new to component-based development find themselves confronted with a steep learning curve. Although individual components are relatively easy to design and use, there is often the sense that it’s a great deal of work for little benefit. Part of the problem lies in the fact that components themselves are not terribly powerful. After all, they are just specialized objects.

This real impact of components isn’t understood until you assemble many components, putting the pieces together into a complete, working application. Since components tend to have fine-grained scope of responsibilities, any realistic application will involve a large number of components. These objects are composed together strategically to express the broader application architecture. At this level you can begin to sense how the component mindset enables architectures that are both flexible and comprehensible.

This leads to an interesting set of challenges. Your architecture requires a number of components that interlock in particular ways. Each component must be instantiated and configured to refer to other components, as well as any lower-level support objects. How does your application determine what to construct, and when? How does it determine the configuration of an individual component?

It is the job of the component framework to provide pragmatic solutions to these and many other challenges. Now we'll consider the problems in detail, and then derive design goals that should help to solve the problems.

2.2.1. The Configuration Challenge

As discussed above, components are property-driven so we can avoid using hard-coded values. If we shouldn't put configuration values into our Java code, where does it go? There must necessarily be resources external to the code that provide these values. Given that, the next question is: how much of the configuration should be specified externally?

We can identify three general areas or dimensions of configuration. First, when is the component created, and when is it destroyed? Second, what is the type of the component? Third, how are the component's properties configured, and how is the component composed with others? As we'll now see, a component framework should support each of these dimensions of configuration in order to meet the needs of a real-life application development and deployment.

Configuring Lifetime

A component's first dimension of configuration is its lifetime. It is generally beneficial to avoid instantiating components unless they are actually needed by the application. For example, a persistence component that opens a pool of database connections should not be instantiated unless the persistence layer is actually used. However, dynamic configuration implies that we may not know until runtime whether this is the case. This means that the framework must support lazy instantiation of components (that is, waiting to create a component until it is first accessed). A more complex example is that of HTTP sessions: a Web application server may need to construct several components for each new browser session, making use of lazy instantiation.

On the other hand, it is also desirable to destroy components when they are no longer needed. For example, when a user's session expires on a Web server, the server should release all of the components and other resources that were allocated to that session. Thus, the framework needs mechanisms to track components throughout the runtime, and ways to add and remove components as necessary.

Configuring Component Types

It is also extremely useful to be able to change the concrete class of a component at different stages of development. For example, it's a common testing technique to replace a database-backed storage layer with an in-memory version that trades long-term capacity and reliability for high speed. If both versions implement a common persistence interface, then a simple configuration change (perhaps as trivial as simply changing a single property defining the name of the concrete class) can switch between the deployable application and one streamlined for testing.

Many well-known design patterns exist to support such late-binding of type, particularly Abstract Factory, Singleton, Facade, and Strategy. However, flexible configuration of types within the component framework greatly reduces the need for these patterns.

Configuring Property Values

Many components will have properties that require different values at each deployment, and at different stages of the build-test-deploy cycle. For example, a server component that responds to requests from the network should have a property that defines the TCP port on which to listen. This port number will probably be different on a developer's machine, during integration testing, and at deployment. This implies that the value must not be hard-coded in the Java, because we must be able to change it at deployment time without recompiling. This in turn implies the need for configuration resources outside the application proper.

The situation becomes even more interesting when we recognize that no component exists in a vacuum. In order to construct a running application from components, they need to be linked together. The outputs of

one component become inputs of another. One component provides a service used by many components throughout the system. The components form an arbitrary web of objects that becomes increasingly complex as an application grows in scope and scale. Beyond the problem of just keeping track of all these components, we must be able to specify what connections are made. For example, if our networking framework provides several TCP connection factories (secure vs. insecure, or maybe a factory for each of several servers), each component making use of the framework must be able to specify which factory it uses. Thus we need a way to identify individual component instances at runtime. Furthermore, we may want to reconfigure any connection (which is really just another property) at deployment time, so the identities must be readable and manageable to humans.

2.2.2. The Bootstrap Challenge

Assuming that we've solved the problems involved with configuring individual components, now we discover another challenge: just getting all of the components constructed in an appropriate order can be extraordinarily daunting task. Writing custom code to handle this process is cumbersome and fragile, especially when the other dimensions are taken into account. For example, if we attempt to integrate a significant new subsystem, there may be a large number of new components to create, and many connections to and from existing components. Changing the type of a component, or changing a connection or two to refer to a different component, can require changing the order in which the components are constructed (and can even affect whether or not a particular component needs to be constructed at all). Our budding component framework must respond to this challenge by automatically determining the order in which components are to be created.

2.2.3. The Modularity Challenge

- install many components together as a complete subsystem
- allows integration of components from multiple sources
- natural unit of distribution and deployment

2.2.4. The Comprehension Challenge

Difficult to understand the program structure... What components are in use? How are they organized? How are they connected?

- Documenting the program structure
- Exploring the running program

2.3. Seedling Design Goals

The challenges presented above lead to the following design goals:

- The framework must be able to dynamically control when components are installed and removed. There should be a facility to automate this when many components have similar lifetime.
- The framework must be able to delay binding values of component properties, including the concrete type of each component and links between components, at least until deployment time, and potentially until application launch.
- The framework must allow the developer to configure components through resources outside of Java code. We call this capability *external configuration*.

- The external configuration resources must be able to identify individual components via some kind of namespace. Since a large number of components may be involved, there must be some mechanism for organizing the namespace.
- The framework must automatically construct, configure, and compose the necessary components as declared by the external configuration. This process must be sensitive to appropriate order of construction, to ensure that components are brought into a well-defined state.

Seedling is a platform designed to provide a powerful way to develop and deploy component-based applications. It is a foundation upon which a developer can express the architecture of her program in terms of components. Seedling can be seen as a “component container” that provides a role similar to that of a J2EE Web container or JNDI, albiet in a much more light-weight fashion so that it can be used across a broader range of applications.

Seedling provides a structural model of component composition, but it does not require or enforce component specifications or contracts. That is, Seedling says nothing about *how* the components work and interoperate. Your components can be as tightly- or loosely-specified as you want.

In order to be a generic as possible, Seedling does not require the use of particular coding idioms, design patterns, or architectural styles. It is entirely possible to construct a Seedling application from custom components that have no coupling whatsoever to any Seedling classes.

Seedling provides advanced facilities for composition and deployment. In Seedling, components are deployed as *nodes* in a hierarchical namespace. Each node has a unique address within the tree, which eases documentation, and provides a mechanism for composing objects: instead of writing code to connect actual object instances together, we write configuration files that connect nodes by name.

Chapter 3. Hello, Seedling!

Sometimes it's easiest to learn by doing. In this chapter we'll walk you through a workday with Seedling, showing you the basic workflow and design concepts step by step.

3.1. Setup

We're going to start with a fresh workspace based on the Seedling Software Development Kit. Begin by going to the Seedling Web site and downloading the latest SDK distribution.

Unpack the SDK archive and place it anywhere in your file system. Go ahead and rename the directory if you want. As you'll see, the Trellis build system on which Seedling is built provides a very self-contained workspace, you can move it around as you wish and nothing's going to break. For the purposes of this tutorial, we'll assume that your workspace is located at the path `~/tutorial`. Once you've got the archive unpacked, things should look like this:

```
[tutorial]$ ls
LICENSE.TXT     bin/            build.xml       seedling/       src/
Welcome.html    build/          lib/            setup
```

Before you do any work, you need to setup your shell environment. This process defines some environment variables so the various tools know where the workspace lives. It also adds the workspace's `bin` directories to your path, so you can run the tools easily.

To setup the environment, just source the `setup` file:

```
[tutorial]$ source setup
Running Ant to install supporting tools and libraries...
Buildfile: build/setupTools.xml

BUILD SUCCESSFUL
Total time: 0 seconds

PROJECT_HOME is /Users/todd/tutorial
To see project targets, type 'ant -projecthelp'
[tutorial]$
```

OK, we're lying here. The first time you go through this process, it's going to be a bit more complicated:

- You'll get an error saying that Ant wasn't found, along with directions about what to put where. Just do as you're told, eat your vegetables, and you'll grow up big and strong.
- When you `setup` again as instructed (Sir, Yes, Sir!), the script will download and install a number of third-party packages into your workspace.
- From then on out, the setup will look as short and sweet as we say above.

Despite our minor subterfuge, you should now be ready to run. Let's go for the gold and build the whole system:

```
[tutorial]$ ant modules
```

```
Buildfile: build.xml

...

module:
Building jar: /Users/todd/tutorial/target/modules/ticker/lib/ticker.jar

BUILD SUCCESSFUL
Total time: 4 seconds
[tutorial]$
```

Guess what? You just compiled a Seedling module! Let's run it!

```
[tutorial]$ runseed ticker
This is Ticker version DEV-todd-200403202332
//tasks/MemoryMonitor: Active memory: 543K/1984K (27%)
//tasks/ClockTicker: The current time is Wed Oct 20 23:18:43 PDT 2004
//tasks/ClockTicker: The current time is Wed Oct 20 23:18:44 PDT 2004
//tasks/ClockTicker: The current time is Wed Oct 20 23:18:45 PDT 2004
```

...and so on. You've just built and run a very simple Seedling application that prints the time every second. Not bad! As simple as it seems, at the very least you've verified that you have a functional workspace, which gives us a good baseline for much bigger things to come.

3.2. A New Module

Warning

This is rough, sorry.

```
$ cp -r src/ticker src/hello
```

Edit build.xml:Add hello to subprojects

Edit hello/build.xml: Set project name. Clear predecessors.

Edit hello/module.properties: Update requires (if necessary).

Now wipe the main and config trees.

```
$ ant modules
$ runseed hello
```

Nothing happens! Well, of course not, our module is empty.

Let's give the module something to do: the insipid yet obligatory “Hello World” application. Everyone knows how to do this, right? Create a new file called `HelloWorld.java` inside the empty `main` directory of the `hello` module. The `main` directory is the root of the module's main Java code; any subdirectories here will correspond to the Java package hierarchy. For now, we'll keep things simple and just define a class in the default package.

```
// file: tutorial/src/hello/main/HelloWorld.java

public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

Wait! Stop! No! Have you forgotten everything you learned in the last chapter? The goal here is not to write procedural applications, but to write component applications. A good component almost never has `static` things, and certainly shouldn't have `main`! Let's try again, placing the output statement in the first available place, the constructor:

```
// file: tutorial/src/hello/main/HelloWorld.java

public class HelloWorld
{
    public HelloWorld()
    {
        System.out.println("Hello, Seedling!");
    }
}
```

There now, doesn't that feel good? We have a nice little component, just waiting to be instantiated into a welcoming world. [1] Let's create another file that will cause just that to happen. The following file declares a Seedling node that will be our first component.

```
# file: tutorial/src/hello/config/Hello.properties

.this = new HelloWorld
```

Some explanation is in order.

- This file is in the `config` directory, not the `main` source directory. The `config` directory is the root of the module's Seedling configuration layer. You'll learn a lot more about it as we work through this tutorial.
- The name of the file corresponds to the name of the Seedling node it configures. This file, `Hello.properties`, configures a node called `Hello`.
- This is a normal Java properties file. This means that comments begin with the hash character, #.
- The only property here looks a little strange, since it starts with a period. This denotes a *meta-property*, meaning that it's an instruction to the container rather than a value being assigned to a property of the component.
- The `.this` meta-property declares an expression that will be evaluated when this node needs to be instantiated. In this case, the `Hello` node will be an instance of the `HelloWorld` class.

[1] Yow, that's annoyingly twee, isn't it? Sorry, bad habit. Do they have a patch for that?

This one little file with one little line is all that's necessary to enable Seedling to construct and install our component. Shall we try it? Don't forget to rebuild the module first.

Chapter 4. Application Deployment

A Seedling application (that is, an application built using the Seedling platform) is made up of one or more *modules*. Each module contributes new components to the tree, and customizes the configuration of existing components. This chapter describes how a Seedling application is organized and launched.

Modules can be broadly classified into two main roles. *Supporting modules* provide specialized frameworks or services, but are not useful in isolation. For example, the `runtime` module provides the core facilities of the Seedling platform, the `jelly` module provides support for XML-based node configuration and scripting, and the `bonsai` module provides a generic GUI framework. *Primary modules* provide an application's custom functionality, typically by pulling together supporting modules and configuring them into a useful system. In general, an application is launched by indicating just its primary module; any supporting modules are automatically loaded by the launcher.

Note

There is no concrete difference between primary and supporting modules; this is just terminology we use to talk about two common ways in which modules can be used.

4.1. Module Structure

A Seedling module has two aspects: a configuration layer that specifies the module's runtime components, and (optionally) a set of Jar files providing classes and resources. Since no module can stand alone (except the core `runtime` module), each one can declare its predecessors: required modules that provide necessary frameworks and/or services. The Seedling runtime launcher will automatically load all necessary modules, determine the appropriate class path, and construct the proper sequence of configuration layers.

A module is deployed in the form of a single directory tree; each module deployed within an application must therefore have a unique name. A module's directory has the following structure:

- ***MODULE/***
  - **`classes/`** (optional) contains a tree of Java `.class` files to be added to the runtime classpath.
  - **`config/`** (optional) contains the module's config tree when “Directory Config Format” is used.
  - **`config.zip`** (optional) contains the module's config tree when “ZipFile Config Format” is used.
  - **`lib/`** (optional) contains additional libraries to be added to the runtime classpath.
  - **`module.properties`** (optional) defines various properties of the module.

The `module.properties` file is a standard Java properties file that is used to configure the module as a whole. There is currently only one property supported:

`requires`
: If defined, this property's value must be a comma-separated list of module names. These are the *predecessors* of this module: each one must be available within the current deployment, and all are loaded into the Seedling before the current module. Note that nodes configured in this module will therefore *override* any configuration provided by predecessor modules.

To start the application, the Seedling launcher analyzes each module and creates a class path with all of the necessary resources. A module may contribute several items to the class path according to these rules:

- If the module defines any predecessor modules, they are processed first, so that their resources *precede* this module in the class path. This rule avoids “class hiding” for security purposes: a module cannot replace classes or resources provided by a predecessor.
- If the module has a `classes` directory, it is appended to the class path.
- If the module has a `lib` directory, any `jar` or `zip` files within it are appended to the class path in an undefined order.

The module's configuration tree can be packaged in several ways. The following list describes the different formats; the first format found will be used.

- In **ZipFile Config Format**, the config tree is stored in a `config.zip` file in the module's directory. This is a standard ZIP file, with resources arranged in subdirectories according to the branch/node hierarchy.
- In **Directory Config Format**, the config tree is found in the module's `config` directory as normal files. You can easily convert this format to the ZipFile format by simply zipping up the *contents* of this directory (but not the `config` directory itself).

The Seedling build system provides an Ant target `buildModule` (defined in `moduleTargets.xml`) that builds the appropriate structures in the module's main Jar file.

4.2. Application Structure

The basic deployment approach is to construct a single directory that contains both the Seedling platform modules and all application modules. The typical file layout is as follows:

- ***APPLICATION-DIR/*** is also known as the `SEEDLING_HOME` directory.
  - **`launcher.jar`** is the Seedling bootstrap jar.
  - **`logs/`** is the default location for any log files created by the application.
  - **`modules/`** contains all of the modules deployed as part of the application.
    - ***MODULE/*** contains a single deployed module. The contents of this directory are described in the Module Structure section.

Since an application is usually launched by naming one or more primary modules, it is acceptable for the deployed system to contain additional modules that may not be loaded (because they are not required by the primary modules). This feature can be used to deploy several applications within one directory tree by providing multiple startup scripts that launch Seedling with different primary modules.

The `modules` directory is known as a *repository*, and Seedling can load modules from more than one of them. The best use for this feature is to avoid modifying the standard Seedling home directory, keeping application code completely separate from the platform, in observance of the Open/Closed Principle. The various launching approaches described in the next section each have a means for utilizing multiple repositories.

4.3. Launching

Seedling provides a spectrum of options for launching the container, depending on how much control the application is willing to give Seedling with regards to `main()` and class loading.

4.3.1. Seedling on the Outside

At one end of the launching spectrum is “Seedling on the outside”, where Seedling takes resposibility for `main()` and for all module and class loading. This is easy to achieve: just run the `launcher.jar` and pass `SEEDLING_HOME` and the names of the required modules.

A typical application can thus be started simply by naming the primary module(s):

```
$ cd APPLICATION-DIR

$ java -jar launcher.jar MODULE ...
```

The launcher inspects the contents of the named modules and recursively loads all of the required modules, each of which must reside in one of the known repository directories. The launcher constructs a custom `ClassLoader` to gain access to the libraries provided by each module (in the `lib` directory).

The *SEEDLING_HOME*`/bin/seedling` script is a generic example of this approach and is very useful during development.

For more information on launching options, see the documentation for the `consciouscode.seedling.launcher.Launcher` class.

4.3.2. Seedling in the Middle

To put Seedling “in the middle”, the application provides `main()` and handles class loading for (at least) the Seedling runtime module, letting Seedling perform dynamic loading of other modules into the container. There are several viable ways to achieve this:

- Add all of the bootstrap jars to the JVM command line. The required jars are *SEEDLING_HOME*`/modules/runtime/lib/*.jar`
- Deploy the application as a jar that has the bootstrap declared in the manifest's `Class-Path` attirbute. Seedling's `launcher.jar` is an example of this approach.
- Construct a dynamic `ClassLoader` that provides the bootstrap jars, then calls a trampoline method via reflection. The trampoline isolates the outer launch code from having a static dependency on Seedling classes.

The go-to class for putting Seedling in the middle is `consciouscode.seedling.boot.SeedlingBuilder` which allows the application to provide `SEEDLING_HOME`, additional module repositories, and the required modules.

4.3.3. Seedling on the Inside

At the other end of the launching spectrum is “Seedling on the inside”, where the application embeds Seedling as a library and handles all class loading itself. This option is not recommended, since it requires the application to determine the transitive closure of required modules and build the appropriate class path.

There is currently minimal platform support for this approach. Again, the important API is the `consciouscode.seedling.boot.SeedlingBuilder` class, particularly the `setDelegatingClassLoading` method.

Chapter 5. A Node's Lifecycle

One of the expected features of a component container is the ability to manage the lifecycle of managed components. By lifecycle we mean the stages through which an object progresses; in plain-old Java code this entails its construction, its preparation and use, and its eventual release for garbage collection.

Seedling defines a more abstract lifecycle model with many extension points that allow you to write components that exhibit desired behavior at each stage of their lives.

The Seedling lifecycle model consists of three broad stages:

- First, a node is *provisioned*, during which it's constructed, configured, and generally massaged into readiness.
- Then, the node is *installed*, which attaches it into the Seedling hierarchy at a specific location, after first activating it as necessary. At this point the node is available for use by other parts of the application.
- Finally, after leading a (hopefully) productive life, the node is *uninstalled* by being removed from the hierarchy and deactivated.

5.1. Provisioning

There's only one way to construct an object in Java: the `new` operator. However, newly-constructed objects are not always immediately ready for use. Seedling defines *provisioning* as the broader process through which an object is prepared for installation into the component hierarchy. A node is provisioned indirectly and lazily when it's requested via `BranchNode.getChild()` or similar access methods: if the requested node isn't already installed in the tree, Seedling will attempt to provision and install it.

There are two provisioning mechanisms: aliasing and creation.

Aliasing is a near-trivial provisioning approach that simply identifies another installed node and inserts a reference to it at a separate location. This serves similar purposes to symbolic links in a file system: it allows the same object to appear in the hierarchy in several different contexts. Since aliasing is so simple, there's not much to say about it here; see the chapter on configuration to learn how to configure an alias.

Creation is when a new Java object is instantiated and configured. This process involves a few more steps and is used in the vast majority of cases. In fact, the ability to flexibly create nodes is one of the main reasons to use Seedling!

Creation of a node consists of a few distinct steps: construction, injection, and decoration.

Construction is the initial acquisition of the object. The standard configuration language implements this step via the `.this` meta-property. Most nodes are constructed by instantiation (keyword `new`), but Seedling can also use a factory method to construct objects indirectly.

Injection is the further modification of the object by calling property setter-methods, as declared by the node configuration.

Decoration takes the injected object and performs one last arbitrary transformation, perhaps resulting in a different object than we first constructed! This step can be used to implement the decorator pattern, to wrap the injected object with a dynamic proxy, or to do to other manipulations of the node before installation.

5.2. Installation

Installation of a node into the tree is a separate lifecyle phase from provisioning. Seedling allows you to install nodes directly via `BranchNode.installChild()`, bypassing provisioning entirely. Still, in most cases nodes are provisioned and installed automatically by the framework.

Regardless of the source of the node, installation consists of two steps: activation and validation.

Activation signals the node that it's being installed into the component hierarchy, so it should start doing whatever it's supposed to do. The standard branches activate nodes using two callbacks.

- If the node implements `LocatableNode`, its `nodeInstalled` method is called with a `NodeLocation` indicating where the node is being installed. This gives the node a reference to its parent `BranchNode`, from which it can access other nodes in the container.
- If the node implements `ServiceNode`, its `startService` method is called.

Note

Activation is not performed on aliases, since the target of the alias must already be active.

After activation, the framework performs final *validation* of the node. This ensures that the node meets all declared constraints before it is exposed to the rest of the application. Typical constraints declare interfaces that must be implemented, but validation is a general process and Seedling will provide further hooks to check that a new node is worthy of installation.

If activation and validation succeed, the node is ready for business and the framework inserts it at the appropriate location in the tree so that other parts of the application can request it.

5.3. Uninstallation

Sooner or later every node will need to be destroyed. This can happen manually via `BranchNode.uninstallChild()`, but usually it happens transitively by deactivating a branch (or the entire Seedling tree).

Compared to provisioning and installation, uninstallation is straightforward: the node is removed from the tree and then deactivated.

Deactivation signals the node that it's being removed from the component hierarchy, so it should stop doing its thing. The standard branches deactivate nodes much like it activates them.

- If the node implements `ServiceNode`, its `stopService` method is called.

And that's it, the node is out of the tree and can be released by the garbage collector (assuming other parts of the system haven't kept references to it).

Chapter 6. The Configuration Language

The workhorse feature of an IoC container is the way it constructs and configures new components. The standard way to configure components in Seedling is through an expression language that's much like plain-old Java but is specialized to the needs of configuration injection. While Seedling doesn't recognize all possible Java expressions, the language should feel natural and the semantics are very close to Java.

6.1. Basic Tech: Properties Files and JavaBeans

The Seedling configuration language is based on two pieces of standard Java technology, properties files and JavaBeans. We should have a quick overview here, but that's yet to be written, so you might want to check out these resources.

Seedling configuration files use the `.properties` format as specified by the documentation for `java.util.Properties` [1] . The expression language inherits a few rough edges from that format:

- Expressions that take more than one line must escape the newline(s) using a backslash.
- String literals must use Pascal-style escapes for the double quote character, as shown below.
- You cannot start end-of-line comments (`# blah`) following other data on the same line.

Seedling leverages "bean properties" as defined by the JavaBeans [2] specification. The most relevant part is the way property getter and setter methods are defined. The Tapestry documentation [3] has a decent overview of how that works.

6.2. Core Literals

The expression language provides literals for these Java features:

- `Boolean`, using keywords `true` and `false`.
- `Integer` and `Long`, using the expected syntax. Unless the (optional) suffix `'L'` or `'l'` is used to force interpretation as `Long`, the type of the literal is “just large enough” to hold the given value. Hexadecimal notation is not currently supported.
- `Float` and `Double`, using the expected syntax. As with Java literals, unless the suffix `'F'` or `'f'` is used, floating-point literals are interpreted as `Double`s. Hexadecimal notation is not currently supported.
- `String`, using Java-style syntax, including backslash escapes (but see below for an important exception).
- `Class`, using the usual fully-qualified syntax. To denote nested classes, use Java's semi-secret internal representation: `package.ParentClass$NestedClass`.
- Static fields can be denoted using fully-qualified syntax. This also works for `enum` constants.

Due to brain-damage in the standard Properties-file syntax, the escape-sequence for the double-quote is Pascal-style, not Java-style: two adjacent double-quote characters.

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/util/Properties.html#load(java.io.InputStream)
[2] http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp
[3] http://tapestry.apache.org/tapestry3/doc/DevelopersGuide/beans.properties.html

```
prop = "Here is a double-quote: ""\nGotta love Pascal."
```

The keyword `null` means the usual thing:

```
prop = null
```

Note that setting a property to null is different from not setting it at all. While the configuration above will cause invocation of `setProp(null)`, one can explicitly avoid calling `setProp()` altogether by declaring an empty expression:

```
prop =
# There's no text in the property value above
```

Such empty configuration is interpreted to mean “there’s no configuration for this property”. That can be useful to override config from another module.

6.3. Identifiers and Paths

Identifiers follow Java syntax. In most cases, an identifier will reference a sibling node of the node being configured, as opposed to referencing a property of the current node.

```
# file: config/branch/Node1

otherNode = Node2
# Configures branch/Node1.otherNode to refer to branch/Node2
```

When two or more identifiers are connected by a slash character, they form a path to another node in the Seedling hierarchy. These paths are interpreted relative to the branch containing the current node.

```
# file: config/branch/Node1

otherNode = childBranch/Node2
# Configures branch/Node1.otherNode to refer to branch/childBranch/Node2
```

6.4. Lists (and Arrays)

Instances of `java.util.ArrayList` can be created by using square brackets around a comma-separated sequence of expressions. The evaluator will automatically convert between lists and arrays when necessary and possible.

```
prop = [ 1, 2, 3 ]
```

Such configuration will work with any of the following setters:

- `void setProp(java.util.ArrayList)`
- `void setProp(java.util.List)`
- `void setProp(java.util.Collection)`

- `void setProp(int[])`

6.5. Concatenation and Addition

Strings, arrays, and `List`s can be concatenated using the `+` operator:

```
url = "http://" + this.host + "/"
statusCodes = /some/Node.errorCodes + [ 2234 ] + OtherNode.codes
```

If the left side is a string (any `CharSequence` will do), then the right side is converted `toString()` before concatenation.

Similarly, integers can be added:

```
port = 80 + this.portOffset
portOffset = 2
```

In either case, if one side of the operator is null, the result is the other side. If both sides are null, the result is null.

6.6. Method Invocation

The expression language can denote method invocation using Java-like syntax. Methods can be invoked on a specific class, in which case the named method must be static, or on another node, in which case the method may be either static or dynamic. Here are some examples.

```
# Invoke a static method on a fully-qualified class.
prop = some.package.Class.staticMethod(12)
```

```
# Invoke a static or dynamic method on a node.
prop = some/branch/Node.method("hello")
```

```
# Ambiguous case resolved by looking for node first, then class.
prop = NodeOrUnpackagedClass.method()
```

Note

Static method invocation currently requires fully-qualified class names.

Method calls can be nested and chained:

```
prop = Node1.method1(Node2.method2()).method3(siblingNode)
```

Methods declared to “return” `void` can be called as expected, with the effective result of `null`.

As a special case of method invocation, sparkly new Java objects can be constructed using the keyword `new` and a fully-qualified class name.

```
prop = new com.eg.Something(12, "hi")
```

In all cases, Seedling will respect declared access restrictions by only invoking public methods.

6.6.1. Selecting Among Overrides

Whenever an expression denotes a method invocation, Seedling must be prepared to select a specific method from a set of overloads. Since the expression language is dynamically typed, we have different information to go on than Java code. In particular, every Java expression has a single static type (since every name has a declared type). Seedling only has dynamic information and any instance may have numerous types associated with it: an instance of class `C` is also an instance of `Object`, and of every other superclass of `C`, and of every interface that `C` inherits. Thus every method-invocation expression with one or more parameters may satisfy several potential overloads.

Seedling approaches this problem in a simple and straightforward fashion. First, it evaluates each parameter expression and notes the type of the result. For each overload with the correct number of parameters, it determines whether each (dynamic) parameter value is assignable to the corresponding (static) parameter type declared by the overload. If the parameter values satisfy exactly one overload, that overload is invoked; otherwise an exception is thrown. When that happens, you'll probably want to use a typecast expression to resolve the ambiguity and select the desired overload.

When a parameter uses this-property notation, the value of the referenced property is first determined by evaluating it as usual. However, the result is then converted to the static property type, just as if it were about to be injected via the property setter. This gives the evaluator more precise type information by which to select a method overload.

6.7. Node Creation

So far we've talked about writing expressions that configure individual properties of a node, but we've not shown how to create the node itself! The value of the current node is declared using the metaproperty `.this` along with an arbitrary expression.

In most cases the `.this` expression will be a normal constructor call:

```
.this = new com.eg.Bean(12, true)
```

```
.this = new java.lang.String("A node can be any object.")
```

More advanced cases leverage method invocation; such methods we herein call *factory methods*. Factory methods can be static or dynamic, just like any method call.

```
# Invoke a static factory method on a fully-qualified class.
.this = some.user.Class.staticFactoryMethod(...)
```

```
# Invoke a static or dynamic factory method on a node.
.this = some/branch/Factory.makeInstance(...)
```

```
# Ambiguous case resolved by looking for class first, then node.
```

```
.this = NodeOrUnpackagedClass.factoryMethod(...)
```

We mentioned before that an empty property expression means “there’s no configuration”, and that goes for `.this` too:

```
.this =
# There's no expression, so the node won't be created.
```

That means you can override a creation expression to completely disable node provisioning. Such configuration is almost equivalent to there being no configuration at all! This can be handy when you're using a module that declares a particular node, but you want to ensure that the node won't be created. (Of course, your empty creation expression could itself be overriden by another configuration layer...)

Another way to end up with a nonexistent node is to have a `.this` expression that evaluates to null, either literally or as the result of a method invocation.

Important

The container will assume that the value of the `.this` expression is not installed elsewhere in the tree, and will perform all of the usual lifecycle operations on it. Violation of this expectation could result in strange behavior, especially if the object uses a callback interface like `ServiceNode`. If you need the same object to appear in more than one location, be sure to use an alias instead of a creation expression that returns the same object.

6.8. This-Property References

An expression can reference properties on the current node by using Java's “this dot” member reference notation:

```
prop1 = "hello"
prop2 = this.prop1
prop3 = node.method(this.prop2)
```

When such notation is used within a `.this = new` constructor expression, the named property will *not* undergo property injection. The evaluator assumes that the property utilizes constructor injection and won't inject it a second time. For nodes with many constructor parameters, this idiom can increase the readability of the configuration file by making the parameter/property association explicit.

```
.this = new com.eg.MyList(this.initialSize, this.expandable)
initialSize = 12
expandable = true
```

Despite the fact that no setters will be invoked for property injection, Seedling still needs to be able to discover the type of the properties. This implies that `MyList` exposes a property called `initialSize`, meaning it has a getter `getInitialSize()` and/or a setter `setInitialSize(...)`. (And the same goes for `expandable`.)

Factory method invocations cannot use this-property names as parameters. That's because the concrete type of such node is not known until after the factory method returns, so the evaluator can't determine what properties are on the node until its too late to reference them.

```
# This will cause an error:
.this = path/to/MyFactory.makeNode(this.prop)
```

6.9. Super References

Property assignments fully replace any expressions declared in super-configuration. This is normal overriding of inherited configuration. But there are times when the subconfiguration needs to extend or otherwise modify the inherited value rather than replacing it. In the Seedling expression language, the `super` keyword means something like “the value of the current property were it not being overridden here”. It's much like the meaning of the keyword in Java itself when invoking a method as implemented in a superclass.

For example, consider a node that has a property `ports` of type `int[]` and a base configuration file declaring:

```
# superconfig
ports = [ 80 ]
```

A subconfig file (usually found in a higher-priority module) can extend the array as follows:

```
# subconfig
ports = super + [8080, 8081]
# The resulting configured value is [80, 8080, 8081]
```

The `super` keyword is not yet supported in all contexts in which it could be useful, for example:

```
# Not supported!
prop = node.method(super)
```

6.10. Class Expressions

Classes are mostly first-class in the config language: for most expressions where Java requires a class literal, Seedling can handle an arbitrary expression evaluating to a `Class` instance.

- If a class-expression is follow by a dot-name chain, then it denotes access to static members, not to members of `Class`. This is consistent with the usual Java syntax for static access. For example, if the `Class` instance `java.util.Collections` is installed at node `T` then the expression `T.EMPTY_LIST` evaluates to `java.util.Collections.EMPTY_LIST`. Similarly, `T.emptyList()` evaluates to `java.util.Collections.emptyList()`.

- To get access to members on the `Class` instance itself, use the special form `T.class.member`. Here, `class` is interpreted similarly to the same notation in Java. For example, if the `Class` instance `java.util.Collections` is installed at node `T` then the expression `T.class.getSimpleName()` and the expression `T.class.simpleName` both evaluate to the string `"Collections"`.

- The one context in which this doesn't behave regularly is invoking a constructor, when the class expression ends with a property access. For example, if node `N` has a `getType()` method returning a `Class`, then one might expect the expression `new N.type(...)` to construct a new instance of that class. However,

that runs into a problem because the syntax looks like an invocation of a method called `type(...)`. To work around this, one can use method invocation syntax to read the property: `new N.getType()(...)` will do the right thing.

6.11. Casts

Values can be coerced to a specific type via the usual Java notation. This can be particularly useful for disambiguating among overridden methods. Here's a ridiculous example:

```
# Disambiguate between Exception(String) and Exception(Thowable)
.this = new java.lang.Exception( (java.lang.String) null )
```

Note that Seedling currently requires all class names to be fully-qualified.