GT2003RealBlocks

Stefano's mini-talk at the GT2003 Hackathon.

Raw notes taken collaboratively during the talk.

Real Blocks - introduction by Stefano

The Naked Cocoon

Today Cocoon is seen as big servlet by users. With Blocks, you would install a basic Cocoon (core components + minimal 'hello world') and you'd plugin at runtime, the functionality you need.

Blocks vs. WAR files

Blocks could be seen as similar to WAR files, but will do more. The WAR file equivalent could be a first phase, but applications loaded from WAR files are isolated and cannot communicate or depend on each other, whereas Blocks are seen as collaborative (yet isolated) components.

- There is no communication between War files, but Blocks can have dependencies and use each other's services.
- Blocks can also inherit and extend the functionality of other Blocks.

What are blocks (dry definitions)

- Application-level component model for Cocoon
- Units of Cocoon functionality, using clean contracts to communicate between the Cocoon core and Blocks, and also between Blocks themselves.
- A block exposes a **sitemap** and **Avalon Components** (both optionally)

Blocks use cases

- Being able to test and rollback installation of new blocks
- Presentation skins: contract between the content model and the skin
- Having different, versioned instances of blocks running side-by-side (how does one specify that in the block 'connection URI'?

Challenges

- What interfaces does a block expose? Some XML grammar it supposedly requires to act upon?
- Tasks of the DeployManager: keeping classloading sane
- Classloading: Blocks will need to expose some java classes to the outside, but not all of their public classes. If a block uses an outside library, for example, the public java classes of the library should not necessarily be available outside of the Block.

Additional notes

Block extension:
Wrap one block with another.

Block Extension:

I can say this block can have a contractual relationship, at runtime, with another block. Calls to base block are extended by (intercepted by) the extending block. If the extender does not handle, it falls back to the original, which can handle it, or pass it back to Cocoon itself.

Block Reuse:

I ask for a PDF block, but there are 3 implementations. The deployer asks, "which one would you like?". I choose one, the deployer goes off to download it, and hooks it up. Another Block asks for PDF. Cocoon 'knows' I am already using it, it can be used by the new Block without duplication.

How do we describe this behaviour? Not in a way that can be validatable

Use a weak typing ?: using a URI to locate blocks

You can switch blocks at runtime, test, and rollback if required

Will not directly expose public methods of enclosed components

Remaining issues

- Any use case/functionality missing?
- How to implement this from scratch?

    - What parts of Cocoon need to be changed?
    - How do we reduce the amount of code we touch?
- How do we manage over-componentisation from a community standpoint?
- How do we implement security checking, with digital certification?
- How do we implement the blocker deployer tool?
- How do we trace and debug dependencies?
  (The block deployer reports on it's job)
- What are we going to do with JMX? For blocks - and for the Cocoon core. Should be orthogonal to the design behind blocks.
- Implementing clustering?
  Block deployer should be able to replicate a snapshot of the running system onto multiple machines