S. BHANU REKHA

ADOBE® CQ5.5 Developer Training

User Training Student Workbook

ADOBE® TRAINING SERVICES

Adobe® CQ5.5 Developer Training Student Workbook

Disclaimer

Adobe reserves the right to revise this publication from time to time, changing content wit

hout notice.

Preface – Formatting Conventions

Style	Description	Example
Cross-reference	Cross-reference to external documents.	See the *Microsoft Manual of Style for Technical Publications*
GUI Item	User interface items.	Click Save.
Keyboard shortcut	Keyboard shortcuts	Press Ctrl+A
Mouse Button	Mouse buttons	Secondary-mouse button (usually the right-mouse button).
Link	Link to anchor-points within the current document and/or external sources.	http://www.day.com
Code	Example of programming code.	`If (weather==sunny) smile;`
User Input	Example of text or commands that you type.	ls *.xml
<Variable User Input>	Example of variable text - you type the actual value needed.	ls <cq-installation-dir>
[Optional Parameter]	An optional parameter.	ls [<options>] [<filename>]
Computer Output	Logging and error messages, computer responses.	The output is ls: cannot access error.log:
Page reference	Reference to a page, file, or component.	Open the *Company* page

When you see this...	It means do this...
Ctrl+A	Hold down the Ctrl key, then press the A key.
Right-click	Press the right-mouse button (or the left-mouse button if your mouse has been configured for left-handed use).
Drag	Hold down the left mouse button while moving the item, then release the mouse button at the new location (or the right mouse button if your mouse has been configured for left-handed use).

Table of Contents

Section 10- CQ5 Development - Advanced Concepts

Appendix A Clone an Author Instance to be a Publish Instance A-1

Section-1 What is CQ 5?

CQ5 is an enterprise-grade content management platform with a wide array of powerful features. With this software people in your organization can:

- Build, author and publish websites, complete with enforcement of corporate design and user access control of editing and publishing rights.
- Define and implement workflows for the creation, editing and publishing of content.
- Manage a repository of digital assets such as images, videos and documents, and integrate these assets into your website.
- Use search queries to find content no matter where it is stored in your organization.
- Set up social collaboration tools like blogs, user groups, and calendars.
- Organize your digital assets and web pages using tagging.

All of these things are done through a rich graphical interface accessible through any modern browser, enabling such desktop-like features as in-place editing of text and graphics, drag and drop of page elements and visual design of workflows.

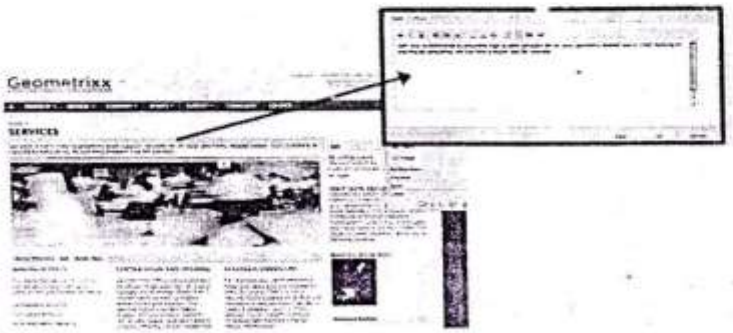
CQ5 Platform

CQ5 is implemented as a Java web application. It runs in any server that supports the Java Servlet API 2.4 (or higher). CQ5 comes pre-configured with its own built-in servlet engine, CQSE, but can also be installed in any compatible third-party application server.

Since the system can be accessed from any computer with a modern web browser, no installation of client software is required, so even large installations with thousands of users can be rolled out and upgraded easily.

The CQ5 User Interfaces

The web-based authoring and administrative interfaces use AJAX to enable a desktop-like user experience. For example, when editing content on a website, authors can drag and drop elements like text paragraph and images right onto the page and immediately see how their changes affect the appearance of the page.

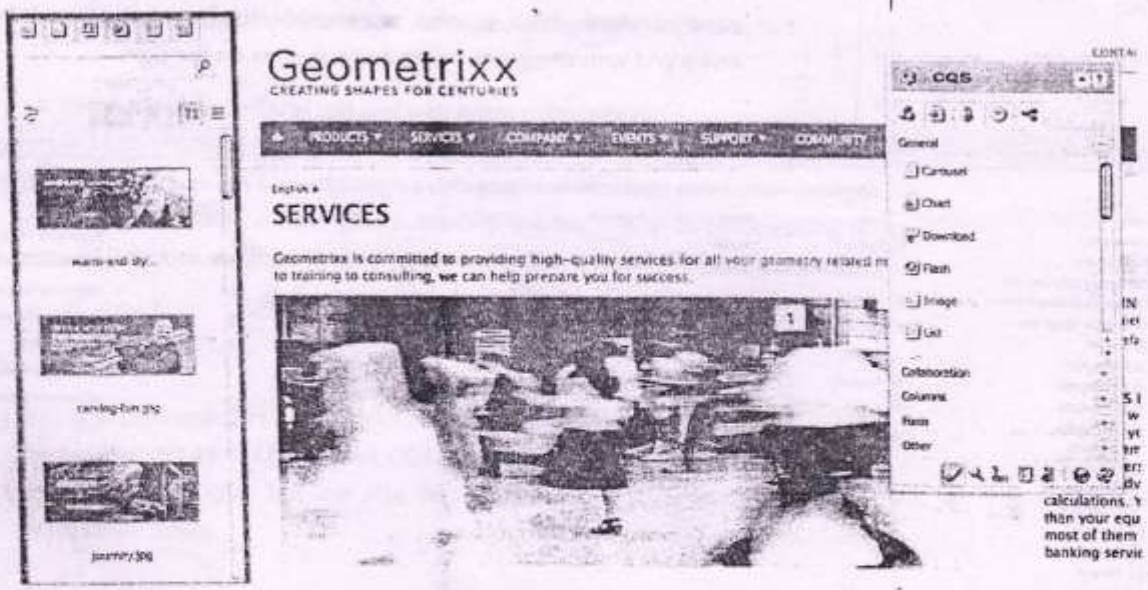


The CQ5 user interface combines the advantages of a web interface with the fluidity and responsiveness usually associated with desktop applications. AJAX (Asynchronous Javascript and XML) technology is used to support interface features such as:

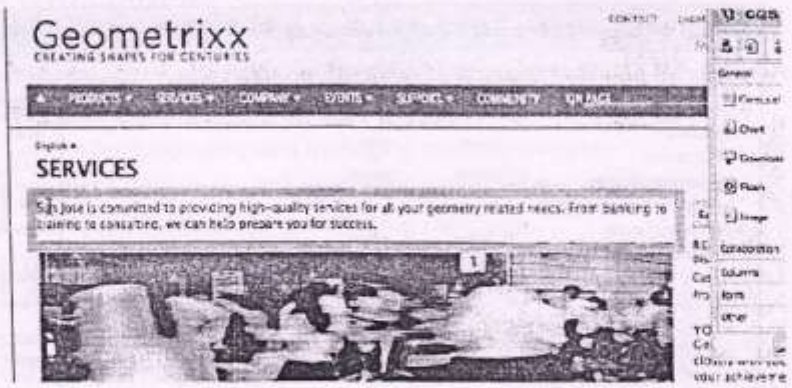
Administration Consoles: The administration consoles for each major CQ5 function (WCM, DAM, Workflow, etc.) present a consistent "explorer" interface. For example, the WCM console features a two-pane interface with a dynamic expandable/collapsable tree on the left and a grid with draggable rows and columns on the right.

Sidekick: A floating "inspector" window, sometimes known as a "floating palette", appears on the editable page from which new components can be dragged and actions that apply to the page can be executed. The Sidekick window acts as the author's toolbox.

Content Finder: On the left side of each editable page, the content Finder provides fully searchable, quick access to digital assets such as other images, Flash elements and documents as well as other pages and paragraphs. These items can be dragged to the page to position assets or create links to other pages, for example.

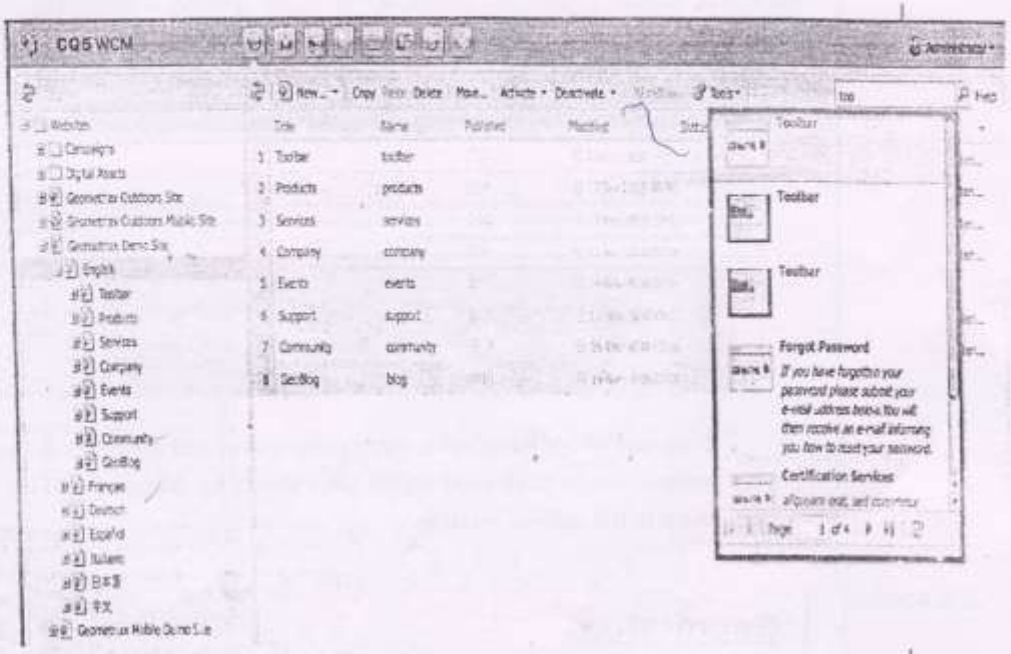


In-place Editing: Text components can be edited directly on the webpage without any intervening dialog box or explicit saving.



Drag and drop: Paragraphs, other components and digital assets such as images can be positioned on the page simply by dragging and dropping them to the desired location.

Search as you type: Searching for content through the CQ5 interface presents dynamic matches as you type the query.



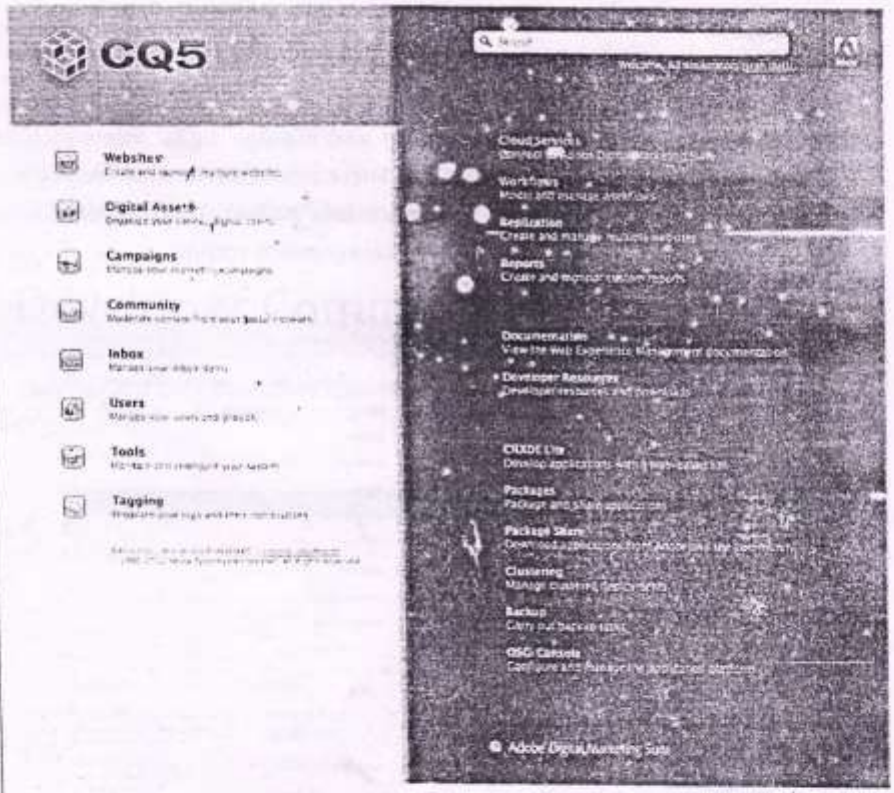
Context menu: Right clicking on most onscreen elements brings up a context menu with appropriate action options, just as in a desktop OS interface.

CQ5 Web Consoles

As mentioned above, the functionality of CQ5 is made available through various specialized web consoles:

- Websites for creating and managing multiple websites.
- Digital Assets for organizing various digital assets.
- Campaigns for managing marketing campaigns.
- Community for moderating content of the social network.
- Inbox for managing your inbox items.
- Users for managing user accounts and groups.
- Tools for maintaining and configuring the CQ5 system.
- Tagging for organizing tags and their namespaces.

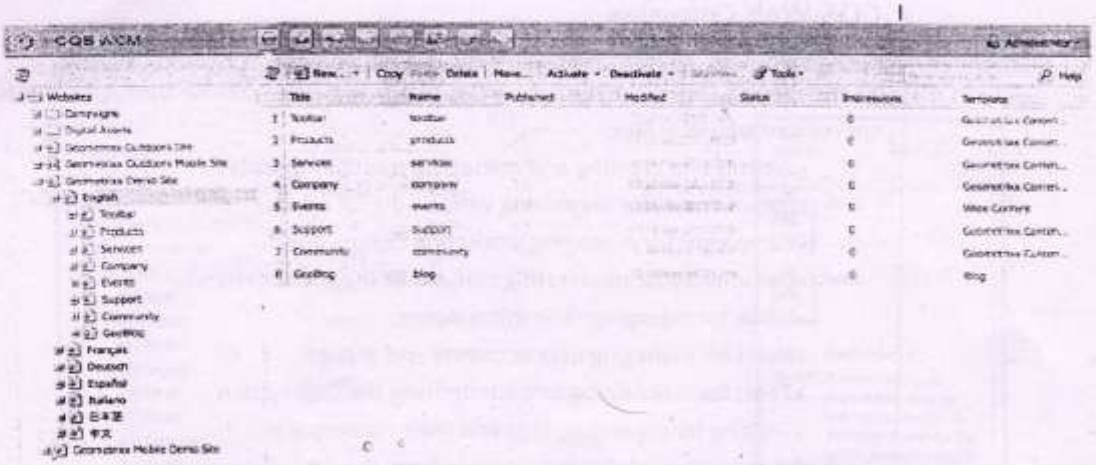
Each of these web consoles is accessible from the CQ5 Welcome Page:



To go to an web console, simply click on the appropriate icon. Once you are within a particular console, you can switch to another console using the tabs at the top of the console page.

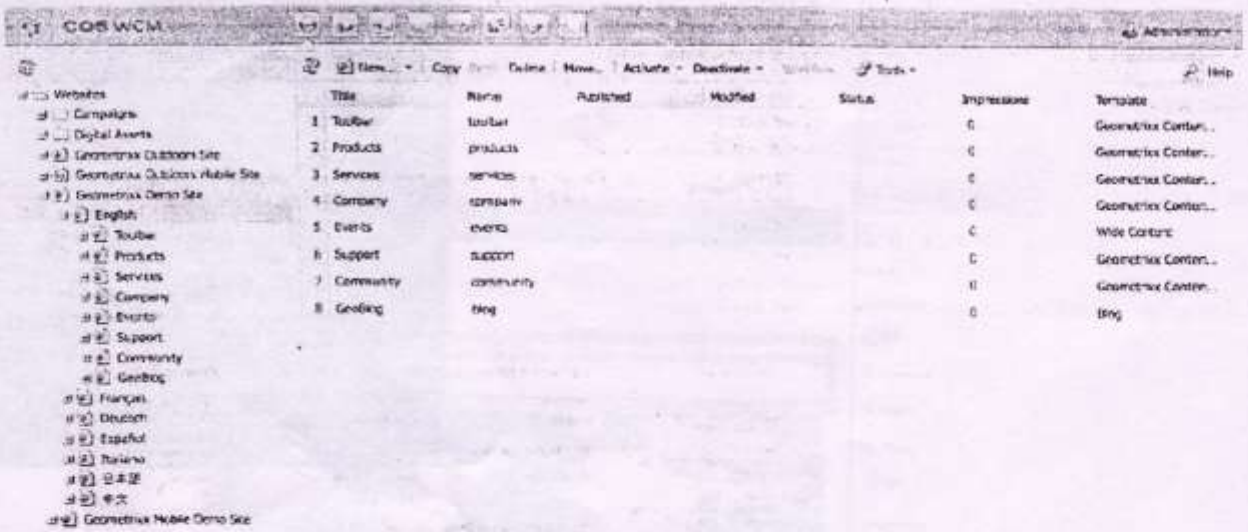
Websites Console

The Websites console, also known as the Site Admin console, lets you create, view and manage websites running on your CQ5 instance.Through this console you can create, copy, move and delete website pages, start workflows, and activate (publish) pages. Double clicking on a page icon either in the explorer tree on the



Digital Assets Console

The Digital Assets console lets you import and manage digital assets such as images, videos, documents and audio files. These assets can then be used by any website running on the same CQ5 instance (see above). This console is also referred to as the CQ5 DAM (Digital Asset Management) console.



Tools Console

The Tools console provides access to a number of specialized tools that help you administer your websites, digital assets and other aspects of your content repository.

See the CQ5 documentation for more information on these and other consoles:
http://dev.day.com/content/docs/en/cq/current/exploring/concepts.html

Developer Community

The CQ5 and CRX developer community site is your one stop shop for all things CQ5 and CRX.

As you can see from the screenshot above, from dev.day.com, you have access to the:

- Documentation - online documentation for authors, developers, and administrators
- Knowledge base - technical articles focussed on "how to" in answer to specific technical questions
- Customer support portal - support and ticket management
- Discussion groups - access to forum discussions among the wide community of developers and administrators using CQ5 and CRX.

Key Principles Underlying the Design and Implementation of CQ5

The following key principles are key to understanding why CQ5 works the way it does.

Standards and Open Source

CQ5 is built using standards and open source products. The most important of those are:

- 100% Java / J2EE
- JSR 283 - a Java Content Repository (JCR)
- Apache Felix - an OSGi framework
- Apache Sling - a Web framework for the Java platform designed to create
- content-centric applications on top of a JSR

Everything is content

This is the philosophy behind CQ5. Templates are content, user data is content, user ACLs are content and — of course — content is content. In addition, the compiled JSPs are also content. And where does content go? Content resides in the content repository. There are no loose files somewhere else to manage. Source code, dynamic modules, configuration and even the state of an application reside side by side with documents and other digital assets such as images, audio and video, etc.

David's Content Model

- Data first, Structure later (maybe)

  Don't worry about a declared data structure in an ERD sense. Initially. Structure is expensive and in many cases it is entirely unnecessary to explicitly declare structure to the underlying storage.

- Drive the content hierarchy, don't let it happen

  The content hierarchy is a very valuable asset. So don't just let it happen, design it. If you don't have a "good", human-readable name for a node, that's something you should probably reconsider. Arbitrary numbers are hardly ever a "good name".

  While it may be extremely easy to quickly put an existing relational model into a hierarchical model, one should put some thought in that process.

- Names should have meaning
  - IDs are evil - use meaningful names

    In relational databases IDs are a necessary means to express relations, so people tend to use them in content models as well. Mostly for the wrong reasons through. If your content model is full of properties that end in "Id" you probably are not leveraging the hierarchy properly. It is true that some nodes need a stable identification throughout their live cycle. Much fewer than you might think though.

  - References can be harmful if they create the need for structure

    References imply referential integrity. It is important to understand that references do not just add additional cost for the repository managing the referential integrity, but they also are costly from a content flexibility perspective.

Authoring Interface

CQ's web-based interface uses AJAX and other modern browser technologies to enable WYSIWYG editing and formatting of content by authors right on the web page. The use of Web 2.0 capabilities produces an intuitive, streamlined (and fun) experience for authors.

Desktop Integration

WebDAV lets you display and edit the repository content. Setting up WebDAV gives you direct access to the content repository through your desktop. Office and PDF files that are dropped into the repository through the WebDAV connection are automatically full-text indexed and can be searched with the standard search interfaces through the standard Java APIs.

CIFS s a Microsoft-based network file sharing system that CRX can use to give you direct access to the content repository through the desktop. It is similar to WebDAV.

OSGi and Apache Sling

CQ5 is built within an OSGi application framework. OSGi is a dynamic module system for Java that provides a framework within which small, reusable standardized components can be composed into an application and deployed.

The Apache Sling framework is designed to expose a JCR content repository through an HTTP-based REST API. Both CQ's native functionality and the functionality of any website built with CQ5 are delivered through this framework.

Clustering

In computing, a cluster is a group of computers linked together to work, in some respects, as a single computer. Every CRX instance comes pre-configured to run within a cluster, even when running a singular instance. This design feature allows the configuration of multi-node clusters with little effort.

CQ5 Functional Building Blocks

CQ5 consists of sets of OSGi bundles that are deployed on the CRX. The CQ5 application modules use the JCR API to manipulate content in CRX. The CQ5 application modules sit on top of the CQ5 shared framework, which contains all application layer functionality that is shared among all the application modules; for example mobile functionality, multi-site manager, taxonomy management, and workflow. In addition to the shared application framework, the CQ5 applications (WCM, DAM, ..) share the same infrastructure and same UI framework. With the install of CQ5 you get all applications, as they are tightly integrated.

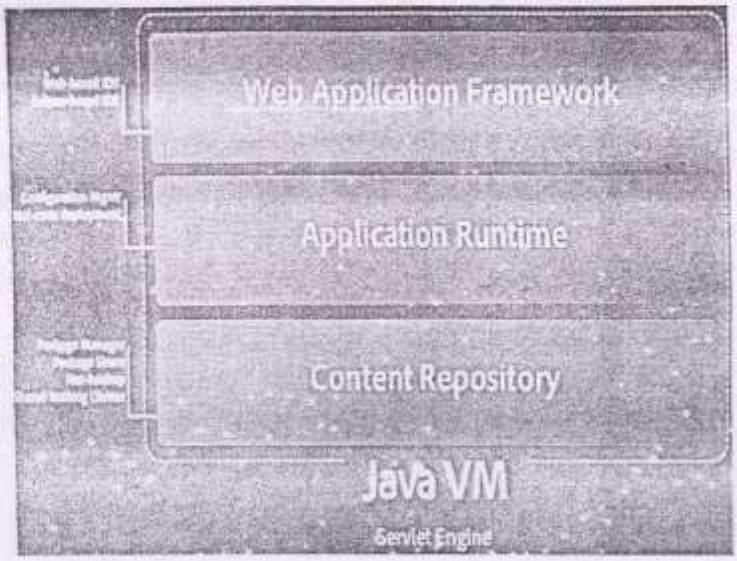
Application functionality that is not used or not licensed can be disabled after the initial install.

3rd party repositories can be integrated with JCR connectors that expose their content into CRX and are available to CQ5. Notice that the connectors are plugged in at the content repository level, which allows the content in the external repositories to appear to authors as if that content existed in the local content repository - a true virtual repository.

CRX Content Platform

The figure below depicts a simplified view of the CRX Content Platform.

The basis of the CRX Content Platform is the JSR-283 (JCR 2.0) and JSR-283 (JCR 2.0) compliant Content Repository. Since "Everything is Content" and all the content is in the content repository, you will note that clustering and backup is done at the repository level.

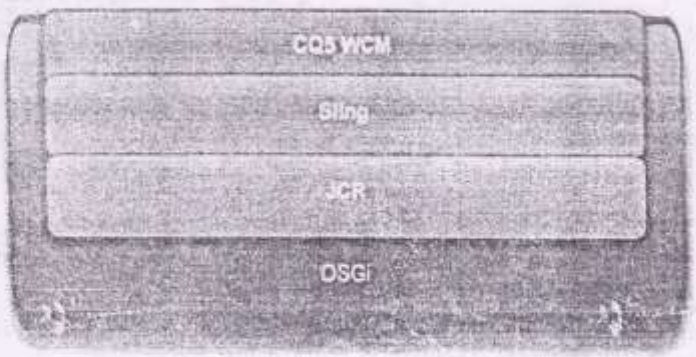


The Application Runtime is OSGi and Apache Sling. At this layer you can hot deploy any code that you wrap up as an OSGi bundle. The OSGi runtime hosts Java applications that can access the repository via the JCR API.

As part of the Application Runtime, we ship Apache Sling, a web application framework that exposes the full repository content via HTTP and other protocols.

CRX ships with it's own default Servlet engine – therefore the only requirement to run is the Java VM. CRX can also be installed into other Servlet containers or Java J2EE application server.

Architecture Stack



Open Systems Gateway Initiative (OSGI)

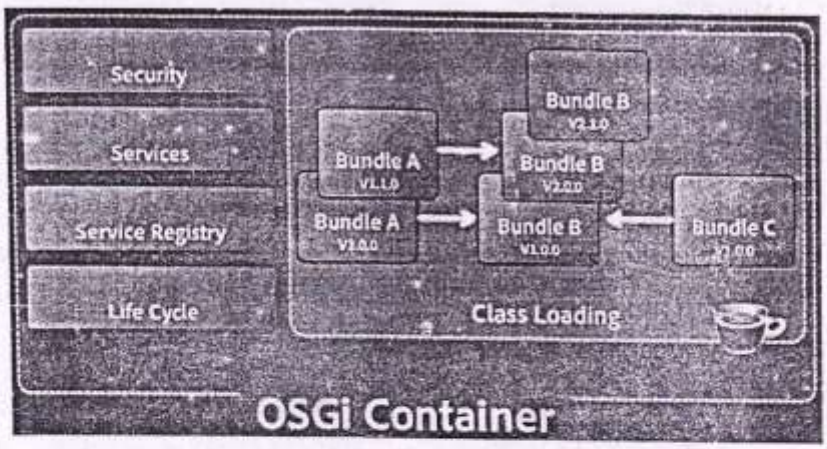
OSGi is a consortium that has developed a specification to build modular and extensible applications. The OSGi module system allows building applications as a set of reloadable and strongly encapsulated services.

Traditional Java application servers use a fairly monolithic approach to application deployment. OSGi "bundles" run inside an OSGi "container". The OSGi container manages relations among bundles, which are simply JAR files that contain extra metadata indicating what services they require and which they provide.

The OSGi specifications define a dynamic component system for Java:

- OSGi specifications enable
  - Development model where applications are (dynamically) composed of many different (reusable) components
  - Components hide their implementations from other components while communicating through services, which are objects specifically shared between components
- Collaborative software environment
  - Application emerges from assembling multiple reusable modules that have no previous knowledge of each other

OSGi is a solution to the new "DLL-hell", which is the Java class loader. You can have more than one version of any bundle running in the container at the same time. Container resolves any version dependencies.



OSGi is a platform in the Java-stack that allows large development teams to be more efficient and the ability to replace code pieces (bundles) during normal operations of the application - in other words, without taking the server down.

The OSGi container (Apache Felix) that is included in CRX is compliant to Version 4.2 of the OSGi specification.

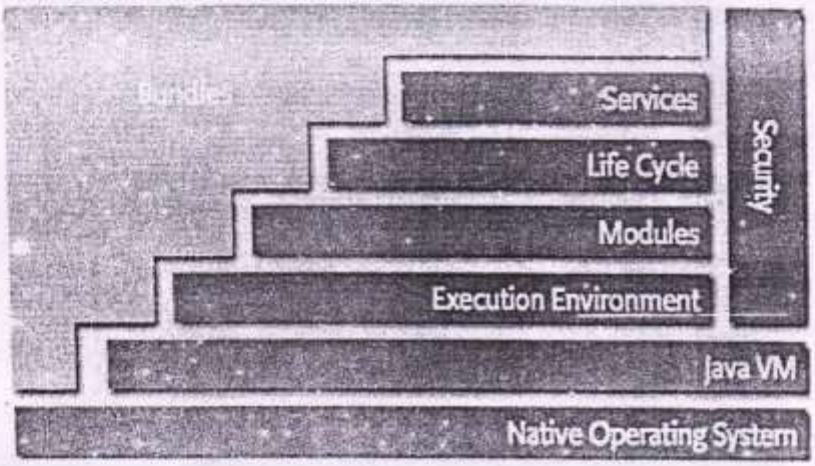
By using an OSGi solution, the following benefits are gained:

- Code is easier to write and test

- Reuse is increased Build systems become significantly simpler
- Deployment is more manageable
- Bugs are detected early
- Runtime provides an enormous insight into what is running

OSGi Framework

The OSGi Framework is made up of three layers - Module, Lifecycle, and Services - that define how extensible applications are built and deployed.



The responsibilities of the layers are:

- Module — Defines how a module, or a Bundle in OSGi-speak, is defined. Basically, a bundle is just a plain old JAR file, whose manifest file has some defined entries. These entries identify the bundle with a symbolic name, a version and more. In addition there are headers which define what a bundle provides - Export-Package - and what a bundle requires to be operative - Import-Package and Require-Bundle.

- Lifecycle — The lifecycle layer defines the states a bundle may be in and describes the state changes. By providing a class, which implements the BundleActivator interface and which is named in the Bundle-Activator manifest header, a bundle may hook into the lifecycle process when the bundle is started and stopped.

- Services — For the application to be able to interact, the OSGi Core Specification defines the service layer. This describes a registry for services, which may be shared.

Key principles of OSGi:

- Universal middleware, dynamic module system
- Service oriented, component-based environment
- Standardized software life-cycle management

The implementation of OSGi that the CQ5 Platform makes use of is Apache Felix.

OSGi Bundles

- OSGi bundles can contain compiled Java code, scripts and content that is to be loaded into the repository, in addition to configuration and / or other files as needed
- Bundles can be loaded and installed during normal operations
- For CQ5, bundles are dropped into specially named folders (.../install) in the repository. They can also be managed by the Apache Felix Management Console

Additional Information

To learn more about OSGi, go to

http://www.osgi.org
http://en.wikipedia.org/wiki/OSGi

To learn more about Apache Felix, specifically, go to

http://felix.apache.org

Java Content Repository (JCR)

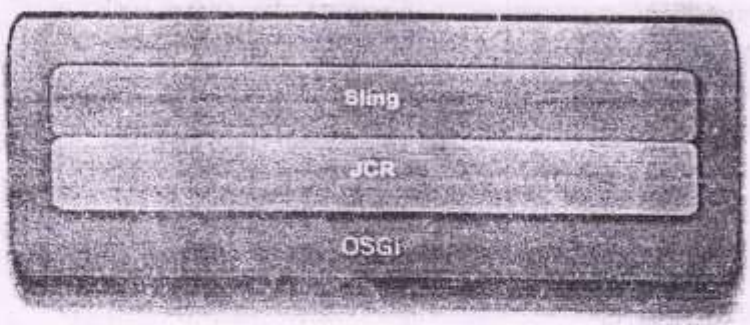
According to JSR-283, *the Java Content Repository API defines an abstract model and a Java API for data storage and related services commonly used by content-oriented applications.*

A Java Content Repository is an object database that provides various services for storing, accessing, and managing content. In addition to a hierarchically structured storage, common services of a content repository are versioning, access control, full text searching, and event monitoring.

The JCR provides a generic application data store for both structured and unstructured content. File systems provide excellent storage for unstructured, hierarchical content. Databases provide excellent storage for structured data due to transactional services and referential integrity functionality. The JCR provides the best of both data storage architectures, plus observation, versioning, and full text search.

An additional advantage of the JCR is support for namespaces. Namespaces prevent naming collisions among items and node types that come from different sources and application domains. JCR namespaces are defined with a prefix, delimited by a single : (colon) character, for example - jcr:title.

CQ5 is designed to store and retrieve content from any JCR-compliant content repository. In its default configuration, content storage is handled by the CRX repository that comes bundled with CQ5. CRX is Adobe System's implementation of the JCR standard and the version of CRX that comes with CQ5 supports JCR 2.x.



In addition to CRX, CQ5 can also work with other JCR repositories, such as Apache Jackrabbit, and with a number of non-JCR data stores, through connectors. Since CQ5 is built-on top of this standard it is capable of pulling in content not just from its built in CRX repository but from any JCR-compliant source, such as a third-party repository (see, for example, Jackrabbit) or a connector that exposes legacy storage through JCR (see Connectors).

JCR defines a Java API for a class of data storage systems called content repositories. A content repository, as defined by JCR, combines features of the traditional relational database with those of a conventional file system, as well as additional services that content-centric applications often need, but that neither file systems nor databases typically provide. See the CRX ad JCR documentation for more information on this topic.

JCR Structure

The JCR consists of a set of one or more workspaces, each containing a tree of nodes with associated properties. Each node may have one primary node type which defines the characteristics of the node. There may also be associated with any node, zero or more mixins, which define additional node characteristics and behaviors.

Beginning with the root node at the top, the hierarchy descends much like the directory structure of a file system: each node can have zero or more child nodes and zero or more properties. Properties cannot have children but do have values.

The values of properties are where the actual pieces of data are stored. These can be of different types: strings, dates, numbers, binaries and so forth. The structure of nodes above the properties serves to organize this data according to whatever principles are employed by the application using the repository.

Nodes may point to other nodes via a special reference type property. In this way nodes in a JCR offer both referential integrity and object-oriented concept of inheritance.

Content Services of the JCR

- Search
- Indexing
- Observation
- Versioning
- Access control / security
- Transactions

Key principles behind the specification of JSR 283 are:

- Common programmatic interface to content repositories
- API not tied directly to underlying architecture, data source, or protocol
- Content organization in repository model
  - Hierarchical modeling

The reference implementation for JSR 283 is Apache Jackrabbit. To learn more about Apache Jackrabbit go to
http://jackrabbit.apache.org
The Adobe implementation of JSR 283 is the Content Repository Extreme (CRX).

Adobe CRX

CRX implements the Content Repository API for Java Technology (JCR). This standard defines a data model and application programming interface (that is, a set of commands) for content repositories. Content is available through a standardized API

- JSR 283
- JCR API
  - Node node.addNode("JCR")
  - Node node.getNode("JCR")
  - Node node.setProperty("opinion","excellent and recommended")
  - Node node.getProperty("opinion").getString()

Built-in Protocols/APIs for the CRX Platform

Add, consume, managed content with these interfaces:

- Java Content Repository API – complete JCR 2.0 implementation
- Content Management Interoperability Services - CMIS 1.0
- WebDAV – with versioning, access control and search
- Windows Network File Share – CIFS/SMB
- RESTful Web API for JavaScript and Flash/Flex
- Java Remoting with RMI and HTTP
- LDAP and any JAAS plug-in
- Native repository interface via Virtual Repository – e.g. Microsoft SharePoint

Representational State Transfer (REST)

An architectural style for distributed systems. Application data and state are represented as a set of addressable resources which present a uniform interface that allows transfers of state (e.g. reading and updating of the resource's state). The best example of a REST-ful architecture is the web, where "resources" have URIs and the "uniform interface" is HTTP.

For most cases, a framework like HTTP (addressable resources + "verbs" + standard ways to transmit metadata) is all you need to build a distributed application. HTTP is actually a very rich application protocol which gives us things like content negotiation and distributed caching. REST-ful web applications try to leverage HTTP in its entirety using specific architectural principles.

1. Resource-oriented. Any piece of information (content object), for example a news entry or product description or photo is a resource.

2. Addressable Resources. With REST over HTTP, every object will have its own specific URI. From the URI, we know how to communicate with the object, as well as where it is in the network and its identity on the server where it resides.

3. A Uniform, Constrained Interface. When using REST over HTTP, stick to the methods provided by the protocol. This means following the meaning of GET, POST, PUT, and DELETE as defined with no additional methods/services.

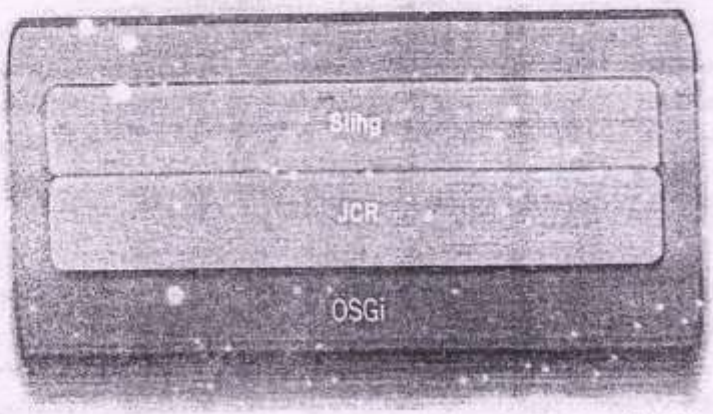
4. Representation oriented. You interact with services using representations of that service. An object referenced by one URI can have different formats/ renderings available. Different clients need different formats. AJAX may need JSON. A Java application may need XML. A browser my need HTML. A human may need a print-friendly rendering.

5. Communicate statelessly. REST as implemented in HTTP tends to be stateless; for example, it doesn't use cookies and clients need to re-authenticate on every request. No client session data is stored on the server. Session-specific is held and maintained by the client and transferred to the server on each request, as needed. Stateless applications are easier to scale.

http://sling.apache.org

Apache Sling

Apache Sling is a web framework that uses a Java Content Repository, such as Apache Jackrabbit or Adobe CRX to store and manage content. Sling applications use either scripts or Java servlets, selected based on simple name conventions, to process HTTP requests in a RESTful way.



REST refers to the software architectural style on which the World Wide Web is based. It describes the key elements that make the Web work, and so provides a set of principles for how web-based software should be designed. When designing an API to be used over the Web, it therefore makes sense to adhere to these "best practices". Apache Sling does just that.

The Sling application is built as a series of OSGi bundles and makes heavy use of a number of OSGi core and compendium services. The embedded Apache Felix OSGi framework and console provide a dynamic runtime environment, where code and content bundles can be loaded, unloaded and reconfigured at runtime.

As the first web framework dedicated to JSR-283 Java Content Repositories, Sling makes it very simple to implement simple applications, while providing an enterprise-level framework for more complex applications.

Being a REST framework, Sling is oriented around resources, which usually map into JCR nodes. Unlike traditional web applications that select a processing script, based on the URL, and then attempt to load data to render a result, Apache Sling request processing takes what, at first might seem like an inside-out approach to handling requests in that a request URL is first resolved to a resource, then based on the resource (and only the resource) it selects the actual servlet or script to handle the request.

4502, 4504, authoring (any 4502, 4505 → publishing could)

Everything is a Resource

The Resource is one of the central parts of Sling. Extending from JCR's "Everything is Content", Sling assumes that "Everything is a Resource". A resource is Apache Sling's abstraction of the object addressed by the URI. Sling resources usually map to a JCR node.

Servlets and Scripts are handled uniformly in that they are represented as resources themselves and are accessible by a resource path. This means, that every script, servlet, filter, error handler, etc. is available from the ResourceResolver just like normal content providing data to be rendered upon requests.

Sling Script Resolution

Using Sling, the type of content to be rendered is not the first processing consideration. Instead the main consideration is whether the URL resolves to a content object for which a script can then be found to perform the rendering. This provides excellent support for Web content authors to build Pages which are easily customized to their requirements.

The advantages of this flexibility are apparent in applications with a wide range of different content elements, or when you need Pages that can be easily customized/ viewed differently.

In Sling, and therefore also CQ5, processing is driven by the URL of the HTTP request. This defines the content to be displayed by the appropriate scripts. To do this, information is extracted from the URL.30

When the appropriate resource (content node) is located, the resource type is extracted from the resource properties. The sling:resourceType property is a path, which locates the script to be used for rendering the content.

Traditional web application frameworks usually begin servicing a request by selecting a procedure (servlet, script etc.) based on the request URI. This procedure then loads some data, usually from a database, and uses it to construct and send back a response. For example, such a framework might use a URI like this:

http://www.example.com/product.jsp?id=1234

This URI addresses the script to be used to render the result (product.jsp), not the resource that is being requested (product #1234).

Sling turns this around by placing the content at the center of the process. Each content item in the JCR repository is exposed as an HTTP resource, so the request URI addresses the data to be processed, not the procedure that does the processing.

Once the content is determined, the script or servlet to be use to handle the request is determined in cascading resolution that checks, in order of priority:

- Properties of the content item itself.
- The HTTP method used to make the request.
- A simple naming convention within the URI that provides secondary information.

For example, a Sling application might use a URL like this:

http://www.example.com/cars/audi/s4.details.html

This URL would be decomposed as follows:

- cars/audi/s4 refers to the repository path to the content item (e.g., /content/cars/audi/s4).
- details.html refers to the script to be used to render this content (e.g. /apps/cars/details/html.jsp).

Because a Sling URI addresses content and not presentation logic, such a URI tends to be more meaningful and more stable over time.

Sling and MVC

Some Java developers, especially those familiar with Struts/Spring may not recognize Sling as MVC. However, as stated by Alexander Klimetschek of Day Software (now Adobe Systems):
"Sling is *totally* MVC:

- model = JCR repository
- controller = Sling engine (url processing, resource + script resolution)
- view = custom rendering scripts/servlets, both for read and write (GET + POST)..."

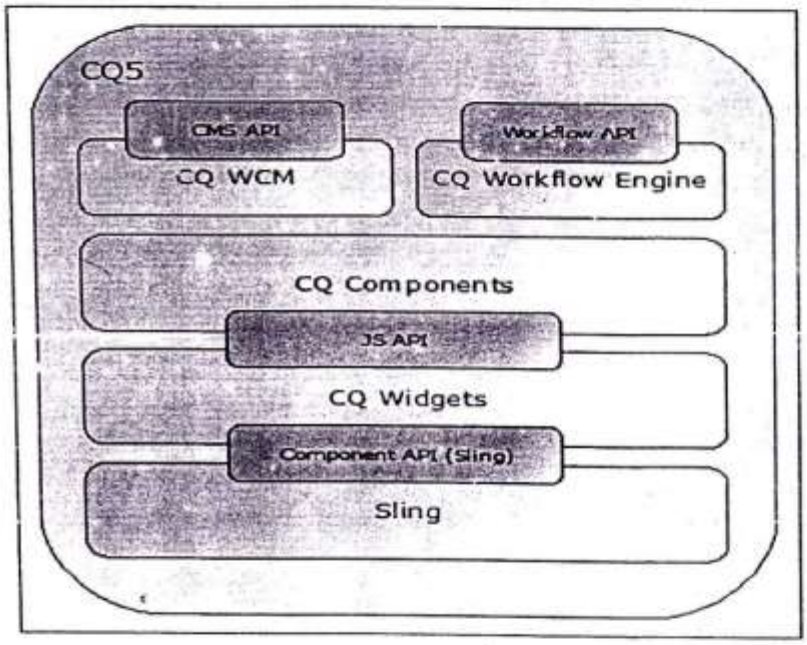
Additional Information

http://sling.apache.org

CQ5 Application Modules

The application module layer has its own functionality and architectural structure. The Key Principles behind the design and implementation of the CQ5 application layer are:

- Ajax/JavaScript user interface, APIs, and libraries
- Dialog and widget framework
- Workflow engine
- CMS application



As an example, WCM CMS application, which enables the creation, management, and control of content embodies full Web 2.0 capabilities:

- Ajax / JavaScript user interface, APIs, and libraries
- Direct client-side JavaScript access to JCR content
- Workflow engine – controls the processing of workflow instances
  - Workflow instances often control the process of generating and publishing content

The Components - independent, reusable pieces of content rendering logic - are actually collections of rendering scripts.

The Widgets - basic ExtJS elements that enable the input of content by authors. These widgets often manifest as dialog boxes.

Apache Sling - A component framework that we discussed earlier, which maps content objects to the appropriate rendering scripts (Components).

Section-2 Installation and Deployment

A Java web application (or web app) is a program that runs on a remote server and is accessed by users through a web browser. It is usually contrasted with a static website, which is simply a collection of documents that reside on the server and that can be viewed through a browser, over the web. A web application, on the other hand, typically assembles the data to be sent back to the browser, on the fly with each request (or even, using AJAX, between requests). Most modern websites nowadays are essentially web applications. CQ5 is also a web application, but one that serves as a platform for building other web applications and managing the content that they deliver.

As a Java web application, CQ5 runs on any server that supports the Java Servlet API. For ease of installation CQ5 comes bundled as a self extracting Quickstart file that needs only to be double clicked, in order to install and start the server (see deployment)

Since CQ5 is Java-based it can run on any system for which a Java Runtime Environment is available. This means, for example, that CQ5 can run on any mainstream operation system, including Windows, Macintosh, Linux, or other flavors of Unix.

While different instances in different environments are all installations of the same CQ5 software, installed in different places in the overall system infrastructure, they differ mainly in the way they are configured. For example, it is that configuration, or *run mode*, that determines whether a CQ5 instance behaves as an author instance or a publish instance.

What is an Author instance?

Author instances are usually located behind the internal firewall. This is the environment where you, and your colleagues, will perform authoring tasks, such as:

- Administer the web properties
- Input your content
- Configure the layout and design of your content
- Activate your content to the publish environment

Content that has been activated is packaged and placed in the author environment's replication queue. The replication process then transports that content to the publish environment.

An Author instance is the CQ5 installation content authors will login to and manage pages. This includes: 1) creating, 2) editing, 3) deleting, 4) moving, 5) etc. In addition, it is the instance run mode you will use for development, as you can easily observe both Author and Publish views from this instance run mode

What is a Publish instance?

A publish environment is usually located in the Demilitarized Zone (DMZ). This is the environment where visitors will access your website and interact with it; be it public, or within your intranet.

- Holds content replicated from the author environment
- Makes that content available to the visitors of your website
- Stores user data generated by your visitors, such as comments or other form Submissions
- May be configured to add such user data to an outbox, for reverse-replication back to the author environment

The publish environment generates your websites content pages dynamically in real-time and the content can be personalized for each individual user.

Installing CQ5

Unlike many other applications, you install CQ5 WCM by using a "Quickstart" self-extracting jar file. When you double-click the jar file for the first time, everything you need is automatically extracted and installed. The CQ5 quickstart jar file includes all files and repository structures required for:

- CRX repository (a fully JCR 2.0 / JSR-283 compliant repository and Apache Sling), virtual repository services, index and search services, workflow services, security and a Web server.
- CQ5 Servlet Engine (CQSE). You can run CQ5 WCM without an application server, but you need a Servlet Engine. Both CRX, and therefore CQ5 WCM, ship with Day's CQSE servlet engine, which you can use freely and which is fully supported.

The first time you start the jar file, it creates an entire JCR-compliant repository in the background, which may take several minutes. After this startup is much quicker as the applications have been installed and the repository already created.

EXERCISE - Install & Start an Author Instance

The following instructions explain how to install and start a CQ5 Author instance. This is important because you will use this Author instance throughout this training to perform typical development tasks.

What will we install?

The CQ5 quickstart jar file is a self-extracting jar that will install:

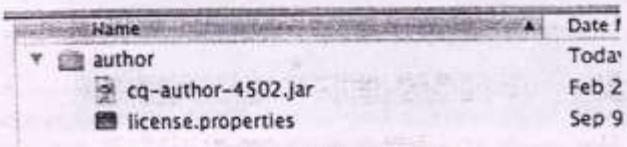
- CRX - Day Software implementation of JSR 283, a Java Content Repository
- CQ5 Platform
- Reference Web site - Geometrixx
- CRXDE Lite - functionality embedded into CRX/CQ5 that enables you to perform standard development tasks in the browser.

How to install an Author instance:

1. Create a folder structure on your file system where you will store, install, and start CQ5. For example:

   - ***Windows:*** C:/adobe/CQ5/author
   - ***MacOS X:*** /Applications/adobe/CQ5
   - ***MacOS X or *x:*** /opt/adobe/CQ5/author).

2. Copy the CQ5 quickstart JAR and license.properties file from <USB>/distribution/CQ5_wcm into the newly created folder structure.



CQ5 author install folder structure

3. Rename the CQ5 quickstart JAR to *cq-author-4502.jar.*

   - cq = the application
   - author = the WCM mode it will run in (e.g. author or publish)
   - 4502 = the port it will run in (e.g. any available port is acceptable)

4. In a Windows or MacOS X environment you can simply double-click the ***cq-author-4502.jar*** file.

   - Installation will take approximately 5-7 minutes, depending on your systems capabilities
   - A dialog will pop-up similar to the one below

CQ5 install/startup dialog

NOTE: In production, you may install CQ5 in any directory structure. However, for the purposes of this class, we recommend the above directory structures so that you can easily find the directory structures mentioned in the course materials.

NOTE: If no port number is provided in the file name, CQ5 will select the first available port from the following list: 1) 4502, 2) 8080, 3) 8081, 4) 8082, 5) 8083, 6) 8084, 7) 8085, 8) 8888, 9) 9362, 10) random..

TIP

There are two ways to install CQ5: graphical and by command line. The latter is more powerful since the user has the possibility to provide additional performance tuning parameters to the JVM.

On Windows, MacOS X or *x, you can also install/start CQ5 from the command line while increasing the Java heap size, which will improve performance. Please see image below:

CQ5 command line start

TIP

Typical command line start:

```
$ java -Xmx1024m -jar cq-auhtor-4502.jar -gui
```

Tuning the JVM is a very important and delicate task and requires a more realistic environment in terms of resources (hardware, operating system, etc) and workload (content, requests, etc). For now it will be enough to know that you can start your instance (either "author" or "publish") using the following parameters:

-Xms --> assigns initial heap size

Default value	64MB for a JVM running on 32bit machines, or 83MB for 64bit machines
Recommended	Specific to physical memory available and expected traffic
Syntax	**-Xms512m** (sets the initial heap size to 512MB)

-Xmx --> assigns the maximum heap size

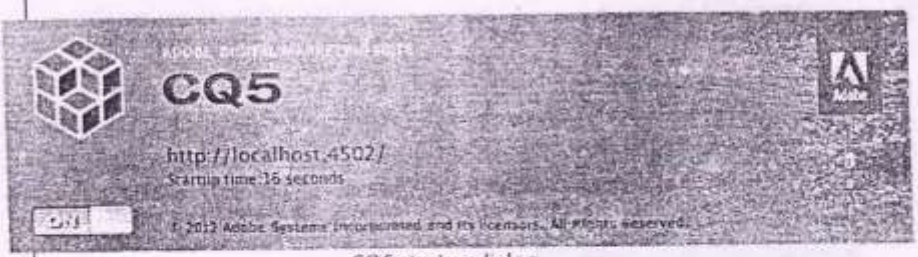
Default value	64MB for a JVM running on 32bit machines, or 83MB for 64bit machines
Recommended	Specific to physical memory available and expected traffic, but should be equal or greater than the initial size. To run CQ5 it's recommended to allocate at least 1024MB of heap size.
Syntax	-Xmx1024m (sets the maximum heap size to 1024MB)

-XX:MaxPermSize --> assigns the heap to hold reflective data of the VM (e.g. Java objects)

Default value	32MB for a JVM running as a client, or 64MB when running as a server
Recommended	Should be set to at least 128MB for "normal sized" Web apps or 256MB for larger Web apps with significant Java activity.
Syntax	-XX:MaxPermSize=128m (sets the initial perm gen size to 128MB)

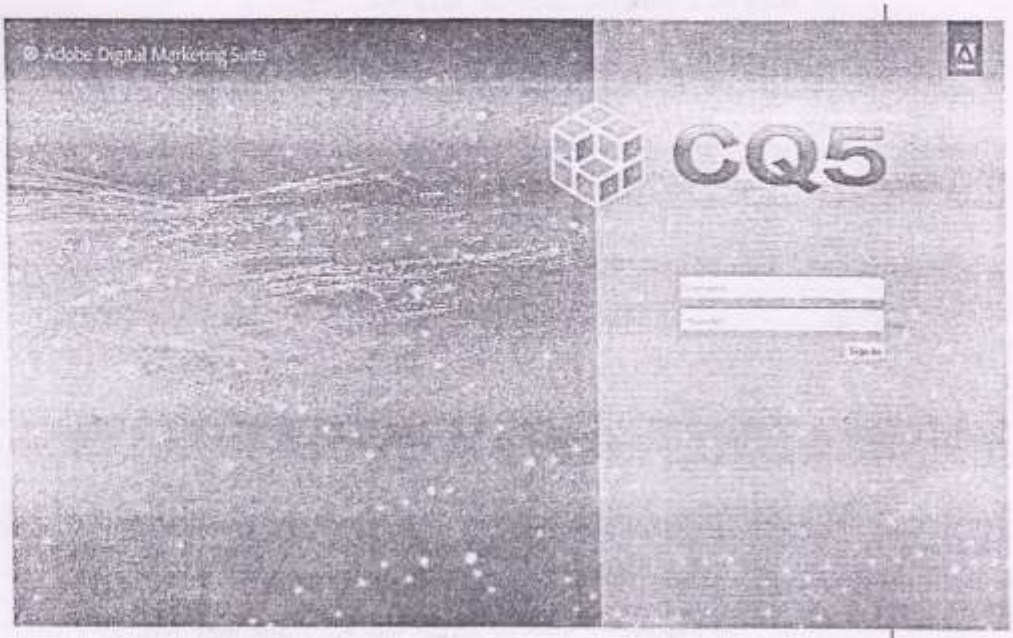
You can now install/start CQ5 from the command line together with increasing the Java heap and perm gen size, which will improve performance.

Once CQ5 has started successfully, the dialog will change to something similar to the following:



CQ5 startup dialog

Also, once the CQ5 is started, your default browser will automatically open to CQ5's start URL (where the port number is the one you defined on installation), e.g., http://localhost:4502.



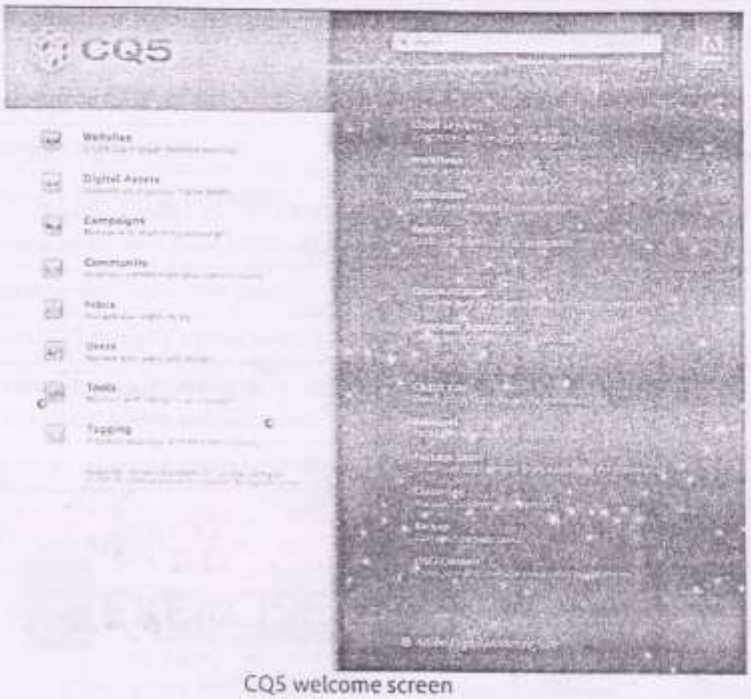
Congratulations! You have successfully installed and started CQ5. To start CQ5 in the future, double-click the renamed CQ5 quickstart JAR file (e.g. cq-author-4502.jar).

EXERCISE - Logging into CQ5

1. Enter the default administrator "Username" (admin) and "Password" (admin) in the CQ5 login screen after your favorite Web browser pops-up - then select OK.

CQ5 login dialog

2. Logging in will take you to the CQ5 Welcome Screen.



CQ5 welcome screen

The Welcome Screen give authors and other users access to the major functional areas of the CQ5 application.

Congratulations! You have successfully logged in to CQ5. Now let's try some authoring!

Authoring in CQ5 WCM

Now that we have a working author instance of CQ5, it is time to become familiar with authoring. The authors are your customers. You will be creating templates and components for their use. The authoring interfaces are your testing interfaces.

As you learned in previous sections, the author works in what is known as the author environment (author run mode). This provides an easy to use, graphical user interface for creating and editing content.

Depending on how your instance and your personal access rights are configured you can perform many tasks, including:

- Generate new content on an existing page
- Edit existing content
- Create new pages using predefined templates
- Move, copy or delete pages

- Activate (or deactivate) pages
- Participate in workflows that control how changes are managed; for example. enforcing a review before publication

Using any of the consoles, authors use the same paradigm:

- Double-click to open items.
- Access items from the tree list. Click + to expand and - to collapse those items.
- Click the icons to access other consoles.
- Available commands are above the items.
- Logging in and out is handled in the upper right corner.
- Collapse the sidebar by clicking the arrow.
- Search for pages using the search box.

What are the available Author interfaces?

CQ5 uses a web-based graphical user interface, so you need a web browser to access CQ5. The graphical user interface is divided into various web-based consoles where you can access all of the CQ5 functionality:

Console	Description
Websites	Access all the pages in your website; create, edit, and delete pages; start a workflow; activate and deactivate pages; restore pages; check external links; and access your user inbox.
Digital Assets	Manage digital assets.
Campaigns	Manage marketing campaigns.
Community	Moderate content from social network.
Inbox	Manage workflow inbox items.
Users	Manage users and permissions.
Tools	Manage packages, designs, importers, workflow templates and scripts, réplication agents and upgrades.
Tagging	Manage your tags and taxonomies.

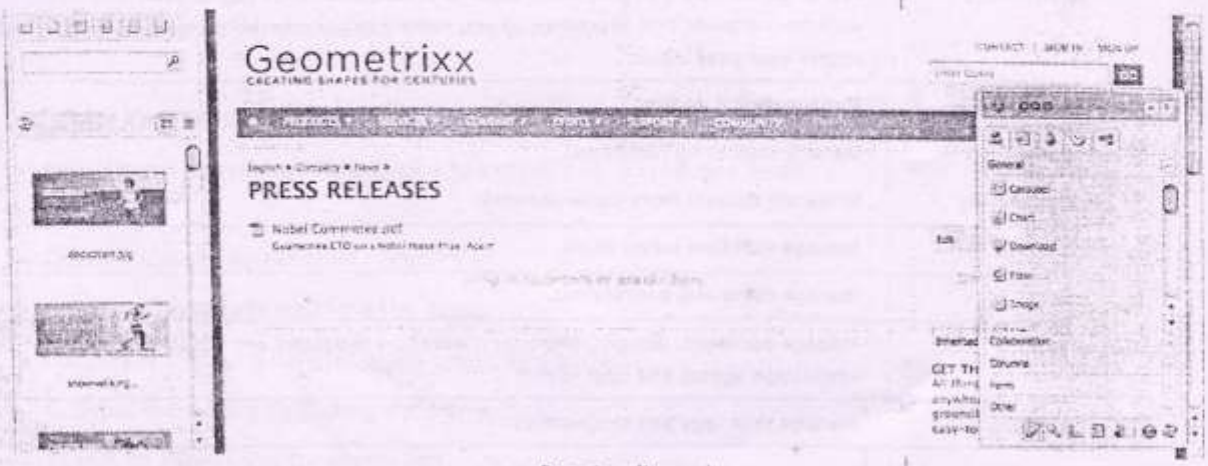
EXERCISE - Edit a page

1. Click on the Websites button on the Welcome Screen. This will take you to the Websites Administrator Console (Site Admin screen).

2. Navigate to and open the Press Releases page. You can open a page to be edited by one of several methods:

   - From Websites tab, you can double-click the page title to open it for editing.
   - From Websites tab, you can right-click the page item, then select Open from the menu

- After you have opened a page, you can navigate to other pages within the site to edit them by clicking the links.

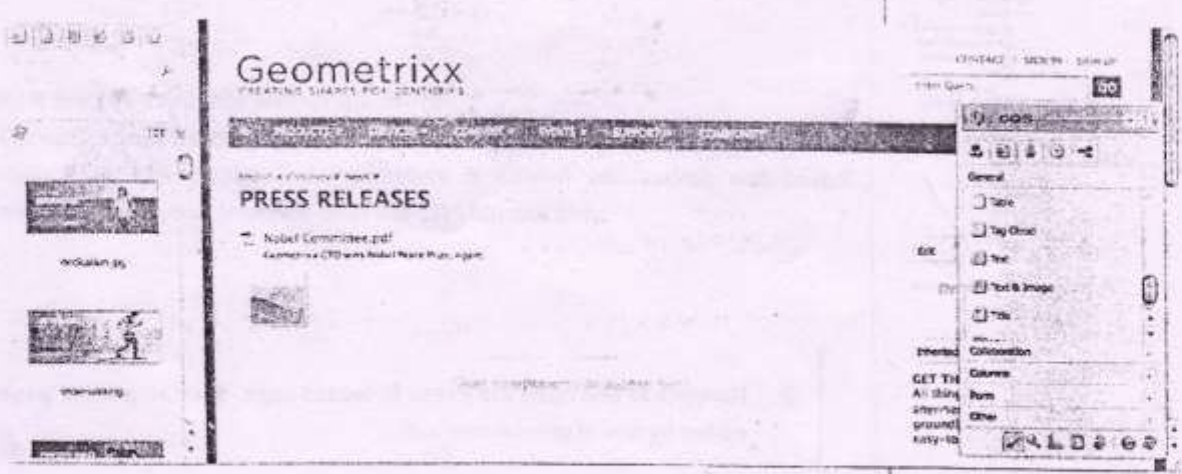


Page in edit mode

After you open the page, you can start to add content. You do this by adding new or editing existing paragraphs (also called components).

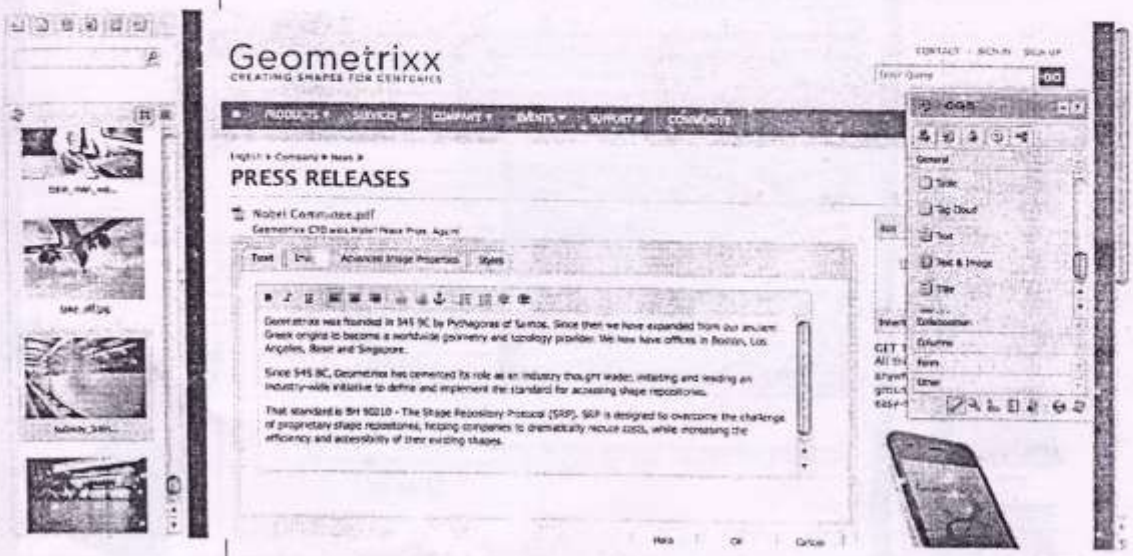
To insert a new paragraph, double-click the area labeled **Drag components or assets here...** or drag a component from the floating toolbar (called sidekick) to insert a new paragraph. This area appears wherever new content can be added, such as at the end of the list if other paragraphs exist or at the end of a column.

3. Drag the Text & Image icon from the sidekick to the center of the dotted rectangle and drop it in. The green check mark will tell you that the drag-and-drop is allowed.

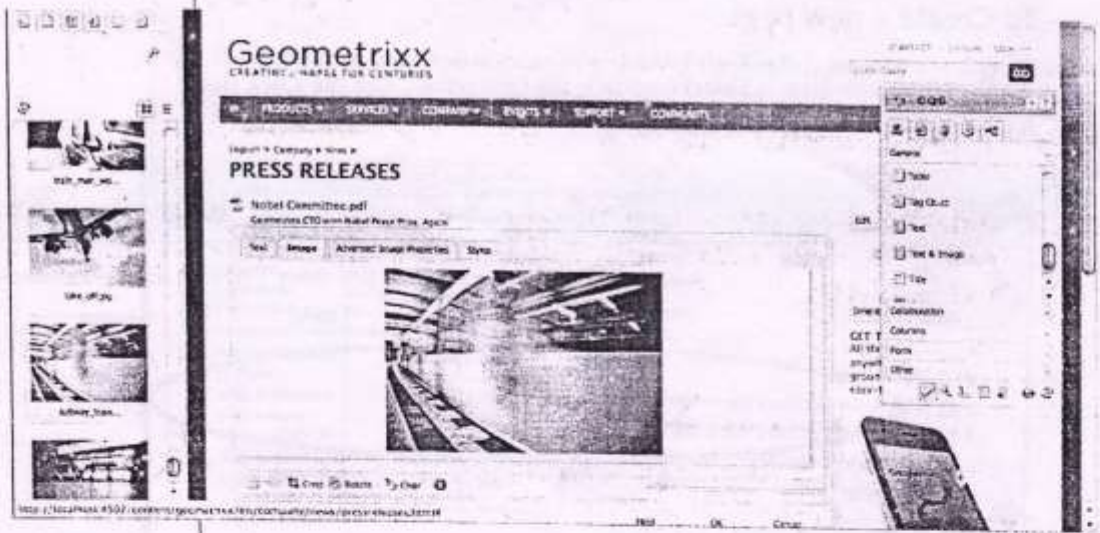


4. Double-click the thumbnail placeholder for the component to open the dialog box.

5. Click the **Image** tab to open the Image pane of the dialog box. Drag-and-drop an image from the Content Finder to the dialog box.



6. Click the **Text** tab. Enter some text.

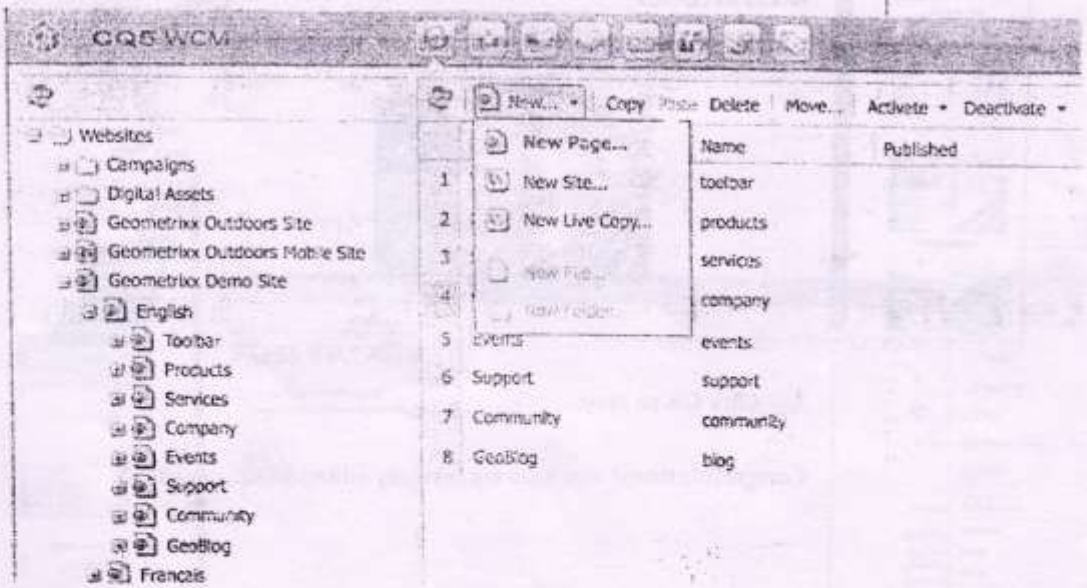


7. Click **OK** so save.

Congratulations! You have successfully edited a CQ5 page!!

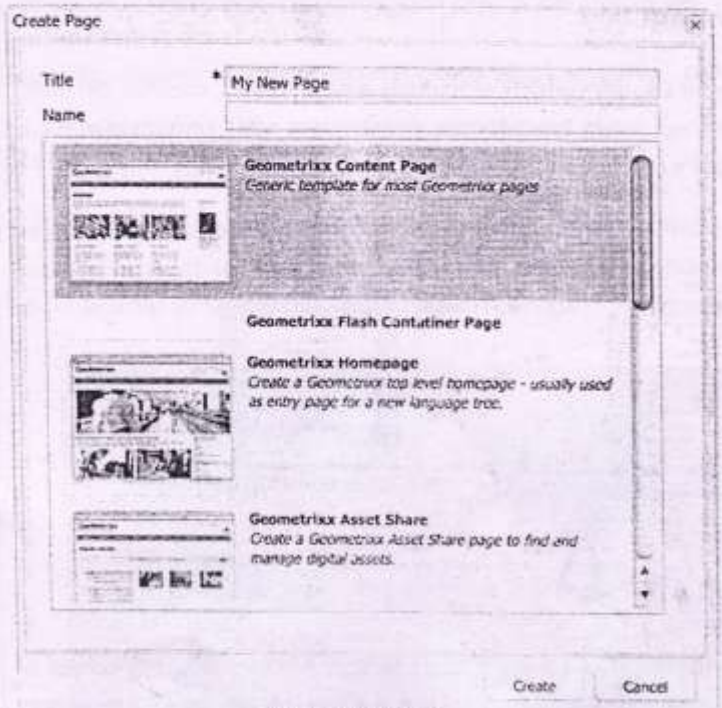
To Create a new page:

On the Site Admin screen, **Select** English in the Geometrixx Web Site. **Click New..** **New Page** on the right-hand toolbar.



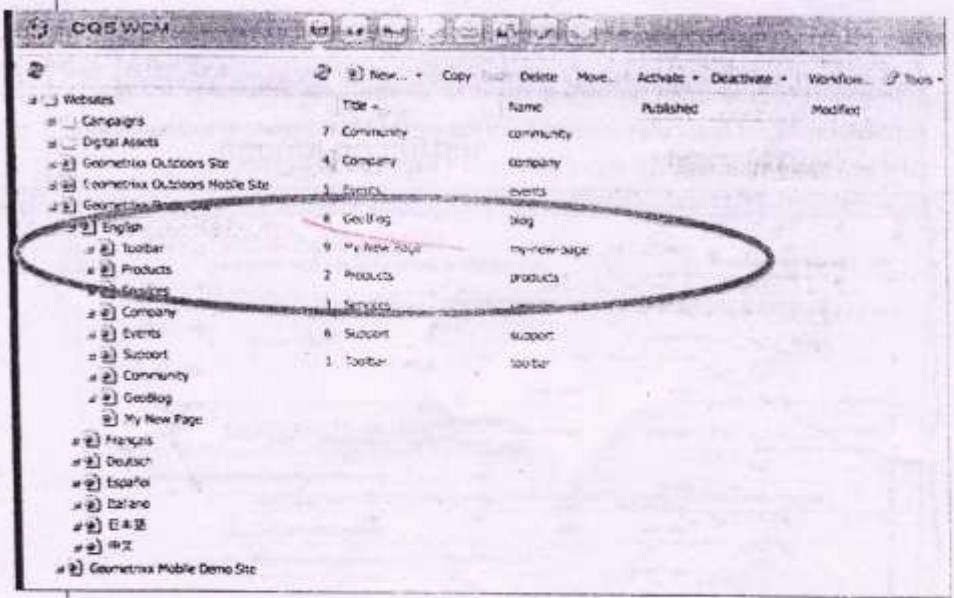
New page menu

8. A list of templates will appear. Fill in the input forms in the dialog:
   - Title: My New Page
   - Name: my_new_page
     Then select **Create** button



New page dialog

9. Your new page will appear as a child of the English page in the Site Admin Console. Open your new page.

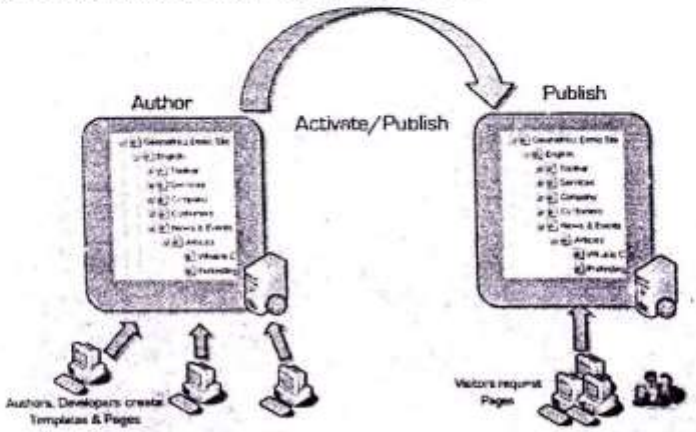


4. Put content on your new page, as you learned above.

Congratulations! You have successfully created a CQ5 page and filled it with content!!

CQ5 Deployment

A CQ5 deployment usually consists of multiple environments, used for different purposes on different levels. A production environment often consists of at least one author instance and one publish instance.



Depending on the scale of a project, a production environment may consist of several author and/or publish instances, and at a lower level, the CRX repository may be clustered among several instances as well. Additionally, separate development and test environment levels may also consist of both author and publish instances, mirroring the production environment to a varying extent.

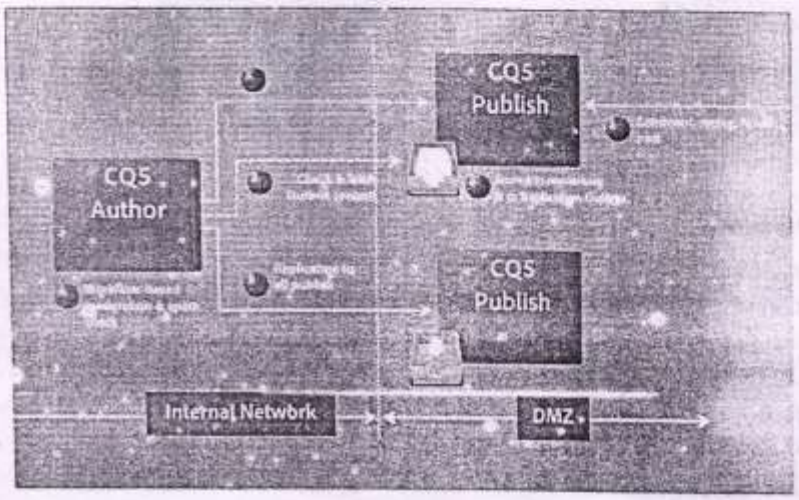
Replication

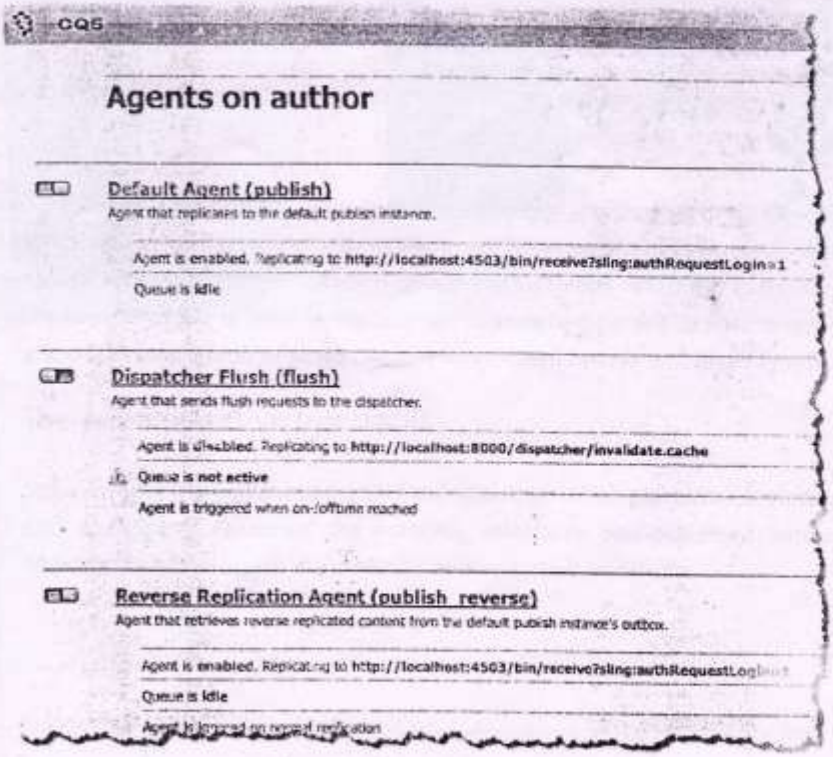
Activation/Publication of content is handled by the Replication Agents. Each Replication Agent replicates content from the current instance to one destination. So if you have 4 publish instances, you need 4 Replication Agents.

Details on creation and management of Replication Agents can be found in the documentation at:
http://dev.day.com/docs/en/cq/current/deploying/configuring_cq.html

Reverse Replication

Reverse Replication allows the transfer of content from the Web site visitors (the publish instance) to the author instance for review, moderation, and/or spam check without violating any firewall rules.

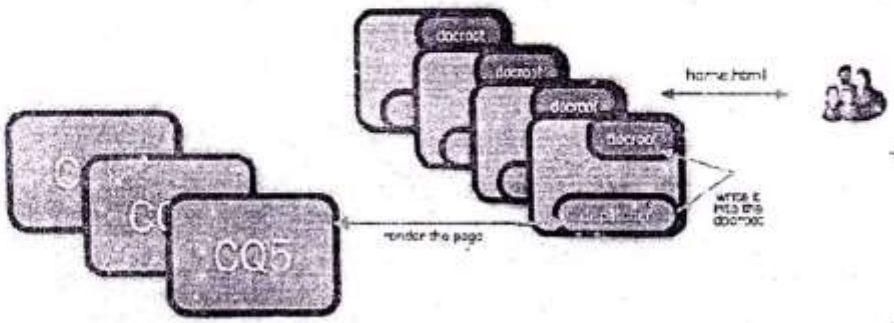




The available Replication Agents

Dispatcher

In the production delivery environment, the publish instance is joined by web server module, called the *Dispatcher*. The Dispatcher is the CQ5 caching and/or load balancing tool.



The Dispatcher helps realize an environment that is both fast and dynamic. It works as part of a static HTML server, such as Apache, with the aim of:

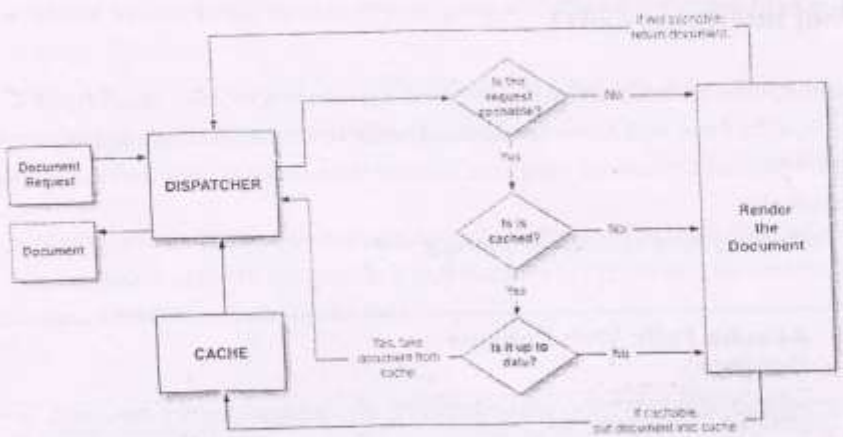
- Storing (or "caching") as much of the site content as is possible, in the form of a static website
- Accessing the layout engine as little as possible.

Which means that:

- Static content is handled with exactly the same speed and ease as on a static web server;additionally you can use the administration and security tools available for your static web server(s).
- Dynamic content is generated as needed, without slowing the system down any more than absolutely necessary.

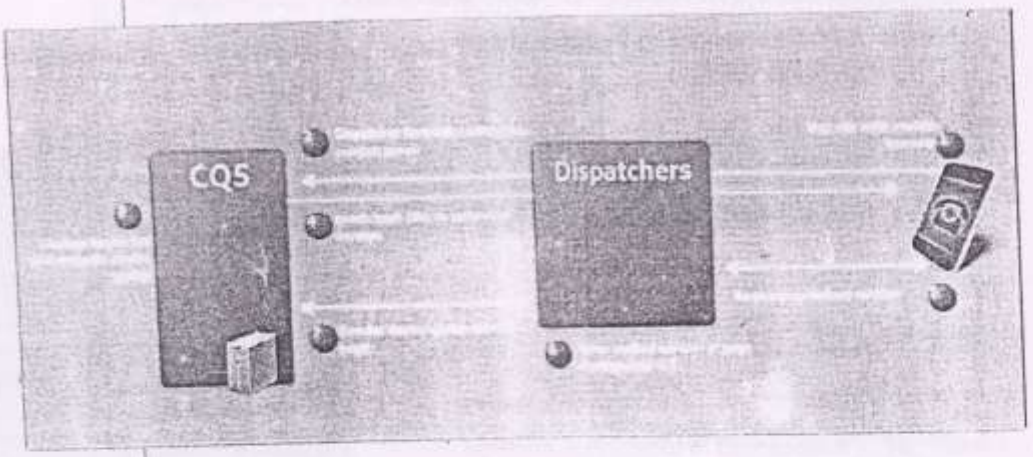
The Dispatcher contains mechanisms to generate, and update, static HTML based on the content of the dynamic site. You can specify in detail which documents are stored as static files and which are always generated dynamically. The figure below demonstrates the algorithm that the Dispatcher uses to determine whether an object should be served from the web server cache or rendered dynamically.

NOTE: If anything went wrong during the publishing process, check on the Author instance for the related Replication Agent log files usin, the configuration panel we used as we edited the settings.



Details regarding the management and configuration of the Dispatcher can be found at:
http://dev.day.com/docs/en/cq/current/deploying/dispatcher.html

The Dispatcher algorithm for mobile content has an extra round trip. Device evaluation is done with redirects and device group specific content is cachable.



The Administrative Interfaces

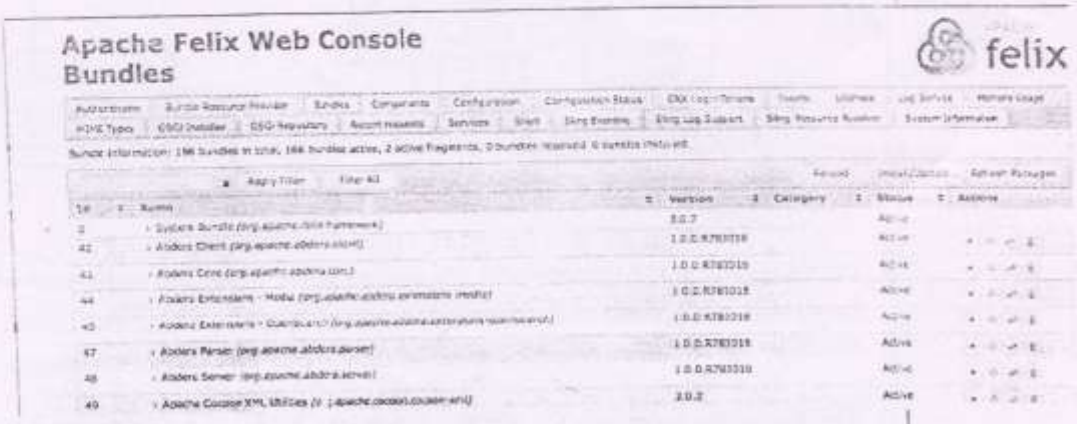
So far we have learned about the CRX and CQ5 Application platforms, installed the CQ5 application, examined the Authoring interfaces and practiced authoring. Now it is time to examine the available Administrative interfaces.

What interfaces exist?

Apache Felix Web Console

The Apache Felix Web Console is a simple tool to inspect and manage the CQ5 OSGi framework instances using your favorite Web Browser. You can access the console at:

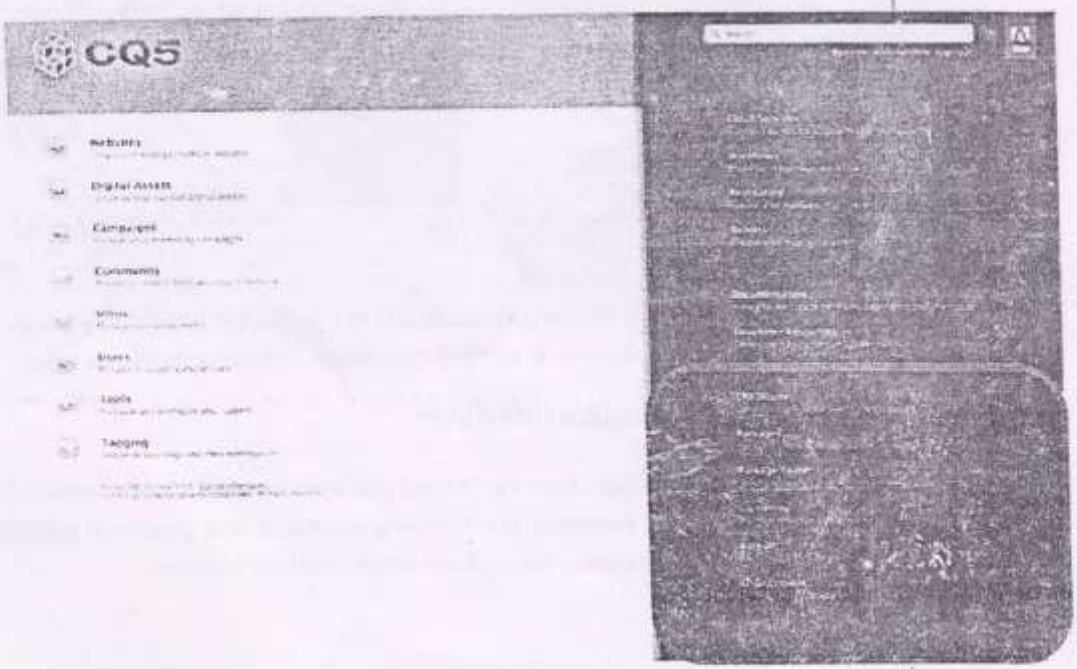
http://localhost:4502/system/console



CQ Welcome Screen Utilities

The CQ Welcome screen has links for tools and utilities that managing the repository. You can access the Welcome Screen at:

http://localhost:4502



The following CRX tools and utilities are accessible from the CQ Welcome Screen and will be useful as you develop CQ5 web applications:

- **Cloud Services** - Connect to and integrate with the Adobe Digital Marketing Suite.
- **Workflows** - Model and manage workflows The Workflow Console provides a centralized location for workflow management.
- **Replication** - Create and manage multiple websites. The replication agents allow you to activate content from one instance of CQ5 to another and keep your content fresh and up-to-date.
- **Reports** - To help you monitor and analyze the state of your instance, CQ5 provides a selection of default reports..
- **Package Manager** - The Package Manager manages the packages on your local CRX installation.
- **Package Share** - The Package Share is a centralized server made publicly available to share Content-Packages.With Package Share you can download these packages, which can include official hotfixes, feature sets, CQ5-updates or CQ5-content generated by other users.You can also upload and share packages within your company.
- **Clustering** - Administer the user and group accounts, together with the related access policies. Typically, you will not use this interface to manage CQ5 users, as the CQ5 application provides its own User/Group Administration Interface.
- **CRXDE Lite** - Browser-based Interactive Development Environment (IDE)
- **Backup** - Import/Export information into/out of the repository. The Loader will upload an XML document of defined format into the CRX repository. The Zipper will export a specified portion of the repository specified format.

CRXDE Lite

CRXDE Lite is the browser-based IDE. CRXDE Lite can be accessed at

http://localhost:4502/crx/de/

CRXDE Lite is embedded into CQ5/CRX. CRXDE Lite enables standard development tasks in a Web browser:

- Create and edit files (e.g. JSP, Java, HTML, etc.), folders, Templates, Components, Dialogs, nodes, properties, and bundles
- Log messages
- Integration with SVN

CRXDE Lite is recommended when:

- No direct access to the CQ5/CRX server
- Developing an application by extending or modifying the out-of-the-box Components and Java bundles
- No need for dedicated debugger or code completion

The CQ5 Repository Structure

The Content Explorer presents the logical structure of the repository as defined by the JCR API. This interface is going to be really useful to you as developers. You will be able to look at the structures that your code writes into the repository.

If you take a close look at the repository structure, you will notice that the repository is has several major areas:

/var - Files that change and are updated by the system; such as audit logs, statistics, event-handling.

/libs - Libraries and definitions that belong to the core of CQ5. The sub-folders in /libs represent the out of the box CQ5 features as for example search or replication. The content in /libs should not be modified as it affects the way CQ5 works. Features specific to your website should be developed under /apps.

/etc - Utilities and Tools. See the documentation for the Tools Console for detailed information.

/apps - Application related. The custom templates and component definitions specific to your website. The components that you develop may be based on the out of the box components available at /libs/foundation/components.

/content - Content created for your website.

/tmp - Temporary working area.

/home - User and Group information.

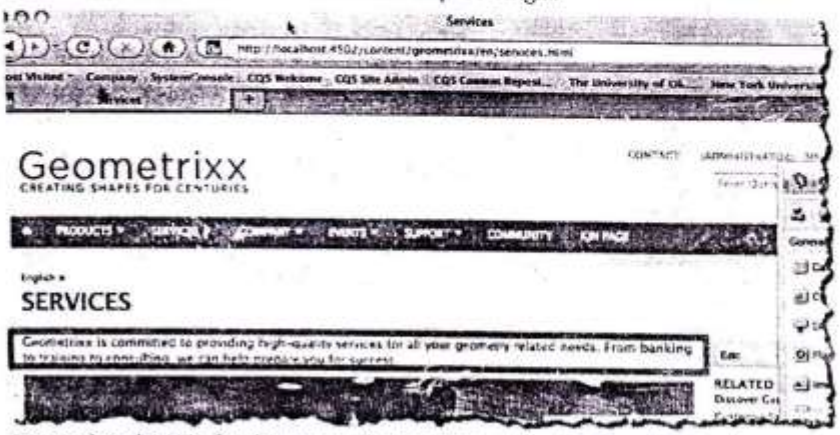
WARNING

You should not modify anything in the /libs path. To make any changes, copy the item to /apps and make the changes within the /apps path. This will make your changes safe across upgrades. Adobe will overwrite /libs during upgrades.

Website Content

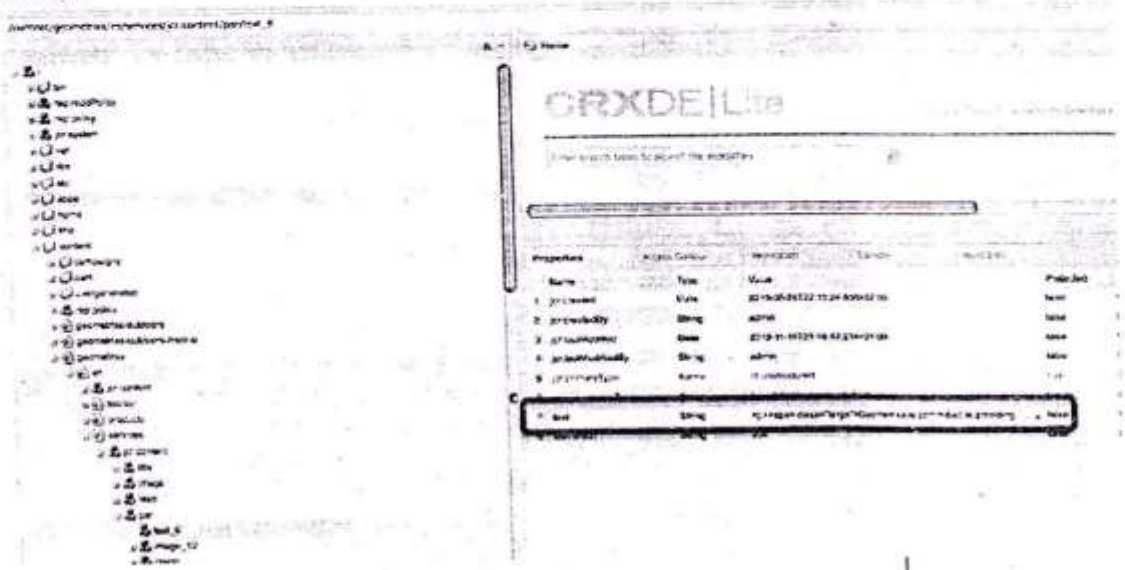
In the repository, the nodes provide the content and the properties store the data. Top fully demonstrate that concept, let's take a close look at the /content path in the repository. As mentioned above, the /content path holds your website content.

Take a look at the Services page in the Geometrixx Web site. Notice the paragraph with the text "Geometrixx is committed to providing ..."



Remember the URL for this page: */content/geometrixx/en/services.html*.

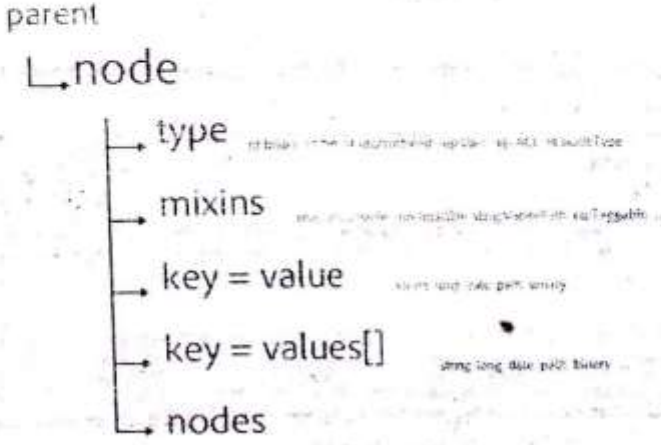
Now use CRXDE Lite to take a look a the repository structure for that page. Notice the path to the node structure that represents this page: */content/geometrixx/en/services*.



Now take a good look at the node named ***text_6*** at the path */content/geometrixx/en/services/jcr:content/par/text_6*. Examine the properties of that node. You should take special note of the value of the text property.

The value of the text property exactly matches the value of the text in the paragraph on the Services page. That is because the paragraph on the Services page is rendered from the **text_6** node and the text property holds the content itself. Again, because it is so important, in the repository, **the nodes provide the structure and the properties store the data.** In fact, if you look at the Geometrixx Web site on your system, you will not find a file named "services.html". When the page is requested by the browser, it is rendered directly from the repository. The html file will not get written until it is cached in the web server's document root.

Review of the Content Repository Structure

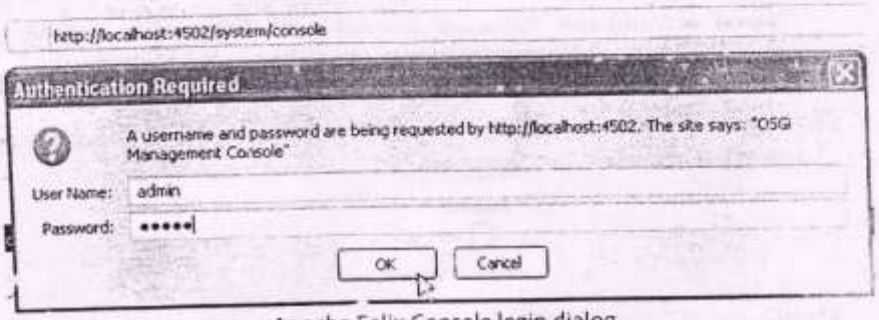


EXERCISE - Browse Related Application/Server Interfaces

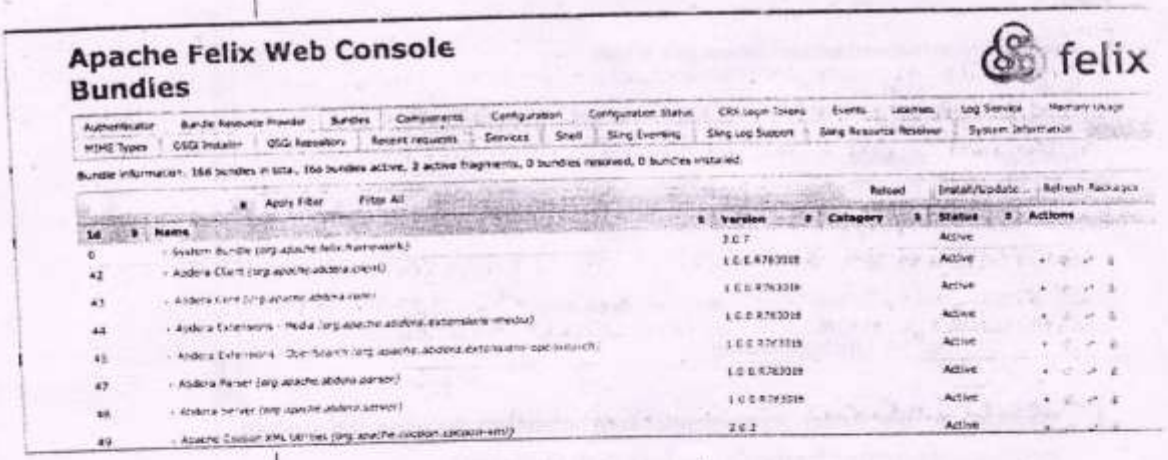
How to browse the Apache Felix interface:

1. Enter the URL http://localhost:4502/system/console in your favorite Web browser's address bar.

2. Enter the default administrator "User name" (**admin**) and "Password" (admin) in the dialog – then select **OK**.

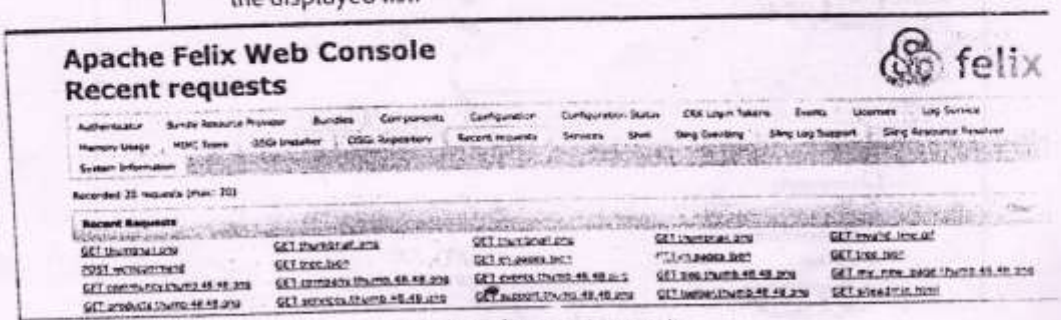


Apache Felix Console login dialog



Apache Felix Console

3. Select **Recent requests** – then select **Clear** to remove recent requests from the displayed list.



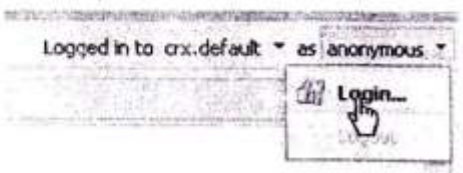
Apache Felix recent requests

Congratulations! You have successfully logged into the Felix application and have removed recent requests from the displayed list. You can use this request information to investigate the internal workings of CQ5.

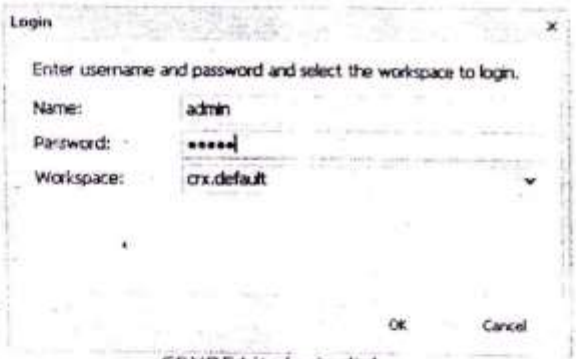
How to use CRXDE Lite:

1. Enter the URL http://localhost:4502/crx/de/ in your favorite Web browser's address bar.

2. Check the username in the upper right hand corner. If you are not logged in as the default administrator, select **Login** – then enter the default administrator "Name" (**admin**) and "Password" (**admin**) in the dialog, while continuing to use the crx.default "Workspace" – then select **OK**.

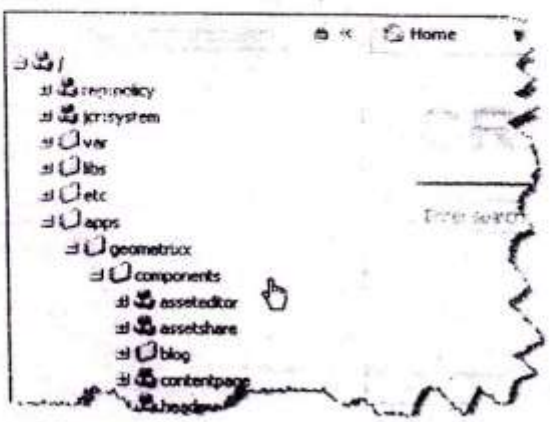


CRXDE Lite login selection



CRXDE Lite login dialog

3. Navigate to the folder */apps/geometrixx/components* to view the custom components created for the Geometrixx Web site/project.



CRXDE Lite geometrixx component display

Congratulations! You have successfully logged into CRXDE Lite and have browsed the custom components created for the Geometrixx Web site/project. Again, CRXDE Lite is embedded into CQ5/CRX and enables you to perform standard development tasks in a Web browser.

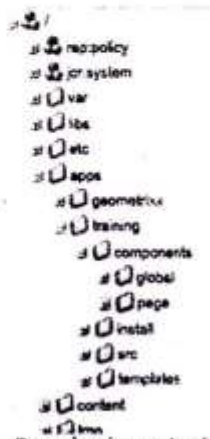
Section-3 Developing CQ5 Web Applications - First Steps

The Application/Project in the repository

An application/project is where you will store CQ5 elements such as Templates, Components, OSGi bundles, and static files. In order to start a new application/project, it is necessary to define a location for these elements. Typically, they are defined as a subfolder of the /apps folder. Adobe recommends that you create the following structure for your application/project:

Structure	Description
/apps/<application name>	The application container
/apps/<application name>/components	The Components container
/apps/<application name>/components/page	The "Page" Components container
/apps/<application name>/templates	The Templates container
/apps/<application name>/src	The OSGi bundles container
/apps/<application name>/install	The compiled OSGi bundles container

The following figure is an example of what a project might look like in the repository:



Page basic content output

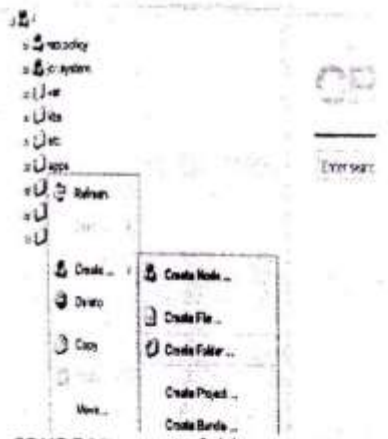
You will note the page folder under the components folder. There are 2 major types of components: page-rendering components and all the rest of the components. In your projects, you will find additional, natural component groupings; for example, if you have multiple chart components, you might create a chart folder under the components folder to hold the chart components.

BEST PRACTICE

Organize your components and templates in ways that allow you to easily maintain them.

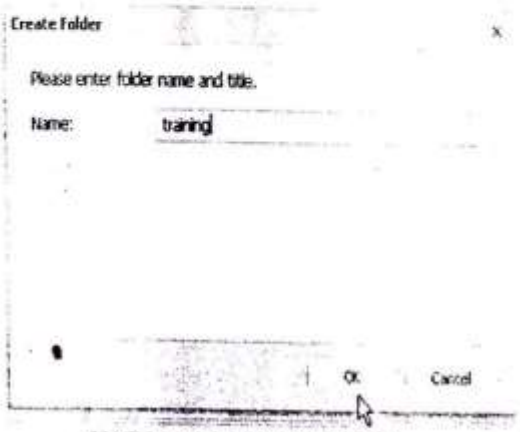
EXERCISE - Create an Application/Project

1. Using CRXDE Lite, right-click the folder under which you want to create the new folder – then select **Create ..., Create Folder ...**



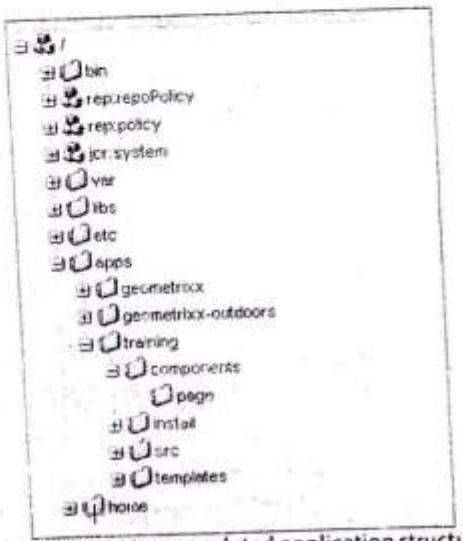
CRXDE Lite create folder selection

2. Enter the folder "Name" (*training*) you wish to create in the dialog – then select **OK**.

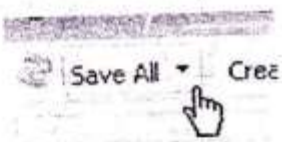


CRXDE Lite create folder dialog

3. Repeat this process until you have created Adobe's recommended application/project structure – then click **Save All**. Make sure you **Save All** before going on to the next exercise.



CRXDE Lite completed application structure



CRXDE Lite save all

Congratulations! You have successfully created an application/project and related structure in CQ5. It is not uncommon to add additional structure to the components folder, as there can be many types of custom Components (e.g. nav, header, footer, etc.). Now we are ready to start creating our Web Application.

Templates

A Template is used to create a Page and defines which components can be used within the selected scope. A Template is a hierarchy of nodes that has the same structure as the page to be created, but without any actual content. Each Template will present you with a selection of components available for use. Templates are built up of Components. Components use, and allow access to, Widgets, which are used to author/render content.

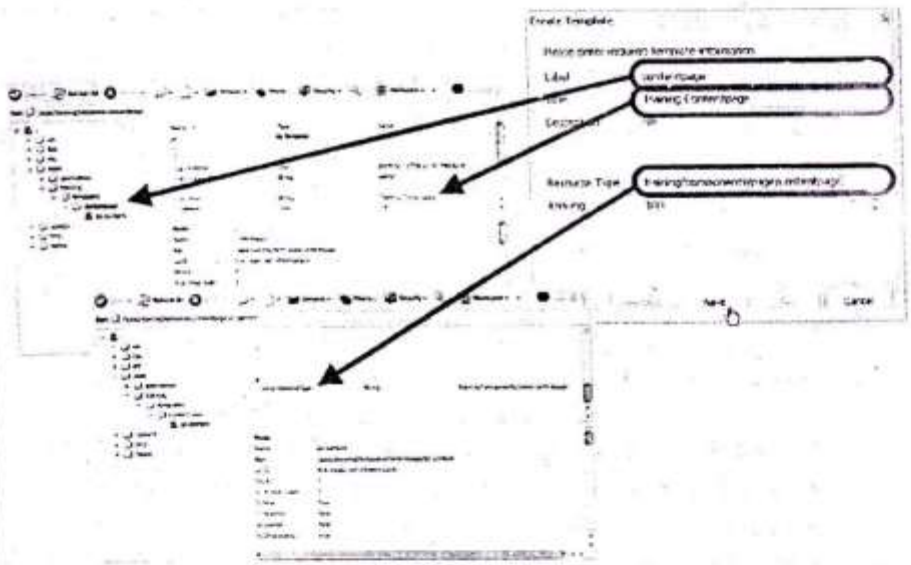
A Template is the basis of a Page. To create a Page, the Template's content must be copied (/apps/<application name>/templates/<template name>) to the corresponding position in the site-tree (this occurs automatically if page is created using CQ5).

This copy action also gives the Page its initial content and the property sling:resourceType, the path to the "Page" Component that is used to render the page.

Template in the Repository

The Create Template dialog/wizard allows you to enter the information necessary to create the complete template structure. As you can see in the figure below:

- Label - cq:Template node name
- Title - jcr:title property
- Resource Type - sling:resourceType property
- Ranking - ranking property
- Allowed Paths - allowedPaths property



You will note that the all of the properties will be defined on the cq:Template node except the sling:resourceType property, which will appear on the jcr:content child node.

The following table describes template properties and child nodes:

Name	Type	Description
.	cq:Template	Current template. A template is of node type cq:Template.
allowedChildren	String[]	Path of a template that is allowed to be a child of this template.
allowedParents	String[]	Path of a template that is allowed to be a parent of this template.
allowedPaths	String[]	Path of a page that is allowed to be based on this template.
jcr:created	Date	Date of creation of the template.
jcr:description	String	Description of the template.
jcr:title	String	Title of the template.
ranking	Long	Rank of the template. Used to display the template in the User Interface.
jcr:content	cq:PageContent	Node containing the content of the template.
thumbnail.png	nt:file	Thumbnail of the template.
icon.png	nt:file	Icon of the template.

EXERCISE - Create a Template

1. Right-click */apps/<application name>/templates* - then select **Create...**, **Create Template ...**

CRXDE Lite create template dialog

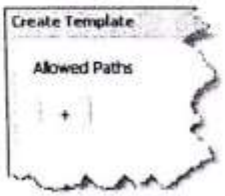
2. Enter the desired Template "Label", "Title", "Description", "Resource Type", and "Ranking" in the dialog - then select **Next**.
   - Label = the name of the Template/node that will be created
     - contentpage
   - Title (property jcr:title) = the title that will be assigned to the Template
     - Training Contentpage
   - Description (property jcr:description) = the description that will be assigned to the Template

- training project content page component

- Resource Type (property sling:resourceType) = the Component's path that will be assigned to the Template and copied to implementing pages
  - training/components/page/contentpage
- Ranking (property ranking) = the order (ascending) in which this Template will appear in relation to other Templates. Setting the Rank to 1, will ensure that our Template appears first in the list.
  - 1

NOTE: The property *sling:resourceType* will be created on the Template's *jcr:content* node. In addition, the Component may not yet exist. This is because you have not yet created it.

3. Select **Next** for **"Allowed Paths"**. Allowed Paths will define paths where this template may be used to create pages.



CRXDE Lite create template allowed paths

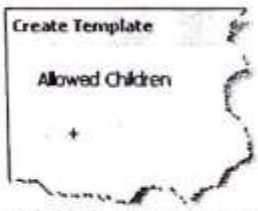
4. You will need to add a path to **Allowed Paths**. Click on the plus sign and enter the following value: */content(/.*)?*

NOTE

If you forget to enter allowedPaths before clicking on Finish, create the following property on your template node:

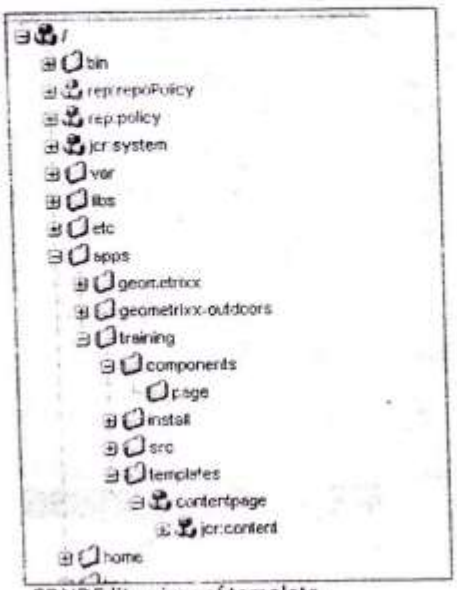
- Name: allowedPaths
- Type: String[] (String Array)
- Value: /content(/.*)?

5. Click **Next** for **"Allowed Parents"** - then select **OK** on **"Allowed Children"**.



CRXDE Lite create template allowed children

Congratulations! You have successfully created a Template in CQ5. To make you more aware of the structure associated with CQ5 objects, it is a good idea to view the nodes and properties that were created in the repository (CRX). Please see the image below.



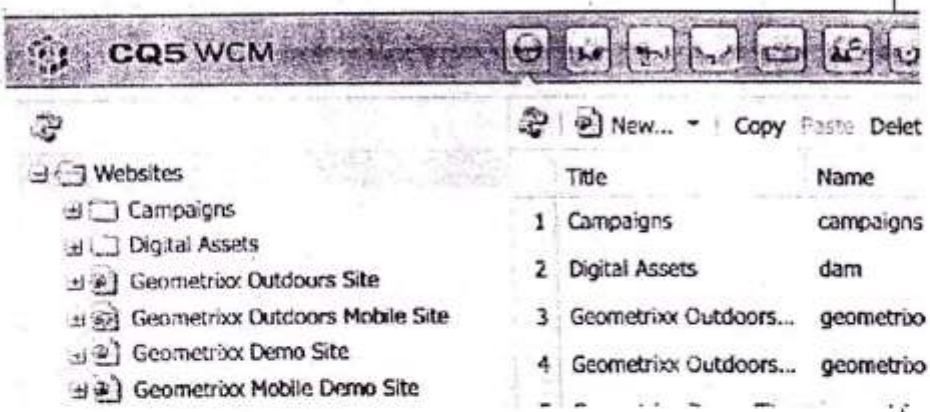
CRXDE lite view of template

NOTE: Don't forget to click **Save All** to persist your changes in the repository.

Testing your Template

Even though you do not yet have a component to render pages based on your new template, you can test the values you set in the Create Template dialog

1. Open the Site Admin Console and Select Websites.



2. Choose the New button from the toolbar.

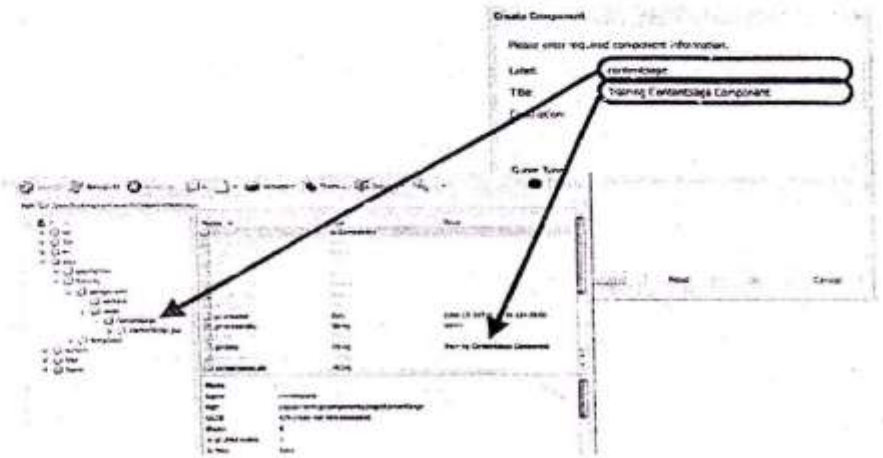
3. You should see your template in the list of available templates in the Create Page Dialog. It will be thin, as it doesn't have a thumbnail, but it should be 1st in the list.

Components

Components are modular, re-usable units that implement specific functionality/ logic to render the content of your Web site. They have no hidden configuration files, can include other Components, and can run anywhere within CQ5 or in isolation (e.g. portal). A Component could be described as a collection of scripts (e.g. JSPs, Java servlets, etc.) that completely realize a specific function. More specific, a "Page" Component is typically referenced by a Template.

The Create Component dialog/wizard allows you to enter the information necessary to create the complete component structure. As you can see in the figure below:

- Label - cq:component node name
- Title - jcr:title
- Description - jcr:description

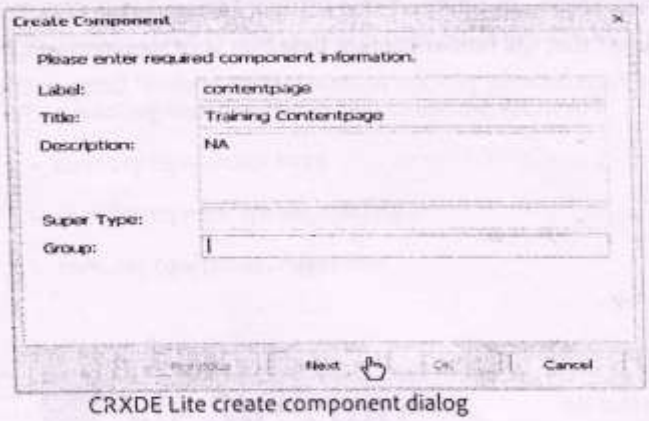


A Component could be described as a collection of scripts (e.g. JSPs, Java servlets, etc.) that completely realize a specific rendering function. In order to realize this specific functionality, it is your responsibility to create any necessary scripts that will do so. Typically, a Component ("Page" or otherwise) will have at least one default script, identical to the name of the Component (e.g. contentpage.jsp).

EXERCISE - Create a "Page-rendering" Component

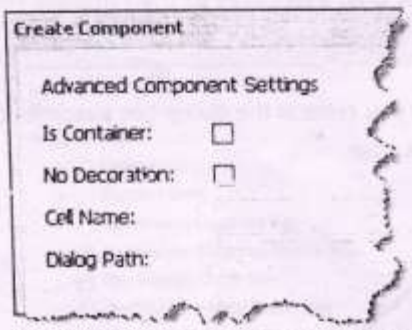
1. Right-click */apps/<application name>/components/page* - then select **Create..., Create Component ...**

2. Enter the desired Component "Label", "Title", and "Description" in the dialog - then select **Next.**
   - Label = the name of the Component/node that will be created
     - contentpage
   - Title (property jcr:title) = the title that will be assigned to the Component
     - Training Contentpage
   - Description (property jcr:description) = the description that will be assigned to the Component
     - NA



CRXDE Lite create component dialog

3. Take the defaults and select **Next** for **"Advanced Component Settings"** and **"Allowed Parents"** - then select **OK** on "Allowed Children".



CRXDE Lite create component advanced settings

Notice that the creation of the component results in the creation of the default **contentpage.jsp** script.

4. Open the script by double-clicking the script object in the left pane. The script contains some sample code that may have to be adjusted or deleted, depending on what the component will do.

5. Enter some HTML code, similar to below – then select Save All.

Contentpage.jsp

```
<html>
    <head>
        <title>Hello World !!!</title>
    </head>
    <body>
        <h1>Hello World !!!</h1>
        <h2>Welcome to a new Day</h2>
    </body>
</html>
```

Congratulations! You have successfully created a "Page" Component in CQ5. You can now create a script that will render content based on your requirements. In addition, you will perform a similar process when creating "Content" Components in the future.

NOTE: Don't forget to click Save All to persist your changes in the repository.

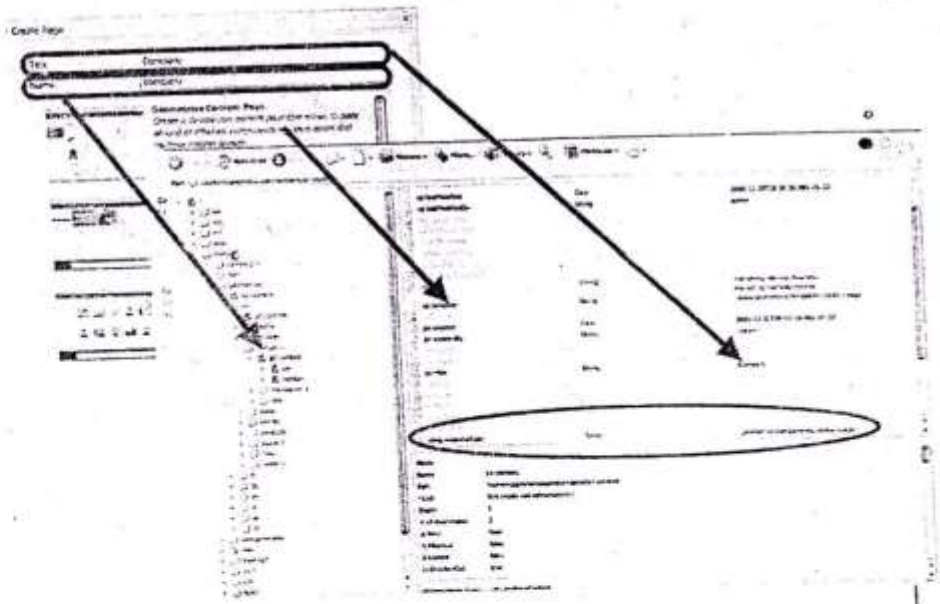
Pages

A Page is many things:

- Web site content container
- Instance of a template
- cq:Page JCR node type (has a mandatory jcr:content child node)

A Page is where content authors will create and edit content that will most likely be published and viewed by site visitors. It is an exact copy of the Template from which it was created.

When creating a page, the content that you enter in the dialog box becomes the nodes and associated properties for that page.



The New..New Page dialog/wizard allows you to enter the information necessary to create the complete page structure. As you can see in the figure below:

- Name - cq:Page node name
- Title - jcr:title property
- Template - cq:template property

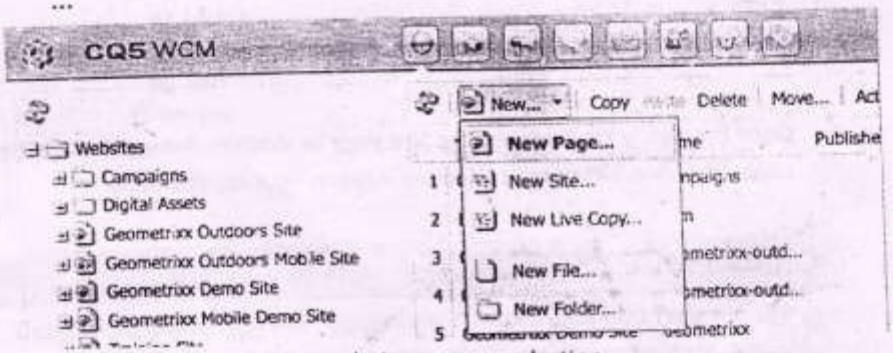
The template's sling:resourceType property gets added as the page's sling:resourceType property, so the page knows where its rendering script is.

When working with pages, it is best to use the WCM API

- com.day.cq.wcm.api.Page
- com.day.cq.wcm.api.PageManager
- com.day.cq.wcm.api.PageFilter

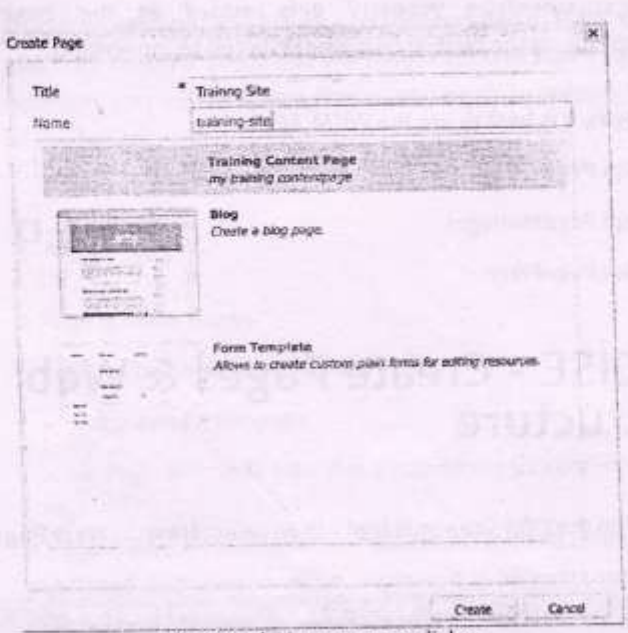
EXERCISE - Create Pages & Web Site Structure

1. Select "**Websites**" in the CQ5 Siteadmin tool - then select New ..., New Page ...



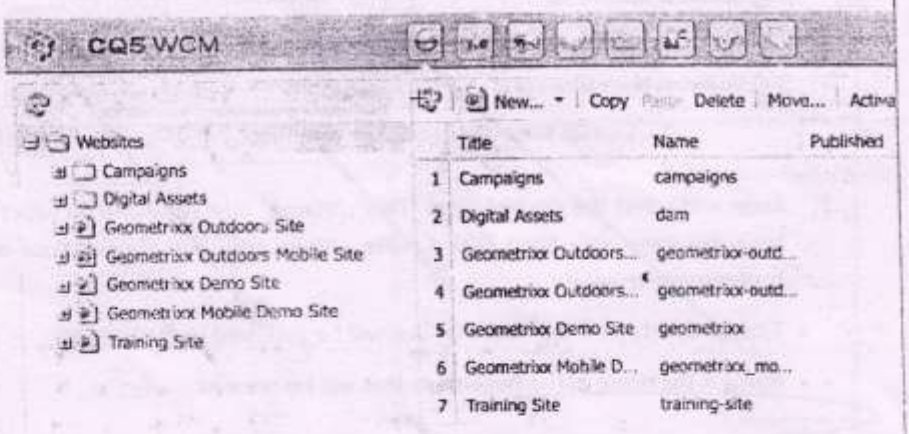
CQ5 siteadmin new page selection

2. Enter and select the desired Page "Title", "Name" and Template on which to base this Page on - then click **Create**. Make sure you choose your new contentpage template.
   - Title (property jcr:title) = the title that will be assigned to the Page
   - Name = the name of the Page/node that will be created



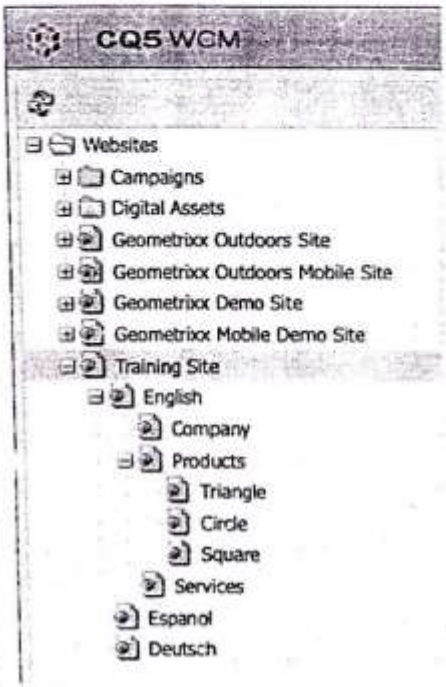
CQ5 siteadmin new page dialog

3. Open the newly created Training Site page by double-clicking it in the right pane - then view its output to ensure it meets your expectations.



CQ5 page view (e.g. /content/trainingSite)

4. Repeat this process until you have created a Web site structure similar to the image below.



CQ5 siteadmin web site structure

NOTE: Since all Pages will be created using the same Template (i.e. Training Contentpage) and implement the same rendering script, every Page will look identical - for now.

Congratulations! You have successfully created Pages and Web site structure in CQ5. This is the type of behavior you can expect from content authors when building out their Web site structure.

CRXDE

CRXDE is a pre-packaged stand-alone Eclipse application. CRXDE is custom-built specifically for CQ5 and CRX and thus enables you to efficiently develop your project. CRXDE gives you a broad set of tools to easily create and manage files, folders, Templates, Components, Dialogs, nodes, properties, scripts and OSGi bundles while logging, debugging and integrating with SVN. CRXDE is built on the Eclipse Rich Client Platform (RCP), leveraging the Eclipse File System (EFS) API.

What are the benefits of using CRXDE vs. CRXDE Lite?

As stated earlier, CRXDE Lite is recommended when you do not have direct access to the CQ5/CRX server, when you develop an application by extending or modifying the out-of-the-box Components and Java bundles, or when you do not need a dedicated debugger, code completion and syntax highlighting. CRXDE is recommended when you are developing complex applications by creating new

components and Java bundles. Again, CRXDE hides some of the complexity of the development process by allowing you to work directly with the repository without the need to synchronize the repository with the file system.

EXERCISE - Install & Start CRXDE

1. Create/identify a folder structure on your file system where you will store, install, and start CRXDE (e.g. C:/day/CQ5).

2. Copy the appropriate CRXDE package from <USB>/distribution/crxde into the folder structure.

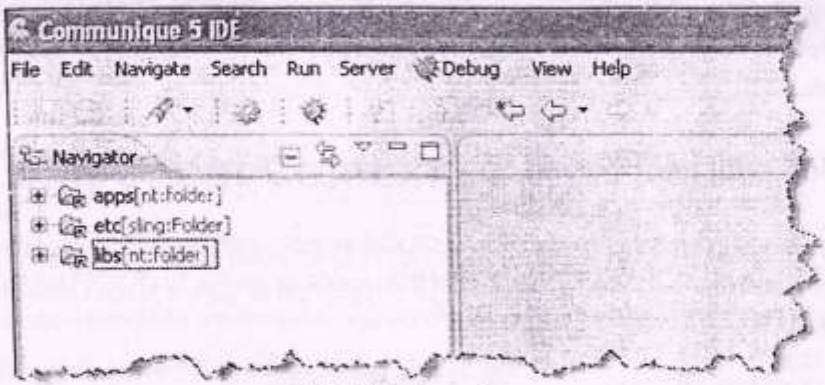
3. Extract the package.

4. Double-click the executable in the CRXDE folder.

5. Enter the "URL" (http://localhost:4502), default administrator "User Name" (admin) and "Password" (admin) in the dialog – then click OK.

CRXDE login dialog

Congratulations! You have successfully installed and started CRXDE. Here you will be able to create and modify CQ5 elements such as Templates, Components, scripts, etc. In addition, traditional IDE tools such as code completion and debugging will now be available to you. Below is a view of what CRXDE looks like after starting.



CRXDE interface view

EXERCISE - Utilize CRXDE

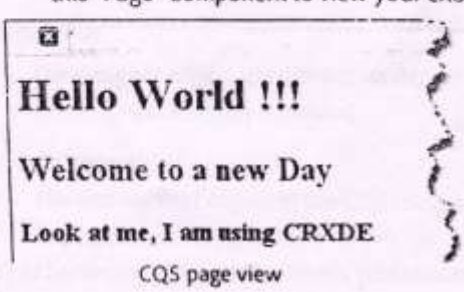
1. Navigate to and open, by double-clicking, the ***contentpage.jsp*** script you created earlier for the Training Contentpage "Page" Component (e.g. /apps/training/components/page/contentpage).

CRXDE view of contentpage.jsp

NOTE: Traditional keyboard shortcuts will work in CRXDE (e.g. Ctrl-S to save, Ctrl-C to copy, etc.).

2. Edit the script to your liking – then click **File, Save**.

3. In the CQ5 Siteadmin tool, navigate to and open any page that is based off of this "Page" Component to view your changes.



CQ5 page view

Congratulations! You have successfully modified a script using CRXDE. For the rest of this training, you will use CRXDE to accomplish more complex development tasks.

Additional Information

You will notice that the tree browser in CRXDE includes only /apps, /etc, and /libs. If you want to see additional branches of the repository, similar to CRXDE Lite, you can use the CRX Content Explorer or CRXDE Lite to export the additional nodes to CRXDE.

How to modify the set of repository paths exported to CRXDE using the CRX Content Explorer:

1. Navigate to http://localhost:4502/crx/explorer
2. Login if you have not already logged in.
3. Select the Content Explorer.
4. In the Content Explorer, navigate to */etc/crxde/profiles/default*
5. Modify the *crxde:paths* property by adding desired paths. (e.g., */content*)

6. Click the green check mark.
7. Save All.

Congratulations! You have successfully modified the paths exported to CRXDE.

You can also add Java API documentation, for CQ5 APIs and your own APIs, to CRXDE. Simply define the location of the javadocs to CRXDE using the Package Explorer pane.

Component Context

When you develop the JSP script of a CQ5 component, it is required to include the following code at the top of the script:

```
<%@include file="/libs/foundation/global.jsp"%>
```

The Adobe provided *global.jsp* script declares the Sling, CQ5 and JSTL taglibs and exposes the regularly used scripting objects defined by the

```
<cq:defineObjects /> tag.
```

This shortens and simplifies the JSP code of your component.
The <cq:defineObjects> tag exposes the following, regularly used, scripting objects which can be referenced by the developer. It also exposes the objects defined by the <sling:defineObjects> tag.

- ComponentContext
  - The current component context object of the request (com.day.cq.wcm.api.components.ComponentContext interface).
- Component
  - The current CQ5 component object of the current resource (com.day.cq.wcm.api.components.Component interface).
- CurrentDesign
  - The current design object of the current page (com.day.cq.wcm.api.designer.Design interface).
- CurrentPage
  - The current CQ5 WCM page object (com.day.cq.wcm.api.Page interface).
- CurrentNode
  - The current JCR node object (javax.jcr.Node interface).
- CurrentStyle
  - The current style object of the current cell (com.day.cq.wcm.api.designer.Style interface).
- Designer
  - The designer object used to access design information (com.day.cq.wcm.api.designer.Designer interface).
- EditContext
  - The edit context object of the CQ5 component (com.day.cq.wcm.api.components.EditContext interface).

- PageManager
  - The page manager object for page level operations (com.day.cq.wcm.api.PageManager interface).
- PageProperties
  - The page properties object of the current page (org.apache.sling.api.resource.ValueMap).
- Properties
  - The properties object of the current resource (org.apache.sling.api.resource.ValueMap).
- Resource
  - The current Sling resource object (org.apache.sling.api.resource.Resource interface).
- ResourceDesign
  - The design object of the resource page (com.day.cq.wcm.api.designer.Design interface).
- ResourcePage
  - The resource page object (com.day.cq.wcm.api.Page interface).

BEST PRACTICE

Create your own equivalent of *global.jsp* and place it in a part of the /apps structure that is accessible by all your projects. Inside of *global.jsp* you will import your commonly used classes, declare your own taglibs, and include our *global.jsp*.

EXERCISE - Include the "global.jsp" in the Page Component

1. Open the *contentpage.jsp* script by double-clicking it.
2. Enter the include statement in your JSP, similar to below – then **Save**.

contentpage.jsp

```
<%@include file="/libs/foundation/global.jsp"%>
<html>
    <head>
        <title>Hello World !!!</title>
    </head>
    <body>
        <h1>Hello World !!!</h1>
        <h2>Welcome to a new Day</h2>
        <h3>Look at me, I am using CRXDE</h3>
    </body>
</html>
```

3. Test your script by requesting a Page in CQ5 Siteadmin that implements this "Page" Component.
   - If successful, you should not notice any difference in the way the Page is rendered

Congratulations! You have successfully included the global.jsp, allowing you access to numerous Sling, CQ5, and Java objects to aid you in your development efforts.

Using the APIs to Display Basic Page Content

This is an introduction of how to render content that lives in the repository, which is a similar concept to querying and displaying data from a database. As we discussed previously, in the repository, the nodes define the structure and the properties hold the data. In order to render content on any page, we need to render the data from those properties. We will start with the basic properties associated with every page.

After inclusion of the *global.jsp* in whatever script you are working on, there are generally three ways to access content in CQ5:

- Via the properties object
  - The properties object is an instance of the ValueMap (see Sling API) class and contains all properties of the current resource. For example:

```
...
String pageTitle = properties.get("jcr:title", "NO TITLE");
...
```

- Via the currentPage object
  - The currentPage object is an instance of the Page (see CQ5 API) class, which provides some methods to access content. For example:

```
...
String pageTitle = currentPage.getTitle();
...
```

- Via currentNode object
  - The currentNode object is an instance of the Node (see JCR API) class, which provides access to content via the getProperty() method. For example:

```
...
String pageTitle = currentNode.getProperty("jcr:title").getString();
...
```

NOTE: Adobe does not recommend using the JCR API directly in CQ5 unless necessary. Since CQ5 is a Sling application, you should deal with resources and not JCR nodes.

EXERCISE - Display Basic Page Content

1. Open the *contentpage.jsp* script.

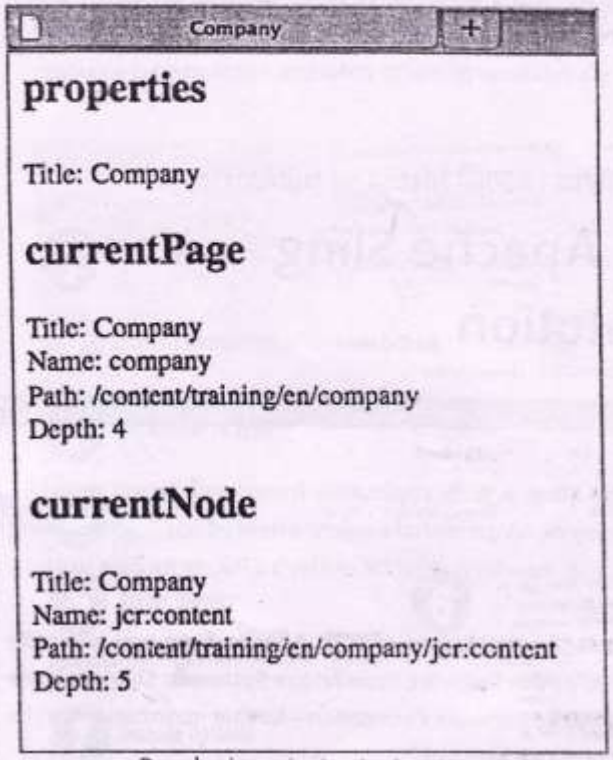
2. Enter some JSP and HTML code, similar to below – then Save.

contentpage.jsp

```
<%@include file="/libs/foundation/global.jsp"%>
<html>
    <head>
        <title><%= currentPage.getTitle() == null ? currentPage.getName() :
currentPage.getTitle() %></title>
    </head>
    <body>
        <h2>properties</h2>
        Title: <%= properties.get("jcr:title") %><br />

        <h2>currentPage</h2>
        Title: <%= currentPage.getTitle() %><br />
        Name: <%= currentPage.getName() %><br />
        Path: <%= currentPage.getPath() %><br />
        Depth: <%= currentPage.getDepth() %><br />

        <h2>currentNode</h2>
        Title: <%= currentNode.getProperty("jcr:title").getString() %><br />
        Name: <%= currentNode.getName() %><br />
        Path: <%= currentNode.getPath() %><br />
        Depth: <%= currentNode.getDepth() %><br />
    </body>
</html>
```

3. Test your script by requesting a Page in CQ5 Siteadmin that implements this "Page" Component.

   - If successful, you should now see content related to the requested Page being displayed dynamically, similar to the image below.



Page basic content output

Congratulations! You have successfully displayed content related to a Page dynamically. This is a monumental step as this is a large portion of what you do as a developer - display content.

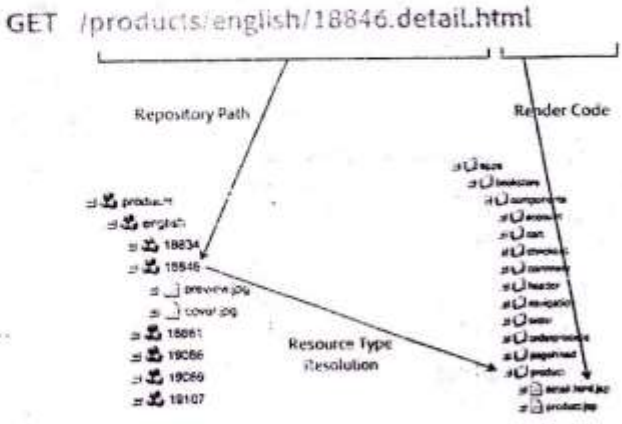
Section 4 - Apache Sling Script Resolution

CQ5 is built using Apache Sling, a Web application framework based on REST principles that provides easy development of content-oriented applications. Sling uses a JCR repository, such as Apache Jackrabbit or Day's CRX, as its data store.

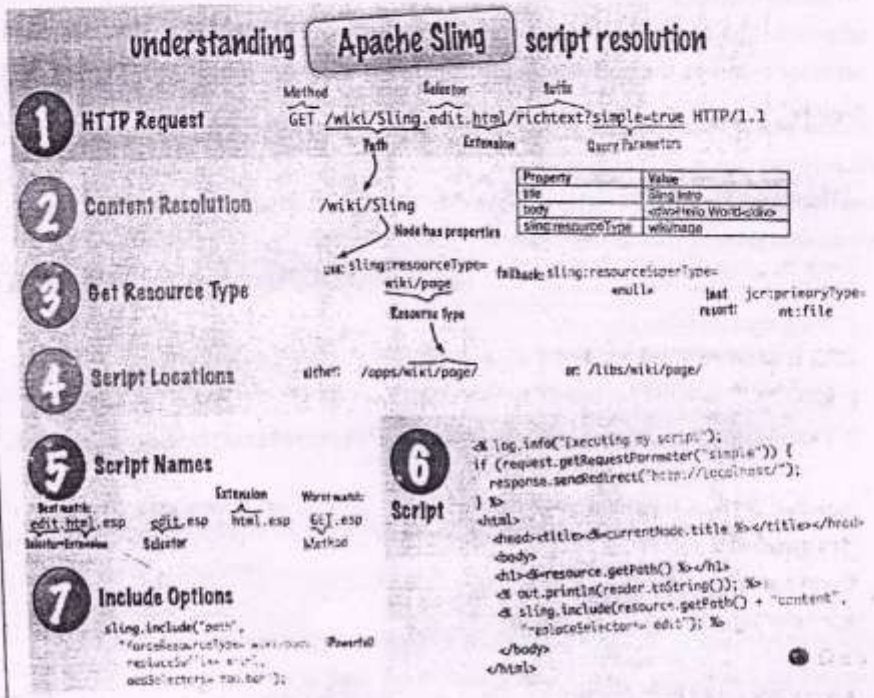
Apache Sling is included in the installation of CQ5. Apache Sling was originally designed and implemented by Day Software (now Adobe Systems). Sling has since been contributed to the Apache Software Foundation - further information can be found at Apache (http://sling.apache.com).

Using Apache Sling, the type of content to be rendered is not the first processing consideration. Instead the main consideration is whether the URL resolves to a content object for which a script can then be found to perform the rendering. This provides excellent support for Web content authors to build Pages which are easily customized to their requirements.

The advantages of this flexibility are apparent in applications with a wide range of different content elements, or when you need Pages that can be easily customized/ viewed differently.

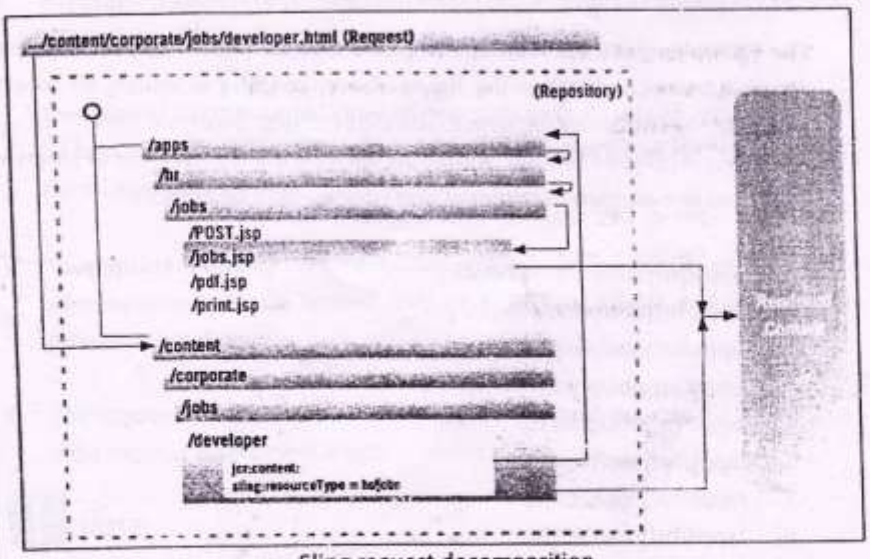


The following diagram explains the Sling script resolution. It shows how to get from HTTP request to content node, from content node to resource type, from resource type to script and what scripting variables are available.



Apache Sling cheat sheet - side 1

The following diagram illustrates another example of how a script is resolved.

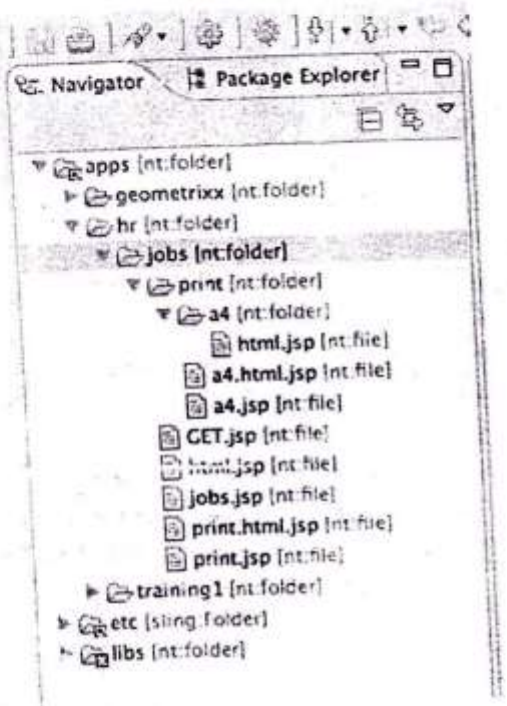


Sling request decomposition

With Sling, you specify which script renders a certain entity (by setting the sling:resourceType property in the jcr:content node). This mechanism offers more

freedom than one in which the script accesses the data entities (as an SQL statement in a PHP script would do) as a resource can have several renditions.

If multiple scripts apply for a given request, the script with the best match is selected. The more specific a match is, the better it is; in other words, the more selector matches the better, regardless of any request extension or method name match.



For example, given the /content structure defined in the last example and the /apps structure defined in the figure above, consider a request to access the resource */content/corporate/jobs/developer.print.a4.html* of type sling:resourceType="hr/jobs". Assuming we have the following list of scripts in the locations shown, then the order of preference would be as shown:

1. /apps/hr/jobs/print/a4.html.jsp
2. /apps/hr/jobs/print/a4/html.jsp
3. /apps/hr/jobs/print/a4.jsp
4. /apps/hr/jobs/print.html.jsp
5. /apps/hr/jobs/print.jsp
6. /apps/hr/jobs/html.jsp
7. /apps/hr/jobs/jobs.jsp
8. /apps/hr/jobs/GET.jsp

From this it can be seen that:

- **Folders (i.e. nodes of type nt:folder) take precedence over jsp file names when resolving using selectors, at least for the first selector.**

NOTE: Selectors can also be used as parameters passed in to the scripts.

- Only one selector in a file name has any effect - any files with names containing two selectors don't ever get selected, but names of folders can be used to match the selectors in the request.
- Scripts with HTTP method names (e.g.,GET.jsp) is selected as a last resort, (after the default script: *jobs.jsp*, in this case).

NOTE

The previous example showing the Apache Sling script precedence should not be taken as an example of good script usage. It is presented as a mechanism to demonstrate Sling's preference when it chooses scripts. Typically, you will have 1-4 rendering options; e.g., web, print/text, maybe mobile, and maybe a defined external application format choice.

The resolution process

The Servlet Resolution Process four elements of a SlingHttpServletRequest:

1. The **resource type** as retrieved through `request.getResource().getResourceType()`.

   Because the resource type may be a node type such as nt:file, the resource type is mangled into a path by replacing any colons contained to forward slashes. Also, any backslashes contained are replaced to forward slashes. This should give a relative path. Of course a resource type may also be set to an absolute path.

2. The **request selectors** as retrieved through `request.getRequestPathInfo().getSelectorString()`.

   The selector string is turned into a relative path by replacing all separating dots by forward slashes. For example the selector string print.a4 is converted into the relative path print/a4.

3. The **request extension** as retrieved through `request.getRequestPathInfo().getExtension()` if the request method is GET or HEAD and the request extension is not empty.

4. The **request method name** for any request method except GET or HEAD or if the request extension is empty.

NOTE

When dealing with multiple selectors, it can often be simplified:
Not creating multiple folders matching selectors, instead use one single jsp, getting all the selectors on the request by using SlinghttpServletRequest.ger-

RequestPathInfo.getSelectors() and then in that one jsp using a series of if.. else
if (selector1.equals("xxx")) then....
else if (selector2.equals("yyy")) then...
else if (selector3.equals("zzz")) then...
etc.. etc.. Creating one single jsp to test all the different cases

Examples:

URI	Content Path	Selectors	Extension	Suffix
/a/b	/a/b	null	null	null
/a/b.html	/a/b	null	html	null
/a/b.s1.html	/a/b	s1	html	null
/a/b.s1.s2.html	/a/b	s1.s2	html	null
/a/b/c/d	/a/b	null	null	/c/d
/a/b.html/c/d	/a/b	null	html	/c/d
/a/b.s1.html/c/d	/a/b	s1	html	/c/d
/a/b.s1.s2.html/c/d	/a/b	s1.s2	html	/c/d
/a/b/c/d.s.txt	/a/b	null	null	/c/d.s.txt
/a/b.html/c/d.s.txt	/a/b	null	html	/c/d.s.txt
/a/b.s1.html/c/d.s.txt	/a/b	s1	html	/c/d.s.txt
/a/b.s1.s2.html/c/d.s.txt	/a/b	s1.s2	html	/c/d.s.txt

NOTE: a request without a suffix doesn't get cached /a/b.s1.html/s/d won't get cached, but/a/b.s1.html/s/d.txt will be cached.

The resource type is used as a (relative) parent path to the Servlet while the *request extension* or *request method* is used as the Servlet (base) name. The Servlet is retrieved from the Resource tree by calling the `ResourceResolver.getResource(String)` method which handles absolute and relative paths correctly by searching relative paths in the configured search path.

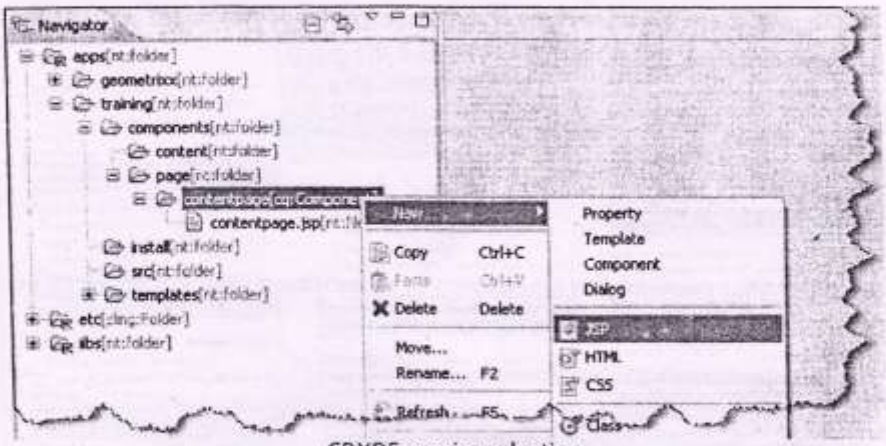
EXERCISE - Create Multiple Scripts/Renderers for the "Page" Component

1. Right-click /apps/<application name>/components/page/contentpage -- then select New, JSP.

How to create multiple scripts/renderers:

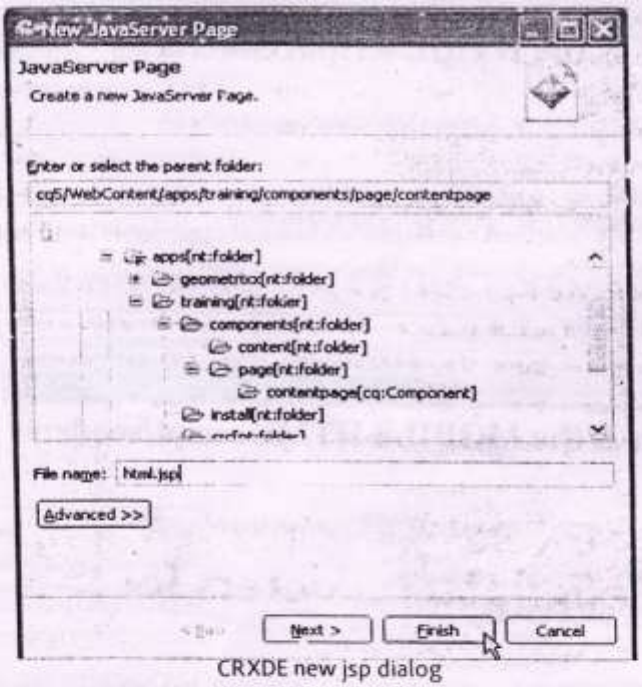
1. Right-click /apps/<application name>/components/page/contentpage -- then select New, JSP.

NOTE: The newly created script will pop-up in the right pane.



CRXDE new jsp selection

2. Enter the desired file "File name" (**html.jsp**) in the dialog - then select **Finish**.



CRXDE new jsp dialog

3. Enter some HTML code, similar to below - then **Save**.

html.jsp

```
<html>
    <head>
        <title>Hello World !!!</title>
    </head>
    <body>
        <h1>This is the HTML script/renderer </h1>
    </body>
</html>
```

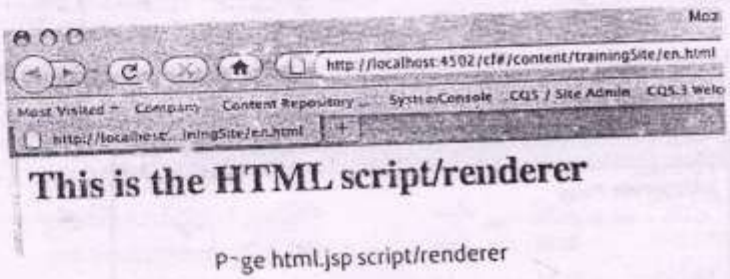
4. Repeat these processes for a script/renderer named *m.html.jsp*. Change the text in the *m.html.jsp* script to read

```
<h1>This is the MOBILE HTML script/renderer </h1>
```

Remember to Save.

5. Test your multiple scripts/renderers by requesting a Page in CQ5 Siteadmin that implements this "Page" Component.

   - /content/trainingSite/en/company.html



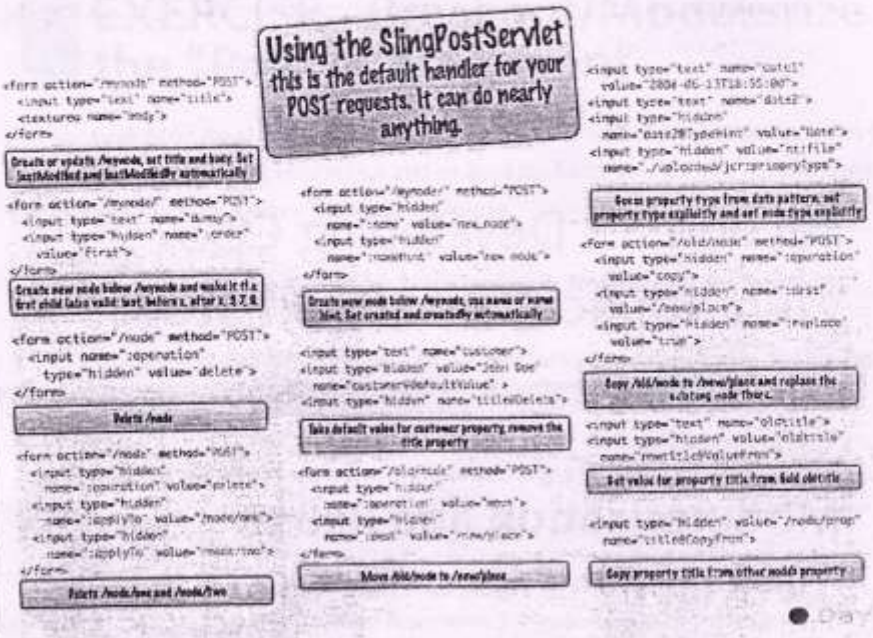
Page html.jsp script/renderer

   - /content/trainingSite/en/company.m.html

Page m.html.jsp script/renderer

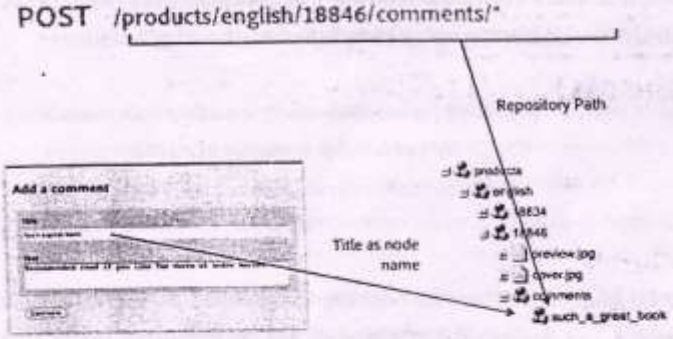
SlingPostServlet

The following diagram explains all the hidden, but powerful request parameters you can use when dealing with the SlingPostServlet, the default handler for all POST requests that gives you endless options for creating, modifying, deleting, copying and moving nodes in the repository.



Apache Sling cheat sheet - side 2

All Apache Sling scripts are stored in subfolders of either /apps or /libs, which will be searched in this order: /apps first and the /libs. In this way, you can override any code shipped in /libs simply by placing the code in the same path in /apps.



Congratulations! You have successfully created multiple scripts/renderers, potentially allowing you to display the same content in two different views.

Section 5 - Developing CQ5 Web Applications - Next Steps

Modularization and Reuse

It is important to modularize a Component into multiple scripts and include them at runtime, promoting Component/script reuse. To support modularization, we will use different methods to include scripts.

What's the difference between a JSP, CQ5, and Sling include?

The <%@ include file="myScript.jsp" %> directive informs the JSP compiler to include a complete file into the current file - at compilation time. It is as if the contents of the included file were pasted directly into the original file.

The <cq:include script="myScript.jsp" /> and <sling:incude resource="%=par%/> directives are different in that they include the resource at runtime.

Often the question is asked: "Should I use <cq:include> or <sling:include>?"

When developing CQ5 components, Adobe recommends that you use <cq:include /> tag. This allows you to directly include script files by their name when using the script attribute.

This takes component and resource type inheritance into account, and is often simpler than strict adherence to Sling's script resolution using selectors and extensions.

NOTE: The resolution of the resource and the script that are included with the <sling:include> tag is the same as for a normal sling URL resolution. By default, the selectors, extension, etc. from the current request are used for the included script as well

EXERCISE - Breakout/Modularize the "Page" Component

1. Because *html.jsp* is the best match for URLs with an *html* extension, remove (or rename) the scripts *html.jsp* and *m.html.jsp* from your "Page" Component by right-clicking them – then select **Delete**.

2. Create a new JSP file named ***body.jsp*** in your "contentpage" Component.

3. Cut the body code from *contentpage.jsp* and paste it in to *body.jsp* – then **Save**.

body.jsp

```
<%@include file="/libs/foundation/global.jsp"%>
<body>
    <h2>properties</h2>
    Title: <%= properties.get("jcr:title") %><br />

    <h2>currentPage</h2>
    Title: <%= currentPage.getTitle() %><br />
    Name: <%= currentPage.getName() %><br />
    Path: <%= currentPage.getPath() %><br />
    Depth: <%= currentPage.getDepth() %><br />

    <h2>currentNode</h2>
    Title: <%= currentNode.getProperty("jcr:title").getString() %><br
/>
    Name: <%= currentNode.getName() %><br />
    Path: <%= currentNode.getPath() %><br />
    Depth: <%= currentNode.getDepth() %><br />
</body>
```

NOTE: Notice the fact the *global.jsp* file has been included at the top of this script. Since this script will be included at runtime, it will be unaware of any previous includes of the *global.jsp*.

4. Open the file contentpage.jsp, and enter some JSP and HTML code, similar to below, replacing the <body> section – then Save.

contentpage.jsp

```
<%@include file="/libs/foundation/global.jsp"%>
<html>
    <head>
        <title><%= currentPage.getTitle() == null ? currentPage.
getName() : currentPage.getTitle() %></title>
    </head>

    <cq:include script="body.jsp" />
</html>
```

5. Test your script by requesting a Page in CQ5 Siteadmin that implements this "Page" Component.
   - If successful, you should see no difference between what is displayed now, and what was displayed before.

Congratulations! You have successfully broken out a portion of your "Page" Component, and have included a script at runtime. Again, this type of technique will allow you to reuse scripts for different Components in the future.

Initialize the WCM

The WCM initialization script performs all necessary steps to provide the whole WCM functionality on a page, including: Dialogs, Widgets, WCM CSS & JS, Sidekick, etc.

EXERCISE - Initialize the WCM

1. Open the file **contentpage.jsp**, and enter some JSP code, similar to below, adding to the <head> section an include of the initialization script - then **Save**.

contentpage.jsp

```
<%@include file="/libs/foundation/global.jsp"%>
<html>
    <head>
        <title><%= currentPage.getTitle() == null ? currentPage.
getName() :
currentPage.getTitle() %></title>
    <cq:include script="/libs/wcm/core/components/init/init.jsp"/>
    </head>

    <cq:include script="body.jsp" />
</html>
```

2. Test your script by requesting a Page in CQ5 Siteadmin that implements this "Page" Component.
   - If successful, you should now see the Sidekick appear, which will give authors the ability to manage content.

NOTE: Some WCM functionality such as the Page properties (no Dialog defined yet) may not work when only the initialization script is included. There are steps missing which we perform in a later exercise.

properties

Title: Company

currentPage

Title: Company
Name: company
Path: /content/training-site/en/company
Depth: 4

currentNode

Title: Company
Name: jcr:content
Path: /content/training-site/en/company/jcr
Depth: 5

Page view with WCM initialization

NOTE: For developers of previous CQ5 versions, you will notice that the init.jsp has changed to take advantage of the new <cq:includeClientLib> tag.

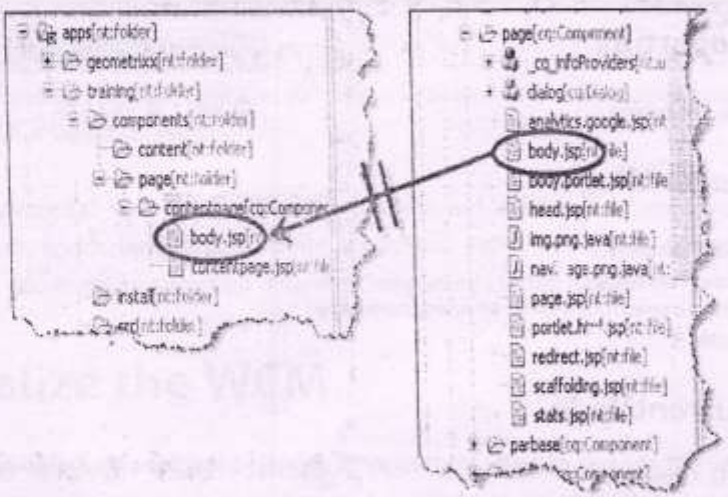
Congratulations! You have successfully initialized the WCM, and all of its related tools. Again, this initialization is critical in enabling numerous CQ5 tools, allowing content authors to work more efficiently.

Component Hierarchy and Inheritance

Components can be given a hierarchical structure to implement the inheritance of included script files, Dialogs, etc. Therefore it is possible for a specific "Page" Component (or any Component) to inherit from a "base" Component. For example, allowing inheritance of a script file for a specific part of the page (e.g. the <head> section).

Components within CQ5 are subject to 3 different hierarchies:

- Resource Type Hierarchy
  - This is used to extend components using the property sling:resourceSuperType. This enables the Component to inherit from a "base" Component. For example, a text Component will inherit various attributes from the foundation text component, including:
    - scripts (resolved by Sling)
    - Dialogs
    - Descriptions (including thumbnail images, icons, etc.)
- Important to note is the fact that a local copy/instance of a Component element (e.g. *body.jsp*) will take precedence over an inherited element.



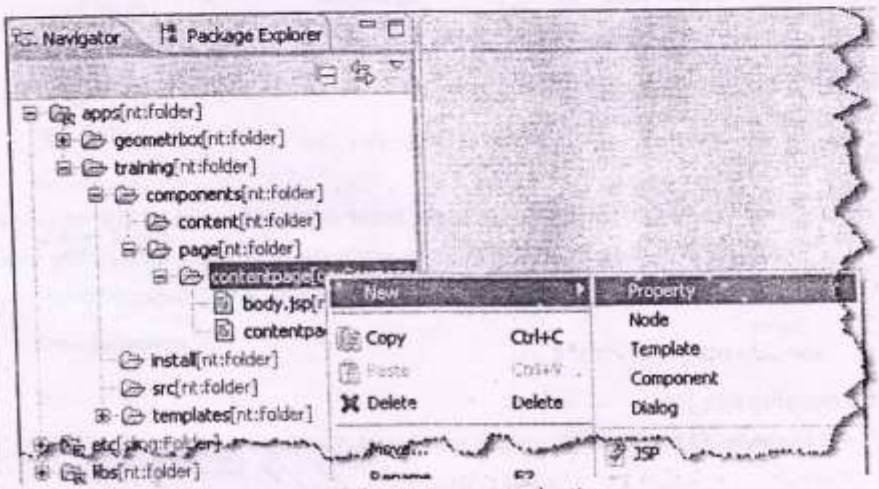
Non-inheritance of local copy

- Container Hierarchy
  - This is used to populate configuration settings to the child component and is most commonly used in a paragraph system scenario. For example, configuration settings for the edit bar buttons, control set layout (editbars, rollover, etc.), Dialog layout (inline, floating, etc.) can be defined on the parent Component and propagated to the children Components.
  - Configuration settings (related to edit functionality) in cq:editConfig and cq:childEditConfig are propagated.

- Include Hierarchy
  - This is imposed at runtime by the sequence of includes.
  - This hierarchy is typically used by the Designer, which in turn acts as the base for various design aspects of the rendering; including layout
  - information, CSS information, the available components in a paragraph system, etc.

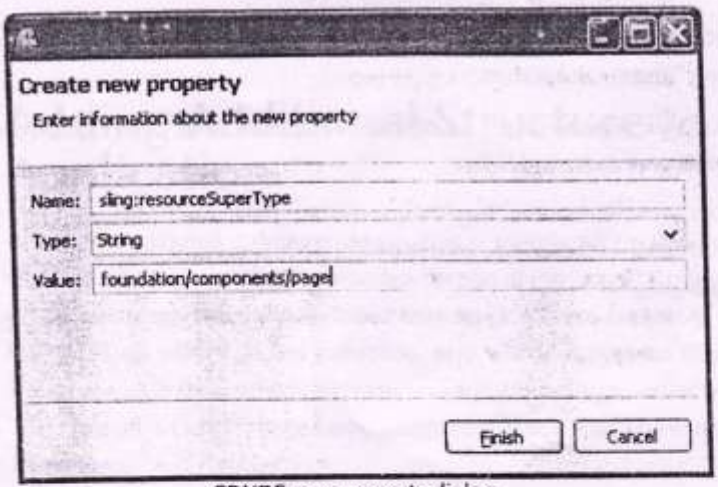
EXERCISE - Extend the Foundation Page Component

1. Right-click the Training contentpage component - then select **New, Property**.

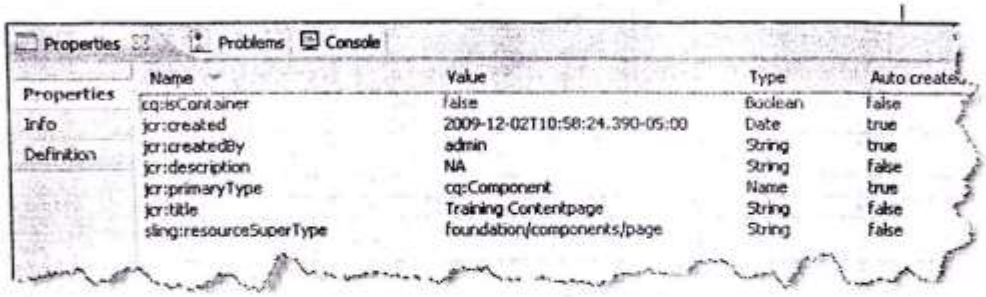


CRXDE new property selectionn

2. Enter the property "Name" (*sling:resourceSuperType*), "Type" (*String*), and "Value" (*foundation/components/page*) in the dialog - then click **Finish**.



CRXDE new property dialog



CRXDE properties view

3. Open the file **contentpage.jsp**, and enter some JSP code, similar to below, removing the <head> section and use <cq:include \> tag to include the head.jsp script that we will inherit from the foundation "Page" Component - then Save.

contentpage.jsp

```
<%@include file="/libs/foundation/global.jsp"%>
<html>
    <cq:include script="head.jsp" />

    <cq:include script="body.jsp" />
</html>
```

4. Test your script by requesting a Page in CQ5 Siteadmin that implements this Training "Page" Component.
   - If successful, you should see no difference between what is displayed now, and what was displayed before.
   - You can also see that the "Page Properties" functionality is now available in your Sidekick, since we inherited the Dialog from the foundation "Page" Component.

NOTE: You can view this property, and others directly related to the Training "Page" Component, by simply selecting and refreshing the "Properties" tab located at the bottom of CRXDE.

Sidekick page properties selection

Congratulations! You have successfully extended an existing component, promoting reuse in your development efforts. Again, this type of technique will allow you to easily reuse scripts, Dialogs, etc. from Components that you may develop in the future.

Adding Additional Structure to the Application

Why add additional structure? Again, this is merely an extension of a previous exercise where you added the <body> section of the "Page" Component to a new script (*body.jsp*), and then included that script. Adobe feels it is important you become well versed in this capability, as it will allow you to more easily reuse Components in the future. In addition, you are attempting to reflect the Geometrixx structure of its Contentpage Component, so that you may recreate the Web site in its entirety.

NOTE

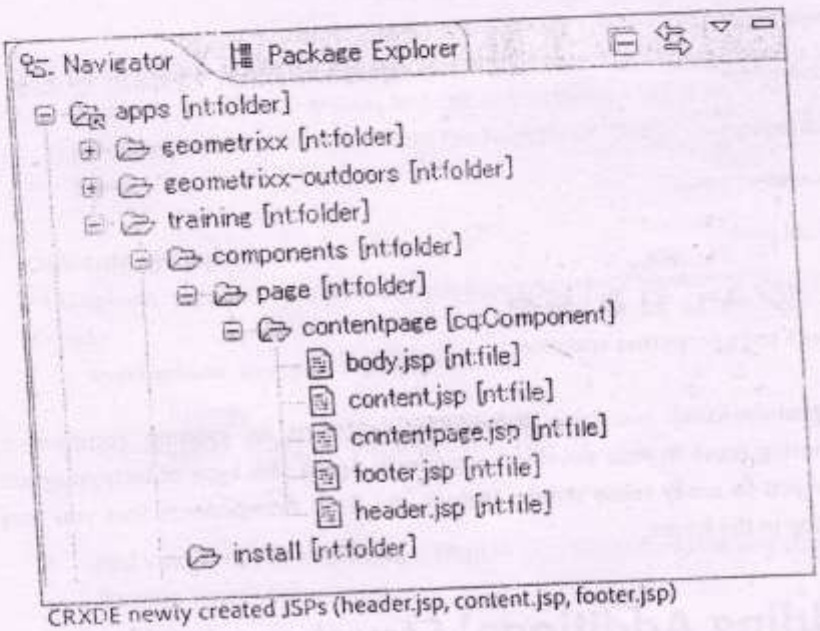
Any style statements in your scripts will reference statements and declarations in the Style Sheets that are included as part of the Design(er). For example:

<h1> </h1>

CQ5 includes a style sheet named static.css by default

EXERCISE - Extend the Script Structure of the "Page" Component

1. Replace the current contentpage.jsp and body.jsp with the scripts from USB/Exercises/extend-script-structure. Your contentpage component should now have the script structure as pictured below:



CRXDE newly created JSPs (header.jsp, content.jsp, footer.jsp)

2. Open the file header.jsp and inspect the HTML and JSP code.

header.jsp

```
<%@include file="/libs/foundation/global.jsp" %>
<div class="header">
    <div class="container_16">
        <div class="grid_8">
            <div> logo </div>
        </div>
        <div class="grid_8">
            <div class="search_area">
                <div> userinfo </div>
                <div> toptoolbar </div>
                <div> search </div>
                <div class="clear"></div>
            </div>
        </div>

        <div> topnav </div>
```

```
    </div>
</div>
```

3. Open the file content.jsp and inspect the HTML and JSP code.

content.jsp

```
<%@include file="/libs/foundation/global.jsp" %>
<div class="container_16">
    <div class="grid_16">
        <div> breadcrumb </div>
        <div> title </div>
    </div>
    <div class="grid_12 body_container">
        <div> par </div>
    </div>
    <div class="grid_4 right_container">
        <div> newslist </div>
        <div> rightpar </div>
    </div>
    <div class="clear"></div>
</div>
```

4. Open the file footer.jsp and inspect the HTML and JSP code.

footer.jsp

```
<%@include file="/libs/foundation/global.jsp" %>

<div class="footer container_16">
    <div class="grid_6">
        <div> toolbar </div>
    </div>
    <div class="clear"></div>
</div>
```

5. Open the file body.jsp and inspect the HTML and JSP code.

body.jsp

```
<%@include file="/libs/foundation/global.jsp" %>
 <body>
     <div class="bg">
         <cq:include script="header.jsp"/>
         <cq:include script="content.jsp"/>
         <cq:include script="footer.jsp"/>
     </div>
 </body>
```

6. Test your script by requesting a Page in CQ5 Siteadmin that implements this Training "Page" Component.
   - If successful, you should see a simple text output (which we will fix soon) of words and the Sidekick, similar to the image below.

Page view of additional structure

Congratulations! You have added additional structure to an existing Component, promoting reuse in your development efforts. Again, Day feels it is important you become well versed in this capability/technique, as it will allow you to more easily reuse Components in the future. Now let's make the output more attractive by introducing you to Designer functionality.

The Design

What is a Design(er) and why do I need it?

By creating and assigning a Design(er) in CQ5, it allows you to enforce a consistent look and feel across your Web site, as well as share global content. The many Pages that use the same Design(er), will have access to common CSS files, defining the formats of specific areas/Components, and images that you use for features, such as backgrounds and buttons.

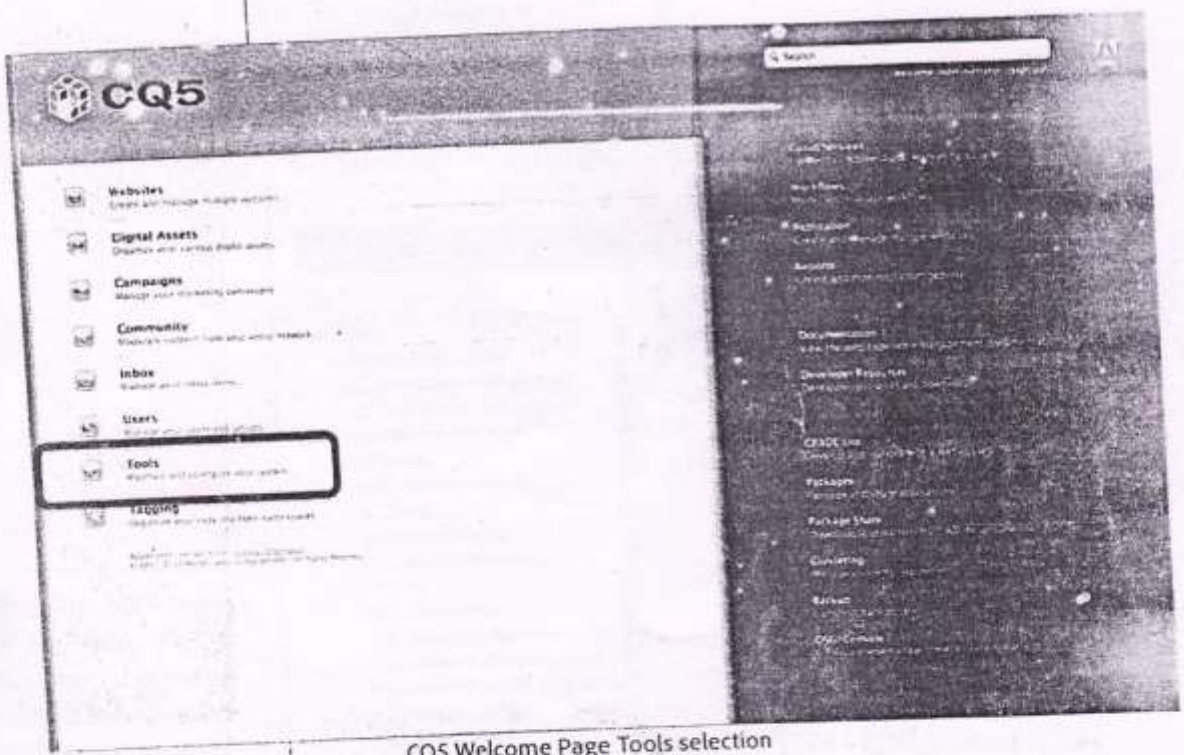
CQ5 has been developed to maximize compliance with the Web Accessibility Guidelines. Web accessibility means that people with disabilities can perceive, understand, navigate, and interact with the Web, and that they can contribute to the Web.

This can include measures such as providing textual alternatives to images (or any non-text item). These can then be used to help people with sight impairment by outputting the text on a Braille keypad, or through a voice synthesizer. Such measures can also benefit people with slow internet connections, or any internet user - when the measures offer the user more information.

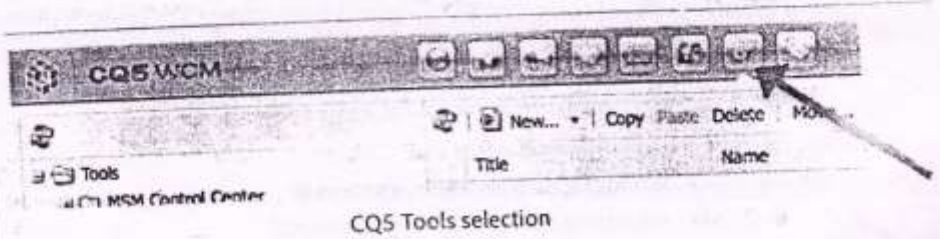
Such mechanisms must be carefully planned and designed to ensure that they provide the information required for the user to successfully understand and use the content. Certain aspects are integral to CQ5, whereas other aspects must be realized during your project development.

EXERCISE - Create and Assign a Design

1. Navigate to the "**Tools**" section of CQ5
   - URL: http://localhost:4502/libs/wcm/core/content/misc.html



CQ5 Welcome Page Tools selection

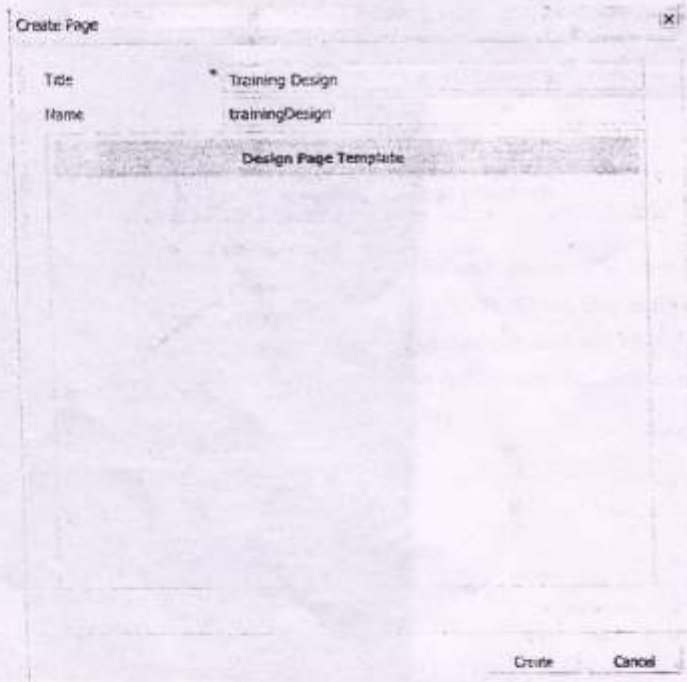


CQ5 Tools selection

2. Select the "Designs" folder - then select **New..., New Page ...**

Tools new page (design) selection

3. Enter the Design(er) "Title" (Training Design) and "Name" (trainingDesign) in the dialog - then select **Create**.



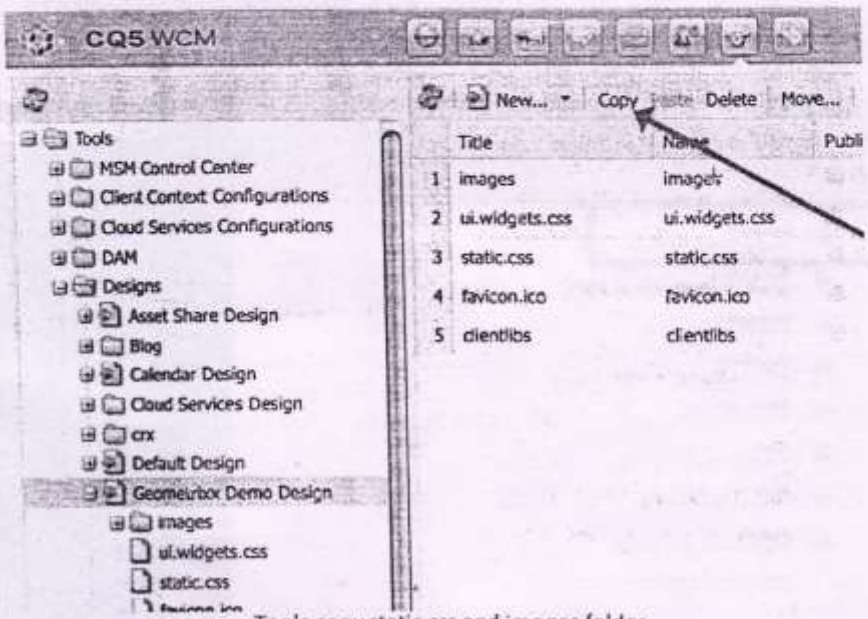
Tools create page (design)

NOTE

If you believe your implementation will have more than one Design(er), which is quite common, it is recommended you create a design structure that will allow multiple Design(er)s to be associated with the one project.

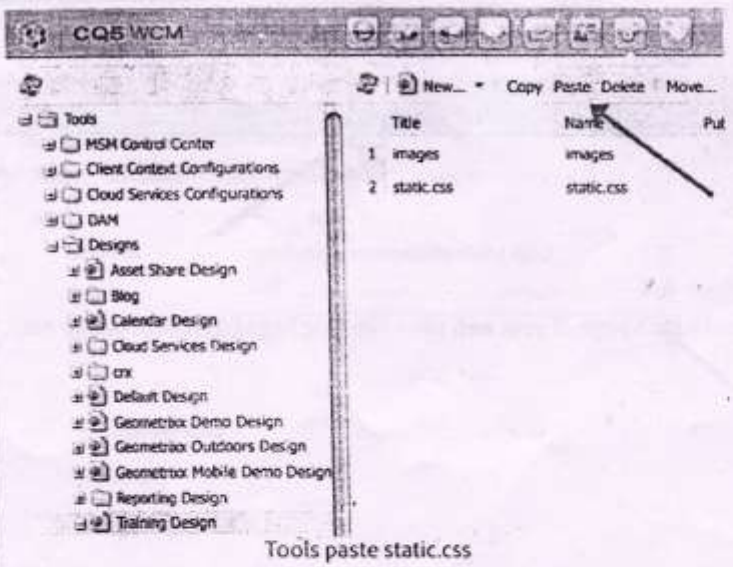
- /etc/designs/trainingDesign
  - /etc/designs/trainingDesign/default
  - /etc/designs/trainingDesign/products

4. Select the "Geometrixx" Design(er), afterward selecting the **static.css** and the **images** folder- then select **Copy.**



Tools copy static.css and images folder

5. Select the Training Design(er) - then select **Paste.**



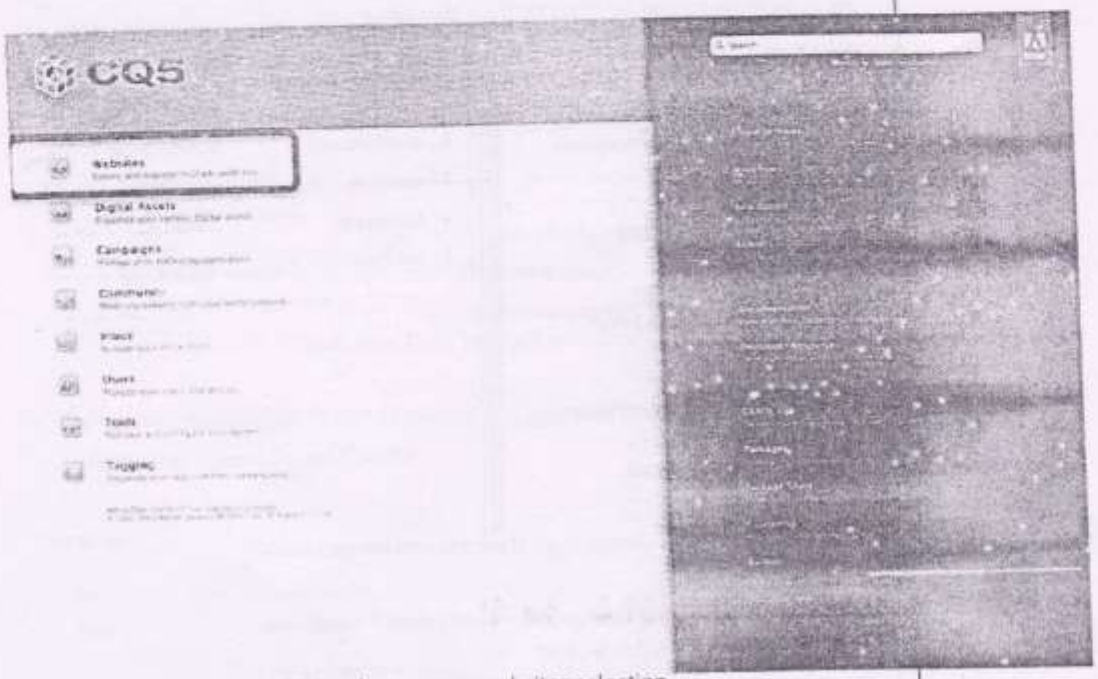
Tools paste static.css

NOTE: In the interest of time, we will create a new Design(er) for Training, but copy the existing static.css file from the Geometrixx Design(er).

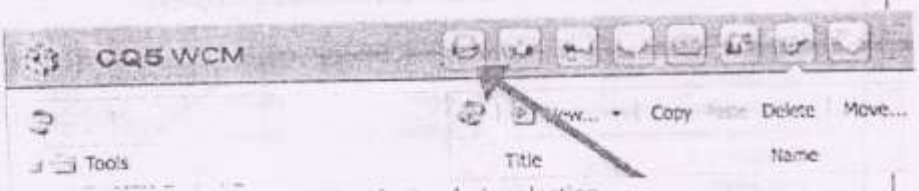
Congratulations! You have successfully created a Training Design(er). You can easily modify the static.css file by using CRXDE and going to /etc/designs/training. Now that we have a Design(er), the next task is to assign that Design(er) to our existing Training Web site structure.

How to assign a Design(er) to your Web site, using CQ5 Siteadmin:

1. Navigate to the "Websites" section of CQ5.
   - URL: http://localhost:4502/siteadmin



CQ5 welcome page websites selection



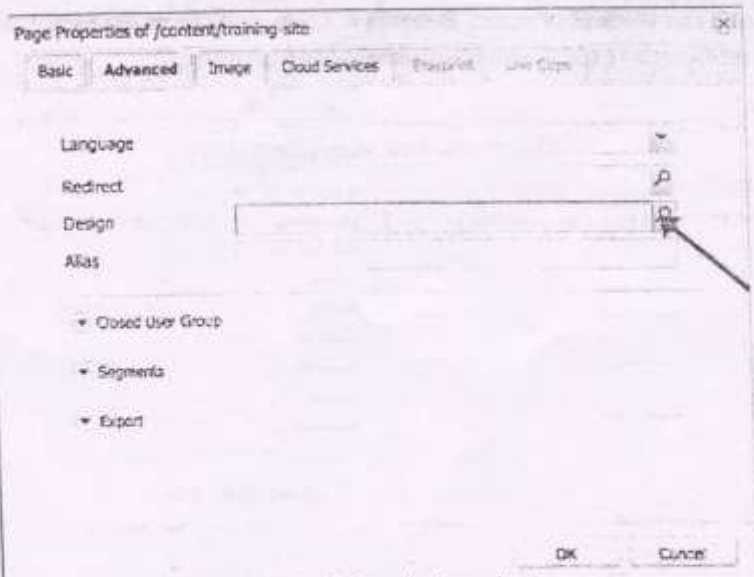
CQ5 tools siteadmin selection

2. Open the root page of your web site - Training Page (*/content/training-site*).

3. Select "Page Properties" in the Sidekick - a dialog will pop-up.

Sidekick page properties selection

4. Select the "Advanced" tab - then select the "Design" option.

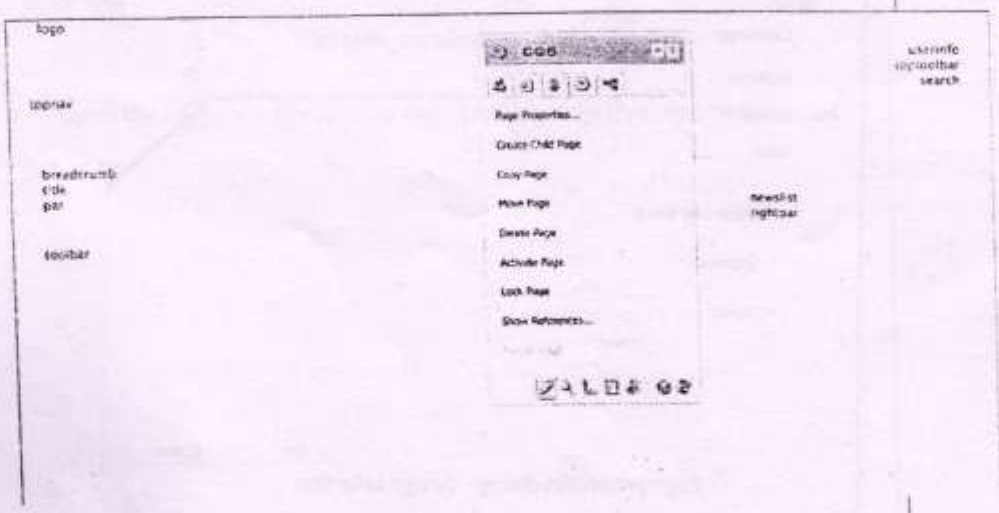


Page properties dialog - design selection

5. Select the newly created **TrainingDesign** Design(er) - then select **OK, OK**.

Design selection dialog

Congratulations! You have successfully assigned a Training Design(er) to your Training Web site. You should now be able to view the "enhanced" Page, which ought to look similar to the image below. You will also notice the Sidekick no longer displays Components. As a Designer, you have not yet declared what Components can be used with this Design(er). Again, you can easily modify the static.css file by using CRXDE and going to /etc/designs/training.



NOTE

You may want to examine the Geometrixx design to discover how themes can be used with the design. We will be discussing the use of Client Library Folders to organize stylesheets, javaScript files, and other design objects when we learn about creating custom input widgets.

Section 6 - Component Basics

As promised, we will now talk about components that are not page-rendering components. These components will make up the bulk of the custom components that you will create.

In addition to properties, components have child nodes that help to control the rendering process. In particular, the following child nodes may be defined for components:

- cq:editConfig
  - controls Author user interface for drag-and-drop from Content Finder
  - controls Author user interface for aspects (edit bar or widget appearance)
- cq:childEditConfig
  - controls Author user interface for child components
- dialog
  - content editing dialog for this component
- design_dialog
  - design editing dialog for this component

Including components into scripts

We introduce here a new format of the <cq:include/> tag. This format is specific to including entire components, instead of just a single script.

```
<cq:include path="topnav" resourceType="training/components/topnav"
/>
```

- path

- The path to the resource object to be included in the current request processing. If this path is relative, it is appended to the path of the current resource whose script is including the given resource.

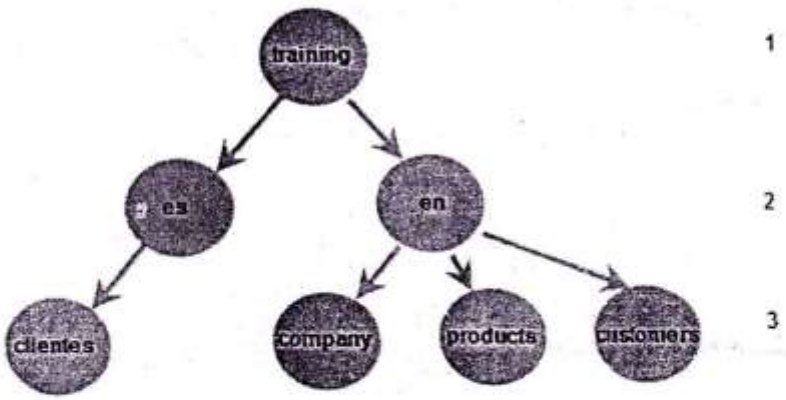
- resource type
  - The resource type of the resource to be included. If the resource type is set, the path must be the exact path to a resource object: in this case, adding parameters, selectors and extensions to the path is not supported.
  - If the resource to be included is specified with the path attribute that cannot be resolved to a resource, the tag may create a synthetic resource object out of the path and this resource type.

Dynamic Navigation

To demonstrate the component creation process, we will create a dynamic text-based navigation Component, allowing for Web site structure to be easily modified and represented/navigated in real-time.

How do I create dynamic navigation?

Providing dynamic navigation capabilities, allowing for the easy addition and removal of Pages, is one of the most important (and sometimes difficult) things you can do as a developer in CQ5. Consider the following image, which represents a simple Web site structure:



Training web site structure

If you want to create a dynamic navigation Component that displays all the valid, children Pages of a language Page (e.g. "en"), you would need to identify the following:

- What Page is being requested?
    - Is it under the "en" section or the "es" section, as you will want to display the appropriate children of the language the visitor is browsing?
- At what level does the requested Page exist?
    - This is important because if you wish to display the children of a language, then a request to a level 1 Page (i.e. "training") would break the assumption requests would only be made to level 2 (e.g. "en") or level 3 (e.g. "company") Pages.
- Is the requested Page valid?
    - In CQ5, it is possible to configure a Page to not display in any dynamic navigation script, as well as define a specific "On Time" and "Off Time".

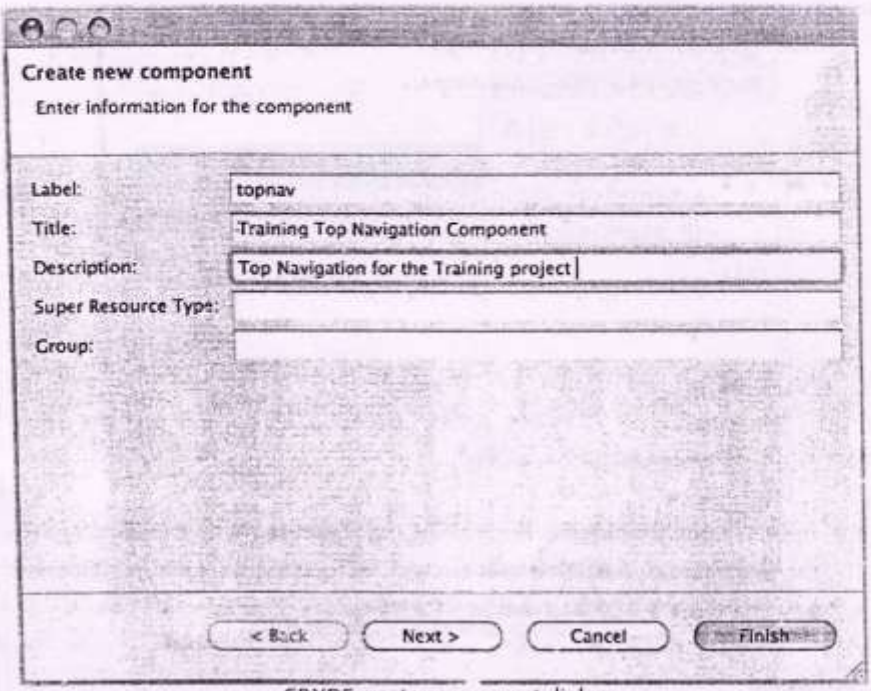
After these questions have been answered, it is relatively easy to collect the title of the Page, which will be displayed, and the path of the Page, which will be used to provide navigation functionality. Important Java classes/interfaces you should understand include:

- com.day.cq.wcm.api.Page
- com.day.cq.wcm.api.PageFilter
- com.day.cq.wcm.api.PageManager

EXERCISE - Create a Dynamic Navigation Component

1. Right-click */apps/<application name>/components* - then select New, Component.

2. Enter the desired Component "Label", "Title", "Description" - then click Finish.
    - Label = the name of the Component/node that will be created
    - Title (property jcr:title) = the title that will be assigned to the Component
    - Description (property jcr:description) = the description that will be assigned to the Component



CRXDE create component dialog

NOTE: This is a similar process to the "Page" Component we created earlier using CRXDE Lite.

NOTE: At this time, we are only concerned with the Label, Title, and Description properties. We will examine all the properties and what they mean as the course continues.

3. Open the file ***topnav.jsp***, and enter some HTML and JSP code, similar to below – then **Save**.

topnav.jsp

```
<%--
Draws the top navigation

--%>
<%@include file="/libs/foundation/global.jsp"%><%
%><%@ page import="java.util.Iterator,
        com.day.text.Text,
        com.day.cq.wcm.api.PageFilter,
        com.day.cq.wcm.api.Page,
        com.day.cq.commons.Doctype,
        org.apache.commons.lang.StringEscapeUtils" %><%

// get navigation root page Page
Page navRootPage = currentPage.getAbsoluteParent(2);

// check to make sure the page exists
if (navRootPage == null && currentPage != null) {
    navRootPage = currentPage;
}
if (navRootPage != null) {
        Iterator<Page> children = navRootPage.listChildren(new
PageFilter(request));
```

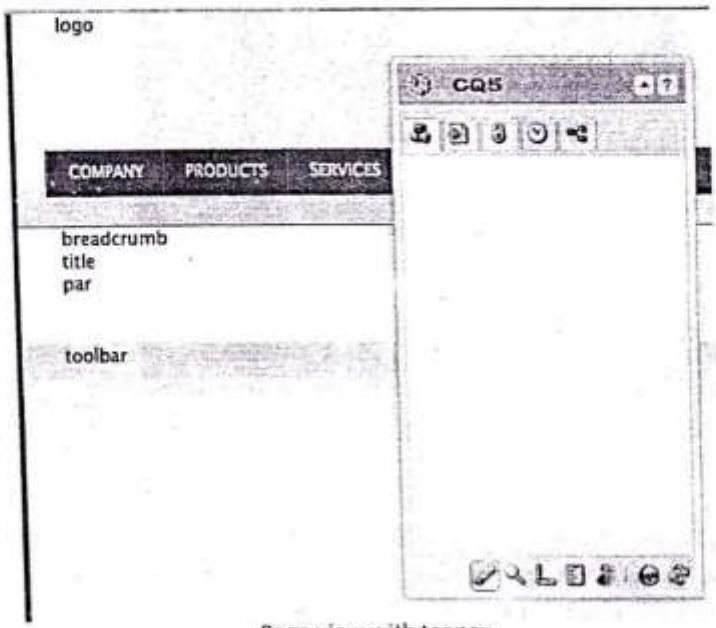
```
%> <ul>  <%
   while (children.hasNext()) {
         Page child = children.next();
%>
         <li>
            <a href="<%= child.getPath() %>.html">
            <%= StringEscapeUtils.escapeXml(child.getTitle())%> </a>
         </li>
 <%
    }
 %> </ul>  <%
 }
 %>
```

4. Open the file *header.jsp* in the Training Contentpage Component and replace the "topnav" section with a <cq:include> of the navigation Component you just created, similar to below – then **Save**.

header.jsp

```
<%@include file="/libs/foundation/global.jsp" %>
<div class="header">
  <div class="container_16">
    <div class="grid_8">
      <div> logo </div>
    </div>
    <div class="grid_8">
      <div class="search_area">
        <div> userinfo </div>
        <div> toptoolbar </div>
        <div> search </div>
        <div class="clear"></div>
      </div>
    </div>

      <cq:include path="topnav" resourceType="training/components/
topnav" />

  </div>
</div>
```

5. Test your script by requesting a Page in CQ5 Siteadmin that implements this Training "Page" Component.
   - If successful, you should see navigation links/titles directly related to your Web site structure.



Page view with topnav

Congratulations! You have successfully created a Component that dynamically represents the navigation structure of your Web site. This is a perfect example of how you can easily create and integrate Components into your Templates.

Logging Messages

Now that we have created our first non-page-rendering component and encountered the occasional problem, it is time to talk about logging. Adding log messages to a Component script will allow you to more easily debug various scripts you may be working on.

In the daily life of a developer, it is often crucial to monitor the values of variables assigned/used. There are several possibilities, in various usability levels. CQ5 and CRXDE make your life a little easier by implementing the popular Log4j framework, which is designed to provide an easy-to-use logging solution.

The initialization of a "Logger" object, called "log", has already been accomplished during the inclusion of *global.jsp* in whatever Component you may be working on. The log file entries are formatted according to the Sling configuration. Two pieces of information are required to append an entry to the log file:

- log level
  - This is provided by the corresponding method call. For example, a log.debug(<message>) produces a message with log level "debug",

while a log.info(<message>) produces a message with log level "info". Possible methods of the "Logger" object include:

- trace()
- debug()
- info()
- warn()
- error()

- message
  - The message itself is provided as a parameter to the method call. For example, log.debug("This is the log message") appends the message "This is the log message" with a log level of "debug" to the error.log file.

NOTE: In a default configuration, the log file is located under <cq-install-dir>/crx-quickstart/logs/error.log.

EXERCISE - Add a log message to the topnav component

1. Open the file *topnav.jsp* in the Training "topnav" Component, and enter some JSP code, similar to below - then Save.

topnav.jsp

```
<%@include file="/libs/foundation/global.jsp"%>
<%@ page import="java.util.Iterator,
        com.day.text.Text,
        com.day.cq.wcm.api.PageFilter,
        com.day.cq.wcm.api.Page,
        com.day.cq.commons.Doctype,
        org.apache.commons.lang.StringEscapeUtils" %>
<%
// get navigation root page Page
Page navRootPage = currentPage.getAbsoluteParent(2);

// check to make sure the page exists
if (navRootPage == null && currentPage != null) {
        navRootPage = currentPage;
}

if (navRootPage != null) {
        // create an iterator object of all nav root's child pages
            Iterator<Page> children = navRootPage.listChildren(new
PageFilter());
        %> <ul> <%
        while (children.hasNext()) {
```

```
            // get next child page
            Page child = children.next();

            //log message
            log.info("Child page [{}] found.", child.getTitle());

            // display the link in an <A HREF...
%>

<li>
            <a href="<%= child.getPath() %>.html">
            <%= StringEscapeUtils.escapeXml(child.getTitle())%> </a>
           </li>
<%
        }
        %> </ul> <%
}
%>
```

2. Test your script by requesting a Page in CQ5 Siteadmin that implements this Training "Page" Component.

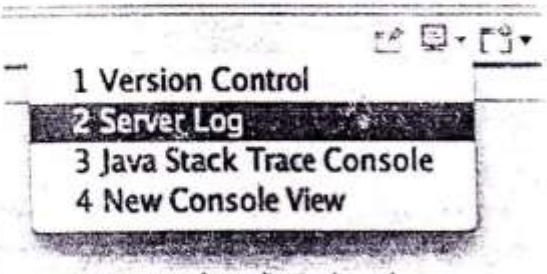
- If successful, you should see no difference between what is displayed now, and what was displayed before. However, if you were to inspect the error.log file, you would see a message similar to below:

  04.12.2009 11:23:22.921 *INFO* [127.0.0.1 [1259943802812] GET /content/trainingSite/en/company.html HTTP/1.1] apps.training.components.content.topnav.topnav$jsp child page [Company] found.

NOTE: In CRXDE you can display the file error.log that is located on the file system at <cq-install-dir>/crx-quickstart/server/logs and filter it with the appropriate log level. This can be done by proceeding as follows in CRXDE:.

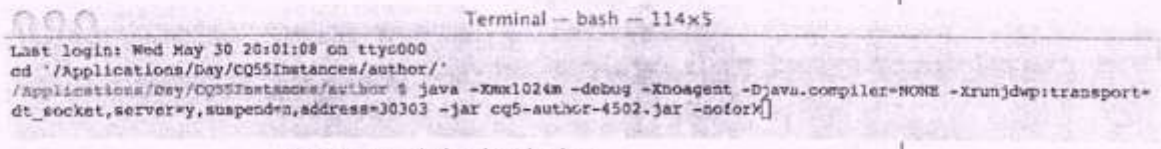
Congratulations! You have successfully added a log message to your script. Again, it is often crucial to monitor the values of variables assigned/used. The "Logger" object is a useful tool that will enable you to do so.

1. Select the **Console** tab located at the bottom of the window – then select the arrow beside the **Open Console** – then select **Server Log**.



CRXDE console application log select

2. Select the desired "log level" in the drop down menu.
   - The error.log is then filtered according to the log level and displayed in the tab.



CRXDE console log level select

Enabling the Debugger

Sometimes, outputting variable values in a log file is not enough for a developer to monitor the sequence of steps executed by the system. The best way to accomplish this is the usage of a debugger. Many modern IDEs like Eclipse and CRXDE provide this functionality.

Again, the goal of this exercise is to learn how to use the built-in debugger in CRXDE. A debugging session consists of a server and a client. In this training, you will use CRXDE as the client, while the server is provided by CQ5. For performance reasons, Day recommends not enabling debugging for CQ5 production instances, as this would consume too many resources, thus hindering performance.

EXERCISE - enabling the debugger

1. Shutdown your CQ5 author and CRXDE instance.
2. Open a terminal window and navigate to the directory where the Java servlet engine (CQSE) has been installed using your command line.
   - e.g. <cq-install-dir>/crx-quickstart/bin
3. Enter the following command to start the CQ5 server in debug mode.
   - Windows = quickstart.bat-debug socket
   - Linux/Unix = ./start -d

DOS command line start/enable of debugger

NOTE

By default, port 30303 will be used by the server for debugging communication. You can customize the server.bat/start file for future use. In addition, you can call the debugger directly:

```
java -debug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:transport=dt_
socket,server=y,suspend=n,address=30303 -jar cq-author-4502.jar
-nofork
```

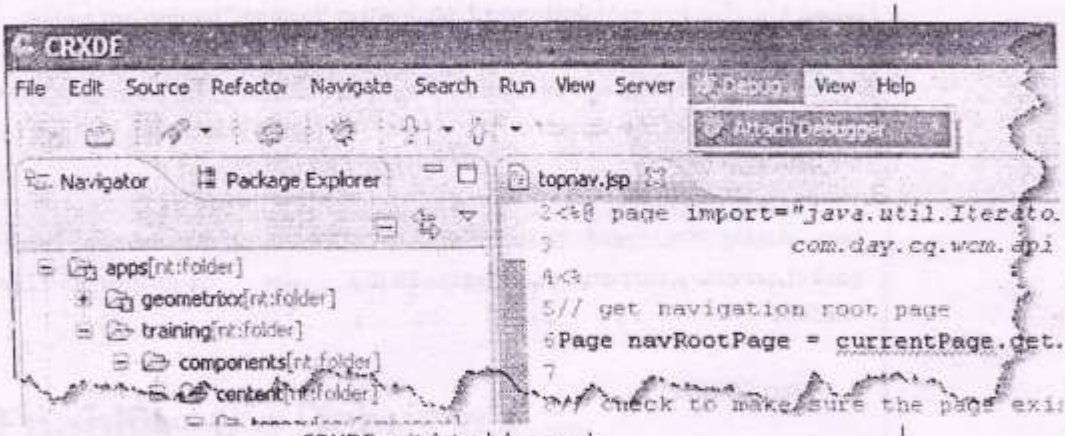
4. Start CRXDE.

5. Open the file ***topnav.jsp*** in the Training "topnav" Component.

6. Right-click the left-side of the JSP editor on a line of code (e.g. line 17: if (navRootPage != null) {}) then select **Toggle Breakpoints**.

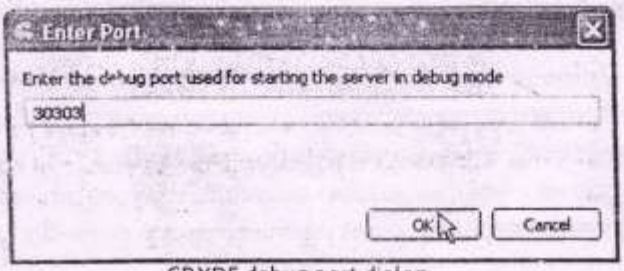
CRXDE right-click JSP editor

7. Switch to debug mode by selecting **Debug, Attach Debugger.**



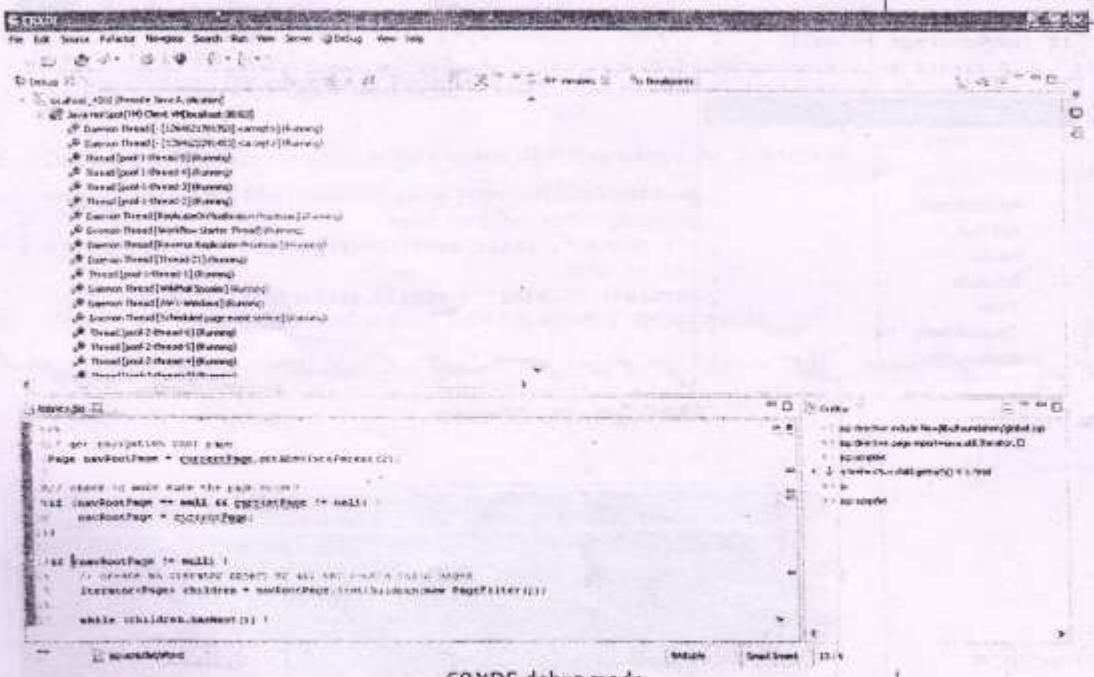
CRXDE switch to debug mode

8. Enter the debug port (30303) used for starting the server in debug mode - then select OK.



CRXDE debug port dialog

NOTE: You are now in debug mode. Your CRXDE view should look similar to the image below.

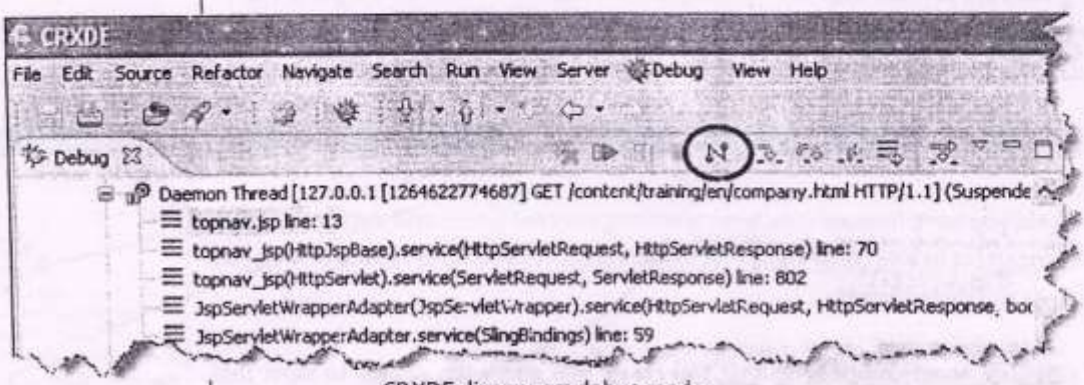


CRXDE debug mode

9. Test your script/debug mode by requesting a Page in CQ5 Siteadmin that implements this Training "topnav" Component.

- In CRXDE, you will notice your script stops at the line of code containing the breakpoint
- Here you will be able to investigate variables, execution, and other related elements, in addition to "stepping" through your code

10. Select the **Disconnect** button to disable debug mode.



CRXDE disconnect debug mode

Congratulations! You have successfully enabled debugging in CQ5 and CRXDE. Again, this is a great tool that will allow you to monitor the sequence of steps executed by the system.

Section 7 - More Components

So far, you have focused mostly on rendering content (static and dynamic), which is important as one could argue that rendering content comprises 50% of what you do as a CQ5 developer.

However, most components that you create will allow the author to input that content. Typically, the mechanism that authors will use to input content is through a dialog box.

ExtJs

CQ's web-based interface uses AJAX and other modern browser technologies to enable WYSIWYG editing and formatting of content by authors right on the web page.

The widget library used by CQ5, called ExtJS, provides the highly polished user interface elements that work across all the most important browsers and allow the creation of desktop-grade UI experiences. The popular ExtJS JavaScript framework gives developers the ability to easily create Rich Internet Applications (RIA) through the use of AJAX. It includes:

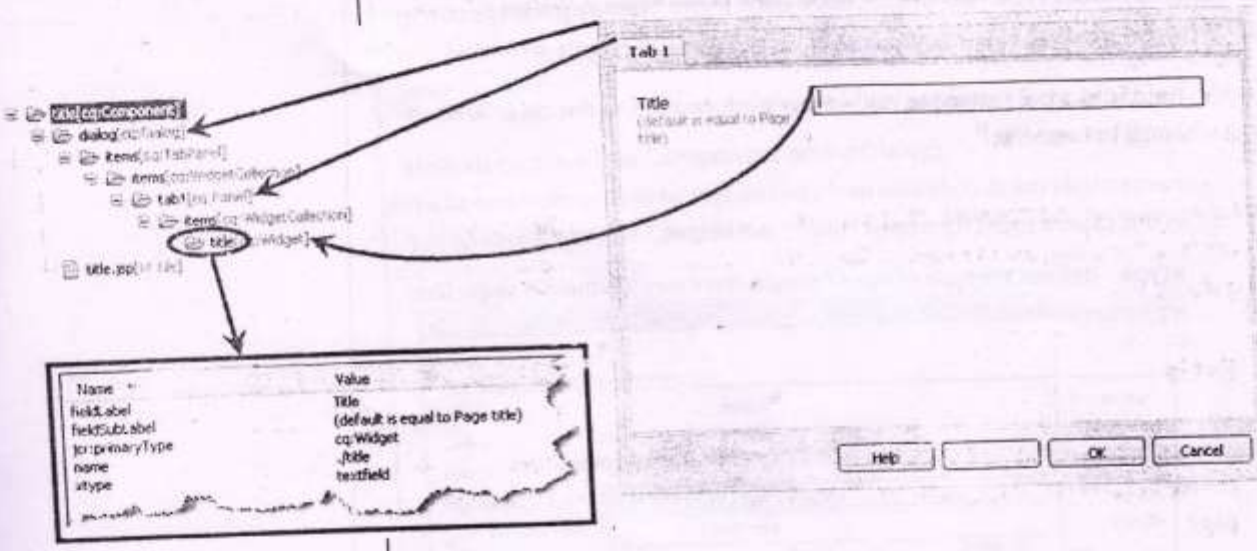
- High performance, customizable UI widgets
- Well designed and extensible Component model
- An intuitive, easy to use API

ExtJs is a cross-browser JavaScript library for building interactive web applications. It supports the rapid development of high performance, customizable User Interface widgets.

Component Dialogs

A CQ5 Dialog is similar to other dialogs you have used/created in the past: it gathers user input via a "form", potentially validates it, and then makes that input available for further use (storage, configuration, etc.).

The Dialog is defined as a child of the component. The root node of the Dialog is of node type "cq:Dialog" and named "dialog". As can be seen in the figure below, the node structure below the cq:Dialog root node defines the structure of the dialog box.



The cq:Dialog node produces the outer light blue outline. The cq:TabPanel node produces the darker blue outline. The cq:Panel node(s) produce the individual tabs - the panel(s) of the Dialog. The input widgets for each panel are placed subordinate to the panel nodes. The "widget nodes" must be of node type "cq:Widget".

You will notice the cq:WidgetCollection node that is a child of the Tab Panel and the Widget Collection node that is a child of the Panel. Every time you can have more than one of any object, you need a Widget Collection to hold the objects. So:

- one cq:dialog node - no WidgetCollection
- one cq:TabPanel node - no WidgetCollection

but, you can have more than one Panel on a Tab Panel, so you need a Widget Collection to hold the Panels. Similarly, you can have more than one Widget on a panel so you need a Widget Collection to hold the Widgets.

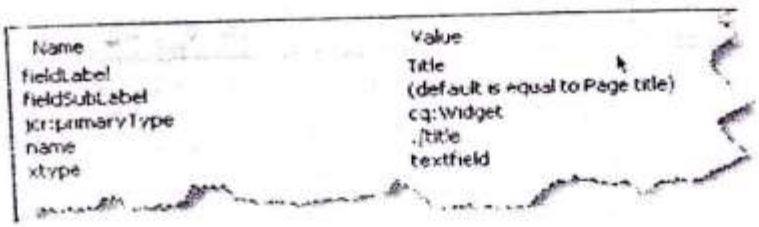
There are few places in the CQ5 environment where specific names must be used. The dialog box is one of those places. Rules:

- Dialog nodes must be of type cq:Dialog and must be named "dialog" ("design_dialog" for design dialog boxes")
- Widget collections are always named "items" . Other nodes may be named "items", but Widget Collections as always named "items".

If you name the nodes other than the expected names, CQ5 will not recognize the nodes and will not produce the expected construct.

Now let's examine the properties on the widget itself. In the figure above, the title widget has 5 properties:

- **jcr:primaryType** - defines the node type, in this case: cq:Widget
- **fieldLabel** - tells the Author what to do, in this case: "Title"
- **fieldSubLabel** - gives the Author more information, in this case: "(default equal to Page title)"
- **name** - tells the JS code that backs this widget what property name to write, in this case: a property named "title" - see below...
- **xtype** - defines the type of input form, in this case: "textfield" - single line, alphanumeric



The name property value, for the Title Component, is "./title". The "./" construct, similar to the same convention in the U*x world, signifies "here, on this node". So, what this means is "When the Author enters content into the textfield widget and click OK, write the content into a property named *title* on this node."

BEST PRACTICE

The dialog box node structure that we have discussed in this section is the structure recommended by Adobe CQ5 Best Practices. This structure is created automatically by the Dialog Creation Wizard in CRXDE. Use the Dialog Creation Wizard from CRXDE or create this node structure using your favorite IDE and the Content Explorer.

You will find that the Foundation components do not follow these best practices. You will note that the Foundation component dialog boxes do not have a Tab Panel node and, sometimes, do not have a Panel node, but begin directly with the final Widget Collection.

NOTE: There is one exception when a Widget Collection is not named "items". For a Selection Widget, the subordinate Widget Collection to the Widget is named "options" .

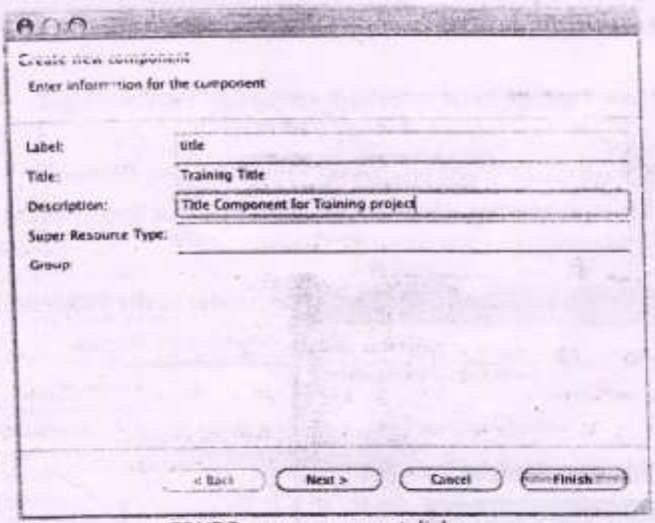
This format works, but then maintenance becomes more difficult. Your dialog boxes will not have a uniform structure. Maintenance tasks become more time consuming. For example, you will have to rearrange the dialog structure should you need to add a Panel to the dialog box.

EXERCISE - Create a Title Component

The following instructions explain how to create a Component that allows authors to manage a Page's title (textual content), through the use of a simple Dialog. This is a beginning step in learning how to create Components that allow authors to write content.

How to create a title Component with a Dialog:

1. Create a new title Component.



CRXDE new component dialog

NOTE: This process is similar to creating the navigation Component.

2. Open the file title.jsp, and enter some HTML and JSP code, similar to below – then Save.

title.jsp

```
<%@include file="/libs/foundation/global.jsp"%>
<h2><%= properties.get("title", currentPage.getTitle()) %></h2>
```

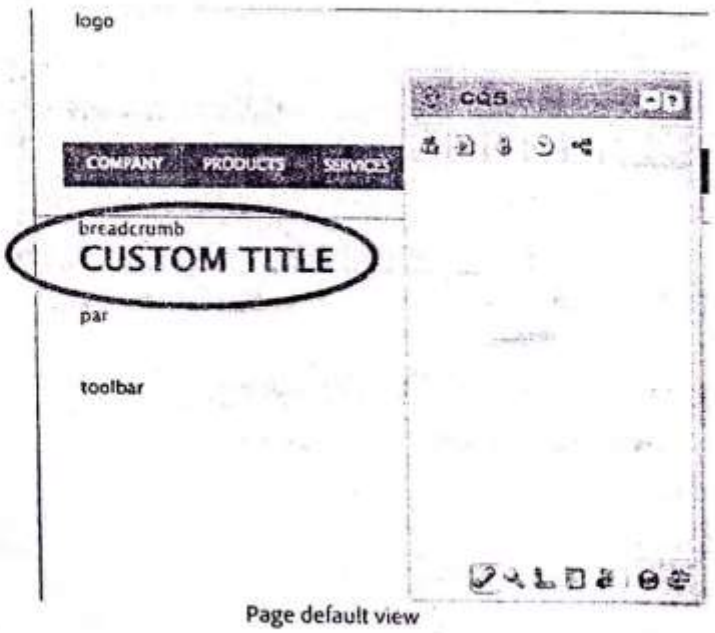
3. Open the file content.jsp in the Training Contentpage Component and replace the "title" section with a <cq:include> of the title Component you just created, similar to below - then Save.

content.jsp

```
<%@include file="/libs/foundation/global.jsp" %>
<div class="container_16">
 <div class="grid_16">
    <div> breadcrumb </div>

       <cq:include path="title_node" resourceType="training/
components/title" />

 </div>
 <div class="grid_12 body_container">
     <div> par </div>
 </div>
 <div class="grid_4 right_container">
     <div> newslist </div>
     <div> rightpar </div>
 </div>
 <div class="clear"></div>
</div>
```

NOTE: You will notice the use of the Sling ValueMap class/ object "properties". This object allows you to easily get properties associated with this instance of this Component.

4. Test your script by requesting a Page in CQ5 Siteadmin that implements this Training title Component.
   - If successful, you should see the default Page title, similar to the image below.



Page default view

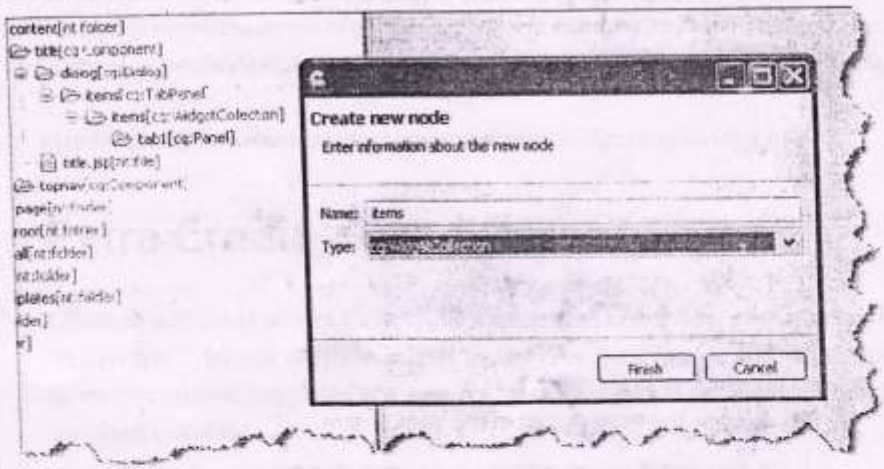
5. Right-click the title Component - then select **New, Dialog.**

6. Enter the desired Dialog "Name" (dialog) and "Title" (Training Title Component).

CRDY new dialog dialog

7. Right-click the *tab1* node - then select **New, Node.**

8. Enter the desired "Name" (items) and "Type" (cq:WidgetCollection).



CRXDE new node dialog

9. Create a new node under the newly created *items* node.
   - Name = title
   - Type = cq:Widget

10. Right-click the newly created *title* node - then select **New, Property.**

11. Enter the desired "Name", "Type", and "Value" for the properties listed below.

- The property that will define where the content is stored
  - Name = name
  - Type = String
  - Value = ./title
- The property that will define the Widget type (i.e. user interface control)
  - Name = xtype
  - Type = String
  - Value = textfield
- The property that will define the label applied to the Widget
  - Name = fieldLabel
  - Type = String
  - Value = Enter Title Here

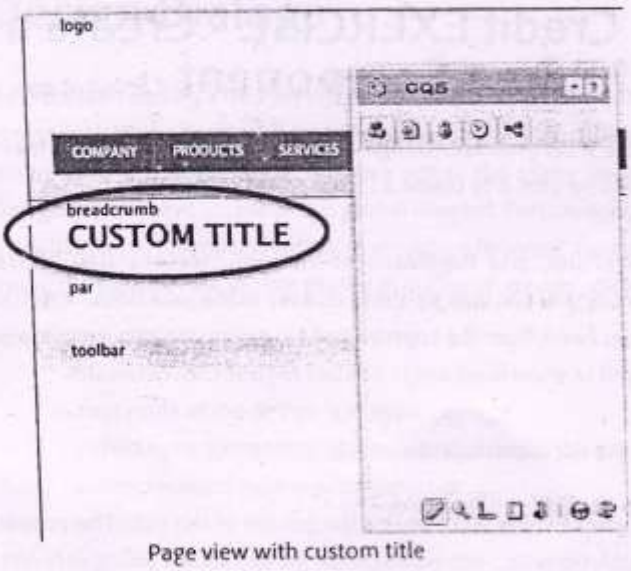
12. Test your script/Dialog by requesting a Page in CQ5 Siteadmin that implements this Training title Component - then double-click the "title" content to invoke the Dialog.

13. Enter the desired custom title - then select **OK.**

Dialog of title

14. Review the Page output of the title content.
    - If successful, you should see the custom Page title, similar to the image below.

NOTE: Notice how tab1's label is "Tab 1." If you want to change this value, simply assign a property named title (type String) to the tab1 node, and assign a value.



Page view with custom title

Congratulations! You have successfully created a title Component, with a Dialog, that allows content authors to write content and have it displayed. This is a significant step in your development capabilities, as you are now able to create CQ5 Components that allow you to write content to the repository.

Additional Information

These are but a few of the many properties and Widgets you can use to create a Dialog that meets the needs of your users/content authors. You can find a complete listing of xtypes/widgets by consulting the widget definitions at the following URL :

http://dev.day.com/content/docs/en/cq/current/widgets-api/index.html

Extra Credit - Try different xtypes

Create an additional widget for the Title Component dialog box, perhaps named "testWidget". Try out different xtypes to see how they appear and what input forms they present. Suggested xtypes to try: textarea, sizefield, pathfield, multifield, datefield, colorfield

TIP

The best way to familiarize yourself with the widget appearance and function is to consult the Foundation Components. Open up the dialog boxes for those components and investigate the widgets. When you find a widget that is interesting, look at that component's dialog structure in the repository to see the xtype that defines that widget.

Extra Credit EXERCISE - Create a List Children Component

The following exercise tests your knowledge of component creation and taking input from an author. The goal is to create a ListChildren component.

Using the Page, PageFilter, and PageManager classes, create a **listChildren** component. Remember you can look up these classes in the java docs. You have all the information you need, from the **topnav** and **title** components, to complete this component.

1. Include/import the appropriate java classes.
2. Get a property whose value will become the parent of the list. The property should be a path.
3. If the property is empty, use a default. (perhaps a static path or path to current page)
4. Use the path that the author entered to get the parent page.
   (hint: the PageManager class has a method to get a page object when you know its path)
5. Set up a page iterator, filtering by valid pages
6. Iterate over the children
7. For each child, write out a link
8. Create a dialog box
9. Place an input widget on the dialog box so the author can enter the list parent.
10. Edit *content.jsp*. Replace the placeholder for *newslist* with a **<cq:include>** of your new component.

Congratulations! You have successfully created a listChildren Component, with a Dialog, that allows content authors to enter content and have it used as a list root. This is a significant step in your development capabilities, as you are now able to create CQ5 Components that allow you to write content to the repository and use that information to create new rendered content.

Design Dialogs

As discussed earlier, a Design(er) can be used to enforce a consistent look and feel across your Web site, as well as share global content. This global content is editable when using a Design Dialog. So once again, the many Pages that use the same Design(er) will have access to this global content. Fortunately, the process to create a Design Dialog is almost identical to creating a "normal" Dialog - the only difference being the name of the Dialog itself (i.e. dialog vs. design_dialog).

- Global content
  - Stored in /etc/designs instead of the local node of the page
  - root node of the design is of type cq:Page
    - child node jcr:content of type cq:PageContent
    - sling:resourceType = wcm/designer

The Design(er) values can be accessed by the currentStyle object provided from the global.jsp. This is different than using the properties object as we have done so far.

Design dialogs are almost identical to component dialogs with the following exceptions:

- name - design_dialog instead of dialog
- content storage - /etc/designs instead of /content/website
- availability - design mode instead of edit mode

To demonstrate the use of the Design, we will be creating a Logo Component. The Logo Component will use the smartimage widget. This is the first time that we will be dealing with binary content.

Therefore, it is important to note is the use of the resource SuperType foundation/components/parbase. The sling:resourcesSuperType property is a key component as it allows components to inherit attributes from other components, similar to subclasses in object oriented languages such as Java, C++, and so on. The Parbase Component defines tree scripts to render images, titles, and so on, so that all components subclassed from this parbase can use this script.

BEST PRACTICE

When using the Dialog Widget smartimage, as you will to complete this exercise, it is recommended to extend this "helper" Component or a similar custom component. It contains the Java source *img.GET.java*, which allows you to more easily manage/transform images in fewer lines of code.

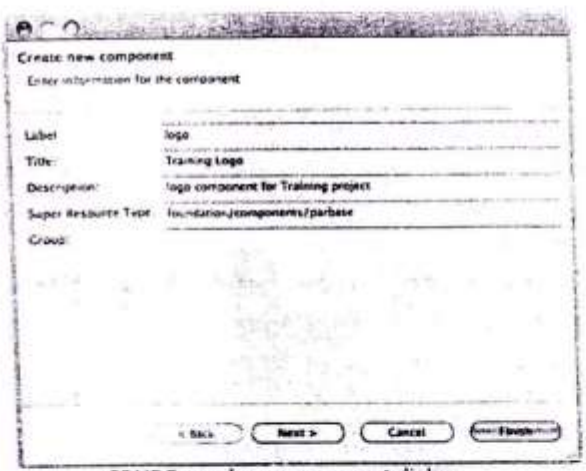
SmartImage Widget

All images rendered from the smartImage widget are processed by *img.get.java* or similar script. that you may create. ....

Name	Value	Type
cropParameter	./imageCrop	String
ddGroups	media	String
fileNameParameter	./fileName	String
fileReferenceParameter	./fileReference	String
jcr:primaryType	cq:Widget	Name
mapParameter	./imageMap	String
name	./file	String
renditionSuffix	/_jcr_content/renditions/cqSdam.we...	String
requestSuffix	.img.png	String
rotateParameter	./imageRotate	String
title	Image	String
uploadUrl	/tmp/upload_test/*	String
xtype	smartimage	String

EXERCISE - Create a Logo Component

1. Using the same steps that we have used before, create a new logo Component



CRXDE new logo component dialog

2. Open the file logo.jsp, and enter some HTML and JSP code, similar to below - then **Save.**

logo.jsp

```
<%@include file="/libs/foundation/global.jsp"%><%
%><%@ page import="com.day.text.Text,
```

```
                com.day.cq.wcm.foundation.Image,
                com.day.cq.commons.Doctype" %><%

    String home = Text.getAbsoluteParent(currentPage.getPath(), 2);
    Resource res = currentStyle.getDefiningResource("fileReference");
    if (res == null) {
        res = currentStyle.getDefiningResource("file");
    }
log.error("path is:" + currentStyle.getpath());

    %><a href="<%= home %>.html"><%
    if (res == null) {
        %>Home Page Placeholder<%
    } else {
        Image img = new Image(res);
        img.setItemName(Image.NN_FILE, "file");

img.setItemName(Image.PN_REFERENCE, "fileReference");
        img.setSelector("img");
        img.setDoctype(Doctype.fromRequest(request));
        img.setAlt("Home Page Placeholder");
        img.draw(out);
    }
    %></a>
```

3. Open the file **header.jsp** in the Training Contentpage Compcnent and replace the "logo" section with a <cq:include> of the logo Component you just created, similar to below – then **Save**.

header.jsp

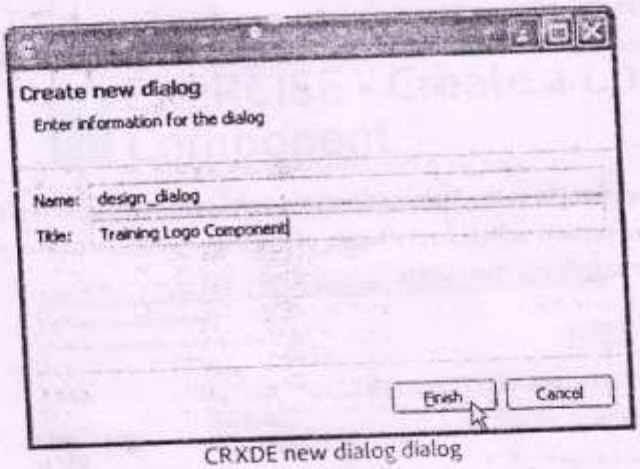
```
<%@include file="/libs/foundation/global.jsp" %>
<div class="header">
    <div class="container_16">
        <div class="grid_8">
                <cq:include path="logo" resourceType="training/
components/logo" />
        </div>
        <div class="grid_8">
            <div class="search_area">
                <div> userinfo </div>
                <div> toptoolbar </div>
                <div> search </div>
                <div class="clear"></div>
            </div>
        </div>
```

```
        <cq:include path="topnav" resourceType="training/components/
topnav" />
    </div>
</div>
```

4. Test your script by requesting a Page in CQ5 Siteadmin that implements this Training logo Component.
   - If successful, you should see the default logo output, similar to the image below.

Page preview of logo component

5. Create a new Design Dialog for the logo Component.



CRXDE new dialog dialog

6. Create an **items** node (nodeType cq:WidgetCollection) under the *tab1* node of the logo Component's Design Dialog.

7. Create an **absParent** node (nodeType cq:Widget) under the *items* node.

8. Assign the following properties to the newly created *absParent* node:
   - The property that defines where the content is stored
     - Name = name
     - Type = String
     - Value = ./absParent
   - The property that defines the Widget type
     - Name = xtype

- Type = String
- Value = textfield

- The property that defines the label applied to the Widget
  - Name = fieldLabel
  - Type = String
  - Value = Parent Level (absolute)

- The property that will give the author extra information about the Widget
  - Name = fieldDescription
  - Type = String
  - Value = (e.g., 1 for /content/site)

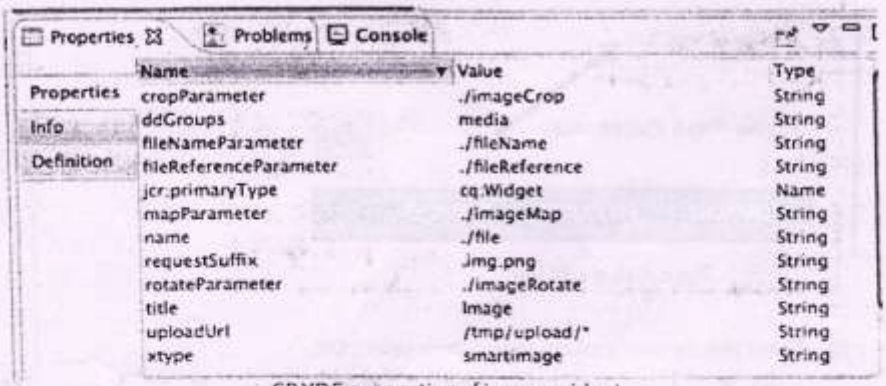
9. To get/define the ***smartimage*** widget that we will use for input and maintenance of the logo image, copy the node */libs/foundation/components/image/dialog/items/image* - then paste to the logo's Design Dialog so that it is a peer of the *tab1* node.

- e.g. /apps/training/components/logo/design_dialog/items/items

NOTE: Instead of always reinventing the wheel, it is often more efficient to copy-and-paste existing Dialogs/Widgets that meet your needs. That being said, it is wise to review what you have just copied to better understand the internal workings of CQ5.

In addition, you may have noticed this Widget was pasted in the location where tabs are typically located. The smartimage xtype (along with a few others) is unique in that it provides a better content author experience when "casted" as a tab.

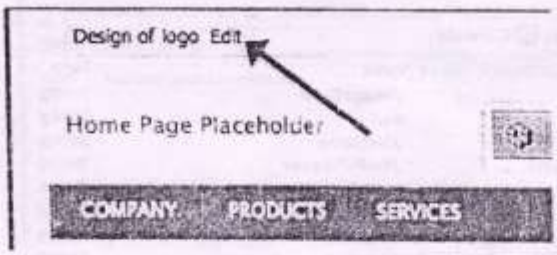
10. Review the properties associated with this ***smartimage*** Widget.



CRXDE properties of image widget

11. Test your script/Design Dialog by requesting a Page in CQ5 Siteadmin that implements this Training logo Component - then select **Design mode** on your Sidekick, and select "edit" for the logo Component to invoke the Design Dialog

Page design mode selection



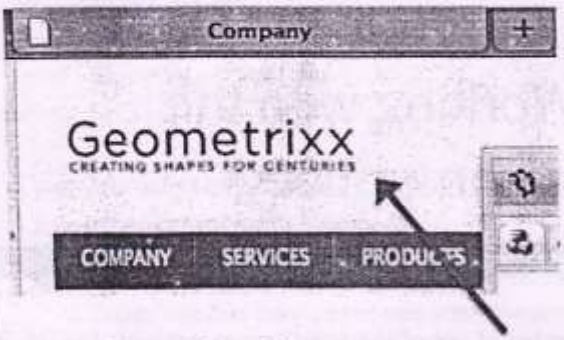
Page dialog edit bar

12. Enter the desired custom logo - then select OK.
    - <usb>/exercises/logo-component/logo.jpg

Design dialog of logo

13. Select **Edit** mode - then review the Page output of the logo Component.

- If successful, you should see the custom Page logo, similar to the image below.



Page preview of logo component

Congratulations! You have successfully created a logo Component, with a Design Dialog, that allows content authors/designers to write global binary content and have it displayed. Once again, this is a significant step in your development capabilities, as you are now able to create CQ5 Components that allow you to write binary content to the repository. In addition, since this content is global, all pages that implement the same Design/Template will have identical logo content.

Section 8-Working with the Foundation Components

Day provides a large number of out-of-the-box Components with its CQ5. They range anywhere from a simple title Component, to the more complex paragraph system Component, which we will cover later. These Components are located at:

- /libs/foundation/components

It is strongly recommended you perform a complete review of each Component before beginning actual development. Not only are they learning tools, providing excellent examples of what you can do as within CQ5 as a developer, but you may find an existing foundation Component that provides a completed solution as required by content authors. In addition, you can extend these Components to provide a custom solution, much like you did with the logo Component (which is an extension of the foundation *parbase* Component)

EXERCISE - Include the Breadcrumb Foundation Component

1. Open the file **content.jsp** in the Training Contentpage Component and replace the "trail" section with a <cq:include> of the foundation breadcrumb Component, similar to below - then Save.

content.jsp

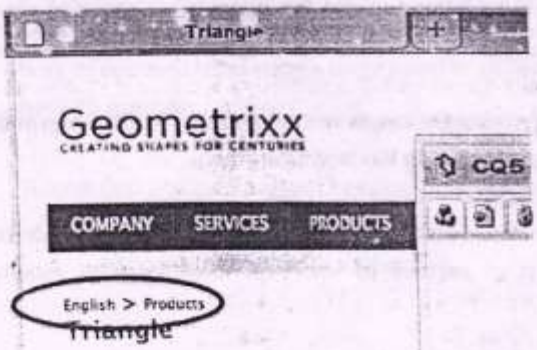
```
<%@include file="/libs/foundation/global.jsp" %>
<div class="container_16">
   <div class="grid_16">
          <cq:include path="breadcrumb" resourceType="foundation/
components/breadcrumb" />
     <cq:include path="title_node" resourceType="training/components/
title" />
   </div>
   <div class="grid_12 body_container">
      <div> par </div>
```

```
    </div>
    <div class="grid_4 right_container">
        <cq:include path="newslist" resourceType="training/components/
listChildren" />
<div> rightpar </div>
    </div>
    <div class="clear"></div>
</div>
```

NOTE: As always, it is a good idea to review what you have just included. Take note of the fact this Component only contains a Design Dialog (i.e. design_dialog), thus can only be modified in Design mode.

2. Test your script by requesting a Page in CQ5 Siteadmin that implements this foundation breadcrumb Component.

- If successful, you should see a breadcrumb trail, similar to the image below.
  - A Cube Page has been added below the Products Page merely for example. Add your own pages below the Products page and open them to test the breadcrumb.
- In Design mode, feel free to modify the depth at which the breadcrumb trail begins.
  - Default is level 2 (i.e. /content/trainingSite/en)



Page preview of breadcrumb

Congratulations! You have successfully included the breadcrumb foundation Component. Again, this is an exercise to get you familiar with not only the process of including a Component into a Template, but also to be more aware of the plethora of Components CQ5 provides out-of-the-box.

Extra Credit - Modify the Foundation Breadcrumb component

The following exercise tests your knowledge of creating design elements. The goal is to modify the Foundation Breadcrumb component so that an author with design privileges can set the path that determines the start of the breadcrumb trail.

1. As always - we never modify anything in /libs so: Create the ***foundation/components*** structure in ***/apps*** and copy the Foundation breadcrumb component to the ***.../components*** folder.

2. Modify the code to allow the author to enter the start of the crumbs as a path, instead of as a level. Use the ListChildren component we created yesterday as a guide.

3. Modify the Design Dialog to allow the author to enter a path, instead of a level.

4. Test your code.

NOTE

If you have previously entered a number (level) with the same property name (e.g.;absParent), the breadcrumb component will throw an error, as it is looking for a string that is a path and encounters a number. To fix this, merely navigate to the design element (/etc/designs/trainingDesign/jcr:content/breadcrumb and delete the ***absParent*** property. Also remember that the Geometrixx application also uses the foundation breadcrumb component. So you will have to modify the geometrixx design in the same fashion to get the geometrixx pages to render properly.

5. Examine the repository to see the values written into the design when author enters values into the design dialog box and saves them.

Congratulations! You have successfully modified the Foundation breadcrumb Component. Again, this is an exercise to use your knowledge to modify a Foundation component.

Extra Credit - Modify your topnav component

The following exercise tests your knowledge of creating design elements. The goal is to modify your local topnav component so that the code allows an author with design privileges to set the parent of the navigation by entering a path.

1. Make whatever modifications that you need to obtain the following result:

   Author/designer can enter a path design element to set the parent of the top navigation list of pages.

Congratulations! You have successfully modified your topnav Component. Again, this is an exercise to use your knowledge of the currentStyle to set and extract design elements and use them to render content objects.

Paragraph System

The paragraph system is the main content area of a web page. By adding the paragraph system to the template you structure and control the areas where content authors can add content.

The use of the CQ5 paragraph system Component is a necessity when wanting to have manageable and scalable Page content, without requiring excessive coding/ Template creation. This foundation ***parsys*** Component can be located at the path below:

- /libs/foundation/components/parsys

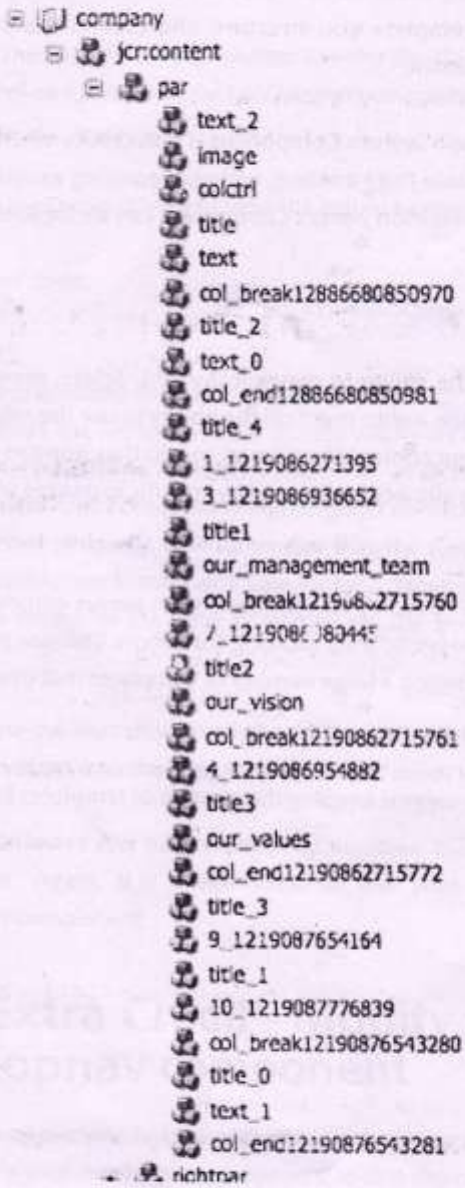
It allows content authors the ability to dynamically add, delete, move, copy, and paste "paragraphs" on a Page, not to mention the ability to use the column control Component to structure your content in columns. In addition, you can decide what "content" Components are allowed to be used by specific instances of the ***parsys***.

BEST PRACTICE

It is Adobe's STRONG opinion that the Foundation ***parsys*** component should be used as often as possible, thus allowing for simple Component configuration, as opposed to creating a large number of Templates that developers then have to maintain.

Adobe Best Practices suggest keeping the number of templates to a minimum.

The figure below demonstrates the content structure of the paragraph system, as it appears in the repository.



The Sidekick, Components and the Design

You may have noticed earlier, after assigning a new Design to your Web site, that no Components exist (are available) in the Components tab of your Sidekick. This is because you have yet to declare, in Design mode, which Components are allowed to be used by a paragraph system. Only then will Components be made available in the Sidekick.

EXERCISE - Include the Foundation Paragraph System Component

1. Open the file **content.jsp** in the Training Contentpage Component and replace the "par" section with a <cq:include> of the foundation paragraph system Component, similar to below - then **Save.**

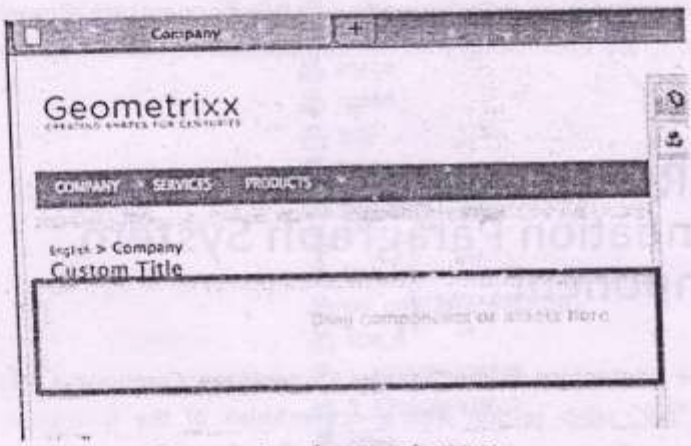
NOTE: As always, it is a good idea to review what you have just included. Take note of the fact this Component only contains a Design Dialog (i.e. design_dialog), thus can only be modified in Design mode.

content.jsp

```
<%@include file="/libs/foundation/global.jsp" %>
<div class="container_16">
   <div class="grid_16">
          <cq:include path="breadcrumb" resourceType="foundation/
components/breadcrumb" />
     <cq:include path="title_node" resourceType="training/components/
title" />
  </div>
  <div class="grid_12 body_container">
    <cq:include path="par" resourceType="foundation/components/
parsys" />
  </div>
  <div class="grid_4 right_container">
      <cq:include path="newslist" resourceType="training/components/
listChildren" />
     <div> rightpar</div>
  </div>
  <div class="clear"></div>
</div>
```

2. Test your script by requesting a Page in CQ5 Siteadmin that implements this foundation paragraph system Component.

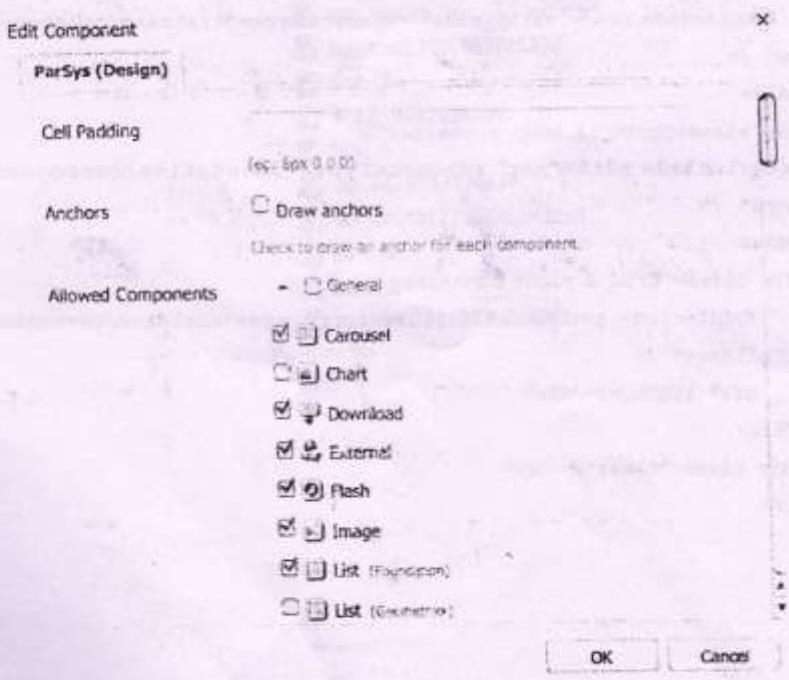
- If successful, you should see the paragraph system enabled, similar to the image below



Page preview of paragraph system

3. Select **Design mode** on your Sidekick - then select **"edit"** for the paragraph system Component to invoke the **Design Dialog.**

4. Select the Components you would like to allow for this instance of the paragraph system - then select **OK.**



Design dialog of the paragraph system

5. Select Edit mode - then review the Page output of the paragraph system Component.

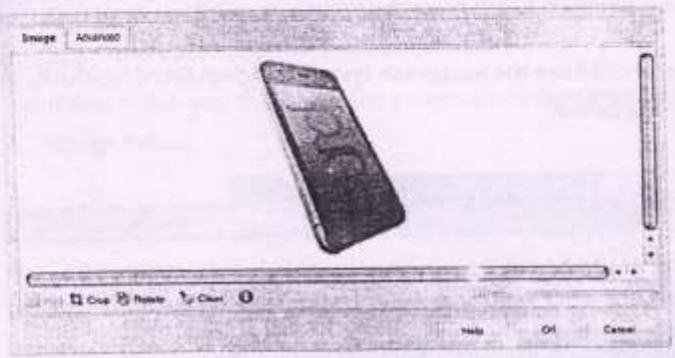
   - If successful, you should see the paragraph system and populated Sidekick, similar to the image below.

Page preview of paragraph system

NOTE: Notice how Components now appear in the Sidekick. These Components are now available to use in the paragraph system you recently added to the Tempiate.

6. Add content (i.e. paragraphs) to the Page by click-and-dragging desired Component(s) into the paragraph system - then write content using the available Dialog.

Page preview of click and drag to paragraph system



Dialog of **Image** component

Congratulations! You have successfully included the paragraph system foundation Component. Day cannot stress enough the importance of this Component and how it can greatly affect your productivity if used wisely. Please be sure evaluate its use and optimization for every Template that is developed/ deployed.

Section 9- CQ5 Development - Various Component Concepts

Dynamic Images

To create dynamic images, you need a mechanism to request the rendering of the image object. To achieve this, you add a selector to the request of the image of a page (e.g. /path/to/a/resource.title.png). Requests with such a selector have to be handled by an image processing mechanism. To achieve this, you will use Apache Sling's request processing mechanism.

You will need an image processing script or Java servlet that will handle all requests with the specific selector(s), e.g., "*title.png*". For the rendering of images with an overlay of text, the abstract Java servlet AbstractImageServlet is very helpful. You merely have to overwrite the method createLayer() to implement your desired logic.

Relevant functionalities (API calls) are:

- com.day.cq.wcm.commons.AbstractImageServlet
- com.day.cq.wcm.commons.WCMUtils
- com.day.cq.wcm.foundation.ImageHelper
- com.day.image.Font
- com.day.image.Layer

For more detailed information concerning the AbstractImageServlet and other Day related image processing capabilities, please review the CQ5 Javadocs provided with CQ5 WCM and on your USB.

EXERCISE - Create a Title Image Component

1. Create a new titleImage Component. Enter the desired Component "Label", "Title", "Description", etc, - then click **Finish**.
   - Label = titleImage
   - Title = Training Title Image
   - Description = Training Custom Image component
   - Group = Training
   - Allowed Parents = */*parsys

CRXDE new component dialog

2. Open the file titleImage.jsp, and enter some HTML and JSP code, similar to below - then **Save**.

titleImage.jsp

```
<%@include file="/libs/foundation/global.jsp"%><%
%><%@ page import="org.apache.commons.lang.StringEscapeUtils,
        com.day.cq.commons.Doctype,
        org.apache.sling.api.resource.ResourceUtil" %>
<%

    // first calculate the correct title - look for our sources if not
set in paragraph
    String title = properties.get("jcr:title", currentPage.getTitle());


    // use image title if type is "small" but not if diff should be
displayed
    if (properties.get("type", "small").equals("small")) {
```

```
        String suffix = currentDesign.equals(resourceDesign) ? "" :
"/" + currentDesign.getId();

        // add mod timestamp to avoid client-side caching of updated
images
        long tstamp = properties.get("jcr:lastModified", properties.
get("jcr:created", System.currentTimeMillis()));
        suffix += "/" + tstamp + ".png";

        String xs = Doctype.isXHTML(request) ? "/" : "";
%>
          <img src="<%= resource.getPath() %>.title.png<%= suffix %>"
alt="<%= StringEscapeUtils.escapeXml(title) %>"<%=xs%>><%

    // large title
    } else {
        %><h1><%= StringEscapeUtils.escapeHtml(title) %></h1><%

    }
%>
```

In the interest of time, Adobe Training has provided the Java servlet and background image that will achieve this image processing/text overlay. You will simply perform a code review in class. These items can be found in: <usb>/ exercises/image-title.

3. Copy the files ***title.png.java and bgImage.png*** from the USB (<usb>/ exercises/title-image-component) - then paste into the Title Image Component.

4. Review the ***title.png.java*** file for a better understanding of the AbstractImageServlet.


WARNING

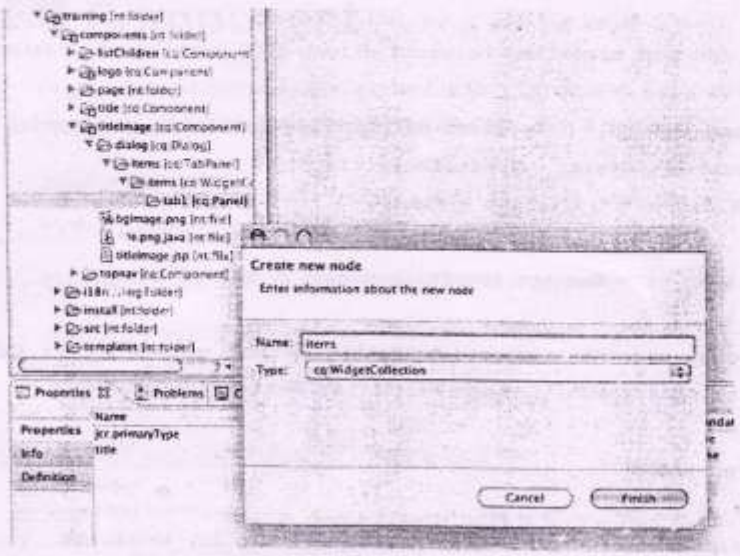
It is possible you will need to modify the Java package to match the current location of the Training Contentpage Component.

- e.g. apps.training.components.title

5. Create a component Dialog for the Title Image Component.

6. Right-click the *tab1* node - then select New, Node.

7. Enter the desired "Name" (items) and "Type" (cq:WidgetCollection).



CRXDE new node dialog

8. Create a new node under the newly created *items* node.
   - Name = title
   - Type = cq:Widget

9. Add the following properties to the *title* node:
   - The property that will define where the content is stored
     - Name = name
     - Type = String
     - Value = ./jcr:title
   - The property that will define the Widget type (i.e. user interface control)
     - Name = xtype
     - Type = String
     - Value = textfield
   - The property that will define the label applied to the Widget
     - Name = fieldLabel
     - Type = String
     - Value = Title

10. Create another node under the *items* Widget Collection.
    - Name = type
    - Type = cq:Widget

11. Add the following properties to the *type* node. You will notice a new xtype being used here. We are introducing a dropdown selection widget.

- The property that will define where the content is stored
  - Name = name
  - Type = String
  - Value = ./type
- The property that will define the Widget type (i.e. user interface control)
  - Name = xtype
  - Type = String
  - Value = selection
- The property that will define the label applied to the Widget
  - Name = fieldLabel
  - Type = String
  - Value = Type/Size
- The property that will define the type of Selection Widget
  - Name = type
  - Type = String
  - Value = select

12. Create a node under the *type* Widget.

- Name = options
- Type = cq:WidgetCollection

13. Create a node under the *options* Widget Collection.

- Name = large
- Type = nt:unstructured

14. Add the following properties to the *large* node

- The property that will define where the content is stored
  - Name = text
  - Type = String
  - Value = Large
- The property that will define the Widget type (i.e. user interface control)
  - Name = value
  - Type = String
  - Value = large

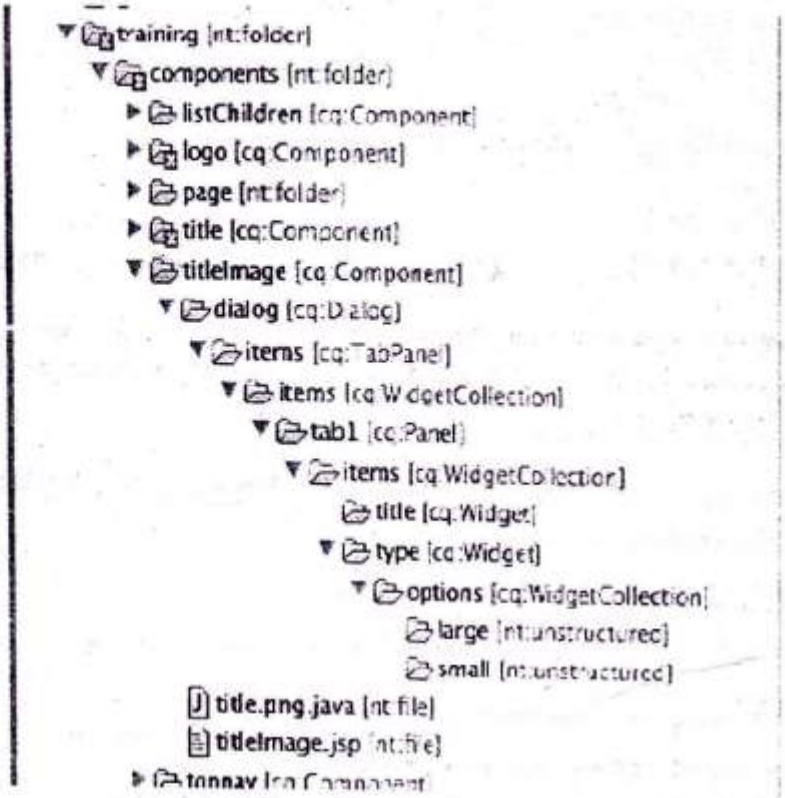
15. Create another node under the *options* Widget Collection.

- Name = small
- Type = nt:unstructured

16. Add the following properties to the *small* node

- The property that will define where the content is stored
  - Name = text
  - Type = String
  - Value = Small

- The property that will define the Widget type (i.e. user interface control)
  - Name = value
  - Type = String
  - Value = small



17. Make sure that the Title Image component is in your Training Component Group and appears in the Sidekick.

18. Test your script by dragging the Title Image component into the Paragraph System and entering a custom title into the dialog box. Choose "small" for the type and click **OK**.

19. Test your new functionality by entering a custom title and choosing "small" for the type.

20. Review the Page output of the title content.

    - If successful, you should see the custom Page title, similar to the image below.

Page view with custom title

Congratulations! You have successfully created a Title Image Component, with a Dialog, that allows content authors to write content and have it displayed as an image.

cq:EditConfig

We introduced the cq:EditConfig construct earlier when we talked about Component child nodes. Now we will explore the cq:EditConfig functionality in more detail.

cq:EditConfig is used to configure content input actions when the dialog box is not in control. For example:

- Drag-and-drop from the Content Finder
- In-line Editing
- Refresh of a dialog box or page after an author action

To demonstrate functionality of cq:EditConfig, we will use drag-and-drop from the Content Finder as an example. To drag assets from the Content Finder to a component, the following are required:

- Node of type cq:editConfig that is a child node to the cq:component node
- Configuration node that is a child of the cq:EditConfig node. This node defines what action you are configuring. In this example: drag-and-drop from the Content Finder - cq:dropTargets
- Asset node that is a child of the cq:dropTargets node. This node defines what types of assets the paragraph will accept. In this example: assets of type image.

On the asset node you must define 3 properties:

- accept - media types to be accepted (image/gif, image/jpeg, etc.)
- groups - groups in the Content Finder from where assets can be accepted
- propertyName - where the asset reference will be stored (e.g., ./fileReference)

cq:include xtype

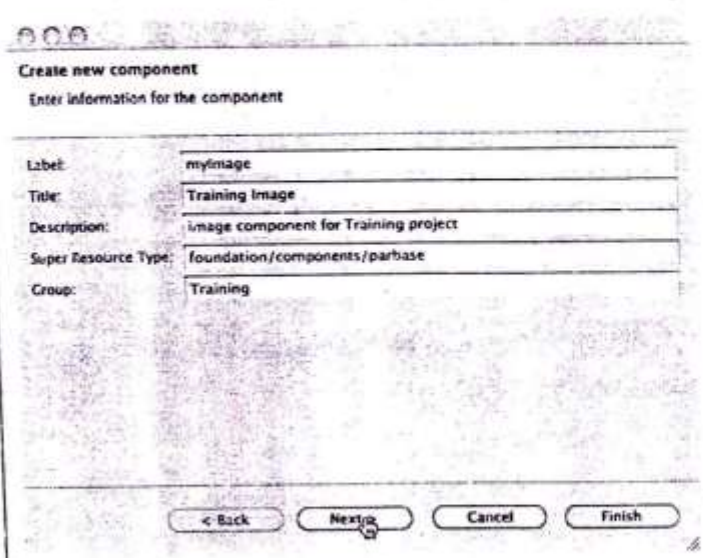
At this point we will introduce an alternate mechanism for reusing dialog structures. Unlike a copy of an existing Dialog structure (which we used in the logo component) or declaring a superType of the entire component, using the **xtype** of *cqinclude* will reference a part of the existing dialog structure, allowing the custom component and its dialog to take advantage of any future changes to the referenced structure.

EXERCISE - Creating a Simple Image Component

In this exercise we create simple Image component as an example of a component that allows drag-and-drop of assets from the Content Finder onto the paragraph thumbnail placeholder. As a result of this exercise, you will feel more confident building a more complex component that renders binary data and allows drag-and-drop.

1. Right-click /apps/<application name>/component - then select New, Component.

2. Enter the desired Component "Label", "Title", "Description" - then click Finish.
   - Label = myImage
   - Title = Training Image Component
   - Description = Simple Image Component
   - Super Resource Type = foundation/components/parbase
   - Group = Training
   - Allowed Parents = */*parsys



3. Enter the following code into **myImage.jsp**:

myImage.jsp

```
<%@ page import="com.day.cq.wcm.foundation.Image" %>
<%@include file="/libs/foundation/global.jsp"%>
<%
    Image image = new Image(resource);

    image.setSelector(".img"); // use image script

    image.draw(out);
%><br>
```

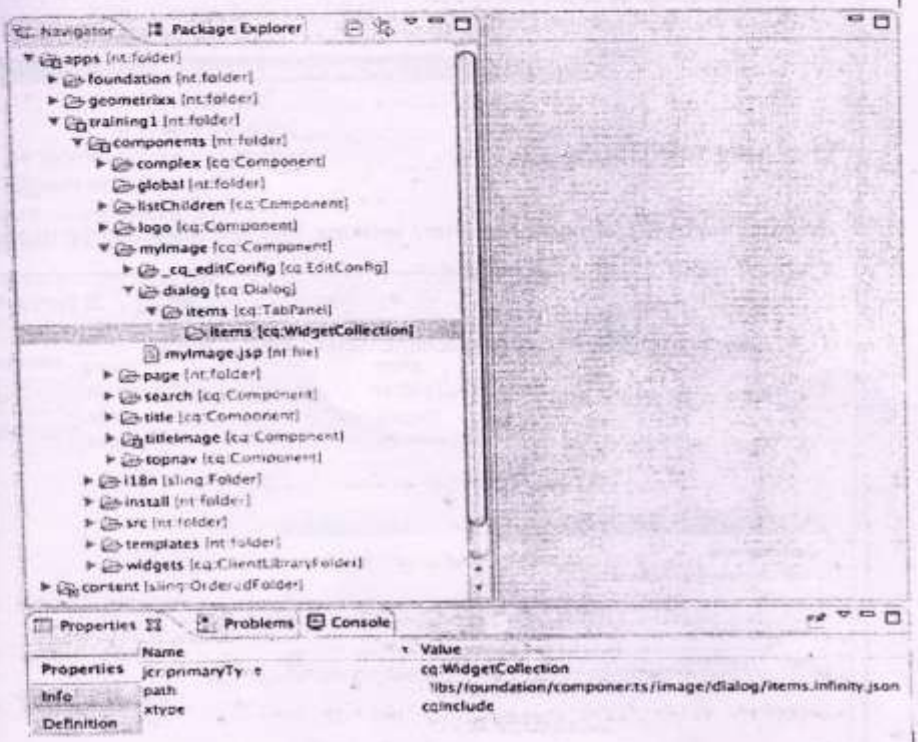
Create a Component Dialog Box for our Image component

1. Right-click your myImage Component and select **New... Dialog**. As before, the Dialog wizard will create the initial structure for the component dialog box.

We will use an **xtype** of *cqinclude* to refer to the smartimage widget in the Foundation Image Component Dialog.

2. Define the following properties on the *items* parent of the *tab1* node:
   - The property that defines widget reference
     - Name = xtype
     - Type = String
     - Value = cqinclude
   - The property that defines the path to the reference
     - Name = path
     - Type = String

•Value = /libs/foundation/components/image/dialog/items.infinity.json



3. Delete the *tab1* node. We don't need this node because we are now referring to the Foundation Image Component dialog structures.

Add the myImage Component to the Sidekick

1. Select **Design** mode by clicking on the ruler icon at the bottom of the Sidekick.
2. Click Edit button on **Design of par** edit bar.
3. Select Training Group
4. Return to **Edit mode** by expanding the Sidekick.

Test your Image component.

1. Drag the Training Image component onto a page.
2. Try to drag an image onto the thumbnail. You will note that the drag-and-drop does not work. This is because we have not yet defined the *editConfig* nodes and properties that will support this functionality.
3. Drag-and-drop into the dialog box does work. This is because of the inherent functionality of the ***smartimage*** widget. Double-click the thumbnail to open the dialog box.

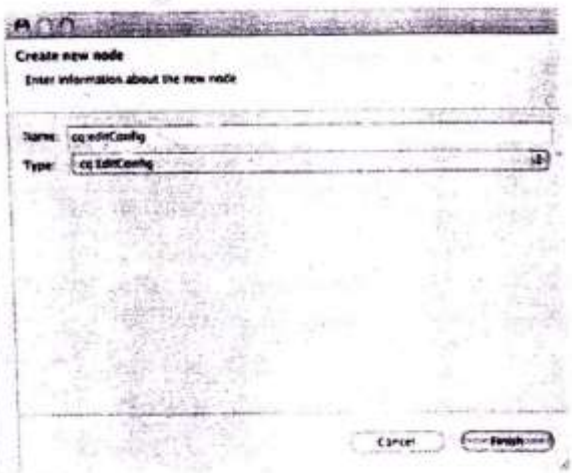
4. Drag an image onto the dialog box.

5. Click **OK**.

Defining editConfig

Now that we have our basic component working, let's add an editConfig to allow drag-and-drop from the Content Finder.

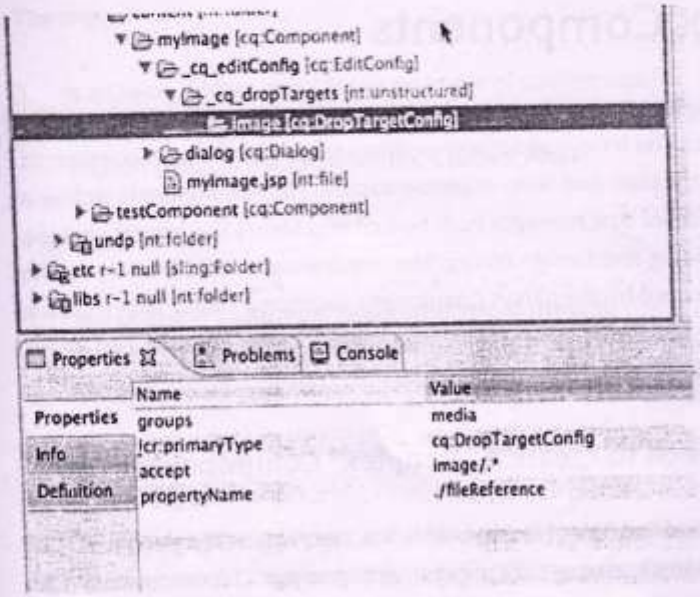
1. Right-click on your Image component. Select **New ... Node**.
   - Name: cq:editConfig
   - Type: cq:EditConfig



2. Right-click the new *cq:editConfig* node and select **New ... Node**.
   - Name: cq:dropTargets
   - Type: nt:unstructured

3. Right-click the new *cq:dropTargets* node and select **New ... Node**
   - Name: image
   - Type: cq:DropTargetConfig

And then enter the *groups*, *accept*, and *propertyName* properties as shown in the figure below:



- Name = groups
- Type = String
- Value = media

- Name = accept
- Type = String
- Value = image/.*

- Name = propertyName
- Type = String
- Value = ./fileReference

4. Test the new functionality by dragging a new Training Image component onto the page and then dragging an image from the Content Finder onto the new paragraph.

Congratulations! You have successfully created a Simple Image component that allows drag-and-drop from the Content Finder. As an exercise, examine a few of the other components in /libs/foundation/components that support drag-and-drop from the Content Finder. For example, the Download, Slideshow, and List components.

Complex Components

So far each component that we have created has taught us one thing. As you know, that is not real life. The following section explores turning end user requirements into a development plan and then implementation. The requirements define a "complex" Component that manages both textual and binary (i.e. image) content, and has both a Dialog and Design Dialog. The requirements include allowing the Component to be used by the parsys Component, creating a Dialog with multiple tabs, enabling the functionality offered by the Content Finder (i.e. drag-and-drop), and other configurations.

When will I need to create a "complex" Component?

Since the creation of "complex" Components is a common occurrence in CQ5, it would benefit you to observe a mock requirements analysis. This exercise provides just that. First, you will observe the needs of a user. Secondly, you will observe how a Day Solution Engineer may translate those needs.

REQUIREMENTS: A Component that can be dropped into a paragraph system and displays an image, rich text, and the path of a Page in the system.

The image:

1. Must be editable by a content author.
2. Can be dragged-and-dropped from the Content Finder

The rich text:

1. Must be editable by a content author
2. Must allow for tables to be created.
3. Must have a default value of "This is some text."

The path of a Page:

1. Must be the same for every instance of this Component, yet editable by a "super" author.
2. Must live under the Web site structure "/content/trainingSite".
3. Widget should have property regexText with an error message if regular expression fails.

SOLUTION ENGINEER'S TRANSLATION: A paragraph system Component that allows for the writing and displaying of three properties (2 paragraph properties, 1 design/style property), and has both a Dialog and Design Dialog.

The image:

1. Is a Dialog Widget, most likely an xtype of smartimage.
2. Must be configured in the Component's cq:editConfig to allow for dragging-and-dropping of images from the Content Finder.

The rich text:

1. Is a Dialog Widget, most likely an xtype of richtext.
2. Widget should enable all the features of the rich t :xt editing plugin table.
3. Widget should populate property defaultValue with "This is some text."

The path of a Page:

1. Is a Design Dialog Widget, most likely an xtype of pathcompletion.
2. Widget should have property regex with a regular expression validating the user's input (e.g. "/^\/content\/training\/(.)*$/").
3. Widget should have property regexText with an error message if regular expression fails.

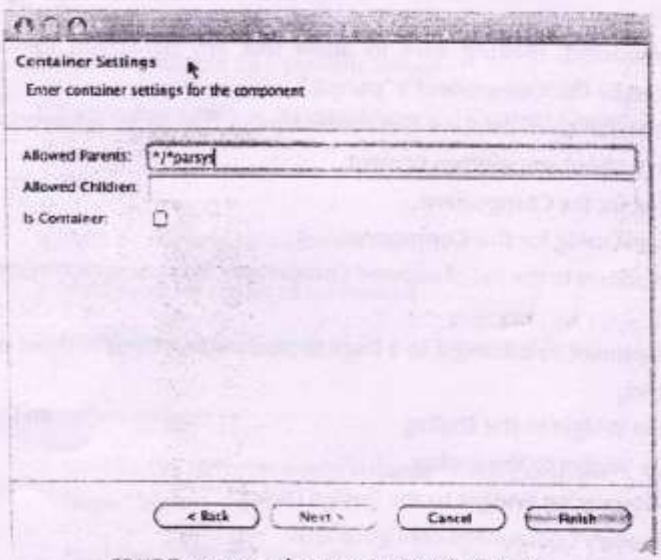
Though the order of sequence can differ, an Adobe Solution Engineer would most likely:

1. Create a Component, making sure to allow that any paragraph system Component can be this Component's "parent."
2. Edit the default JSP so it displays the content/properties in an appropriate manner, even without any written content.
3. Create a Dialog for the Component.
4. Create a Design Dialog for the Component.
5. Add this Component to the list of allowed Components for a paragraph system Component.
6. Test this Component by adding it to a Page to observe its output without any written content.
7. Add a rich text Widget to the Dialog.
8. Add an image Widget to the Dialog.
9. Add a path completion Widget to the Design Dialog.
10. Perform necessary Component configurations.
11. Test this Component by writing content using the newly created Dialog and Design Dialog.

EXERCISE - Create a "Complex" Component

1. Create a new complex "content" Component.

CRXDE new complex component dialog - 1



CRXDE new complex component dialog - 2

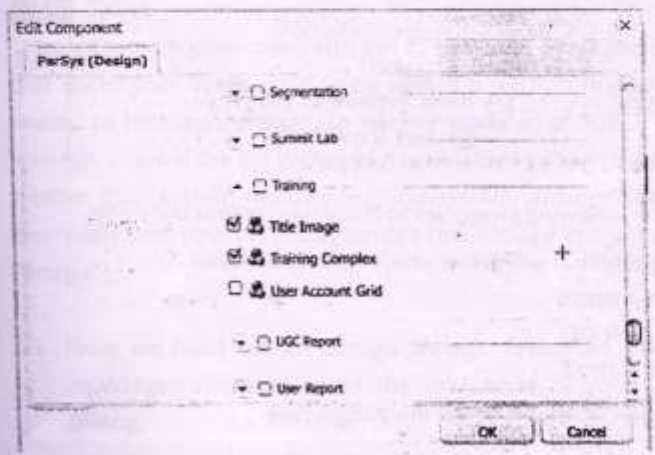
2. Open the file **complex.jsp**, and enter some HTML and JSP code, similar to below - then **Save.**

complex.jsp

```
<%@include file="/libs/foundation/global.jsp"%>
<%@ page import="com.day.cq.wcm.foundation.Image" %>
<%
// getting the image from the Dialog/resouce
// notice the use of the second parameter "image" -- this signifies
that the
// image will live on a resource (called image) below the requested
resource
Image image = new Image(resource, "image");
// setting the selector so that the "parbase" can work its magic
image.setSelector(".img");
// getting the rich text from the Dialog
String text = properties.get("text", "TEXT NA");
// getting the path from the Design Dialog
String path = currentStyle.get("path", "PATH NA");
%>
<h2><%= path %></h2>
<%= text %><br />
<%
image.draw(out);
%>
```

3. Create a Dialog and Design Dialog for the complex Component.
   - You will worry about Widgets and configurations later - the focus now is to see your new Component in action.

4. Add your complex Component to the paragraph system Component in Design mode.

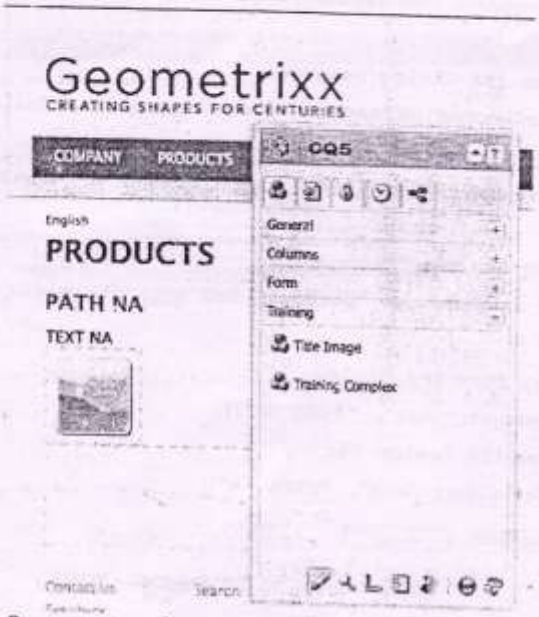


Design dialog of the paragraph system

NOTE: For this exercise, do not concern yourself greatly with how the content is displayed, as this can easily be altered via code changes and/or CSS.

5. Test your script by adding an instance of this Component (i.e. paragraph) to the paragraph system Component of a Training Page.

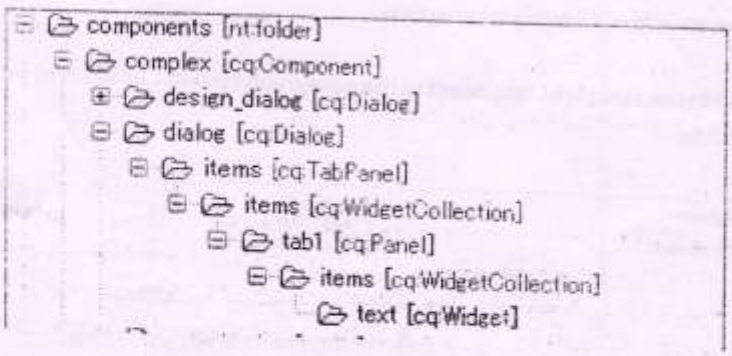
- If successful, you should see the default complex Component output, similar to the image below.



Page preview of complex component - no content

6. Create an **items** node (nodeType cq:WidgetCollection) under the *tab1* node of your Component's Dialog.

7. Create a **text** node (nodeType cq:Widget) under the newly created *items* node.



8. Assign the following properties to the newly created *text* node:

- The property that will define where content is stored
  - Name = name
  - Type = String
  - Value = ./text
- The property that will define the Widget type
  - Name = xtype
  - Type = String

NOTE: Notice how your complex Component is listed under the Training group. This is because you declared such during the creation of your Component.

Concepts 9-17

- Value = richtext

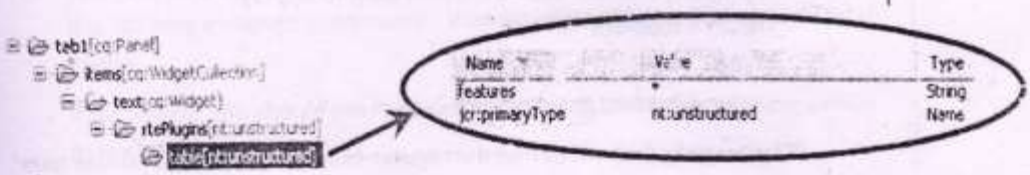
- The property that will define to hide the label of the Widget
  - Name = hideLabel
  - Type = Boolean
  - Value = true

- The property that will define the the default value of the Widget
  - Name = defaultValue
  - Type = String
  - Value = This is some text.

9. Create an **rtePlugins** node (nodeType nt:unstructured) under the newly created *text* node.

10. Create a **table** node (nodeType nt:unstructured) under the newly created *rtePlugins* node.

11. Assign the features property (Type = String, Value = *) to the newly created *table* node.



CRXDE rtePlugin table plugin structure and property

12. Copy the nodes *tab2* and *tab3* under the node */libs/foundation/components/textimage/dialog/items* - then paste to the complex Component's Dialog so that they are a peer of *tab1* (e.g. /apps/training/components/complex/dialog/items/items).

- tab2 = the smartimage tab
- tab3 = advanced image properties

NOTE

Once again, it is often more efficient to copy-and-paste existing Dialogs/Widgets that meet your needs. That being said, it is wise to review what you have just copied to better understand the internal workings of CQ5. If you examine closely enough, you will see the Widget is actually storing image related content at a level deeper than current resource (e.g. ./image/file, ./image/fileReference, etc.). This ties nicely with your previously written code (Image image = new Image(resource, "image");).

13. Now we build out the Design Dialog. Create an **items** node (nodeType cq:WidgetCollection) under the *tab1* node of your Component's Design Dialog.

14. Create a **path** node (nodeType cq:Widget) under the newly created ***items*** node.

15. Assign the following properties to the newly created *path* node:

- The property that will define where content is stored
  - Name = name
  - Type = String
  - Value = ./path
- The property that will define the Widget type
  - Name = xtype
  - Type = String
  - Value = pathfield
- The property that will define the label applied to the Widget
  - Name = fieldLabel
  - Type = String
  - Value = Enter a Path
- The property that will define the root path to display
  - Name = rootPath
  - Type = String
  - Value = /content/training-site
- The property that will define the regular expression used to evaluate user input
  - Name = regex
  - Type = String
  - Value = /^\/content\/training-site\/(.)*$/
- The property that will define the error message if a user's input fails the regular expression
  - Name = regexText
  - Type = String
  - Value = Please insert a Page that "lives" under /content/trainingSite

16. Copy the node /libs/foundation/components/textimage/cq:editConfig - then paste to the root node of your complex Component (e.g. /apps/training/components/complex).

- Enables drag-and-drop capabilities from the Content Finder

In order to be able to drag-and-drop assets from the Content Finder to a Component on a Page, there must be a drop targets configuration node called cq:dropTargets (of type nt:unstructured) below the edit configuration node (cq:editConfig) of a Component.

17. Using CRXDE, navigate to

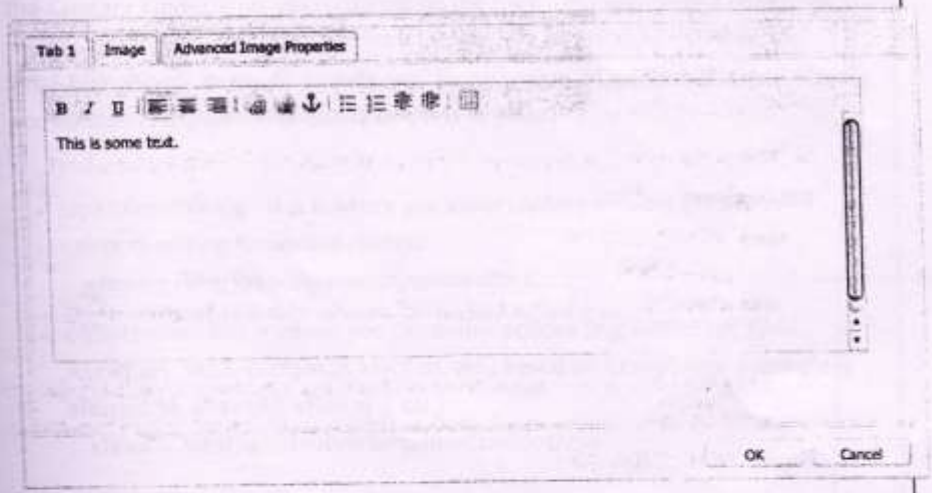
<path-to-complexcomponent>/cq:editConfig/cq:dropTargets/image

Validate that the image node has the following properties:

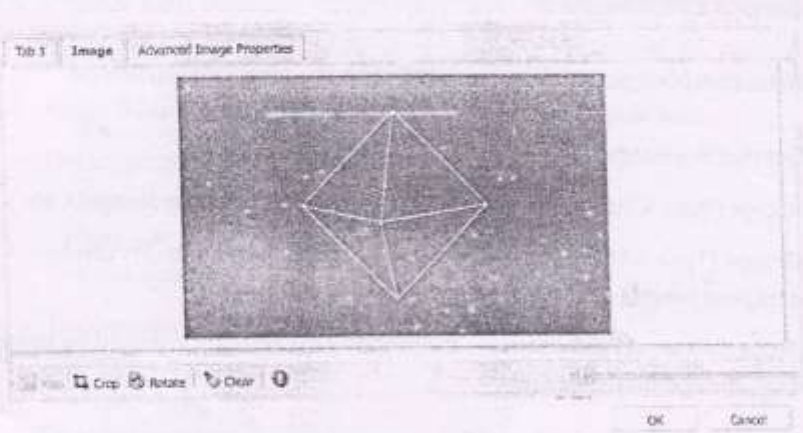
- Accept (Type = String) - the media types to be accepted (e.g. image/.*, etc.)
- Groups (Type = String) - the groups in the Content Finder assets can be accepted from (e.g. media)
- PropertyName (Type = String) - the property the reference should be stored (e.g. ./image/fileReference)

18. Navigate to the ***parameters*** node that is a child to the ***image*** node. Change the value of the *sling:resourceType* property to have the value: *training/components/complex*. Remember that the *sling:resourceType* property should reference the location to find a rendering script.

19. Test your script/Dialog by requesting a Page in CQ5 Siteadmin that implements this Training complex component - then interact with the Dialog and Design Dialog.
    - If successful, you should see Page output, a Dialog and Design Dialog similar to the images below.

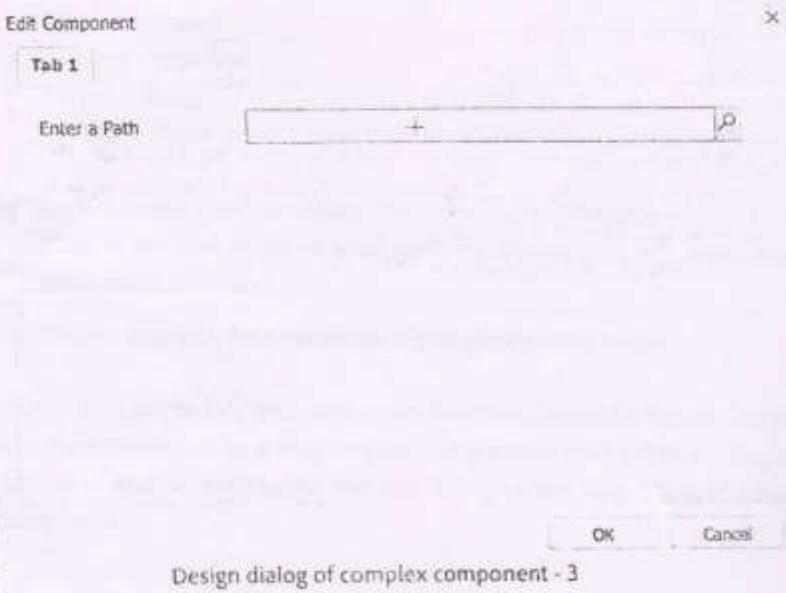


Dialog of complex component - 1



Dialog of complex component - 2

Dialog of complex component - 3



Design dialog of complex component - 3

Page preview of complex component - with content

Congratulations! You have successfully created a complex Component that contains a Dialog, Design Dialog, default values, custom configuration, and validation of user input, in addition to enabling drag-and-drop functionality from the Content Finder. Give yourself a pat on the back - this was quite a challenge.

That being said, it would benefit you to be aware of additional cq:editConfig possibilities/variations. For example:

- cq:inplaceEditing - this is where you allow content authors to have in-context-editing for textual content
  - review /libs/foundation/components/text
- cq:listeners - this is where you can define actions (e.g. REFRESH_PAGE, REFRESH_SELF, custom JS function, etc.) based on Component events (e.g. afterinsert, afteredit, aftercopy, etc.)
  - review /libs/foundation/components/slideshow

Mobile Functionality

Mobile channels gained considerable importance in the last few years. With CQ you can manage websites for mobiles devices, view a mobile page in a way that emulates a mobile device or switch between several views.

CQ enables you to author webpages with an iPad. You can open a page on your iPad with the embedded Safari browser, move paragraphs around, add pictures, change paragraph titles and once done with editing you can publish the page.

The CQ5 GO is an iPhone/iPad application available through the App Store that lets you manage your CQ workflow inbox on the iPhone or an the iPad. With CQ5 GO you can configure several inboxes accessing several CQ instances. Once the inbox is configured, you can manage the items that are in your inbox: you can navigate between items, add a comment and complete a step, step back, delegate the item to a co-worker or view the payload.

Mobile Emulators

CQ5 provides several device emulators that allow you to see how your web content will appear on those mobile devices. The default emulators can be found in the following location in the repository:

/libs/wcm/mobile/components/emulators

Those emulators are grouped, based on sets of capabilities, e.g., "supports images" or "support rotation".

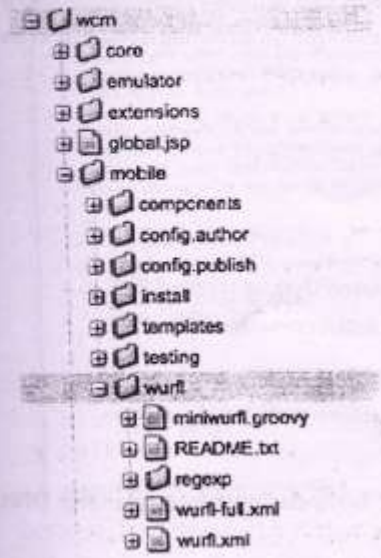
WURFL

WURFL stands for Wireless Universal Resource FiLe. CQ5 uses *wurfl* to determine which rendition of the page will be rendered to the mobile device. *Wurfl* is an xml database that stores the different web browsing capabilities of a mobile device based on its User Agent. For CQ5, *wurfl* is used to match the User Agent of the mobile device that is browsing the CQ5 pages to the CQ5 rendering engine that displays the page. You can find more information about *wurfl* at this url:

http://wurfl.sourceforge.net/

The CQ5 *wurfl.xml* file can be located at this URI:

/libs/wcm/mobile/wurfl



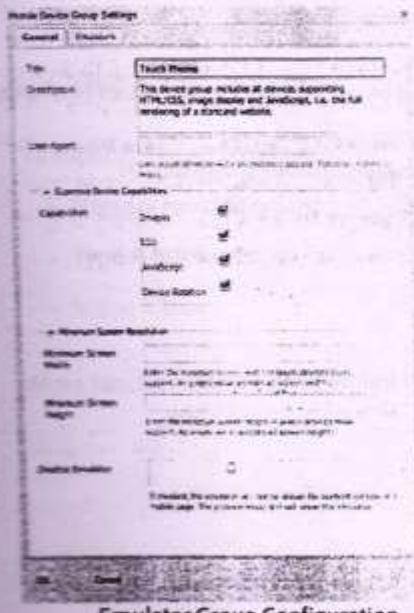
Each page in the mobile website allows you to specify the group of mobile devices that will be rendering the page. When the mobile page rendering component inherits from the Foundation mobile page component

/libs/wcm/mobile/components/page

the emulator functionality is automatically integrated in the page.

Emulator Groups

Emulator groups can be configured to specify the features that can be expected from it as well as the minimum screen resolution that can be expected from devices of such group:



Emulator Group Configuration

CQ enables authors to view a page in an emulator that simulates the environment in which an end-user will view the page, as for example on a mobile device or in an email client.

Emulator Framework

The CQ emulator framework:

- Provides content authoring within a simulated User Interface (UI), e.g. a mobile device or an email client (used to author newsletters).
- Adapts the page content according to the simulated UI.
- Allows the creation of custom emulators.

The emulator works by wrapping the HTML body contents into emulator DIVs. See the following example HTML code:

```
<body>
<div id="wrapper" class="page mobilecontentpage ">
    <div class="topnav mobiletopnav">
    ...
    </div>
    ...
</div>
</body>
```

When you create a mobile emulator, it will inherit many of its characteristics from the Foundation emulator component's init.html.jsp. You can find the Foundation emulator at:

/libs/wcm/mobile/components/emulators/base

The emulator appearance is controlled by a CSS client library. As an example, refer to

/libs/wcm/mobile/emulators/android/css/source/emulator.css

emulator.css

```
/*
 * Copyright 1997-2009 Day Management AG
 * Barfuesserplatz 6, 4001 Basel, Switzerland
 * All Rights Reserved.
 *
 * This software is the confidential and proprietary information of
 * Day Management AG, ("Confidential Information"). You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Day.
 */
#cq-emulator.android.vertical {
    background: url("../../resources/bg.png") no-repeat;
    height: 454px;
    padding: 131px 0 202px;
    width: 388px;
}

#cq-emulator.android.horizontal {
    background: url("../../resources/bg_horizontal.png") no-repeat;
    padding: 33px 0 38px;
    width: 862px;
}

#cq-emulator-content.android {
    background-color: #fff;
```

If needed, define a JS client library, for example to define a specific plugin:

- Name = js
- Node Type = cq:ClientLibrary

As an example, you can refer to the node:

/libs/wcm/mobile/components/emulators/base/js

If the emulator supports specific functionalities defined by plugins (like touch scrolling), create a configuration node below the emulator:
name = cq:emulatorConfig, node type = nt:unstructured and add the property that defines the plugin. For example, to support Rotation:

- Name = canRotate
- Type = Boolean
- Value = true

To support Touch Scrolling:

- Name = touchScrolling
- Type = Boolean
- Value = true

More functionalities can be added by defining your own plugins.

Mobile components

Creating mobile components is done actually in a very similar way than creating regular components.
The only difference between a regular component and a mobile component is that the component needs to be aware if the emulator that is used to display the page is able to handle the kind of content that he needs to display.
For that we have to main classes that come quite handy every time that we are developing mobile components.

- com.day.cq.wcm.mobile.core.MobileUtil
- com.day.cq.wcm.mobile.api.device.capability.DeviceCapability

The class MobileUtil provides you with the following methods

static DeviceGroup	getDefaultDeviceGroup(Page page) This method finds the device groups currently assigned to the given page or its closest ancestor.
static String	getDeviceGroupSelector(SlingHttpServletRequest request) Returns the last selector found in the request URL.
static boolean	hasCapability(SlingHttpServletRequest request, DeviceCapability capability) This method retrieves the DeviceGroup from the current request and checks the group whether it offers the given capability.
static boolean	isDeviceGroup(Page page) Checks whether the given Page represents a WCM Mobile Device Group.
static boolean	isDeviceGroup(Resource resource) Checks whether the given Resource represents a WCM Mobile Device Group.
static boolean	isMobileRequest(SlingHttpServletRequest r) True if the request's user agent is a mobile device.
static boolean	isMobileResource(Resource r) True if given Resource is to be handled as a mobile resource.
static boolean	isNoMatch(String[] selectors)

The most important method for us is the method hasCapability, which will tell us if we should render or not a piece of content depending on the capability of the emulator to handle the kind of content.

The interface DeviceCapability on the other hand has the following fields

Field Summary	
static DeviceCapability	CAPABILITY_CSS
static DeviceCapability	CAPABILITY_DEVICEROTATION
static DeviceCapability	CAPABILITY_IMAGES
static DeviceCapability	CAPABILITY_JAVASCRIPT

It is very easy then to use both elements in order to create a component that will render the appropriate content based on the device capabilities of the mobile device (and the emulator) used to display the page.

EXERCISE: Create a mobile time component

1. Right-click /apps/training/components - then select New, Component.
2. Enter the desired Component "Label", "Title", "Description" - then click Finish.
   - Label = mobiletime
   - Title = Mobile Time
   - Description = Component that displays the current time
   - Group= Mobile

CRXDE create Mobile Time component

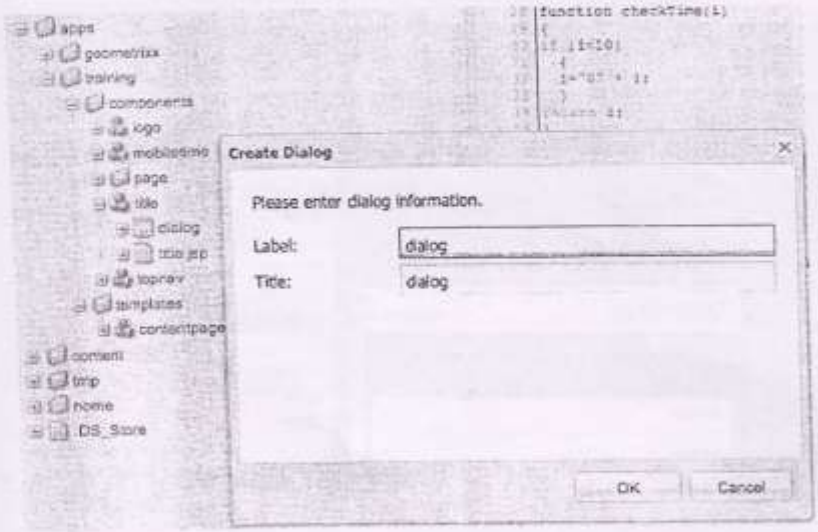
3. Click on next and then on next enter */*parsys for allowed parents.
4. Click on next again and your component will be created.
5. Click on next again and your component will be created.

mobiletime.jsp

```
<%@    page    import="com.day.cq.wcm.mobile.api.device.capability.
DeviceCapability,
                    com.day.cq.wcm.mobile.core.MobileUtil" %> <%
%>
<%@include file="/libs/foundation/global.jsp"%>
```

```
<%
    // only show the times if the device supports javascript
 if (MobileUtil.hasCapability(slingRequest, DeviceCapability.
CAPABILITY_JAVASCRIPT)) {
%>
     <script type="text/javascript">
       function startTime() {
         var today=new Date();
         var h=today.getHours();
         var m=today.getMinutes();
         var s=today.getSeconds();

         // add a zero in front of numbers<10
         m=checkTime(m);
         s=checkTime(s);
         document.getElementById('timing').innerHTML=h+":"+m+":"+s;
         t=setTimeout('startTime()',500);
        }

        function checkTime(i) {
          if (i<10) {
            i="0" + i;
          }
          return i;
        }
     </script>

     <div id="timing" style="color:yellow; font-style:bold"></div>

     <script type="text/javascript">
       startTime();
     </script>
<% } else { %>

    <p>Your device does not support javascript</p>
<% } %>
```

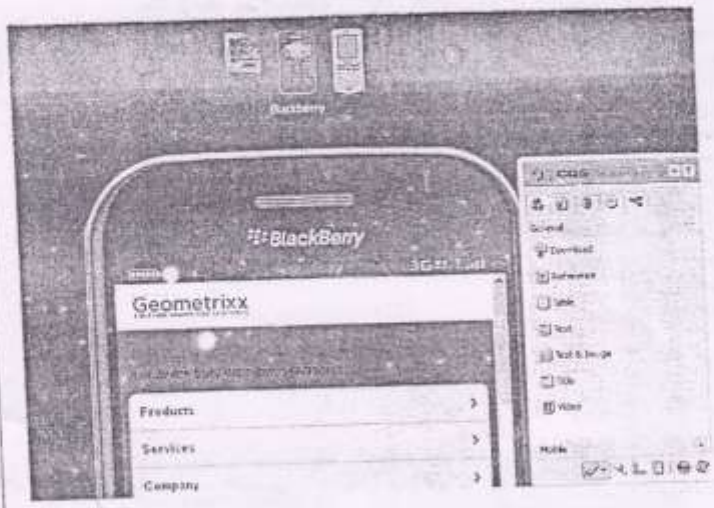
6. Select your component and create am empty dialog for it (we need this so that it appears in the list of components available to our paragraph system).



7. Go into Design mode and add your mobiletime component to the design. Return to edit mode.

8. From the sidekick, drag the Mobile Time component into the paragraph system of a mobile page. See the result in the devices that support javascript.

And those who don't support javascript:

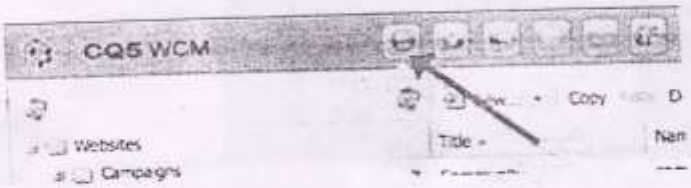


Congratulations! You have successfully created a mobile component and tested it into your different mobile emulators.

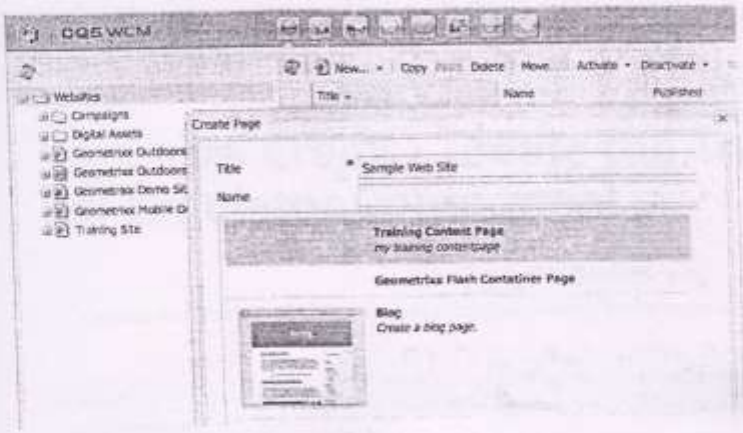
Creating Mobile Web Sites

EXERCISE - Re-purpose desktop web site content for mobile devices

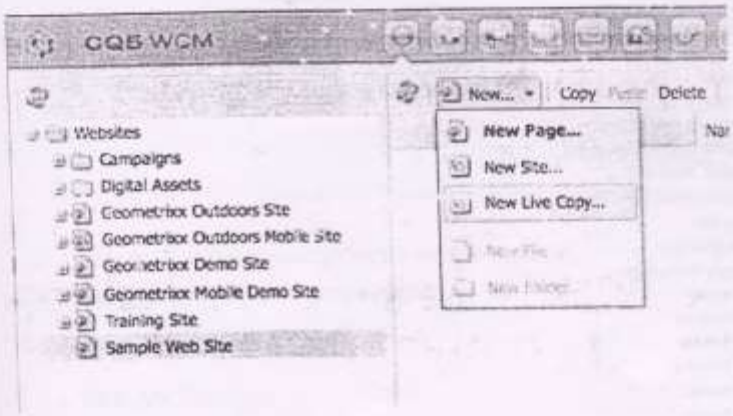
1. Navigate to the Websites (Site Admin) Console - http://localhost:4502/libs/wcm/core/content/siteadmin.html.



2. Create a new Page under Websites. Choose the Training Content Page template.



3. Create a new Live Copy under your new Sample Web Site page. Select New... New Live Copy

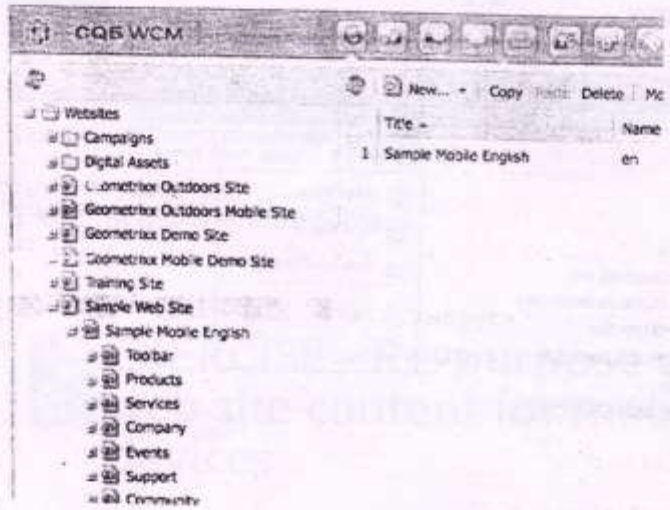


4. Give your Mobile Live Copy a Title and an optional name.

5. Select the Geometrixx Demo site (/content/geometrixx/en) as the source.

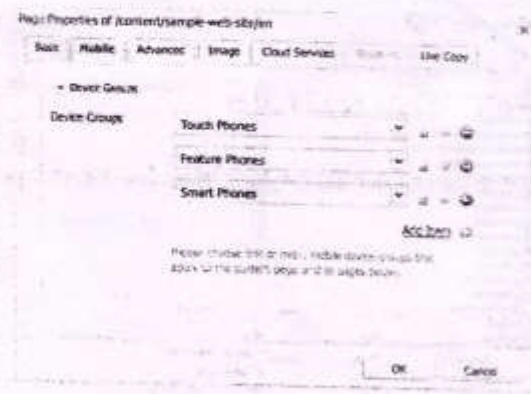
6. Add two Sync Configs: Standard Rollout Config and Geometrixx Mobile.

7. Click **CREATE**. You should see the following web site hierarchy appear:



8. Now you need to enable the site for the Mobile Device Groups. Open the page properties for the root of your site.

9. Select the Mobile Tab. Add all of the Mobile Device Groups.



10. Open one of the new pages. Note that the content from the desktop web site is now repurposed to the mobile site.

Congratulations! You have successfully created a mobile site that repurposes desktop website content.

Extra Credit - Create the Training Emulator Component

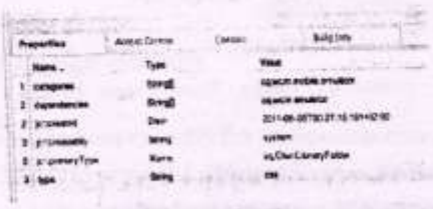
1. Create the folder emulators under */apps/training/components*.

2. Below */apps/training/components/emulators*, create a ***myemulator*** component. (New...Component)
   - label: myemulator
   - title: My Emulator
   - description: Training Emulator
   - resourceSuperType: wcm/mobile/components/emulators/base

3. **Delete** the default jsp, ***myemulator.jsp***. We won't need it. Everything we need is in the file ***init.html.jsp*** that we inherit from the SuperType "base". This ***init.html.jsp*** contains the emulator initialisation code and serves the same purpose as *init.jsp* that we use in the regular components.

Create the emulator Client Library Folder

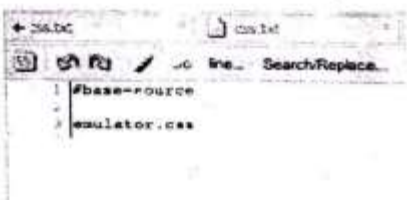
1. **Create** a css node (nodeType cq:ClientLibraryFolder) under the *myemulator* component node.



2. **Define** the following properties on the newly created ClientLibraryFolder:

   - The property that will define how the client library is registered
     - Name = categories
     - Type = String []
     - Value = cq.wcm.mobile.emulator
   - The property that will define the Library dependencies
     - Name = dependencies
     - Type = String []
     - Value = c .wcm.emulator
   - The property that will define the Library type
     - Name = type
     - Type = String
     - Value = css

3. **Create** a *css.txt* file under the ***css*** Client Library Folder.



4. Enter the following lines in the *css.txt* file:

```
#base=source
emulator.css
```

Notice that the first line in css.txt file is telling us to find the emulator.css file and any other css files listed here in a folder named *source*. For convenience sake, we will copy the *source* folder and associated *emulator.css* file from the blackberry emulator. Any listed css files will be included in the order listed.

5. **Copy** the *source* folder from the blackberry emulator (*libs/wcm/mobile/components/emulators/blackberry/css)* and paste it into the *myemulator* ***css*** Client Library Folder.

6. Edit the *emulator.css* file to have the following lines:

```
#cq-emulator.myemulator {
    background: url("../../resources/bg.png") no-repeat;
    width: 455px;
    height: 295px;
    padding: 156px 38px 436px 42px;
}

#cq-emulator-content.myemulator {
```

```
    width: 455px;
    height: 295px;
    background-color: #fff;
    overflow: auto;
}
```

Take the time to look at the same *emulator.css* file for the android emulator, as it is a more complete emulator, which also supports rotation.

Or, instead of Steps 1-6, you can copy the entire css Client Library Folder from
/libs/wcm/mobile/components/emulators/blackberry/css
And paste it into */apps/training/components/emulators/myemulator.*

Emulator Configuration

If the emulator supports specific functionality, defined by plugins (like touch scrolling), **create** a configuration node below the emulator component:

1. Right-click the myemulator component and select New...Node.
   - **Node name** = cq:emulatorConfig
   - **Node type** = nt:unstructured
2. Add the properties that define the plugin:
   - The property that will define rotational functionality
     - Name = canRotate
     - Type = Boolean
     - Value = true
   - The property that will define touch scrolling functionality
     - Name = touchScrolling
     - Type = Boolean
     - Value = true

You can check the Emulator interface for more information:
http://dev.day.com/docs/en/cq/current/javadoc/com/day/cq/wcm/emulator/Emulator.html
In our case we are going to create an emulator close to the Blackberry so we don't need any of these definitions, but you should take a look at the ***cq:emulatorConfig*** node of the Android emulator.

Create the emulator image library

1. Create a new folder called ***resources*** under the *myemulator* component node. We use this *resources* folder to store the different images. The images stored there hold the main background image of your emulator as well as the icon

image that appears in the sidekick, this is used for fast emulator switching. Those images are referenced in the emulator.css file.

2. Copy the files *bg.png* and *sidekick-icon-myemulator.png* from your USB stick under */exercises/mobile/resources* to the repository under *resources* folder that you just created.

3. Copy the file thumbnail.png from the USB stick under */exercises/mobile/resources* to the repository under into */apps/training/components/emulators/myemulator.*

   This is the thumbnail used in the upper carrousel that allows you to switch emulators.

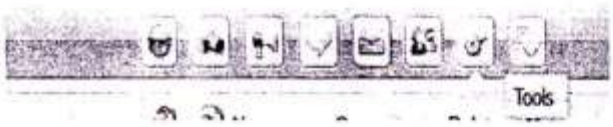
Congratulations! You have successfully created and configured your own emulator, now is time to use it in your own mobile device pages.

Adding your Mobile Emulator to a Device Group

Before we can use our custom emulator to view our mobile pages we need to add it to a mobile device group.

The emulator we created is quite similar to the Blackberry emulator, so we are going to add our custom emulator to the same mobile device group as the one with which the Blackberry emulator is currently associated.

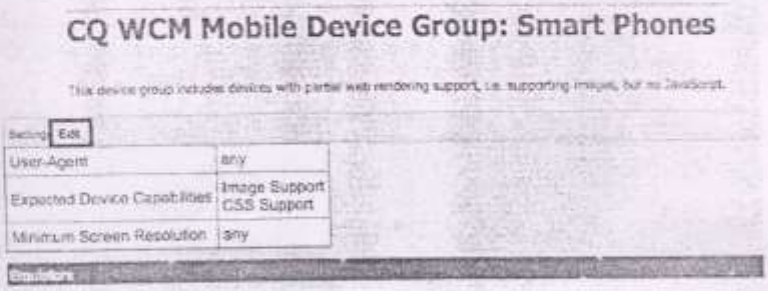
1. In your administration panel open the Tools section



2. Select the mobile section and double click in the "Smart Phones" device group. A new window appears, showing the different characteristics of the Smart Phones device group.

Editing the Smart Phone device group

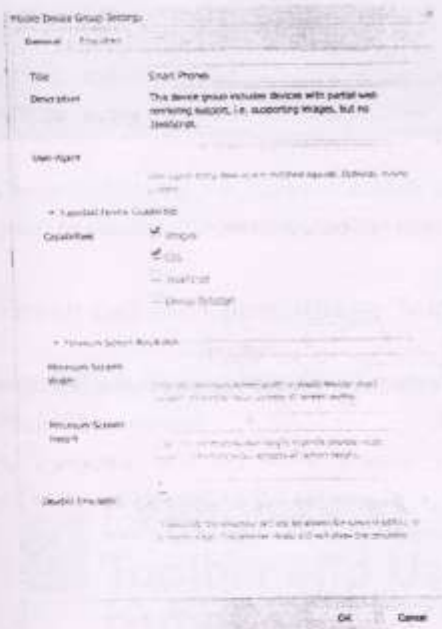
3. Click on the Edit button in order to add the new emulator. A new window appears. The first tab shows the general capabilities of the devices associated with this group.



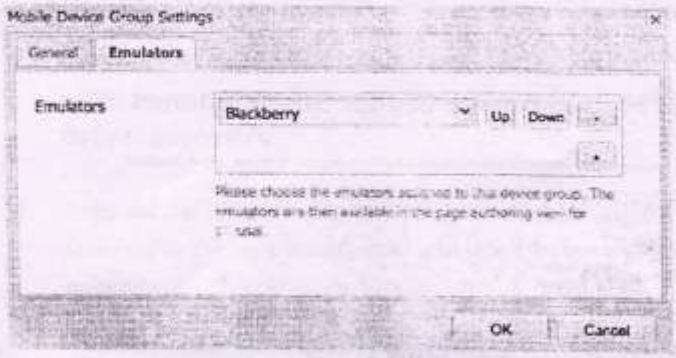
NOTE

In a real case scenario, the User-Agent of your new emulator should match the Regex used for the User-Agent of this device group. You can check your User Agent by opening the following URL:

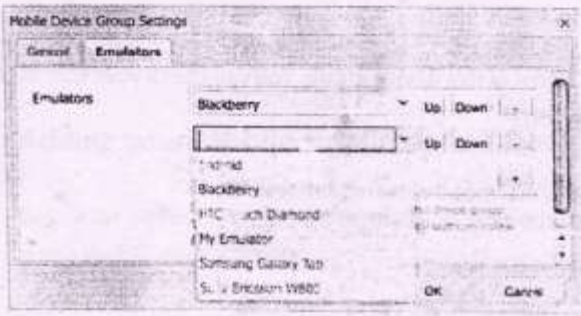
http://localhost:4502/etc/mobile/useragent-test.html



4. Select the Emulators tab. A list of emulators in this device group appears.



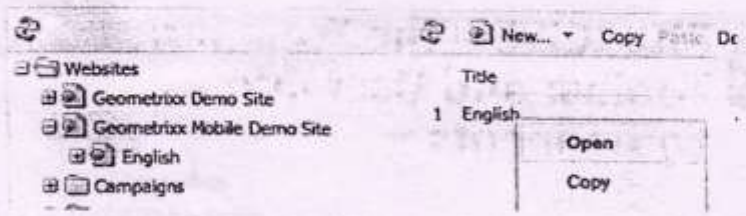
5. Add your custom emulator to the list.



Congratulations! You successfully added your own emulator to a group of mobile device emulators.

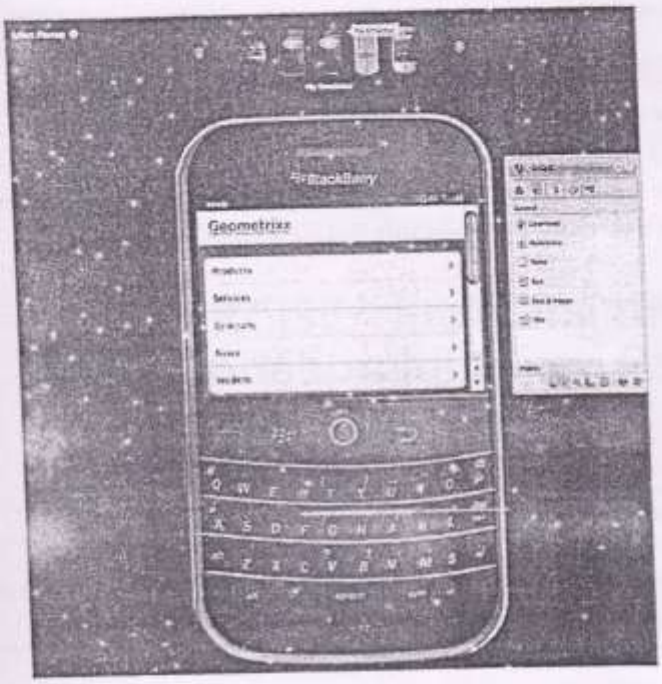
Using the Custom Emulator

1. In the Tree List of Site Admin, select "English" page under the "Geometrixx MobileDemo Site" and Open it.



2. The different mobile devices are grouped into different groups. Those groups or categories can be seen in the top left corner of your page. Click on the device selector until you have selected your custom emulator.

Your new emulator is used now to display the English page.



Congratulations! You have successfully created your own emulator and use it in order to display a Geometrixx mobile page.

Finish out the Contentpage Template

Now it is time to apply finishing touches to the look, feel and layout of the Training "Page" Component.

EXERCISE - The Foundation Toolbar and User info components

1. Open the file *header.jsp* in the Training Contentpage Component and replace the "userinfo section with a <cq:include> of the foundation userinfo Component - then Save.
   - path = "userinfo"
   - resourceType = "foundation/components/userinfo"

2. Test your script by requesting a Page in CQ5 Siteadmin that implements these multiple foundation Components.
   - If successful, you should see a Page view similar to the image below.

Page preview of completed page component

Congratulations! You have successfully included multiple foundation Components. Again, this exercise is used to apply finishing touches to the look and feel of the Training "Page" Component.

Section 10- CQ5 Development - Advanced Concepts

End User Search

Searching for content in CQ5 is very similar to "traditional" searches you have created in the past.

1. An HTML form is needed to collect the user's input (search string).
2. Once the form has been submitted, you need to capture the search string.
3. You need to prepare a query statement based on the search string.
4. A query object needs to be created that will connect to the content repository, and implement the query statement.
5. You need to collect and parse the query results, if any.
6. Output related to the query results should be displayed appropriately.

When querying a Java Content Repository (JCR) using the JCR API, some basic functionality (API calls) need to be implemented:

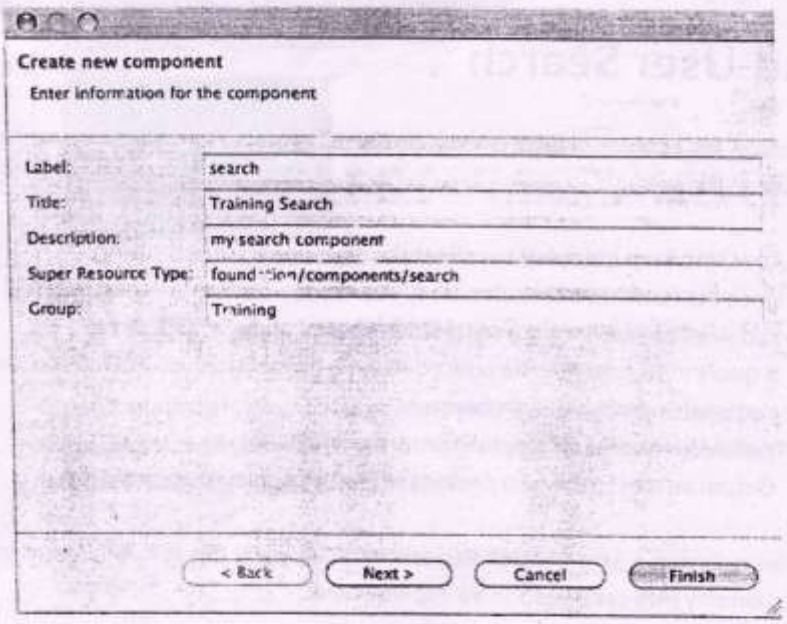
- javax.jcr.Session - JCR session, can be reached for example by Node.getSession()
- javax.jcr.Workspace - JCR workspace, can be reached for example by Session.getWorkspace()
- javax.jcr.query.QueryManager - QueryManager is used to create a Query object and can be reached via Workspace.getQueryManager()
  - createQuery(String statement, String language) - creates the Query object for the provided statement in the provided language (e.g. SQL)
- javax.jcr.Query
  - execute() - executes this Query and returns a QueryResult object
- javax.jcr.query.QueryResult
  - getRows() - returns an iterator over the Rows of the query result table

For more detailed information, please review the Javadocs for the JCR and CRX provided in the documentation.

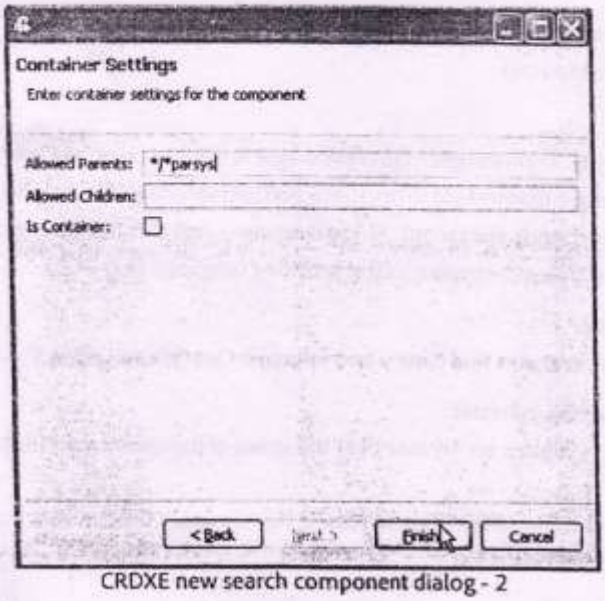
EXERCISE - Create a Search Component

The following instructions explain how to create a Component that will allow visitors to search the content of the Web site/repository. This exercise will demonstrate the differences among the multiple Search APIs. The search Component can be placed in the parsys of any Page, and has the ability to search the content of the Web site based on a query string provided in the request.

1. Create a new search "content" Component.



CRDXE new search component dialog - 1



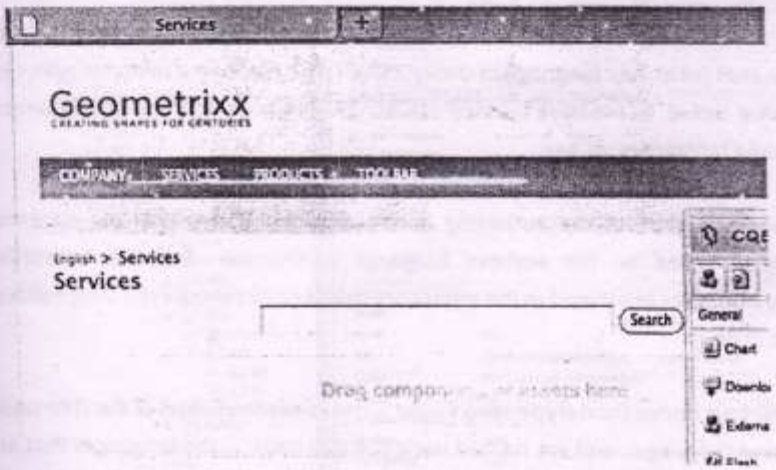
CRDXE new search component dialog - 2

2. Replace *search.jsp* with the *search.jsp* from the USB drive.

3. Update search.jsp to ensure that the search path matches the path to your training site.

4. Add your search Component to the paragraph system Component in **Design mode.**
   - Notice how the Component will be found in the "Training" group - this was defined during the creation of the Component

5. Test your script by adding an instance of this Component (i.e. paragraph) to the paragraph system Component of a Training Page.
   - If successful, you should see the default complex Component output, similar to the image below.



Page preview of search component

6. Search for a word you know exists on a separate Page in your Training Web site structure.
   - You may need to perform this search in Page preview mode, to ensure a "clean" request

7. Now replace the contents of */content/training-site* with the contents of *search.CQ5wcm.jsp*.

8. Examine the code to see the differences between the JCR search API and the WCM search APIs.

9. Try the search again.

10. Now replace the contents of *search.jsp* with the contents of *searchenhanced.CQ5wcm.jsp*.

11. Examine the code to see a good example of using Expression Language (JSTL) with CQ5. You should also note the extensive use of properties set in the dialog and the use of facets. The code also builds a search term tag cloud - called "Search Trends".

12. Try the search again.

Congratulations! You have successfully created a search Component that queries the content in your Training Web site structure. You can further enhance this Component by adding Widgets to the Dialog to output default messages, written by a content author, if the search was successful, or unsuccessful.

Internationalization of the Authoring Interface

The CQ5 Authoring interface ships in 7 languages allowing authors to enter and manage content in their language of choice. When you create your own components and dialog boxes, as we have learned earlier, you can provide those dialog boxes in multiple languages, as well.

Internationalization of the authoring interface allows you to provide dynamic messaging based on the authors' language preference. Internationalization message bundles are stored in the repository under nodes (nodeType sling:Folder) named ***i18n***.

The children nodes (nodeType *sling:Folder + mixin mix:language*) of the *i18n* node represent languages and are named using the ISO code of the languages that are supported (e.g. en, de, fr, etc.). Below these language nodes are the message bundle nodes (nodeType sling:MessageEntry), which will contain a simple key-message pair.

The location of the i18n node within the repository determines the scope of the message bundles. If located in a project directory (e.g. /apps/training), it should contain only messages related to the current project. However, if located in a Component hierarchy, it should contain only Component specific translations. Globally used messages should be place in /apps/i18n.

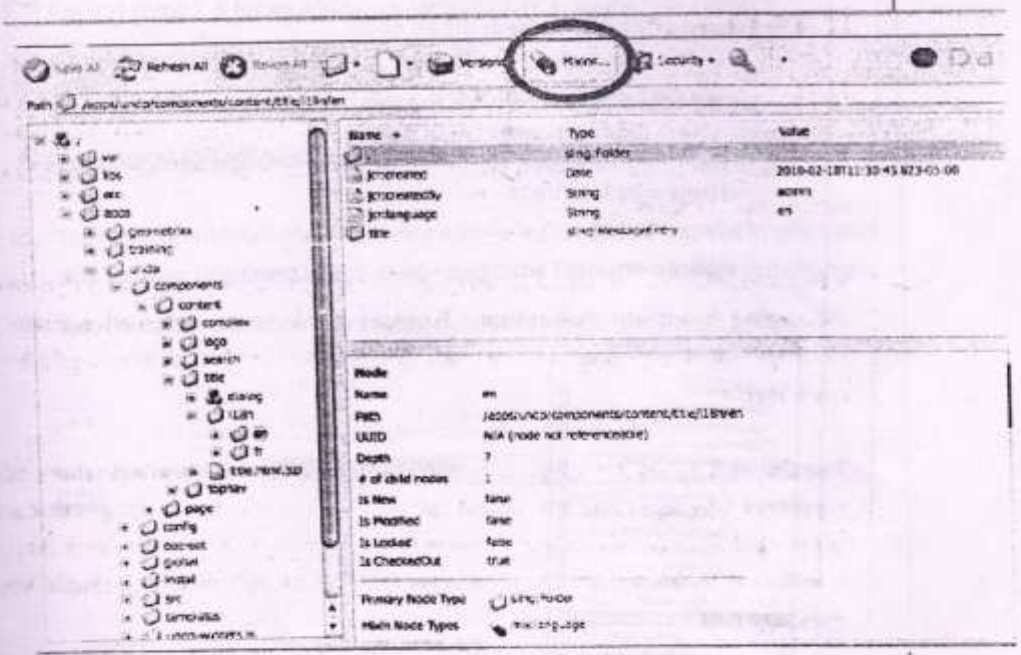
EXERCISE - Apply internationalization to the Title Component Dialog

The following instructions explain how to apply CQ's internationalization (i18n) capabilities in a Dialog. This will allow you to provide dynamic Dialog "messages" based on the authors' language preference.

1. Create an **i18n** node (nodeType sling:Folder) under the root node of the title Component.
   - e.g. /apps/training/components/title

2. Create an **en** node (nodeType sling:Folder) under the newly created *i18n* node.

3. At the current time, CRXDE does not support the addition of *mixins* to nodes. So you must use either CRXDE Lite or the Content Explorer. Open a Content Explorer window and assign the **mixin**, mix:language to the newly created en node. Although they appear to be represented as properties, **mixins** are used to add functionality to a node. They are basically reusable feature sets that can be added to any node type.

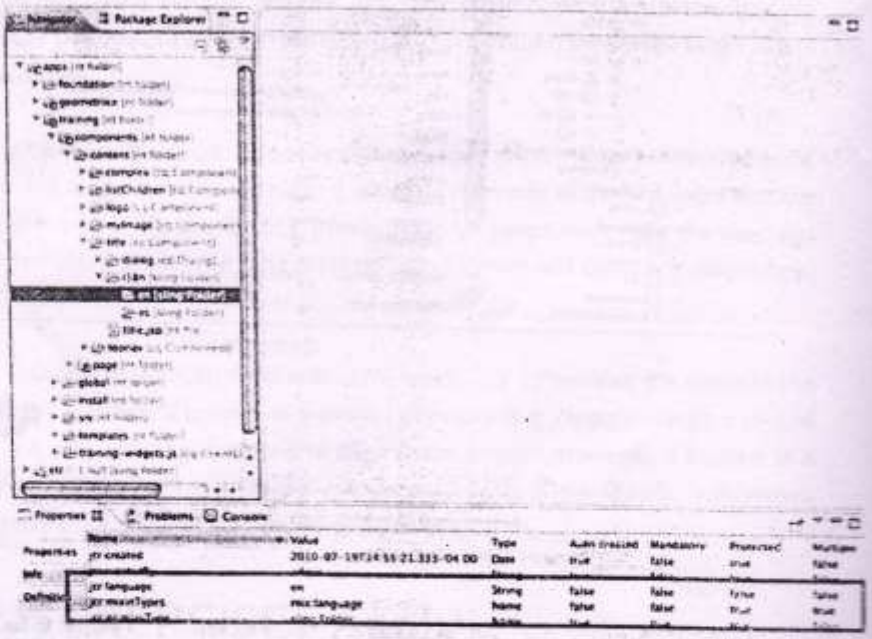


choose mixin

4. Return to CRXDE. Refresh the *en* node.

5. Set the following property on the newly created *en* node:

   - The property that identifies the language using the ISO language code
     - Name = jcr:language
     - Type = String
     - Value = en



6. Create a **fieldLabel** node (nodeType sling:MessageEntry) under the newly created *en* node.

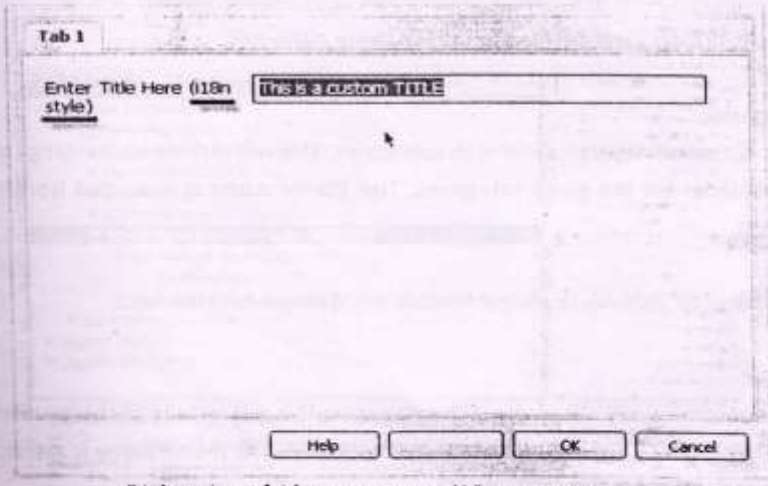
7. Assign the following properties to the newly created *fieldLabel* node:

- The property that identifies the message key
  - Name = sling:key
  - Type = String
  - Value = i18n-title
- The property that identifies the message
  - Name = sling:message
  - Type = String
  - Value = Enter Title Here (i18n style)

8. Change the **fieldLabel** property (Value = i18n-title) on the title Component's Dialog *title* node.
   - *i.e. <path-to-component>/dialog/items/items/tab1/items/title*

9. Repeat steps 2-8 for an additional language. (e.g. Spanish or French)
   - ISO code for Spanish = es
   - sling:key value = i18n-title
   - sling:message value = Titulo

10. Test your script/Dialog by requesting a Page in CQ5 Siteadmin that implements this Training title Component then invoke the Dialog to observe the changes.
    - If successful, you should see Dialog output similar to the image below.



Dialog view of title component - i18n message

Congratulations! You have successfully applied an i18n message to the title Component's Dialog. Since the content author's default/current language preference is English, the en message you defined earlier should appear. Change your default language to Spanish and see the Spanish label.

Adding New xtypes

You may never need to create a custom Widget. The CQ5 and ExtJS Widgets provided to you OOTB cover a large array of potential user input use-cases and are highly configurable. It is Adobe's strong recommendation you thoroughly study the available Widgets and potential configurations via the CQ5-WIDGETS-API at the following URL:

http://dev.day.com/docs/en/cq/current/widgets-api/index.html

Several steps are required to create and register a new xtype:

- Create a JS library to hold your new xtypes/Widgets
- Extend an existing xtype/Widget or create a new one
- Register the new xtype/Widget with CQ5
- Use the new xtype/Widget in a Dialog

<cq:includeClientLib /> tag

The *<cq:includeClientLib>* tag includes a CQ5 html client library, which can be a *js*, a *css* or a *theme* library. For multiple inclusions of different types, for example *js* and *css*, this tag needs to be used multiple times in the jsp. This tag is a convenience wrapper around the *com.day.cq.widget.HtmlLibraryManager* service interface.

The *<cq:includeClientLib>* tag has the following attributes:

categories
A list of comma-separated client lib categories. This will include all Javascript and CSS libraries for the given categories. The theme name is extracted from the request.

Equivalent to: *com.day.cq.widget.HtmlLibraryManager#writeIncludes*

theme
A list of comma-separated client lib categories. This will include all theme related libraries (both CSS and JS) for the given categories. The theme name is extracted from the request.

Equivalent to: ***com.day.cq.widget.HtmlLibraryManager#writeThemeInclude***

js
A list of comma-separated client lib categories. This will include all Javascript libraries for the given categories.

Equivalent to: *com.day.cq.widget.HtmlLibraryManager#writeJsInclude*

css
A list of comma-separated client lib categories. This will include all CSS libraries for the given categories.

Equivalent to: *com.day.cq.widget.HtmlLibraryManager#writeCssInclude*

themed
A flag that indicates of only themed or non themed libraries should be included. If omitted, both sets are included. Only applies to pure JS or CSS includes (not for categories or theme includes).

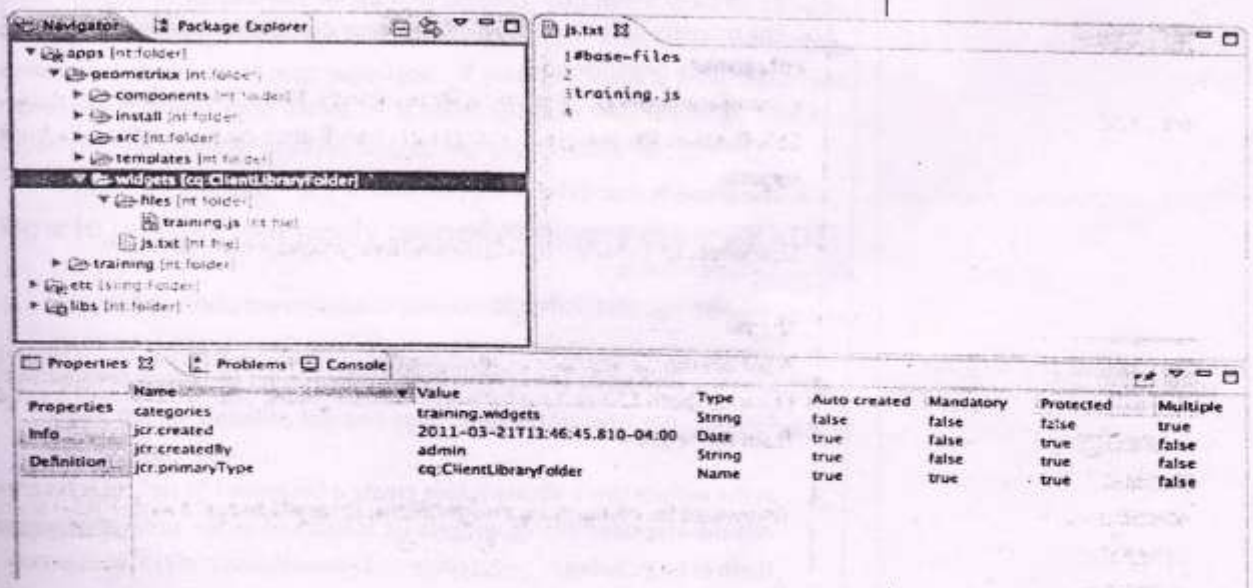
The *<cq:includeClientLib>* tag is documented at the following location: http://dev.day.com/docs/en/cq/current/howto/taglib.html.

EXERCISE - Create, register and use a new xtype

The following instructions explain how to create and register a CQ5 Widget. This will allow you to apply custom Widgets to your Dialogs and Design Dialogs.

How to create a JS library for new xtypes/Widgets:

1. Create a **widgets** node (nodeType *cq:ClientLibraryFolder*) under the root node of the Training project (*e.g., /apps/training/widgets*).



NOTE
The location of the client library can technically be anywhere in the repository as long as the node type is of cq:ClientLibraryFolder. For widget libraries

/apps/<myproject>/clientlib would be a good convention, however, any .cs and .js libraries that must be available to visitors in a publish instance should be placed into /etc/clientlib or into /etc/designs, depending on the purpose of the library.

As a best practice, you should place the client library folders that are associated with the design under /etc/ designs/<mydesign> and no longer under /apps, if these are required for the public website. They can/should probably be placed under /apps/<myproject> if this clientlib is meant to be as an extension to the authoring UI and therefore not accessible by web site visitors.

Most of the core CQ5 library structures that need to be accessed on publish instances by web users, such as css, images, etc., have been moved out out of /libs and /apps, as the goal is to make those folders non-readable for anonymous visitors (to generally protect any code in there or authoring UIs residing under / libs). Thus those core CQ5 clientlibs (those that are used on publish instances) have been moved to /etc, such as /etc/clientlibs, or into the respective designs, as the design contains css and images already and /etc/designs and such should be readable on publish instances.

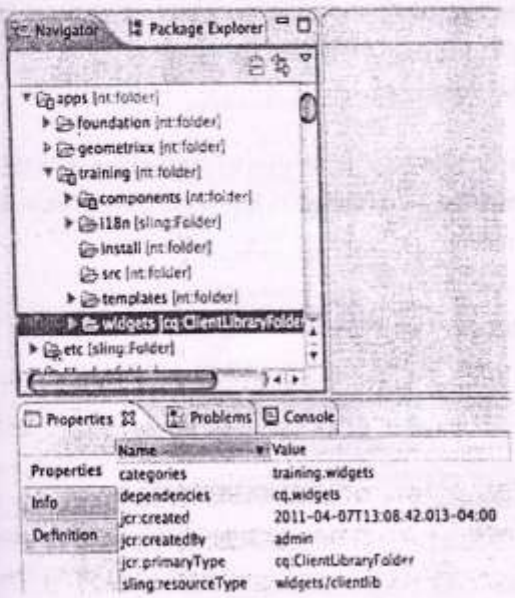
2. Assign the following properties to the newly created *widgets* node:
   - The property that identifies the category these custom Widgets will be referenced by
     - Name = categories
     - Type = String[]
     - Value = training.widgets
   - The property that identifies a dependency on core cq libraries
     - Name = dependencies
     - Type = String []
     - Value = cq.widgets
   - The property that defines the component this resource is based on
     - Name = sling:resourceType
     - Type = String
     - Value = widgets/clientlib

3. Create a **files** node (nodeType sling:Folder) under the newly created *widgets* node.

4. In the *widgets* client library folder, create a file named "*js.txt*". In *js.txt* create a comma-separated list of widgets (js filenames) in the order that you want them to be included.

For example:

```
#base=files
training.js
```



Congratulations! You have successfully created a JS library for custom xtypes/Widgets. When you wish to add custom Widgets, all you need to do is drop it in the newly created files folder.

Before you can use any custom xtypes/Widgets, you must first register the library in which they exist, which occurs during the WCM initialization process. To register your JS library you will need to modify */libs/foundation/components/page/headlibs.jsp* to add the reference to your Client Library Folder. Since you will want these widgets to be available for all your projects, you might want to consider creating a local page superType and move *headlibs.jsp* from your contentpage component to the local page superType. If you do create a local superType, remember that you must decide what other reusable files might go into that superType - for example: *head.jsp, body.jsp* and any reusable pieces.

How to register your newly created JS library using CRXDE:

1. Copy the file */libs/foundation/components/page/headlibs.jsp. Paste headlibs.jsp into the /apps/<application name>/components/page/contentpage.*

2. Open the file **headlibs.jsp**, and enter some HTML and JSP code, similar to below - then Save.

headlibs.jsp

```
<%@include file="/libs/foundation/global.jsp" %><%
%>
  <cq:includeClientLib categories="cq.foundation-main"/>
  <cq:includeClientLib js="training.widgets" />
<%
    currentDesign.writeCssIncludes(pageContext);
%>
```

Congratulations! You have successfully registered a JS library for custom xtypes/ Widgets. Again, when you wish to add custom Widgets, all you need to do is drop it in the files folder of a registered JS library.

Finally, we will create a new xtype/Widget by extending an existing xtype/Widget. ExtJS is a large, powerful, and somewhat complicated JS framework. If possible, it may benefit you to attend ExtJS training

How to create and test a custom xtype/Widget:

1. Create a new file in the folder */apps/training/widgets/files.*

CRXDE files folder

2. Open the file **training.js**, and enter some JS code, similar to below - then Save.

```
// Create the namespace
Training = {};
// Create a new class based on existing CompositeField
Training.Selection = CQ.Ext.extend(CQ.form.CompositeField, {
    text: "default text",
    constructor : function(config){
        if (config.text != null) this.text = config.text;
        var defaults = {
                height: "auto",
                border: false,
                style: "padding:0;margin-bottom:0;",
                layoutConfig: {
                    labelSeparator: CQ.themes.Dialog.LABEL_SEPARATOR
                },
                defaults: {
                    msgTarget: CQ.themes.Dialog.MSG_TARGET
                }
```

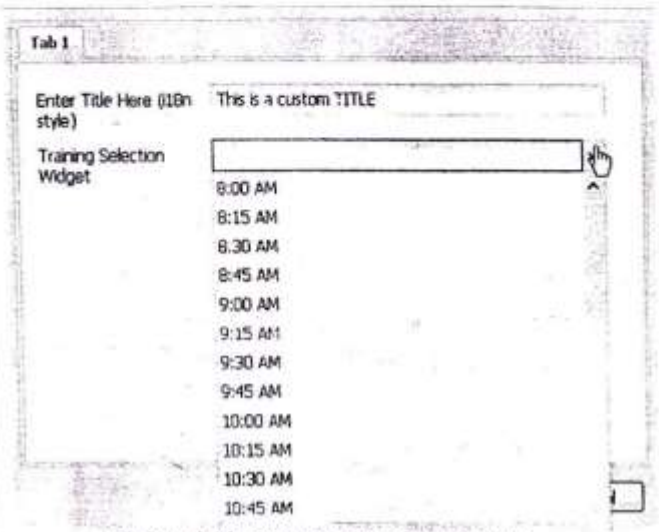
```
        };
        CQ.Util.applyDefaults(config, defaults);
        Training.Selection.superclass.constructor.call(this, config);
        this.selectionForm = new CQ.Ext.form.TimeField({
            name: this.name,
            hideLabel: true,
            anchor: "100%",
            intDate: new Date(),
            validateValue: function(value) {return true}
        });
        this.add(this.selectionForm);
    },
    processRecord: function(record, path){
        this.selectionForm.setValue(record.get(this.getName()));
    }
});
CQ.Ext.reg("trainingSelection", Training.Selection);
```

3. Now it is time to use our new xtype. Create a **training** node (nodeType cq:Widget) under the Training title Component's Dialog *tab1* widget collection.
   - e.g. /apps/training/components/title/dialog/items/..ems/tab1/items

4. Assign the following properties to the newly created *training* node:
   - The property that will define where the content is stored
     - Name = name
     - Type = String
     - Value = ./training
   - The property that will define the Widget type
     - Name = xtype
     - Type = String
     - Value = trainingSelection
   - The property that will define the label applied to the Widget
     - Name = fieldLabel
     - Type = String
     - Value = Training Selection Widget

5. Test your script/Dialog by requesting a Page in CQ5 Siteadmin that implements this Training xtype/Widget (i.e. the title Component) - then double-click the "title" content to invoke the Dialog.

   - If successful, you should see the custom Dialog xtype, similar to the image below.



Dialog of title component with new xtype

Congratulations! You have successfully created and applied a custom xtype/ Widget. This is a significant step in your development capabilities, as you are now able to create custom CQ5 xtypes/Widgets. To fully complete this exercise, you would output/use the user's selection in manner deemed appropriate.

Using jQuery with Ajax, and Apache Sling

Sling lets you drive your content repository from within a webpage by making an Ajax request with jQuery. jQuery is a cross-browser JavaScript library designed to simplify the client-side script of HTML. jQuery works like JavaScript where its used to help with interaction and effects with your development code. jQuery makes it easy to handle DOM objects, events, effects and Ajax, automatically takes care of JavaScript leaks, and has countless third-party plugins.

jQuery is not a language but it is a well written JavaScript code. The most common jQuery effects are drop down menus, drag and drop elements, animations and form validation. Developers have also connected this with other coding languages like JSP, ESP, ASP, etc.

The jQuery library has a full suite of AJAX capabilities. The functions and methods provided allow you to load data from the server without a browser page refresh.

EXERCISE - Dynamic User Account Grid

Goal: Create a custom component that will display a dynamic grid of user accounts. This component will make use of the jQuery plugin that uses AJAX to retrieve data from CQ5. The jQuery Flexigrid plugin is the target for this exercise. The plugin will execute a callback to a JSP in CQ to retrieve JSON formatted data

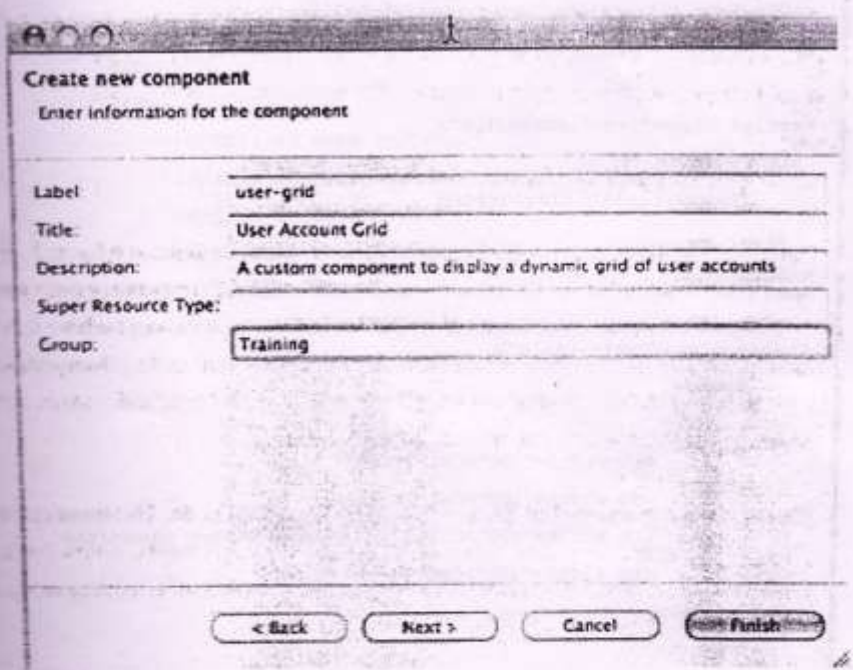
Note:

The default Sling servlet in CQ5 can be used to retrieve content in JSON format for pretty much anything, however this use case does not use the default Sling servlet. We will provide our own request handler so we can understand the complete workflow involved.

Create the component

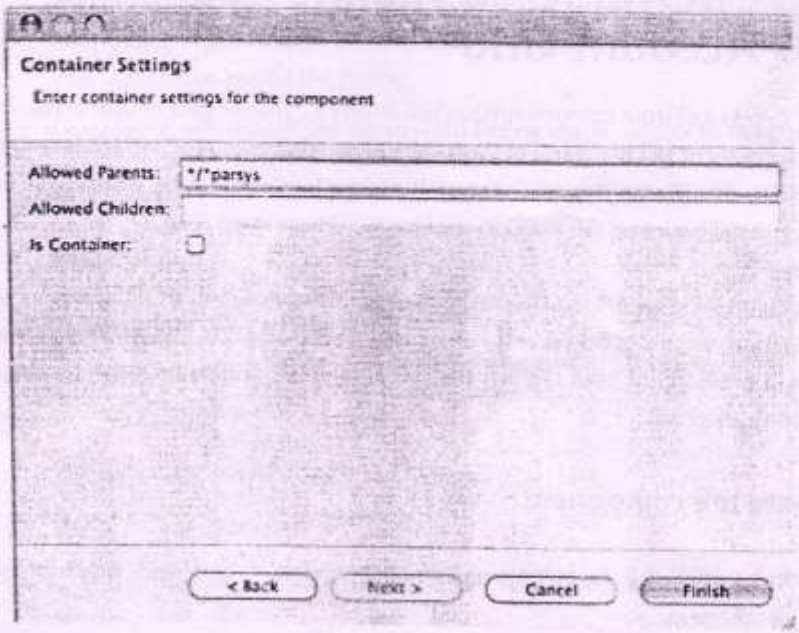
1. Right click the /apps/training/components folder and add the user-grid component.

   Name = user-grid
   Title = User Account Grid
   Description = A custom component to display a dynamic grid of user accounts
   Group = Training



2. Click **Next** and enter the following value:

   AllowedParents = */*parsys



3. Click **Finish**.

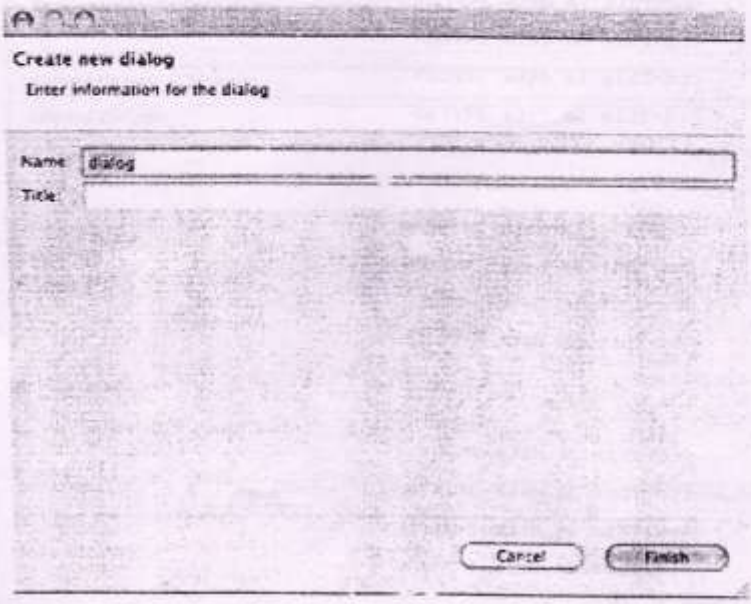
4. Add the following code to your user-grid component:

```
<%@include file="/libs/foundation/global.jsp"%><%
%>

<script type="text/javascript">
 $CQ(function ($) {
    $('.user-table').flexigrid();
  });
</script>

<table class="user-table">
    <thead>
        <tr>
            <th width="100">Col 1</th>
            <th width="100">Col 2</th>
            <th width="100">Col 3 is a long header name</th>
            <th width="300">Col 4</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>This is data 1 with overflowing content</td>
            <td>This is data 2</td>
            <td>This is data 3</td>
```

```
        <td>This is data 4</td>
    </tr>
    <tr>
        <td>This is data 1</td>
        <td>This is data 2</td>
        <td>This is data 3</td>
        <td>This is data 4</td>
    </tr>
    <tr>
        <td>This is data 1</td>
        <td>This is data 2</td>
        <td>This is data 3</td>
        <td>This is data 4</td>
    </tr>
    <tr>
        <td>This is data 1</td>
        <td>This is data 2</td>
        <td>This is data 3</td>
        <td>This is data 4</td>
    </tr>
    <tr>
        <td>This is data 1</td>
        <td>This is data 2</td>
        <td>This is data 3</td>
        <td>This is data 4</td>
    </tr>
    <tr>
        <td>This is data 1</td>
        <td>This is data 2</td>
        <td>This is data 3</td>
        <td>This is data 4</td>
    </tr>
  </tbody>
</table>
```

NOTE: This is just placeholder code which we will make dynamic in a following section of this exercise.

5. Create an empty dialog box (for now) so that the component will be on the AllowedComponents list in the Paragraph System design dialog. This will allow us to indicate that we want this component in the Sidekick.

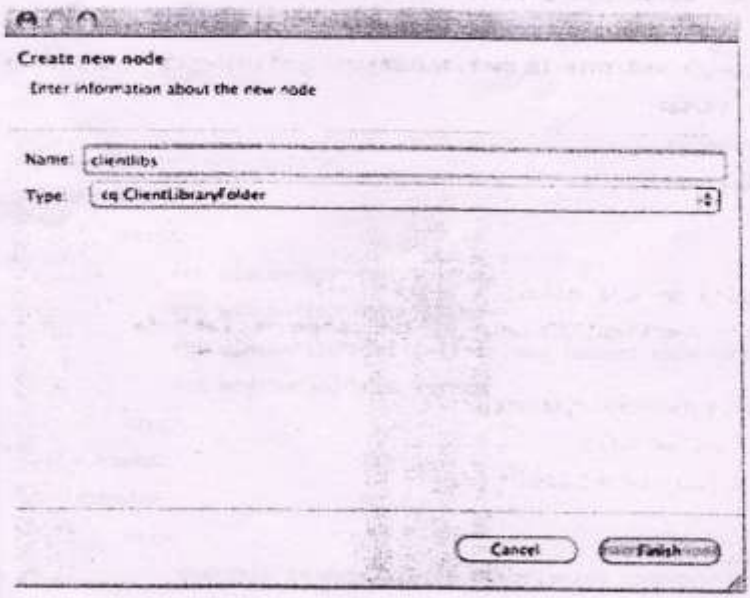


Create the Client Library

1. Right click on the user-grid component and Select New.. Node.

   Name = clientlibs

   Type = cq:ClientLibraryFolder.



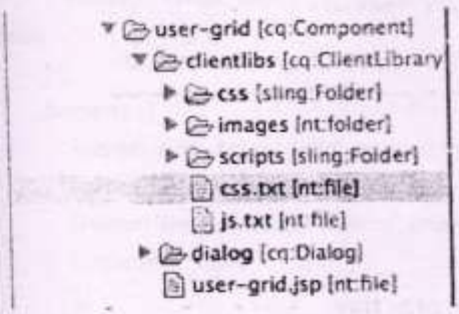
2. Add the following properties to the new client library folder:

   Name = dependencies
   Type = String[]
   Value = cq.jquery

   Name = categories
   Type = String[]
   Value = componentlab

3. Open USB/exercises/user-grid-component. Copy the css, scripts, and css folders into your client library folder.

4. Copy the js.txt and css.txt files into your client library folder. The structure of your client library folder should now look like the following figure:



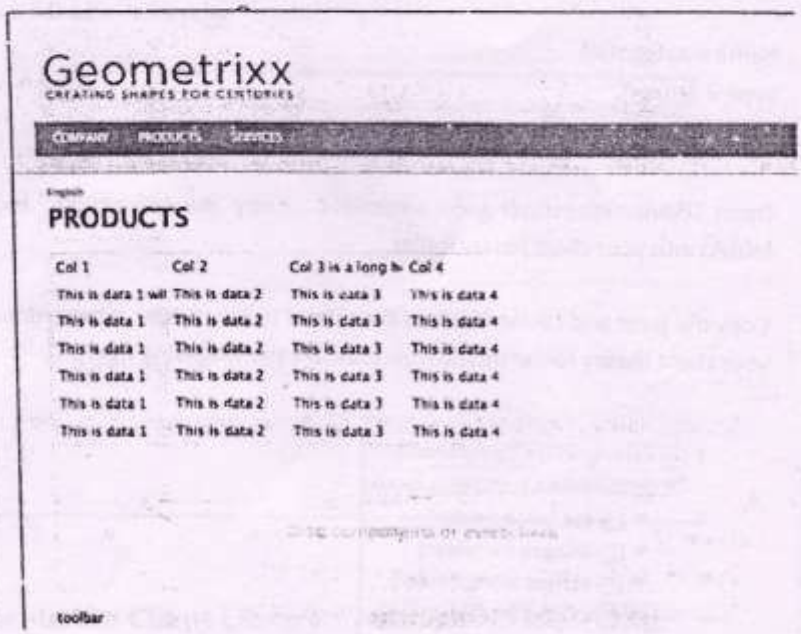
5. We have provided the unpacked flexigrid.js and flexigrid.css files on the USB drive. Examine so you understand what they do.

6. Make the following change to the beginning of user-grid.jsp to pick up the client library:

```
<%@include file="/libs/foundation/global.jsp"%><%
%>

<!-- pick up the client libraries -->
<cq:includeClientLib categories="componentlab" />

<script type="text/javascript">
 $CQ(function ($) {
    $('.user-table').flexigrid();
  });
</script>
...
```

7. Using the procedures introduced previously, add your User Account Grid component to the Paragraph System design.

8. Test your component. It should look like the following:



Make the component Interactive

1. Modify user-grid.jsp by adding the following code:

```
<%
%><%@include file="/libs/foundation/global.jsp"%><%
%><%@page session="false" %>

<!-- Pick up the client libraries  -->
<cq:includeClientLib categories="componentlab" />

<script type="text/javascript">

/* Grab the JCR path to the content entry that calls this
component
 * with Sling, you cannot call a script, you must call the
jcr content
 * node that resolves to the representation (script).
 */
var baseURL = "<%= currentNode.getPath() %>";

$CQ(function () {

   ...
```

2. Modify the $CQ(function()) to match the following:

```
CQ(function () {
  $CQ('.user-table').flexigrid({
    url: baseURL + '.json', //This will trigger a POST request back to CQ
    dataType: 'json', //The expected response will be JSON formatted data
     colModel : [ {
        display : 'User ID', name : 'id', width : 215, sortable : true, align : 'left', hide:
false
      }, {
        display : 'First Name', name : 'givenName', width : 100, sortable : true, align
: 'left', hide: false
      }, {
        display : 'Last Name', name : 'familyName', width : 100, sortable : true, align
: 'left', hide: false
      },{
        display : 'Email', name : 'email', width : 215, sortable : true, align : 'left', hide:
 false
      }],
      buttons : [
        {name: 'Add', bclass: 'add', onpress : userAdd},
        {name: 'Edit', bclass: 'edit', onpress : userEdit},
        {name: 'Delete', bclass: 'delete', onpress : userDelete},
        {separator: true}
        ],
      searchitems : [
        {display: 'User ID', name : 'user_id',isdefault: true},
        {display: 'First Name', name : 'givenName'},
        {display: 'Last Name', name : 'familyName'}
        ],
      sortname: "id",
      sortorder: "asc",
      usepager: true,
      title: "User Account Grid",
      useRp: true,
      rp: 15,
      showTableToggleBtn: false,
      singleSelect: true,
      width: 700,
      height: 200
    });
  });
```

NOTE: In the above code, many advanced jQuery flexigrid plugin options have been provided. The goal of this lab is not to cover flexigrid in detail. You can find additional information about Flexigrid for jQuery at http://flexigrid.info.

3. Now add the following javascript functions just before the closing </script> tag.

```
function userAdd() {
    alert("Add button clicked");
}

function userEdit() {
    alert("Edit button clicked.");
}

function userDelete() {
    alert("Delete button clicked.");
}
```

4. Delete the hardcoded text between the <table class="user-table"> and </table> tags.

5. Your user-grid.jsp file should now look like this:

```
<%--

User Account Grid component.

A custom component that will display a dynamic grid of user accounts

--%><%
%><%@include file="/libs/foundation/global.jsp"%>

<!-- Pick up the client libraries  -->
<cq:includeClientLib categories="componentlab" />

<script type="text/javascript">

/* Grab the JCR path to the content entry that calls this component
 * with Sling, you cannot call a script, you must call the jcr content
 * node that resolves to the representation (script).
 */
var baseURL = "<%= currentNode.getPath() %>";

$CQ(function () {
    $CQ('.user-table').flexigrid({
      url: baseURL + '.json', //This will trigger a POST request back
to CQ
      dataType: 'json', //The expected response will be JSON formatted
data
```

```
        colModel : [ {
            display : 'User ID', name : 'id', width : 215, sortable :
true, align : 'left', hide: false
        }, {
            display : 'First Name', name : 'givenName', width : 100,
sortable : true, align : 'left', hide: false
        }, {
            display : 'Last Name', name : 'familyName', width : 100,
sortable : true, align : 'left', hide: false
        },{
            display : 'Email', name : 'email', width : 215, sortable :
true, align : 'left', hide: false
        }],
        buttons : [
            {name: 'Add', bclass: 'add', onpress : userAdd},
            {name: 'Edit', bclass: 'edit', onpress : userEdit},
            {name: 'Delete', bclass: 'delete', onpress : userDelete},
            {separator: true}
            ],
       searchitems : [
            {display: 'User ID', name : 'user_id',isdefault: true},
            {display: 'First Name', name : 'givenName'},
            {display: 'Last Name', name : 'familyName'}
            ],
         sortname: "id",
         sortorder: "asc",
         usepager: true,
         title: "User Account Grid",
         useRp: true,
         rp: 15,
         showTableToggleBtn: false,
         singleSelect: true,
         width: 700,
         height: 200
    });
});

function userAdd() {
    alert("Add button clicked");
}

function userEdit() {
    alert("Edit button clicked.");
}

function userDelete() {
```

```
    alert("Delete button clicked.");
}


</script>

<table class="user-table"></table>
```

Create the callback jsp file

1. Create the user-grid.json.POST.jsp callback script. Right click on the user-grid component node and select New..JSP.

2. Enter the following code into your new user-grid.json.POST.jsp script:

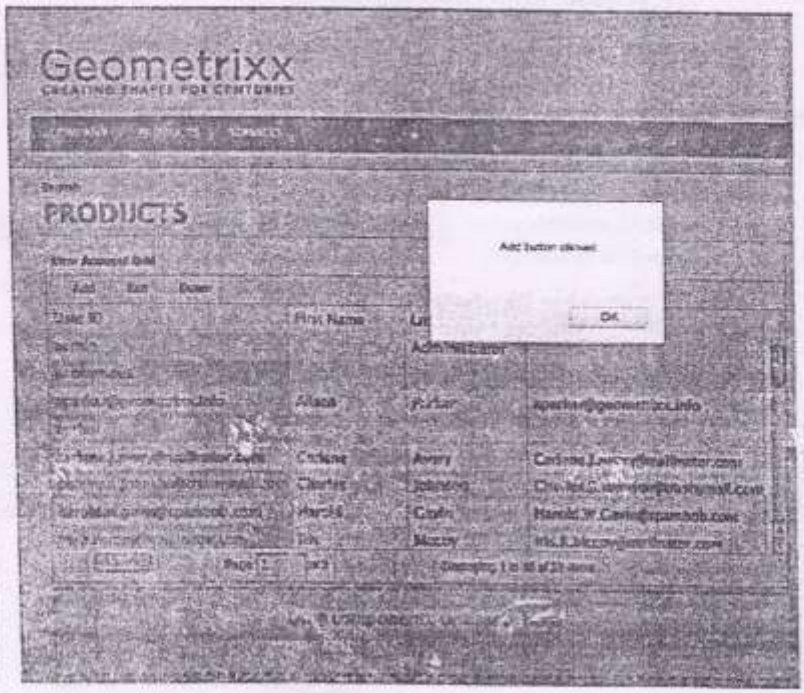
```
<%@page session="false" %>
<%@include file="/libs/foundation/global.jsp"%>
<%@ page import="org.apache.sling.jcr.api.SlingRepository" %>
<%@ page import="com.day.cq.security.UserManager" %>
<%@ page import="com.day.cq.security.UserManagerFactory" %>
<%@ page import="com.day.cq.security.User" %>
<%@ page import="com.day.cq.security.Authorizable" %>
<%@ page import="com.day.cq.security.profile.Profile" %>
<%@ page import="java.util.Iterator" %>
<%@ page import="java.util.List" %>
<%@ page import="java.util.ArrayList" %>
<%@ page import="com.day.cq.commons.TidyJSONWriter" %>

<%

//Local variables
final SlingRepository repos = sling.getService(SlingRepository.class);
final UserManagerFactory umFactory = sling.
getService(UserManagerFactory.class);

Session session = null;
Iterator<User> userIterator = null;
Iterator<Authorizable> authorizableIterator = null;
try
{
    // Ensure that the currently logged on user has admin privileges.
    session = repos.loginAdministrative(null);

    final UserManager um = umFactory.createUserManager(session);
        final TidyJSONWriter writer = new TidyJSONWriter(response.
getWriter());
```

```
userIterator = um.getUsers();
List<User> users = new ArrayList<User>();
User tmpUser;

// copy iterator into a List for additional manipulations.
while(userIterator.hasNext())
{
    tmpUser = userIterator.next();
    users.add(tmpUser);

}

//Begin writing JSON response
writer.setTidy("true".equals(request.getParameter("tidy")));
writer.object();
writer.key("page").value(1);
writer.key("total").value(users.size());
writer.key("rows").array();

for(int i=0; i < users.size(); i++)
{

    User aUser = users.get(i);
    Profile aProfile = aUser.getProfile();

    writer.object();
    writer.key("id").value(aUser.getID());
    writer.key("cell").array();
    writer.value(aUser.getID());
    writer.value(aProfile.getGivenName());
    writer.value(aProfile.getFamilyName());
    writer.value(aProfile.getPrimaryMail());
    writer.endArray();
    writer.endObject();
  }

    writer.endArray();
    writer.endObject();
    session.logout();
}
catch (Exception e)
{
    System.out.println("myajaxsample Exception Occured: " + e.
getMessage());
}
finally
{
    session.logout();
    session = null;
}
%>
```

3. Test your component by refreshing the page that contains the component in the Paragraph System.

4. Notice the the interactive grid now appearing on the page.

5. Click on the action buttons (Add, Edit and Delete) to see the appropriate message box appear.



NOTE: Not all of the grid features are functional. The search and pagination functions have not yet been implemented. This is left as an extra credit exercise.

Creating OSGi Bundles

Whenever you need to add new functionality to your application in the form of new Java classes, you will do this by creating an OSGi bundle with the Java class inside. This will allow you to create and use custom Java classes in your JSP scripts, allowing for more traditional Java development and library reuse.

What exactly is an OSGi Bundle?

As discussed previously, OSGi defines an architecture for developing and deploying modular applications and libraries (it is also known as the Dynamic Module System for Java). OSGi containers allow you to break your application into individual modules (are JAR files with additional meta information and called bundles in OSGi terminology) and manage the cross-dependencies between them with:

- services implemented within the container
- a contract between the container and your application

These services and contracts provide an architecture which enables individual elements to dynamically discover each other for collaboration.

An OSGi framework then offers you dynamic loading/unloading, configuration and control of these bundles - without requiring restarts.

This architecture allows you to extend Sling with application specific modules. Sling, and therefore CQ5, uses the Apache Felix implementation of OSGi and is based on the OSGi Service Platform Release 4 Version 4.2 Specifications. They are both collections of OSGi bundles running within an OSGi framework.

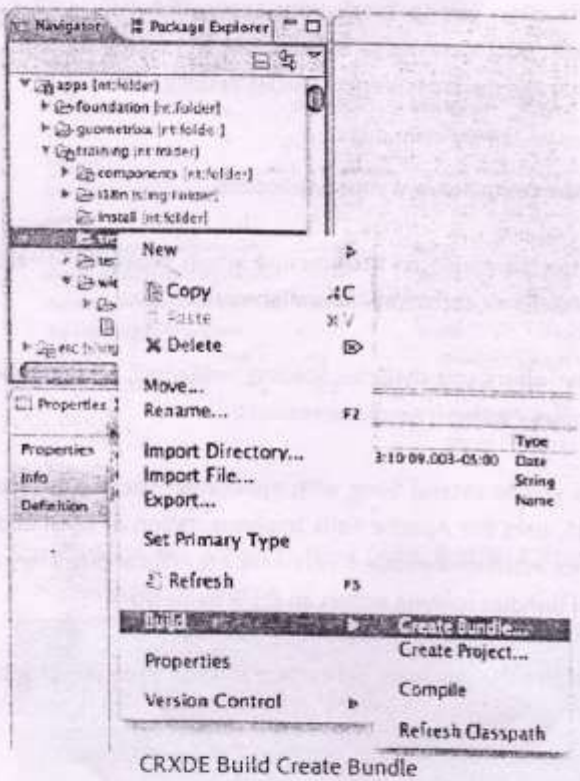
This enables you to perform the following actions on any of the packages within your installation:

- install
- start
- stop
- update
- uninstall
- see the current status
- access more detailed information (e.g. symbolic name, version, location, etc) about the specific bundles

EXERCISE - Create and consume an OSGi bundle

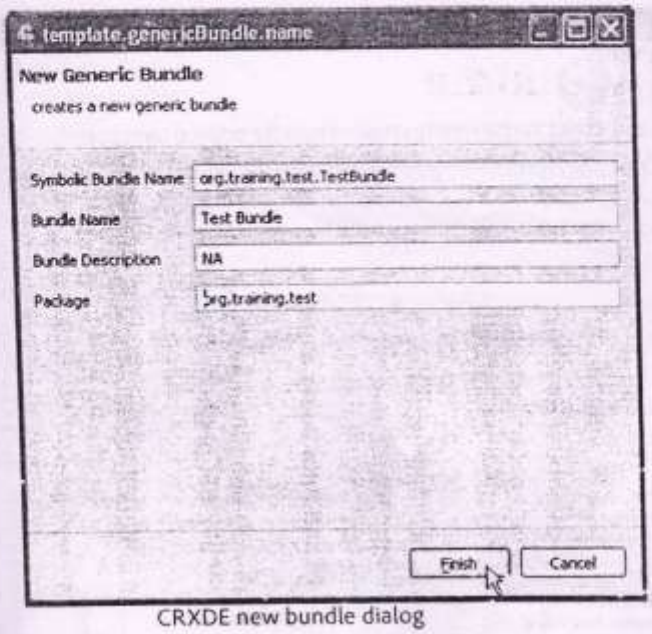
In this first exercise dealing with OSGi bundles, we will learn the basics of creating an OSGi bundle and consuming the Java classes defined inside the bundle. In a later exercise, we will create a more complex Java class and bundle.

1. Select and right-click the Training projects OSGi bundle source directory (i.e. / apps/training/src) - then select **Build .. Create Bundle**.



CRXDE Build Create Bundle

2. Enter the "Symbolic Bundle Name", "Bundle Name", "Bundle Description", and "Package" - then click Finish.



CRXDE new bundle dialog

3. Open the newly created bundle's file org.training.test.TestBundle.bnd - verify that the text -SNAPSHOT has been added to the Bundle-Version.

- This ensures the bundle is automatically installed and started in the OSGI container every time the bundle is built

```
Export-Package: *
Import-Package: *
Private-Package: *
# Include-Resource:
Bundle-Name: Test Bundle
Bundle-Description: NA
Bundle-SymbolicName: org.training.test.TestBundle
Bundle-Version: 1.0.0-SNAPSHOT
Bundle-Activator: org.training.test.Activator
```

NOTE

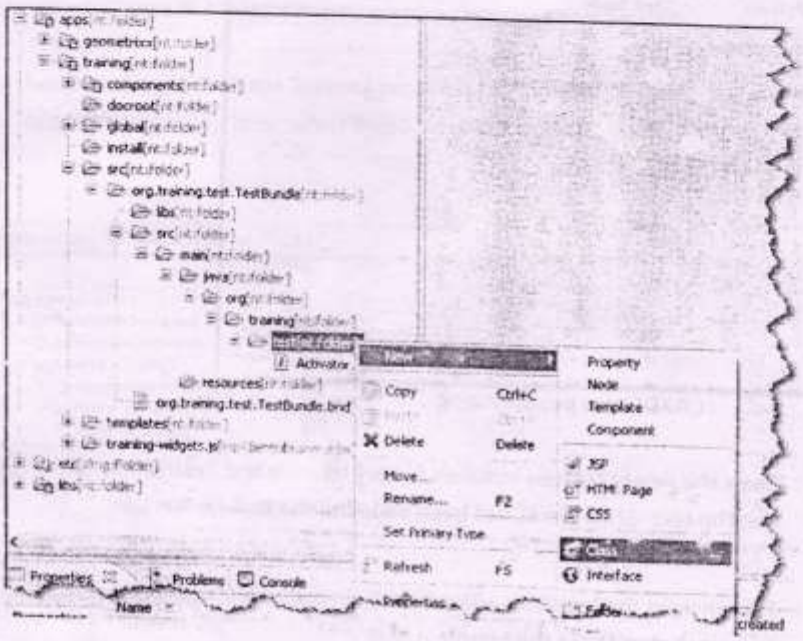
The Bundle wizard creates the following elements:

- The node org.training.test.TestBundle of type nt:folder - it is the bundle container node
- The file org.training.test.TestBundle.bnd - it acts as deployment descriptor for your bundle and consists of a set of headers
- The folder structures:
  - src/main/java/org/training/test - it will contain the packages and the Java classes
  - src/main/resources - it will contain the resources used within the bundle

- The Activator.java file - it is the optional listener class to be notified of bundle start and stop events

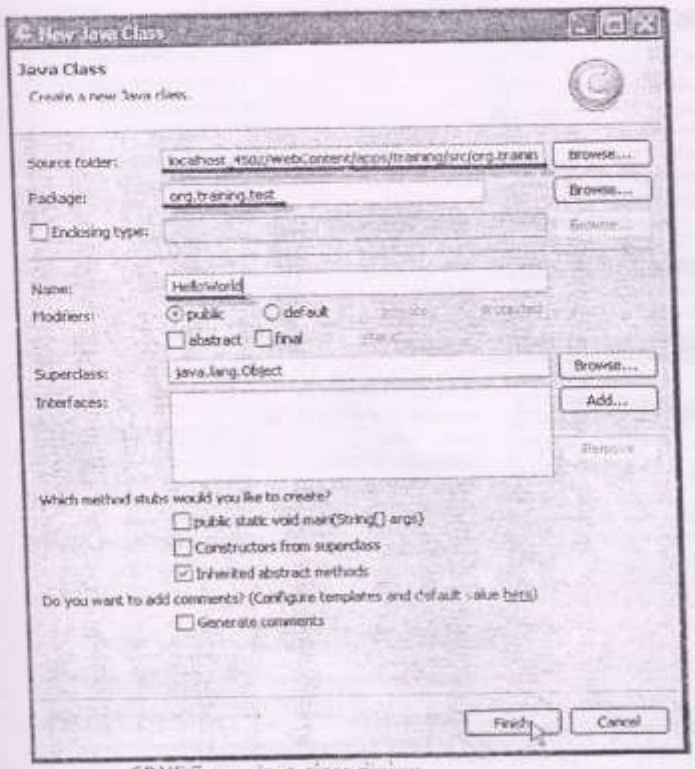
NOTE: The java methods, respectively the classes that implement the executable Java method are registered as OSGI services, enabling you to add methods at anytime during runtime.

4. Create a **new Java class** in the newly created bundle source directory (i.e. src/main/java/org/training/test).



CRXDE create a new java class

5. Enter the "Source folder", "Package", and "Name".
   - Source folder = localhost_4502/WebContent/apps/training/src/org.training.test.TestBundle/src/main/java
   - Package = org.training.test
   - Name = HelloWorld

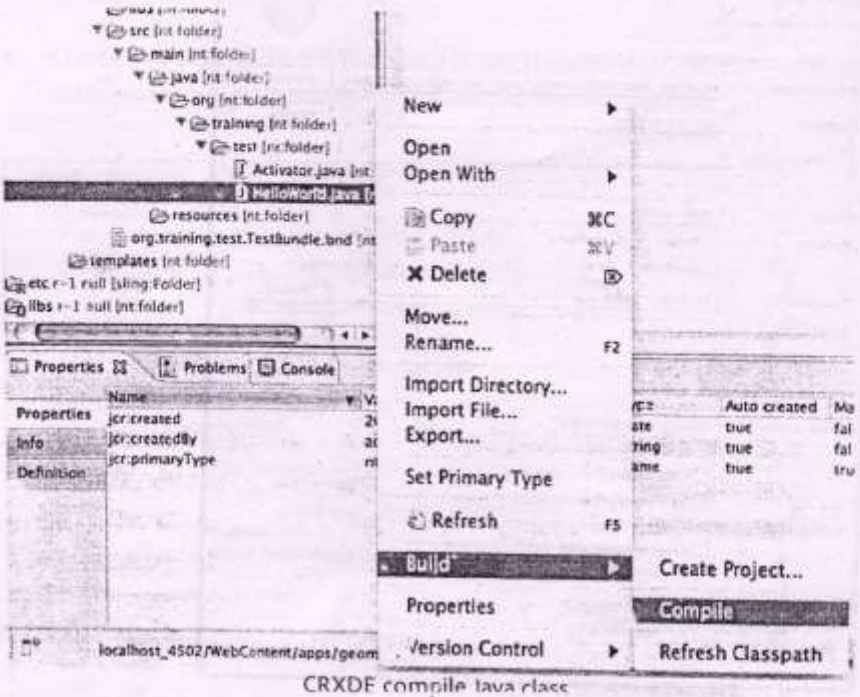


CRXDE new java class dialog

6. Open the newly created Java class file ***HelloWorld.java***, and enter some Java code, similar to below - then **Save**.

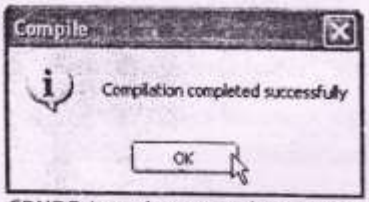
```
package org.training.test;
public class HelloWorld {
public String getMessage() {
      return "Hello World !!!";
   }
}
```

7. Select and right-click the *HelloWorld.java* file - then select **Build .. Compile**.



CRXDE compile Java class

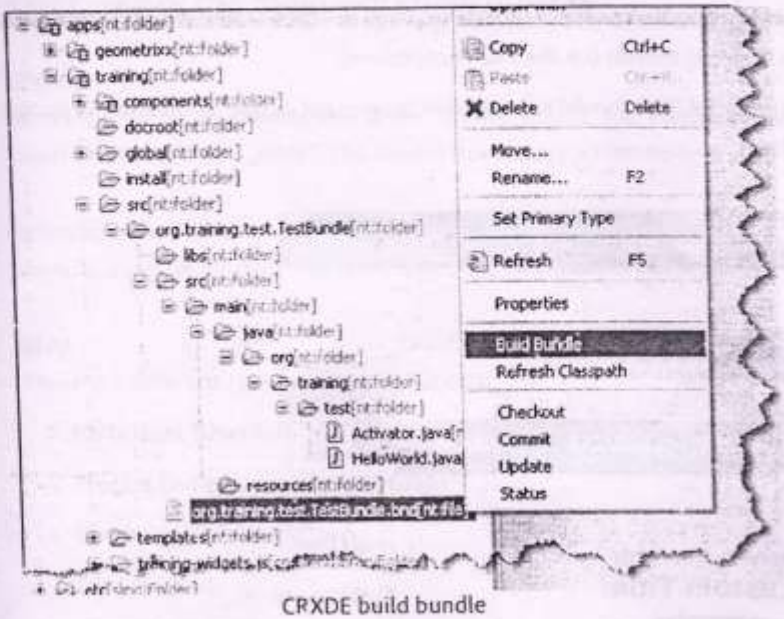
- You should get a Compilation completed successfully message dialog.



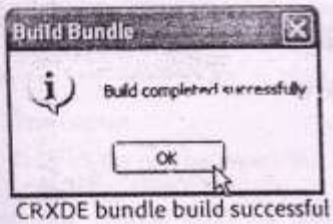
CRXDE Java class compilation successful

8. Select and right-click the *org.training.test.TestBundle.bnd* file - then select **Build Bundle**.

- You should get a Build completed successfully message dialog.



CRXDE build bundle



CRXDE bundle build successful

9. Open the Training title Component's ***title.jsp***, and enter some HTML and JSP code, similar to below - then Save.

- The focus is to import the newly created Java class and use it

title.jsp

```
<%@include file="/libs/foundation/global.jsp"%>
<%@ page import="org.training.test.HelloWorld" %>
<h1><%= properties.get("title", currentPage.getTitle()) %></
h1>
<%
HelloWorld hw = new HelloWorld();
%>
<h3><%= hw.getMessage() %></h3>
```

10. Test your bundle/script by requesting a Page in CQ5 Siteadmin that implements this Training bundle (i.e. the title Component).
    - If successful, you should see the title Component output, similar to the image below.

Page view of title component (bundle)

Congratulations! You have successfully created an OSGi bundle, and have consumed the associated Java class. Once again, this is a significant step in your development capabilities, as you are now able put more "heavy logic" type of code into Java classes, where it belongs, in addition to being able to create Java helper classes that can be re-used by your development team.

Basics of the Workflow Console

CQ5 encompasses several applications that are designed to interact and complement each other. In particular, the Workflow Engine can be used in tight conjunction with several of the other applications.

For example, within CQ5, Web Content Management (CQ5 WCM) is key. This enables you to generate and publish pages to your website. This functionality is often subject to organizational processes, including steps such as approval and sign-off by various participants. These processes can be represented as workflows, which in turn can be defined within CQ5, then applied to the appropriate content pages.

Overview of the main workflow objects

Model

Is made of WorkflowNodes and WorkflowTransitions. The transitions connect the nodes and define the „flow". The Model has always a start node and an end node.

Workflow models are versioned. Running Workflow Instances keep the initial workflow model version that is set when the workflow is started.

Step

There are different types of workflow steps:

- Participant (User/Group)
- Process (Script, Java method call)
- Container (Sub Workflow)
- OR Split/Join
- AND Split/Join

All the steps share the following common properties: **AutoAdvance** and **Timeout** alerts (scriptable).

Transition

Defines the link between two consecutive steps. It is possible to apply rules to the Transition.

WorkItem

The workItem is the "there is a task identifier" and is put into the respective inbox. A workflow instance can have one or many WorkItems at the same time (depending on the workflow model).

The WorkItem references the workflow instance. In the repository the WorkItem is stored below the workflow instance.

Payload

References the resource that has to be advanced through a workflow.

The payload implementation references a resource in the repository (by either a path or an UUID) or a resource by a URL or by a serialized java object. Referencing a resource in the repository is very flexible and in conjunction with sling very productive: for example the referenced node could be rendered as a form.

Lifecycle

Is created when starting a new workflow (by choosing the respective workflow model and defining the payload) and ends when the end node is processed.

The following actions are possible on a workflow instance:

- Terminate
- Suspend
- Resume
- Restart

Completed and terminated instances are archived.

Inbox

Each logged in user has its own workflow inbox in which the assigned WorkItems are accessible. The WorkItems are assigned either to the user itself or to the group to which he belongs.

Workflow Console

The Workflow console is the centralized location for workflow management in CQ5. It can be accessed via the Workflows button on the Welcome page or through the Workflows button in the toolbar on any CQ5 console (for example: Websites, Tools, Tagging).

Within the console there are 4 tabs:

Models

Lists the workflow models currently available. Here you can create, edit or delete workflow mode

Instances

Shows you details of workflow instances which are currently active. These instances are also version dependent.

Archive

Enables you to access details of workflow instances which have terminated, for whatever reason.

Launcher

Allows you to define a workflow to be launched if a specific node has been updated.

NOTE: The payload is assigned to the current version of the workflow; if the main copy of the workflow is updated later, the changes will have no impact on the currently running instance.

Starting a Workflow

There are four methods of manually starting a workflow:

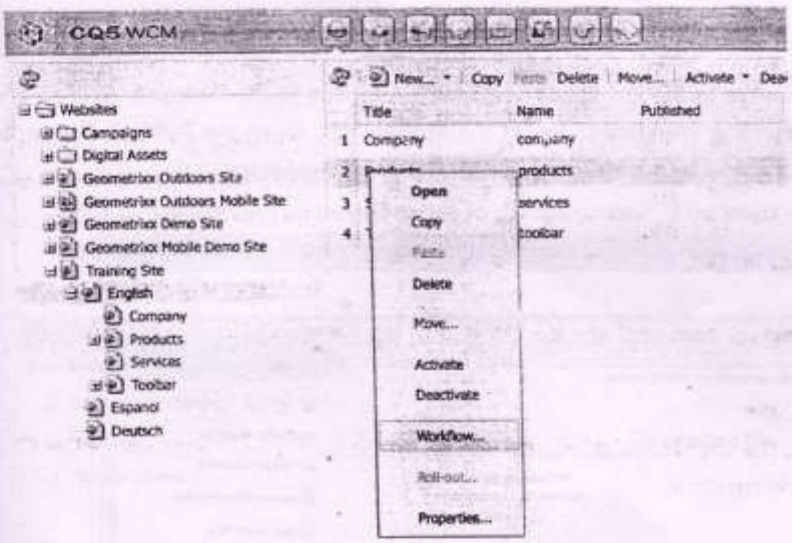
- Workflow Console
- SiteAdmin Console (Websites tab)
  - Right Context Menu
  - Workflow item in Toolbar
- Sidekick

EXERCISE - Explore Workflow Basics

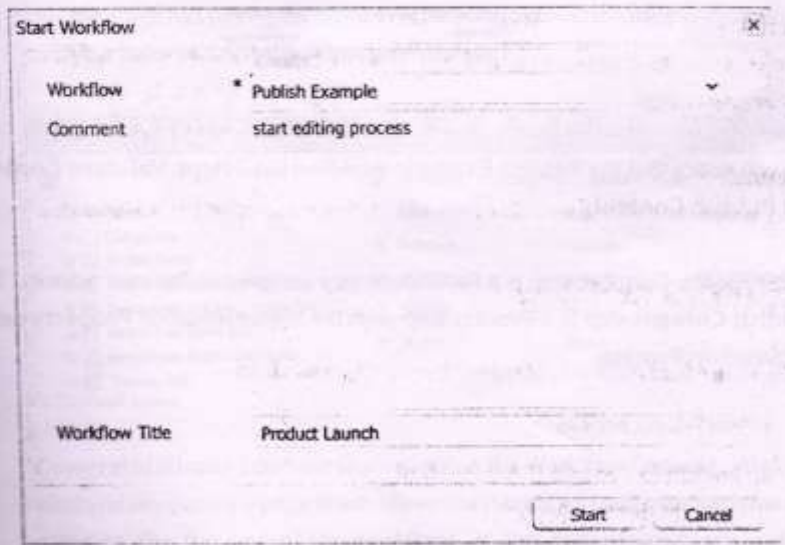
We will be starting the workflow from the SiteAdmin console.

1. Open the SiteAdmin console, either by clicking on the Websites button from the Welcome page or by clicking on the globe icon at the bottom of your sidekick.

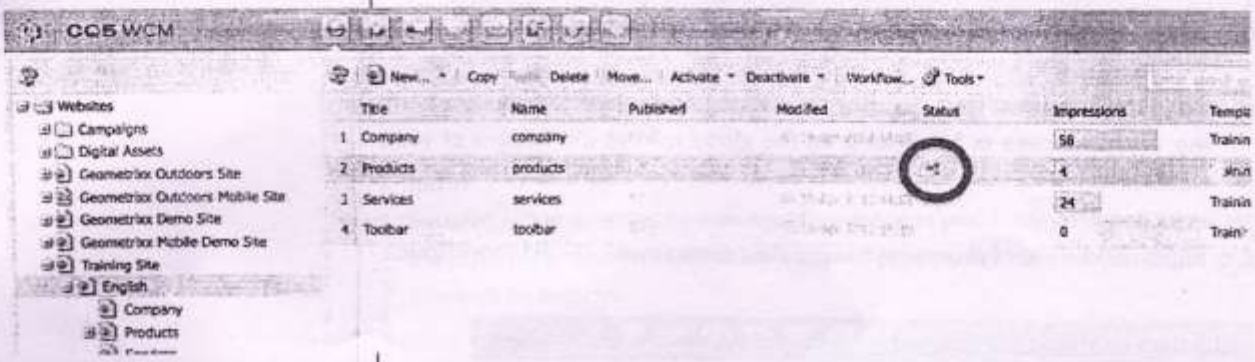
2. Right-click on one of your pages to open the context menu. Select **Workflow...**



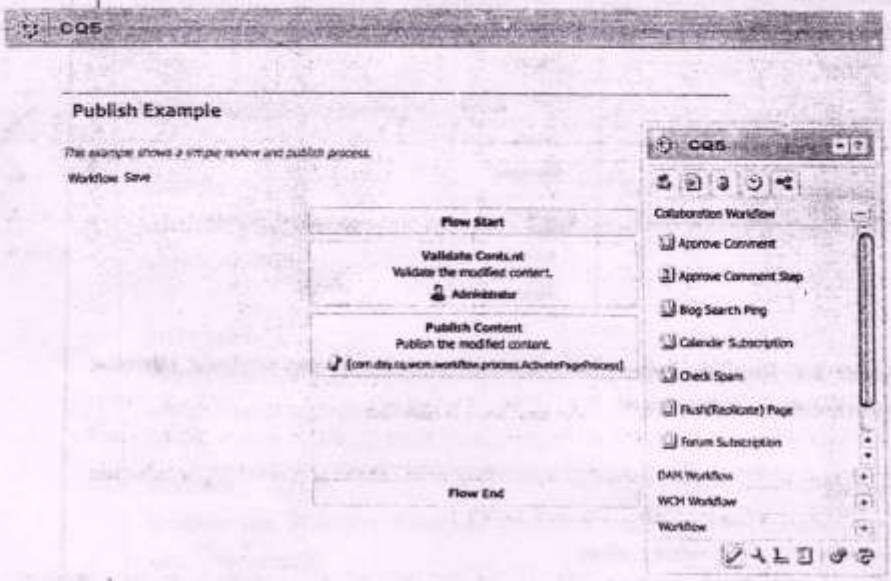
3. Select the **Publish Example** workflow and fill in any desired optional information and click **Start**



4. You will notice that the workflow icon shows up in the SiteAdmin console next to the page you chose.



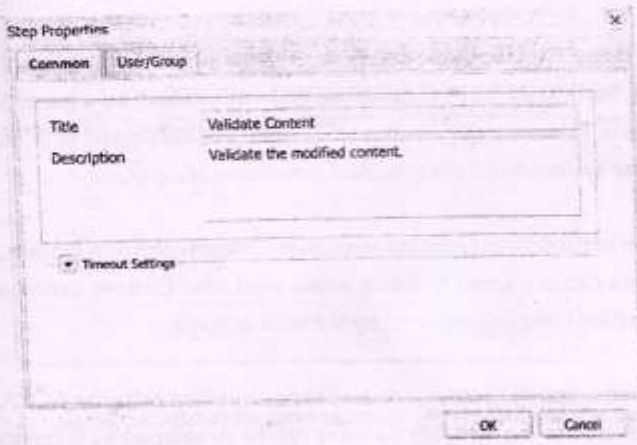
5. Change to the Workflow Console by returning to the Welcome Screen and selecting Workflows. Select the **Models** tab. Open the **Publish Example** workflow by double-clicking the workflow model entry.



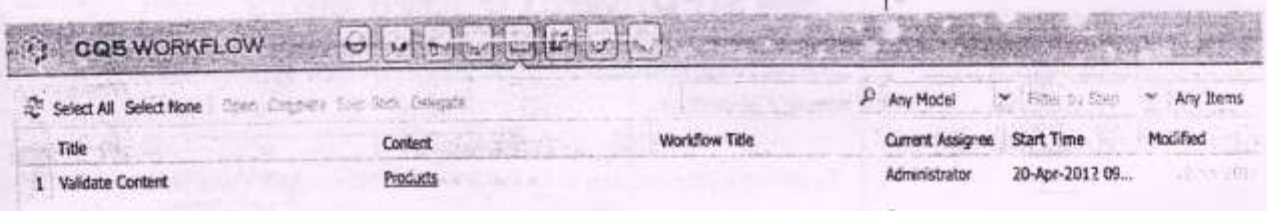
You will notice that the **Publish Example** workflow has 2 steps: **Validate Content** and **Publish Content**.

The **Validate Content** step is a Participant step assigned to the user admin. The **Publish Content** step is a Process step with the Implementation Property set to ActivatePageProcess.

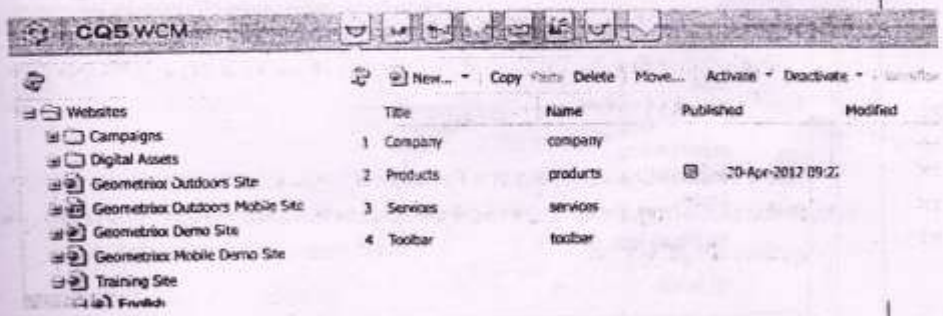
You can validate the values of the step properties by double-clicking on the step to open the dialog box.



6. Switch to the Inbox by changing to either the Welcome Screen or the SiteAdmin Screen and clicking on the **Inbox** tab to change context. The Inbox will show the WorkItem entry for the page you put into workflow.



7. Select the WorkItem **Validate Content** and click **Complete**. Notice that the WorkItem disappears from the Inbox. You will also note that the page is now no longer in workflow (the workflow icon is gone from the status column) and the page has been published or is pending publication (depending on whether you have a CQ5 Publish instance running).



Congratulations! You have learned about the Workflow Console. Also, you have successfully placed a page in workflow and moved that page through the workflow steps to the the end of the workflow. In the next exercise, you will use this knowledge to help you create a custom process step and define a workflow to use that custom process step.

Create a Workflow Implementation Step

A workflow is made of steps. The steps that define a workflow are either participant steps or process steps. Participant steps require manual intervention by a person to advance the workflow. Process steps, on the other hand, are automatic actions that are executed by the system if certain specified conditions are met.

CQ5 provides a number of predefined process steps that perform common actions, which an administrators can use when building a new workflow. Custom process steps can also be added for tasks not covered by the built-in steps.

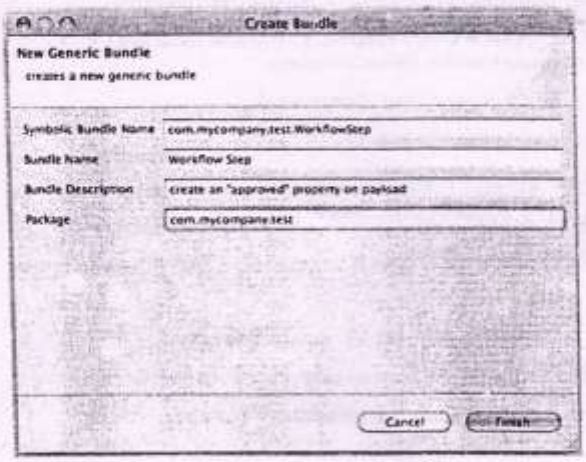
Process steps, also called automated steps, can be defined by using either an ECMA script or a service (a Java class in a bundle). Services can be developed to listen to special workflow events and perform tasks according to the business logic.

EXERCISE - Defining a process step: with a Java class

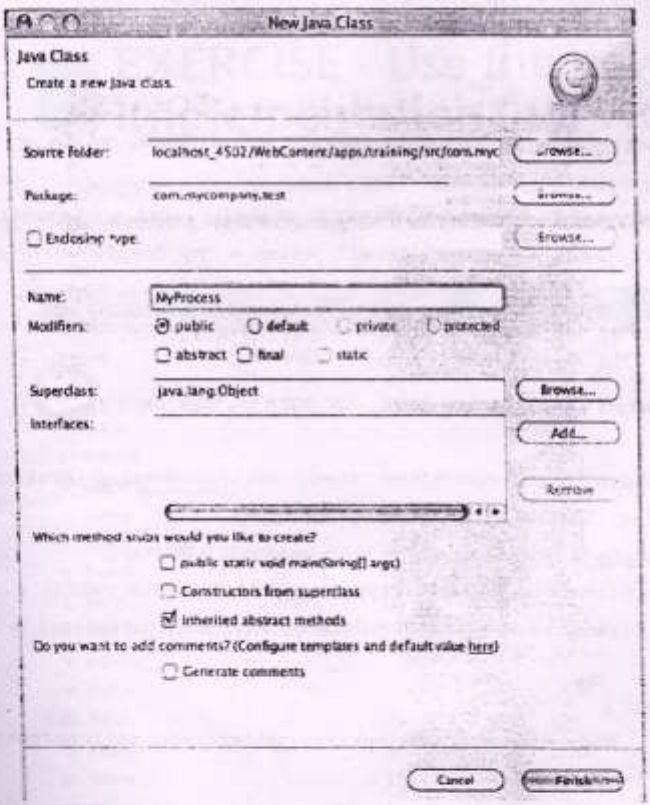
The following instructions explain how to create a workflow custom process step using a Java class.

To define a process step as an OSGI service component (Java bundle):

1. Create the bundle to hold the Java class using **Build..Create Bundle**.



2. Create a Java class to implement the creation of the **approved** property.



3. Open the newly created Java class file ***MyProcess.java***, and enter some Java code, similar to below - then **Save**.

```
package com.mycompany.test;

import com.day.cq.workflow.WorkflowException;
import com.day.cq.workflow.WorkflowSession;
import com.day.cq.workflow.exec.WorkItem;
import com.day.cq.workflow.exec.WorkflowData;
```

```
import com.day.cq.workflow.exec.WorkflowProcess;
import com.day.cq.workflow.metadata.MetaDataMap;
import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Properties;
import org.apache.felix.scr.annotations.Property;
import org.apache.felix.scr.annotations.Service;
import org.osgi.framework.Constants;

import javax.jcr.Node;
import javax.jcr.RepositoryException;

/**
 * Sample workflow process that sets an <code>approve</code> property
to the payload based on the process argument value.
 */
@Component
@Service
@Properties({
         @Property(name = Constants.SERVICE_DESCRIPTION, value = "A
sample workflow process implementation."),
        @Property(name = Constants.SERVICE_VENDOR, value = "Adobe"),
        @Property(name = "process.label", value = "My Sample Workflow
Process")})
public class MyProcess implements WorkflowProcess {

    private static final String TYPE_JCR_PATH = "JCR_PATH";

      public void execute(WorkItem item, WorkflowSession session,
MetaDataMap args) throws WorkflowException {
        WorkflowData workflowData = item.getWorkflowData();
        if (workflowData.getPayloadType().equals(TYPE_JCR_PATH)) {
              String path = workflowData.getPayload().toString() + "/
jcr:content";
            try {
                Node node = (Node) session.getSession().getItem(path);
                if (node != null) {
                    node.setProperty("approved", readArgument(args));
                    session.getSession().save();
                }
            } catch (RepositoryException e) {
                throw new WorkflowException(e.getMessage(), e);
            }
        }
    }

    private boolean readArgument(MetaDataMap args) {
```

```
        String argument = args.get("PROCESS_ARGS", "false");
        return argument.equalsIgnoreCase("true");
    }
}
```

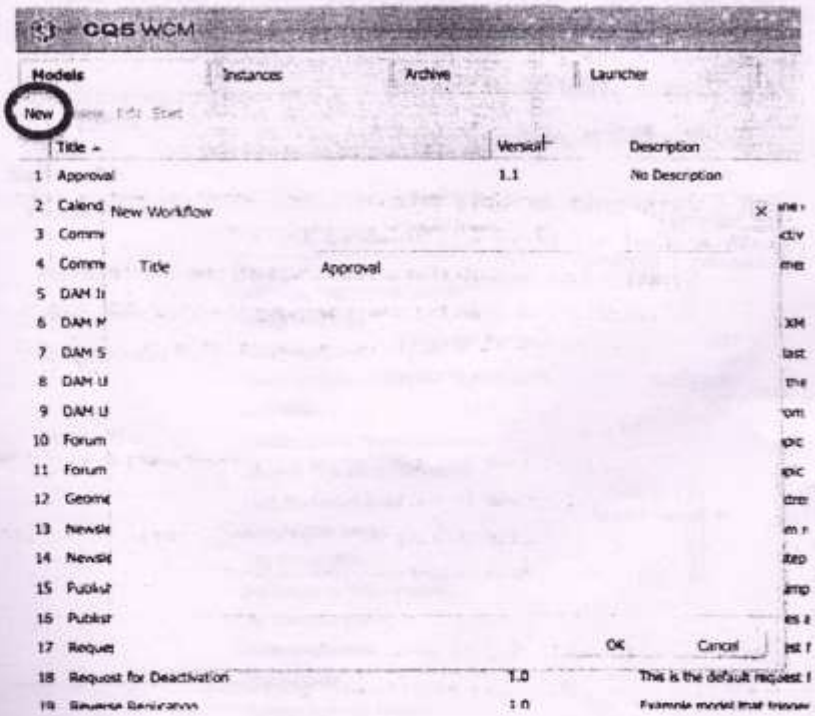
4. Compile the Java class using Build...Compile. You should get exactly 7 error, associated with the scr annotations. However, if you examine the Problems tab in CRXDE, you will see that the compile succeeded.

5. Build the Bundle using **Build..Build Bundle**. Once you get the "Build Successful" message, you will notice that the bundle *.jar* file shows up in the install directory. The bundle is now uploaded and installed into CQ5.

Congratulations! You have created a new Java class that can be used as a custom Implementation in a Process step.

EXERCISE - Use the new Process Implementation in a Workflow

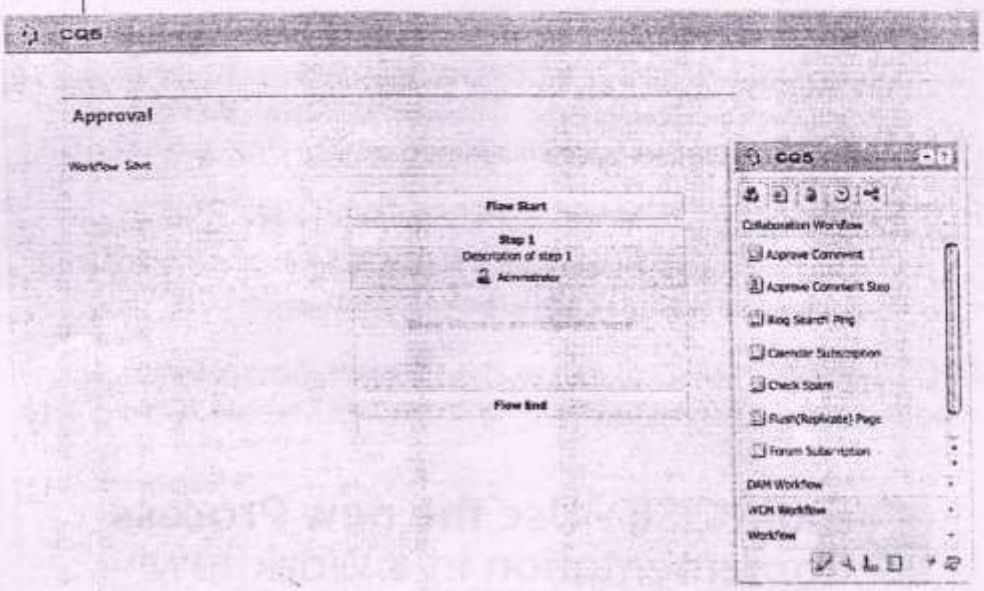
1. Navigate to the Workflow panel. Choose the **Models** tab. Create a workflow by choosing **New** from the toolbar. Enter *Approval* as the Title of the workflow.



2. Open the Approval workflow by double-clicking on the Name.

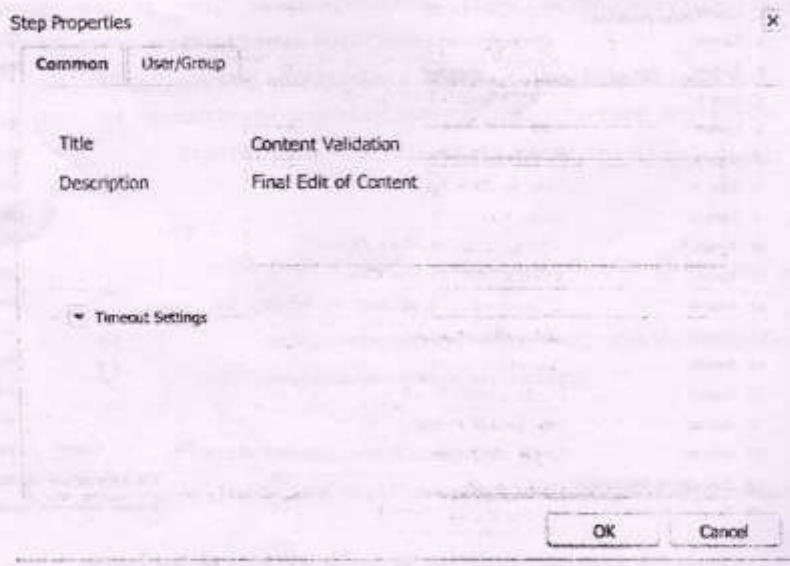
NOTE: The java methods, respectively the classes that implement the executable Java method are registered as OSGI services, enabling you to add methods at anytime during runtime.

3. You will notice that the new **Approval** workflow model has a Start, End and Step 1.

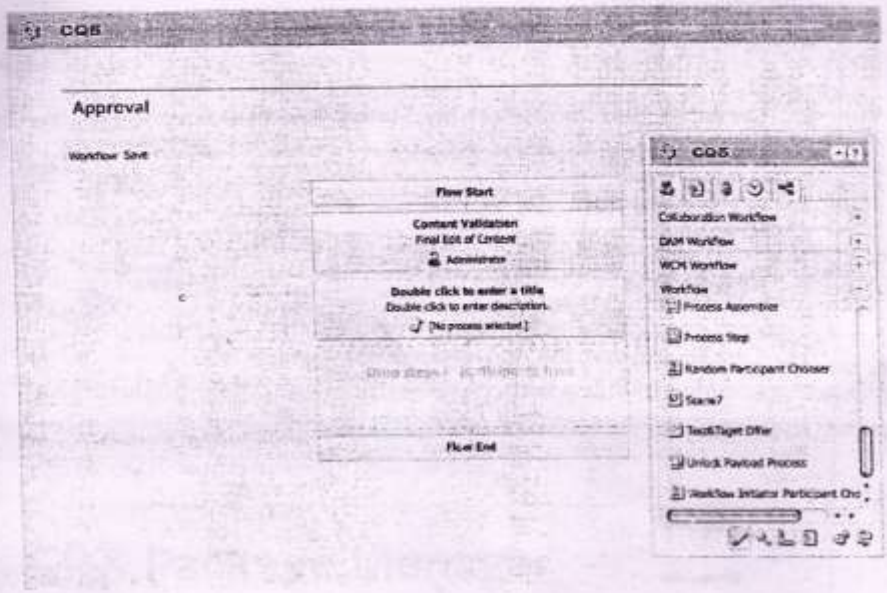


To edit Step 1, double-click on the step itself.

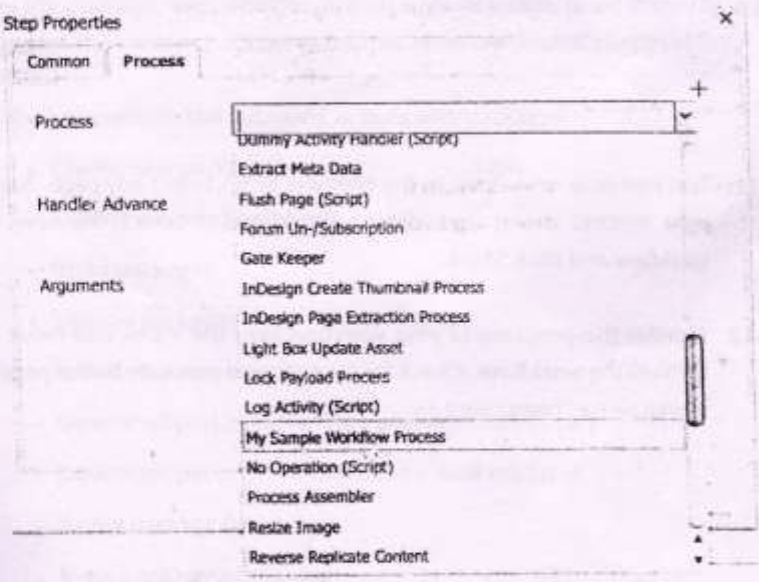
4. Assign the following values for the Step 1 properties:
   - Title = Content Validation
   - Description = Final Edit of Content
   - User/Group = /home/groups/contributors



5. Add a Process step to the workflow by dragging-and-dropping the Process Step component onto the workflow model.



6. Open the Process Step dialog and set the properties of your new step:
   - Description = Set "approved" property
   - Implementation = My Sample Workflow Process
   - Title = Final Approval
   - Process Arguments = true



NOTE: Select this option to automatically advance the workflow to the next step after execution. If not selected, the implementation script must handle the workflow advancement.

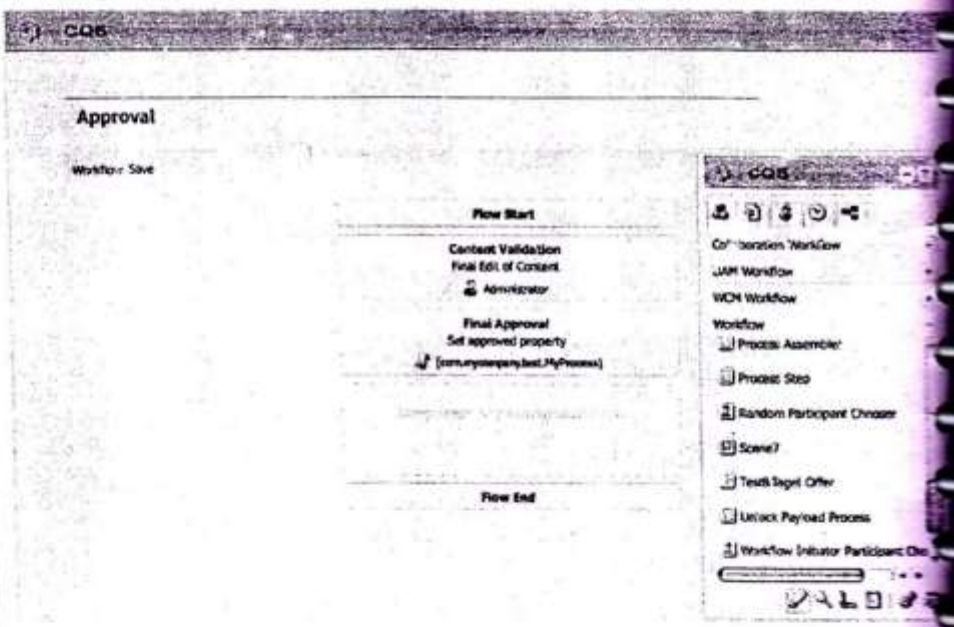
7. On the Common Tab fill in the Title and Description:

Title = Final Approval
Description = Set approved property

8. On the Process Tab, select My Sample Workflow Process from the list of processes. Check Handler Advance.
   - Arguments = true

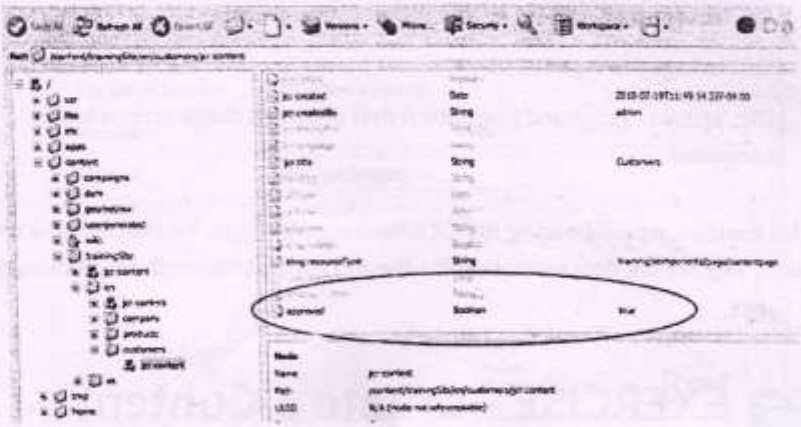
9. Click OK to close the dialog box.

10. Click Save to save the Approval workflow.



11. Test your new workflow. In the WebSites panel, select any page. Bring up the right context menu and choose **Workflow...** Select the new Approval workflow and click **Start.**

12. Monitor the progress of your workflow from the inbox and move the page through the workflow. Check for the *approved* property on the page once the workflow has completed.



Congratulations! You have successfully created a custom Implementation for a workflow process step. You can use this same methodology for any custom Implementation you need to create for your workflows.

CRX Package Manager

Packages can include content and project-related data. A package is a zip file that contains the content in the form of a file-system serialization (called "vault" serialization) that represents the content from the repository as an easy-to-use-and-edit representation of files and folders.

Additionally, it contains vault meta information, including a filter definition, and import configuration information. Additional content properties can be included in the package, such as a description, a visual image, or an icon. These properties are for the content package consumer for informational purposes only.

You can perform the following actions with packages:

- Create new packages
- Modify existing packages
- Build packages
- Upload packages
- Install packages
- Download packages from the package share library
- Download packages from CQ5 to a local machine
- Apply package filters
- View package information

There are multiple ways to manage content packages:

- CRX Package Manager - using the direct CRX interface
- CQ5 Package Manager
- cURL actions - command line option that allows package actions to be automated

In this exercise, we will be using the CRX Package Manager. You should take some time to explore the documentation for the CQ5 and cURL methods of managing packages.

EXERCISE - Create a Content Package of Everything We Have Done

The following instructions explain how to create a CQ5 package that will combine all elements of the Training project, minus all jpegs. This is a good example of packaging application content, which you could then distribute to team members for review.

1. Navigate to the CRX Main Console.
2. Select the **Package**.

CQ 5.5 packages

3. Select **Create Package**.
4. Enter the package "Package Name" (CQ5Dev Training) and "Group Name" (my-project).

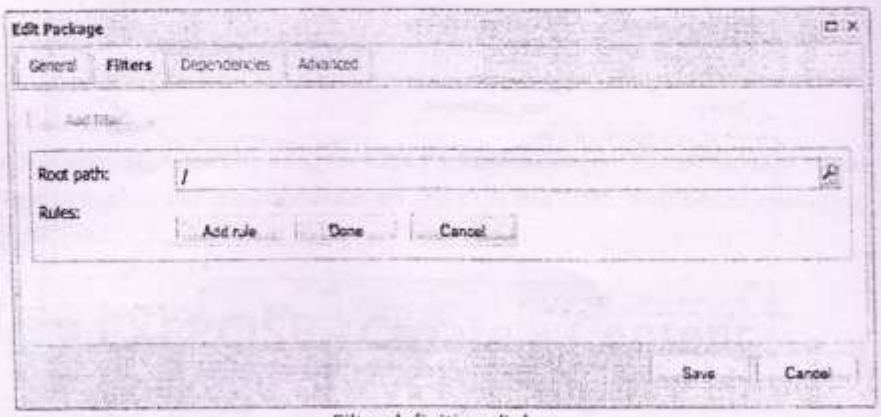
CRX Package Manager new package dialog

5. The result is an empty package definition.

6. Click the Edit button to open the Edit Dialog.

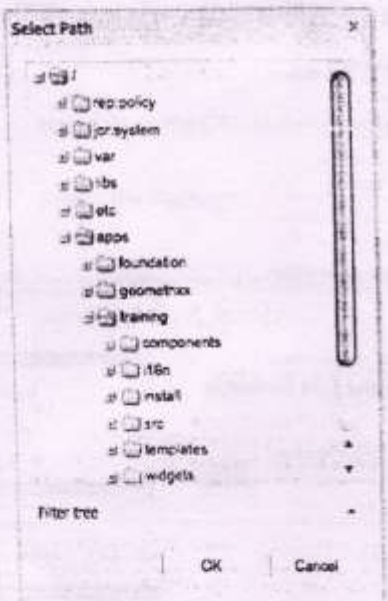
CRX Package Manager Edit Dialog

7. Select the Filter tab and Click **Add Filter.**



Filter definition dialog

8. Click on the browser icon to select the paths to be included in the package. Choose */apps/training*.



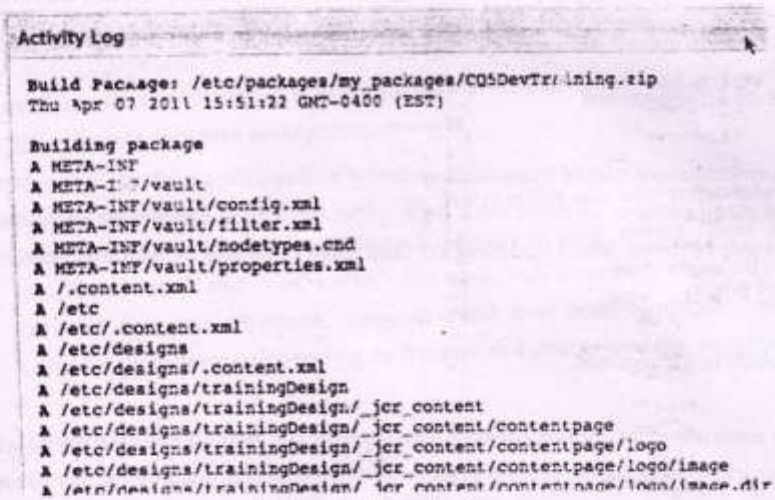
Package Filter Paths

Click **OK** and then **Done.**

9. Repeat for the following paths:
   - /apps/foundation/components/breadcrumb
   - /etc/designs/trainingDesign
   - /etc/workflow
   - /content/trainingSite

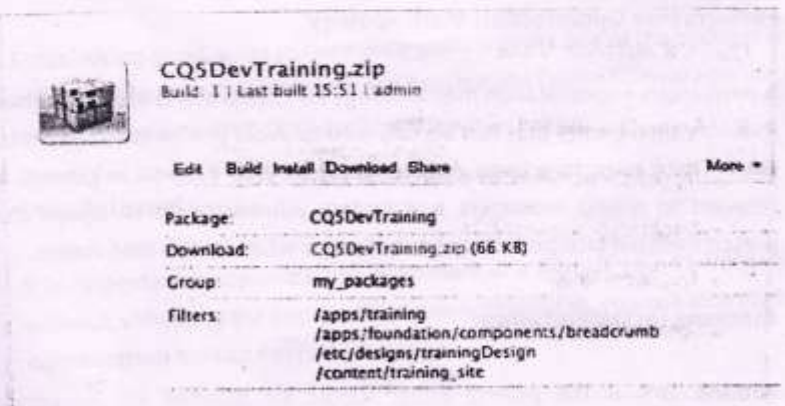
This creates a package definition to save everything that we have done this week into the content package. You can then upload all these structures into any instance of CQ5.5 .

10. Preview by clicking **More...Coverage** and then Build the package. The Activity Log will show you what is being included in the package.



Package build output

11. Download the package by clicking on the package name



Congratulations! You have successfully created a package, added a rule to the filter definition, built the package, and have downloaded the package, which you can now share with your CQ5 development team.

Performance

What performance optimization concepts should I consider?

A key issue is the time your Web site takes to respond to visitor requests. Although this value will vary for each request, an average target value can be defined. Once this value is proven to be both achievable and maintainable, it can be used to monitor the performance of the Web site and indicate the development of potential problems.

The response times you will be aiming for will be different on the author and publish environments, reflecting the different characteristics of the target audience:

Author Environment

This environment is used by authors entering, and updating content, so it must cater for a small number of users who each generate a high number of performance intensive requests when updating content pages and the individual elements on those pages.

Publish Environment

This environment contains content which you make available to your users, where the number of requests is even greater and the speed is just as vital, but since the nature of the requests is less dynamic, additional performance enhancing mechanisms can be leveraged, such as that the content is cached or load-balancing is applied.

Performance Optimization Methodology

A performance optimization methodology for CQ5 projects can be summed up to five very simple rules that can be followed to avoid performance issues from the start. These rules, to a large degree, apply to Web projects in general, and are relevant to project managers and system administrators to ensure that their projects will not face performance challenges when launch time comes.

Planning for Optimization

Around 10% of the project effort should be planned for the performance optimization phase. Of course, the actual performance optimization requirements

will depend on the level of complexity of a project and the experience of the development team. While your project may ultimately not require all of the allocated time, it is good practice to always plan for performance optimization in that suggested range.
Whenever possible, a project should first be soft-launched to a limited audience in order to gather real-life experience and perform further optimizations, without the additional pressure that follows a full announcement.

Once you are "live", performance optimization is not over. This is the point in time when you experience the "real" load on your system. It is important to plan for additional adjustments after the launch.

Since your system load changes and the performance profiles of your system shifts over time, a performance "tune-up" or "health-check" should be scheduled at 6-12 months intervals.

Simulate Reality

If you go live with a Web site and you find out after the launch that you run into performance issues there is only one reason for that: Your load and performance tests did not simulate reality close enough.

Simulating reality is difficult and how much effort you will reasonably want to invest into getting "real" depends on the nature of your project. "Real" means not just "real code" and "real traffic", but also "real content", especially regarding content size and structure. Keep in mind that your templates may behave completely different depending on the size and structure of the repository.

Establish Solid Goals

The importance of properly establishing performance goals is not to be underestimated. Often, once people are focused on specific performance goals, it is very hard to change these goals afterwards, even if they are based on wild assumptions.

Establishing good, solid performance goals is really one of the trickiest areas. It is often best to collect real life logs and benchmarks from a comparable Web site (for example the new Web site's predecessor).

Stay Relevant

It is important to optimize one bottleneck at a time. If you do things in parallel without validating the impact of the one optimization, you will lose track of which optimization measure actually helped.

Agile Iteration Cycles

Performance tuning is an iterative process that involves, measuring, analysis, optimization and validation until the goal is reached. In order to properly take this aspect into account, implement an agile validation process in the optimization phase rather than a more heavy-weight testing process after each iteration.

This largely means that the developer implementing the optimization should have a quick way to tell if the optimization has already reached the goal, which is valuable information, because when the goal is reached, optimization is over.

Basic Performance Guidelines

Generally speaking, keep your uncached html requests to less than 100ms. More specifically, the following may serve as a guideline:

- 70% of the requests for pages should be responded to in less than 100ms.
- 25% of the requests for pages should get a response within 100ms-300ms.
- 4% of the requests for pages should get a response within 300ms-500ms.
- 1% of the requests for pages should get a response within 500ms-1000ms.
- No pages should respond slower than 1 second.

The above numbers assume the following conditions:

- measured on publish (no authoring environment and/or CFC overhead)
- measured on the server (no network overhead)
- not cached (no CQS-output cache, no Dispatcher cache)
- only for complex items with many dependencies (HTML, JS, PDF, ...)
- no other load on the system

There are a certain number of issues that frequently contribute to performance issues which mainly revolve around (a) dispatcher caching inefficiency and (b) the use of queries in normal display templates. JVM and OS level tuning usually do not lead to big leaps in performance and should therefore be performed at the very tail end of the optimization cycle.

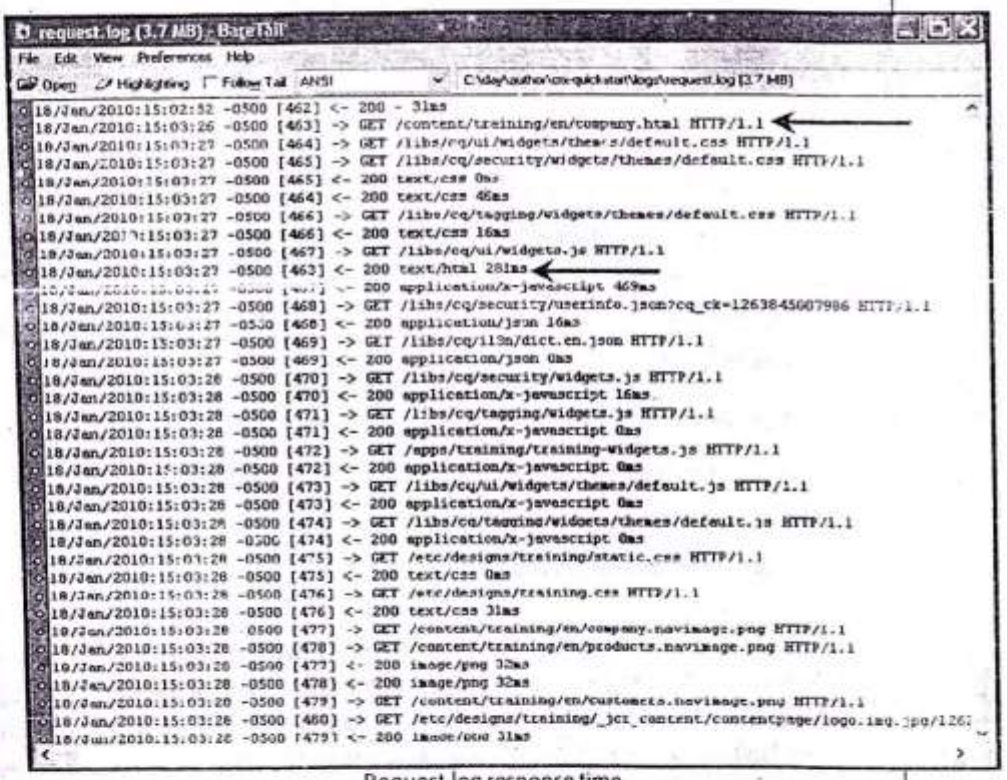
Your best friends during a usual performance optimization exercise are the request.log, component based timing, and last but not least - a Java profiler.

EXERCISE - How to monitor Page response times

1. Navigate to and open the file request.log located at *<cq-install-dir>/crx-quickstart/logs.*

2. Request a Page in author that utillizes your Training Template and Components.
   - e.g. /content/trainingSite/en/company

3. Review the response times directly related to the previous step's request.
   - A Page request of /content/trainingSite/en/company



Request.log response time

NOTE

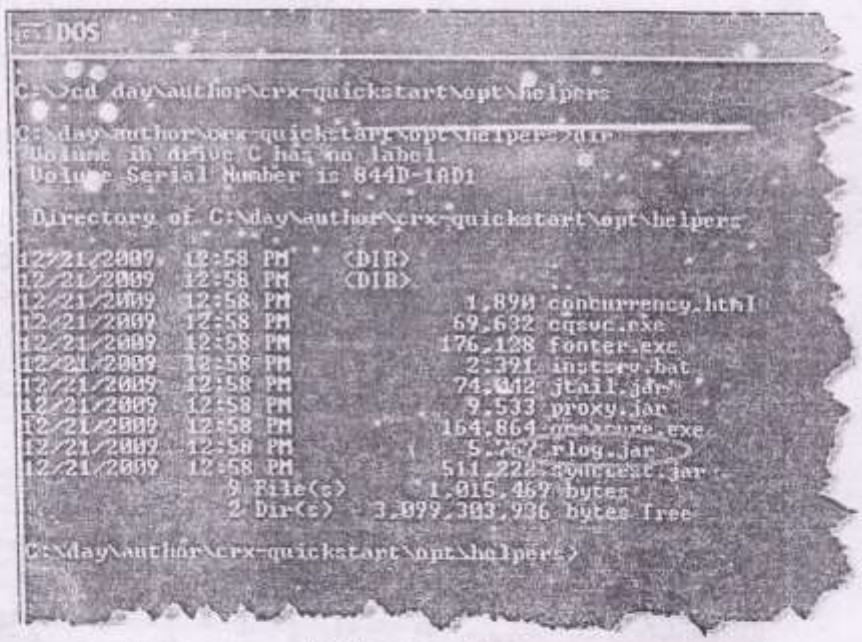
Though this is clearly the response time of a Page while in author, it is good practice to be able to identify where response times are located (request.log) and how to identify a specific Page request/response in the log file itself. Ideally, this test would occur on the publish instance to get a better idea of its expected behavior in production.

Congratulations! You have successfully reviewed the response time of a Page in CQ5 using the request.log. Again, this will aid your development in being able to

monitor response times of Pages that implement custom Templates and Components, and comparing said time to your project goals.

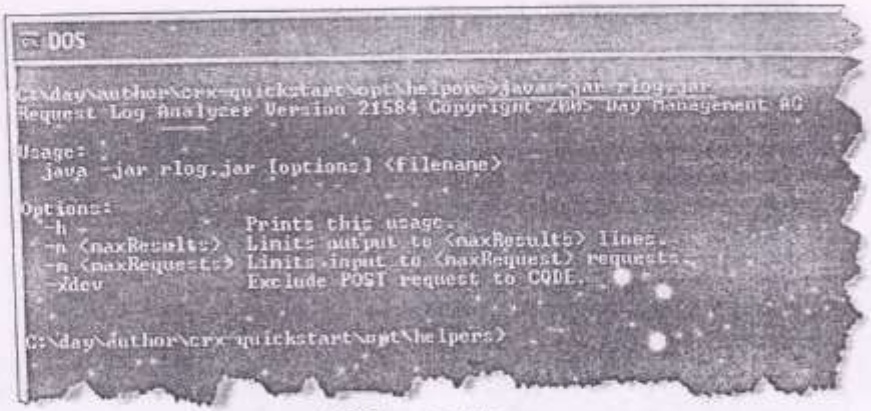
EXERCISE - How to find long lasting request/response pairs

1. Navigate to the helper tool rlog.jar located in <cq-install-dir>/crx-quickstart/opt/helpers using your command line.



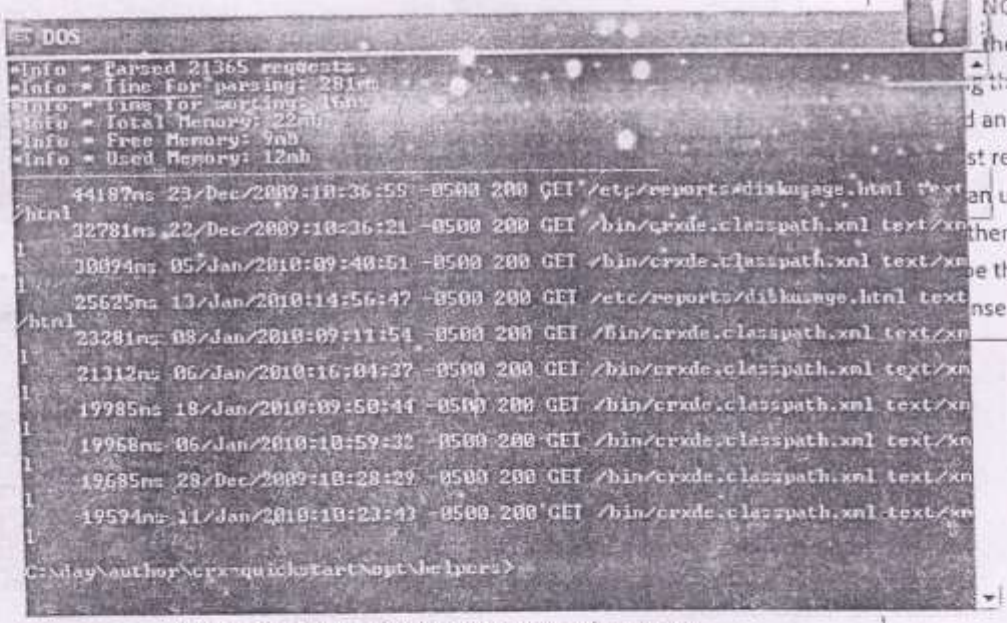
DOS location of rlog.jar

2. Enter the command java -jar rlog.jar in your command line to get help concerning possible arguments.



DOS rlog.jar help

3. Enter the command **java -jar rlog.jar -n 10 ../../logs/request.log** in your command line to view the first 10 requests with the longest response duration.



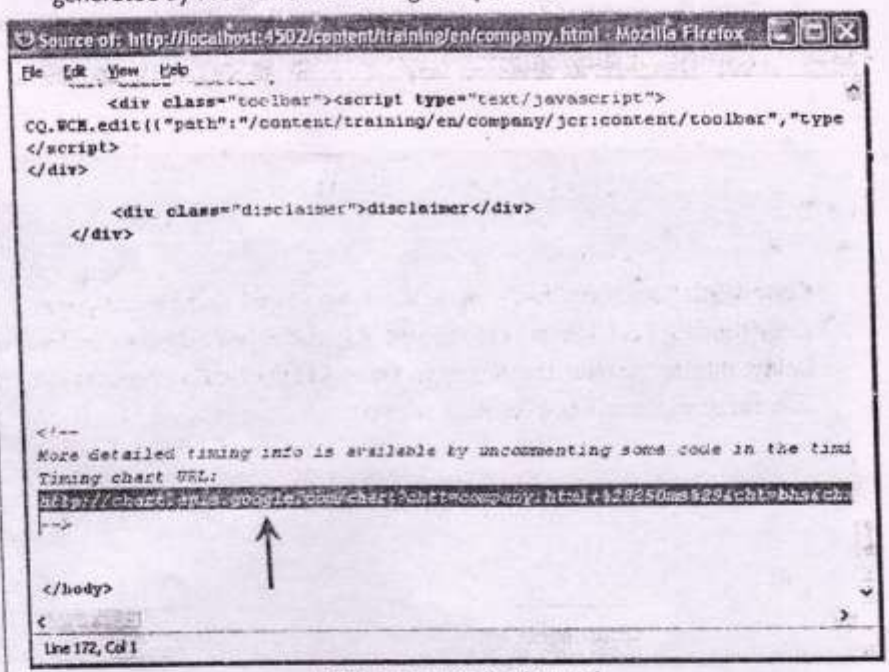
DOS rlog.jar view of 10 longest requests/responses

NOTE: Essentially, the rlog.jar tool is the request.log and display the 10 st requests/responses. an use this information ther investigate what e the cause of the long nse.

Congratulations! You have successfully found and displayed long lasting requests/responses in CQ5. Again, this is just one of many tools to help you meet your project's performance goals. Finally, we will view timing statistics for Page rendering.

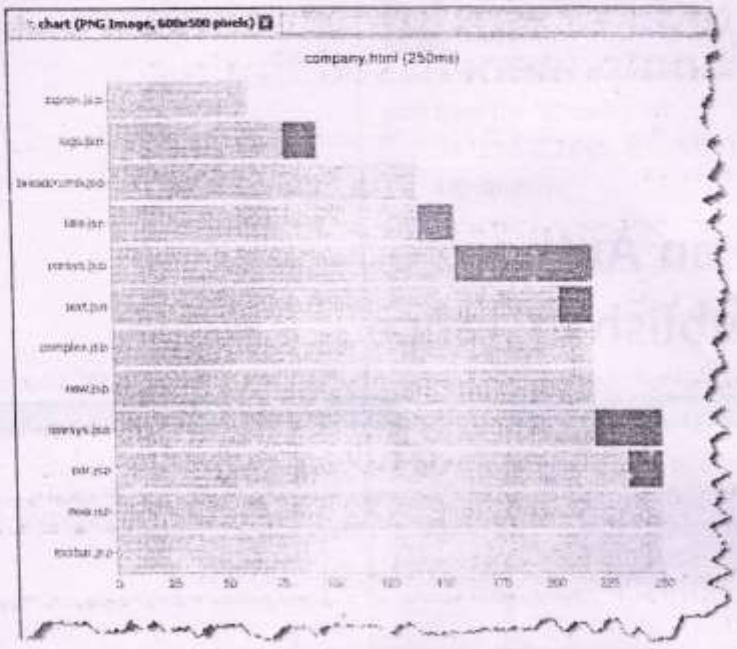
EXERCISE - How to monitor Component based timing

1. Request a Page in author that was created from your Training Template and Components.
   - e.g. /content/trainingSite/en/company
2. View the HTML source of the Page requested in step 1.
3. Navigate to and select the "Timing chart URL" located in the HTML source.
   - You will find this URL most likely near the bottom of the HTML source, as it is generated by the foundation timing Component



HTML source timing chart url

4. Copy the "Timing chart URL" - then paste it in the address bar of your favorite Web browser.
5. Investigate the visual output to identify any Component that may be causing a slow response time.



Timing chart component response times

Congratulations! You have successfully monitored Component based timing using the foundation timing Component. Again, this will aid your development in being able to monitor the response times of specific Components within the context of an actual Page/Template request.

NOTE: The light blue color identifies when the Component/JSP started. The dark blue color identifies how long the Component/JSP took to respond.

Appendix A Clone an Author Instance to be a Publish Instance

What is an Publish instance?

A Publish instance is the CQ5 installation from which web site visitors will request pages.

EXERCISE - How to clone an Author instance

The following instructions explain how to clone an Author instance. This is important because you will often need a Publish instance as part of your component testing.

1. Make sure that your Author instance is stopped. You cannot successfully clone a running CQ5 instance.
2. Copy your author folder (e.g. C:/day/CQ5/author).
3. Paste your author folder into the same folder structure (e.g. C:/day/CQ5).
4. After the copy rename your author-copy folder to publish (e.g. C:/day/CQ5/publish).
5. Rename the CQ5 quickstart JAR to cq-publish-4503.jar.
   - cq = the application
   - publish = the WCM mode it will run in (e.g. author or publish)
   - 4503 = the port it will run in (e.g. any available port is acceptable)

If no port number is provided in the file name, CQ5 will select the first available port from the following list: 1) 4502, 2) 8080, 3) 8081, 4) 8082, 5) 8083, 6) 8084, 7) 8085, 8) 8888, 9) 9362, 10) random.

NOTE: The cloning method used here is a developer convenience. To clone a CQ5 instance for production, please consult the documentation and the CQ5 knowledge base.

NOTE

If no port number is provided in the file name, CQ5 will select the first available port from the following list:

1) 4502, 2) 8080, 3) 8081, 4) 8082, 5) 8083, 6) 8084, 7) 8085, 8) 8888, 9) 9362, 10) random.

6. Double-click the cq-publish-4503.jar file.
   - A dialog will pop-up similar to the one below

CQ5 install/startup dialog

See the installation section above for instructions on starting CQ5 with the command line or the batch jobs.

Congratulations! You have successfully cloned and started a CQ5 publish instance. To start CQ5 in the future, double-click the renamed CQ5 quickstart JAR file (e.g. cq-publish-4503.jar).