Adobe Experience Manager Ramp Up Training

Index

Day 1:

An AEM Introduction

Today's content:

- WCM
- AEM
- OSGi (introduction)
- JackRabbit
- Content Repository EXtreme
- AEM environment
- Development (basic)

WCM (web content manager)

A Web Content Management System (WCMS) is a software system that provides website authoring, collaboration, and administration tools designed to allow users with little knowledge of web programming languages or markup languages to create and manage website content with relative ease. A robust WCMS provides the foundation for collaboration, offering users the ability to manage documents and output for multiple author editing and participation.

The author works in what is known as the **author environment**. This provides an easy to use, graphical user interface for creating the content. It is usually located behind a company's firewall that provides full protection.

When ready the content is published to the **publish environment**. For a normal Web site, this is located inside the demilitarized zone; in other words, available to the Internet, but no longer under the full protection of your internal network.

AEM allows you to manage campaigns through marketing automation software that helps you manage multi-channel campaigns. In the AEM, you manage segments, all your campaigns, as well as lists, leads, and reports.

AEM components and templates form a very powerful tool kit. They can be used by developers to provide website business users, editors, and administrators with the functionality to adapt their websites to changing business needs (content agility) while retaining the uniform layout of the sites (brand protection).

The product has been completely redesigned from Communiqué 4, allowing Day to use new architecture and technologies, thus increasing functionality while reducing complexity. Extensive use of standards helps ensure long-term stability.

Features

DAM

AEM DAM (Digital Asset Management) is a digital asset management tool that is fully integrated with the AEM platform and enables your enterprise to share and distribute digital assets. Users across an organization can manage, store, and access images, videos, documents, audio clips, and rich media such as Flash files for use on the web, in print, and for digital distribution through a Web interface (or a CIFS or WebDAV folder).

In-Context-Editing

During edition, gives the author the ability to change text over the page without using dialogs or changing code, this will happen thanks to some node configuration and the effect will be as if working on the page from a text editor.

Social Collaboration

Moderate content from social network, create forums, generate feeds and manage components like calendars.

Key Principles

AEM is based on standards (open and close) and Open Source.

Open standards refer to open code along with the specification often referred as open source but normally used to apply to lesser technologies or protocols widely used and well known. Close standard is a widely use technology which is proprietary and only the supplier knows how is implemented.

- Java/J2EE
- JSR (Java Specification Request) 283 – A Java Content Repository (JCR) by David Nuescheler
- Apache Felix – implementation of the service platform (console) based on OSGi Framework
- Apache Sling – a web framework for the Java platform designed to create content-centric applications on top of JCR
- Platform independence.
- Everything is content. From HTML to a servlet, from images to videos

AEM

A typical challenge for a person responsible for a website, or set of websites (for example in a branch office of a global enterprise), is to introduce a new type of content presentation on their websites.

AEM is a set of content management services built on a content-centric Web Application Framework on top of a Java Content Repository

Day Software was an enterprise content management software company headquartered in Basel, Switzerland and Boston, Massachusetts. Day Software was founded in 1993 in Basel, Switzerland. Day creates content management infrastructure software in the Day CRX product line and enterprise content management applications in the

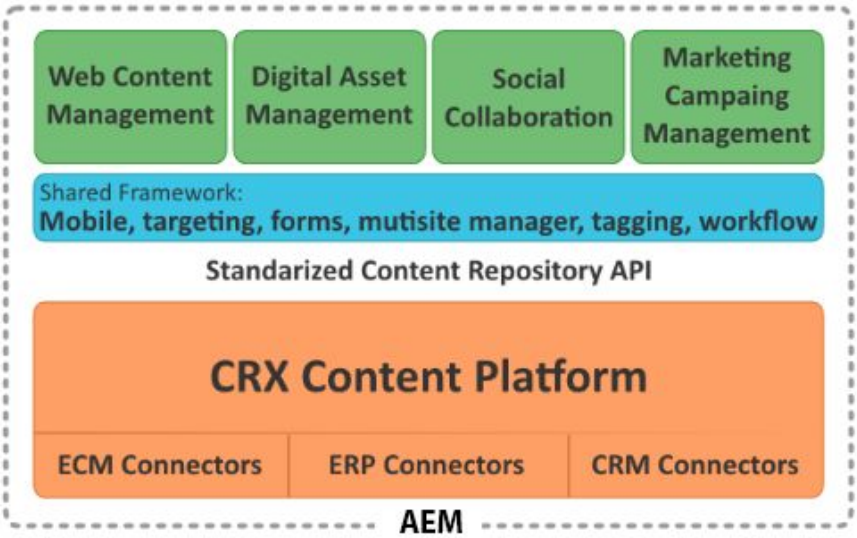
Day Communiqué product line. Its most important product is Day Communiqué WCM, a Web content management system that is used for public-facing websites and intranets of large corporations such as McDonald's and Daimler.

Day is engaged in the content repository API for Java standardization process and contributes to open source software projects such as Apache Jackrabbit and Apache Sling.

Day Software was acquired by Adobe Systems on 28th July, 2010 for $240 Million USD, and its star product was rebranded as Adobe CQ Platform (now known as AEM).

AEM Architecture

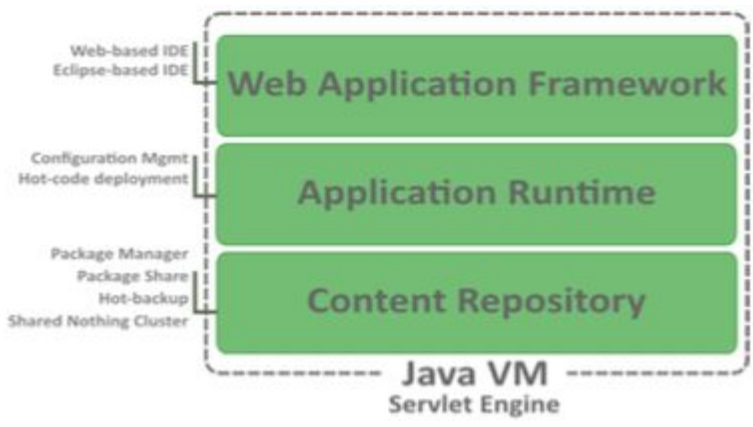
AEM uses a web-based graphical user interface, so you need a web browser to access AEM. The Architecture of AEM is based on multiple building blocks.



From a top-down perspective we have a group of applications used to interact with the user and provide management options to the content ranging from creating new content to modify it to delete it. Web Content Management (previously discussed) is one of the major features and focus to most web development, adding the DAM (Digital Asset Manager) to the formula full simple web sites can be created from scratch. Social Collaboration provides unification of all social networking and collaboration applications within WCM, things like blogs, wikis and forums are part of this.

The Marketing Campaign Manager (MCM) is marketing automation software that helps you manage multi\channel campaigns. In the CQ MCM, you manage segments, all your campaigns, as well as lists, leads, and reports. From the MCM dashboard, you can access leads, lists, and campaigns.

Moving to the second layer of the building blocks we have a list of "tools" and APIs meant to support all the process generated on the first

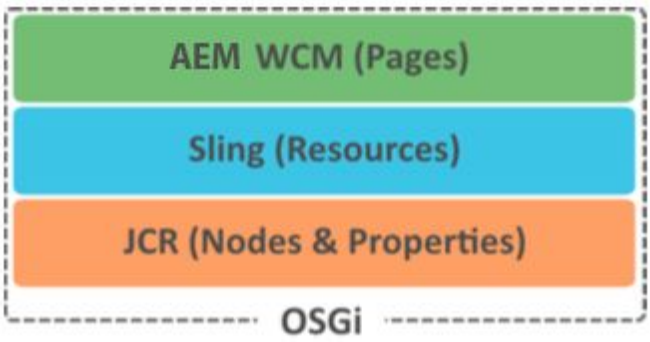


layer. Things like mobile support for different device groups, forms and its interaction with workflows, as well as workflows by itself are some of the supporting features of this technology.

On the last layer we have the CRX working as either an ECM (Enterprise Content Manager) or connecting to an existing one, like Interwoven or SharePoint. Also allows connections ERPs as Lotus Notes (Domino Server) and CRMs (Customer Relationship Management).

While developing for AEM the developer will run into the different parts of AEM, like the Web Application Framework (Sling implementation) usually accessed through one of AEM official tools. A web-based IDE named CRXDE Lite and its counterpart, an eclipse-based IDE named CRXDE. The CRXDE comes with the basic AEM installation and CRXDE should be downloaded. They have the same basic feature but they have some obvious cons\&\pros. While CRXDE Lite is fast to load and take low memory, is lacking on autocomplete and configuration. And the CRXDE is based on eclipse and its memory footprint tends to be a bit high, some machines might have problems with this.

The AEM technology, thanks to the OSGi Framework and Sling Framework, handles hot-code deploy, meaning that there is no need to restart the server (instance) after adding a new site, component or library, this all could be resolved during runtime.



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.From all this we can summarize that for JCR everything is stored as nodes and properties, this is seen as “content”. For Sling everything is a “resource” and by extension for the WCM everything is a “page” or could be rendered as one.

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 is a set of specifications that define a dynamic component (or module) system for Java.

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.

Apache! Felix is a community effort to implement the OSGi R4 Service Platform and other interesting OSGi-related technologies under the Apache license. The OSGi specifications originally targeted embedded devices and home services gateways, but they are ideally suited for any project interested in the principles of modularity, component-orientation, and/or service-orientation. Felix is the OSGi Console used by AEM to implement OSGi services.

OSGi Dynamic Module System

For AEM, bundles are dropped into specially named folders in the repository.

Universal middleware, dynamic modules system

The OSGi platform solves the dynamic loading, versioning, and lifecycle management issues for Java-based services and also provides services to develop an ecosystem around it.

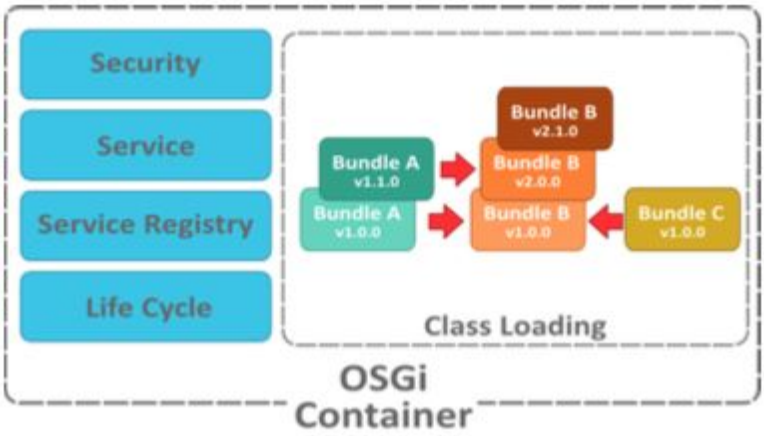
The OSGi ecosystem platform provides various service interfaces that can be implemented by vendors, depending on the nature of their applications. The OSGi framework may provide a permission administration service, conditional permission administration service, a package administration service, and URL handler service. The OSGi console offers various system services such as a log service, and administrative services for managing configuration, event administration, users, devices, and applications.

Service oriented, component-based environment

OSGi also provides a programming model based on SOA (Service Oriented Architecture). It's important to note that OSGi natively provides service\oriented programming in a dynamic environment. Unlike the RMI service registry, the OSGi service registry is native to the container - the services are automatically registered during the load without having to register them programmatically. A publisher can register a service using the *registerService* method in its BundleContext. Similarly, a service

requester can discover a service using the *getServiceReference* method from its *BundleContext*.

The service layer adds a dynamic behavior to the OSGi platform in which bundles can come and go. The heart of this layer is a service registry in which services are registered and discovered. The framework handles automatic registration and deregistration and triggers lifecycle events.



OSGi further provides a declarative model to express service registrations and dependencies in an XML declaration. This declarative framework supports lazy(delayed) loading of resources by loading them only when they're actually needed.

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 that define what a bundle provides Export-Package - and what a bundle requires to be operative - Import-Package and Require-Bundle.

Standardized software life-cycle management

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.

A Life Cycle layer adds bundles that can be dynamically installed, started, stopped, updated and uninstalled. Bundles rely on the module layer for class loading but add an API to manage the modules in runtime. The lifecycle layer introduces dynamics that are normally not part of an application. Extensive dependency mechanisms are used to assure the correct operation of the environment. Lifecycle operations are fully protected with the security architecture.

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.

Components hide their implementations from other components while communicating through services, which are objects specifically shared between components.

Application emerges from assembling multiple reusable modules that have no previous knowledge of each other.

Bundles can be loaded and installed during normal operations.

JCR (Jackrabbit)

It's a database that looks like a file system and features good stuff like unstructured, versioning, observation...

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. The current version used by CQ 6.0 is **JCR-2.0.**

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.

Seven Rules

David Nuescheler is co\founder and CTO of Day Software AG, a leading provider of global content management and content infrastructure software. He also led the development of JSR-170, the Java Content Repository (JCR) Application Programming Interface (API), the technology standard for content management. He defined seven simple rules to define the purpose and use of the JCR.

Rule #1: Data First, Structure Later. Maybe

Structure is expensive and in many cases it is entirely unnecessary to explicitly declare structure to the underlying storage. There is an implicit contract about structure that your application inherently uses.

Rule #2: 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 probably that you should reconsider. Arbitrary numbers are hardly ever a "good name".

Rule #3: Workspaces are for clone(), merge() and update().

If you don't use clone(), merge() or update() methods in your application a single workspace is probably the way to go.

"Corresponding nodes" is a concept defined in the JCR spec. Essentially, it boils down to nodes that represent the same content, in different so-called workspaces. If you don’t have them, then probably you don’t need multiple workspaces.

Rule #4: Beware of Same Name Siblings.

While Same Name Siblings (SNS) have been introduced into the spec to allow compatibility with data structures that are designed for and expressed through XML and therefore are extremely valuable to JCR, SNS come with a substantial overhead and complexity for the repository.

Any path into the content repository that contains an SNS in one of its path segments becomes much less stable, if an SNS is removed or re-ordered, it has an impact on the paths of all the other SNS and their children.

For import of XML or interaction with existing XML SNS maybe necessary and useful but I have never used SNS, and never will in my "green field" data.

Rule #5: References considered harmful.

References imply referential integrity. References do not just add additional cost for the repository managing the referential integrity, but they also are costly from a content flexibility perspective. Moving files, changing structure used to imply an impact on references, is best to maintain simple links (soft links) as paths.

Rule #6: Files are Files are Files.

If a content model exposes something that even remotely smells like a file or a folder I try to use (or extend from) nt:file, nt:folder and nt:resource. If you need to store the filename and the mime-type then nt:file/nt:resource is a very good match. If you could have multiple "files" an nt:folder is a good place to store them. If you need to add meta information for your resource, let's say an "author" or a "description" property, extend nt:resource not the nt:file. I rarely extend nt:file and frequently extend nt:resource.

Rule #7: IDs are evil.

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 though. 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 life-cycle, much fewer than you might think, though. The mix:referenceable provides such a mechanism built into the repository, so there really is no need to come up with an additional means of identifying a node in a stable fashion.

Keep also in mind that items can be identified by path, and as much as "symlinks" make way more sense for most users than hard links in a UNIX file system, a path makes a sense for most applications to refer to a target node.

Features

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.

In JCR, JSR 170 supports both SQL and Xpath, but JSR 283 deprecates Xpath use. SQL usually gains better results (faster) for all repository, Xpath is better for one directional searches (e.g. all nodes in given sub path, but not named “xyz”).

> *When checking for certain date (e.g. created/modified before a given date) make sure date in the query is in ISO format.*

Key Principles

File system-like features supported by JCR include:

- **Hierarchy:** Content in a JCR repository can be addressed by path. This is useful when delivering content to the web since most websites are also organized hierarchically.
- **Semi-structured content:** JCR can store structured documents, like XML, either as opaque files (as a file system would) or as structures ingested directly into the JCR hierarchy.
- **Access Control and Locking:** JCR can restrict access to different parts of the content hierarchy based on policies or ACLs. It also supports locking of content to prevent conflicts.

Database-like features supported by JCR include:

- **Query Access:** JCR supports querying with languages such as SQL as seen above.
- **Structured Content:** JCR can enforce constraints on data structures according to schema.
- **Referential Integrity:** JCR can enforce referential integrity between content items.
- **Transactions:** Interactions with a JCR repository can be bracketed in transactions and rolled back when needed.

In addition, JCR provides the following services that content-centric applications often need, but that neither file systems nor databases typically provide:

- **Unstructured Content:** JCR can also support arbitrary dynamic data structures without schema constraints.
- **Full-text search:** JCR supports full-text search of content.
- **Sort order:** Items within the hierarchy can maintain their ordering, if desired.
- **Observation:** API clients can register listeners to react to changes made to the
- repository.
- **Versioning:** JCR supports an advanced versioning system for repository content.

CRX (Content Repository eXtreme)

The CRX is Day Software implementation of JSR 283, a java Content Repository.

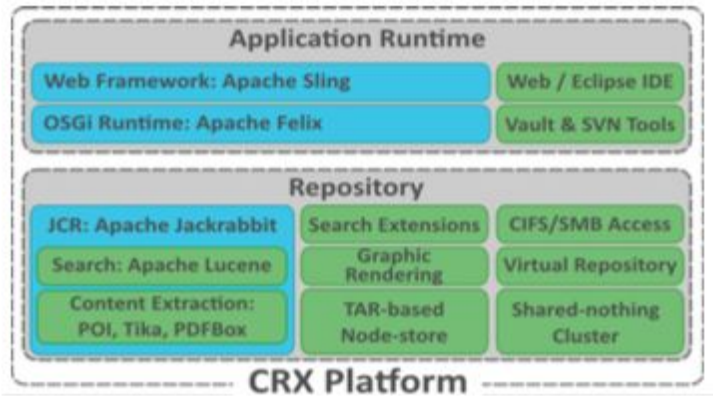
CRX is Day's proprietary implementation of the JCR.

CRX is a data storage system specifically designed for content-centric applications. 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. A JCR repository can be thought of as a "super file system". It combines characteristics of conventional file systems with those of relational databases, and adds a number of additional features that content applications often need. CRX is the sum of the implementation of the JSR 283 (JCR) and the Sling Web Framework.

Every node and property in the workspace has a name and every node has an identifier. The names allow each node and property to be addressed by path, much like in a file system.

CRX Blocks



The CRX 2.2 (on CQ 5.4) platform is form by several technologies and tools, from a top-down perspective, starting with the Application Runtime based on Sling and the OSGi. A full web development environment (integrated) as well as an Eclipse based IDE (or pure eclipse with the correct setup) integrates to CQ through tools like FileVault and SVN (or any versioning system).

FileVault and SVN (or any versioning system).

FileVault will be seen further ahead on the document but its function is to map the content of a CRX/CQ instance to your file system. The VLT tool has similar functions as source control system client (such as a Subversion (SVN) client), providing normal check-in, check-out and management operations, as well as configuration options for flexible representation of the project content.

The Repository is mainly form by the JCR and multiple extensions. The JCR itself contains references to some widely used Apache projects, like Apache Lucene, which is a high-performance, full-featured text search engine library written entirely in Java. Also rendering extension as POI, a Java APIs for manipulating various file formats based upon the Office Open XML standards (OOXML) and Microsoft's OLE 2 Compound Document format (OLE2), know as MS Excel, MS Word and MS Powerpoint. The Tika project which is a document-parsing library. The PDFBox project is an open source Java tool for working with PDF documents. This project allows creation of new PDF documents, manipulation of existing documents and the ability to extract content from documents. The JCR also works with some persistency managers, out of the box, like TarPM, but can also be linked to MySQL or others RDBM. It could be accessed in a file-system-like presentation with CIFS/SMB, and be used as a repository to hold bundles or resources.

CRX adds, consumes and manages content with these interfaces:

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

Representational State Transfer (REST)

An architectural style for distributed systems. Application data and state are represented as a set of addressable resources presenting a uniform interface that allows transfers of state (e.g. reading and updating of the resource's state).

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.

Sling Framework

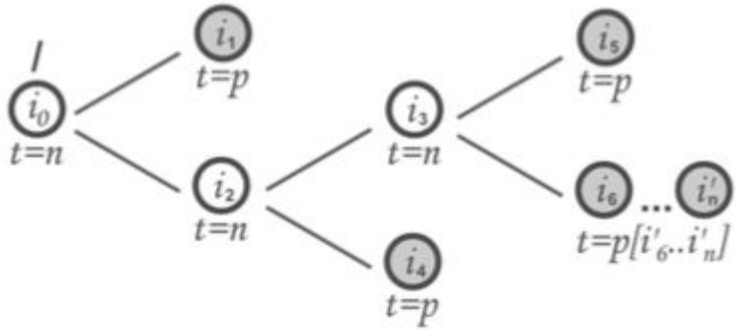
Apache Sling is a web framework. Sling applications use either scripts or Java servlets, selected based on simple name conventions, to process HTTP requests in a RESTful way.

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.

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.

Nodes & Properties

A JCR repository is composed of one or more persistent workspaces, each consisting of a directed acyclic graph of items where the edges represent the parent-child relation.



An item (i) is either a **node** (t=n) or a **property** (t=p). A node can have zero or more child items. A property cannot have child items but can hold zero or more values.

The nodes of a workspace form the structure of the stored data while the actual content is stored in the values of the properties.

Each workspace contains at least one item, the root node. The root node is the only item in the workspace without a parent node; all other items have at least one parent.

The path from the root node to an item can describe the location of such an item in the workspace graph. The path consists of the name (and index in cases of same-name siblings) of each interceding node in order from root to target item, much like a file system path. Relative paths can also be used to describe the location of one item with respect to another.

A relative JCR path begins with a segment that is neither a root segment nor an identifier segment. Its lexical form therefore begins with either a JCR name, ".." (two dots) or "." (dot).

All data stored within a JCR repository is ultimately stored as the values of properties.

Every property is of one of the following types: STRING, URI, BOOLEAN, LONG, DOUBLE, DECIMAL, BINARY, DATE, NAME, PATH, WEAKREFERENCE or REFERENCE. A property may be a single\value or a multi\value property.

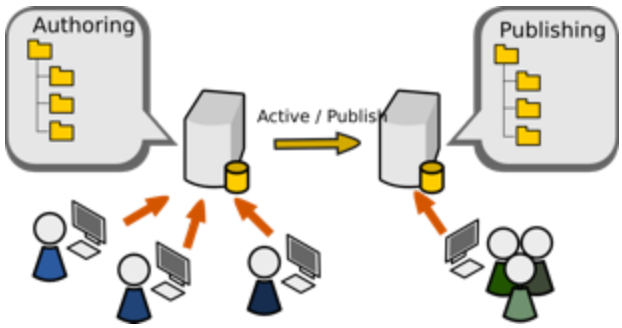
AEM Environment

"In a nutshell, AEM enables all of the ECM (Enterprise Content Management) functionality to allow non5technical authors to edit/publish/manage content based on a very wide scale of

requirements and is designed for customizations to meet most any implementation requirements while leveraging the JCR spec, CRX implementation and Sling framework."

55 Jay Kerger, System Engineer/Architect, Day Software

Generally, when you set up Adobe CQ, you need to set up an Author and a Publish instance.



Install AEM

This procedure describes how to set up a default Author instance on port 4502 of the desired host.

1) On the host file system, create a directory and name it author.
2) Copy the cq5-<version>.jar file into author/.
3) Rename cq5-<version>.jar to cq5-**author-4502**.jar.
4) Copy a valid license.properties file into author/.
5) Run it with: *java -XX:MaxPermSize=128m -Xmx384M -jar cq5-author- 4502.jar*.
6) Or if using a GUI file\system explorer, double\click the cq5-author-4502.jar file.

When the installation is completed, you are automatically presented with a new web browser window opening http://localhost:4502/.

The procedure to install should be the same, just changing the name of the cq5-<version>.jar to cq5-**publish-4503**.jar (where 4503 is the port that will be used by the publishing instance).

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.

Exercise 1

- Install AEM

Authoring & Publishing Environments

Authoring

An Author instance is the AEM installation content authors will login to and manage pages. This includes:

1) Creating.
2) Editing.
3) Deleting.
4) Moving.

In addition, it is the installation you will be developing against as you can easily observe both Author and Publish views. By default is installed on port 4502.

Publishing

A Publish instance is the AEM installation from which web site visitors will request pages.

CRX Launchpad

CRX Launchpad is an enhanced welcome screen and repository dashboard, providing access to the most frequently used repository functions. Additionally, it enables users to work with the repository content directly from desktop via WebDAV integration and to find content easily through search screen.

Console	Description
Websites	Access all the pages in a website: create, edit, and delete pages; start a workflow; activate and deactivate pages; restore pages: check external links; and access your user inbox.
Digital Asserts	Manage digital assets.
Campaigns	Manage marketing campaigns.
Community	Moderate content from social network.
Inbox	Manage workflow inbox Items.

User	Manage users and permissions.
Tools	Manage packages, designs, importers, workflow templates and scripts, replication agents and upgrades.
Tagging	Manage your tags and taxonomies.

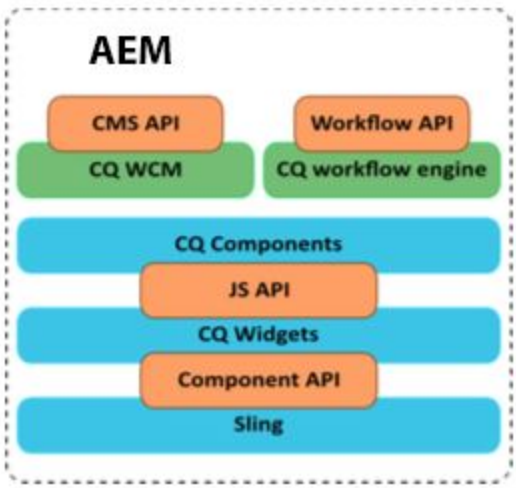
Exercise 2

- Navigate CRX Launchpad

AEM Application Layer

Components:

- **APIs and libraries**, the JCR API package, javax.jcr.* is used for the direct access and manipulation of repository content.
- **Ajax/JavaScript user interface**, CQ WCM has been developed using the ExtJS library of widgets.
- **Dialog and widget framework**
- **Workflow engine**, your content is often subject to organizational processes, including steps such as approval and sign\off by various participants. These processes can be represented as workflows, defined within CQ, then applied to the appropriate content pages or digital assets as required.
- **CMS application**



Development

Development environment

The basic development environment is based either on the web based IDE, CRXDE Lite, or in the Eclipse based IDE, CRXDE. Both are fully supported by Adobe.

The other alternative is setting an Eclipse IDE and FileVault as part of the configuration to make changes to the AEM (JCR) instance on the file system and the upload it back to the instance.

Basic project structure

Based on experience and Adobe's/Day's recommendation this is a sample structure of a simple project and all it's essential parts, not all will be explain during Day 1 or Day 2, but, they will be implemented completely during the whole training.

To start working on CQ there several terms that need to be explain. As will be seem on the table below all paths start with a "/" (slash) character or "root". This is the default AEM's "folder" (remember those are nodes type nt:folder on the JCR) structure :

/var/	Contains variable data (possibly binary) that changes during execution
/libs/	Contains the basic structure of CQ, libs, process, also used as base for tailored modifications
/etc/	This folder contains configuration information and the system's look and feel
/apps/	This is the application container, code and libraries (bundles)
/content/	The content container, all content is here and is rendered by the /apps/ code
/tmp/	Temporal files
/home/	User and groups information

An application/project is where you will store AEM 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.

A basic project on AEM must contain the next folder list (or at least an equivalent functional value):

Structure/Path	**Description**
`/apps/<application_name>`	The application container
`/apps/<application_name>/templates`	The templates container
`/apps/<application_name>/components`	The components container
`/apps/<application_name>/components/page`	The page components container
`/apps/<application_name>/components/content`	The content components container

`/apps/<application_name>/src`	The OSGi bundle source container
`/apps/<application_name>/install`	The OSGi compiled bundles container
`/etc/designs/<application_name>`	The design (CSS, JS) container
`/content/dam/<application_name>`	The DAM assets container

The two basic structure needed to create a site are “templates” and “page components”.

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.*

A template is use during the page creation to select the right page component for the content to be display. A template is node of cq:Template type.

The most important property of the template is “sling:resourceType”, this defines the path the “page component” used to render the content. This property will be created on the jcr:content node. Also needed is the “allowedPaths” property, this will define the place where the template could be used by the WCM.

Components

***Components** are modular units which realize specific functionality to resent your content on your website, are developed as self-contained units within one folder of the repository, can contain other components and components are modular, you can develop a new component on your local instance, then deploy this seamlessly to your test, then live environments.*

Components could be categorized (not in a strict sense) as “page components” and “content components”. The component node is of cq:Component type. A component usually responds to the need of a sling:resourceType to render data.

A **“page component”** is used to render the basic page layout much like the concept of a template on other platforms. Page components usually are formed by dialogs, head, header, footer and body scripts either generated for that specific component or inherited from parent component such as */libs/foundation/components/page*.

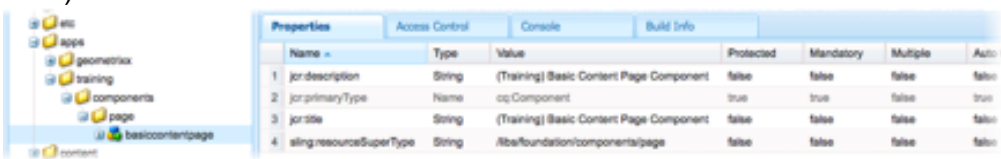
A **“content component”** is used to render the actual data, usually making use of CSS, J (JavaScript) and accessing the JCR nodes. Content components usually are formed by dialogs, CSS and JS resources and JSP (PHP, Ecma, etc.) scripts either generated for that specific component or inherited from parent component such as */libs/foundation/components/parsys*.

Exercise 3

- Create a **new template** and **page component.**
- Code could be found as a CQ5 package on the training resource folder.

Process

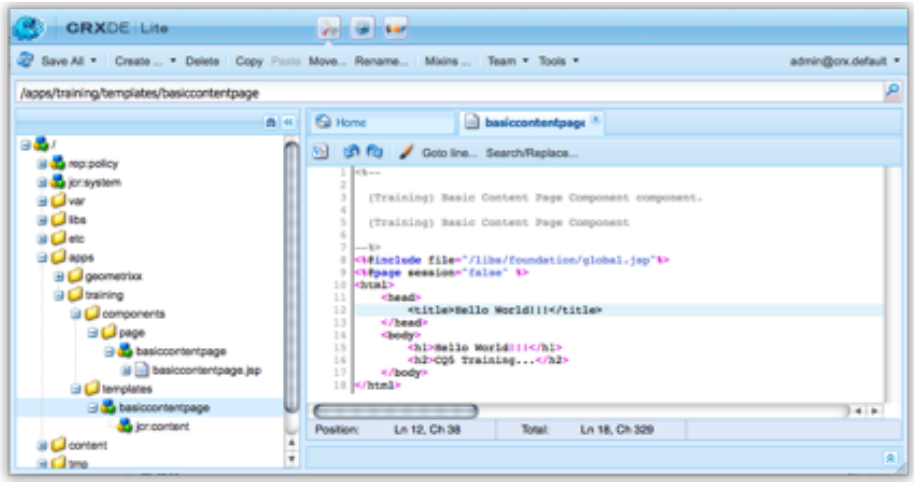
1) Using CRXDE Lite under the /apps/ folder create a new folder (using the context menu – left click) named “training”.
2) Create a folder under “training” named “components”.
3) Under the “components” folder create a new folder named “page”.
4) Under the “page” folder create a new component (using the context menu or the “create...” button on the bar), you should see a dialog like this:
    a) Label: basiccontentpage
    b) Title: (Training) Basic Content Page Component
    c) Description: (Training) Basic Content Page Component
    d) Super Type: /libs/foundation/components/page
    e) Group: “” (empty).
5) Click on “next”.
    a) Do not mark or fill anything.
6) Click on “next”.
    a) Do not change “Allowed Parents”.
7) Click on “next”.
    a) Do not change “Allowed Children”.
8) Click “OK”
    a) Click on “Save all”
    b) The structure should look like this:



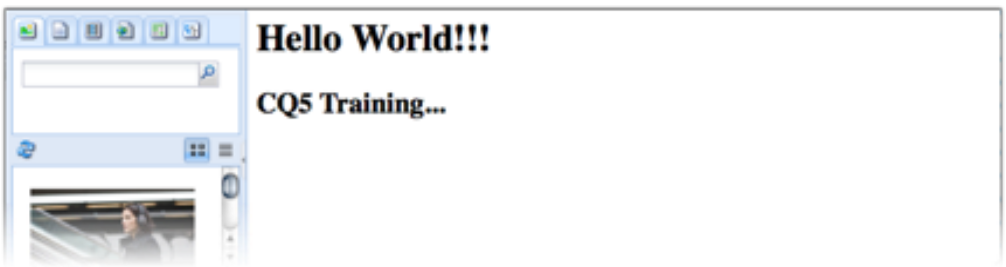
9) A new file was created under the basiccontentpage component, named after the component itself, basiccontentpage.jsp, open it (by double click or the context menu).
    a) Add this code:

```
<%--

  (Training) Basic Content Page Component component.

  (Training) Basic Content Page Component

--%>
<%@include file="/libs/foundation/global.jsp"%>
<%@page session="false" %>
    <html>
            <head>
            <title>Hello World!!!</title>
   </head>
    <body>
            <h1>Hello World!!!</h1>
            <h2>CQ5 Training...</h2>
</body>
    </html>
```

10) Click on “Save All”
11) Under the “templates” (created on point 1) folder create a new “template” (using the context menu or the “create...” button on the bar), you should see a dialog:
    a) Label: basiccontentpage
    b) Title: (Training) Basic Content Page Template
    c) Description: (Training) Basic Content Page Template
    d) Resource Type: /apps/training/components/page/basiccontentpage
    e) Raking: 1
12) Click on “next”
13) Add an “Allowed Path”
    a) Click on the button marked by the + (plus) sign.
    b) Fill the textfield with: /content(/.*)?
14) Click on “next”
    a) Do not change “Allowed Parents”.
15) Click on “next”
    a) Do not change “Allowed Children”
16) Click “OK”.
17) Click on “Save All”
    a) The structure should look like this



18) Now let's check the result
19) Back on the WCM, make sure you have websites item selected on the left side, a press the "New..." button.
20) Select "New Page"
    a) Title: Training Site
    b) Name: training\site
21) Select from the template list: (Training) Basic Content Page Template.
22) Click on create.
23) There should be a new page under "Websites" named "Training Site".
24) Open the page by double clicking or using the context menu.
    a) The page should look like this.



Walkthrough

The /apps/ folder is used to contain the applications code. The code is usually declared under the application name in this case "training", with a components\template structure. Following the structure we include a new "Page Component" named "basiccontentpage", as defined above is used to define the basic layout of our page, thus the simple structure and hard\coded content on it.

The assignment of /libs/foundation/components/page as the super type of the components is due to the need to extend certain behavior already existing in that parent component, that's a usual practice to make the most out of the basic code and more advance components of CQ.

We did not fill "Allow Parents" as we didn't want to limit the places where we could use our component, this most useful for "Content Component" to avoid insertion of the component on pages lacking on information needed for the component or that has restricted access to some functionality. The same goes to "Allow children".

Once the component is generated a new JSP script with the name is generated as well, this is just some basic behavior and still need to take into account the Sling Script Resolution process that will be seen further ahead. For now you just need to know that since it is the only and closes match to the any search for "result type" then is the one that's going to be executed.

The template is used give access to the component through the WCM, thus creating a copy of the component code with the correct references on the content folder. The "Allow Paths" property is declaring that the template can be used in any place under /content/ (the place where all content is placed), this could be narrowed to a single branch or under some specific node but that not our focus in this exercise.

On the template "resourceType" should be pointing to the "page component" previously created (not necessarily since it could be pointing to a non existing path but obviously no page will be displayed).

A new page is creating by using our template (Basic Content Page Template) precisely under the website folder showing our hard\coded message.

Day 2:

Working with AEM

Today's content:

- WCM.
- Designer.
- Apache Sling.
- Components.
- Logger.
- Debugger.

WCM

Exercise 4

- Install CRXDE
- Explore the IDE features
- Modify the text on “Basic Content Page” page component from CRXDE. Change message to “This is a AEM page component modified on CRXDE”.

Process

Once CRXDE has been installed browse the tree structure and open the .jsp (script) file under: /apps/training/components/page/basiccontentpage/basiccontentpage.jsp.

Replace the text on the script to something like this:

```
<%--

  (Training) Basic Content Page Component component.

  (Training) Basic Content Page Component

--%>
<%@include file="/libs/foundation/global.jsp"%>
<%@page session="false" %>
    <html>
            <head>
            <title>Hello World!!!</title>
   </head>
    <body>
            <h1>This is a CQ5 page component modified on CRXDE</h1>
    </body>
    </html>
```

Refresh the page on your browser.

Walkthrough

CRXDE is an editor based on Eclipse, thus for some of you it might look familiar. You’ll notice that things like code completion and syntax highlighting will work better in CRXDE that in CRXDE Lite but memory consumption might be an issue. These IDE are highly coupled to AEM (CRX) making the usual work on CQ easier on some ways even thou IDE’s like Eclipse or NetBeans could be used in a seemly equivalent way but with some extra effort.

Initialization

While setting the environment is an easy enough task there still things to learn to begin coding processes.

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

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

The Day provided global.jsp declares the Sling, CQ 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 CQS 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 CQ 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 CQS 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).

Also as part of the initialization of the WCM on the authoring instance adding the init.jsp file will handle the lunch of the sidekick. The sidekick is actually a dialog used by the page to gain access to the components and process related to the page and defined by the CQ environment. In most cases the sidekick will help you to access component or page properties.

```
1| <cq:include script="/libs/wcm/core/components/init/init.jsp" />
```

You can see the code used to load the sidekick on that script.

Exercise 5

- Modify the basic content page component to display properties values

Process

1) Using CRXDE browse to the apps/training/components/page/basiccontentpage component.
2) Modify the basicontentpage.jsp file. Replace code with this:

```
<%--
   (Training) Basic Content Page Component component.
   (Training) Basic Content Page Component + properties
   --%>
<%@include file="/libs/foundation/global.jsp"%>
<%@page session="false" %>
<html>
   <head>
       <title>CQ5 Basic Content | <%= currentPage.getTitle() %></title>
       <cq:include script="/libs/wcm/core/components/init/init.jsp" />
   </head>
   <body>
       <!--
       ---------------------------------------------
       using the default objects loaded
       by the <cq:defineObjects/> tag on global.jsp
       ---------------------------------------------
       -->
        <!-- currentPage is the current CQ WCM page object, type com.day.cq.wcm.api.Page interface
       -->
       <h1><%= currentPage.getTitle() %></h1>
       <hr/>
       <h3>Page builtin properties</h3>
       <p>Page Title (getPageTitle): <%= currentPage.getPageTitle() %></p>
       <p>Page Name (getName): <%= currentPage.getName() %></p>
       <p>Page Description (getDescription): <%= currentPage.getDescription() %></p>
       <p>Page Path(getPath)<%= currentPage.getPath() %></p>
       <p>Page Depth(getDepth)<%= currentPage.getDepth() %></p>
       <br/>
       <h3>Page properties (properties object)</h3>
 <p>Page Name ("jcr:title"): <%= properties.get("jcr:title",String.class) %></p>
       <p>Page Name ("pageTitle"): <%= properties.get("pageTitle",String.class) %></p>
       <p>Page Name ("title"): <%= properties.get("title",String.class) %></p>
```

```
        <p>Page Name ("name"): <%= properties.get("name",String.class)%></p>
        <p>Page Depth("depth"): <%= properties.get("depth",String.class)%></p>
        <hr/>
        <!-- currentNode is the current JCR node object, remember a page is a node type cq:Page,
the objec type is javax.jcr.Node interface -->
        <h3>Node builtin properties</h3>
        <p>Node Name (getName): <%= currentNode.getName() %></p>
        <p>Node Path (getPath): <%= currentNode.getPath() %></p>
        <p>Node Depth (getDepth): <%= currentNode.getDepth() %></p>
        <br/>
        <h3>Node properties</h3>
        <p>Node title ("jcr:title"): <%=
        <p>Page Description ("jcr:description"): <%=
properties.get("jcr:description",String.class) %></p>
        <p>Page Description ("description"): <%= properties.get("description",String.class)%></p>
        <p>Page Path("path"): <%= properties.get("path",String.class)%></p>
        currentNode.getProperty("jcr:title").getString() %></p>
        <%--  Node title ("title"): <%= currentNode.getProperty("title").getString() %> --%>
        <hr/>
        <!-- currentStyle is the current style object of the current cell, design properties are
inherit by child nodes, the objec type is com.day.cq.wcm.api.designer.Style interface -->
        <h3>Style builtin properties</h3>
        <p>Style path (getPath): <%= currentDesign.getCssPath() %></p>
        <p>Style ID (getId): <%= currentDesign.getId() %></p>
    </body>
</html>
```

3) save the page and refresh the test page on the browser.

Walkthrough

This exercise uses the properties of the objects provided by global.jsp. First include the global.jsp file (line 8). Once the file is included you'll get access to objects like currentPage (line 12). All this objects have get methods pointing to a set of basic properties like getName or getTitle, pointing to a child property named "name" and "title" respectively. If any of these properties aren't present, meaning they haven't been initialized, the result of the get method is "null". Also added the call to the init.jsp script to display the sidekick.

When a property has not been initialized (no initial value have been set or it is created during runtime) it doesn't exist on the JCR until a value is set. Few properties are created empty or null when a new Page is created, mostly jcr namespaced properties like jcr:language (as it is inherited) and replication properties. When accessing a property directly using the property list from the page, node or design object first check for "null" values

Properties like title might have more that one representation on the Page object. Namespacing serves to prevent naming collisions among items and node types that come from different sources or application domains, that's why CQ uses "jcr:title" as its basic page title, this one the JCR knows and can depend on, you can create your own "title" property and there won't be a collision with the "jcr" namespace. When coding using the basic set of properties you should be sure to use either the correct namespace ("jcr:xxx") or to always refer to "local" properties like "name" or "title", the only issue is that they might not exists yet.

A Page node like currentPage has a list of child property (line 39) that can be access through the "getProperties()" method or the "get" method is the object used is "properties", this returns a list (Map) of the key:value properties representations (Property objects) and

these properties will affect only the Page node (cq:Page). A Node node like currentNode, refers to actual rendered content node by default, in the a cq:Page it would be pointing to the jcr:content node (right under the page). The jcr:content node has his own list of child properties, usually containing the content references of the components on the page.

Designer

By creating and assigning a designer in AEM, it enforces a consistent look and feel across the Web site. Designs are located in /etc/designs, but there are different ways to apply a design to site\page. A design can be apply through “page properties” inherit from the Page component on
/libs/foundation/components/page, the *headlibs.jsp* script uses a call to current design object *currentDesign.writeCssIncludes(pageContext)* allowing the selected design on the page properties to be include on the HTML code. There is also the application of ClientLibs, which allows us to wrap multiples CSS and JS into categories to be used when needed

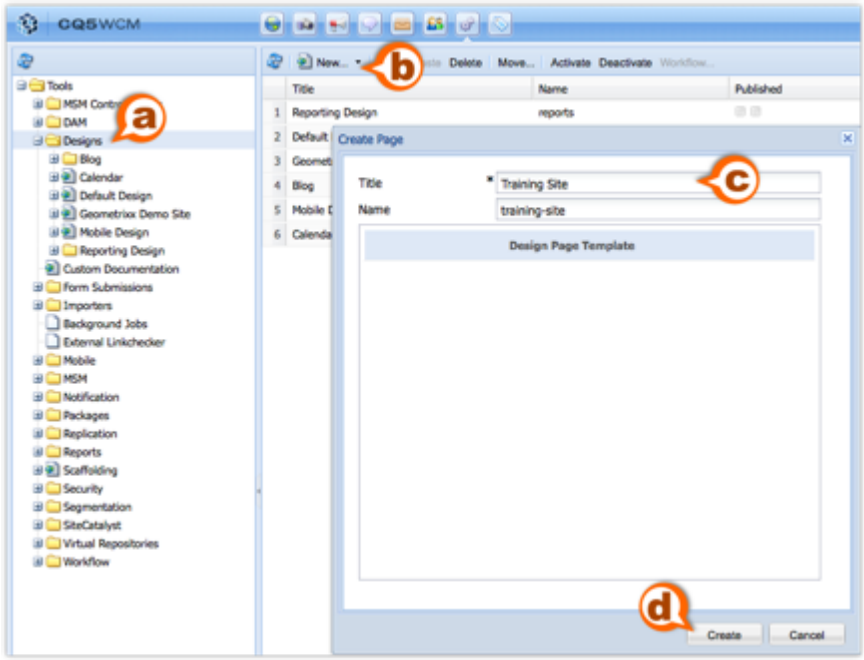
Exercise 6

- Modify the basic content page component now displaying properties values to have a design (CSS) applied.

Process

1) On the CQ Launchpad screen select the option “Tools”.

2) Select “Designs” from the folder tree (a).



3) From the top menu bar, select “New...” (b).
4) Name the new design structure as “Training Site” (c).
    a) Title: Training Site
    b) Name: training\site
5) Click on “Create” (d).
6) On CRXDE browse to /etc/designs/training\site.
7) Create a new CSS file named “static.css” and copy the following code.

```
body {
        width: 600px;
        font-family: Georgia, "TimesNewRoman", 5| Times, serif;
        background-color: white
}
table {
        border-spacing: 0px;
        padding-left: 3px;
        padding-right: 3px;
        margin: 0px;
        background-color: #0099CC;
        background-color: #00CCCC;
        border: 1px dotted gray;
        Replace the code on the Basic Component Page with this:
}
tr th {
        background-color: #0099CC;
}
tr.odd-line td {
        background-color: #00CCCC;
}
hr {
        border: 1px dotted gray;
        width: 80%;
}
```

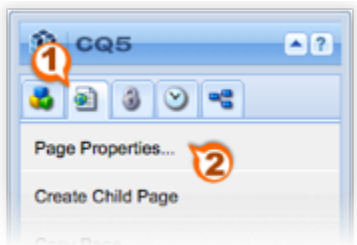
8) Replace the code on the Basic Component Page with this:

```
<%--

  (Training) Basic Content Page Component component.

  (Training) Basic Content Page Component + properties

--%>
<%@include file="/libs/foundation/global.jsp"%>
<%@page session="false" %>
  <html>
      <head>
  <title>CQ5 Basic Content | <%= currentPage.getTitle() %></title>
  <cq:include script="/libs/wcm/core/components/init/init.jsp" />
  <%currentDesign.writeCssIncludes(pageContext); %>
  </head>
  <body>
  <!--
  ---------------------------------------------
  using the default pbjects loaded
  by the <cq:defineObjects/> tag on global.jsp
  ---------------------------------------------
  -->
  <!-- currentPage is the current CQ WCM page object,
  type com.day.cq.wcm.api.Page interface -->
  <h1><%= currentPage.getTitle() %></h1>
  <hr/>
  <table>
  <tr class="table-title">
      <th colspan=2><h3>Page builtin properties</h3></th>
  </tr>
  <tr class="table-title">
      <th>Properties</th><th>Value</th>
  </tr>
  <tr class="odd-line"><td>Page Name (getPageTitle)</td><td><%= currentPage.getPageTitle() %></td></tr>
  <tr><td>Page Name (getName)</td><td><%= currentPage.getName() %></td></tr>
```

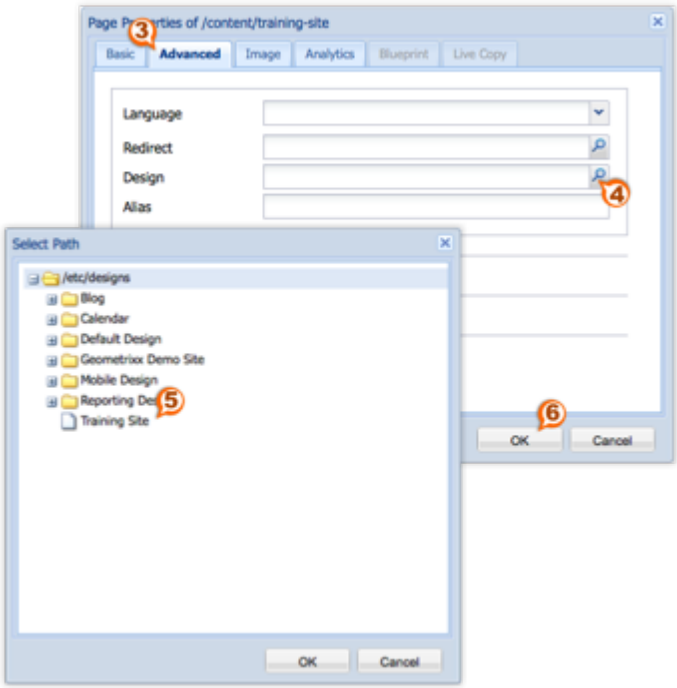
```
  <tr class="odd-line"><td>Page Description (getDescription)</td><td><%= currentPage.getDescription() %></td></tr>
  <tr><td>Page Path(getPath)</td><td><%= currentPage.getPath() %></td></tr>
  <tr class="odd-line"><td>Page Depth(getDepth)</td><td><%= currentPage.getDepth() %></td></tr>
  </table>
  <br/>
  <table>
  <tr class="table-title">
      <th colspan=2><h3>Page properties (properties object)</h3></th>
  </tr>
  <tr class="odd-line"><td>Page Name ("jcr:title")</td><td><%= properties.get("jcr:title",String.class) %></td></tr>
  <tr><td>Page Name ("pageTitle")</td><td><%= properties.get("pageTitle",String.class) %></td></tr>
  <tr class="odd-line"><td>Page Name ("title")</td><td><%= properties.get("title",String.class) %></td></tr>
  <tr><td>Page Name ("name")</td><td><%= properties.get("name",String.class) %></td></tr>
  <tr class="odd-line"><td>Page Description ("jcr:description")</td><td><%=  properties.get("jcr:description",String.class)
%></td></tr>
  <tr><td>Page Description ("description")</td><td><%= properties.get("description",String.class) %></td></tr>
  <tr class="odd-line"><td>Page Path("path")</td><td><%= properties.get("path",String.class) %></td></tr>
  <tr><td>Page Depth("depth")</td><td><%= properties.get("depth",String.class) %></td></tr>
  </table>
  <hr/>
  <!-- currentNode is the current JCR node object, remember
  a page is a node type cq:Page, the objec type is
  javax.jcr.Node interface -->
  <table>
  <tr>
  <tr class="odd-line"><td>Page Description (getDescription)</td><td><%= currentPage.getDescription() %></td></tr>
  <tr><td>Page Path(getPath)</td><td><%= currentPage.getPath() %></td></tr>
  <tr class="odd-line"><td>Page Depth(getDepth)</td><td><%= currentPage.getDepth() %></td></tr>
  </table>
```

```
40|   <br/>
41|   <table>
42|   <tr class="table-title">
43|         <th colspan=2><h3>Page properties (properties object)</h3></th>
44|   </tr>
45|   <tr class="odd-line"><td>Page Name ("jcr:title")</td><td><%= properties.get("jcr:title",String.class) %></td></tr>
46|   <tr><td>Page Name ("pageTitle")</td><td><%= properties.get("pageTitle",String.class) %></td></tr>
47|   <tr class="odd-line"><td>Page Name ("title")</td><td><%= properties.get("title",String.class) %></td></tr>
48|   <tr><td>Page Name ("name")</td><td><%= properties.get("name",String.class) %></td></tr>
49|   <tr class="odd-line"><td>Page Description ("jcr:description")</td><td><%=  properties.get("jcr:description",String.class)
%></td></tr>
50|   <tr><td>Page Description ("description")</td><td><%= properties.get("description",String.class) %></td></tr>
51|   <tr class="odd-line"><td>Page Path("path")</td><td><%= properties.get("path",String.class) %></td></tr>
52|   <tr><td>Page Depth("depth")</td><td><%= properties.get("depth",String.class) %></td></tr>
53|   </table>
54|   <hr/>
55|   <!-- currentNode is the current JCR node object, remember 56|    a page is a node type cq:Page, the objec type is
57|   javax.jcr.Node interface -->
58|   <table>
59|   <tr>
70|   </tr>
71|   <tr class="odd-line"><td>Node title ("jcr:title")</td><td><%= currentNode.getProperty("jcr:title").getString() %></td></tr>
72|   <%--  <tr><td>Node title ("title")</td><td><%= currentNode.getProperty("title").getString() %></td></tr> --%>
73|   </table>
74|   <hr/>
75|   <!-- currentStyle is the current style object of the current cell,
76|   design properties are inherit by child nodes, the objec type is
77|   com.day.cq.wcm.api.designer.Style interface -->
78|   <table>
79|   <tr>
80|         <th colspan=2><h3>Style builtin properties</h3></th>
81|   </tr>
82|   <tr class="odd-line"><td>Style path (getPath)</td><td><%= currentDesign.getCssPath() %></td></tr>
83|   <tr><td>Style ID (getId)</td><td><%= currentDesign.getId() %></td></tr>
84|   </table>
85|         </body>
86|   </html>
```

9) Save the changes and move to the WCM Websites view.
10) On the Training Site page select the “page” tab on the top of the sidekick.
11) Click on “Page Properties”.



12) Select the “Advanced tab”.



13) Browse to the “Training Site” node and click “OK”.
14) Again click “OK” on the “Advanced tab”.
15) Refresh the page and it should have some formatting now.

Walkthrough

The WCM “Tools” view give us access to Designs directory (it is located on */etc/designs/*) and help us to create a new node type cq:Page that works as entry point to the design structure. The design other than having a cq:Page node as parent behaves much like a directory. Under the “training\site” folder there should be file named static.css. By default static.css is loaded as the style file. The code **currentDesign.writeCssIncludes(pageContext);** loads first whatever is under the “Training Site” location, such as static.css and a file named *training-site.css* (just like the design structure name) under */etc/designs/*. The generated code is like this:

```
1| <link href="/etc/designs/training-site/static.css" rel="stylesheet" type="text/css">
2| <link href="/etc/designs/training-site.css" rel="stylesheet" type="text/css">
```

Clientlibs

The <cq:includeClientLib> tag Includes a CQ 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 (just as adding libraries would need normally). This tag is a very convenience wrapper around the *com.day.cq.widget.HtmlLibraryManager* service interface. This service manages the client

side includes like JS and CSS to manage which files should be included in the page as well as ensuring duplicate files are not sent.

It 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.
- **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.
- **js**, a list of comma-separated client lib categories. This will include all JavaScript libraries for the given categories.
- css, a list of comma-separated client lib categories. This will include all CSS libraries for the given categories.
- **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 can be used as follows in a JSP:

```
1| <%-- all: js + theme (theme-js + css) --%>
2| <cq:includeClientLib categories="cq.wcm.edit" />
3|
4| <%-- only js libs --%>
5| <cq:includeClientLib js="cq.collab.calendar, cq.security" />
6|
7| <%-- theme only (theme-js + css) --%>
8| <cq:includeClientLib theme="cq.collab.calendar, cq.security" />
9|
10|    <%-- css only --%>
11|    <cq:includeClientLib css="cq.collab.calendar, cq.security" />
```

*Adding **?debugClientLibs=true** argument in the path, it will include individual JS / CSS files again on the fly, which helps during debugging in Firebug. And you can see a list of the different clientlibs of your instance through the following URL, which again helps debugging: http://localhost:4502/libs/cq/ui/content/dumplibs.html*

Exercise 7

- Add a clientLib to the Basic Content Page component to overwrite the color of the “pair” lines.
- Add a JS reference to JQuery and add an effect to the table titles.

Process

1) Under the Basic Page Content component create a new node:
    a) Name: clientlibs

b) Type: cq:ClientLibraryFolder

2) Add property to the new folder.
   a) Name: categories
   b) Type: String[]
   c) Value: traininglib
3) Create a new folder named *"css"*.
4) Under the "css" folder create a new file named *"overwrite.css"*.
5) Add the following code to the file:

```
1| tr td{
2|         background-color: #00CCFF;
3| }
```

6) Create a new file under clientlibs named *"css.txt"*.
7) The file content is:

```
1| #base=css
2| overwrite.css
```

8) Create a new folder under clientlibs named *"js"*.
9) Copy by dragging (on CRXDE) the jquery sources into the *"js"* directory.
10) Create a new file under clientlibs named *"js.txt".*
11) The file content is:

```
3| #base=js
4| jquery-ui-1.8.19.custom.min.js
```

12) Add the following code to the Basic Content Page component before closing the <head> tag, from line 16 to 23.

```
1| <%--
2|
3|    (Training) Basic Content Page Component component.
4|
5|    (Training) Basic Content Page Component + properties
6|
7| --%>
8| <%@include file="/libs/foundation/global.jsp"%>
9| <%@page session="false" %>
10|    <html>
11|         <head>
12|    <title>CQ5 Basic Content | <%= currentPage.getTitle() %></title>
13|    <cq:include script="/libs/wcm/core/components/init/init.jsp" />
14|    <%currentDesign.writeCssIncludes(pageContext); %>
15|    <cq:includeClientLib categories="traininglib"/>
16|    <script type="text/javascript">
17|    // This function is called on page ready. It bootstraps the div tag with the jQuery UI dialog class.
18| jQuery(function ($) {
19|     $("table").click(function () {
20|         $(this).hide("explode",{ pieces: 48 }, 1000);
21|     }})}
22|
23|         </script>
24|  </head>
25| <body>
26| ...
```

13) Refresh the training page. By clicking on the tables the jQuery effect should be shown.

Walkthrough

By adding the cq:includeClientLib tag and specifying the category (traininglib) the source files (CSS and JS) will be included on the page with a tag similar to this one:

```
1| <script type="text/javascript"
2| src="/apps/training/components/page/basiccontentpage/clientlibs.js"></script>
3| <link rel="stylesheet" href="/apps/training/components/page/basiccontentpage/clientlibs.css"
4| type="text/css">
```

The clientlibs.js will contain all the JavaScript code defined on the “js.txt” file. The first line of the js.txt file sets the “base” directory of the source files, meaning that the location for those JS files will be under the “js” directory. The same effect happens on the clientlibs.css, all the CSS references will be included in this file. After that the “explode” effect is part of jQuery.

Apache Sling

“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.”

Content Orientation

Sling is content\centric. This means that processing is focused on the content as each (HTTP) request is mapped onto content in the form of a JCR resource (a repository node):

- The first target is the resource (JCR node) holding the content.
- Secondly, the representation, or script, is located from the resource properties in combination with certain parts of the request (e.g. selectors and/or the extension)

The context drives the way content is rendered, but context is not limited to device type. The content is rendered to HTML (or any other format) when needed based on metadata of the content (binary, plain text, tags, etc.) and the context (devices, emulators, media) the content is being viewed/displayed.

Restful

REST stands for "Representational State Transfer" and is a style of software architectures for distributed hypermedia systems; Dr. Roy Fielding, Day Chief Scientist, introduced it in 2000. The best\known example of a system running according to REST principles is the World Wide Web. REST is also a popular choice for implementing systems based on web services.

To achieve a restful architecture this principles must be follow:

Client–server

A uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. This is accomplished on AEM by separating the code implementation on the presentation layer and the business logic on bundles.

Stateless

The client–server communication is further constrained by no client context being stored on the server between requests. Each request from any client contains all of the information necessary to service the request, and any session state is held in the client. The server can be stateful; this constraint merely requires that server\side state be addressable by URL as a resource. Sling provides the mechanisms to accomplish this on AEM.

Cacheable

As on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients reusing stale or inappropriate data in response to further requests. On AEM and other system this could be achieve by JSP directives and server configuration.

Layered system

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.

Code on demand (optional)

Servers are able temporarily to extend or customize the functionality of a client by the transfer of executable code. Examples of this may include compiled components such as Java applets and client\side scripts such as JavaScript.

Uniform interface

The uniform interface between clients and servers, discussed below, simplifies and decouples the architecture, which enables each part to evolve independently. This makes access to resources simpler. Sling represents this unified form of access to the resources.

URL Decomposition

During the Resource Resolution step, the client request URL is decomposed into the following parts:

- **Content Path** \ The longest substring of the request URL such that the resource path is either the complete request URL or the next character in the request URL after the resource path is a dot (.).
- **Selectors** \ If the first character in the request URL after the resource path is a dot, the string after the dot up to but not including the last dot before the next slash character or the end of the request URL. If the resource path spans the complete request URL no selectors exist. If only one dot follows the resource path before the end of the request URL or the next slash, also no selectors exist.
- **Extension** \ The string after the last dot after the resource path in the request URL but before the end of the request URL or the next slash after the resource path in the request URL.
- **Suffix Path** \ If the request URL contains a slash character after the resource path and optional selectors and extension, the path starting with the slash up to the end of the request URL is the suffix path. Otherwise, the suffix path is empty. Note, that after the resource path at least a dot must be in the URL to let Sling detect the resource path.

Protocol	host	content path	selector	extension		suffix		parameters
http://	host/	content/spy	.print.a4	.html	/	a/b	?	x=12

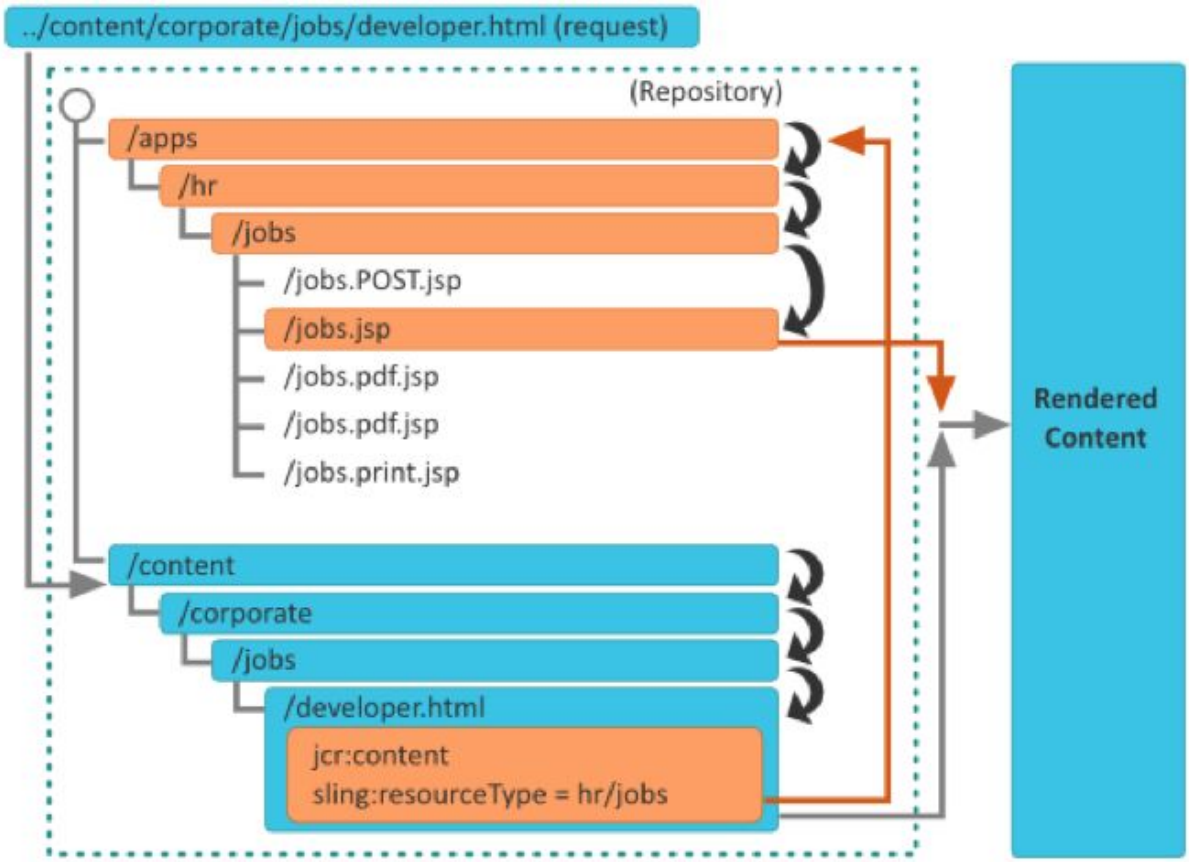
Script Resolution

When a request is processed by Sling, one step is to resolve the request URL into a Content object. This works by checking the request URL for the longest match with an existing JCR repository node. The path of this node is then used to load the Content object through the ContentManager.load(String) method. If no mapping exists for the given node, an exception is thrown and the request fails. The type of content is not the first consideration. The first consideration is whether the URL resolves to a content object, a resource, for which a script can be found to render the content (sling:resourceType).

A path can be absolute (starting from “/” – “root”) or relative (on the same folder or from “.” – “dot”). Sling scripts must be stored on /apps/ or /libs/. If the script is not found on /apps/ (primary location) then the process look into /libs/ with the same tree structure.
When the appropriate resource (content node) is located, the sling resource type is extracted. This is a path, which locates the script to be used for rendering the content. Must me clarified that when the Method (GET, POST) is required, it will be specified in

upper\case as according to the HTTP specification e.g. jobs.POST.jsp, even when the method GET/HEAD is still used by default, so the extension could be a simple .html.

An extended example following:



1. First, Sling checks whether a node exists at the location specified in the request; e.g. "../content/corporate/jobs/developer.html".
    a. If no node is found, the extension is dropped and the search repeated; e.g. "../content/corporate/jobs/developer"
    b. If no node is found then Sling will return the http code 404 (Not Found).
2. When the Method (GET, POST) is required, it will be specified in uppercase as according to the HTTP specification e.g. jobs.POST.jsp
3. If the sling:resourceType is hr/jobs then:
    a. GET/HEAD requests, and URLs ending in .html (default request types, default format). The script will be /apps/hr/jobs/jobs.jsp; the last section of the sling:resourceType forms the file name.
    b. b. POST requests (all request types excluding GET/HEAD, the method name must be uppercase). POST will be used in the script name. The script would be /apps/hr/jobs/POST.jsp.
    c. URLs in other formats, not ending with .html. For example, "../content/corporate/jobs/developer.pdf". The script would be /apps/hr/jobs/jobs.pdf.jsp; the suffix is added to the script name.
    d. URLs with selectors. Selectors can be used to display the same content in an alternative format. For example a printer friendly version, an rss feed or a summary. If we look at a printer friendly version where the selector could be

print; as in "../content/corporate/jobs/developer.print.html". The script will be /apps/hr/jobs/jobs.print.jsp; the selector is added to the script name.

e. If no *sling:resourceType* has been defined then the content path will be used to search for an appropriate script (if the path based ResourceTypeProvider is active). For example, the script for ../content/corporate/jobs/developer.html would generate a search in /apps/content/corporate/jobs/. The primary node type will be used.

f. If **no script is found at all** then the default script will be used. The default rendition is currently supported as plain text (.txt), HTML (.html) and JSON (.json), all of which will list the node's properties (suitably formatted). The default rendition for the extension .res, or requests without a request extension, is to spool the resource (where possible). For http error handling (codes 403 or 404) Sling will look for a script at either:

   i. The location /apps/sling/servlet/errorhandler for customized scripts

   ii. The location of the standard scripts **/libs/sling/servlet/errorhandler/403.jsp**, or **404.jsp** respectively.

You can also use the standard Sling Script Resolution Sheet to guide you through the process.

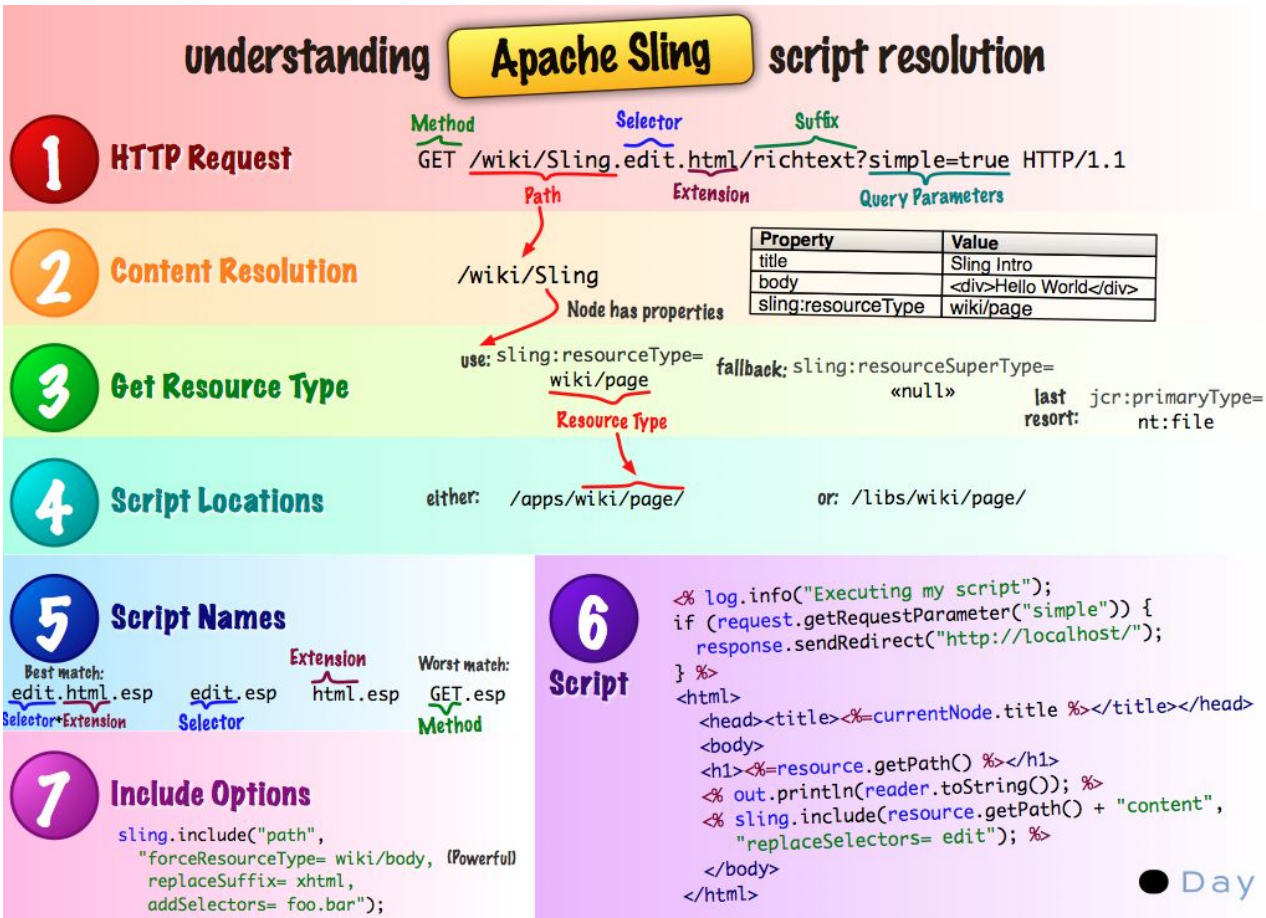


image source: https://docs.adobe.com/docs/en/cq/5-6-1/developing/the_basics.html

Exercise 8

- Create multiple renders for a page.

- Suppose you have a normal (rich html), and a tight width representation of the Basic Content Page.

Process

1) Under /apps/training/components/page/basiccontentpage/ create a new JSP file named plain.jsp.
2) Copy the following code into plain.jsp:

```
<%--

  (Training) Basic Content Page Component (plain) component.

  (Training) Basic Content Page Component + properties (plain)

--%>
<%@include file="/libs/foundation/global.jsp"%>
<%@page session="false" %>
    <html>
        <head>
    <title>CQ5 Basic Content | <%= currentPage.getTitle() %> (Plain)</title>
    <cq:include script="/libs/wcm/core/components/init/init.jsp" />
    </head>
    <body class="print">
                <!--
        ---------------------------------------------
        using the default pbjects loaded
        by the <cq:defineObjects/> tag on global.jsp
        ---------------------------------------------
        -->
        <!-- currentPage is the current CQ WCM page object,
        type com.day.cq.wcm.api.Page interface -->
        <h1><%= currentPage.getTitle() %> | (Plain)</h1>
        <hr/>
        <table>
                <tr>
                        <td colspan=2><h3>Page builtin properties</h3></th>
                </tr>
                <tr>
                        <td>Properties</td><td>Value</td>
                </tr>
                <tr><td>Page Name (getPageTitle)</td><td><%= currentPage.getPageTitle() %></td></tr>
currentPage.getPageTitle() %></td></tr>
        <tr><td>Page Name (getName)</td><td><%= currentPage.getName()
        <tr><td>Page Description (getDescription)</td><td><%= currentPage.getDescription() %></td></tr>
        <tr><td>Page Path(getPath)</td><td><%= currentPage.getPath()
        <tr><td>Page Depth(getDepth)</td><td><%= currentPage.getDepth() %></td></tr>
    </table>
</body>
</html>
```

3) Under /apps/training/components/page/basiccontentpage/ create a new JSP file named narrow.jsp
4) Copy the following code into narrow.jsp:

```
<%--
  (Training) Basic Content Page Component component.
  (Training) Basic Content Page Component + properties
--%>
<%@include file="/libs/foundation/global.jsp"%>
<%@page session="false" %>
  <html>
      <head>
  <title>CQ5 Basic Content | <%= currentPage.getTitle() %> (Narrow)</title>
  <cq:include script="/libs/wcm/core/components/init/init.jsp" />
  <%currentDesign.writeCssIncludes(pageContext); %>
  <cq:includeClientLib categories="traininglib"/>
  <script type="text/javascript">
  // This function is called on page ready. It bootstraps the div tag with the jQuery UI dialog class.
jQuery(function ($) {
      $("table").click(function () {
              $(this).hide("explode",{ pieces: 48 }, 1000);
}))
</script>
</head>
<body class="narrow">
<!--
-----------------------------------------------
using the default pbjects loaded
by the <cq:defineObjects/> tag on global.jsp
-----------------------------------------------
-->
<!-- currentPage is the current CQ WCM page object, type com.day.cq.wcm.api.Page interface -->
<h1><%= currentPage.getTitle() %> | (Narrow)</h1>
<hr/>
<hr/>
```

```
      <table>
            <tr>
            <td colspan=2><h3>Page builtin properties</h3></th>
            </tr>
            <tr>
            <td>Properties</td><td>Value</td>
            </tr>
            <tr><td>Page Name (getPageTitle)</td><td><%= currentPage.getPageTitle() %></td></tr>
            <tr><td>Page Name (getName)</td><td><%= currentPage.getName() %></td></tr>
            <tr><td>Page Description (getDescription)</td><td><%= currentPage.getDescription() %></td></tr>
            <tr><td>Page Path(getPath)</td><td><%= currentPage.getPath() %></td></tr>
            <tr><td>Page Depth(getDepth)</td><td><%= currentPage.getDepth() %></td></tr>
      </table>
  </body>
</html>
```

Sling structure and JCR/WCM

As mentioned previously, everything in JCR is a resource, accessed via Sling. So, all the content stored for the pages are resources saved in nodes inside JCR, with its own properties that have information that will be displayed to the final user.

As CQ5 is a Web Content Management (WCM) system, the content editors can introduce all kind of information and assets by the components. These components will be part of a specific page, and the pages are part of the sites. This hierarchy is how JCR stores the content, in a tree structure, with nodes, children nodes and properties.

The next concepts are very important to have in mind:

- **Node**: is the basic unit inside the JCR, which contains a set of elements where the content is saved. Each node has a type associated, which determines its behavior. Also have other child nodes and properties.
- **Property**: is where the content is saved. Is a pair of (key, value), accessed like a hash map. The value can have one or multiple “items” (structured in an array). Each property has also a type, which determines what kind of value it has.
- **Page**: is the web site content container, which has a child node named jcr:content, that contains the information to show to the user. It’s an instance of a template.

Components

A software component is a system element offering a predefined service or event, and able to communicate with other components.

Within CQ a component is often used to render the content of a resource. When the resource is a page, the component rendering it is called a Top\Level Component or a Page Component. However, a component does not have to render content, nor be linked to a specific resource; for example, a navigation component will display information about multiple resources.

Components are modular units which realize specific functionality to present your content on your website, due it design they are re\usable, self\contained units within one folder of the repository; they have no hidden configuration files and can contain other components.

A CQ component can run any where within any CQ system thanks to its standardized user interface, are easy to use and author\oriented components use widgets.

As components are modular, you can develop a new component on your local instance, then deploy this seamlessly to your test, then live environments.

Each CQ component:

- Is a resourcetype.
- Is a collection of scripts that completely realize a specific function.
- A component can function in "isolation"; this means either within CQ or a portal.

Some of the components included with CQ are:

- Paragraph system.
- Header.
- Image, with accompanying text.
- Toobar

CQ WCM has the page component (type cq:Page); this is a particular type of resource, which is important for content management.

Exercise 9

- Copy and split the Basic Component Page into “scripts”, the new component name will be “basicpage”, and it should inherit from foundation’s page component.
    - Create a head scripts. Loading libraries and styles. Add the foundation title component to the script.
    - Create a header scripts. Using the header div of the existing component and adding leaving div placeholder for a title component.
    - Create a content scripts. Removing the information displayed (tables) and leaving a div placeholder for a content component.
    - Create a footer scripts.

Process

1) Create a new component under /apps/training/components/page/ named “basicpage”, remember to set the sling:resourceSuperType to “/libs/foundation/components/page” and componentGroup as “.hidden”.
2) On basicpage.jsp write the reference to the other scripts on the component, first set the HTML structure, and then add the head script as well as the body script. Like this:

```
1| <%--
2|
3| (Training) Basicpage component.
4| (Training) Basicpage page component.
5|
6| Is using the base for all "page" components. Basically includes the "head" and the "body" scripts.
7|
8| ==============================================================================
9|
10| --%><%@page session="false"
11|     contentType="text/html; charset=utf-8" import="com.day.cq.commons.Doctype,
12|     com.day.cq.wcm.api.WCMMode, com.day.cq.wcm.foundation.ELEvaluator" %><%
13| %><%@taglib prefix="cq" uri="http://www.day.com/taglibs/cq/1.0" %><%
14| %>
15| <!--  no need to include all taglibs just CQLib -->
16| <cq:defineObjects/>
17| <html>
18|     <cq:include script="head.jsp"/>
19|     <cq:include script="body.jsp"/>
20| </html>
```

3) Create a new file under the same component named “head.jsp” where all the head tags will be placed.

```
1| <%--
2|
3|   Default head script.
4|
5|   Draws the HTML head with some default content:
6|   - includes the WCML init script
7|   - includes the head libs script
8|   - includes the favicons
9|   - sets the HTML title
10|   - sets some meta data
11|
12|   ==============================================================================
13|
```

```
14|   --%><%@include file="/libs/foundation/global.jsp" %><%
15|   %><%@ page import="com.day.cq.commons.Doctype,
16|         org.apache.commons.lang.StringEscapeUtils" %><%
17|   String xs = Doctype.isXHTML(request) ? "/" : "";
18|   String favIcon = currentDesign.getPath() + "/favicon.ico";
19|   //if we had a fevicon or desing here it world be referenced
20|   if (resourceResolver.getResource(favIcon) == null) {
21|         favIcon = null;
22|   }
23|   %><head>
24|   <meta http-equiv="content-type" content="text/html; charset=UTF-8"<%=xs%>>
25|   <meta http-equiv="keywords" content="<%= StringEscapeUtils.escapeHtml(WCMUtils.getKeywords(currentPage, false))
%>"<%=xs%>>
26|   <meta http-equiv="description" content="<%= StringEscapeUtils.escapeHtml(properties.get("jcr:description", ""))
%>"<%=xs%>>
27|   <cq:include script="headlibs.jsp"/>
28|   <cq:includeClientLib categories="cq.foundation-main"/>
29|   <!-- includes our css -->
30|   <%currentDesign.writeCssIncludes(pageContext); %>
31|   <!-- sidekick -->
32|   <cq:include script="/libs/wcm/core/components/init/init.jsp"/>
33|   <% if (favIcon != null) { %>
34|   <link rel="icon" type="image/vnd.microsoft.icon" href="<%= favIcon %>"<%=xs%>>
35|   <link rel="shortcut icon" type="image/vnd.microsoft.icon" href="<%= favIcon %>"<%=xs%>>
36|   <% } %>
37|   <title><%= currentPage.getTitle() == null ? currentPage.getName() : currentPage.getTitle() %> | Training</title>
38|   </head>
```

4) Create a new file named “body.jsp” under the same component and place all “content” structure under this file, either separated into scripts or a single file, we’ll be following the multiple script approach.

```
1| <%@include file="/libs/foundation/global.jsp" %>
2| <%@page session="false"%>
3| <body>
4|         <cq:include script="header.jsp"/>
5|  <cq:include script="content.jsp"/>
6|         <cq:include script="footer.jsp"/>
7| </body>
```

5) Create “header.jsp” to hold the upper section of the page.

```
1| <%--
2|
3|   Simple header.
4|
5|   ==========================================================
6|
7| --%><%@include file="/libs/foundation/global.jsp" %>
8| <div class="header-container">
9|   <div id="logo">
10|         <p>Logo</p>
11|   </div>
12|   <h1>title</h1>
13|   </div>
```

6) Now create a “content.jsp” that will hold the entire specific content (diverse) from the page. So far this will only be a placeholder for future references.

```
1| <div>
2| <p>Content placeholder</p>
3| </div>
```

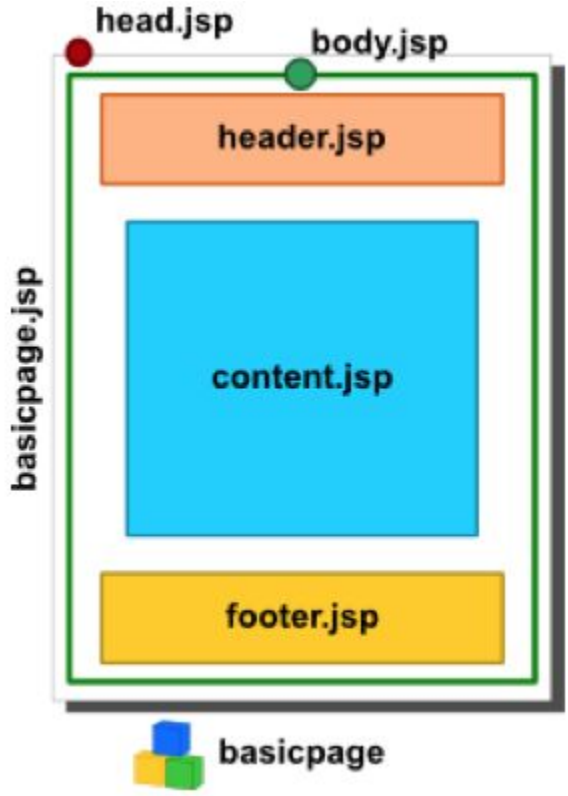
7) And finally a “footer.jsp” script that will work just as the header but for the lower section of the page. Notice the order of the scripts so far.

```
1| <%--
2|
3|  Simple footer.
4|
5| ============================================================================
6|
7| --%><%@include file="/libs/foundation/global.jsp" %>
8| <div class="footer-container">
9|         <div class="first-footer-link">Home</div>
10|        <div class="footer-link">Disclaimer</div>
11|        <div class="footer-link">Contact Us</div>
12|        </div>
```

8) Create a new template under /apps/training/templates/ named “basicpage” pointing the sling:sourceType property within the jcr:content node o /apps/training/components/page/basicpage.
9) Use the late template to create a new page under /content/training\site/ named training\basicpage (Training Basic Page).

Walkthrough

The new page component was created using a sample (not rigid) methodology where the component “parts”, defined as sections, could work as atomic scripts with a specific purpose.



The “*basicpage*” component is composed of a “principal” script (this will be the name used until we get further into the sling script resolution process), sections and subsections.

The *basicpage.jsp* script function is to put together the header section (usually represented as <head></head>) and the body section (represented as <body></body>) by using the sling:include tag discussed on the previous exercise.

By separating the body on subsections we can assure that header and footer can be reused by later components or be overwriting if needed, this will be cover on the hierarchy section.

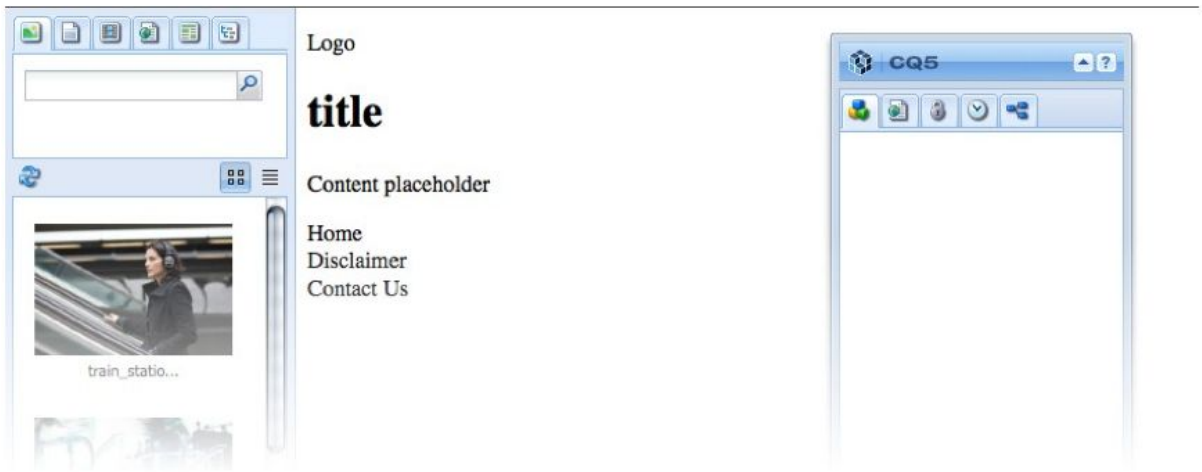
The *head.jsp* script contains html meta\tag information and initializes the AEM working environment. Header script contains a logo placeholder for future use as well as a header title.

The content.jsp script is a placeholder for future content, this will define the basic layout even when is probable that this basicpage component is going to be used as a reference (much like an abstract class on java) for future page components.

For the sake of testing will be creating pages using the basicpage component during this exercise but will NOT encourage such practice for future exercises or development practices.

The footer script contains generic information like disclaimers and links to static

places. The page should look like this:



Hierarchy

Components within CQ 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; for example a text component will inherit various attributes from the standard component.

- Scripts (resolved by Sling).
- Dialogs.
- Descriptions (including thumbnail images, icons, etc.).

Container Hierarchy

This is used to populate configuration settings to the child component and is most commonly used in a parsys scenario.

For example, configuration settings for the edit bar buttons, control set layout (editbars, rollover), dialog layout (inline, floating) can be defined on the parent component and propagated to the child components.

Configuration settings (related to edit functionality) in cq:editConfig and cq:childEditConfig are propagated.

Exercise 10

- Create a new page component that inherits from basicpage, named “landingpage”.
    - Create a new content scripts that overwrites the behavior of the content script from basicpage.

Process

1) Create a new component under /apps/training/components/page/ named “landingpage”, be sure to add the sling:resourceSuperType property as “/apps/training/components/page/basicpage”.
2) Remove the script named “landingpage.jsp” and create a new script named “content.jsp”.
3) Add the following code to the script:

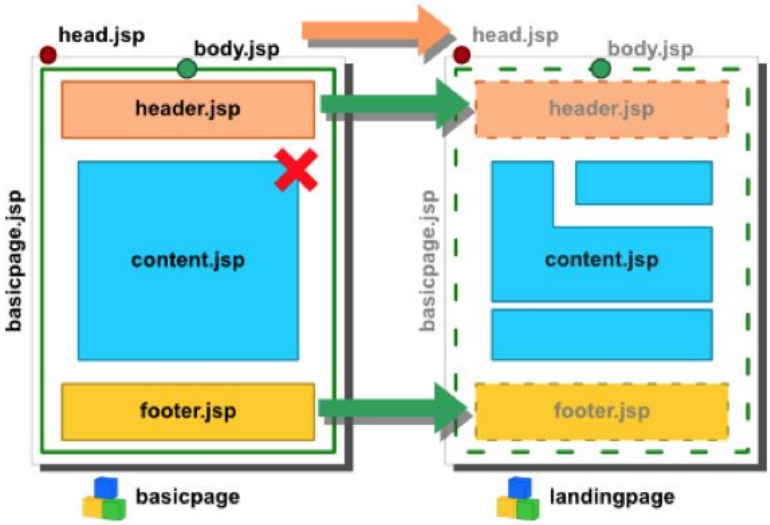
```
1| <div>
2| <p>This is the content placeholder for a landing page<p>
3| </div>
```

4) Create a new template under /apps/training/templates/ named “landingpage”, pointing the sling:sourceType property within the jcr:content node to /apps/training/components/page/landingpage.
5) Create a new page using the template from the step (4) named “training\landingpage” (Training Landing Page), next to training\basicpage from the last exercise.

Walkthrough

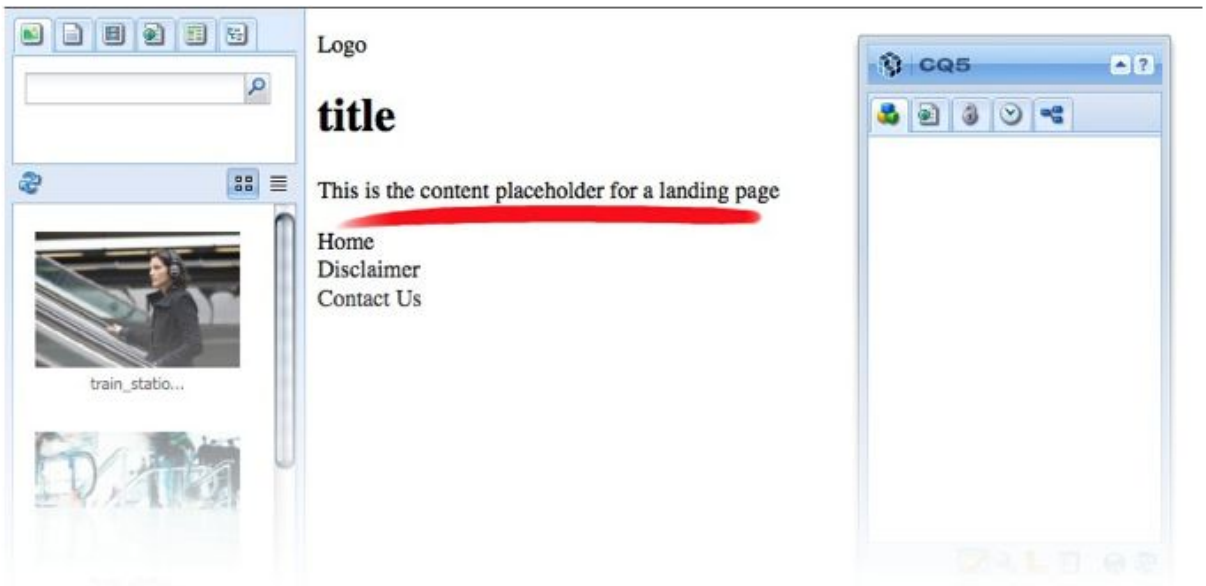
Based on the previous exercise a structure was set to enable inheritance between components. The “*basicpage*” component will work as our parent component and inherit its structure (scripts and dialogs) to “*landingpage*”, this will be effective thanks to pointing *sling:resourceSuperType* to “*basicpage*”.

When a page using “landingpage” as *sling:resourceType* is pulled from the JCR, Sling will attempt to execute its script resolution process; once the process reaches the script selection step it will work somehow like this:



1. Look for a “hit” on the component scripts within the current component. Failing.
2. The next step would be to check the parent component for such “hit”. The “basicpage.jsp” script will get a “hit” from the process as the name of the component and start execution.
3. Then “head.jsp” will be called, since there is no “hit” for “head.jsp” on “landingpage” the one on “basicpage” would be use.
4. The same happens to “body.jsp”. Within “body.jsp” there is a call for “header.jsp”, this will resolve into using the one on “basicpage”.
5. Then the call to “content.jsp” will get a “hit” on “landingpage”, meaning that the actual script to be run is from the current component and not from its parent.
6. The footer will be inherited as well from the parent component.

Basically what we did is inherit the structure from the parent component and change just the content script to work with a new structure, as complex or simple as needed, which can be extended by the use of other scripts or components. The result would look like this:



Compare the page with the one using “basicpage”. Notice that the structure is the same, there is a logo message and title, as well as the list on the footer. The difference is noted in the content. You could try changing the structure on the header.jsp script adding different messages and reload both pages; they should still reflect the same header script rendering the changed structure on both pages.

Structure

AEM WCM components are located under **/libs/foundation/components**, these are the default (basic) components included on the CQ platform distribution, they can be referenced, copied, modified, inherit from, used as base or simply disregard.

Components for a specific application are located under **/apps/<sitename>/components** (usually), these are the current application components which are related to a site by <sitename> but can be access all across the platform.

Components on foundation can be overridden by creating a similar structure under **/apps/**. As seen on the Sling section, the resource locator service works by looking for scripts either on **/apps/** or **/lib/** (in that order).

AEM standard components are defined as cq:Component (with exceptions like cq:Page) and have the elements:

- A list of *jcr:properties;* these are variable and some maybe optional though the basic structure of a component node, its properties and sub nodes are defined by the *cq:Component* definition.
- The dialog defines the interface allowing the user to configure the component
- The dialog (of type *cq:Dialog*) and *cq:editConfig* (of type *cq:EditConfig*) must both be defined as otherwise the component will not appear
- Resources: define static elements used by the component
- Scripts are used to implement the behavior of the resulting instance of the component
- Thumbnail: image to be shown if this component is listed within the paragraph system

Exercise 11

- Create a custom title component.
    - Replace the title component on the basicpage component header script with the custom.
- Add some foundation components to the landingpage component.
    - Add a parsys component to the content script.
        - Add some text components from the author interface.
    - Add an image component to the content script.
        - Add an image from the author interface.

Process

1) Add a new folder under the structure /apps/training/components named “content”.
2) Create a new component under the new “content” folder:
    a) Name: basictitle
    b) Title: (training) Basic Title
    c) Description: (training) Basic Title Component
    d) Group: .hidden
    e) Leave the rest empty.

3) Replace with the following code the default on “basictitle.jsp”:

```
<%--

  Basic Title Component component.

  Basic Title Component

--%><%
%><%@include file="/libs/foundation/global.jsp"%><%
%><%@page session="false" %><%
    %><%
    String title = currentPage.getTitle() == null ? currentPage.getName() : currentPage.getTitle();
String subtitle = currentPage.getProperties().get("subtitle",String.class);
%>
<div class="title-container">
        <div class="page-title">
<h1><%= title %></h1>
</div>
<%
if (subtitle != null && !subtitle.equals("")){
    %>
        <div class="page-title">
                <h2><%= subtitle %></h2>
        </div>
<% }
%>
</div>
```

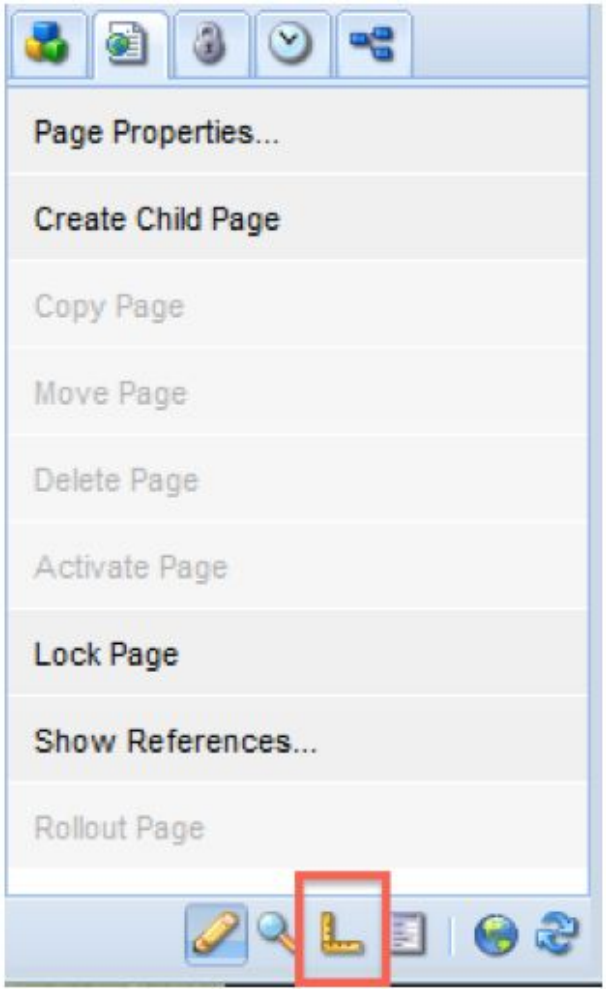
4) Modify the header script on “basicpage” to look like this:

```
<%--

  Simple header.

  ==============================================================================

--%><%@include file="/libs/foundation/global.jsp" %>
<div class="header-container">
    <div id="logo">
    <p>Logo</p>
</div>
    <cq:include path="title" resourceType="training/components/content/basictitle"/>
</div>
```

5) Modify the content script on “landingpage” to look like this:

```
<%@include file="/libs/foundation/global.jsp" %>
<div>
<cq:include path="par" resourceType="foundation/components/parsys" />
</div>
```

6) Reload the landing page on the browser.
7) Change the view mode to “design” by clicking the “ruler” icon on the bottom of the “sidekick”:



8) Click on the “edit” (2), is located on the “editbar” right over the “par” (parsys) section.
9) Select the whole “general” and “training” (3) component groups and click “ok” (4).
10) Return to the “edit mode” by expanding the “sidekick”.
11) Add a “General” **image** (5) component from the sidekick by dragging it from into the **parsys** section (6); do the same with a **text** component (7).
12) Notice that each time a component is added a new placeholder for components is appended at the end of the **parsys** (8).

Walkthrough

The new folder structure /apps/training/components/**content** will hold all components related to render content as modules, paces of code that can be reused, as opposite of the “page” structure that contains placeholders (as pages) for such components.

The very first component created is a simple “Title component”, used to display the information on the *“jcr:title”* property as an <h1> tag; and if exists display the subtitle as well. Notice that the “*componentGroup*” for this component is “.hidden”, we are not going to use this component from the sidekick, this component will be “hard\coded” on top of our landing pages, so there should always be a title line in every page. This will enforce by adding the component to the header on “**basicpage**”, remember that “header.jsp” will be

used across all the “**basicpage**” children as long as is not overwritten another header script.

On the “**landingpage**” content script a **parsys** is added to enable drag\and\drop of component from the sidekick. The **parsys** allows moving the components around, deleting them and adding new ones. The list of available components for this **parsys** will be kept under the page design structure, this will be covered further ahead.

Node types

Every node has one and only one primary node type. A primary node type defines the characteristics of the node, such as the properties and child nodes that the node is allowed to have. In addition to the primary node type, a node may also have one or more mixin types. A mixin type acts a lot like a decorator, providing extra characteristics to a node. A JCR implementation, in particular, can provide three predefined mixin types:

- mix:versionable: allows a node to support versioning.
- mix:lockable: enables locking capabilities for a node.
- mix:referenceable: provides an auto\created jcr:uuid property that gives the node a unique, referenceable identifier.

Predefined node types

The repository support the primary node type, **nt:base**. There are a number of other common node types that a repository may support:

nt:unstructured is the most flexible node type. It allows any number of child nodes or properties, which can have any names.
nt:file represents files. It requires a single child node, called jcr:content. This node type represents images and other binary content.
nt:folder node types can represent folders, like those in a conventional file system.
nt:resource commonly represents the actual content of a file.
nt:version is a required node type for repositories that support versioning.

Exercise 12

- Create base version (version 1.0.0) of the lading page.
    - Check for changes on properties and nodes on the CRXDE.
    - Change some text on the landing page and return to the previous version.

Process

1) Refresh the “training\landingpage” page under /content/.
2) Click on the versioning tab (fourth tab from left to right, on top of the sidekick).
3) Click on “More >>” and set a version name for the initial state of the page: 1.0.0
4) Add as comment: “Initial version”
5) Click on “Create Version”.
6) Change the text and image on the landing page.

7) To restore the page to its previous state click again on the versioning tab.
8) Click on the “Restore Version” section.
9) There should be a list of versions (just one in this exercise).
10) Select the one named “1.0.0” and click on “Restore”.
11) The page should have returned to its previous saved state.

Structured Properties

A Property object represents the smallest granularity of content storage. A property must have one and only one parent node. A property does not have children. When we say that node A "has" property B, it means that B is a child of A.

A property consists of a **name** and a **value**.

Content structure & properties within content

According to David Neuscheler “Rules”:

Rule #2: Drive the content hierarchy, don't let it happen.

The content structure could be guided; following this idea is possible to express content hierarchy and even purpose by designing its structure.

Exercise 13

- Modify the custom title property to update the current page title value.
    - Enable the component to look for multiple options of titles, in this case to use the title property in the dialog and the title of the page as fallback.
- Use a text component to create a “disclaimer” section, “Home” link, and “Contact Us” link within the footer; this can be different on each page.
    - The order is: “Disclaimer”, “Home link”, “Contact Us”
- The header script content should be placed under a node named “header”.
- The page content should be placed under a “content” node.
- The footer content should be placed under a “footer” node.

Process

1) Remove the content on “training\landingpage” (page). This could be avoided by moving the nodes under a new structure but we won’t do that now.
2) Replace the code on **/apps/training/components/content/basictitle** “basictitle.jsp” script with:

```
<%--

   Basic Title Component component.

   Basic Title Component

--%><%
%><%@include file="/libs/foundation/global.jsp"%><%
%><%@page session="false" %><%
    %><%
    String title = currentPage.getTitle();
    if(properties.get("title",String.class)!=null)
        title = properties.get("title",String.class);

String subtitle = currentPage.getProperties().get("subtitle",String.class);
%>
<div class="title-container">
<h1><%= title %></h1>
</div>
<%
if (subtitle != null && !subtitle.equals("")){
%>
    <div class="page-title">
                <h2><%= subtitle %></h2>
    </div>
<% }
%>
</div>
```

3) Modify the title's component path on "header.jsp" from **basicpage**. Add a "header" node to the path, like this:

```
<%--

   Simple header.

   ============================================================================

--%><%@include file="/libs/foundation/global.jsp" %>
<div class="header-container">
   <div id="logo">
        <p>Logo</p>
   </div>
   <cq:include path="header/title" resourceType="training/components/content/basictitle" />
   </div>
```

4) Within "content.jsp" on **landingpage** modify the path to the **parsys** component, append a node "content" before the "par" node, like this:

```
<%@include file="/libs/foundation/global.jsp" %>
<div>
<cq:include path="content/par" resourceType="foundation/components/parsys" />
</div>
```

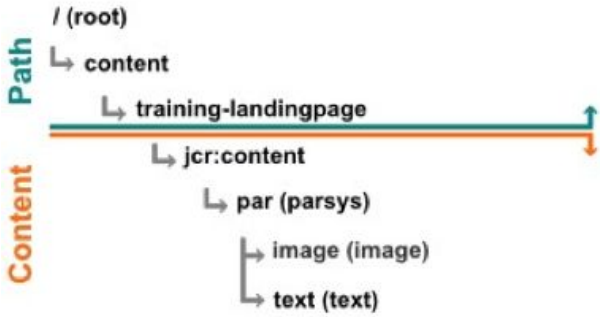
5) Replace the "footer.jsp" code on basicpage, with this one:

```
<%--

Simplefooter.

============================================================================

--%> <%@include file="/libs/foundation/global.jsp" %>
<div class="footer-container">
        <cq:include path="footer/disclaimer" resourceType="foundation/components/text" />
    <cq:include path="footer/home" resourceType="foundation/components/text" />
    <cq:include path="footer/contactus" resourceType="foundation/components/text"/>
</div>
```

6) On the “training\landingpage” (page) add the image and text components again from the last exercise.
7) Update the page content.

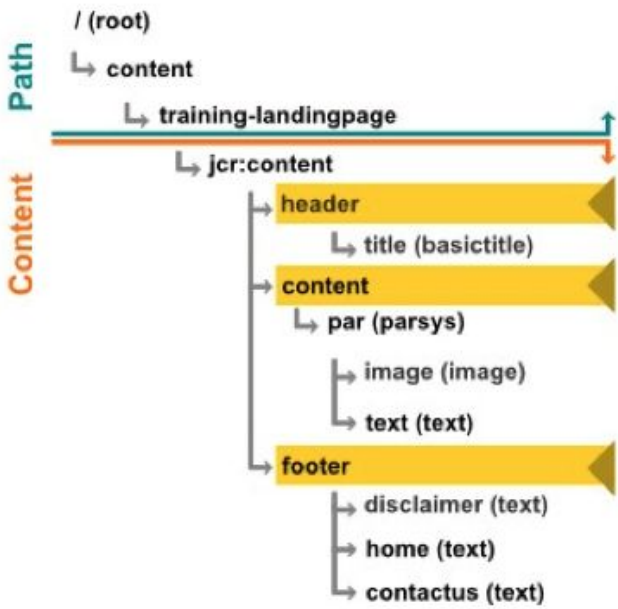
Walkthrough

The idea behind the exercise is to guide the content into a more natural structure thus gaining on readability and order. Until the previous training the content structure looked like this:



The path **/content/trainingElandingpage** (type cq:Page) represents the absolute location of the page node containing the jcr:content, which in time is the actual location of the page content. With the previous exercise the structure under the jcr:content node was “wild”, no actual order was followed and the components created their nodes (or used) right under the content node. The “path” property on the *cq:include* tag works as a location reference, either the content node pointed to exists or is created; the “*resourceType*” property defines the rendering for such content, notice that depending on the “resourceType” property the content will be completely, partially or not display at all. Every new node under the **parsys**, due to its default behavior, will be added under the “par” node (“path” property on the cq:include tag) , every repeated node name, like “image” would get an index number appended to.

Now, guiding the node structure to more natural order we have the current exercise. By modifying the “header.jsp” code and replacing the “path” on the **basictitle** component a new node named header was added. Remember that if the content node doesn’t exist it is created. The same happens to the “content.jsp” script, in charged of managing the dynamic content (actual page content displayed), by replacing the “path” on the **parsys’s** *cq:include*, a new node under jcr:content named “content” which represents the dynamic structure between the header and footer sections. The footer script will change some more due to the need of adding text components to represent the “disclaimer”, “home” and “contact us”. All the new nodes created because of to the “path” property change are of type *“nt:unstructured”*.



Children Nodes

There are specific children nodes that controls certain aspects of the component, like:

- cq:editConfig (cq:EditConfig): Visual aspects.
- cq:childEditConfig (cq:EditConfig): Visual aspects for child components.
- dialog (nt:unstructured): Dialog for editing this component.
- design_dialog (nt:unstructured): Design editing options for this component.

This will be discusses in depth further ahead.

Exercise 14

- Create a new page component named "**infopage**". It should inherit from **basicpage**.
  - Create at least 3 child pages under landing page, using the new **infopage** component.
  - Every **infopage** will have an information section with a component that displays page information, and a navigation section with a list of sibling pages.
- Create a developer only component (for training purposes)
  - List all of the page's properties.
- Create a navigation component.
  - List all child pages under the selected page that will be used by the **basicpage** header to navigate through child pages.

Process

1) Create a new component under **/apps/training/components/page** named infopage, be sure to add the sling:resourceSuperType property as "**/apps/training/components/page/basicpage**".
2) Remove the "infopage.jsp" file. The structure defined on **basicpage** will be used.
3) Add a "content.jsp" script to **infopage** with the following code:

```
1| <%@include file="/libs/foundation/global.jsp" %>
2| <%@page import="com.day.cq.wcm.api.WCMMode" %>
3| <div>
4|   <%
5|   if (WCMMode.fromRequest(request) == WCMMode.EDIT && WCMMode.fromRequest(request)
!= WCMMode.DISABLED) {
6|         %> <cq:include path="content/list" resourceType="training/components/content/listproperties" /> <%
7|   }
8|   %>
9|   <cq:include path="content/par" resourceType="/libs/foundation/components/parsys"
/>
10|   </div>
```

4) Create a new template under /apps/training/templates/ named "infopage", pointing the sling:sourceType property within the jcr:content node to /apps/training/components/page/infopage.

5) Create a new component under **/apps/training/components/** named "**listproperties**":
    a) Name: listproperties
    b) Title: (training) List Properties
    c) Description: (training) List Properties
    d) Group: .hidden
    e) Leave the rest empty.
6) Replace the code on listproperties.jsp with:

```
1| <%@include file="/libs/foundation/global.jsp"%>
2| <%@page import="java.util.Iterator,
3|                 com.day.cq.wcm.api.Page,
4|                 javax.jcr.Property,
5|                 java.util.ArrayList,
6|                 java.util.Set"%>
7|
8| <p>Properties:</p>
9| <ul>
10| <%
11| Set<String> props = currentPage.getProperties().keySet();
12| for (String key : props ) {
13|     String prop = currentPage.getProperties().get(key).toString();
14|     if(prop.length() <  50){
15|     %><li><%= key %> - <%= prop %></li><%
16| } else {
17|     %><li><%= key %> - <%= prop.substring(0,40) %>(<ahref="http://localhost:4502<%=
18| }}
19| %>
20| </ul>
21| <hr>
```

7) Create a new component under **/apps/training/components/content/** named **headernavigation**.
    a) Name: headernavigation
    b) Title: (training) Header Navigation
    c) Description: (training) Header Navigation
    d) Group: .hidden
    e) Leave the rest empty.
8) Add a new dialog named "dialog".
9) Use the basic dialog structure created by the wizard and add a new node under "tab1":
    a) Name: items
    b) Type: cq:WidgetCollection
10) Create a new node under "items":
    a) Name: path
    b) Type: cq:Widget
11) Add a "name" property:
    a) Name: name
    b) Type: String
    c) Value: ./initialPath
12) Add a "xtype" property:
    a) Name: xtype
    b) Type: String
    c) Value: pathfield

13) Add a “fieldLabel” and “fieldDescription” properties, both are String’s and contains information like the label of the field and what is it for.
14) On the “headernavigation.jsp” script add the following code:

```
<%--

   Header Navigation Component component.

   Header Navigation Component

--%><%
%><%@include file="/libs/foundation/global.jsp"%><%
%><%@page session="false" %>
   <%@ page import="java.util.Iterator,
   com.day.cq.wcm.api.PageFilter,
        com.day.cq.wcm.api.Page" %>
   <%
   // get starting point of navigation
   Page navRootPage = null;
if (properties.get("initialPath",String.class)!=null){
        navRootPage = currentPage.getPageManager().getPage(properties.get("initialPath",String.class));
}

if (navRootPage != null) {
    Iterator<Page> children = navRootPage.listChildren(new PageFilter(request));
    %> <ul> <%
    while (children.hasNext()) {
        Page child = children.next(); if(!child.getPath().equals(currentPage.getPath())){
        %><li><a href="<%= child.getPath() %>.html"><%=child.getTitle()
        %></a></li>
        <% } }
                %> </ul> <%
        } else { %>
        <p>Path...</p>
   <% } %>
```

15) Add the **headernavigation** component to the header script on **basicpage** right under the **basictitle** component.

```
<%--

   Simple header.

   ============================================================================

--%><%@include file="/libs/foundation/global.jsp" %>
<div class="header-container">
  <div id="logo">
        <p>Logo</p>
   </div>
   <cq:include path="header/title" resourceType="training/components/content/basictitle" />
   <cq:include path="header/navigation" resourceType="training/components/content/headernavigation" />
   </div>
```

16) Modify the footer script on basicpage to group the links on the bottom and the disclaimer over them, like this:

```
1| <%--
2|
3|   Simple footer.
4|
5|   ============================================================================
6|
7| --%><%@include file="/libs/foundation/global.jsp" %>
8| <div class="footer-container">
9|    <div class="disclaimer-container">
10|    <cq:include path="/content/training- landingpage/jcr:content/footer/disclaimer"
resourceType="foundation/components/text" />
11|    </div>
12|    <div class="links-container">
13|    <cq:include path="/content/training- landingpage/jcr:content/footer/home"
resourceType="foundation/components/text" />
14|    <cq:include path="/content/training- landingpage/jcr:content/footer/contactus"
resourceType="foundation/components/text" />
15|          </div>
16|    </div>
```

17) Add 2 pages under /content/training\landingpage, using the infopage template. Name the pages “training\infopage_1” (Training Information Page 1) and “training\infopage_2” (Training Information Page 2).
18) Load “training\infopage_1” from the WCM Website page. Set **/content/trainingE landingpage** as path to the **headernavigation** component by right clicking on the “*Path...*” message and select edit. Repeat the same procedure on the “training\ landingpage”.

Walkthrough

The new page component will display much more information that the previous exercises. The idea behind the **infopage** page component is to display the properties of the page, actually the properties on the jcr:content node using the **listproperties** component, during edit mode only (developer and author view). This serves no real purpose, is just an example of browsing through the page properties and the edit mode.

The **headernavigation** component is more an example on using the page manager functionality, getting the page child (not children nodes but children pages) and using the page filter class; this follows the same line of the **listproperties** component but using pages. Notice that on the **headernavigation**, line 16, the property “*initialPath*” is being query using the **currentStyle** object, this will try to get the property from the design (this will be cover further ahead) or from the local component property, which is our case.

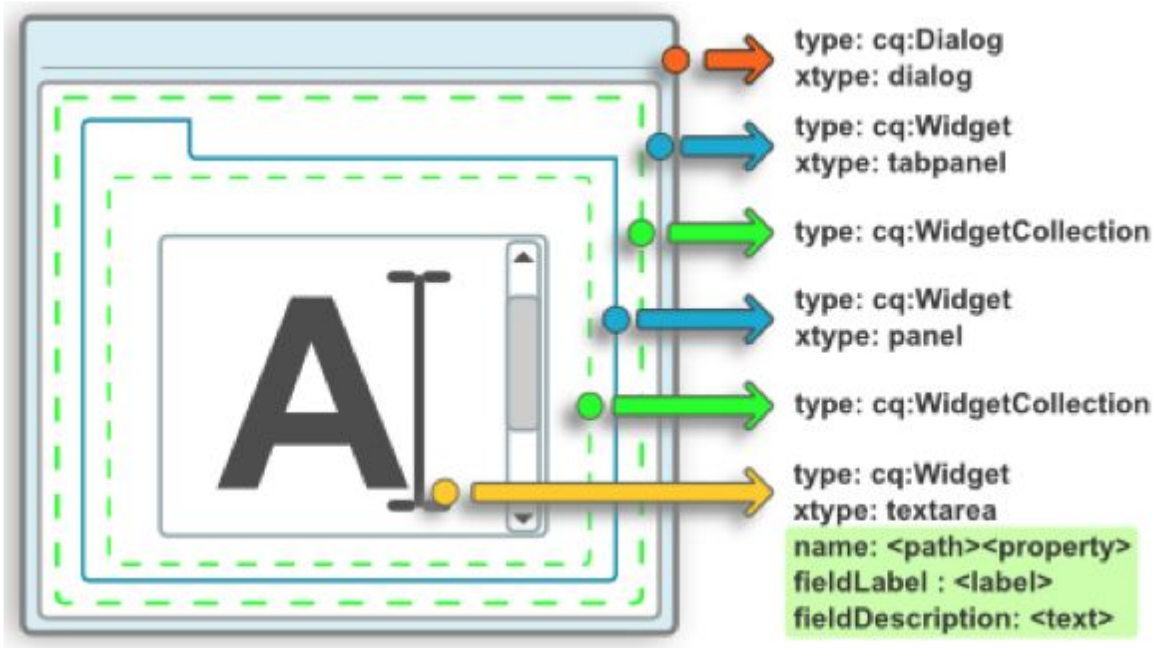
Dialog

The Dialog is a special kind of window with
a form in the body and a button group in the footer.
It is typically used to edit content,
but can also just display information.

Dialogs are used usually to edit content can use a large array of out\of\the\box widgets with simple JavaScript logic, using ExtJS. A dialog can contain forms and panels, use tabs and dynamic widgets that can modify its behavior or the data being inputted.

Dialog are generally used to access and modify properties on a component, this can be done by either referencing the property by name or accessing a “remote” property (out of the component scope) with either absolute or relative paths. The scope of a dialog property is usually the same of the containing component. Nodes within the dialog structure represent its properties.

This is the basic structure of a dialog (with tabs):

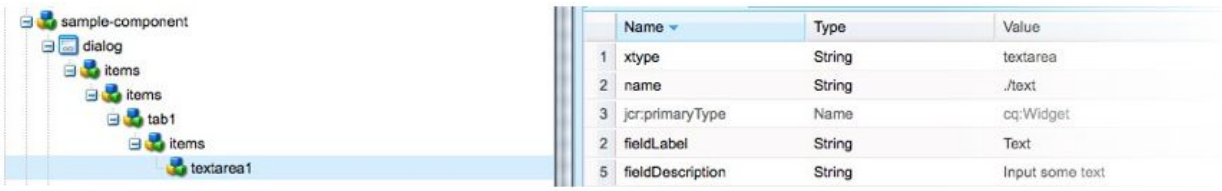


This is the JSON notation (most commonly used format to represent the nodes):

```
{
        "xtype": "dialog",
        "jcr:primaryType": "cq:Dialog",
        "items": {
        "xtype": "tabpanel",
        "jcr:primaryType": "cq:Widget",
        "items": {
        "jcr:primaryType": "cq:WidgetCollection",
        "tab1": {
                "title": "Tab 1",
                "jcr:primaryType": "cq:Panel",
                "items": {
                "jcr:primaryType": "cq:WidgetCollection",
                "textarea1": {
                "xtype": "textarea",
                "fieldLabel": "Text",
                "name": "./text",
                "jcr:primaryType": "cq:Widget",
```

```
                "fieldDescription": "Input some text"
                }
                }
        }
        }
        }
}
```

On CRXDE would look like this:



Dialogs can be inherited just as scripts and properties by components referencing a parent component using the *sling:resourceSuperType* property.

Exercise 15

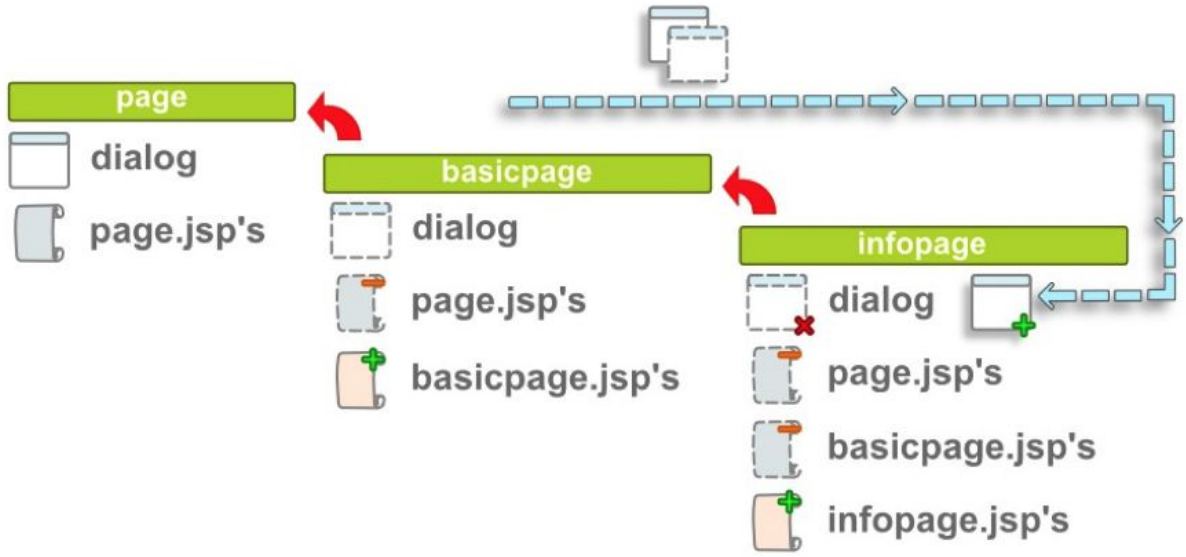
- Modify tabs on the page properties dialog for infopage.
    - Remove all tabs but “basic” and “advance”.
    - On “basic” remove the “On/Off Time” section.
    - On “advance” remove the “Tracking”, “Close User Group” and “Segments” section.

Process

1) Copy the dialog from foundation’s page component (**/libs/foundation/components/page**) into **/apps/training/component/page/infopage.**
2) On **infopage**, open the dialog structure until find the list of tabs; remove everything but “basic” and “advance”.
3) Copy from foundation’s page component the **tab_basic** and **tab_advanced** nodes into **/apps/training/component/page/infopage**.
4) Remove the “**./items/onofftime**” node from tab_basic.
5) Remove the following nodes from tab_advanced: **./items/analytics**, **./items/cug** and **./items/segments.**
6) Check the change on one of the **infopage** pages.

Walkthrough

Thanks to the component inheritance schema dialog, scripts and properties can be passes from one *resourceSuperType* to a child component. Our exercise component, **infopage**, uses **basicpage** as *resourceSuperType*, which in turn uses foundation’s **page** component. The **infopage** component “has” a dialog, which is the one inherited from **page**. However, as any inherited object, a new object defined on the component could override it.



The dialog is copied from page to keep the same structure and save some work; no new nodes need to be created just modified. All the tab nodes on the dialog work as a placeholder by using the *xtype* "cqinclude", which means that the referenced path will be appended to the current node. The *path* property is pointing to a panel node, this panel contains the dialog structure for each tab. Notice that path look like this "***<path_to_tab_node>.infinity.json"***, this means that sling will attempt to resolve the content by applying to selectors, "infinity" and "json". By applying "infinity" a recursion level selector for all the levels below the path to the is used along with the "json" output, which means everything under the node will be represented as "json". The cqinclude *xtype* handles the "json" input and appends the nodes to the dialog structure in runtime.

Logger

CQ offers you the possibility to configure:

- Global parameters for the central logging service.
- Request data logging; a specialized logging configuration for request information.
- Specific settings for the individual services; for example, an individual log file and format for the log messages

Exercise 16

- Create a new configuration to write log messages from the training application to a specific file (pending).

Day 3:

AEM Build up & Build-ins

Today's content:

- Design Dialogs (more on dialogs).
- Widgets.
- Foundation Components.
- Components.
- Content finder.
- Persistency.
- Internationalization
- Debugger.

Design Dialogs (more on dialogs)

The 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).

Access to this global content (properties) is gain through the currentStyle object (com.day.cq.wcm.api.designer.Style, injected to our code using <cq:defineObjects>).

Exercise 17

- Create a logo component that will be used across all pages.
    - Use an image servlet to display the logo.
    - Use foundation's logo as base but rename it as traininglogo.
    - Add the logo to the header script on basicpage.
- Modify the header navigation component to use a design dialog.

Process

1) Copy the component from /libs/foundation/components/logo into /apps/training/components/content/logo.
    a) Rename the logo.jsp as traininglogo.jsp.
    b) script as Change the component title to "(Training) Logo" and add a description property with the same text.
    c) Replace line 12 on img.GET.java with the correct package:
        i) `package apps.training.components.content.logo;`
2) Modify the header script on **basicpage** to look like this:
3) Modify the headernavigation.jsp using this code:

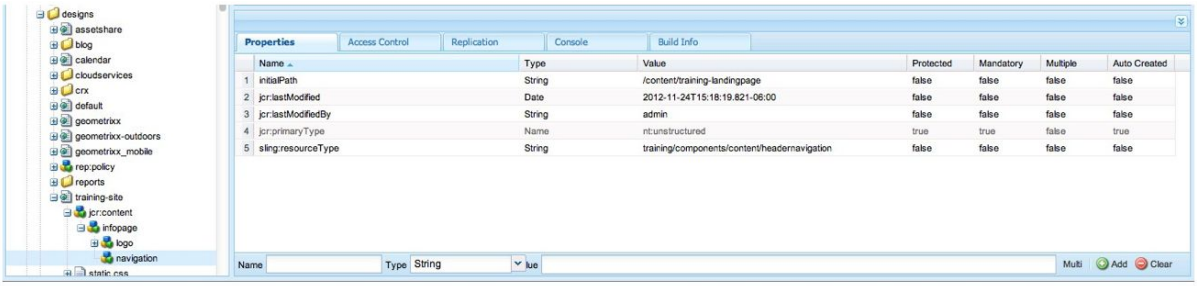
```
1| <%--
2|
3|   Simple header.
4|
5|  ============================================================================
6| --%><%@include file="/libs/foundation/global.jsp" %>
7| <div class="header-container">
8|   <cq:include path="header/logo" resourceType="training/components/content/logo" />
9|   <cq:include path="header/title"
resourceType="training/components/content/basictitle" />
10|   <cq:include path="header/navigation"
resourceType="training/components/content/headernavigation" />
11|   </div>
```

4) Copy the dialog under **headernavigation** under the same component but rename it as "design_dialog".
5) On one of the **infopage** pages:
    a) Change to design mode and edit the logo component use from the content fiender the image "news_new.png".
    b) Still in design mode edit the path value of the **headernavigation** component to point to the parent landingpage.

Walkthrough

Notice that all we did was to copy the logo component from foundation's structure and change the package name of the image servlet. Remember that every script within the component is a render for the content; therefore, calling the component itself with an "img" selector will result on rendering of an image thanks to the image servlet. The most important of this is the result across all the **infopage** pages, where all the page "instances" (as in instances of **infopage**) have the same logo. The same will happened for the **headernavigation** component. On the code for the **headernavigation** the line 16 was added to check on the **currentStyle** object the value of the *initialPath* property. The **currentStyle** object references the Designer helper assigned to this pages, in this case

the assigned the design, “training\site”, from under ***/etc/designs/***, contains the properties related to its design. Like this:



Widgets

AEM uses the ExtJS widgets library, which provides the highly polished user interface elements that work across all the most important browsers and allow the creation of desktop\grade UI experiences.

These widgets are included within AEM and, in addition to being used by AEM itself, can be used by any website built using AEM.

For a complete reference of all the available widgets in AEM you can refer to the widget API documentation or to the list of existing xtypes.

The out\of\the\box widgets shipped with AEM should cover most use cases. However it might be sometimes necessary to create a custom widget to cover a project\specific requirement.

Custom widgets can be created by extending existing ones.

There is a complete API to help you configure or code new behavior on the current set of widgets. Some of these widget definitions include code samples.

Common used widgets

In the ExtJS language, an xtype is a symbolic name given to a class. Read here for a detailed explanation about what an xtype is and how it can be used.

For complete information on all the available widgets in AEM refer to the http://dev.day.com/docs/en/cq/current/widgets\api/index.html

xtype	Description
smartimage	The SmartImage is an intelligent image uploader. It provides tools to process an uploaded image, for example a tool to define image maps and an image cropper.
tags	`CQ.tagging.TagInputField` is a form widget for entering tags. It has a popup menu for selecting from existing tags, includes auto\completion and many other features.
textarea	Multiline text field. Can be used as a direct replacement for traditional textarea fields, plus adds support for auto\sizing.
textfield	Basic text field. Can be used as a direct replacement for traditional text inputs, or as the base class for more sophisticated input controls (like CQ.Ext.form.TextArea and CQ.Ext.form.ComboBox).
richtext	The RichText provides a form field for editing styled text information (rich text).
checkbox	Single checkbox field. Can be used as a direct replacement for traditional checkbox fields.

Foundation Components

Parbase – Parsys

Parbase 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 defines tree scripts to render images, titles, and so on, so that all components sub\ classed from this parbase can use this script.

Users do not need access to the parbase.

The Paragraph System (know as **parsys**) is a key part of a website as it manages a list of paragraphs. It is used to structure the individual pieces of content on a website. You can create paragraphs inside the Paragraph System, move, copy, and delete paragraphs and also use a column control to structure your content in columns. It is based on *Parbase*, which is a key component as it allows components to inherit attributes from other components. Users do not need access to this component.

The Paragraph System provided in AEM WCM foundations covers most of the variants needed and can also be configured by allowing you to select the components to be activated/deactivated within your current paragraph system. Can be found here:

```
1| /libs/foundation/components/parsys
```

Each paragraph type is represented as a component.
The paragraph system itself is also a component,
which contains the other paragraph components.

Exercise 19

- Create a new parsys component with a limit capacity.

Process

1) Copy the component from /libs/foundation/components/parsys into /apps/training/components/content/ and rename it limitedparsys.
    a) Rename the parsys.jsp as well.
2) On the limitedparsys.jsp script replace the code with this one:

```
1|
2| <%--
3|
4| LimitedParsys
5|
6| --%>
7|
8|<%@page import="com.day.cq.wcm.foundation.Paragraph,
9| com.day.cq.wcm.foundation.ParagraphSystem,
10|   com.day.cq.wcm.api.components.IncludeOptions,
11|   com.day.cq.commons.jcr.JcrConstants,
12|     com.day.cq.wcm.api.WCMMode,
13|   org.apache.sling.api.request.RequestParameterMap,
14|   org.apache.sling.api.request.RequestParameter " %>
15|
16|       <%@include file="/libs/foundation/global.jsp"%>
17|
18| <%
19|
21| String suffix = slingRequest.getRequestPathInfo().getSuffix();
22| int limit = 0;
23| if (suffix != null) {
24|       try {
25|               limit = Integer.parseInt(suffix);
26|       } catch (NumberFormatException e) {
27|               // Default limit of zero elements
28|       }
29|   }
30| // ============================================
31|
32|   ParagraphSystem parSys = ParagraphSystem.create(resource, slingRequest);
33| String newType = resource.getResourceType() + "/new";
34|
35|   boolean hasColumns = false;
36| for (Paragraph par: parSys.paragraphs()) {
37|               if (editContext != null) {
38|                       editContext.setAttribute("currentResource", par);
39|       }
```

```
40|         switch (par.getType()) {
41|                 case START:
42|                 if (hasColumns) {
43|                                 // close in case missing END
44|                                 %></div></div><%
45|                 }
46|         if (editContext != null) {
47|                 // draw 'edit' bar
48|                         IncludeOptions.getOptions(request, true).getCssClassNames().add("section");
49|                         %><sling:include resource="<%= par %>"/><%
50|         }
51|     } // open outer div
52| }
53| %><div class="parsys_column <%= par.getBaseCssClass()%>"><%
54| // open column div
55| %><div class="parsys_column <%= par.getCssClass() %>"><% hasColumns = true;
56| break;
57| case BREAK:
58| if (editContext != null) {
59| // draw 'new' bar
60| IncludeOptions.getOptions(request, true).getCssClassNames().add("section");
```

```
61|    %><sling:include resource="<%= par %>" resourceType="<%= newType %>"/><%
62|    }
63|    // open next column div
64|    %></div><div class="parsys_column <%= par.getCssClass() %>"><%
65|    break;
66|    case END:
67|    if (editContext != null) {
68|         // draw new bar
69|    IncludeOptions.getOptions(request, true).getCssClassNames().add("section");
70|    %><sling:include resource="<%= par %>" resourceType="<%= newType %>"/><%
71|    }
72|    if (hasColumns) {
73|    // close divs and clear floating
74|    %></div></div><div style="clear:both"></div><%
75|    hasColumns = false;
76|    }
77|    if (editContext != null && WCMMode.fromRequest(request) == WCMMode.EDIT) {
78|    // draw 'end' bar
79|    IncludeOptions.getOptions(request, true).getCssClassNames().add("section");
80|         %><sling:include resource="<%= par %>"/><%
81|    }
82|    break;
83|    case NORMAL:
84|    // include 'normal' paragraph
85|    IncludeOptions.getOptions(request, true).getCssClassNames().add("section");
86|
87|    // draw anchor if needed
88|    if (currentStyle.get("drawAnchors", false)) {
89|         String path = par.getPath();
90|    path =
path.substring(path.indexOf(JcrConstants.JCR_CONTENT)
91|         + JcrConstants.JCR_CONTENT.length() + 1);
92|    String anchorID = path.replace("/", "_").replace(":", "_");
93|    %><a name="<%= anchorID %>" style="visibility:hidden"></a><%
94|    }
```

```
95|   %><sling:include resource="<%= par %>"/><%
96|   break;
97|   }
98|   }
99|   if (hasColumns) {
100|    // close divs in case END missing. and clear floating
101|        %></div></div><div style="clear:both"></div><%
102|   }
103|
104|   // ================== Comment #2 ======================
105|   if (parSys.paragraphs() != null && (parSys.paragraphs().size() < limit || limit == 0)) {
106|   if (editContext != null) {
107|   editContext.setAttribute("currentResource", null);
108|   // draw 'new' bar
109|   IncludeOptions.getOptions(request, true).getCssClassNames().add("section");
110|   %><sling:include path="*" resourceType="<%= newType %>" replaceSuffix="<%= (new Integer(limit -
parSys.paragraphs().size())).toString() %>"/><%
111|   }
112|   }
113|   // ================================================
114|
115|   %>
116|
117|   <!-- ================== Comment #3 ====================== -->
118|   <% if (WCMMode.fromRequest(request) == WCMMode.EDIT) { %>
119|   Max components <%= limit %>, size <%= parSys.paragraphs().size() %>
120|  <% } %>
121|  <!-- ================================================ -->
122|
```

At this level, 3 modifications (wrapped by comment lines) are required in order to control 3 phases: Keep track, control and inform:

a) Obtain the replaceSuffix value set during the <sling:include> declaration, which specifies the amount of accepted elements. In case an invalid value is sent, just defaults to zero.
b) Compare the number of current elements in the parsys with the limit value previously obtained. If the parsys reached the limit of elements, then DO NOT display the drop area used to add more elements.
c) As a visual alert, and in EDIT mode only, just display a visual alert with the limit of elements for this parsys, and the number of elements currently inside of it.

3) Optionally, we can also remove the visual alert in the main JSP and change the /new/new.jsp file in order to display a custom message in the drop area. By default, the message parsys display here is "Drag components or assets here", but now we need it to inform authors how many elements they can add before reaching the elements limit:

```
123|   <%--
124|
125|   Limit parsys/new component
126|
127|   This component does not provide any output and is used only for the "new"
128|   edit bar. It respects the "currentResource" edit context attribute that allows
129|   correct inserting components in the paragraph system.
130|
131|   --%>
132|
133|   <%@ page session="false" import="
134|   com.day.cq.wcm.api.components.EditContext,
135|   com.day.text.Text,
136|   org.apache.sling.api.resource.Resource,
137|   com.day.cq.wcm.commons.WCMUtils" %>
138|
139|   <!-- ================== Comment #2 ====================== -->
140|   <%@include file="/libs/foundation/global.jsp"%>
141|   <!-- ================================================== -->
142|
143|   <%
144|
145|   // ================== Comment #1 ======================
146|   String suffix = slingRequest.getRequestPathInfo().getSuffix();
147|   editContext.getEditConfig().setEmpty(true);
148|   if (Integer.parseInt(suffix) > 0) {
149|       editContext.getEditConfig().setEmptyText(suffix + " component(s) to go");
150|   }
151|   // ==================================================
152|
153|   if (editContext != null) {
154|   if (editContext.getParent() != null) {
155|       Resource curRes = (Resource) editContext.getParent().getAttribute("currentResource");
156|    if (curRes != null) {
157|    String prev = Text.getName(curRes.getPath());
158|    editContext.getEditConfig().setInsertBehavior("before " + prev);
159|    }
160| }
```

```
161|   }
162|
163| %>
```

As you can see, just a few minor modifications are required:

a) The replaceSuffix value is retrieved from the request and used in the custom message.
b) The global.jsp reference is added because we need access to the slingRequest object to retrieve the suffix value.

4) Create a test page and add a limited parsys that allows only 5 elements. It's just as easy as this:

```
164|   <sling:include path="content-path" resourceType="myapp/components/content/limited- parsys" replaceSuffix="5"/>
```

Author interactions

The child node cq:editConfig controls visual aspects; for example, it can define the appearance of a bar or widget, or can add customized controls. The properties used to Configure the cq:EditConfig node are:

- cq:actions, defines one or several actions that can be performed on the component.

Property Value	Description
text:<some text>	Displays the static text value <some text>
\	Adds a spacer
edit	Adds a button to edit the component
delete	Adds a button to delete the component
insert	Adds a button to insert a new component before the current one
copymove	Adds a button to copy and cut the component

- cq:layout, defines how the component can be edited.

Property Value	Description
rollover	The component edition is accessible "on mouse over" through clicks and/or context menu. For advanced use, note that the corresponding client side object is: CQ.wcm.EditRollover.
editbar	The component edition is accessible through a toolbar. For advanced use, note that the corresponding client side object is: CQ.wcm.EditBar.
auto	The choice is left to the client side code.

- cq:dialogMode, defines how the component dialog will be opened.

Property Value	Description
floating	The dialog is floating.
inline	The dialog is anchored over the component.
auto	if the component width is smaller than the client side CQ.themes.wcm.EditBase.INLINE_MINIMUM_WIDTH value, the dialog is floating, otherwise it is inline.

- cq:emptyText, defines text that is displayed when no visual content is present.
- cq:inherit, defines if missing values are inherited from the component that it inherits from.

Drag & Drop

The *cq:dropTargets* node (node type *nt:unstructured*) defines a list of drop targets that can accept a drop from an asset of the **content finder**. It serves as a collection of nodes of type *cq:DropTargetConfig*.

Each child node of type cq:DropTargetConfig defines a drop target in the component. The node name is important because it must be used in the JSP as follows to generate the CSS class name assigned to the DOM element that is the effective drop target:

```
1| <drop_target_css_class> = <drag&drop_prefix> + <node_name_drop_target>
```

The <drag&drop_prefix> is defined by the Java property *com.day.cq.wcm.api.components.DropTarget.CSS_CLASS_PREFIX.*

For example:

```
1| String ddClassName = DropTarget.CSS_CLASS_PREFIX + "file";
```

"file" being the node name of the drop target in the edit configuration of the Download component.

Property Value	Description
accept	The dialog is floating.
groups	if the component width is smaller than the client side CQ.themes.wcm.EditBase.INLINE_MINIMUM_WIDTH value, the dialog is floating, otherwise it is inline.

Sample node structure:

- (N) cq:editConfig (T= cq:EditConfig)
    - (N) cq:dropTarget (T= nt:unstructured)
        - (N) image (T= cq:DroptTargetConfig)
            - (P) accept (T= String[]) (V= image/.*)
            - (P) groups (T= String[]) (V= media)
            - (P) propertyName (T= String) (V= ./filereference)

In place editing

The cq:inplaceEditing node (node type cq:InplaceEditingConfig) defines an inplace editing configuration for the component.

Property Value	Description
active	True to enable the inplace editing of the component.
configPath	Path of the editor configuration. The configuration can be represented as a configuration node.
editType	Editor type. ● **plaintext**: to be used for non HTML content. ● **title**: is an enhanced plaintext editor that converts graphical titles into a plaintext before editing begins. Used by the Geometrixx title component. ● **text**: to be used for HTML content (uses the Rich Text Editor).

Sample node structure:

- (N) cq:editConfig (T= cq:EditConfig)
    - (N) cq:inplaceEditing (T= cq:inplaceEditingConfig)
        - (P) active (T= String) (V= {Boolean}true)
        - (P) editorType (T= String) (V= plaintext)

Component actions

The cq:listeners node (node type cq:EditListenersConfig) defines what happens before or after an action on the component.

Property Value	Description
beforedelete	The handler is triggered before the component is removed.
beforeedit	The handler is triggered before the component is edited.
beforecopy	The handler is triggered before the component is copied.
beforeinsert	The handler is triggered before the component is inserted.
beforemove	The handler is triggered before the component is moved.

beforechildinsert	The handler is triggered before the component is inserted inside another component (containers only).
afterdelete	The handler is triggered after the component is removed. (REFRESH_SELF)
afteredit	The handler is triggered after the component is edited. (REFRESH_SELF)
aftercopy	The handler is triggered after the component is copied. (REFRESH_SELF)
afterinsert	The handler is triggered after the component is inserted. (REFRESH_INSERTED)
aftermove	The handler is triggered after the component is moved. (REFRESH_SELFMOVED)
afterchildinsert	The handler is triggered after the component is inserted inside another component (containers only).

Sample node structure:

- (N) cq:editConfig (T= cq:EditConfig)
  - (N) cq:listeners (T= cq:EditListenersConfig)
    - (P) afterdelete (T= String) (V= REFRESH_PAGE)
    - (P) afteredit (T= String) (V= REFRESH_PAGE)
    - (P) afterinsert (T= String) (V= REFRESH_PAGE)
    - (P) afterMove (T= String) (V= REFRESH_PAGE)

Exercise 19

- Create a “training image” component, using as base foundation’s image component.
  - Leave out any tracking and design properties information.
  - Make sure to have drag & drop features enabled.
  - Force to reload the page after editing the component.

Persistency

Day CRX by default stores the data in tar files, using the Tar PM (Tar Persistence Manager). This is quite different from Jackrabbit (the JCR reference implementation), which uses a SQL database.

The Tar PM supports ACID (atomicity, consistency, isolation, durability), but it doesn't know SQL, or relational integrity, and stores the data in another way.

The biggest difference to a regular database is: the Tar PM is append-only.

- It doesn't do any in\place updates.
- Whenever there is a change, the Tar PM appends an entry to the newest file.
- TARPM is faster than a regular database because each change results in only one write operation.
- Unused, old data is removed in a separate optimize process during off\peak hours, for example at night.
- The Tar PM uses a special append\only index. Keys are stored in one or multiple index segments. Each segment is a sorted list, and each persistent segment is a file.

A key lookup goes like this:

1. First, the Tar PM looks in the in\memory index segment, which is implemented as a hash table.
2. If the key is not found, then a lookup in the cache is made, which is also just a hash table.
3. Afterwards, the Tar PM checks the persistent index segments, newest segment first.
4. Entries are sorted by key (131\903) in the index segment. The index contains pointers ([1]\[9]) where the actual content is stored in the data file.
5. Like the data area, the index is append\only, that means data is never overwritten. Old, unused index segments are removed once they are no longer needed.

The file format of most database systems is proprietary, which makes it hard or impossible to read. The Tar PM uses the standard tar file format. If you are interested, you can inspect the files using one of the many tools that support this format. The tar format is future proof, and has a number of other advantages. One example is point\in\time recovery: while not directly implemented in the Tar PM yet, it is actually quite simple to do manually \ just truncate the tar file at a given entry.

Another advantage of the append\only nature is 'backup\ability': Files are never modified after they are written, so backing up the repository is very simple \ just copy all files.

Internationalization

The **i18n** capabilities allow you to provide dynamic "messages" based on the visitors language.

Internationalization is 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 after the ISO code of the languages that should be supported (e.g. es, en, de, fr, etc.). Below these language nodes are the message nodes (nodeType *sling:MessageEntry*), which will contain a simple key\message pair.

Exercise 20

- Follow the instructions on the “i18n.pdf” document.

Debugger

Using the following command you could initiate the instance with a debugger port attached:

```
java  -Xdebug -Xnoagent -Djava.compiler=NONE
   -  Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=30303
   -  - XX:+HeapDumpOnOutOfMemoryError -XX:MaxPermSize=256M -Xmx1024m
      -Dorg.apache.sling.commons.log.level=INFO     -jar  cq5-author-4502.jar  -p
      4502   - verbose –nofork
```

This will create a port on 30303, which can be used by CRXDE or eclipse to connect to and fetch symbolic data to the debugger session.

Day 4:

Under the hood

Today's content:

- ExtJS Overview.
- OSGi.
- Workflows.
- Content Packages.
- Content finder.
- Performance.
- Security.

ExtJS.

(In Progress) References to Adobe Documentation.
http://dev.day.com/docs/en/cq/current/developing/widgets.html
http://dev.day.com/docs/en/cq/current/widgets\api/index.html

OSGi.

Bundles

(In Progress) Hello World Bundle using CRXDE.

Exercise 21

- Bundles

Services

An OSGi service is a java object instance, registered into an OSGi framework with a set of properties. Any java object can be registered as a service, but typically it implements a well\known interface. The client of a service is always an OSGi bundle, i.e. a piece of java code possible to start via the BundleActivator interface. Each bundle may register zero or more services. Each bundle may also use zero or more services. There exists no limit on the number of services, more than the ones given by memory limits or java security permissions. On the other hand, a component can be a consumer or a provider of a service, or both, or neither, and may have any of the following features, in combination or alone:

- A component may be started and stopped; this would be considered an "active" component, though that is also an informal term. A component that doesn't need to be started or stopped is called passive.
- A component may publish itself as an OSGi service.
- A component may bind to or consume OSGi services.

The service concept is a very general\purpose tool, but some examples are:

- Export functionality from a bundle to other bundles.
- Import functionality from other bundles.
- Expose external devices, such as UPnP devices or even hardware, to other OSGi bundles.

- Expose java code running in OSGi to an external network, e.g. via the UPnP or SOAP protocols.
- Bundle configuration, using the Configuration Manager (i.e., the Felix Console).

Generally, services are the preferred method bundles should use to communicate between each other.

In order to register a class as a service, some annotations provided by Apache Felix need to be used. The only required annotation is @Component which is the one that actually declares the class as a component. The following class has been declared as such, and uses some of the optional annotations to provide information about itself (for more information, please go to SRC annotations):

```
1| import org.apache.felix.scr.annotations.Component;
2| import org.apache.felix.scr.annotations.Properties;
3| import org.apache.felix.scr.annotations.Property;
4| import org.apache.felix.scr.annotations.Service;
5| import org.osgi.framework.Constants;
6| import org.osgi.service.component.ComponentContext;
7|
8| @Component(immediate = true, metatype = true, enabled = true, label = "Example component")
9| @Service
10|    @Properties({
11|    @Property(name = Constants.SERVICE_DESCRIPTION, value = "My example OSGi component"),
12|    @Property(name = Constants.SERVICE_VENDOR, value = "Razorfish"),
13|    @Property(name = "process.label", value= "A sample component" })
14| public class ExampleComponent implements ServiceInterface {
15|
16|     protected void activate(ComponentContext context) {
17|         // Optional startup routines.
18|     }
19|
20| }
```

The activate method is always called as soon as the service is started (depending on configuration, done by setting “immediate=true”), making it useful for any initialization routines. Once the bundle is compiled and uploaded to the AEM instance, the new component can be viewed from the Web Console:

- Click the Components tab.
    - Lookup the name of the component. In this example, look for "ExampleComponent", and click it.
    - A small pane shows up containing the basic information and properties of an OSGi service.

Referencing other services

The #1 rule for referencing OSGi services is:

Both the referencing and referenced services must be OSGi services

---

To do this, the @Reference annotation will be used to mark an instance variable as an OSGi reference that will be injected by Felix.

```
1| import org.apache.felix.scr.annotations.Component;
2| import org.apache.felix.scr.annotations.Properties;
3| import org.apache.felix.scr.annotations.Property;
4| import org.apache.felix.scr.annotations.Reference;

5| import org.apache.felix.scr.annotations.Service;
6| import org.osgi.framework.Constants;
7| import org.osgi.service.component.ComponentContext;
8|
9| @Component(immediate = true, metatype = true, enabled = true, label = "Example component")
10|    @Service
11|    @Properties({
12|    @Property(name = Constants.SERVICE_DESCRIPTION, value = "My example OSGi component"),
13|    @Property(name = Constants.SERVICE_VENDOR, value = "Razorfish"),
14|    @Property(name = "process.label", value= "A sample component" })
14| public class ExampleComponent implements ServiceInterface {
14|
15|         @Reference
16|         InjectedComponent component;
17|
18|         protected void activate(ComponentContext context) {
19|                 // Optional startup routines.
20|         }
21|}
```

Felix itself will lookup and inject the concrete service class that actually implements the InjectedComponent interface to the component instance variable. Additionally, we can also specify the concrete instance that will be injected in case a specific implementation is required. Consider the case of an Object Factory used by the ExampleComponent service to create controlled instances of business objects:

```
1| import org.apache.felix.scr.annotations.Component;
2| import org.apache.felix.scr.annotations.Properties;
3| import org.apache.felix.scr.annotations.Property;
4| import org.apache.felix.scr.annotations.Service;
5| import org.osgi.framework.Constants;
6|
7| @Component(immediate = true, metatype = true, enabled = true, label = "Object Factory")
8| @Service(CoreDynamicFactory.class)
9| @Properties({
10|    @Property(name = Constants.SERVICE_DESCRIPTION, value = "My Business Objects Factory"),
10| @Property(name = Constants.SERVICE_VENDOR, value = "Razorfish"),
10| @Property(name = "type", value = "coreFactory") })
10| public class CoreFactory {
10|     public <T> T create(String objectType) {
10|         // Your code here.
10|     }
11| }
```

A new @Property has been added to the service annotations: @Property(name = "type", value = "coreFactory"). It allows the developer to reference a specific implementation of a service using a unique String instead of having Felix resolve the implementation class based on the interface. In the referencing class, the @Reference annotation of the instance variable will now have a target attribute:

```
1| import org.apache.felix.scr.annotations.Component;
2| import org.apache.felix.scr.annotations.Properties;
3| import org.apache.felix.scr.annotations.Property;
4| import org.apache.felix.scr.annotations.Reference;
5| import org.apache.felix.scr.annotations.Service;
6| import org.osgi.framework.Constants;
7| import org.osgi.service.component.ComponentContext;
8|
9| @Component(immediate = true, metatype = true, enabled = true, label = "Example
component")
10|    @Service
11|    @Properties({
12|    @Property(name = Constants.SERVICE_DESCRIPTION, value = "My example OSGi component"),
13|    @Property(name = Constants.SERVICE_VENDOR, value = "Razorfish"),
14|    @Property(name = "process.label", value= "A sample component" })
14| public class ExampleComponent implements ServiceInterface {
15|
16| @Reference(target="(type=coreFactory)")
17| InjectedComponent component;
18|
19|         protected void activate(ComponentContext context) {
20|                 // Optional startup routines.
21|         }
22| }
```

Excercise 22

- Bundle(Eclipse)

Workflows

The basic showcase where workflows appear on AEM is based on the idea that content is often subject to organizational processes, including steps such as approval and sign\off by various participants.

A workflow is made of steps. Automated steps, also called process steps, can be defined by using either an ECMA script or a service (a Java class in a bundle). Rules for OR split can be defined with ECMA scripts. Services can be developed to listen to special workflow events and perform tasks according to the business logic.

Workflows are at the center of how form submissions in CQ WCM are typically processed. Such task usually implies notifications (emails), submission and content modification (jcr).

Models

Workflow models consist of a series of steps of various types, which can be extended with scripts to provide the functionality, and control you require. Is made of ***WorkflowNodes*** *and* ***WorkflowTransitions****.*

The transitions connect the nodes and define the "flow", and define the link between two consecutive steps. 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.

A workflow (model) can be triggered by an event or applied to a resource (asset, page, etc.).

Steps

Every process step is a component in the sidekick. It allows building step specific and user\ friendly edit dialogs, especially for configuring process and input arguments.

Participant Step

A participant step enables you to assign ownership for a particular action.

The workflow will only proceed when the user has manually acknowledged the step. Although not directly related, user authorization must be considered when assigning an action; the user must have access to the page (or resource).

A Participant step “Step 1” is automatically added to every new workflow model.

A Dynamic Participant Step allows selecting custom participant chooser implementations for easy integration into the workflow model.

Process Step

Process steps are automatic actions that are executed by the system if certain specified conditions are met.

A process step allows you to specify scripts and bundles to be executed.

Using the Process Step component allows a single argument string in the components edit dialog. Arguments are mapped with the key PROCESS_ARG; could be used as a single parameter or split it various tokens.

Flow control

The flow control steps work by splitting the execution flow into paths.

- AND
- OR

An “AND” creates a split in the workflow; both branches will be active. This component enables you to introduce multiple processing paths into the workflow. For example you can allow certain review steps to occur in parallel, saving time.

The “OR” creates a split in the workflow, whereby only one branch is active. This allows you to introduce conditional processing paths into your workflow. This conditional processing if done by using an ECMA script with a function named “check()”. Like this:

```
1| function check(){
2|    return true;
3| }
```

A much more meaningful code would the one that executes one of the branches if the resource being processed is located within a certain structure (path).

```
1| function check(){
2|    var workflowData = workItem.getWorkflowData();
3|    if (workflowData.getPayloadType() == "JCR_PATH") {
4|           var path = workflowData.getPayload().toString();
5|    if (path.indexOf("/content/training_site/en") == 0)
6|           return true;
7|    }
8|    return false;
9| }
```

Exercise 23

- Develop a simple authorization (publish) workflow with
    - 1 participant step where the Administrator should approve or disapprove a publish process.
    - 2 built\in process steps
        - 1 for the publishing process (approval)
        - 1 NOOP (no process) for disapproval.

Workflow step programming

Programming workflows required knowledge on either Java programming language or in ECMA scripting. We are not covering either since is not the aim of the training but some general java developing guidelines will be mentioned.

This programming can be done on either CRXDE (lite) or in your favorite Java IDE. For CRXDE (lite) development you should take into account which editor you are using due to some differences.

If you are trying to reference an OSGi service or component using the SCR annotations, like the SlingRepository in the code below, then you may encounter a NullPointerException when you try to use these objects.

```
1| /**
2| * @scr.component immediate="true"
3| * @scr.service interface="SampleService"
4| */
5| public class SampleServiceImpl implements SampleService {
6|
7| /**
8| * @scr.reference
9| */
10|    private SlingRepository repository; ...
```

Another symptom of this problem is that your SampleService component may not show up or register correctly in the AEM Web Console “components” tab (***system/console/components***).

The reason for this is SCR annotations are deprecated in the latest builds of Apache Felix.

You should note that CRXDE and CRXDE Lite are both configured to automatically resolve these annotations and build the relevant XML files for you. In other IDE environments you will have to include the maven\scr\plugin to resolve these annotations and build the XML files yourself.

The Solution to correctly reference these objects in the latest versions you should use the following syntax (note: you will have to explicitly import the Felix scr classes):

```
1| import org.apache.felix.scr.annotations.Component;
2| import org.apache.felix.scr.annotations.Service;
3| import org.apache.felix.scr.annotations.Reference;
4|
5| @Component(immediate="true")
6| @Service(interface="SampleService")
7|
8| public class SampleServiceImpl implements SampleService {
9|
10|    @Reference
11|    private SlingRepository repository;
12|    ...
```

Process Step

Process steps are coded by implementing the **WorkflowProcess** interface.

```
1| import com.day.cq.workflow.exec.WorkflowProcess;
2|
3| @...
4|
5| public class myClass implements WorkflowProcess {
6|
7|          public void execute(WorkItem item, WorkflowSession session, MetaDataMap args) throws WorkflowException { .. }
8| }
```

The workflows usually work over some entity (node, asset, page) and can be selected in a implicit manner from the context where a workflow instance started.

```
1| WorkflowData workflowData = item.getWorkflowData();
2| workflowData.getPayloadType();
3| workflowData.getPayload();
```

The Session object provides read\and\write access to the content of a particular workspace in the repository and the WorkflowSession provides all functionality (depending on the users rights) for managing WorkflowModels, Workflow instances and their execution.

```
1| SlingHttpServletRequest slingReq = (SlingHttpServletRequest) request;
2| Session session = slingReq.getResourceResolver().adaptTo(Session.class);
3| //---
4| WorkflowSession wfSession = workflowService.getWorkflowSession(session);
```

The parameters from the process step components can be retrieved like this:

```
1| String argument = args.get("PROCESS_ARGS", "default value");
```

Accessing nodes on a workflow process can be achieved like this (needs a session):

```
1| Node node = (Node) session.getSession().getItem(path);
2| Node node = session.getSession().getNode(path);
```

Participant Step

Participant steps are coded by implementing the **ParticipantStepChooser** interface.

```
1| import com.day.cq.workflow.exec.ParticipantStepChooser;
2|
3| @...
4|
5| public class myClass implements ParticipantStepChooser {
6|
7|         public String getParticipant (WorkItem item, WorkflowSession session, MetaDataMap args) throws
8| WorkflowException { .. }
9| }
```

This “getParticipant()” method returns the participant id.

Exercise 24

- Create a new workflow that deletes all children pages from a given path.
    - All nodes will be deleted but the jcr:content
    - Extra:
        - Add a process that moves all children pages under another path to archive such pages.
        - Make the admin choose between delete or archive.

Workflow execution

The Workflow console is the centralized location for workflow management in CQ. There are several ways a workflow can be started; from a form submission, from the Workflow Console, from the Websites Console, or from the workflow tab in the Sidekick.

In the “Models” tab of the workflow console, it’s possible to kick off a new workflow instance by selecting the model and clicking the “Start” button (on top). A payload should be assigned, which includes pages, nodes, resources, and so on. You can use the drop down menu to browse the repository.

Also, a rule can be used to run workflow by defining a “relationship” on the “launcher” tab. Various definitions are included with the standard installation. These are used for digital asset management and social collaboration tasks.

The “relationship” is defined by some parameters like:

- **Event Type:** The event type that will launch the workflow, like Created, Modified or Removed.
- **Notetype:** The type of node that the workflow launcher applies to, usually a page or its content, and asset or regular nodes like nt:folder.
- **Path:** The path that the workflow launcher applies to, this is the place where the event observer will be waiting.
- **Conditions**: Conditions for node values that, when evaluated, determine whether the workflow is launched.
- **Workflow**: The workflow to launch when the Event Type occurs on the Nodetype and/or Path under the defined Condition.
- **Activate**: Controls whether the workflow launcher is activated.
- **Run Mode(s)**: The type of server that the workflow launcher applies to. Select Author, Publish, or Author & Publish.

A workflow can be started from the “sidekick”, the current page would be use as payload.

Content Packages.

A CRX package is used to package up arbitrary content from a CRX repository. ‘Packaging up’ in this case means that you can select branches inside the content repositories, which are then added to a file downloadable from the repository. That file can be uploaded to any instance and the content contained therein can be installed.

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 displays the content from the repository as an easy\to\use\and\edit representation of files and folders.

Additionally, it contains vault metadata, 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 that are not used for package extraction; these properties are for the content package consumer for informational purposes only.

Performance.

Monitoring

Performance (or the lack of it) is one of the first things that your users notice, so as with any application with a user interface, performance is of key importance. To optimize the performance of your CQ installation you need to monitor various attributes of the instance and its behavior.

The problems that cause performance issues are often difficult to track down, even when their effects are easy to see.

A basic starting point is a good knowledge of your system when it is operating as normal. Unless you know how your environment "looks" and "behaves" when it is performing properly, it can be difficult to locate the problem when performance deteriorates. This means that you should spend some time investigating your system when it is running smoothly and ensure that collecting performance information is an ongoing task. This will provide you with a basis for comparison should the performance suffer.

There is a list of possible error

- Dispatcher caching inefficiency.
- Use of queries in normal display
- Poorly written component scripts

Keep in mind that there are multiple logs designed to give as much information as possible about warming and failures of the system as seem before. Those logs, a java profiler and a good programing logic are your best friends.

Optimization

Certain rules should be kept in mind when optimizing performance:

- Performance tuning must be part of every project.
- Do not optimize early in the development cycle.
- Performance is only as good as the weakest link.
- Always think about capacity vs. volume.
- Optimize important things first.
- Never optimize without realistic goals.

Common performance problems:

Area	Symptom(s)	To increase capacity...	To reduce volume...
Client	High client CPU usage.	Install a client CPU with higher performance.	Simplify (HTML) layout.
	Low server CPU usage.	Upgrade to a faster browser.	Improve client-side cache.
	Some clients fast, some slow.		
Server			
Network	CPU usage low on both servers and clients.	Remove any network bottlenecks.	Improve/optimize the configuration of the client cache.
	Browsing locally on the server is (comparatively) fast.	Increase network bandwidth.	Reduce the "weight" of your web pages (e.g. less images, optimized HTML).
Web-server	CPU usage on the web-server is high.	Cluster your web-servers.	Reduce the hits per page (visit).
		Use a hardware load- balancer.	
Application	Server CPU usage is high.	Cluster your AEM instances.	Search for, and eliminate, CPU and memory hogs (use code review, timing output, etc).
	High memory consumption.		Improve caching on all levels.
	Low response times.		Optimize templates and components (e.g. structure, logic).
Repository			
Cache			

Security

Adobe has a basic list that addresses security concerns and basic instance administration.

1. **Change Default Passwords**. An out\of\the\box installation of CQ includes various accounts to enable you to administrate and use the instance.
2. **Configure replication and transport users**. Create users with specific, restricted access rights for building replication content (Author environment) and for receiving content (Publish environments) instead of using the admin user.
3. **Disable WebDAV**. CRX and CQ come with WebDAV support that lets you display and edit the repository content. Setting up WebDAV gives you direct access to the content repository through your desktop. WebDAV should be disabled on the publish environment.
4. **Restrict Access via the Dispatcher**. The Dispatcher filter can be used to allow or deny external access to specific areas of CQ. To protect your instance you should configure the Dispatcher to restrict external access as far as possible.
5. **Protect against CrossDSite Scripting (XSS)**. Cross\site scripting (XSS) allows attackers to inject code into web pages viewed by other users. This security vulnerability can be exploited by malicious web users, which can bypass access controls. CQ applies the principle of filtering all user\supplied content upon output. Additionally, a web application firewall can be configured to add protection.
6. **Preventing Denial of Service (DoS) Attacks**. A denial of service (DoS) attack is an attempt to make a computer resource unavailable to its intended users, and it's often done by overloading the resource. There are a few methods that can be used with CQ to help prevent such attacks.
7. **Default Access to User Profile(s) is everyone**. By default, everyone (the built\in group) has read access to all user profile(s). If such access is not appropriate for your installation you can change these default settings.
8. **Issues with CrossDSite Request Forgery (CSRF)**. This is a security issue from the CRX Security Checklist. To address known security issues with (CSRF) in CRX WebDAV and Apache Sling you can configure the Referrer filter.
9. **Disable the CQ WCM Debug Filter on production systems**. This is useful when developing as it allows the use of suffixes, but should be disabled on a production instance to ensure performance and security.

Day 5:

Sites and Mobile

Today's content:

- Mobile
- MSM.

Mobile

It is increasingly important that websites offer specific “views” for mobile devices, where these “views” typically are separate sites that share some content with the “normal” website. CQ assists you in creating mobile websites: when authoring a mobile page, the page is displayed in a way that emulates the mobile device, called an **emulator**. When authoring the page, you can switch between several emulators to get a real view of what the end\user will see when accessing the page.

AEM can also help make seamless updates to native mobile applications using the MSM. This helps cut down on the time and effort required to get application store approvals each time a new version of the application comes out.

*Mobile devices are usually divided into **groups**, but these groups can be added and modified to fit the site needs. For every device group, users can define user agents, device capabilities, screen resolution and other variables, and be matched to a particular content page.*

The Java packages containing the mobile classes are:

- **com.day.cq.wcm.mobile.api** \ defines MobileConstants.
- **com.day.cq.wcm.mobile.api.device** \ defines Device, DeviceGroup and DeviceGroupList.
- **com.day.cq.wcm.mobile.api.device.capability** \ defines DeviceCapability.
- **com.day.cq.wcm.mobile.api.wurfl** \ defines **WurflQueryEngine**.
- **com.day.cq.wcm.mobile.core** \ defines MobileUtil, which provides various utility methods revolving around WCM Mobile.

The mobile page development is just as simple as the regular web page component development, but using a different *sling:resourceSuperType* hierarchy. This process uses the CQ mobile page component (***/libs/wcm/mobile/components/*page**):

- It adds the Mobile tab to the page properties dialog.
- Through its head.jsp, it retrieves the current mobile device group from the request and if a device group is found, uses the group's drawHead() method to include the device group's associated emulator init component (only in author mode) and the device group's rendering CSS. This CSS are located under ***/etc/mobile/groups***.

AEM Mobile integrates a Wireless Universal Resource File (WURFL), which is an XML configuration file that resides in the AEM repository containing information about the capabilities and features of many mobile devices. WURFL is updated frequently as new devices emerge. When a content request is made, the user agent string is instantly matched against the WURFL data in the repository, and only the corresponding set of capabilities for that device (varying Ajax capabilities, supported HTML, screen resolution, and so on) are used. Thus, when device matching is done, mobile users receive the optimal content and application features for that specific device.

Exercise 25

- Follow the instructions on the “CQ5 Mobile.pdf” document.

Sources

- OSGi
  - http://en.wikipedia.org/wiki/OSGi
  - http://soa.sys\con.com/node/492519?page=0,2
- Sling
  - http://sling.apache.org/
- JCR
  - http://www.ibm.com/developerworks/java/library/j\jcr/
- General
  - http://blogs.adobe.com/mtg/
  - http://www.netcentric.biz/2012/09/features\i\like\about\cq5\content\packages/
  - http://dev.day.com/microsling/content/blogs/main/tarpm.html
- Mobile
  - http://www.adobe.com/it/special/eseminar\enterprise/pdf/CQ5_Mobile.pdf
  - http://dev.day.com/docs/en/cq/current/developing/mobile.html
  - http://dev.day.com/docs/en/cq/current/wcm/mobile.html
- Performance and Optimization
  - http://dev.day.com/docs/en/cq/current/deploying/performance.html
- General sources
  - http://www.wemblog.com/
  - http://www.cqblueprints.com/xwiki/bin/view/Main/

Day 6:

Build AEM Project with Maven

Today's content:

- Requirements
- Configuring the Maven settings.xml file
- Project Structure.
    - Bundle Module
    - Content Module

Requirements

To set up a development environment based on Maven the following software must be installed before starting:

- Apache Maven 3.x or higher
- AEM 6.0 or higher
- Oracle Java SE 7

Configuring the Maven settings.xml file

Most of the time, We work with own maven repositories to handle the maven dependencies, that cause that maven can find the Adobe AEM dependencies when you create an AEM projects, to fix that you can add the adobe repositories to the settings.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository>${user.home}/.m2/repository</localRepository>
  <interactiveMode>true</interactiveMode>
  <profiles>
        <profile>
        <id>adobe-public</id>
        <activation>
                <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
                <releaseRepository-Id>adobe-public-releases</releaseRepository-Id>
                <releaseRepository-Name>Adobe Public Releases</releaseRepository-Name>
                <releaseRepository-URL>
                        http://repo.adobe.com/nexus/content/groups/public
                </releaseRepository-URL>
        </properties>
        <repositories>
                <repository>
                        <id>adobe-public-releases</id>
                        <name>Adobe Basel Public Repository</name>
                        <url>http://repo.adobe.com/nexus/content/groups/public</url>
                        <releases>
                                <enabled>true</enabled>
                                <updatePolicy>never</updatePolicy>
                        </releases>
                        <snapshots>
                                <enabled>false</enabled>
                        </snapshots>
                </repository>
        </repositories>
```

```
        <pluginRepositories>
                <pluginRepository>
                        <id>adobe-public-releases</id>
                        <name>Adobe Basel Public Repository</name>
                        <url>
                                http://repo.adobe.com/nexus/content/groups/public
                        </url>
                        <releases>
                                <enabled>true</enabled>
                                <updatePolicy>never</updatePolicy>
                        </releases>
                        <snapshots>
                                <enabled>false</enabled>
                        </snapshots>
                </pluginRepository>
        </pluginRepositories>
        </profile>
 </profiles>
 <activeProfiles>
        <activeProfile>adobe-public</activeProfile>
 </activeProfiles>
</settings>
```

Project Structure

For this tutorial We will use Adobe Maven Archetypes to make setup of new projects. We will use the **multimodule-content-package-archetype** to create an initial structure for our maven project. In the following example:

```
mvn -Padobe-public archetype:generate \
   -DarchetypeRepository=http://repo.adobe.com/nexus/content/groups/public/ \
   -DarchetypeGroupId=com.day.jcr.vault \
   -DarchetypeArtifactId=multimodule-content-package-archetype \
   -DarchetypeVersion=1.0.2
```

Note: *the break line make it reading simple, but it goes in the same line.*

Then Maven will request you complete some variables to create the project:

1. **groupid:** prodigious-aem-training
2. **artifactId:** aemtraining
3. **version:** 1.0-SNAPSHOT
4. **package:** com.prodigious.aemtraining
5. **appFolderName:** aem-training
6. **artifactName:** "AEM Training"
7. **cqVersion:** “5.6.1”
8. **packageGroup:** “Prodigious”

you can specify the properties when you run the command, for example if you need to specify the CQ version:

```
...
-DarchetypeArtifactId=multimodule-content-package-archetype \
-DarchetypeVersion=1.0.2 \
-DcqVersion="6.2"
```

Maven will create a folder for the project, move into the project folder:

$ cd aem-training

Finally, you just need to build and install the project into AEM instance, now you can see the project into the CRXDE.

$ mvn -PautoInstallPackage install

Bundle Module

The bundle module is set up to contain OSGi Services required in your project. This Maven Module encapsulated all the components that may has the project, Its Module come with basic Service and filters.

Code Structuure:

- *src/main/java*
  - *com.prodigious.aemtraining*
    - HelloService.java
    - package-info.java
  - *com.prodigious.aemtraining.impl*
    - HelloServiceImpl.java
  - *com.prodigious.aemtraining.impl.filters*
  - LoggingFilter.java
- *src/test/java*
  - *com.prodigious.aemtraining*
    - SimpleUnitTest.java

Content Module

The content module is set up to build an AEM content package which combines:

- components
- templates
- sample code
- configuration

There into Java source in this module, Into the folder src/main/content you can find the the folder **jcr_root** where all the Javascript, CSS and Content configuration, This module has defined a maven profile **autoInstallPackage**, which is the profile to deploy the application into AEM after it has been built.

Note: when you are creating components with Eclipse, you need to deploy the project every time that you want to see the changes into the AEM instance, in the otherwise it common to modify the project into the CRXDE Web Editor, it does not update the filesystem project. To update the filesystem project you need to configure the VLT Tool.

Day 6:

Sightly

Today's content:

- What is Sightly?
    - Advantages
    - Disadvantages
- Operators and block statements
    - Examples with REPL
- Sightly vs. JSP
- Integration with Java
- Other examples
    - Use of templates
    - Integration with JavaScript

What is Sightly?

Sightly is an HTML templating language, introduced with AEM 6.0. It takes the place of JSP (Java Server Pages) and ESP (ECMAScript Server Pages) as the preferred templating system for HTML. The name “Sightly” (meaning “pleasing to the eye”) highlights its focus on keeping your markup beautiful, and thus maintainable, once made dynamic.

The Sightly advangares are:

- Cross-Site-Scripting (XSS) Security by Default.
- Keeps the markup clean and maintainable.
- Sightly is a valid HTML5 file.
- Layer Separation (view and Logic).
- Easy way to start developing from mockups.
- Sightly components can work together with JSP or ESP components.

Operators and block statements

Sightly expressions are used to access the data structures that provide the dynamic elements of the HTML output. A Sightly expression is delimited by **${** and **}**.

Comments

```
<!--/* This is a Sightly comment */-->
<!-- HTML ${currentPage.title} comment -->
```

Literals Expressions

Boolean

```
${true} ${false}
```

Numbers

There is only one number type: integers

```
${2015}
```

String

They represent textual data, and can be **single** or **double** quoted:

```
${'Simple quotes Text'}

${"Double quotes Text"}
```

you can also escape sequences and Unicode sequiences

```
${'it\'s great escape sequence'}

${'it\0027s great unicode sequence'}
```

Operator

Operator	Sightly	Description
Or	${varOne || varTwo}	can be used to provide string defaults
And	${varOne && varTwo}	
Not	${!isNumber}	
Conditional	${varChoice ? varOne : varTwo}	
Comparison	${varOne <|>|<=|=> varTwo}	
Equality and inequality	${varOne ==|!= varTwo}	
Grouping parentheses:	${varOne && (varTwo || varThree)}	

Expression Options

Expression options can act on the expression and modify it, or serve as parameters when used in conjunction with block statements.

```
${myVar @ optOne}
```

Multiple options are separated by commas:

```
${myVar @ optOne, optTwo=bar}
```

Block Statements

Sightly block statements are custom data attributes added directly to existing HTML. This allows easy and unobtrusive annotation of a prototype static HTML page.

- **Text**: Replaces the content of its host element with the specified text.

```
<h1 data-sly-text="${currentPage.title}">Lorem ipsum</h1>
```

- **Attribute**: Adds attributes to the host element.

```
<a href="#" data-sly-attribute.href="${currentPage.path}.html">Link</a>
```

- **Element**: Replaces the element name of the host element. This accepts only common tags name.

```
<h1 data-sly-element="h3">text</h1>
```

- **Unwrap**: Removes the host element from the generated markup while retaining its content.

```
<div data-sly-unwrap="true">content unwrap</div>
```

- **Use** : Initializes a helper object (defined in JavaScript or Java)

```
<div data-sly-use.nav="navigation.js">${nav.foo}</div>
<div data-sly-use.nav="org.path.Navigation">${nav.foo}</div>
```

- **Test**: Conditionally removes the host element and it's content

```
<p data-sly-test="${isShown}">text</p>
<p data-sly-test.abc="${a || b || c}">is true</p>
<p data-sly-test="${!abc}">or not</p>
```

- **List**: Repeats the content of the host element for each enumerable property

```
<dl data-sly-list="${currentPage.listChildren}">
   <dt>index: ${itemList.index}</dt>
   <dd>value: ${item.title}</dd>
</dl>
<dl data-sly-list.child="${currentPage.listChildren}">
   <dt>index: ${childList.index}</dt>
   <dd>value: ${child.title}</dd>
 <dd>isFirst: ${childList.first}</dd>
</dl>
```

- **Include:** include another resource given by the attribute value

```
<section data-sly-include="path/to/template.jsp"></section>
```

- **Resource**: Includes the result of rendering the indicated resource

```
<article data-sly-resource="path/to/resource"></article>
<footer data-sly-resource="${'footer' @ resourceType='prodigious/components/global/content/footer-sly' }">
```

Context Objects

A default set of objects is provided within the execution context of every Sightly template. These objects are in addition to any that may be introduced through the Use-API.
Note: *Full objects supported: https://docs.adobe.com/docs/en/aem/6-0/develop/sightly/global-objects.html*

Enumerable Objects

These objects provide convenient access to commonly used information.

Variable Name	Description
properties	List of properties of the current Resource. Backed by org.apache.sling.api.resource.ValueMap.
pageProperties	List of page properties of the current Page. Backed by org.apache.sling.api.resource.ValueMap.
inheritedPageProperties	List of inherited page properties of the current Page. Backed by org.apache.sling.api.resource.ValueMap.

Java-backed Objects

These objects provide the standard AEM execution context (as global.jsp does for JSPs, for example). Each object is backed by the corresponding Java object.

Variable Name	Description
currentNode	javax.jcr.Node
currentPage	com.day.cq.wcm.api.Page
currentSession	javax.servlet.http.HttpSession

Moving from JSP to Sightly

Slightly offers better feature to reduce the development time and also do readable code using a simple HTML5 file.

Render Element:

Sightly

```
<a href="${properties.link || '#'}" title="${properties.jcr:title}">
${properties.jcr:description @ context='html'}
</a>
```

JSP

```
<a href="<%= xssAPI.getValidHref(properties.get("link", "#")) %>" <%
 String title = properties.get("jcr:title","");
 if (title.length() > 0) {
          %> title= "<%= xssAPI.encodeForHTMLAttr(title) %>"<%
 } %>>
 <%= xssAPI.encodeForHTML(properties.get("jcr:description","")) %>
</a>
```

Or operator to render Title

Sightly

Slightly evaluate if the value is null or empty to pass to the next variable, if It is not empty the evaluator return this value and stop the evaluation expression.

JSP -JSTL

```
<%
 String title = properties.get(NameConstants.PN_TITLE, String.class);
 if (title == null || title.equals("")) {
 title = currentPage.getNavigationTitle();
 }
 if (title == null || title.equals("")) {
    title = currentPage.getPageTitle();
 }
 if (title == null || title.equals("")) {
    title =  currentPage.getTitle();
 }
%>
 <h1><%= title %></h1>
```

WCMMode verification

Sightly

```
<div data-sly-test.valid="${properties.title}">${properties.title}</div>
<div data-sly-test="${!valid && wcmmode.edit}">Please enter a title for this page</div>
```

JSP

```
<c:set var="isEditMode" value="<%= (WCMMode.fromRequest(request) == WCMMode.EDIT %>" />
<c:choose>
        <c:when test="${not empty properties.title}">
        <div>
        <span>${properties.title}</span>
        </div>
        </c:when>
        <c:otherwise>
        <c:if test="${isEditMode}">
        Please enter a title for this page
        </c:if>
        </c:otherwise>
</c:choose>
```

Integration with java

Sightly can access helper methods in Java class. This allows to encapsulate business logic to the Java code, while the Sightly code deals only with direct markup production.

Define a AEM Model

In AEM, We can define REST resource defining the a model with the Annotation *Model* from the sling framework.

Sling Model

```
package com.prodigious.aemtraining.model;
...

@Model(adaptables = Resource.class)
public class Modal {

   @Inject
   @Optional
   private String title;

   @Inject
   @Optional
   private String message;
...
}
```

JSON Response

accessing to sling json you can get the JSON Info:

```
{
        "sling:resourceType": "my-example/component/modal",
        "title": "Example Title",
        "message": "This is Some Example Content."
}
```

HTML Response

Under the Component we can create a simple HTML with the tag ***<sly data-sly-use.modal/>*** to import the java Class, then you can use a the **modal** key to access the methods

```
<sly data-sly-use.modal="${'com.prodigious.aemtraining.model.Modal'}" ></sly>
<div>
   <h1>${modal.title}</h1>
   <p>${modal.description}</p>
</div>
```