BlockImplementation

Design Constraints

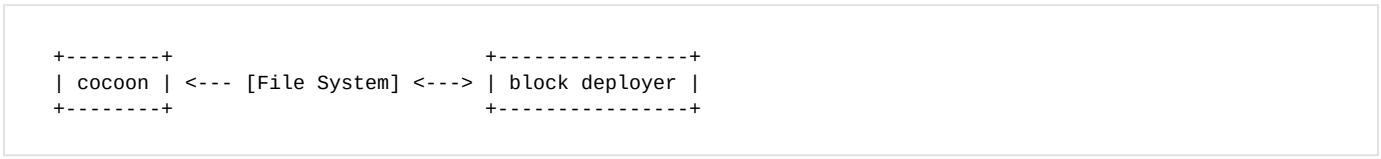
1. impact on back compatibility should be minimal, optimally none. that is: everything that worked before the introduction of blocks should continue to work with no required changes (this will reduce migration issues)
2. the implementation should be incremental and evolutionary. no radical changes in the cocoon architecture should be created (this will reduce the amount of code to write and also provide better regression)
3. security of the architecture for block managing and deploying is a **TOP** priority and should be introduced up front.
4. deployment should be system administrator-friendly. That is: should *NOT* require GUIs or webapps (even if it should allow them to be possible).

The overall architecture

Let's start with the first requirement: security.

Blocks are functional components at the webapp level. If a user is able to change the block wiring, the user is, potentially, able to execute his/her own code with the same security level of the entire cocoon application.

For this reason, the block wiring information should be located in a configuration file that is "read-only" by cocoon and "read/write" by the block deployer.



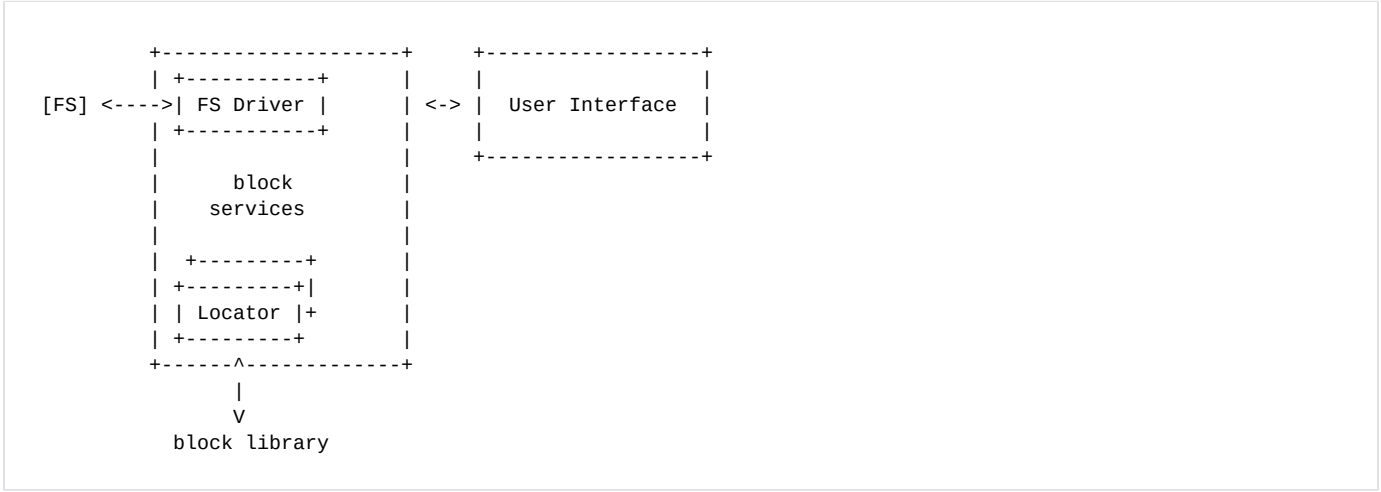
Note that the block deployer *could* be anything (a CLI, a webapp, an eclipse plugin). The above meets our second requirement: user friendlyness for all types of users.

Also note that it meets, potentially, the ability for the system administrators to perform actions such as 'staging' and 'cluster replication' by simply performing a file copy. Cocoon should be able to reload the block wiring information if this is changed.

In order to improve security and avoid DoS, there is *no way* for the block deployer to signal directly information to the cocoon instance (and no way for the cocoon instance to modify the wiring information or to communicate directly with the block deployer). Everything is performed thru the use of the file system.

The block deployer

The block deployer architecture is the following



which is composed by four main parts:

1. the file system driver: the part responsible for reading/writing the block wiring information and block configurations, to extract the files from the blocks distrubution archives and physically deploy the extracted files on the file system. There is no need for polymorphism for this part since there needs to be a solid file system contract between this driver and the cocoon block manager (included inside cocoon) which will need to read the block wiring info and locate the files on the file system.
2. the block locator: the part responsible for locating the metadata associated with a given block identifier and thus, provide enough data for the block services and the user interface to drive the installation process. This part needs polymorphism. Potential implemenation of this locator are:
    - "file system"-based locator: the block metadata and location information is stored in a file on disk.
    - "network service"-based locator: the block metadata is provided by a network service (for example, a web service).

The block deployer can use multiple locators at the same time, in a cascading way: it should be possible to configure the block deployer with the kind of location services and provide a priority for which one to use. This allows, for example, to provide an architecture for block discovery that could work like this:

```
block deployer ---> company block library -> cocoon official library
```

(a collection of blocks is called a "block library". the application that, given a block identifier, looks up its metadata is called "block librarian".)

1.the block services: the part that is shared by all potential block deployers (no matter how the user inteface is implemented).

1. the user interface: the part that is driving the block services but it's dependent on the user interface.

The Block Manager

The block manager is the part that is responsible for handling the block wiring information. This is included inside cocoon and it can read and interpret the block wiring information written by the block deployer.

The block manager is the only part of cocoon that knows how block are wired together and where their actual location on disk is.

The block manager will be queried by all the cocoon internal services that need to locate block-dependent stuff, that is:

1. the sitemap interpreter: to find out where the blocks sitemaps are mounted in the main sitemap URL space
2. the block: protocol: to locate the services provided by the blocks
3. the component manager: to locate components provided by the blocks (either avalon components, sitemap components and VirtualComponents)