Apache Avalon

Application Framework For Dummies ☺

Presentation Goals

Raise and organize users' knowledge on the Avalon Framework & its containers to the point when participants are able to complete their own "HelloWorld" application.

Involve team members in a constructive discussion on the Avalon-based development best practices.

Agenda For Our Journey

- Introduction to the Avalon "umbrella" project.
- A discovery trip into Avalon's framework.
- Introduction to Avalon's libraries/subprojects.
- Fortress as an entry level, lightweight container.
- Analysis of the Phoenix application server.
- Application server deployment & administration.
- A brief comparison of Merlin and Phoenix.
- Framework's advantages and drawbacks.
- Questions and open forum.

What's Avalon After All?

- Framework
- Excalibur
- Cornerstone
- Fortress
- Phoenix
- Merlin
- Small Utilities

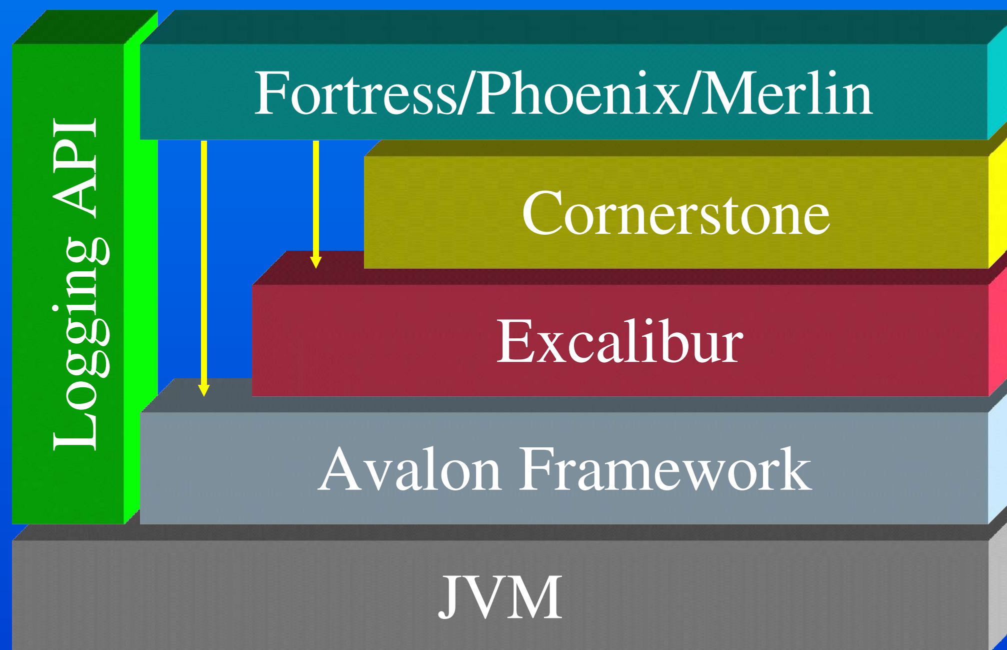
Avalon subprojects with their own development lifecycle and releases.

Avalon's Major Players

- Framework ⟵ Interfaces, contracts, default impl.
- Excalibur ⟵ Server-side component library
- Cornerstone ⟵ Server-side services library
- Fortress ⟵ Lightweight container for depl. & exec.
- Phoenix ⟵ Server kernel for deployment & exec.
- Merlin ⟵ Server kernel for deployment & exec.

Subprojects' Dependencies

- They can be used together or independently.



Avalon Umbrella Project (1)

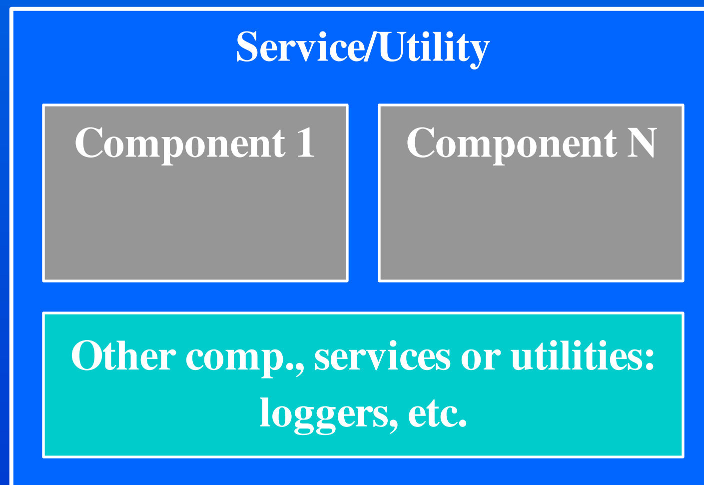
- Major project properties:
  - Focused on the server-side solutions.
  - Well suited for horizontal market frameworks.
  - Can be easily integrated with any J2EE framework.
  - Subprojects can be used for java GUI applications.
- Four major design concepts:
  - Inversion of Control (IoC).
  - Separation of Concerns (SoC).
  - Component Oriented Programming (COP).
  - Service Oriented Programming (SOP).

Avalon Umbrella Project (2)

- Advantages of the COP architectures:
  - System can be broken up into small, reusable facilities.
  - Provides distinct work interface and its implementation.
  - Uses work and lifecycle interfaces and their contracts:
    - Allows for easy component replacement.
    - Reduces complexity between business units.
    - Gives us a higher level of integration than with classes.
  - Provides a loose coupling between logical units.
  - Can be used to create low-level subsystems.

Avalon Umbrella Project (3)

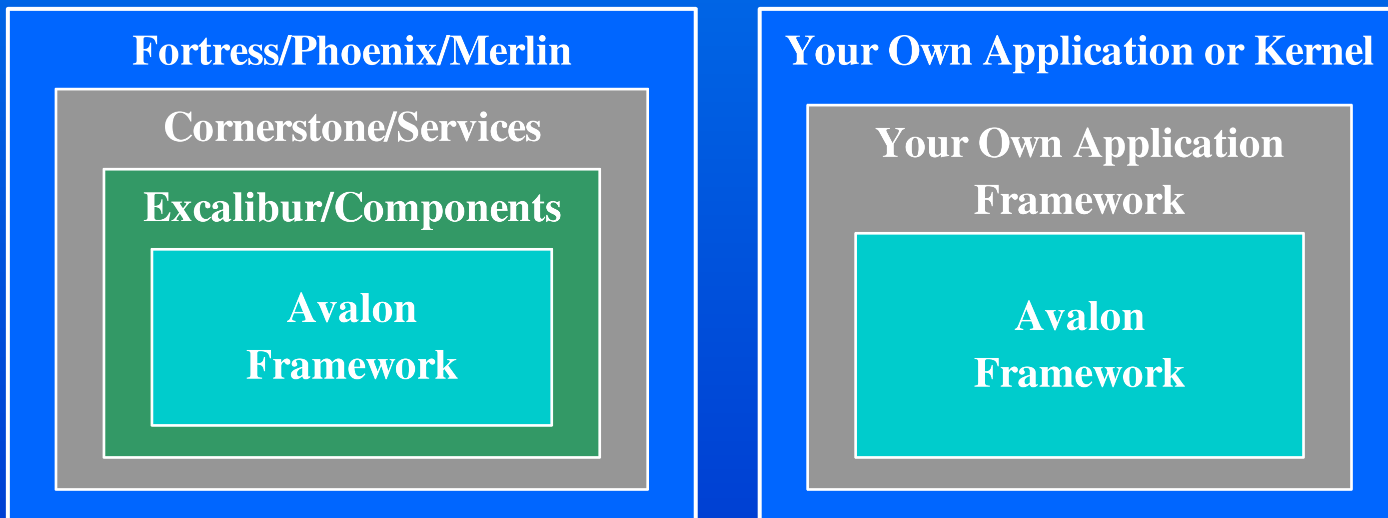
- Advantages of the SOP architectures:
  - Excellent for top-down system design approach.
  - One or more services can provide a complete solution.
  - Allows for creating high-level subsystems …
  - … that can be upgraded to an executable utility level.



One or more service(s) can be wrapped up by (or become) your parent project.

Avalon Framework (1)

- Framework's relationship to other projects:
  - Provides basis for all other Avalon subprojects.
  - Addition to your work interfaces and implementations.
  - Can be used independently in your own solution(s).

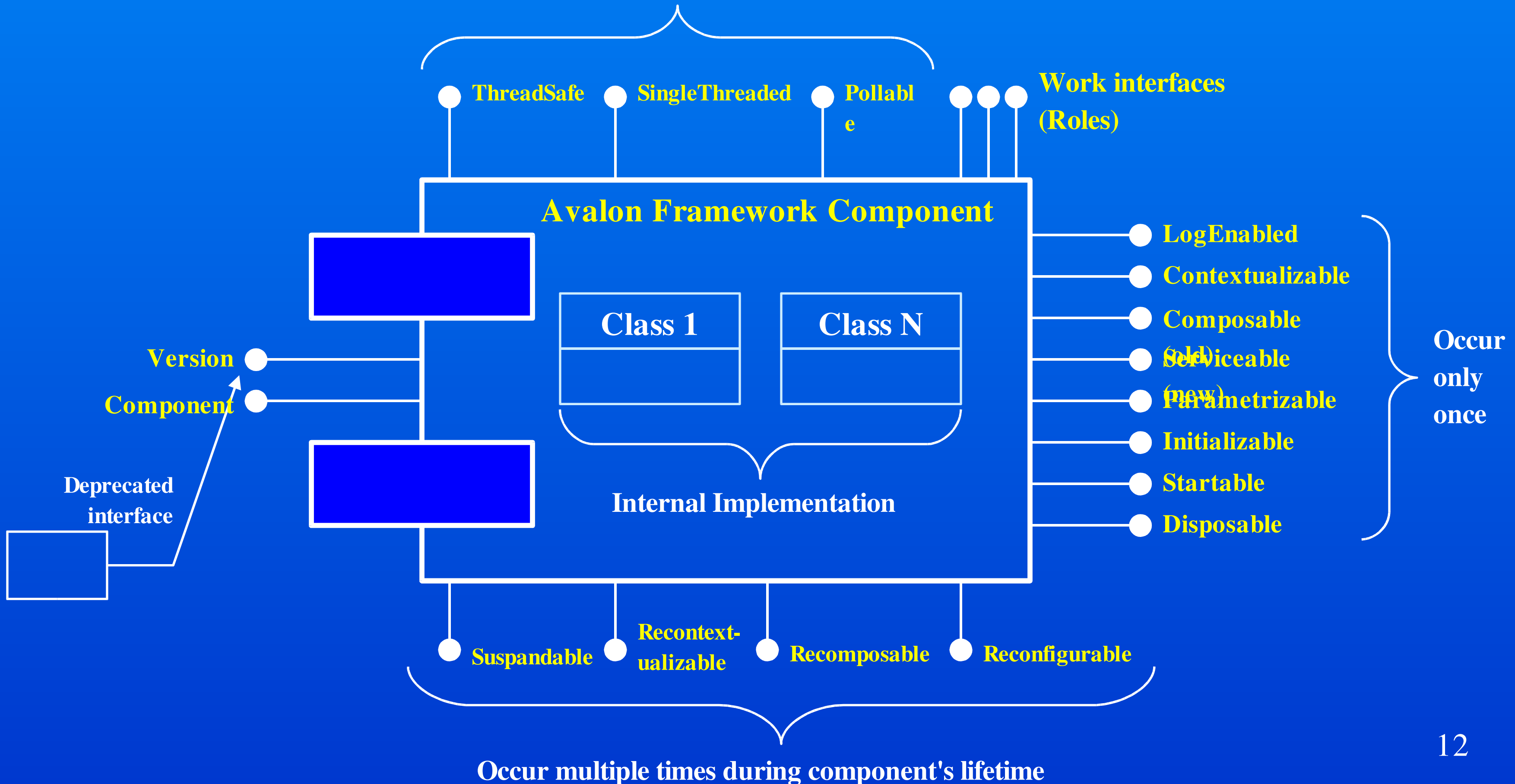


Avalon Framework (2)

- Components as major app. building blocks:
  - They live within a context called a container.
  - Can implement several interfaces to identify all the areas of concern for the project.
  - The container is required to take a component through all of its lifecycle stages.
  - Some of the lifecycle events handled by a component must be propagated to all of its children implementing the appropriate interfaces.
    - » This includes Initializable, Startable, Suspendable and Disposable.

Avalon Framework (3)

- What is an Avalon component?



Avalon Framework (4)

- Component's lifecycle:
  - –Specifies the finite number of component's states.
  - –Specifies the order in which transitions may occur.
  - –Its event (method invocation) always represents a transition to a specific component phase.
  - –Specifies method invocation/event multiplicity.
  - –Leaves containers with a decision which component-supported methods it needs to honor.
  - –Component may choose to implement some/none.
  - –Is specified in the Avalon Framework API.

Avalon Framework (5)

- Three major phases of a component lifecycle:
  - Initialization
    - Startable, LogEnabled, Serviceable, Configurable, Initializable, etc.
  - Active Service
    - Suspendable, Recomposable, Reconfigurable, etc.
  - Destruction
    - Startable (start/stop), Disposable

Note:

Component developer can always rely on the order of events during run time.

Avalon Framework (6)

- Component identification:
  - Components have responsibilities in their systems …
  - … responsibilities are described by work interfaces …
  - … work interfaces become comp. roles in the system.

Consequently:

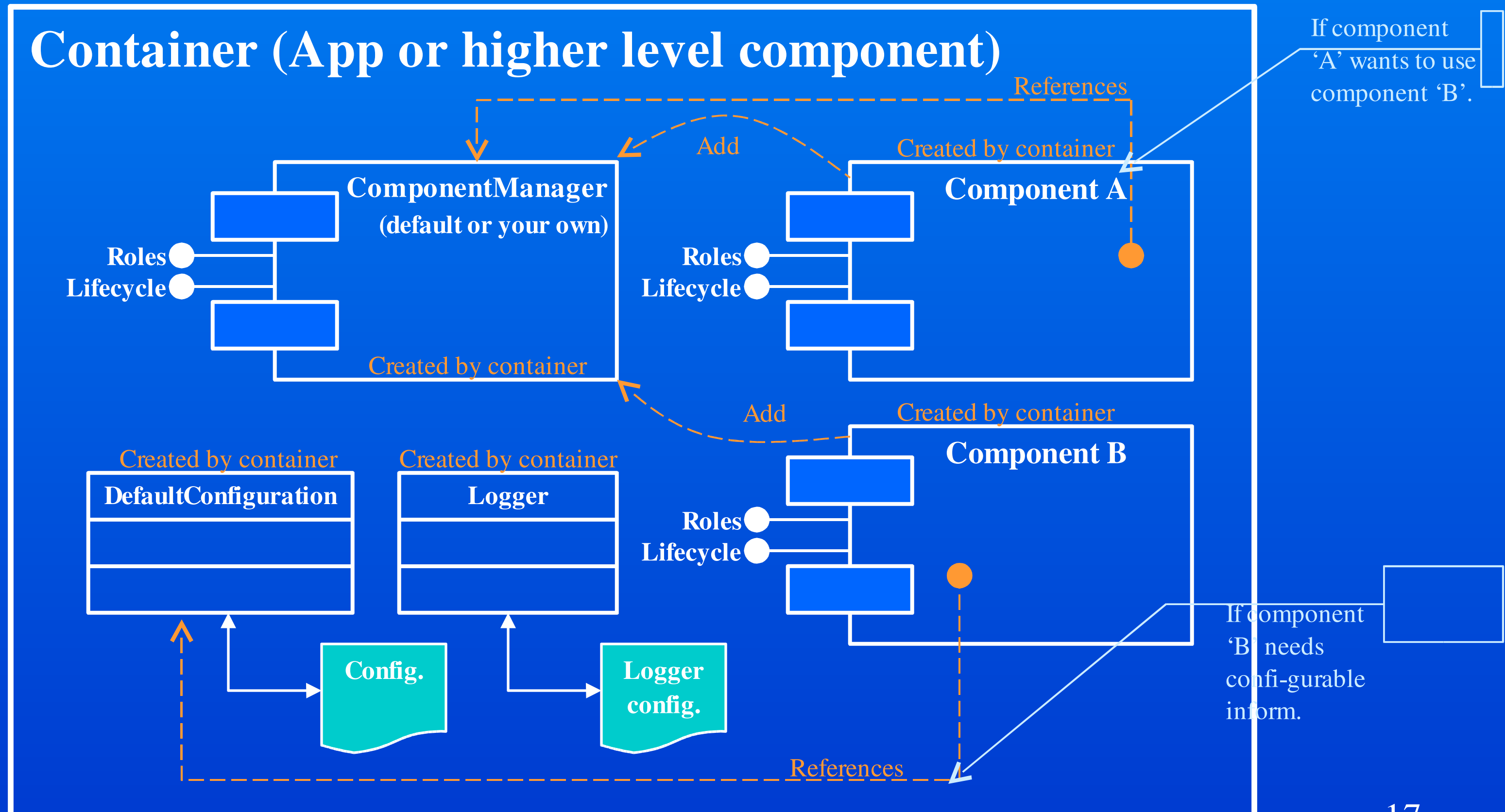
  - Since components are identified by their roles …
  - … roles become components' signatures.

Avalon Framework (7)

- What are component containers?
  - Parent components, services or applications instantiating and controlling child components' lifecycles.
  - Entities that supposed to keep references of their child components for their entire lifetime.
  - They can be as simple as the static 'main' methods called from a command line.
  - According to the IoC, creation and life-cycle methods should follow from a container to a component only.
  - One can control component's work environment using Context and configuration objects as well as Component (now Service) Manager.

Avalon Framework (8)

- Containers managing components manually:



Avalon Framework (9)

- ServiceManager (prev. ComponentManager):
  - Supply component dependencies to other components.
  - Allow simple component/service lookup and discovery.
  - Designed for cases when you have multiple types of components with distinct roles.
  - Requires that you release any component for which you have obtained a reference (SM needs full control over components). There may be situations when you don't!
  - SM can be passed to a child component but only with component roles that this child requires (remember, a child component can NOT talk directly to its parent).

  Note:

  ServiceManager IS NOT ServiceSelector.

Avalon Framework (10)

- ServiceSelector (prev. ComponentSelector):
  - Designed for cases where you have multiple types of Components playing the same role in your system.
  - Can be used by Containers and other Components.
  - Component Selectors are Components themselves.
  - A Component can get a Component Selector from ServiceManager by its role (Component can receive ServiceManager via Serviceable).
  - A Selectors use arbitrary objects for hints to select one of many Components for a role, i.e. String, Local, etc.

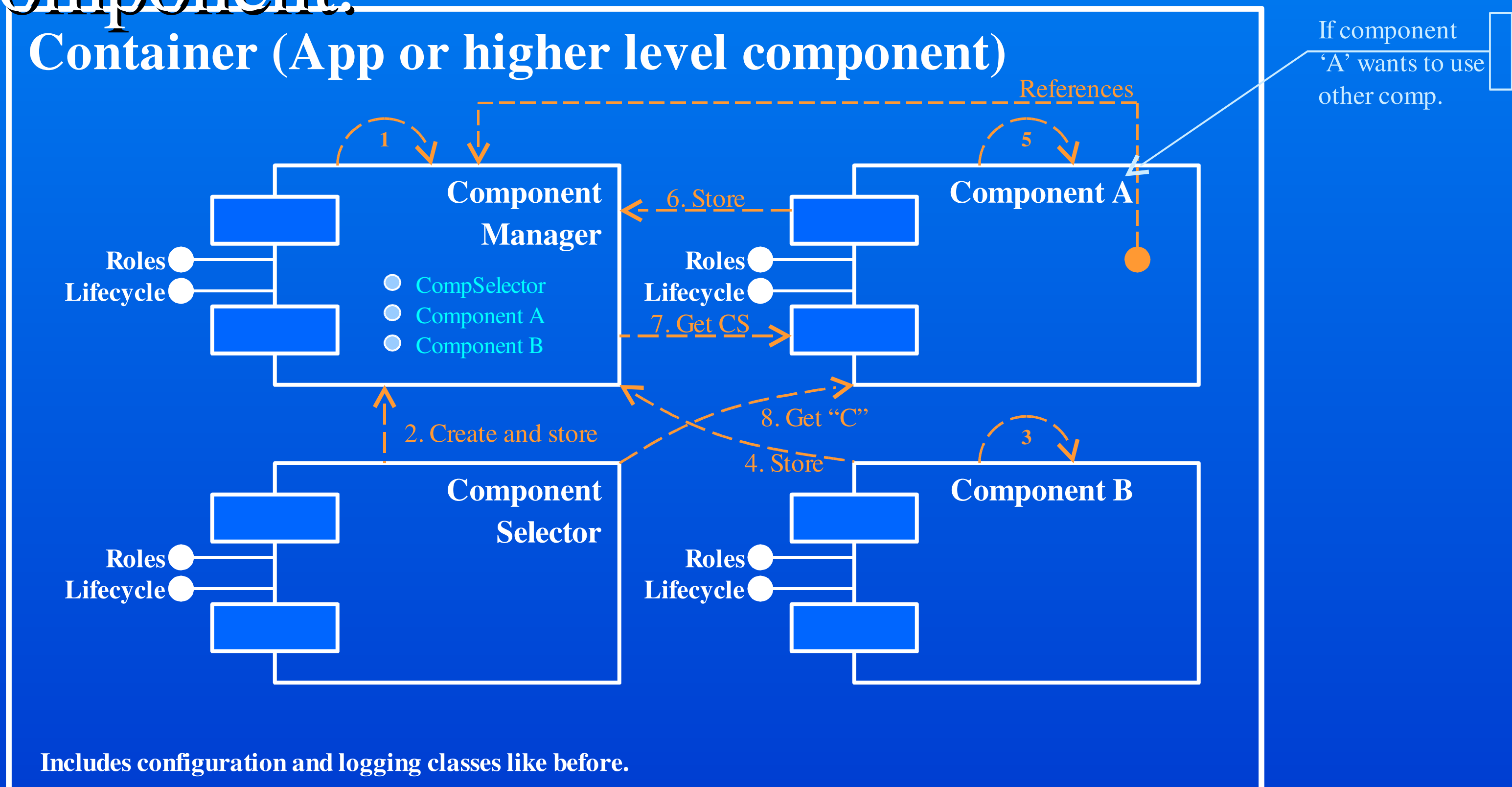
Their use has been discouraged.

Useful suggestion:

If you want to use "ComponentSelector" in your application, follow this naming convention: take the role name assigned to your component and append word "Selector". This will create a name such as: "org.apache.myserver.MyCompNameSelector".

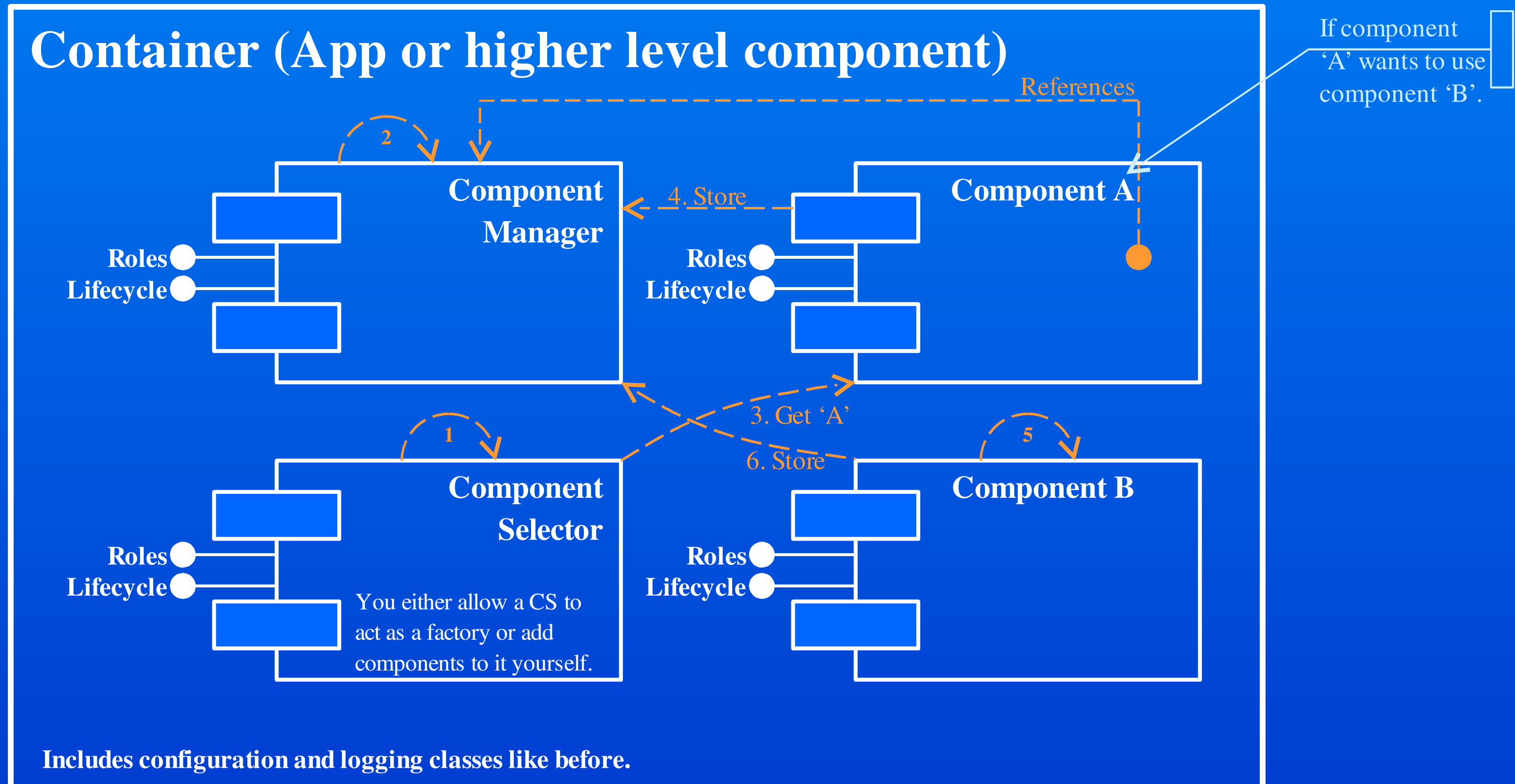
Avalon Framework (11)

- Well, ComponentSelector is just a Component:



Avalon Framework (12)

- Alternative manual component management:

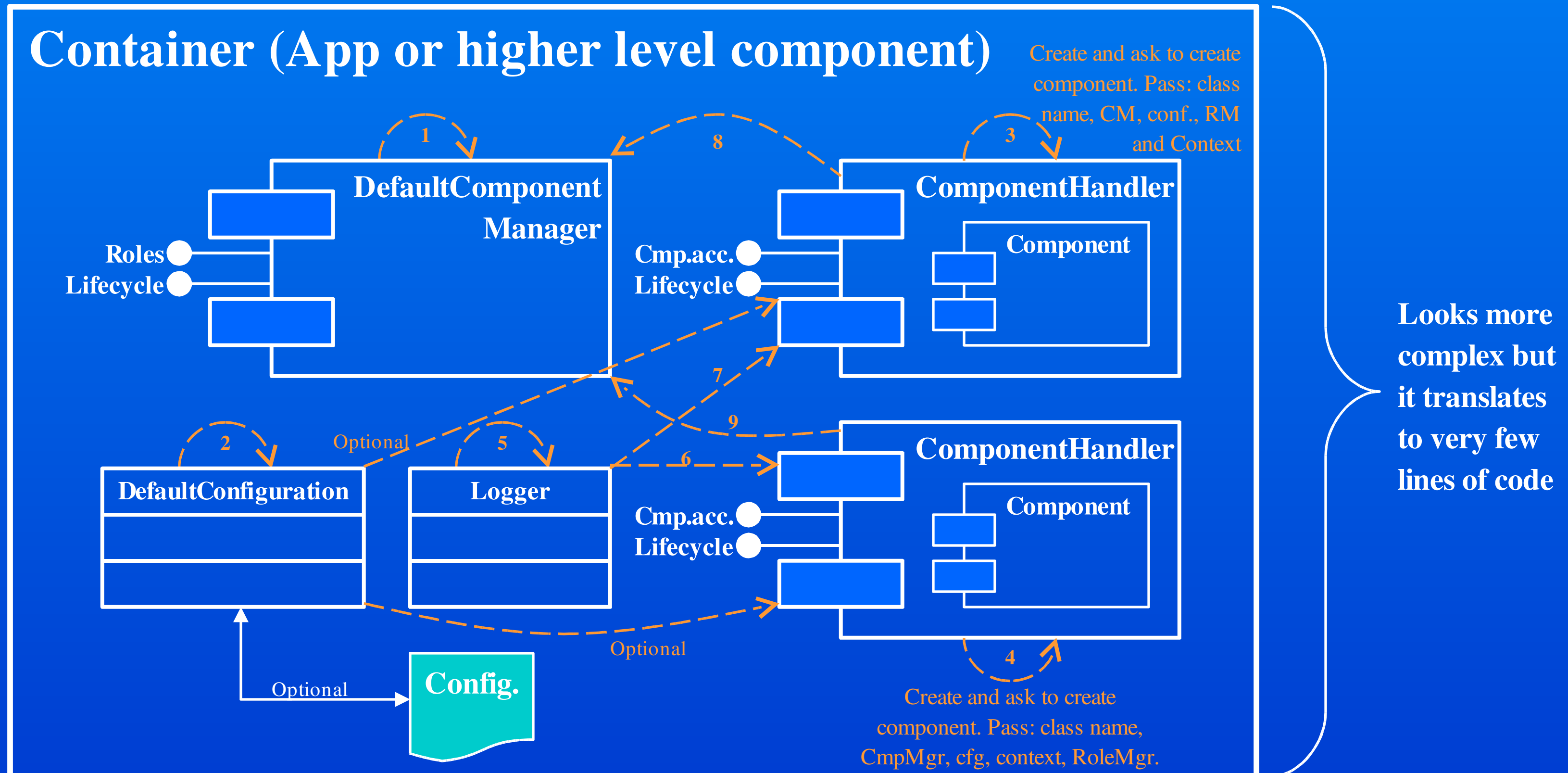


Excalibur Comp. Library (1)

- Set of utility projects used by other container facilities (comp., services & applications).
- Coarse-grained, ready to use components for:
  - Component lifecycle management.
  - Container configuration management.
  - Data source management & internationalization.
  - Asynchronous event handling.
  - Integrating logging & resource monitoring.
  - Thread-safe resource pooling.
  - XML utilities/wrappers & other.

Excalibur Comp. Library (2)

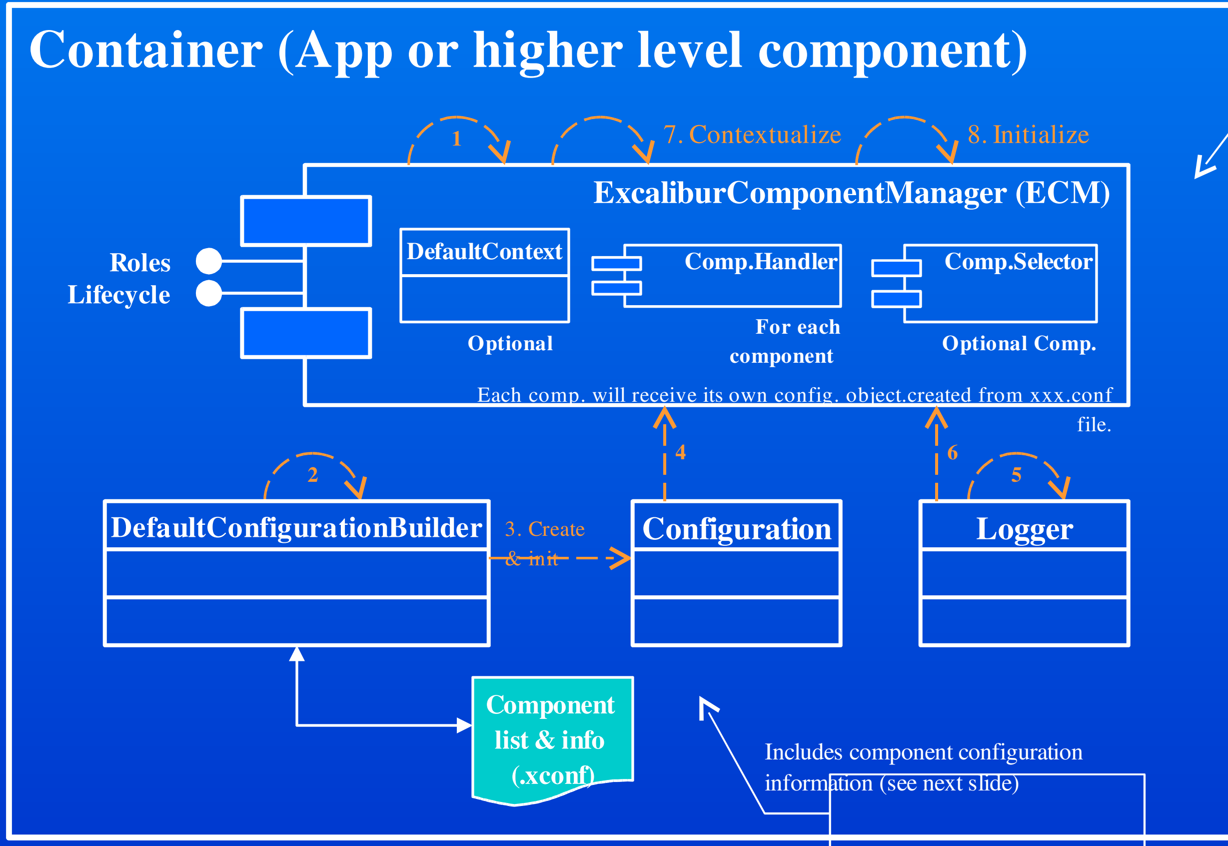
- Low-level component mgmt with handlers:



Note: setting up ComponentHandler is performed through its constructor.

Excalibur Comp. Library (3)

- Declarative component management:
  - Case where the “RoleManager” component is not used.



Use of ECM has been recently discouraged since it will be soon deprecated.

Excalibur Comp. Library (4)

- ECM configuration file:

```
<my-system>
 <component  role="org.apache.avalon.excalibur.datasource.DataSourceComponentSelector"
                class="org.apache.avalon.excalibur.component.ExcaliburComponentSelector">
   <component-instance name="documents"
class="org.apache.avalon.excalibur.datasource.JdbcDataSource">
      <pool-controller min="5" max="10"/>
      <auto-commit>false</auto-commit>
      <driver>org.gjt.mm.mysql.Driver</driver>
      <dburl>jdbc:mysql:localhost/mydb</dburl>
      <user>test</user>
      <password>test</password>
   </component-instance>
   <component-instance name="security"  class="org.apache.avalon.excalibur.datasource.JdbcDataSource">
      <pool-controller min="5" max="10"/>
      ………
   </component-instance>
 </component>
 <component  role="org.apache.bizserver.docs.DocumentRepository"
                class="org.apache.bizserver.docs.DatabaseDocumentRepository">
   <dbpool>documents</dbpool>
 </component>
</my-system>
```

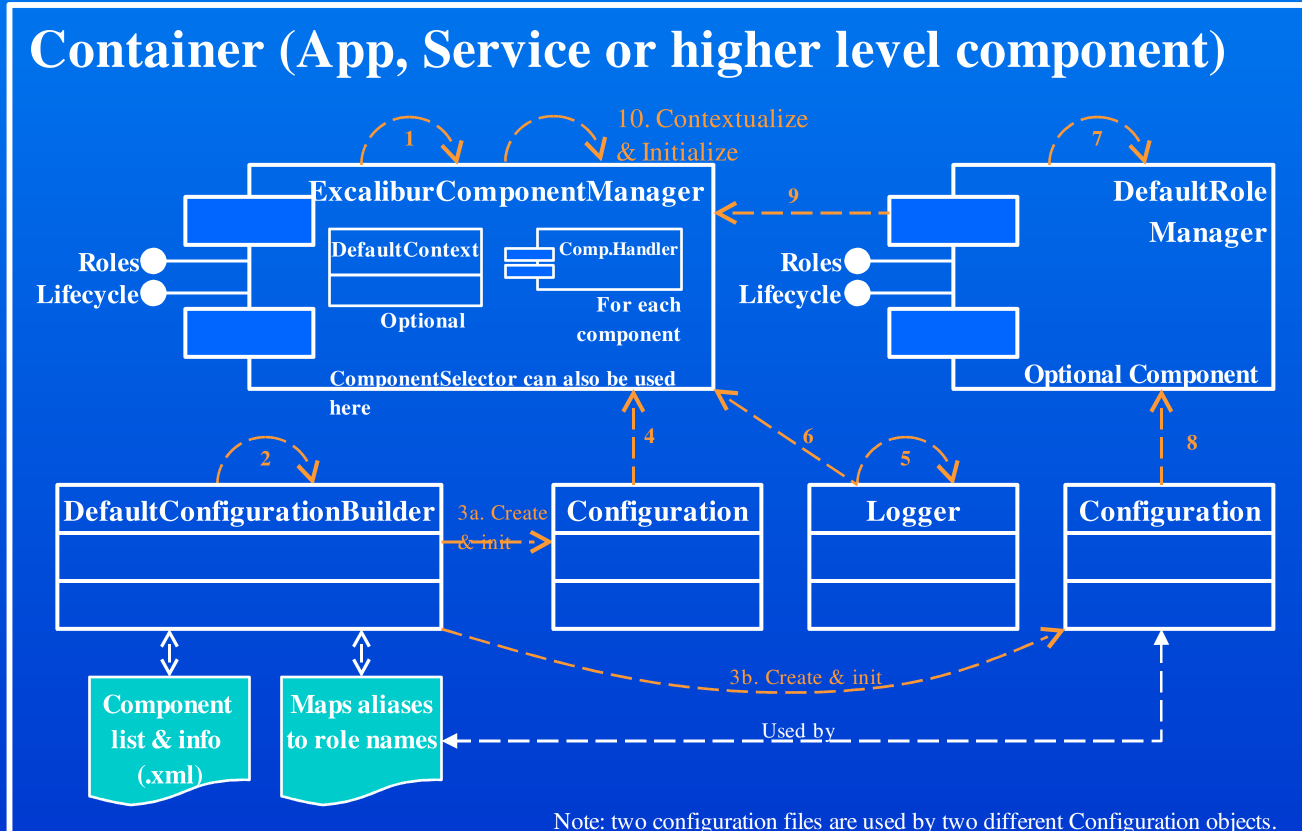
Child components for the ComponentSelcetor.

Component’s pseudo-name/alias that can be used for CS lookups.

myComponents.xconf

Excalibur Comp. Library (5)

- Declarative Component management:
  - Using "RoleManager" Component.



Even fewer lines of code; however, RM is being **deprecated**.

Excalibur Comp. Library (6)

- Managing configuration with aliases.

```
<role-list>
    <role
name="org.apache.avalon.excalibur.datasource.DataSourceComponentSelector"
shorthand="datasources"
              default-
class="org.apache.avalon.excalibur.component.ExcaliburComponentSelector">
        <hint shorthand="jdbc"
class="org.apache.avalon.excalibur.datasource.JdbcDataSourceComponent"/>
        <hint shorthand="j2ee"
class="org.apache.avalon.excalibur.datasource.J2eeDataSourceComponent"/>
    </role>
    <role name="org.apache.bizserver.docs.DocumentRepository"
shorthand="repository" default-
             class="org.apache.bizserver.docs.DatabaseDocumentRepository"/>
</role-list>
```

myapp.roles

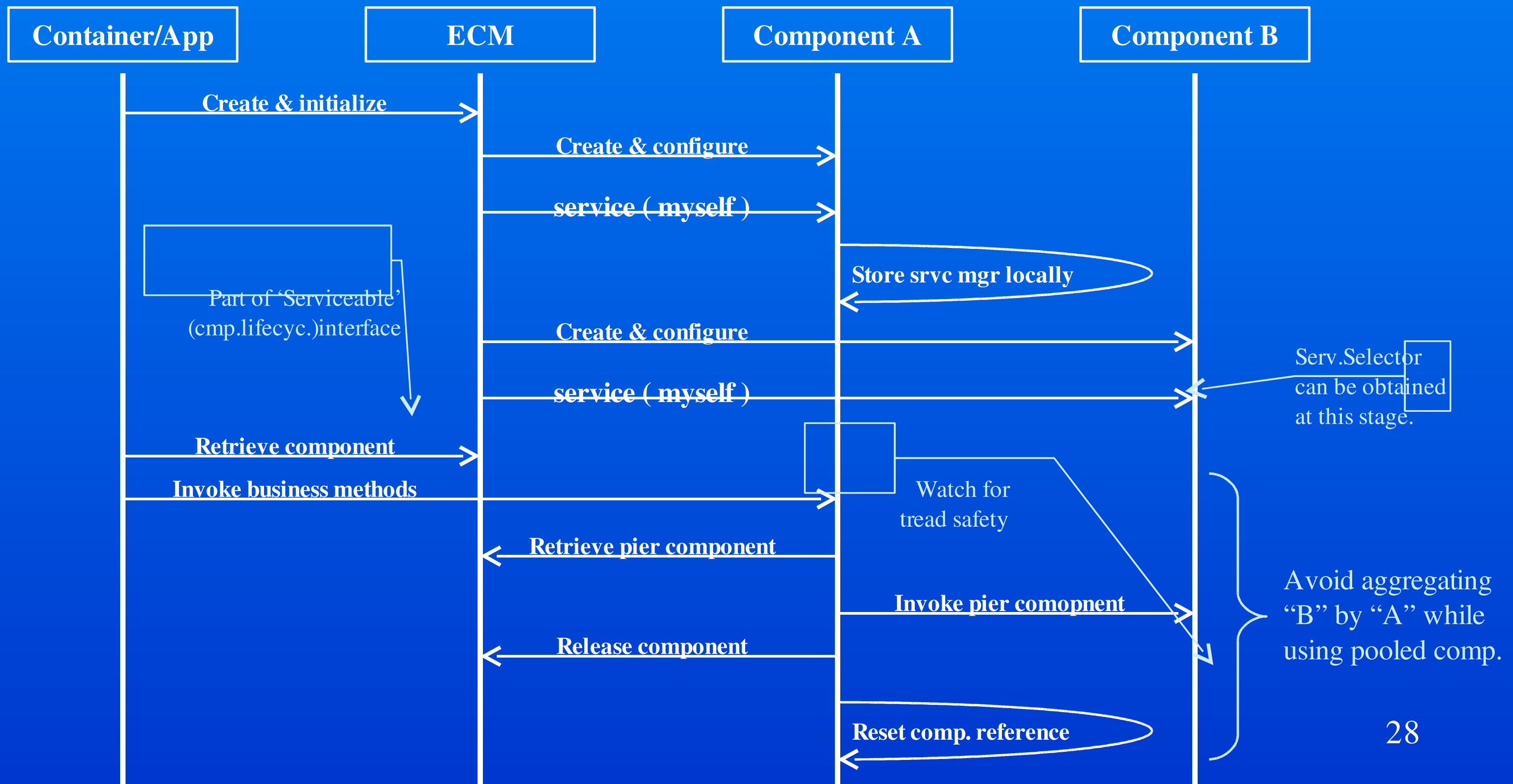
```
<my-system>
  <datasources>
      <jdbc name="documents">
         <pool-controller min="5" max="10"/>
         <dburl>jdbc:mysql:localhost/mydb</dburl>
         ............
      </jdbc>
      <jdbc name="security">
         ..........
      </jdbc>
  </datasources>
  <repository>
      <dbpool>documents</dbpool>
  </repository>
</my-system>
```

myapp.xconf

Lookup performed by the ExcaliburComponentManager

Excalibur Comp. Library (7)

- Using the component management infrastructure.
  - Letting components use their piers through ECM:



Excalibur Comp. Library (8)

- Fully re-usable Excalibur components & APIs:
  - Command line arguments processor (CLI).
  - Collection utilities (Java collections enhancements).
  - Component life-cycle management (ECM).
  - Logging utility wrapper/LogKitManager (depricated).
  - JDBC data source wrappers (DataSourceComonent).
  - I/O specific utilities (i.e. FileFiler).
  - Component pool implementations (i.e. DefaultPool).
  - Multithreading assisting utilities (Lock, Event, etc.).
  - … and much more.

What is Avalon Cornerstone?

- Reusable library of higher level, component-like entities called Blocks.
- One or more components per Block providing a complete solution or utility.
- Implement Services that other server application Blocks can use and depend on.
- Cornerstone reusable services include:
  - Connection and socket management.
  - Principal/role management.
  - Scheduling and others.

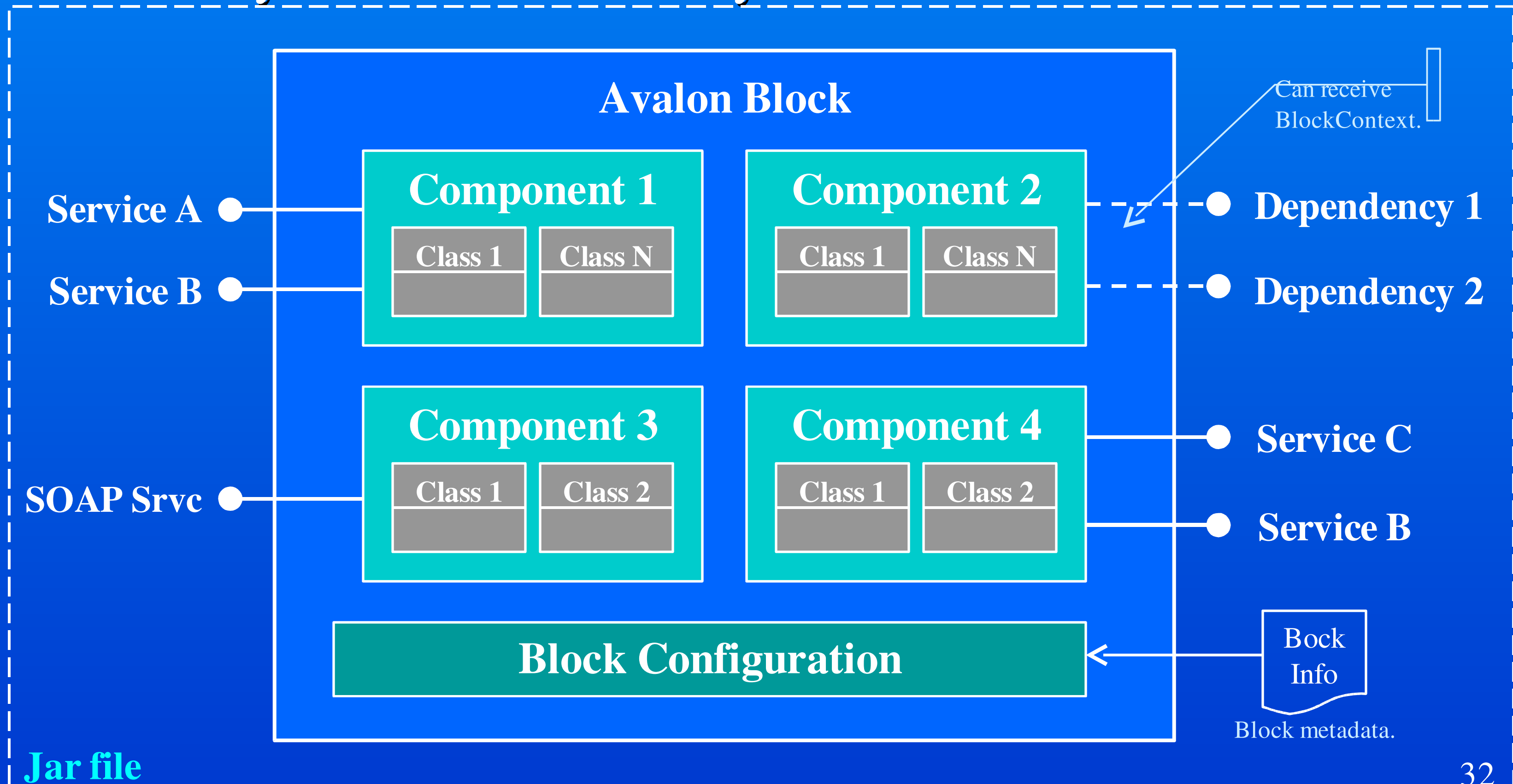
Note: there is a very little difference between a Block and a Component; well, one uses Context the other BlockContext (BlockContext are used by Phoenix only).

Avalon Cornerstone (1)

- Properties of Avalon Blocks:
  - Provides a Service to other Blocks using an interface.
  - Uses meta-data to expose its Services to others.
  - Uses meta-data to specify its own dependencies.
  - Implements Services using component(s) or its internal class(es), where the top-level class implements a Service.
  - Is a versioned entity, as specified in its meta-data.
- Properties of Avalon Services:
  - Specify how other Blocks can utilize its services.
  - Note: a need for Block interface extending a Component interface has been removed from Avalon.

Avalon Cornerstone (2)

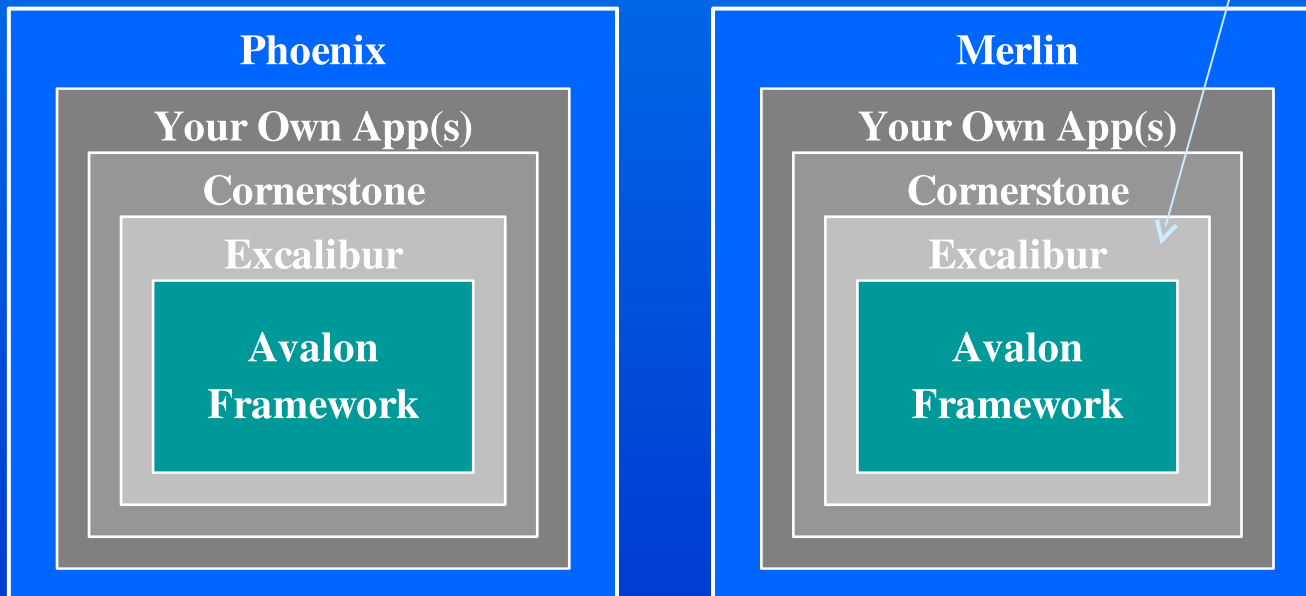
- Blocks are core of the SOA.
  - They are treated differently in Phoenix & Merlin!



Block clients can see only its implemented Services A, B, C, and D.

Avalon Cornerstone (3)

- Will work with Phoenix and Merlin servers!
  - Provide natural transition path for your code.
  - Ready (?) for any future kernel architectures.

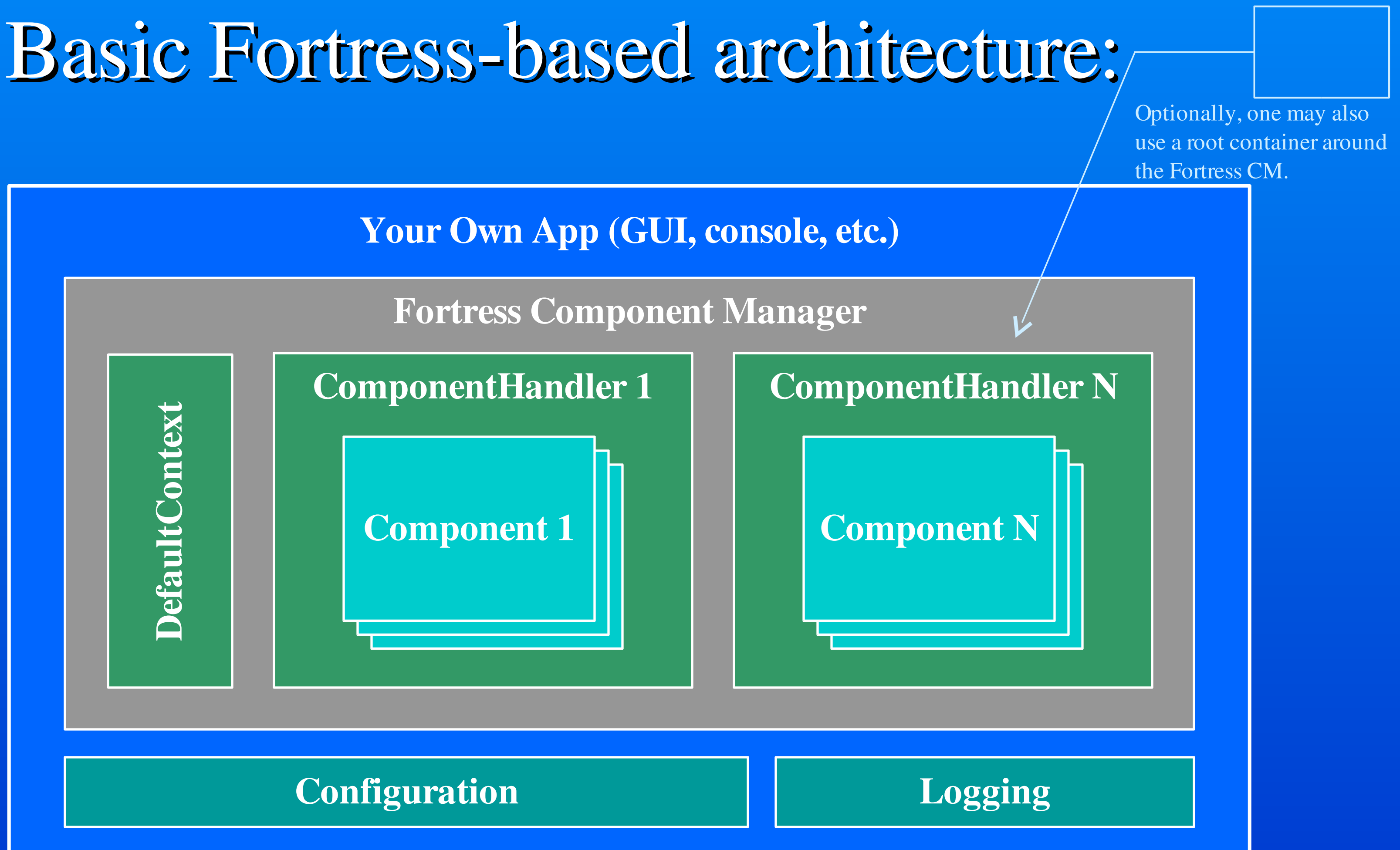


Avalon Fortress (1)

- Fortress container replacing deprecated ECM:
  - Provides asynchronous component management.
    - Through the use of tightly-controlled background threads.
  - Alleviates some of the comp. maintenance difficulties.
    - Hard-coded comp. Roles replaced with JavaDoc tags.
    - Hand-coded metadata files replaced by ant-generated.
  - Integrated with Instrumentation package.
    - GUI view of the system health inspection at run-time.
    - Monitor component instances per component handler.
  - Easier to work with in your GUI, embedded, etc. app.
    - This includes servlet-like engine environments!
  - Provides support for the life-cycle extensions.

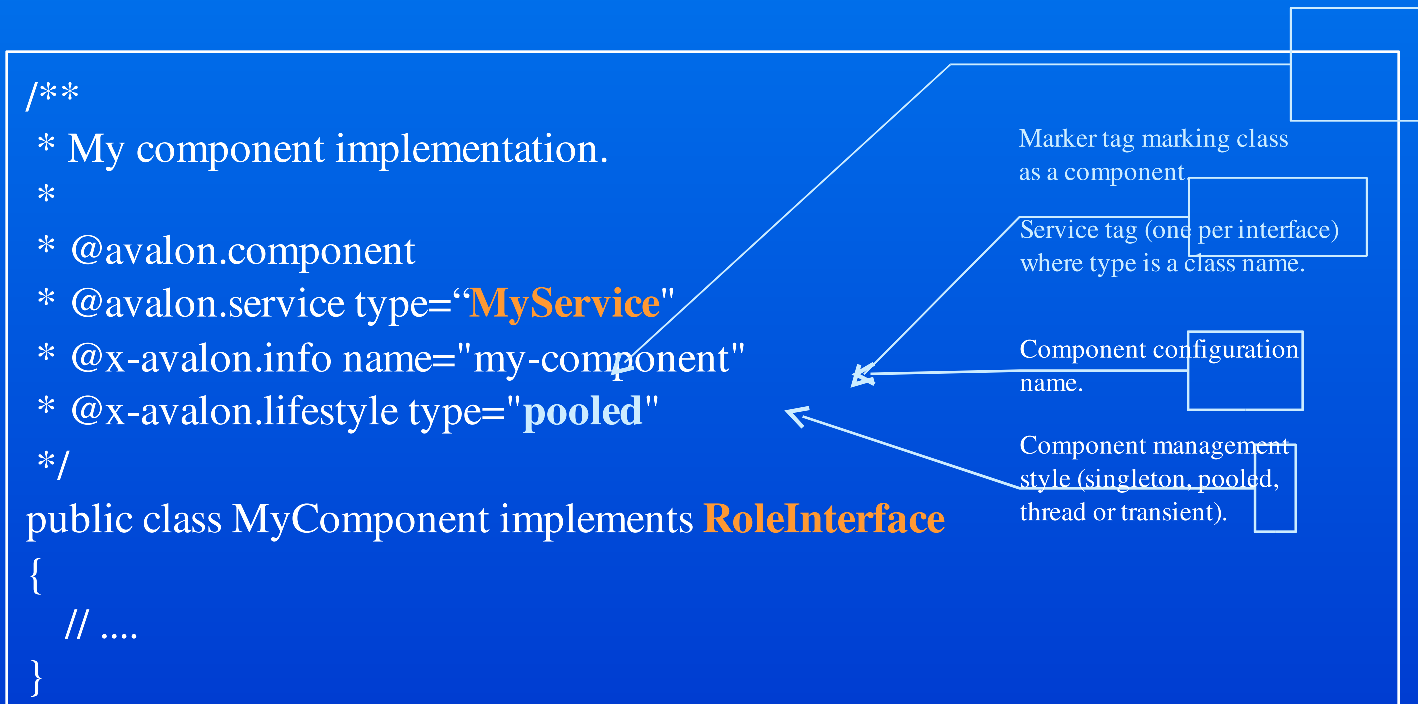
Avalon Fortress (2)

- Basic Fortress-based architecture:



Avalon Fortress (3)

- Configuring components via meta info tags:
  - Allows for comp. circular dependency discovery.
  - Meta information closely tight to its source.



Avalon Fortress (4)

- Specifying dependencies via meta info tags:
  - Component needs to implement “Serviceable”.
  - Fortress does the rest of the heavy-duty work.
  - It provides ant-task to collect all the metadata info.

```
/**
 * Get all the dependencies.
 *
 * @avalon.dependency type=“MyService"
 * @avalon.dependency type="OtherService"
 */
public void service( ServiceManager manager )
{
   // ...
}
```

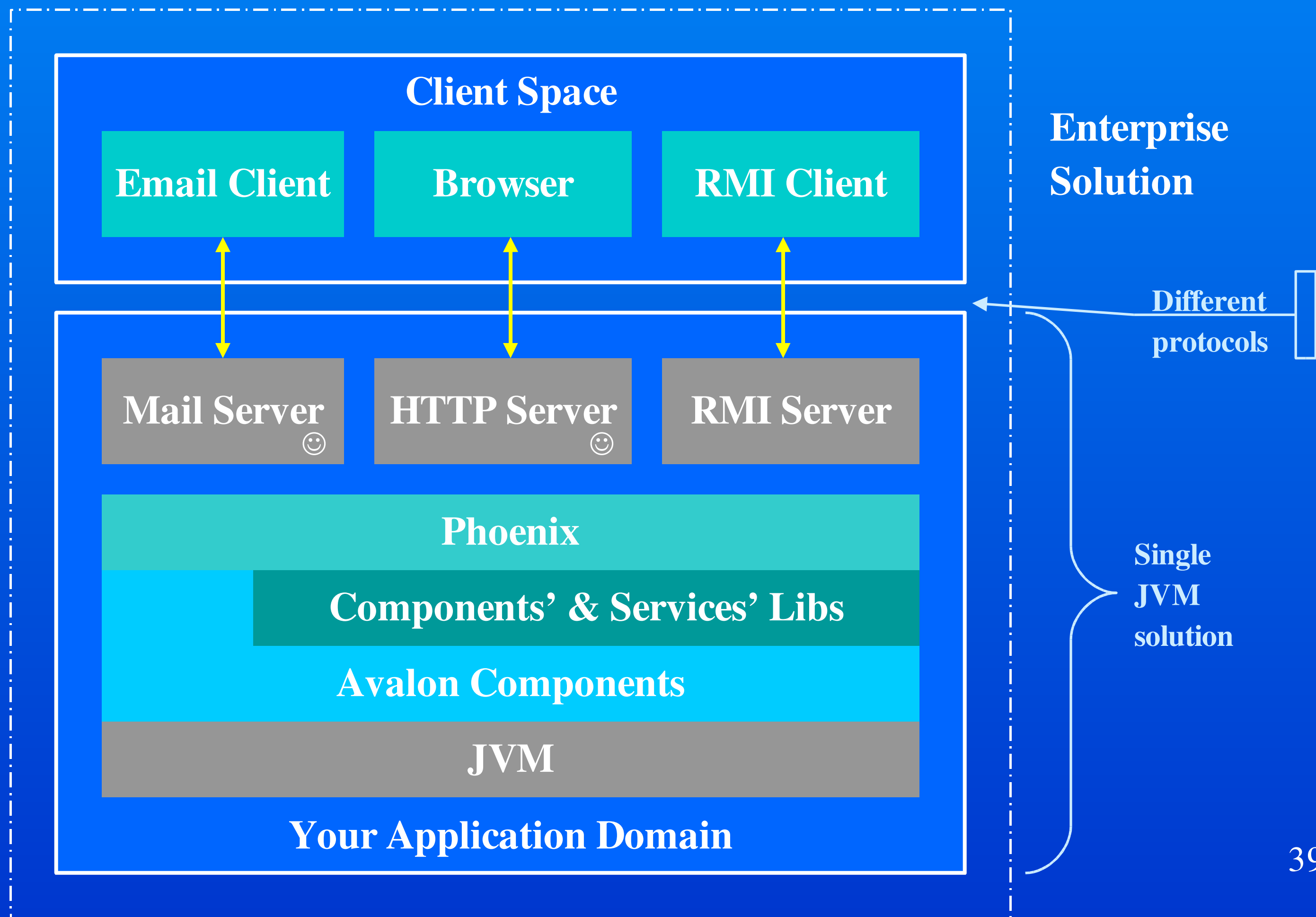
As defined in a class header of another component.

What Is Phoenix?

- A high-level container for componentized, stand-alone server applications.
- An environment for running multiple server applications within the same JVM.
- Configurable environment for rapid application assembly and deployment.
- Deployment platform with a high level of application isolation and easy management.
- An extensible & customizable micro-kernel.
- Container for a set of Avalon components.

Phoenix In An Enterprise

- Suitable for multiple deployments in one JVM.

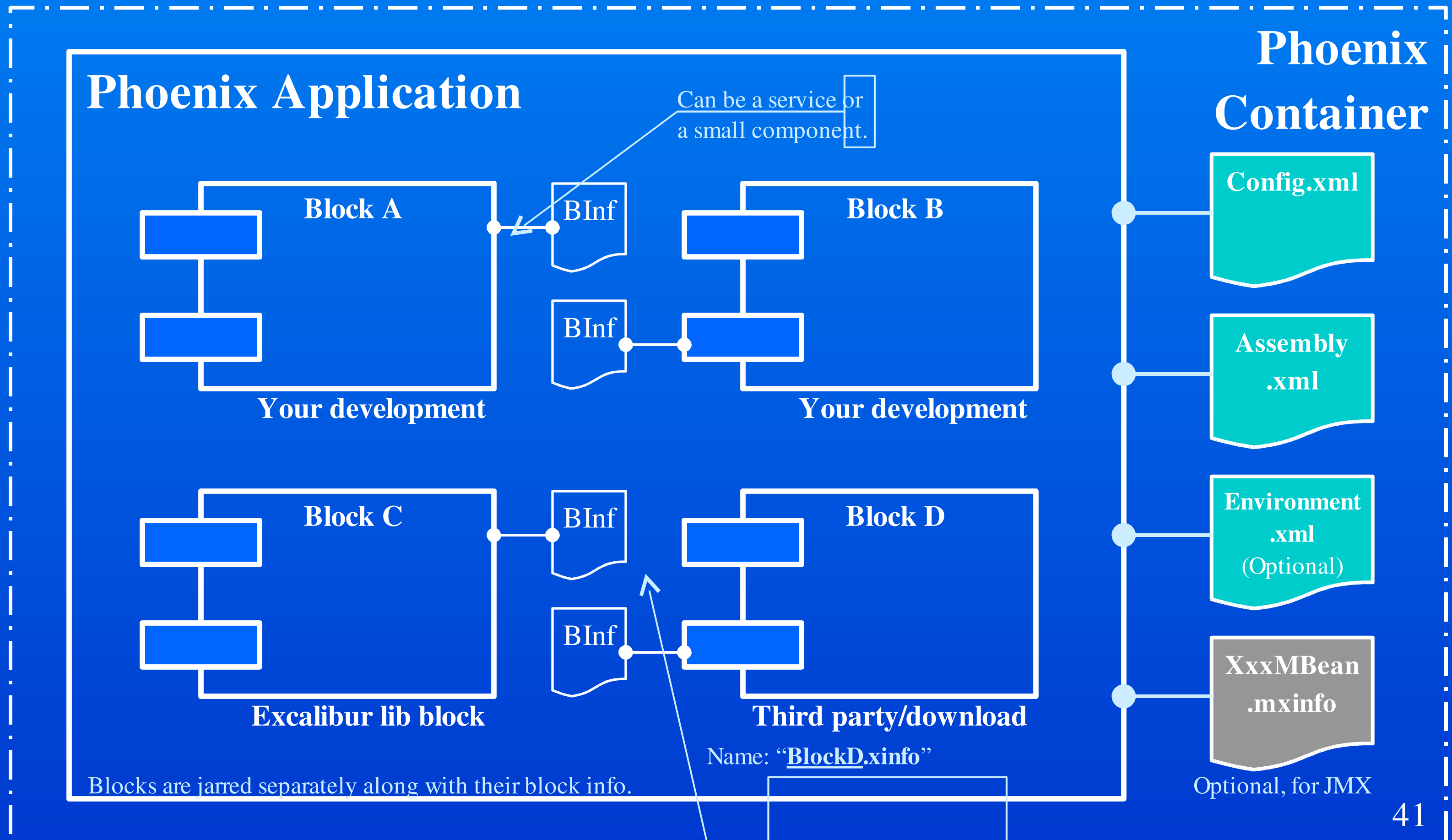


Phoenix Kernel (1)

- Provides facilities to manage server application environment, including:
  - Log management, class-loading, thread management & security.
  - Uses XML configuration files for rapid (re)assembly.
- Provides native support for use in:
  - Command-line, stand-alone applications.
  - Unix daemon & Windows services (if service wrapper used).
  - Other environments, i.e. servlet containers.
- Written as an extensible micro-kernel:
  - Plug and/or remove services and functionality easily.
  - Customize existing functionality at the administrative level.
  - Leverage proven Avalon Framework for compatibility & reuse.

Phoenix Kernel (2)

- Basic application server architecture:

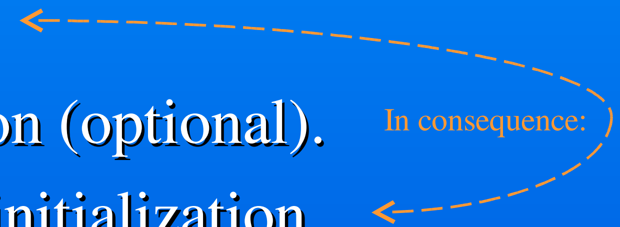


Phoenix Kernel (3)

- Block properties:
  - Follows a standard Avalon component format.
  - Can receive an instance of BlockContext (ext. Context).
  - Can implement any of the Avalon lifecycle methods…
  - … except to “suspend” and “resume”.
  - Accesses pier-blocks through Serviceable interface.
  - Configured through the BlockInfo file.
  - Packaged with its BlockInfo in a jar file.
  - Can be used as a wrapper for non-Avalon code/utility.
  - Managed by the Phoenix (or other) server kernel.
  - Needs to be documented in order to be re-useable!
  - Should be thread-safe to be used in Phoenix.

Phoenix Kernel (4)

- Block properties (cont.):
  - Empty public constructor.
  - Setter methods for its configuration (optional).
  - “initialize()” method for setup & initialization.
  - Should avoid Singleton design pattern.
  - Can be tested independently of the Phoenix.
    - » Use the main-method execution wrapper.
  - … and most importantly:



Provides truly plug-able solution with help of the separation of interface and implementation supported by the “as sembly.xml”.

Phoenix Kernel (5)

- Block information file.

Can be generated using ant task.

```
<?xml version="1.0"?>
<blockinfo>
    <block>
     <name>Fully qualified name of the service interface</name>
     <version>1.2.3</version>
  </block>
   <services>
     <service name="com.biz.cornerstone.someservices.CertifRequest" version="2.1.3" />
  </services>
    <dependencies>
     <dependency>
        <role>com.biz.cornerstone.someservices.Authorizer</role>
        <service name="com.biz.cornerstone.someservice.Authorizer" version="1.2"/>
     </dependency>
     <dependency>
        <!-- Role defaults to the name of service. The service version defaults to "1.0" -->
        <service name="com.biz.cornerstone.someservice.RoleMapper"/>
     </dependency>
  </dependencies>
</blockinfo>
```

Name (optional, rarely used) and block version.

Services provided by the block: this is your work interface you want to expose to other users.

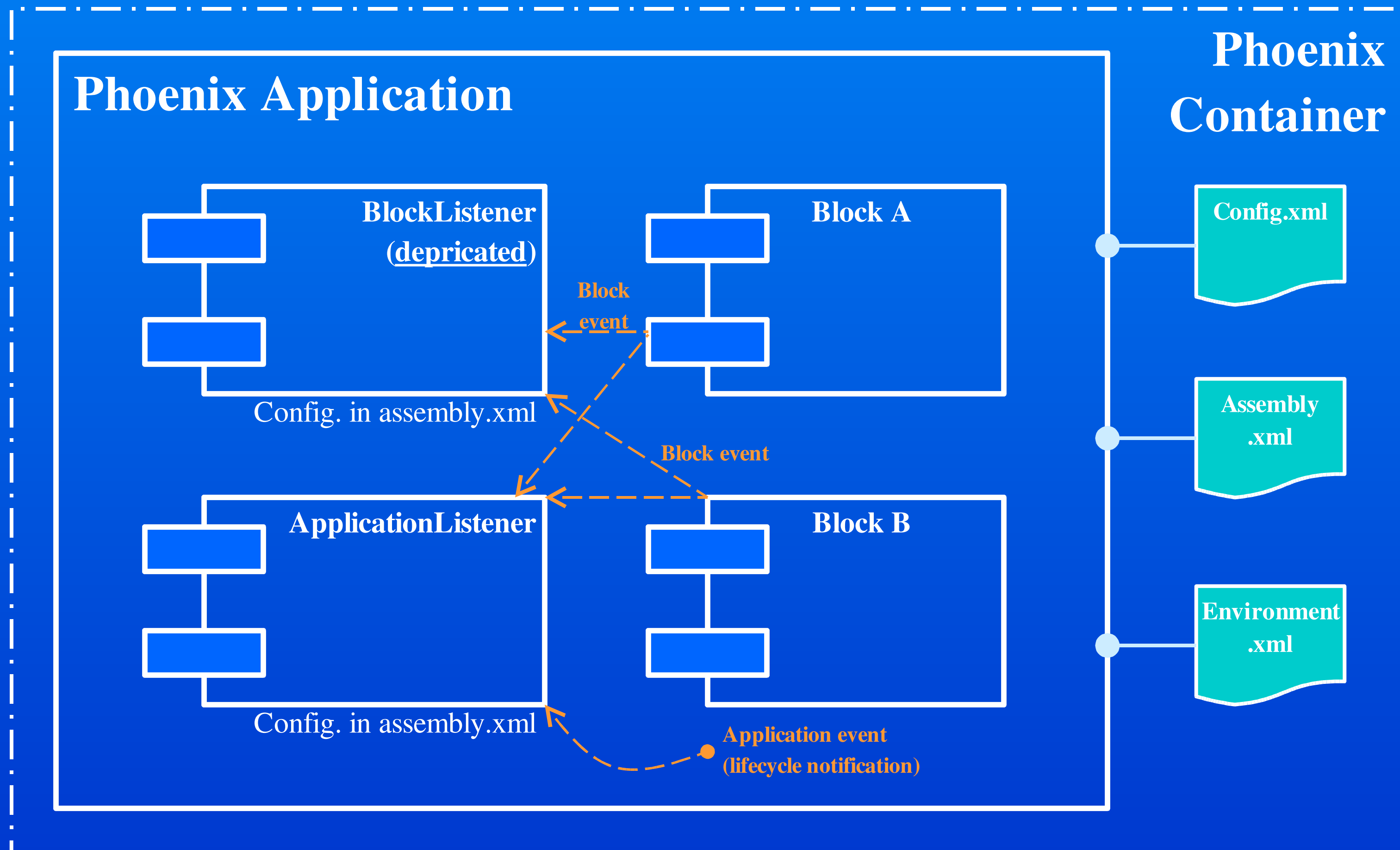
Services used by the block: other blocks work interface. Used in your code to lookup services! (can be alias instead, i.e. "AUTHORIZER").

Optional management access point goes here.

CertifRequestServer.xinfo

Phoenix Kernel (6)

- Application architecture utilizing listeners:



Phoenix Kernel (7)

- Block Listener (deprecated):
  - They are Blocks themselves.
  - Receive notifications during lifecycle of other blocks.
  - Events include block addition/removal from the app.
  - Configured through the “assembly.xml” file.
- Application Listener:
  - Like above, they are Blocks themselves.
  - Created before any other application Blocks.
  - Receive notifications during lifecycle of application.
  - Receive notifications during lifecycle of app’s blocks.
  - Must implement ApplicationListener interface.
  - Configured through the “assembly.xml” file.

Phoenix Kernel (8)

- Block-based application development process:
  - Select reusable library Blocks your app will rely on.
  - Design and create your own application Blocks.
  - Create a separate jar file for each new Block.
  - Write application “config.xml” file.
  - Write application “assembly.xml” file.
  - Write application “environment.xml” file.
  - Package components and their resources into a ‘sar’ file.

Phoenix Kernel (9)

- "assembly.xml" file.
  - Defines how kernel assemblies a server application.
  - Provides apps' block names and their dependencies.
  - Defines the application listeners.

```
<?xml version="1.0"?>
 <assembly>
  <!-- Certification Request Processor Factory Block -->
  <block class="com.biz.cornerstone.someservice.CertifRequestServer"
name="certification">
     <provide name="someName" role="Role name1 from the .xinfo file"/>
     <provide name="someName" role="Role name2 from the .xinfo file"/>
     <provide name="time" role="Role name from the .xinfo file"/>
  </block>
  <block> class="someClass" name="someName"</block>
    <!-- More assembly information/blocks go here. -->
 <proxy>Asks kernel to provide block wrapper.</proxy>
 <listener name="myListener" class="fully qualified name">
 </listener>
 </assembly>
```

Lists blocks in your application.

Block's <u>implementation:</u> fully qualified class name (references block info file & impl. class). There <u>must</u> be "**CertifRequestServer.xinfo**" file.

Linking 2 blocks

Block name used in the configuration file & referred to locally.

References a dependent block/service by its name: connecting blocks together.

Defines the <u>application</u> listener (name referenced in the conf. file).

Phoenix Kernel (10)

- “config.xml” file.
  - Provides block configuration data.
  - Some blocks may not require any.
  - If used, will be block-specific.

```
<?xml version="1.0"?>
  <config>
     <certification>
       <!-- ...configuration data here... -->
     </certification>
     <someName>
       <param1>param1-value</param1>
       <an-integer>2</an-integer>
       ...
     </someName>
  </config>
```

Name corresponding to a name of a block, as specified in the “assembly.xml” file.

Block-specific parameter format wrapped by the Configuration object (remember, all bets are opened here).

Phoenix Kernel (11)

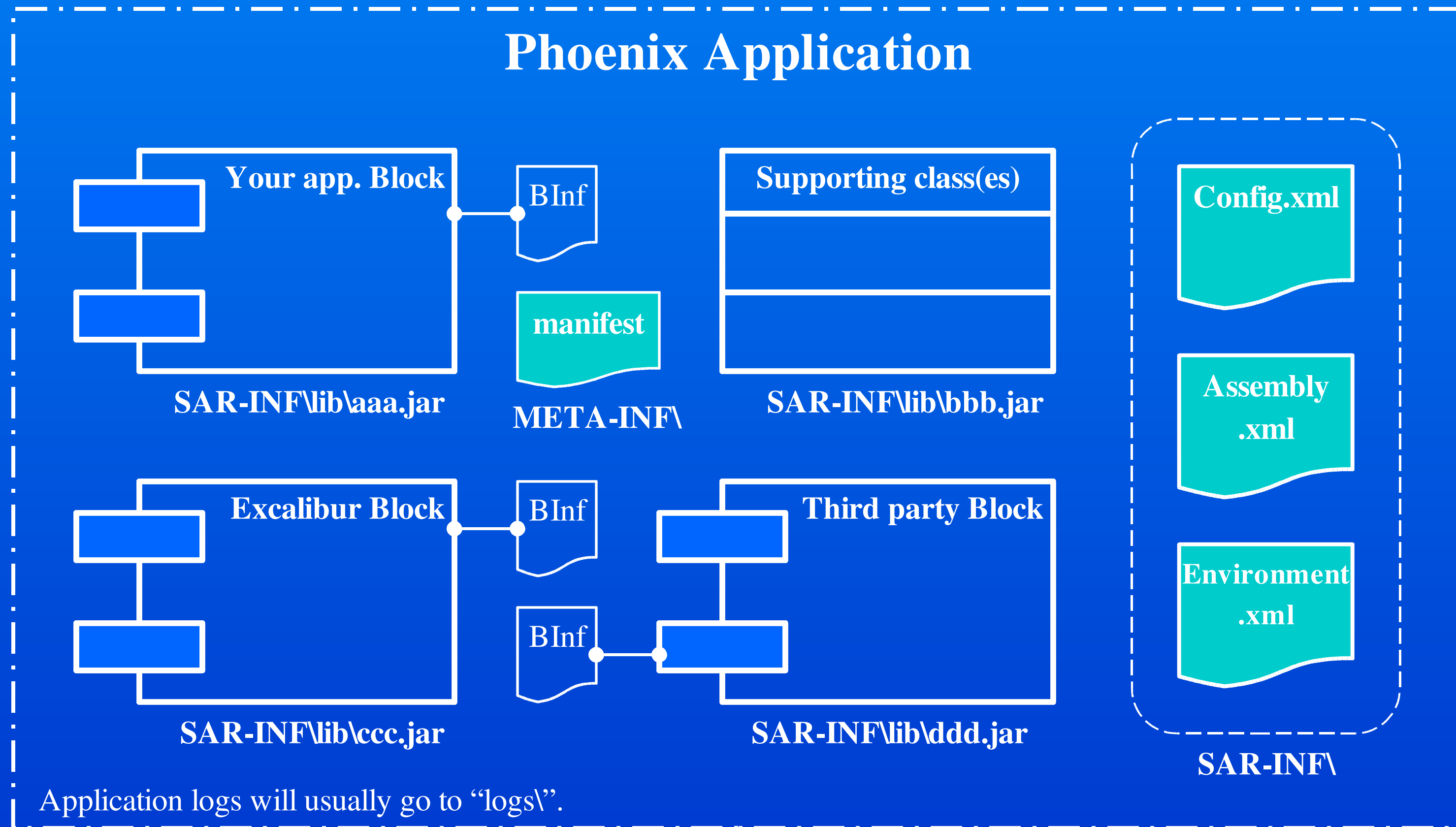
- “environment.xml” file (optional).
  - Provide server-wide settings including:
    - » Code-based security management & log management settings.

```
<?xml version="1.0"?>
  <environment>
     <logs>
       <category name="" target="default" priority="DEBUG" />
       <category name="myAuthorizer" target="myAuthorizer-target" priority="DEBUG" />
         <log-target name="default" location="/logs/default.log" />
       <log-target name="myAuthorizer-target"  location="/logs/authorizer.log" />
   </logs>
     <policy>
       <keystore name="foo-keystore" location="sar:/conf/keystore" type="JKS" />
         <grant code-base="file:${app.home}${/}some-dir${/}*" key-store="foo-keystore">
           <permission class="java.io.FilePermission" target="${/}tmp${/}*" action="read,write“ />
       </grant>
         <grant signed-by="Bob" code-base="sar:/SAR-INF/lib/*" key-store="foo-keystore">
           <permission class="java.io.FilePermission" target="${/}tmp${/}*" action="read,write" />
       </grant>
   </policy>
  </environment>
```

Note: thread pooling configuration deprecated.

Phoenix Kernel (12)

- Application deployment directories.
  - Note: we use dedicated launcher.



A single distribution file: …\phoenix-4.0.3\dist\apps\myApplication.sar

Deploying Phoenix Application

- Cold deployment:
  - Drop your application's "sar" file into the server's *apps/* directory.
  - Restart Phoenix application server.
- Hot deployment:
  - None!
  - Since Phoenix development has been halted, don't expect this to be done soon, if ever.

Administering Phoenix Application

- Starting and stopping Phoenix server:
  - Windows: invoke *distrDir\bin\run.bat* to start-up.
  - Windows: *Ctrl-C* for shutdown
  - Unix: invoke *distrDir/bin/run.sh* to start-up.
  - Unix: invoke *distrDir/bin/phoenix.sh start(or stop)*.
  - All: un-deploy all Phoenix applications to stop.
  - Use Java Service Wrapper to:
    - » Install Phoenix as an NT service.
    - » Restart application if crashed or frozen.
- Take advantage of the JMX.
  - Create an MBean interface (use WebServerMBean).
  - Implement WebServerMBean interface in your Block.
  - Connect to http://localhost:8082.

Phoenix Development Summary

- Use Avalon components for your low-level logic:
  - Select and/or create new components for your application.
  - Add Service- and/or component Manager only when needed.
  - Create roles file if Role Manager used ("my Comp.roles").
  - Create component's configuration file ("myComp.xconf").
  - Jar each component and its associates in a separate file.

Optional

- Expose high-level logic through Blocks/Services:
  - Create new wrapper-Blocks around existing components.
  - Add Blocks with their own business logic with/without components.
  - Create "config.xml", "as sembly.xml" and "en vironment.xml" files.
  - Use BlokContext for pushing data down from your Containers.
  - Jar each application Block and its associates in a separate file.

Continued on the next page:

Phoenix Development Summary

- Add ApplicationListener object when necessary:
  - Create listener class implementing ApplicationListener interface.
  - Update your application configuration files.
  - Note: use it if your application requires high-level event notification.
- If your project's schedule permits, add JMX:
  - Update your application configuration files.
  - Some coding efforts are necessary: new MBean, etc.
- Create a single sar file for all Components & Blocks.
- Deploy the sar file into the Phoenix application server.

Merlin: 2–Minute Intro

- Next generation container, a.k.a. Service Management Platform.
- Replaces Phoenix.
  - Most Phoenix applications will run on Merlin.
- Major features:
  - Composite Component Management.
  - Automated Assembly.
  - Lifestyle Management.
  - Life Cycle Management.

Merlin vs. Phoenix

- Provides simpler environment configuration:
  - One configuration file, as opposed to three.
  - Better suited for embedded servers/applications.
- Supports remote service dependencies.
- Well supported by developers and user groups.
  - Cleaner and well-maintained web documentation.
- Currently under heavy development:
  - Still moving-target development environment.
  - Includes too many unstable/under development features.
  - Somewhat ‘stable’ version released at the end of 2003.

Avalon Best Practices

- Dynamic class loading:
  - Use “this.getClass().getClassLoader().loadClass(String)” …
  - … instead of “Class.forName(String).newInstance()”.
- Use “static” and singletons with caution:
  - They cause unpredictable behavior within multiple class loaders.
- Organize loggers using areas of concern:
  - Avoid class-based approach as it may be too fine-grained.
  - Implement “LogEnabled” interface.
- Componentize concepts but write serviceable solutions:
  - Apply top-down system’s architecture design methodology.
  - Divide your system along the business lines.

Avalon Advantages

- Reasonably solid application framework:
  - Exercised and proven by a number of production releases.
- Freely available source code.
  - One can do his own bug fixes and improvements.
  - Can be used in place of missing documentation.
- User groups and project developers can be valuable source of information & support.
- Free access to the utilities and third-party components that are ready to use.
  - … well, if you can ever match their dependencies. ☺
  - Watch for “to do” syndrome on Cornerstone utilities!!

Avalon Disadvantages

- Still poorly documented application framework:
  - Documentation is often outdated, if ever existing.
  - Documents may occasionally be contradictory to each other.
  - Missing product's big-picture: architecture diagrams, etc.
- Missing production-level software features:
  - Clustering, fail-over & load balancing not even started.
- One ends-up working with a moving target.
- Difficult to make implementation examples working with the correct release version(s).
- Frequently changing Avalon APIs.

??????????