AvalonFortressDesignDocs

Fortress Design Docs

Fortress has two design goals: facilitate heirarchical containers and take management functions outside of the *critical path*. The critical path is the code execution path that is required to find and use a component. Fortress assumes that the developer has explicit knowledge of his domain--which Fortress itself would never have any knowledge of. It also assumes that there is one root container, although it does not force that upon the developer.

Asyncronous Management

Due to the long startup times of certain components like the DataSourceComponent ECM based code suffered from slowness. The problem was also made worse by the delayed loading and running of components. Components would only be instantiated when they were first looked up--which made problems for components that needed to be started immediately.

Fortress makes use of the Event package's CommandManager so that all components can be started up immediately, but it is done in the background. That means that components are still starting while Fortress is ready to work. If a component hasn't been started yet before it is needed, then Fortress will make sure it starts before it turns over the requested component. It will also make sure no component gets started twice.

All component pool sizing and management is done by background threads so that as Fortress responds to requests for components, it manages resources without adding that cost to the client code. That means the critical path (the code that actually does the work of the system) is not delayed unnecessarily.

Hierarchical Containers

Part of the design concept for Fortress heirarchical containers is to use a ContainerManager to make sure all the necessary services are set up and running. For example, the Fortress container needs a CommandManager--so the ContainerManager checks to see if it is already set up and uses it. That way we can have one Container that has one or more ContainerManagers that all use the kernel level services of the parent container.

The kernel level services are: CommandManager, InstrumentManager, LoggerManager and ThreadManager. The actual setup and configuration of these services are done using a Context. The choice for the Context object was a conscious decision because we didn't want to extend the objects in a proprietary manner (LoggerManageable, etc.) like the ECM did. By passing the kernel services in the context, the kernel services can be forwarded to any child containers.

To assist in the setup of the Context, Fortress uses a ContextManager. The ContextManager will either set up the context based on a Context passed in, or from a default context. Once the ContextManager assists the ContainerManager to set up any missing kernel services, you can get the Container from the ContainerManager and start using it.

Why Not Set Up a Standard Container Interface?

Each domain has its own needs. For instance, Cocoon is based on a request/response processing model. Component based tools are based on a useage model. Swing based Apps are based on other models. There is no one size fits all solution, and Fortress can be used in all of these solutions. As an interim solution, the DefaultContainer does have one public method exposed: getServiceManager().

Why Not Use a Central Kernel?

This was actually planned in a future release. There are some issues to work out with a central kernel though. Those issues include how to detect and set up sub-containers, how to make sure the container instance you want is set up instead of the default version, etc. In essence, what is needed is *meta information*. Meta information is information about the container heirarchy and the components involved. In the future Avalon Container: Merlin release, we will have a proper meta model.