COREMEDIA

www.coremedia.com

CoreMedia Training-Center

Web Application Development

Web Application Development Training

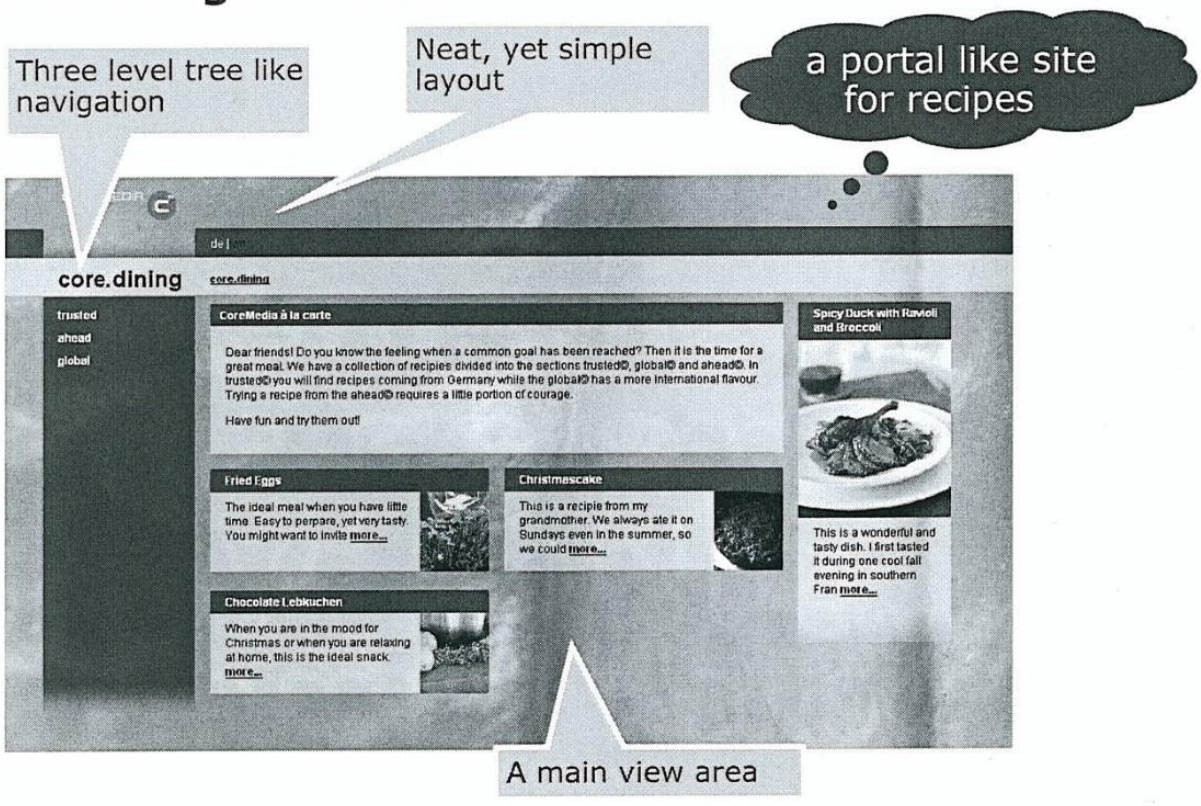
Developing Web Applications with the Content Application Engine

Your Assignment: Create a site guided by a click dummy

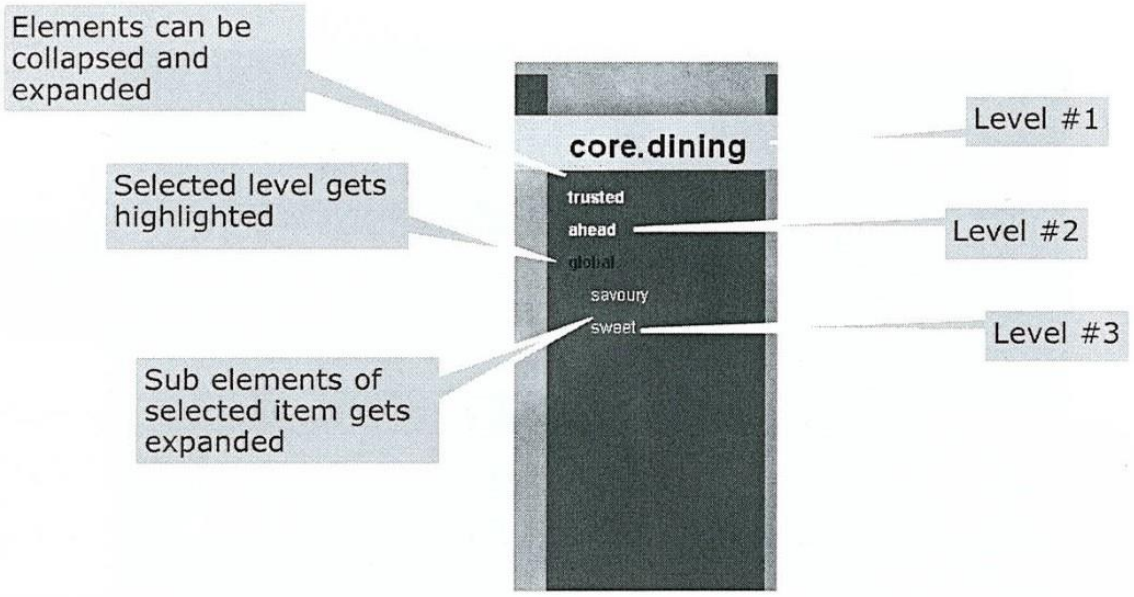
Your Assignment

- → You are asked to develop a cooking web site
- → You should use the CoreMedia Content Application Engine (CAE)
- → The style guide is defined through a click dummy
- → In this first step you have to develop a sub set only
  - → display in different views:
    - → text
    - → images
  - → basic layout
  - → basic left hand tree like navigation

core.dining: Overview



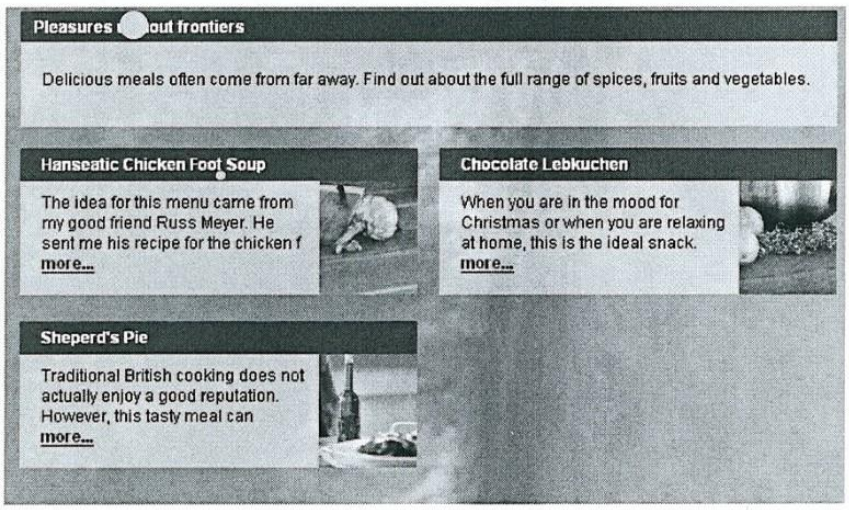
core.dining: Navigation



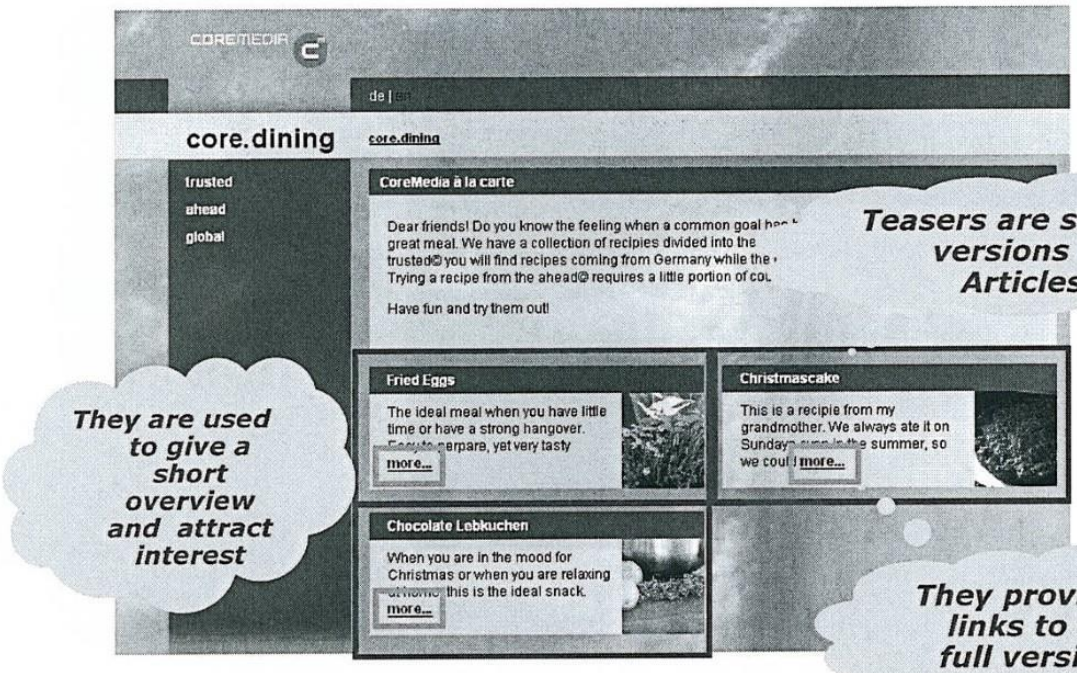
core.dining: Main area

a two column list of teasers

composed of a large size intro and ...



core.dining: Teasers



This means Teaser is same as article and we do not need a doc type for teaser

COREMEDIA

7

COREMEDIA

You might need some planning now

8

Proposed steps towards the solutions

Task	Technical Solution
Step #1 Content type model	→ Use the pre-built content type model already provided
Step #2 Basic rendering and layout	→ Basic Content bean generation and configuration → Basic JSP programming
Step #3 Rendering the complex navigation	→ Extending content beans with business logic → Advanced JSP programming → Programming Request and Link Handlers
Step #4 Making Layout Configurable	→ Delivering CSS and JavaScript from Content Repository

Required knowledge for that task

Knowledge you should already have

→ Java

→ Servlet-API, JSP, JSTL, EL

→ Eclipse or similar IDE

... and optional, but recommended

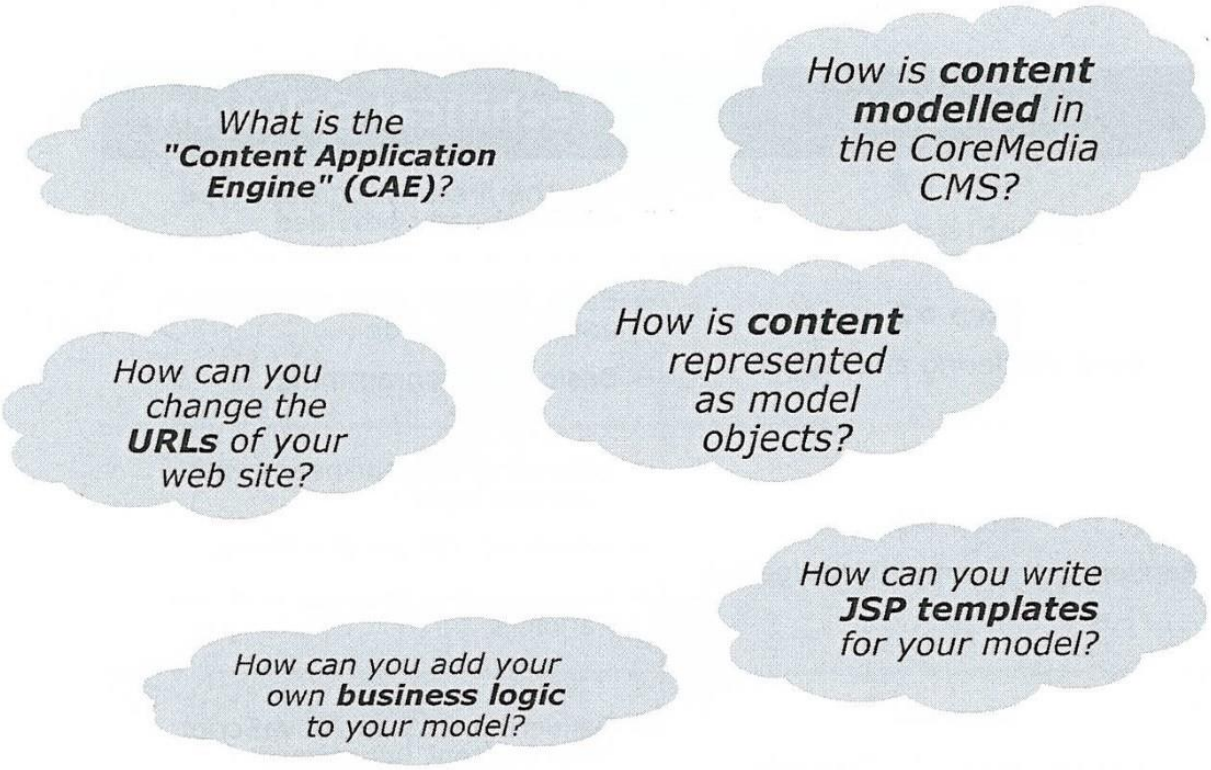
→ XML and XML processing (SAX, DOM, javax.xml)

→ Maven

→ Spring

...and things you will learn in this course...

What can this course teach you?



Agenda

1. Framework Basics
2. Content Type Model and Content Beans
3. Template Development
4. Business Logic
5. Request and Link Handlers
6. Page layouts and CSS
7. Modularization
8. Advanced View Programming
9. Caching with Dataviews
10. Integration of External Content

Day 1 ***Day 2*** ***Day 3***

Agenda

1. Framework Basics
2. Content Type Model and Content Beans
3. Template Development
4. Business Logic
5. Request and Link Handlers
6. Page layouts and CSS
7. Modularization
8. Advanced View Programming
9. Caching with Dataviews
10. Integration of External Content

CoreMedia

1. Framework basics

→ Introducing the CAE

→ Introducing Spring and Spring MVC

→ CAE Implementation of Spring MVC

→ Maven Workspace

CoreMedia

Introducing CAE: Content Application Engine

The main rendering component for content management and delivery

Based on

→the Spring framework for general configuration and dependency injection

→Spring MVC for a state of the art Model-View-Controller web framework

→JSP standard

Features

→independent layers

→highly effective and configurable caching

→can render any bean

CAE Highlights

Absolutely no Java code in JSPs

Standard Taglibrary (JSTL) can and should be used

Just a few CoreMedia tags

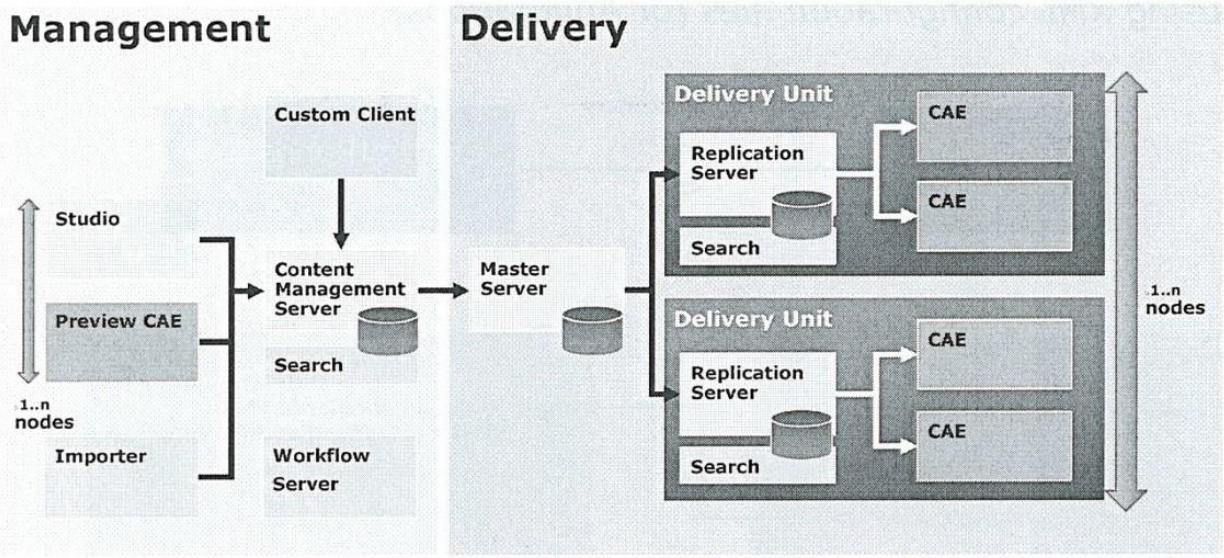
JSPs are used for rendering

CMS Content gets mapped to beans

Configuration is completely done in Spring

Beans can be directly displayed in JSTL based JSPs

CoreMedia Architecture (coarse grain)

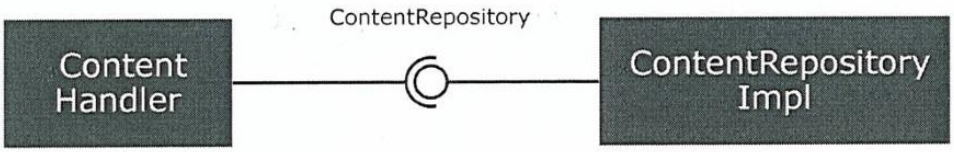


1. Framework basics

- Introducing the CAE
- Introducing Spring and Spring MVC
- CAE Implementation of Spring MVC
- Maven Workspace

Spring Framework Basics

Main Idea: *Assembling applications by connecting Java-Beans using XML configuration files (or Annotations).*



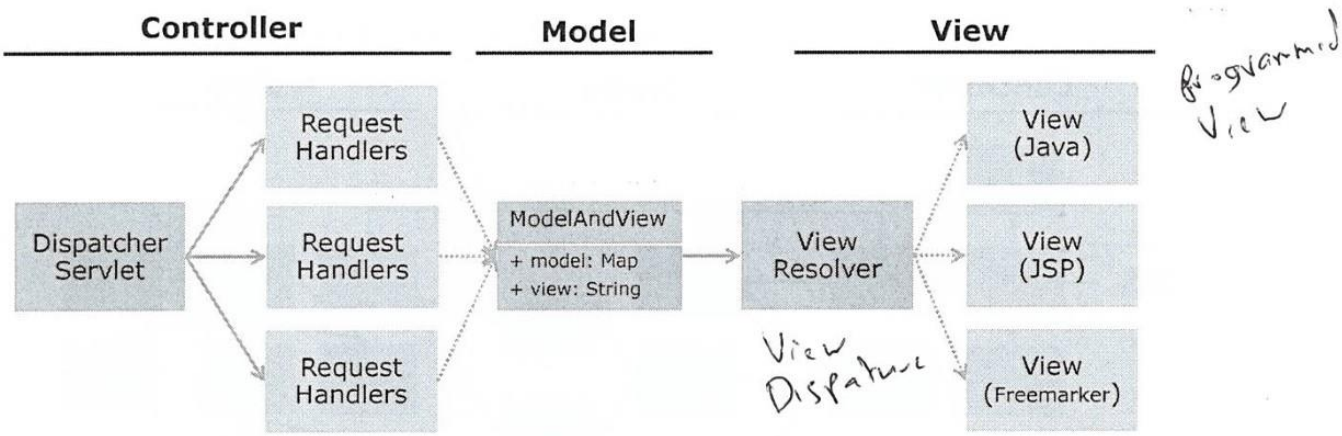

```
<bean id="contentHandler"
      class="c.c.c.h.ContentHandler">

  <property name="repository"
      ref="contentRepository" />

</bean>
```

```
<bean id="contentRepository"
      class="...ContentRepositoryImpl" />
```


Spring MVC Framework

- → MVC architecture without predetermined view technique
- → Built around central DispatcherServlet with full access to Spring application contexts
- → Features form and automatic error handling through custom tag library
- → Allows for interceptors (e.g. for security checks)
- → Supports i18n
- → Basis for CoreMedia CAE
  - → adds an object oriented view dispatcher
  - → using JSP view technology and beans as model

Spring MVC Framework



- Incoming Requests are **dispatched** to Handlers/Controllers
- based on URL patterns (XML or Annotations)

- **Handlers/Controllers** return a ModelAndView object
- **ViewResolver** is using the ModelAndView to find the appropriate view
- **Views** can be implemented in different ways: Java, JSP, Freemarker, ...

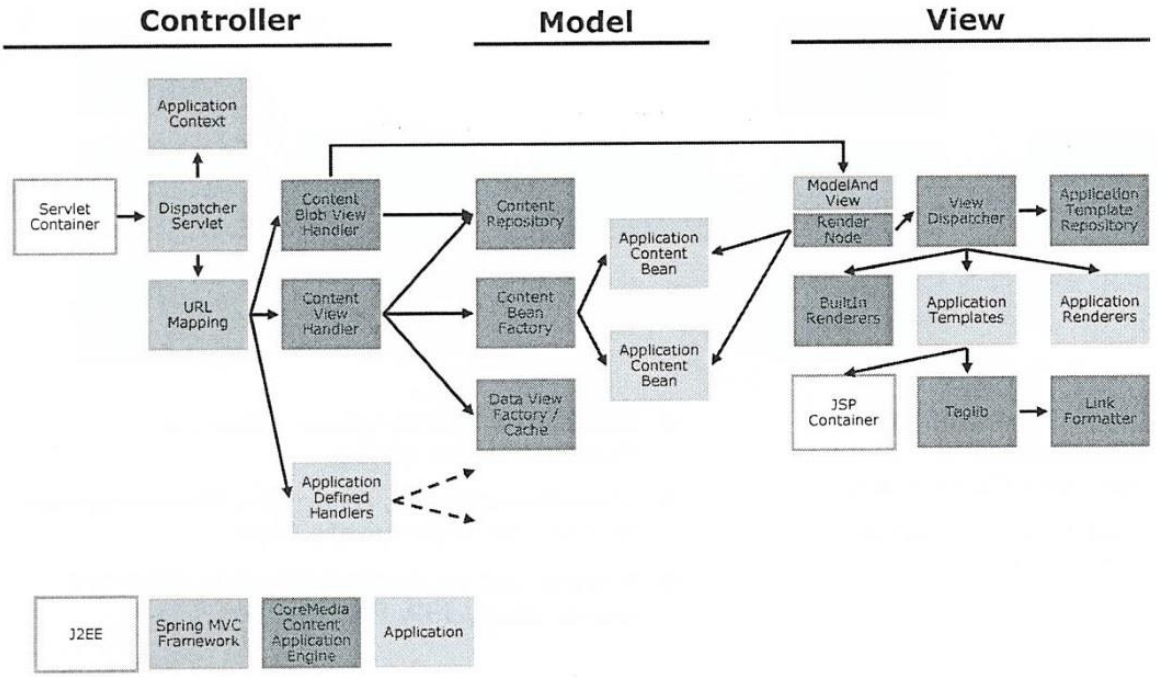
COREMEDIA

1. Framework basics

- Introducing the CAE
- Introducing Spring and Spring MVC
- CAE Implementation of Spring MVC
- Maven Workspace

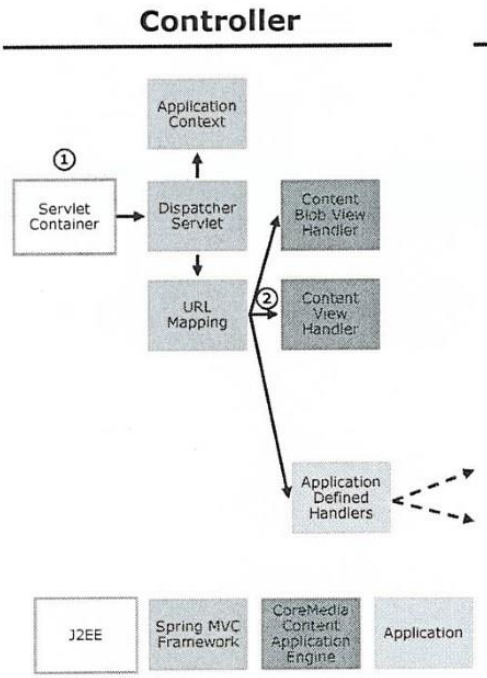
COREMEDIA

CAE architecture (fine grain)



CONTENTMEDIA

A typical CAE request

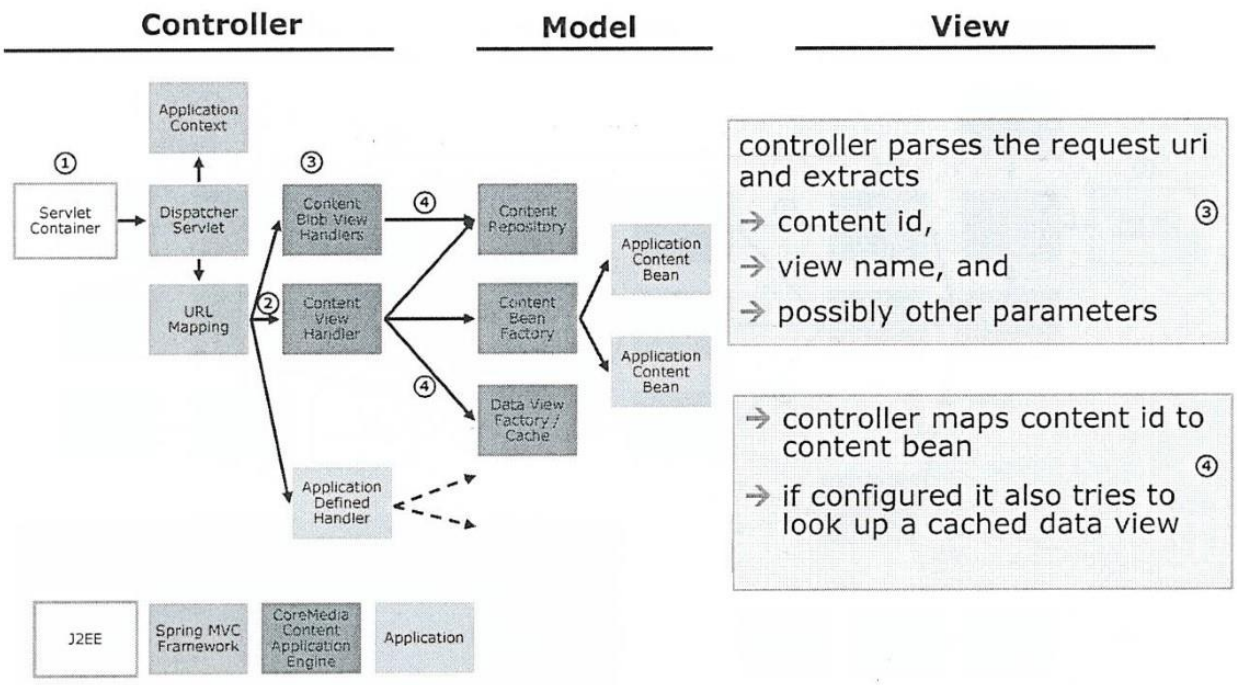


request is passed from servlet container to Spring dispatcher servlet ①

dispatcher servlet dispatches request to a handler using URL mapping ②

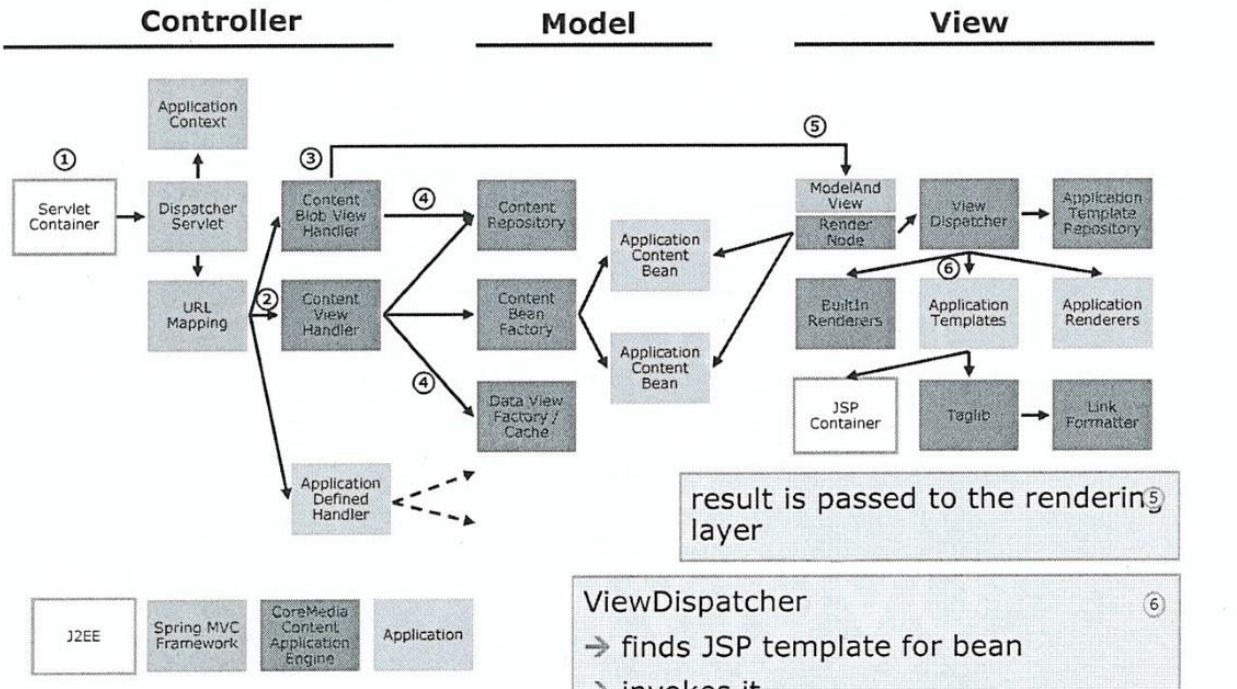
CONTENTMEDIA

A typical CAE request



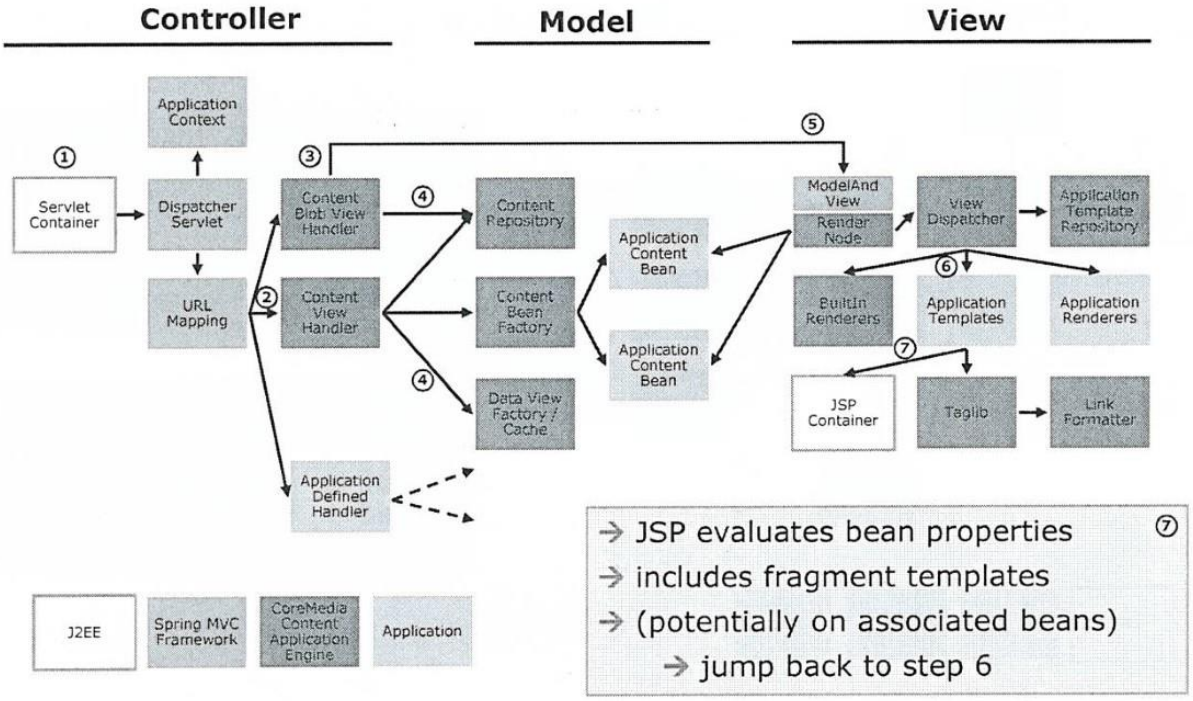
CONTENT

A typical CAE request



CONTENT

A typical CAE request



CONTENTMEDIA

You should receive a color printout of this overview **now**

COREMEDIA

You can use it as a reference throughout the whole course

COREMEDIA

It's time to get more practical now...

COREMEDIA

The project structure, the build and deploy scripts have already been prepared for you.

1. Framework basics

- → Introducing the CAE
- → Introducing Spring and Spring MVC
- → CAE Implementation of Spring MVC
- → Maven Workspace

CoreMedia Project Workspace

- **workspace**
  - **modules**
    The development workspace of the project.
    This is where you develop your CM application
  - **packages**
    The deployment workspace of the project.
    This is where you configure your RPMs. We are not going to use this module in this course.
  - **boxes**
    In here you are able to create Oracle Virtual Boxes running your CM components. We are not going to cover this module in this course.
  - **test-data**
    Content and User data for server import.
  - **workspace-configuration**
    Contains some database start scripts and IDE-specific configuration.
  - *pom.xml*
    Main maven build descriptor
    Default ports and host names are defined in here.

CoreMedia Project Workspace

- **modules**
  - **cae**
    Aggregator module for web application development
  - **cmd-tools**
    Commandline tools for the server components
  - **editor-components**
    Aggregator module for the Site Manager (a.k.a Java Editor)
  - **search**
    Aggregator module for search customizations and custom CAE feeder development
  - **server**
    Aggregator module for content and workflow server customization
  - **studio**
    Aggregator module for Studio customizations
  - **shared**
    Common maven dependencies: database-drivers
  - **extensions**
    The place for project extensions
  - **extensions-config**
    The extension points of the CoreMedia Extension Mechanism.
  - *pom.xml*
    Main maven build descriptor

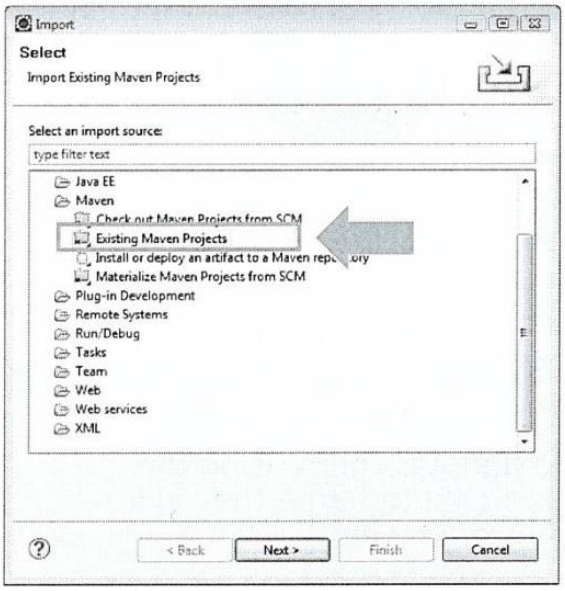
CoreMedia Project Workspace

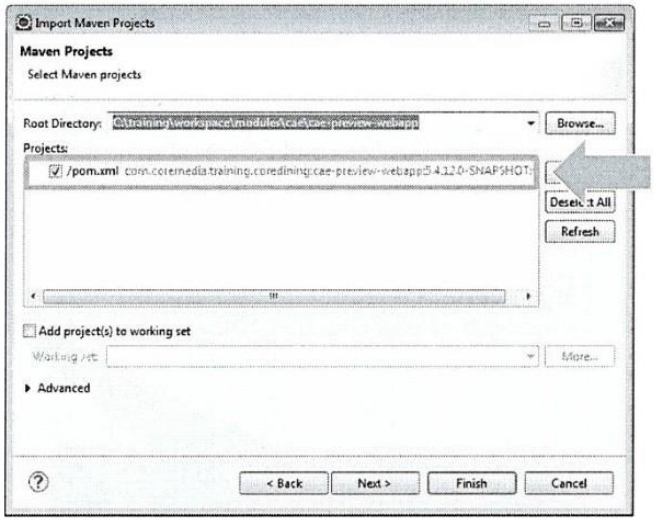
- **cae**
  - **cae-preview-webapp**
    Web application module for the preview web application.
  - **cae-live-webapp**
    Web application module for the live web application.
  - **cae-base-component**
    The component module that bundles all common CAE specific libraries and registers the spring config files (CoreMedia Extension Mechanism)
  - **cae-base-lib**
    This library contains common CAE specific implementation, configuration and templates.
  - **contentbeans**
    Java archive module containing the application wide domain objects (a.k.a ContentBeans) and their configurations.
  - *pom.xml*
    Module specific maven build descriptor

Structure of the cae-preview-webapp module

- **cae-preview-webapp**
  - src/main/java - Source directory for Java implementation
  - src/main/resources - Classpath resources
    - framework/spring - Spring configuration files
  - src/main/webapp - Source directory for the web application
    - WEB-INF/templates - template directory
      WEB-INF/*application.properties* - main configuration file
    - WEB-INF/*logback.xml* - logging configuration file
  - target - Build results
  - *pom.xml* - Maven build descriptor
  - *build.xml* - Training specific ANT script

Importing Maven projects into Eclipse using M2Eclipse





COREMEDIA

Maven goals for building the development workspace

→ Building and installing the workspace or single modules:

```
C:\training\workspace> mvn clean install
```

all modules

→ Building cae-base-webapp and all modules it depends on

```
C:\training\workspace> mvn clean install -pl :cae-preview-webapp -am
```

→ Running the module cae-base-webapp as web application

```
...\cae-preview-webapp> mvn tomcat7:run
```


COREMEDIA

JSTL Basics: EL

- Beans keep their type in the JSP
- you can access their properties using the "." notation
- this expression maps to a call of the beans getTitle() method

```
<body>
<h1>This page displays content of ${self.title}</h1>
</body>
```

- expressions in the JSTL expression language (EL) are enclosed in ${ ... }
- they can be directly embedded into JSP pages

- Beans can be directly accessed from the page context
- address them using the attribute name they were stored with
- in this case the bean was stored in attribute *"self"*

Exercises 1 + 2 + 3

COREMEDIA

Install and start your system, make changes and deploy them

Scheduled time: 30 minutes

COREMEDIA

JSTL Basics: EL

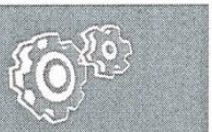
- Beans keep their type in the JSP
- you can access their properties using the "." notation
- this expression maps to a call of the beans getTitle() method

```
<body>
<h1>This page displays content of ${self.title}</h1>
</body>
```

- expressions in the JSTL expression language (EL) are enclosed in ${ ... }
- they can be directly embedded into JSP pages

- Beans can be directly accessed from the page context
- address them using the attribute name they were stored with
- in this case the bean was stored in attribute *"self"*

Exercises 1 + 2 + 3



Install and start your system, make changes and deploy them

Scheduled time: 30 minutes

Agenda

1. Framework Basics
2. Content Type Model and Content Beans
3. Template Development
4. Business Logic
5. Request and Link Handlers
6. Page layouts and CSS
7. Modularization
8. Advanced View Programming
9. Caching with Dataviews
10. Integration of External Content

2. Content Type Model and Content Beans

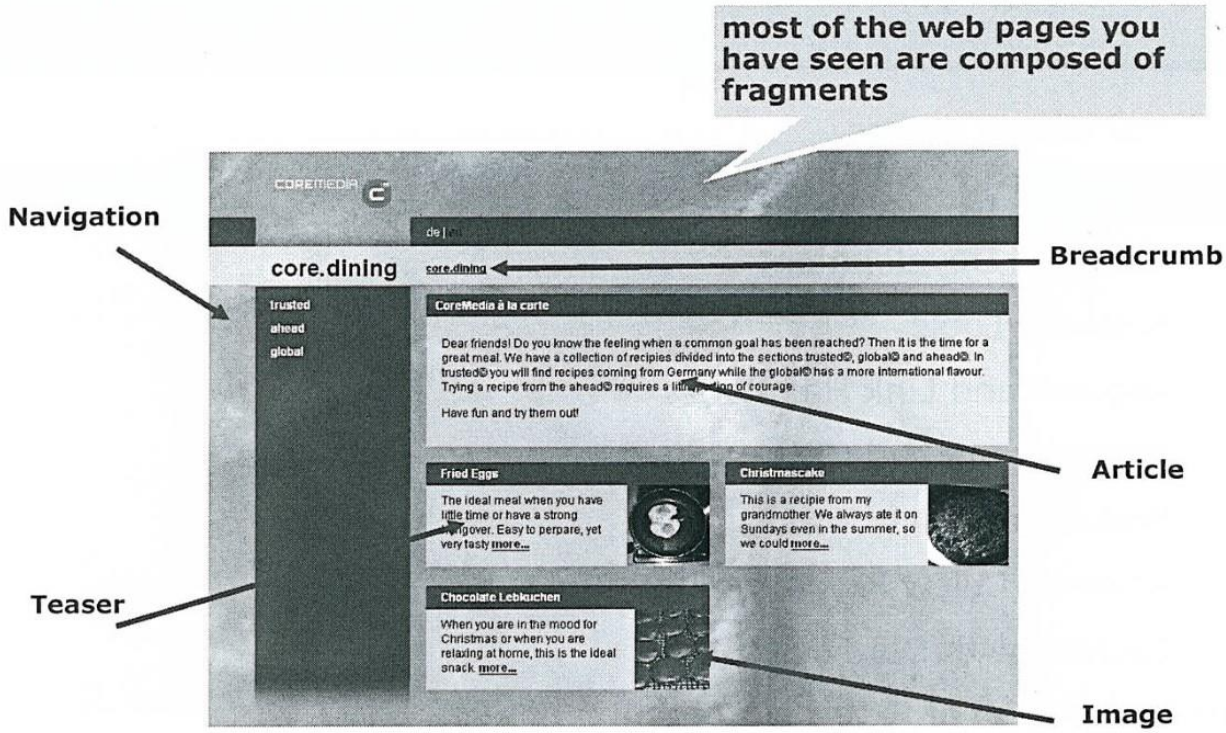
→ From Style Guide to Content Type Model

→ Elements of a Content Type Model

→ The Core.Dining Content Type Model

→ Content Beans

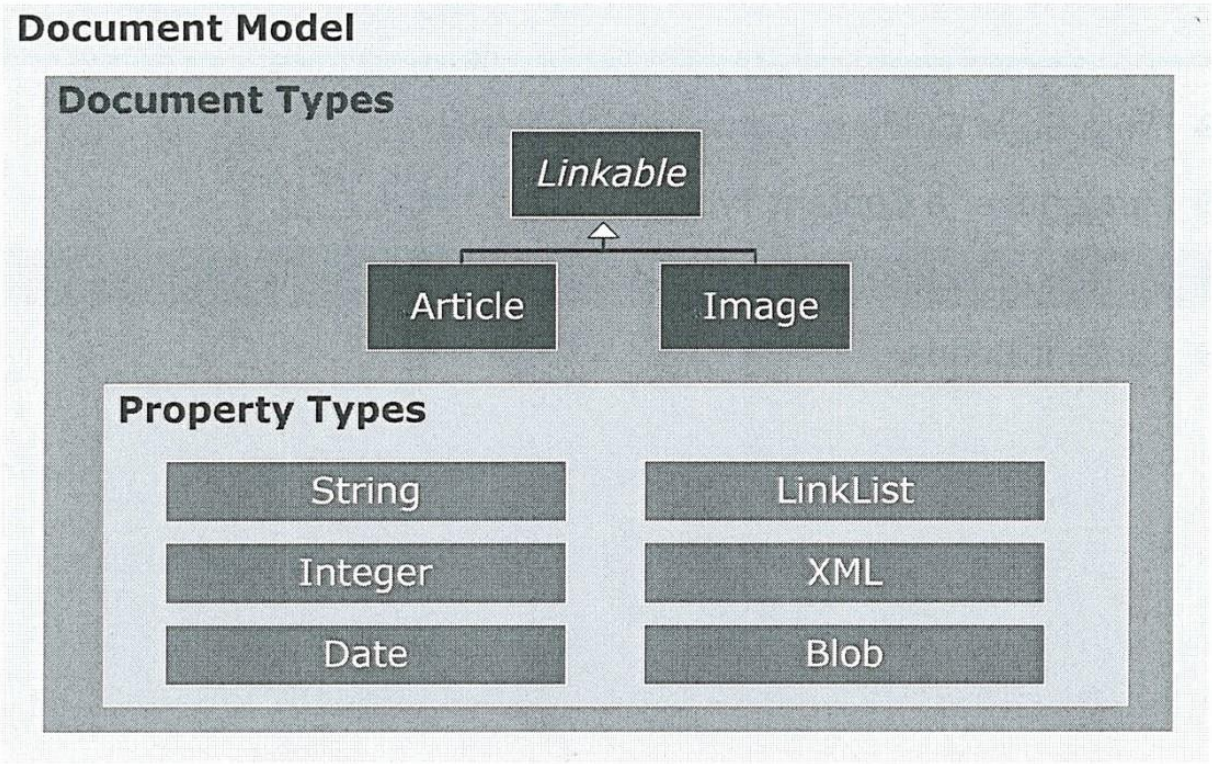
Identifying building blocks of a site



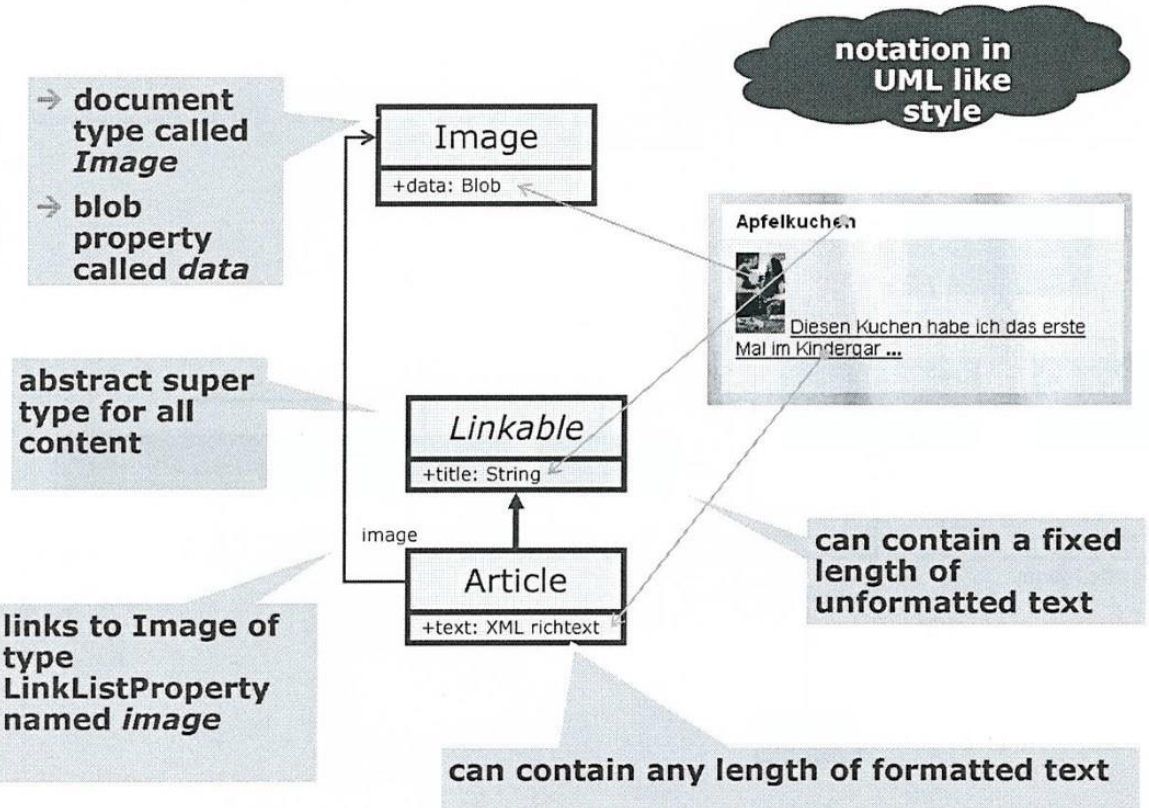
2. Content Type Model and Content Beans

- From Style Guide to Content Type Model
- Elements of a Content Type Model
- The Core.Dining Content Type Model
- Content Beans

Components of a document model



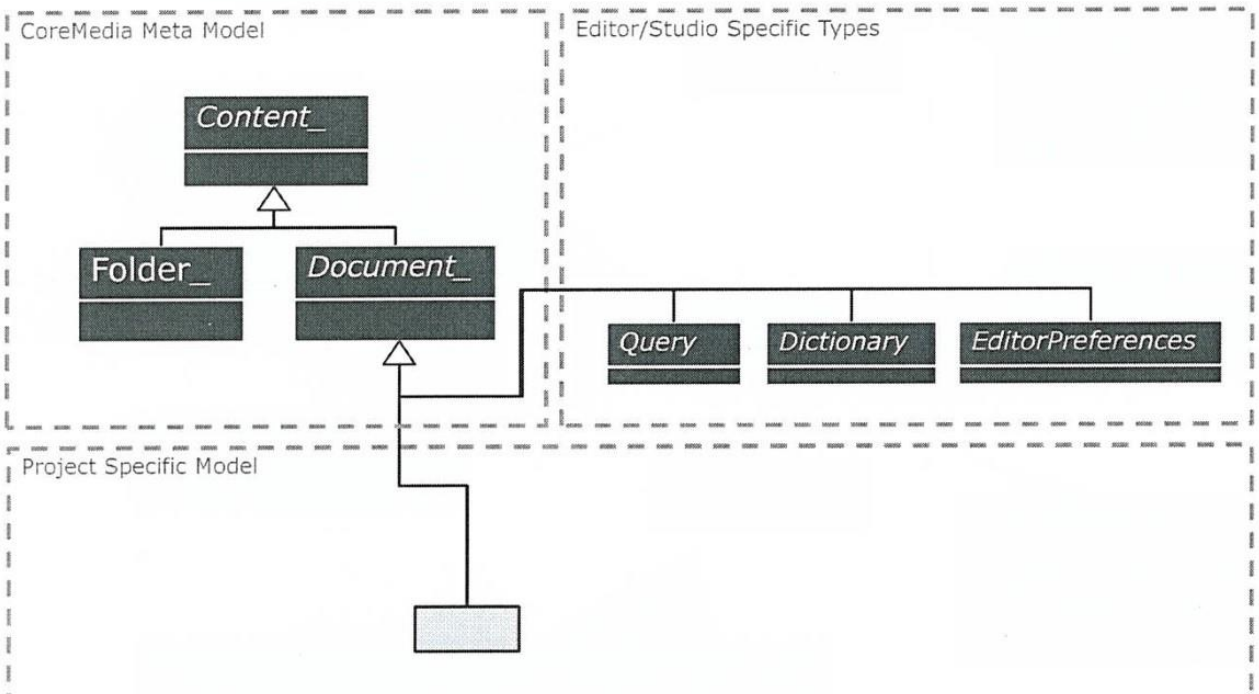
All content data is stored in properties



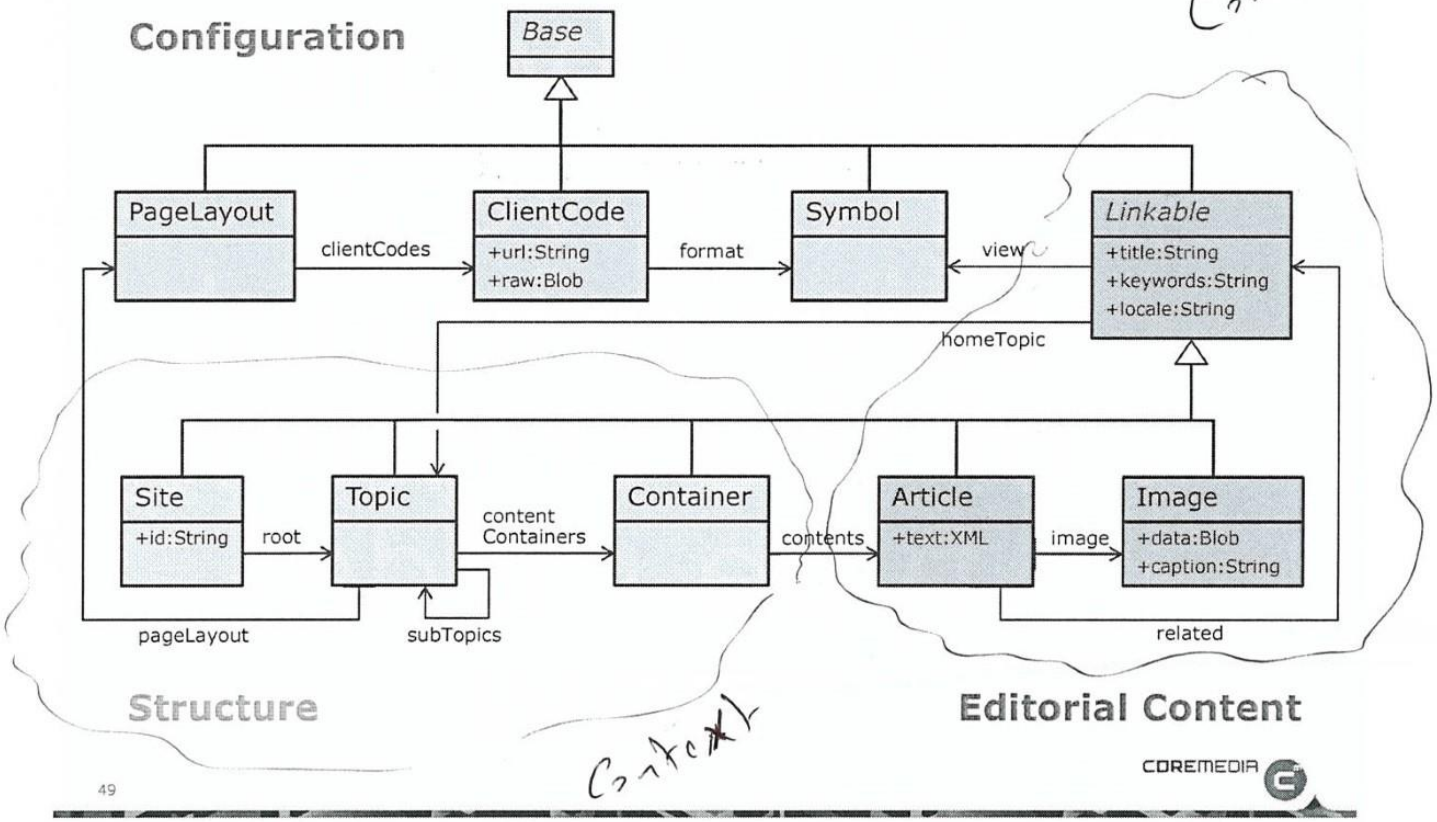
2. Content Type Model and Content Beans

- From Style Guide to Content Type Model
- Elements of a Content Type Model
- The Core.Dining Content Type Model
- Content Beans

CoreMedia Content Model

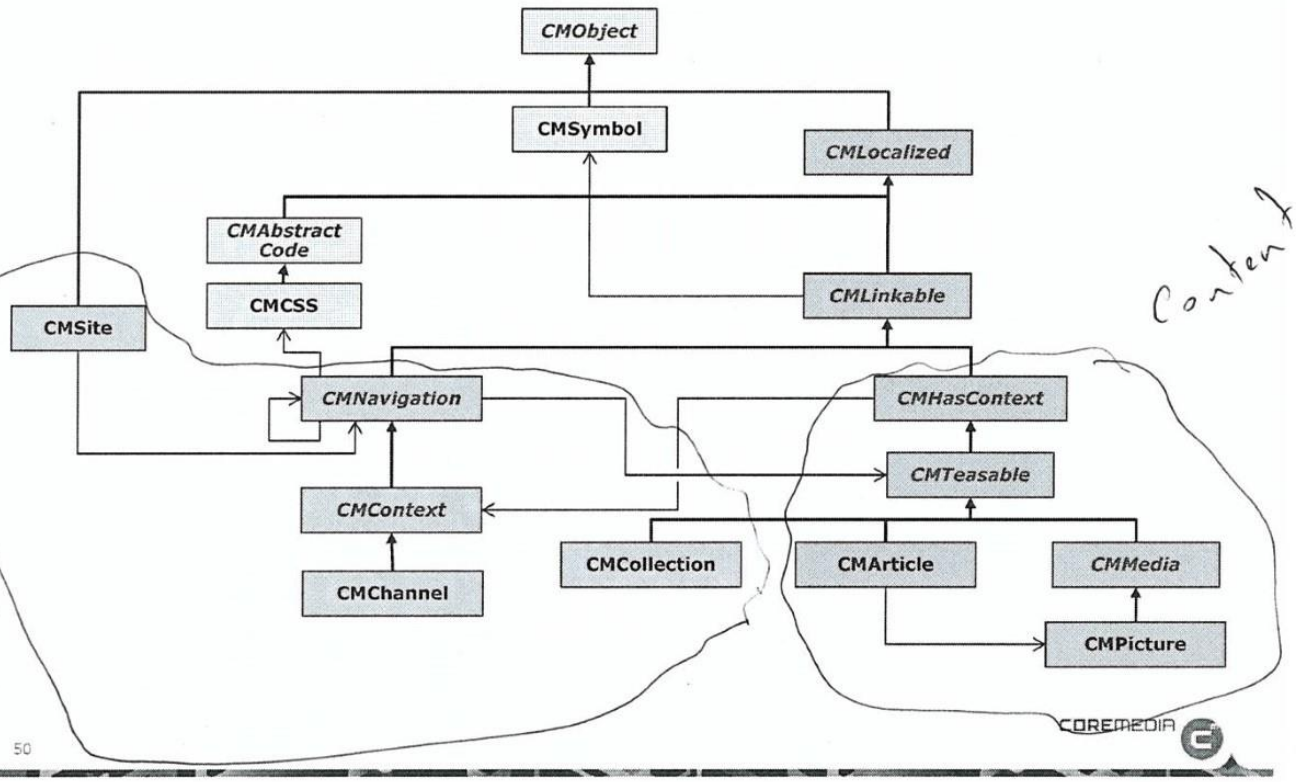


Core.Dining Content Type Model



COREMEDIA

Blueprint Content Type Model (Extract)



COREMEDIA

COREMEDIA

You should receive a color printout of the content type model **now**

COREMEDIA

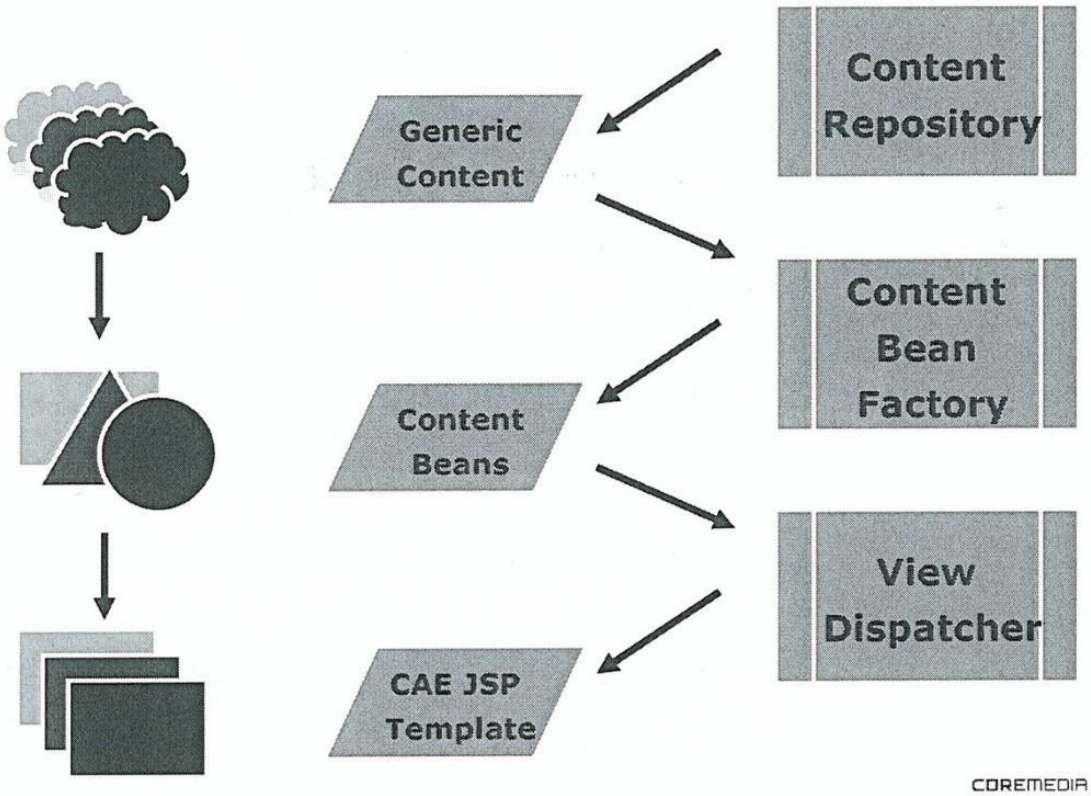
You can use it as a reference throughout the whole course

CAE JSP templates operate on beans. You need to map the content type model to beans.

2. Content Type Model and Content Beans

- From Style Guide to Content Type Model
- Elements of a Content Type Model
- The Core.Dining Content Type Model
- Content Beans

Content beans data flow



General properties of Content beans

- **Content beans are**
  - the domain objects of your web application
  - a typed representation of CoreMedia CMS content
  - the basis of your business and application logic
- **Technical specification**
  - Java beans: they expose their values via standard getters.
  - light-weight: they do not actually *contain* any content, but rather retrieve it on demand
  - dynamic: created by a configurable content bean factory
  - immutable: getters should return unmodifiable values only
- **Content beans can be**
  - automatically generated from your content type model
  - extended with your custom methods
  - accessed from JSPs using JSTL and its EL

Bean Generator

→ Content Beans can be generated using a tool called BeanGenerator

→ The Bean Generator is available as Maven artifact:

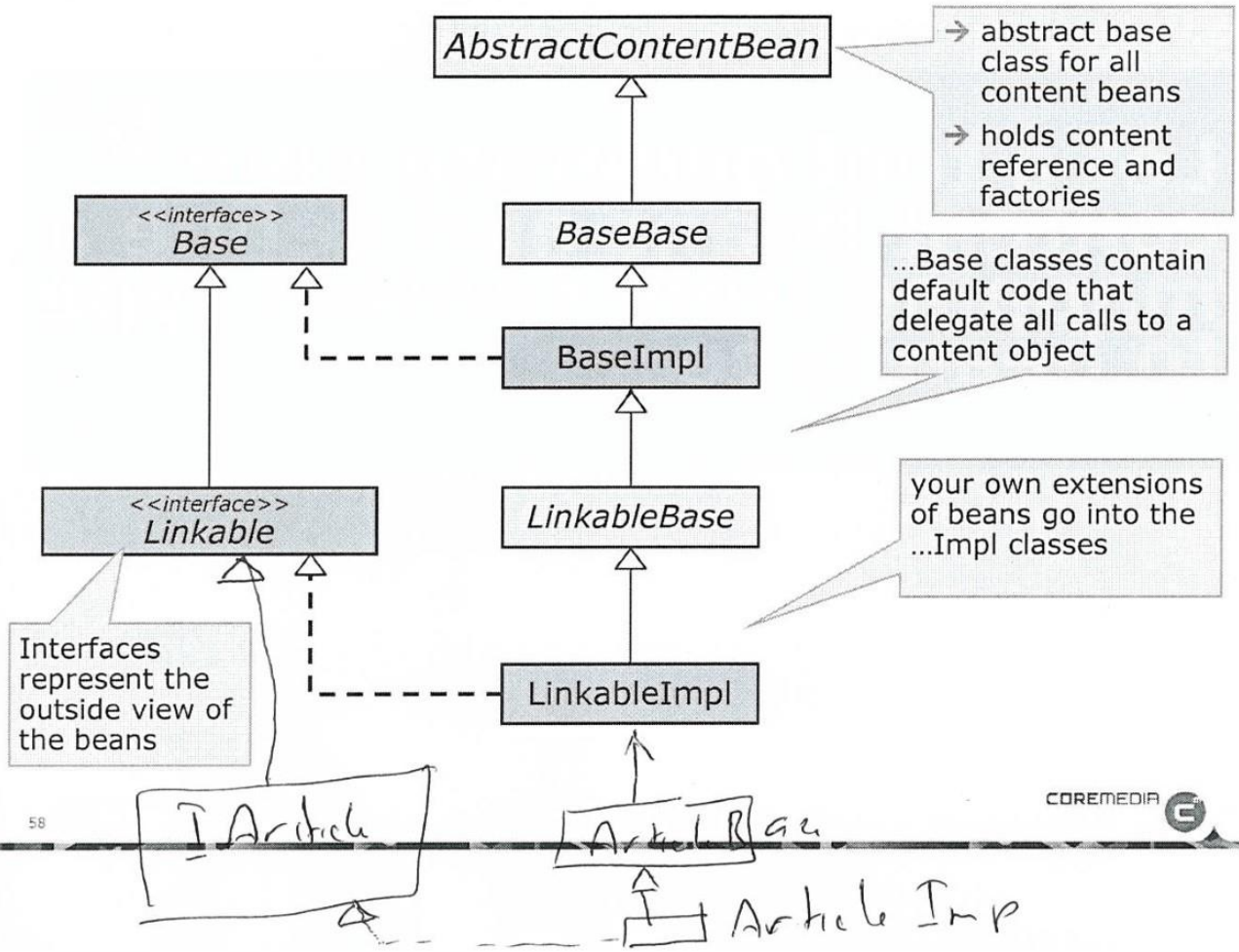
```
<dependency>
   <groupId>com.coremedia.cms</groupId>
   <artifactId>bean-generator</artifactId>
   <scope>compile</scope>
</dependency>
```

→ Calling the BeanGenerator from the command-line:

```
java com.coremedia.objectserver.beans.codegen.impl.BeanGenerator
     -generics
     -o src/java
     -p com.coremedia.corebase.contentbeans
     -d path/to/projectspecific/doctypes.xml
```


COREMEDIA

Content beans: Sample type hierarchy



COREMEDIA

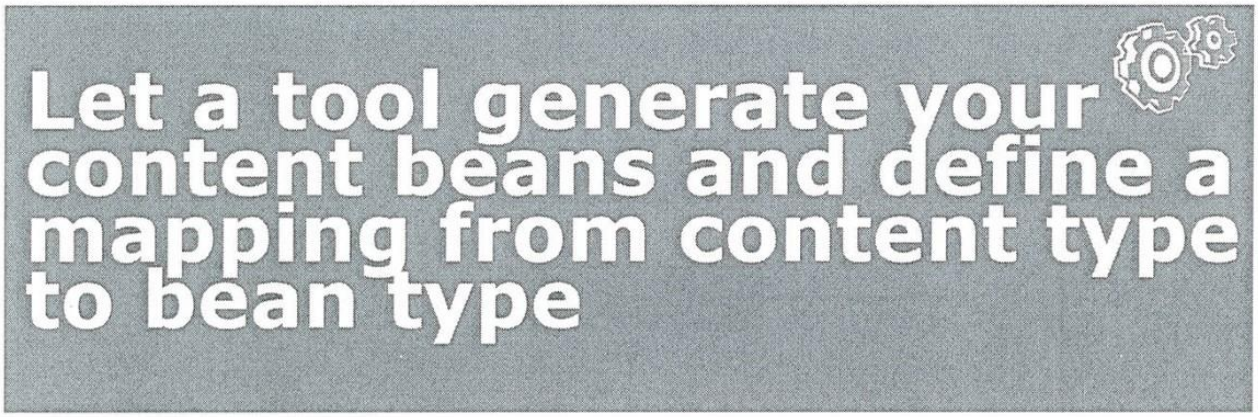
Content Bean configuration

→ Content types of the content repository are mapped to Content Bean implementations.

→ This mapping is defined in a spring configuration file of your web application, e.g. framework/spring/coredining-contentbeans.xml

```
<bean name="contentBeanFactory:Article"
  scope="prototype"
  class="com.coremedia.coredining.contentbeans.ArticleImpl">
</bean>
```


Exercises 4 + 5



Proposed steps towards the solutions

Task	Technical Solution	
Step #1 Content Type Model	→ Use the pre-built content type model already provided	☑
Step #2 Basic rendering and layout	→ Basic Content bean generation and configuration → Basic JSP programming	☐
Step #3 Rendering the complex navigation	→ Extending content beans with business logic → Advanced JSP programming → Programming Request and Link Handlers	
Step #4 Making Layout Configurable	→ Delivering CSS and JavaScript from Content Repository	

Agenda

1. Framework Basics
2. Content Type Model and Content Beans
3. Template Development
4. Business Logic
5. Request and Link Handlers
6. Page layouts and CSS
7. Modularization
8. Advanced View Programming
9. Caching with Dataviews
10. Integration of External Content

COREMEDIA

You need some knowledge about standard and CoreMedia taglibs to write your templates.

CoreMedia Taglib: link

Example: creating a link to a blob property

```
<%@taglib prefix="cm"
   uri="http://www.coremedia.com/2004/objectserver-1.0-2.0" %>
```

→ the CoreMedia tag library
→ **cm** is the recommended prefix

```
...

<cm:link target="${self.data}" var="imgLink" />
<img src="${imgLink}" />
```

generates a textual link to a content bean
→ **target**: the bean to link to.
→ **view** (optional): view to render the bean with
→ **var** (optional): the URL is stored as a page scope attribute of that name

COREMEDIA

JSTL Basics: EL Arrays and Lists

→ you can access the elements of an Array or a List with the well known **[*n*]** syntax

→ results keep their dynamic types as well

```
<img src="<cm:link target="${self.image[0].data}"/>" />
```

→ content beans have no cardinality in lists. The content bean method signature for this link list property is
`List<? extends Image> getImage()`

→ This expression maps to the "data" property of the first entry in the list.

CoreMedia Taglib: include

```
<cm:include self="${self.image[0]}" view="thumbnail" />
```

Renders the first image as thumbnail.

Parameters:

→ **self**: the object to be rendered

→ **view** (optional): how the object 'self' should be rendered.

→ **ignoreNull** (optional): if set to 'true' includes are ignored if 'self' is null.

result from the view dispatcher is included in the JSPs output at this specific position

CoreMedia Taglib: param

the "cm:param" tag can only be used inside the "cm:include" or the "cm:link" tag

→ it sets a request attribute for the name given in "name"

→ the value is given in "value"

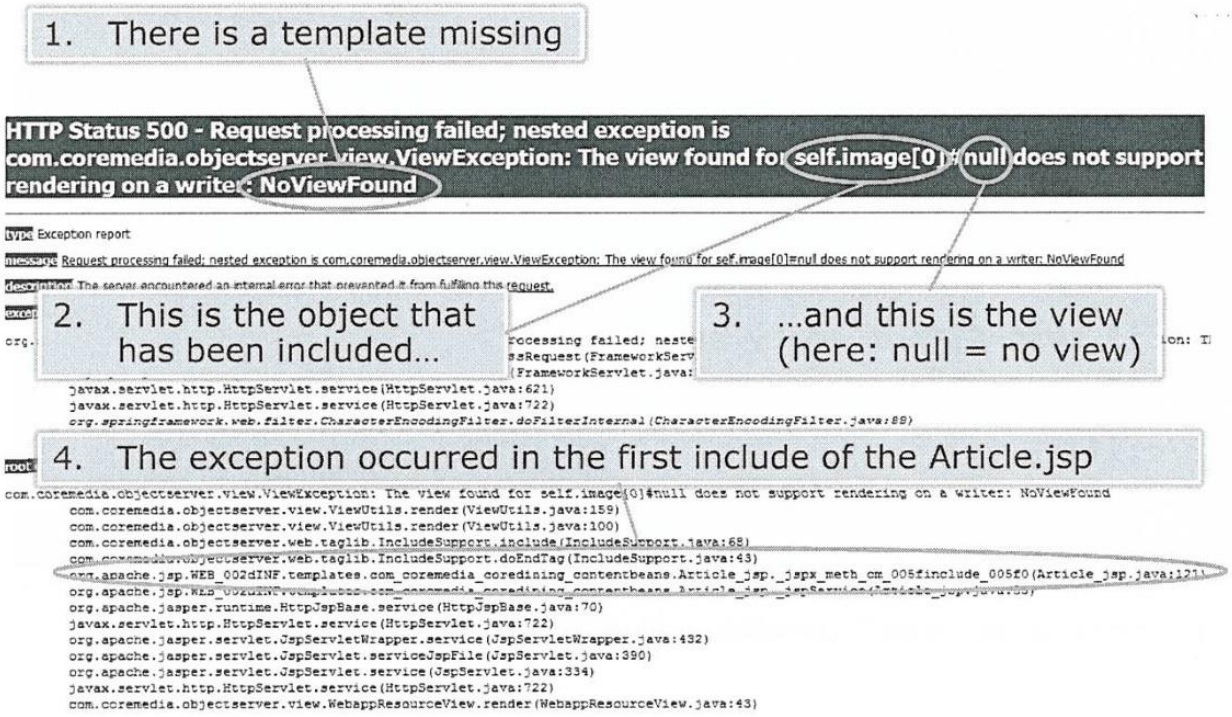
→ the view implementation (JSP) or link scheme can access these parameters

```
<cm:include self="${self.rootTopic}">
  <cm:param name="parentTopic" value="${self.rootTopic}"/>
  <cm:param name="index" value="1"/>
</cm:include>
```


Exercises 6 and 7

Develop the first version of your application

Reading Exceptions



The JSTL core tag library

- *"JavaServer Pages Standard Tag Library (JSTL) encapsulates as simple tags the core functionality common to many Web applications."* (source: Oracle)
- The "core" tag library contains structural tags for conditional rendering, iterations and HTML encoding. It is the most commonly used tag library in the CAE.
- Declaration:

```
<%@ taglib prefix="c"
           uri="http://java.sun.com/jsp/jstl/core" %>
```

"c" is the recommended prefix for the JSTL core library

JSTL Basics: Conditionals

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core" %>

<c:if test="${not empty self.title}">
  <h1>${self.title}</h1>
</c:if>
```

The operator "**empty**" checks for *null*, empty string, empty list or empty array

everything inside the <c:if> element is only evaluated in case the test succeeds

→ There is no "else" condition in the c:if tag.

JSTL Basics: Conditionals

```
<c:choose>
  <c:when test="${not empty self.title}">
    <h1>${self.title}</h1>
  </c:when>
  <c:otherwise>
    <h1>No title</h1>
  </c:otherwise>
</c:choose>
```

One or more when clauses, containing test conditions.

An else-clause which is choosen, if none of the "when" clauses was valid.

→ The first valid <when> clause is chosen. If none of the when clauses succeeded, the <otherwise> clause is rendered.

JSTL Basics: Iterations

```
<c:forEach items="${self.related}" var="rel">
   <li>${rel.title}</li>
</c:forEach>
```

iterates over all values in the attribute "items"

"items" is supposed to hold a list or an array

in case it is empty (or null) nothing will be done

every value – one after the other – of the list specified in attribute "items" is stored in a variable named like this

you can access a variable by passing its name to an EL expression

this simply dumps the string representation

JSTL Basics: HTML and URL Encoding

```
<h1><c:out value="${self.title}" /></h1>
```

The tag <c:out> assures that all special characters (especially '<' and '>') are encoded using HTML Entities (e.g. '<', '>')
You should use this tag for all strings in order to prevent HTML code injection!

```
<link rel="stylesheet"
      href="<c:url value="/css/structure.css" />"
      type="text/css" media="all"/>
```

The tag <c:url> can be used in order to properly include static URLs. The base uri of the web application is added to the given value. URL encoding can be activated as well.

here: the output will be: /coredining/css/structure.css

COREMEDIA

Adding checks and loops...

COREMEDIA

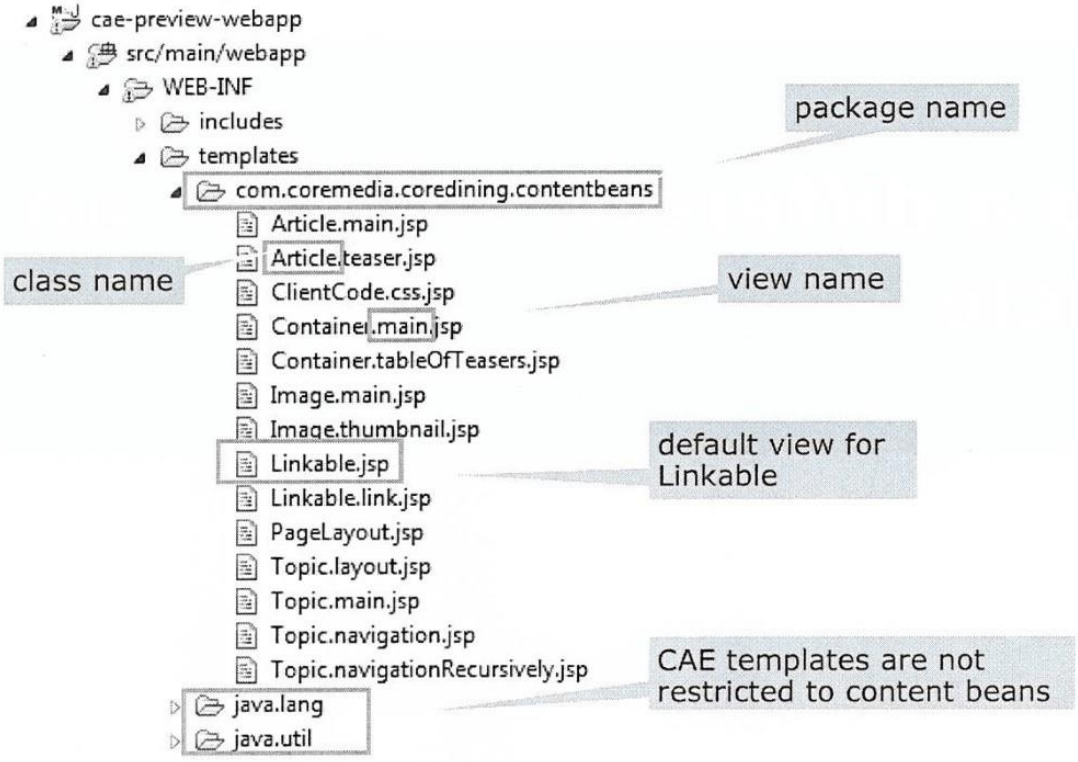
Stepping up one level: Understand how the View Dispatcher works

CoreMedia View Dispatcher

- JSP templates reside in a directory named after the content bean package
- template names are composed of
  - the content bean class name
  - optionally a view name separated by a "."
  - "JSP" as suffix
- in case there is no template named after the content bean the view dispatcher tries to find a template for
  1. one of the implemented interfaces
  2. one of content beans super classes
  3. recursively for all super classes (breadth-first)

object oriented dispatch

Template Repository



Example: Dispatching Article with default view

- Existing templates
  - Article.teaser.jsp
  - Linkable.jsp
- Steps:
  1. content type Article maps to ArticleImpl (contentbeans.xml)
  2. There is no *ArticleImpl.jsp* **->** check for all interfaces

     `ArticleImpl extends ArticleBase implements Article`

  3. There is no Article.jsp **->** check for all super classes

     `ArticleBase extends LinkableImpl`

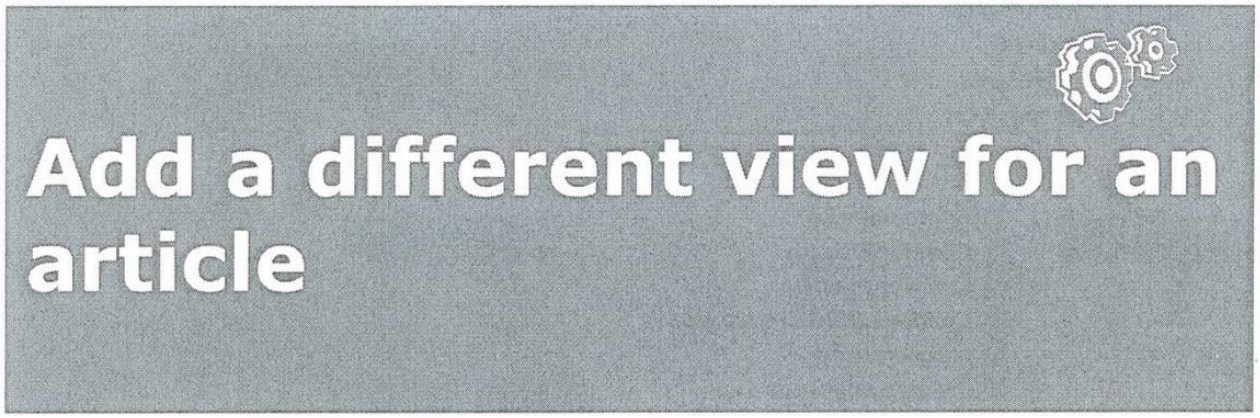
  4. There is no *ArticleBase.jsp* and no additional interfaces for that **->** check for all super classes
  5. There is no *LinkableImpl.jsp* **->** check for all interfaces

     `LinkableImpl extends LinkableBase implements Linkable`

  6. **There actually is *Linkable.jsp* thus the template is taken**

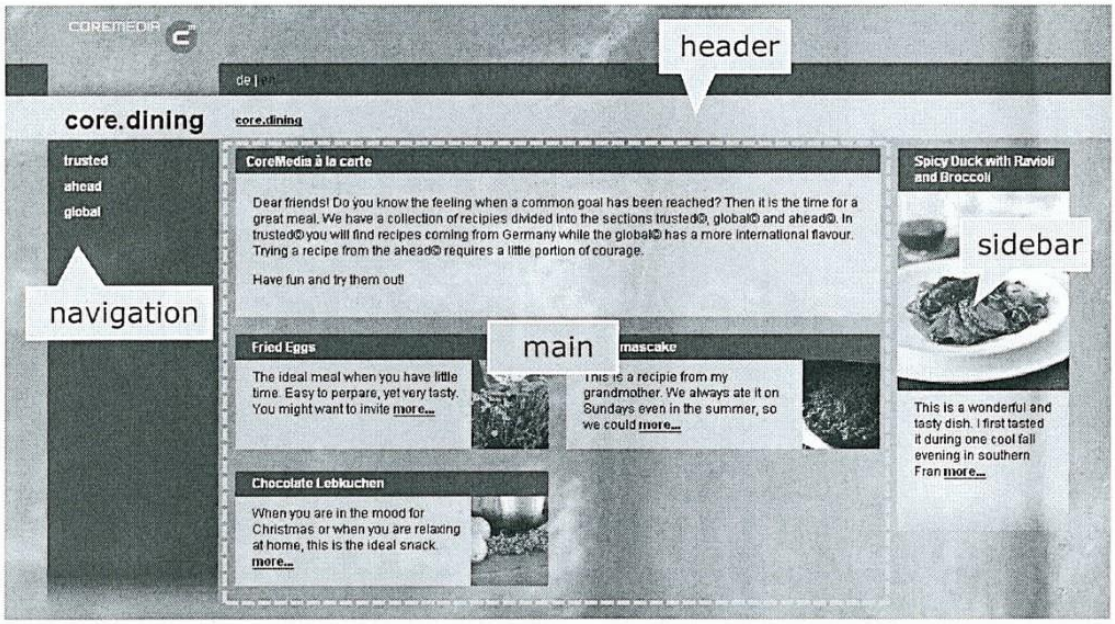
Exercises 10

COREMEDIA



COREMEDIA

Core.Dining Page Structure



→ The Core.Dining website has a common frame.

→ Only the main area is different for articles and topics

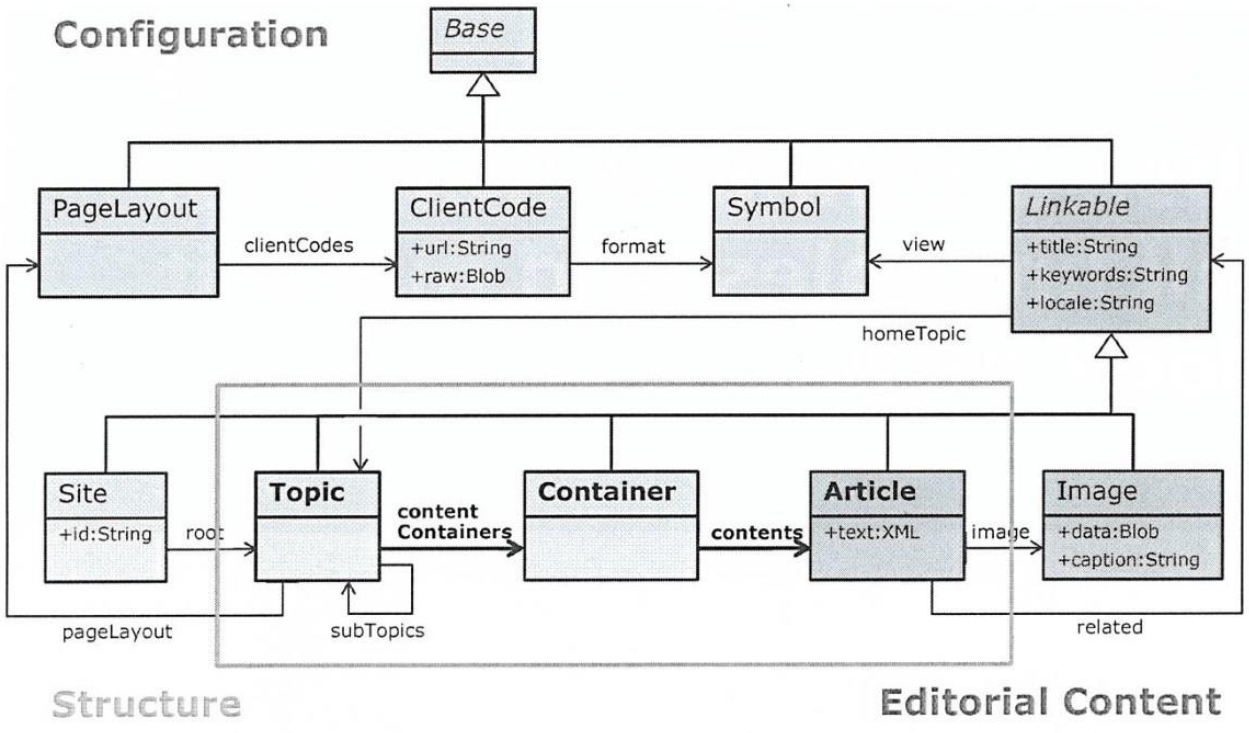
Exercises 11 + 12

Use a common frame for all linkables and make it look nice...

CORE MEDIA

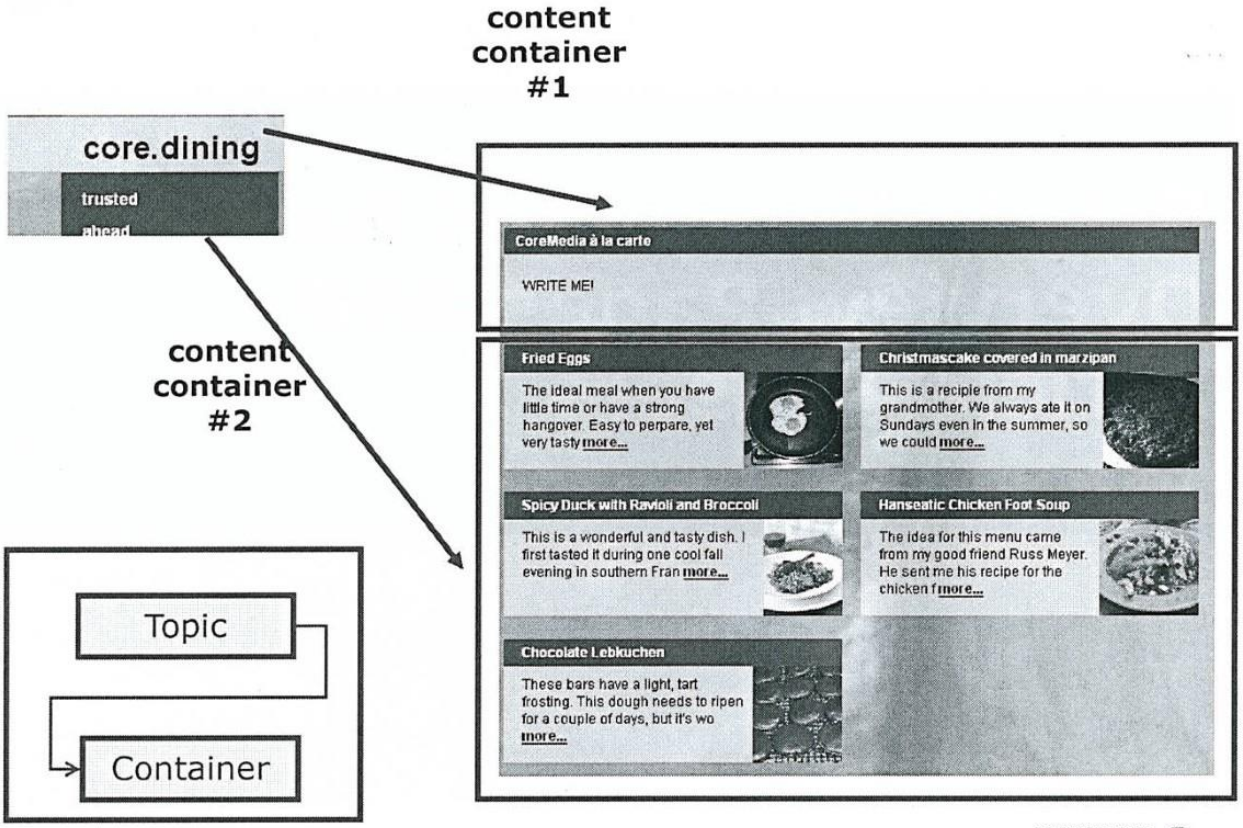
Overview Pages are rendered by **Topic.main.jsp**

Core.Dining Content Type Model



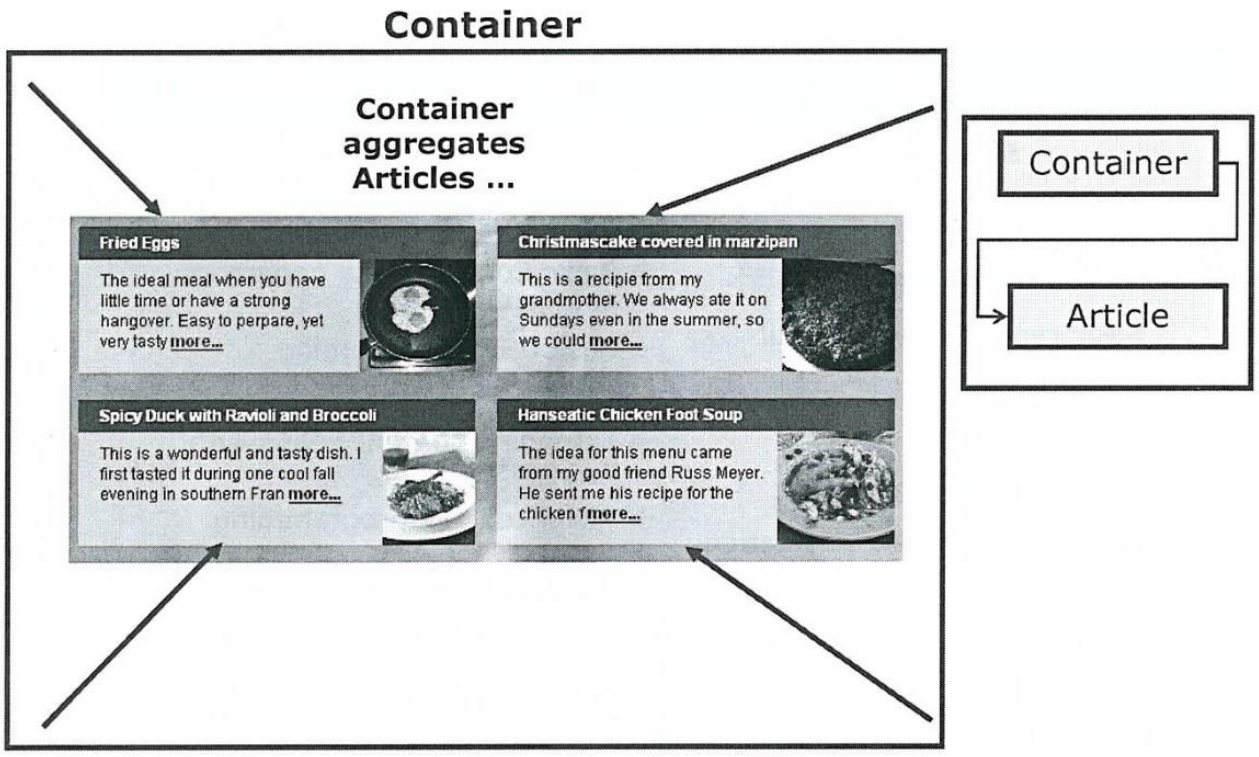
CORE MEDIA

Topic.main.jsp



COREMEDIA

Container.main.jsp



COREMEDIA

COREMEDIA

Create Templates for Overview Pages and Article Teasers

COREMEDIA

Proposed steps towards the solutions

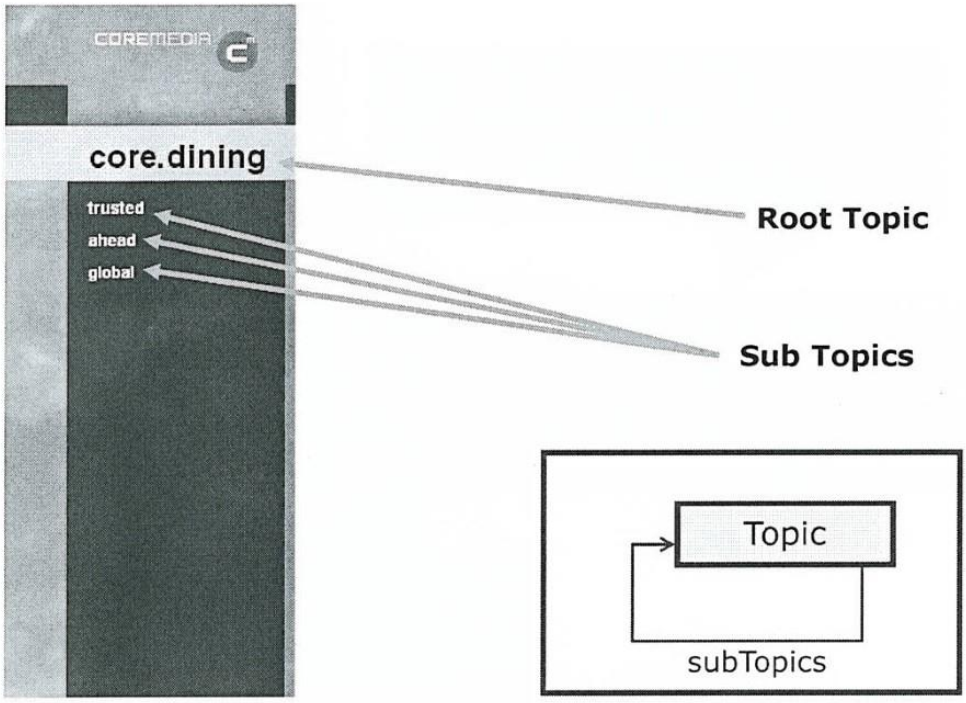
Task	Technical Solution	
Step #1 Content type model	→ Use the pre-built content type model already provided	☑
Step #2 Basic rendering and layout	→ Basic Content bean generation and configuration → Basic JSP programming	☑
Step #3 Rendering the complex navigation	→ Extending content beans with business logic → Advanced JSP programming → Programming Request and Link Handlers	☐
Step #4 Making Layout Configurable	→ Delivering CSS and JavaScript from Content Repository	

COREMEDIA

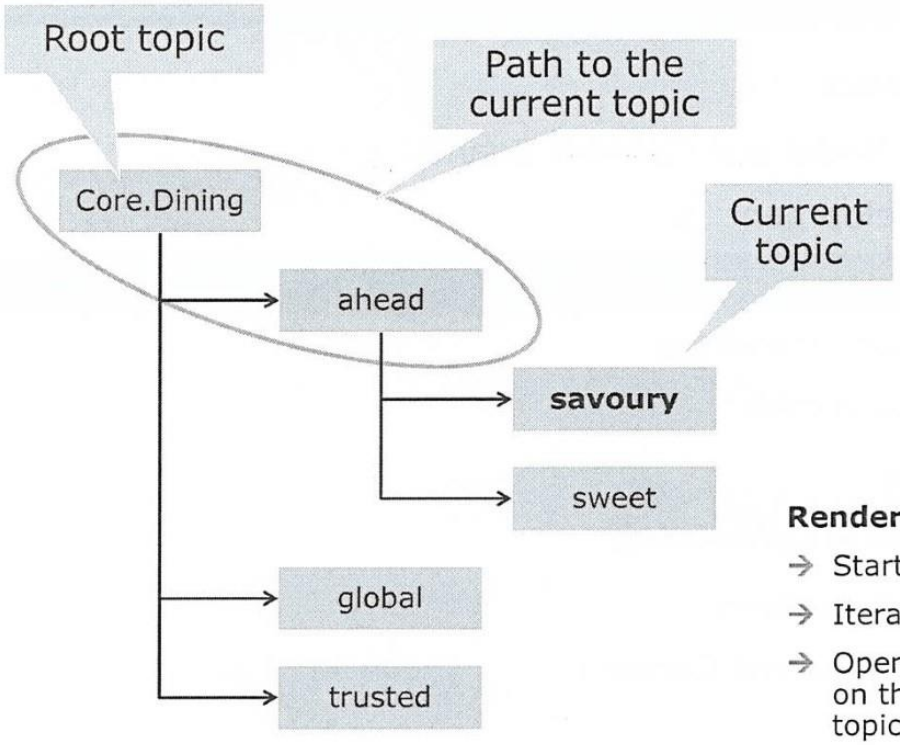
Agenda

1. Framework Basics
2. Content Type Model and Content Beans
3. Template Development
4. Business Logic
5. Request and Link Handlers
6. Page layouts and CSS
7. Modularization
8. Advanced View Programming
9. Caching with Dataviews
10. Integration of External Content

From style guide to implementation: Left Navigation



Understanding the navigation



Rendering the navigation

- Start with the root topic
- Iterate all subTopics
- Open all nodes which are on the path to the current topic (recursion)

COREMEDIA

Template Logic: Starting recursion to display navigation in Topic.navigation.jsp

- you start with the root Topic
- it will not be displayed as link
- but all its sub topics will...

```
<cm:include self="${self.rootTopic}" view="navigationRecursively">
  <cm:param name="index" value="1"/>
  <cm:param name="pathElements" value="${self.pathElements}"/>
</cm:include>
```

- you pass initial parameters for
- the index counter for the Topic path,
- the complete Topic path, and

COREMEDIA

Template Logic: Recursively displaying topics: Topic.navigationRecursively.jsp

iteratively displays all sub topics of the current topic

```
<c:forEach items="${self.subTopics}" var="topic">
  <a href="<cm:link target="${topic}"/>">
    <c:out value"${topic.title}" />
  </a>
  <c:if test="${topic eq pathElements[index]}">
    <cm:include self="${topic}" view="navigationRecursively">
      <cm:param name="index" value="${index + 1}"/>
      <cm:param name="pathElements" value="${pathElements}"/>
    </cm:include>
  </c:if>
</c:forEach>
```

adds a link for each topic

you need to display all sub topics of all elements in your path to the root:

recursive call to all sub topics that are to be displayed

Only rendering should be in the templates. Business and application logic goes into the beans.

CoreMedia

Topic needs some application logic to display the navigation...

Topic.java
Two methods required for navigation

```
/**
 * Builds a list of topics that reflect the path from this
 * topic to the root topic in reverse order. This topic
 * will not be included in the path.
 *
 * @return list of parents starting with the overall
 *         root topic at index 0.
 */
public List<? extends Topic> getPathElements();

/**
 * Traverses the topic hierarchy to find the root of
 * the topic tree.
 *
 * @return the root topic of this topic
 *         (might be this topic itself)
 */
public Topic getRootTopic();
```

CoreMedia

COREMEDIA

... we start with a basic method to determine the parent of a Topic ...

Business Logic: TopicImpl.getParent

```
public class TopicImpl … {
…
public Topic getParent() {
  Set<? extends Content> incoming = getContent()
        .getReferrersWithDescriptor("Topic","subTopics");
  if (incoming.isEmpty()) {
    return null;
  } else {
    Content content = incoming.iterator().next();
    return (Topic) createBeanFor(content);
  }
}
```

get all content objects that link to this topic via the "subTopics" property of type "Topic"

if there are no such links, this topic does not have a parent

if there actually are such links (parents), there must be exactly one, as the structure is a tree

create the content bean from the content object using the content bean factory inherited from *AbstractContentBean*

COREMEDIA

... next we use this method to get a path from a Topic to the root Topic ...

COREMEDIA

... which is important to project the complete navigation tree to the Topics to be displayed ...

Business Logic: TopicImpl.getPathElements

```
public List<? extends Topic> getPathElements() {

  // Proposed steps:
  // (1) create an ArrayList to contain the result
  // (2) start with the parent of this topic
  // (3) if there is no parent, you are done
  // (4) otherwise add the parent to the *front* of list
  //     to return
  // (5) repeat from (3) with parent of the current parent
  // (6) when you are done return the constructed list
}
```


... in the next step you implement a third method and use the existing ones.

COREMEDIA

Create templates and code for rendering the navigation

COREMEDIA

COREMEDIA

It is not possible to display a Topic within its home Topic, as it has none.

COREMEDIA

Information relevant to all or many templates or beans should be set by a **controller**...

COREMEDIA

... like your variable for the current Topic!

Agenda

1. Framework Basics
2. Content Type Model and Content Beans
3. Template Development
4. Business Logic
5. Request and Link Handlers
6. Page layouts and CSS
7. Modularization
8. Advanced View Programming
9. Caching with Dataviews
10. Integration of External Content

5. Request and Link Handlers

- → Request Handling
  - → Controller (classic)
  - → Handler (preferred)
- → Link Creation
  - → Link Schemes (classic)
  - → Link Handler (preferred)

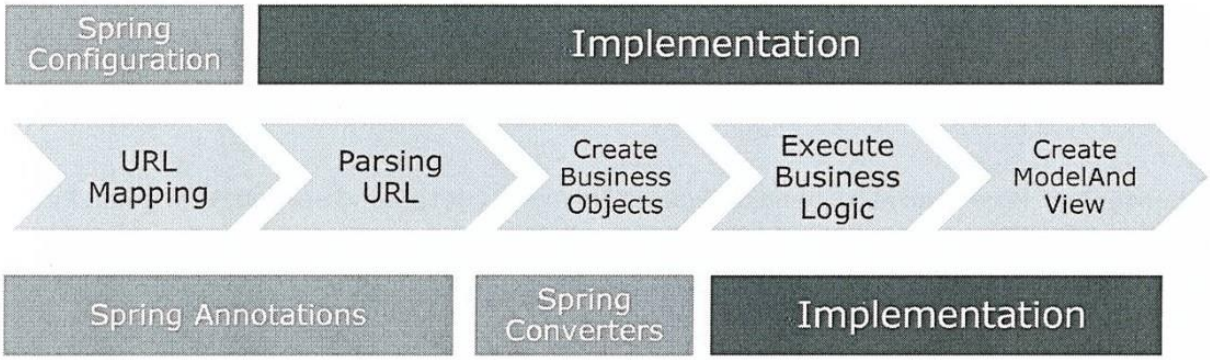
Controllers vs. Handlers

Since Spring MVC 3.0 there are two concepts for handling incoming requests:

- **Controllers**
  - Custom controllers need to extend AbstractController
  - Programmatic approach:
    Use Servlet API to access properties of the incoming Servlet Request
  - Configuration:
    Use Spring configuration to register controllers for URL patterns
- **Handlers**
  - Added in Spring 3.0
  - Uses Annotations to access request properties and to register controllers for URL patterns
  - Use Spring Converters for converting URL segments to business objects.

Controllers vs. Handlers

Controllers



Handlers

Comparison

- Handlers require less implementation
- Controllers offer more flexibility

5. Request and Link Handlers

- → Request Handling
  - → Controller (classic)
  - → Handler (preferred)
- → Link Creation
  - → Link Schemes (classic)
  - → Link Handler (prefered)

ContentViewController

default controller is based on logic inherited from Spring MVC

public abstract class **AbstractViewController extends** org.springframework.web.servlet.mvc. **AbstractController**

```
public class ContentViewController extends
   com.coremedia.objectserver.web.AbstractViewController {

  protected ModelAndView handleRequestInternal(HttpServletRequest request,
  HttpServletResponse response) throws Exception {
     …
  }

  protected Object resolveBean(String controllerPathInfo,
    Map parameters,
    HttpServletRequest request) {
     …
  }
}
```

- → central entry point for all requests
- → has the naked request and response as parameters only

- → figures out which bean actually is to be displayed
- → to do that, it has access to *path*, *parameters* and *full request* for special input

ContentViewController #handleRequestInternal()

```
String controllerPathInfo = controllerPathInfo(request);
Map parameters =
  ControllerUtils.parseParameters(request);

Object bean =
     resolveBean(controllerPathInfo, parameters, request);
String view =
     resolveView(controllerPathInfo, parameters, request);

bean = loadDataViewFor(bean, view);

ModelAndView result = ControllerUtils.createModelAndView(view);
ControllerUtils.setSelf(result, bean);

return result;
```

strips the configured prefix from our URL path. (evaluates to "**/22**")

uses the method you will see on the next slide

extracts the view parameter

loads the cached version if there is one

creates a ModelAndView object with:

- the bean as "self" attribute
- the view

COREMEDIA

ContentViewController #resolveBean()

```
protected Object resolveBean(String controllerPathInfo,
    Map parameters,
    HttpServletRequest request) {

    int contentId =
       Integer.parseInt(controllerPathInfo.substring(1));

    Content content =
      contentRepository.getContent(
         IdHelper.formatContentId(contentId));

    Object bean =
       contentBeanFactory.createBeanFor(content);

    return bean;
 }
```

default controller only uses information encoded in URL

gets you the id (**22** in our previous example) of the content object...

... which is used to retrieve the content from the repository configured with the controller

the content bean factory is used to create content beans from the generic content objects

COREMEDIA

Controller Registration

all Spring MVC resources are configured in XML configuration files

this is how you can define constant values

```
<beans>
  <bean id="contentViewController"
  class="com.coremedia.objectserver.web.ContentViewController">
    <property name="prefix" value="/content"/>
    <property name="contentRepository" ref="contentRepository"/>
    <property name="contentBeanFactory" ref="contentBeanFactory" />
    <property name="dataViewFactory" ref="dataViewFactory"/>
  </bean>
</beans>
```

this is a reference to bean defined somewhere else (dependency injection)

Controller URL Mapping

A customizer adds the given entries to the map 'controllerMappings' which is defined within the framework of the CAE

```
<customize:append id="exampleControllerMappingsCustomizer"
    bean="controllerMappings" order="100">
    <map>
       <entry key="/content/*" value-ref="contentViewController"/>
       <entry key="/contentblob/*/**"
              value-ref="contentBlobViewController"/>
       ...
    </map>
</customize:append>
```

default controller matches URLs like e.g. this:
http://localhost:40081/coredining/servlet/content/***22***

5. Request and Link Handlers

- Request Handling
  - Controller (classic)
  - Handler (preferred)
- Link Creation
  - Link Schemes (classic)
  - Link Handler (prefered)

ContentViewHandler

Plain Java object, no special inheritance

```
@RequestMapping
public class ContentViewHandler {

  @RequestMapping("/content/{id}")
  public ModelAndView handleContent(
       @PathVariable("id") ContentBean self,
       @RequestParam(value="view", required=false) String view) {

    if (self==null) {
      return HandlerHelper.notFound();
    }

    ModelAndView modelAndView =
              HandlerHelper.createModelWithView(self, view);

    return modelAndView;
  }
}
```

An easy way to return a 404 response

Create a ModelAndView

Handler Annotations

- Handler class need to be annotated with `@RequestMapping`
- Handler method must be annotated with `@RequestMapping`, specifying a URL Pattern. URL Patterns also support regular expressions in variables.

```
@RequestMapping("/content/{id}")
@RequestMapping("/content/{id:\\d+}")
```

- Handler method parameters can be bound to path variables:

```
@PathVariable("id") ContentBean
```

- Type conversion and dataview loading is done automatically!
- Handler method parameters can be bound to request parameters:

```
@RequestParam("view") String view
@RequestParam(value="view", required=false) String view
```

- In the first example, the method will only map to request which have a request parameter "view".

Content Handler Registration in Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context">

  <context:annotation-config />

  <import resource="classpath:/com/coremedia/cae/handler-services.xml"/>

  <bean id="contentViewHandler"
        class="com.coremedia.coredining.handlers.ContentViewHandler" />

</beans>
```

Activates parsing for spring annotations.

Adding the handler to the application context is sufficient. There is no special mapping required.

Content Handler
Registration of Spring Converters

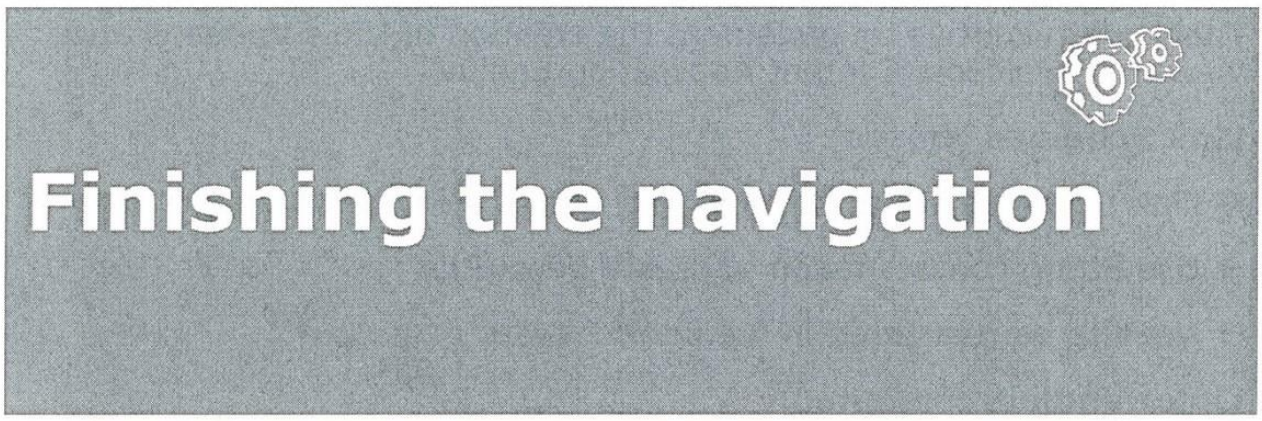
```
<import resource="classpath:/com/coremedia/cae/handler-services.xml"/>

...

<customize:append id="corediningBindingConverterCustomizer"
                  bean="bindingConverters">
  <set>
    <ref bean="idGenericContentBeanConverter" />
  </set>
</customize:append>
```

Standard converters are defined in handler-services.xml

Use a customizer to register converters

"idGenericContentBeanConverter" maps id segments to ContentBeans

Exercises 16



5. Request and Link Handlers

- → Request Handling
  - → Controller (classic)
  - → Handler (preferred)
- → Link Creation
  - → Link Schemes (classic)
  - → Link Handler (prefered)

We want Search engin optimized link

Link Schemes vs. Link Handlers

There are two ways to implement the creation of URLs to beans and views in CoreMedia Content Application Engine

Link Schemes (classic)

- → Link Schemes are classes derived from the interface LinkScheme
- → Use String concatenation to create new URLs
- → Need to be registered in the correct order

Link Handlers (preferred)

- → Link Handlers are methods with @Link annotation.
- → Link Handlers and Request Handlers can be implemented in the same class
- → Automatically registered via Spring annotation-config
- → Can use the same URL pattern, used by @RequestMapping

COREMEDIA

Link creation and request handling always go

hand in hand

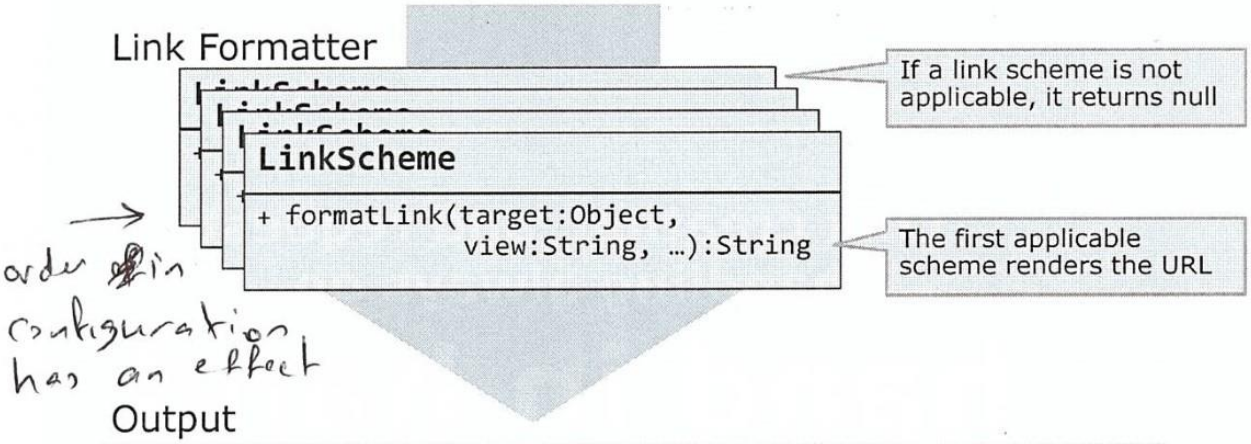
5. Request and Link Handlers

- Request Handling
  - Controller (classic)
  - Handler (preferred)
- Link Creation
  - Link Schemes (classic)
  - Link Handler (prefered)

What does a LinkScheme do?

Template

```
<a href="<cm:link target="${self}" view="pdf" />">PDF</a>
```



Output

```
<a href="/content/220?view=pdf">PDF</a>
```


COREMEDIA

What does the default link scheme do?

```
public String formatLink(Object o, String view, …) {
    if(!(o instanceof ContentBean)) return null;

    ContentBean bean = (ContentBean) o;

    int id = IdHelper.parseContentId(bean.getContent().getId());
    String query = "view="+view;

    String path = getPrefix() + "/" + id;

    String result = new URI(null, null, path, query, null).toString();
    return postProcess(result, request, response, forRedirect);
  }
```

this link scheme only matches content beans

this way we get the internal content id

view gets encoded as an ordinary query parameter

For every link scheme there must be a matching controller!

the path simply is the prefix plus the content id, exactly what the default controller expects!

don't forget URL encoding!

adding session-ID and base URL!

COREMEDIA

Configuration of a link scheme

link scheme bean definition

```
<bean id="optimizedLinkScheme"
      class="com.coremedia.corebase.linkschemes.OptimizedLinkScheme">
   <property name="prefix" value="/cms" />
</bean>

<customize:append id="linkSchemesCustomizer" bean="linkSchemes"
                                  order="100">
    <list>
      <ref bean="optimizedLinkScheme" />
      <!-- DEFAULT LINKSCHEMES -->
      <ref bean="contentLinkScheme"/>
      <ref bean="contentBlobLinkScheme"/>
      <ref bean="beanPropertyBlobLinkScheme"/>
    </list>
</customize:append>
```

Remember!!!
You also need a matching controller...

Order is important!

5. Request and Link Handlers

- Request Handling
  - Controller (classic)
  - Handler (preferred)
- Link Creation
  - Link Schemes (classic)
  - Link Handler (prefered)

Link Handler Annotation

- Classes containing link handler methods must be annotated with **@Link**
- Link handler methods need to be annotated with **@Link** which may have the following attributes:
  - **type** (Class) :
    restricts this link handler to a specific target bean type
  - **uri** (String) :
    defines a URI pattern with place holders.
  - **view** (String) :
    restricts this link handler to a specific view name
  - **parameter** (String) :
    restricts this handler to the existence of a <cm:param> parameter with the given name.
- Like with **@RequestMapping**, the Link handlers will be automatically registered, when adding them to the spring application context.

Link Handler Return Value

Link handler methods may return one of the following types

- `java.lang.String`
- `java.net.URI`
- `org.springframework.web.util.UriComponents`
- `org.springframework.web.util.UriComponentsBuilder` (recommended)
- `java.util.Map<String, Object>` - a map of URI variable placeholders (recommended)

UriComponents and UriComponentsBuilder are contain convenient methods for building URIs (append/prepend segments, request parameters, encode, ...)

UriComponentsBuilders works together with URI patterns containing variable placeholders...

```
UriComponentsBuilder builder = UriComponentsBuilder.newInstance();

builder.path("/content/{id}/{name}.{view}");

int id = IdHelper.parseContentId(article.getContent().getId());

return builder.buildAndExpand(id, article.getTitle(), view);
```

Configuration of a link scheme

link scheme bean definition

```
<bean id="optimizedLinkScheme"
      class="com.coremedia.corebase.linkschemes.OptimizedLinkScheme">
   <property name="prefix" value="/cms" />
</bean>

<customize:append id="linkSchemesCustomizer" bean="linkSchemes"
                                  order="100">
    <list>
      <ref bean="optimizedLinkScheme" />
      <!-- DEFAULT LINKSCHEMES -->
      <ref bean="contentLinkScheme"/>
      <ref bean="contentBlobLinkScheme"/>
      <ref bean="beanPropertyBlobLinkScheme"/>
    </list>
</customize:append>
```

Remember!!!
You also need a matching controller...

Order is important!

5. Request and Link Handlers

- → Request Handling
  - → Controller (classic)
  - → Handler (preferred)
- → Link Creation
  - → Link Schemes (classic)
  - → Link Handler (prefered)

Link Handler Method Parameters

Link handler methods uses the following parameter type convention to map method parameters:

- `Object, ContentBean, ArticleImpl, ...`: the target bean
- `String`: the view name
- `Map<String, Object>`: maps to <cm:param> parameters
- `UriComponentBuilder`: can be used together with the `@Link(uri="...")` annotation.

The order of the parameters and the name of the method is arbitrary.

```
@Link(type=ContentBean.class, uri="/content/{id}")
public Map<String, Object> buildLink(ContentBean target, String view) {
  Map variables= new HashMap();
  variables.put("id", IdHelper.parseContentId(target.getContent().getId());
  return variables;
}
```


@LinkPostProcessor

- Typically, a link scheme should build links relative to the servlet context only (think of HTTPD URL Rewrite), e.g. /content/220 rather than /coredining/servlet/content/220
- The prefix (base URI) can be added by a link post processor like

```
@LinkPostProcessor
public Object prependBaseUri(UriComponents originalUri, HttpServletRequest request) {

  String baseUri = ViewUtils.getBaseUri(request);
  return UriComponentsHelper.prependPath(baseUri, originalUri);
}
```

- The annotation @LinkPostProcessor must be added to the containing class as well.
- See API documentation and the manual for details on
  - com.coremedia.objectserver.web.links.Link
  - com.coremedia.objectserver.web.links.LinkPostProcessor

Benefit of Handler vs. Link Scheme and Controllers

→ The annotation based approach allows implementing link creation and request handling in a single class.

```
@RequestMapping @Link @LinkPostProcessor
public class OptimizedContentHandler {
  public static final String PATTERN="/content/{id}";

  @RequestMapping(PATTERN)
  public ModelAndView handleContent(@PathVariable("id") ContentBean)
  {…}

  @Link(uri=PATTERN, type=ContentBean.class)
  public UriComponent buildLink(ContentBean target, UriComponentBuilder b)
  {…}

  ....

}
```

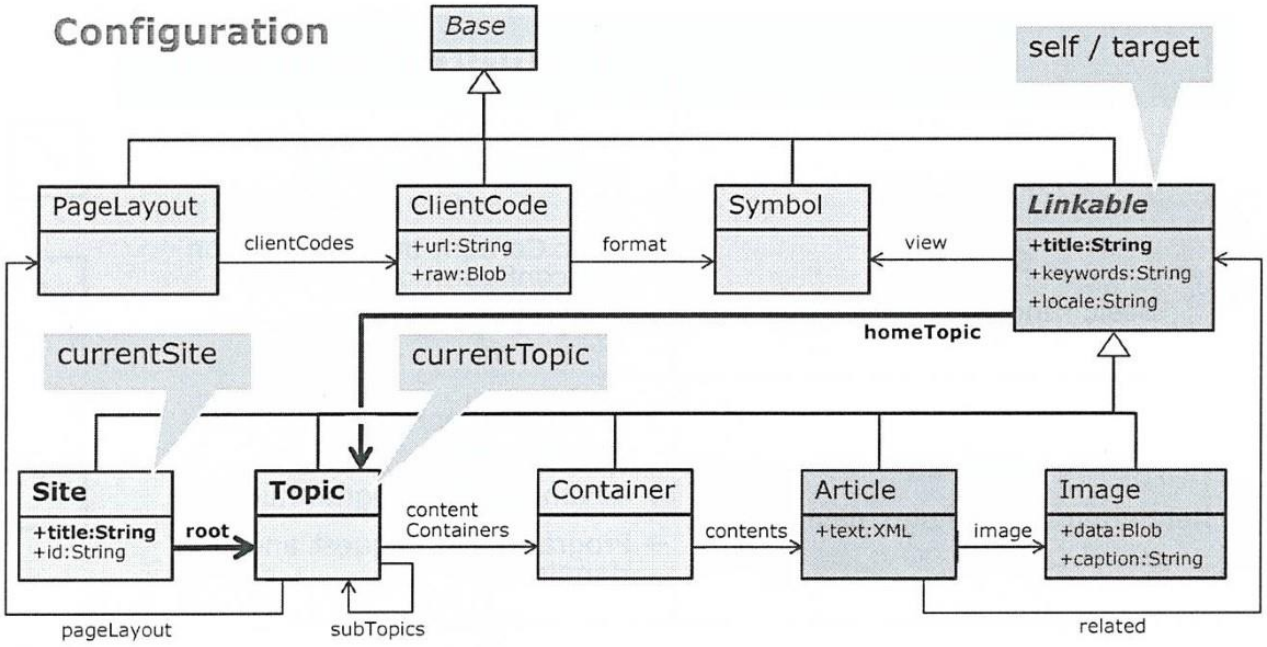

CONTENTMEDIA

Aim: Have a handler and a link scheme that reveals the full navigation hierarchy

An Optimized URL Format for Linkable

- **currentSite**: the Site document referring to the root topic. Use the „title" property in the URL.
- **currentTopic**: use getPathElements() to format the URL segments for the navigation path.
- **self**: use the numerical contentId and the title of the linkable. If the title is null, use "index" instead.
- **view**: use the view name as extension. If no view is defined (default), use "html".

Core.Dining Content Type Model



CoreMedia

Create an Optimized Link Handler

CoreMedia

Proposed steps towards the solutions

Task	Technical Solution	
Step #1 Document model	→ Use the pre-built document model already provided	☑
Step #2 Basic rendering and layout	→ Basic Content bean generation and configuration → Basic JSP programming	☑
Step #3 Rendering the complex navigation	→ Extending content beans with business logic → Advanced JSP programming → Programming Request and Link Handlers	☑
Step #4 Making Layout Configurable	→ Delivering CSS and JavaScript from Content Repository	☐

CoreMedia

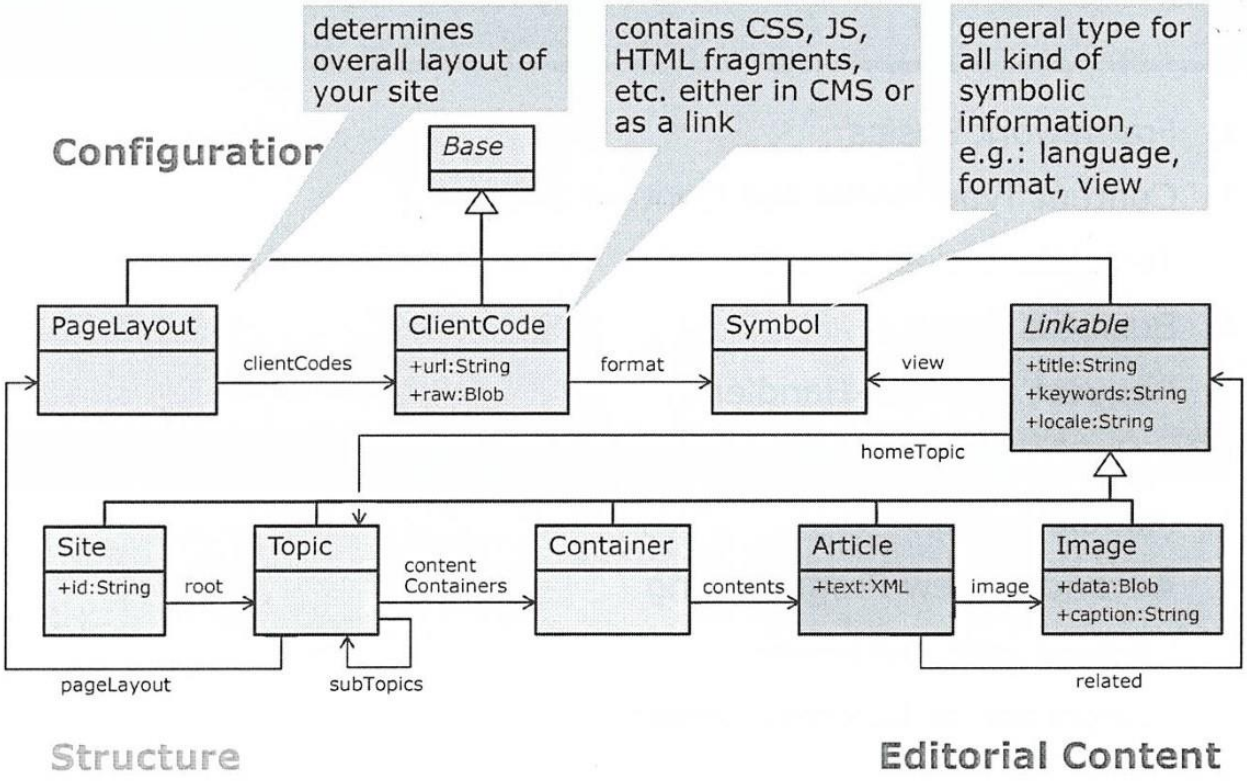
Agenda

1. Framework Basics
2. Content Type Model and Content Beans
3. Template Development
4. Business Logic
5. Request and Link Handlers
6. Page layouts and CSS
7. Modularization
8. Advanced View Programming
9. Caching with Dataviews
10. Integration of External Content

Configurable page layouts

→ A web site editor should be able to choose the layout without writing CSS or changing templates

Core.Dining Content Type Model



CoreMedia

Exercise 18

CoreMedia

Add dynamic page formatting based on CMS content

CoreMedia

Proposed steps towards the solutions

Task	Technical Solution	
Step #1 Document model	→ Use the pre-built document model already provided	✓
Step #2 Basic rendering and layout	→ Basic Content bean generation and configuration → Basic JSP programming	✓
Step #3 Rendering the complex navigation	→ Extending content beans with business logic → Advanced JSP programming → Programming Request and Link Handlers	✓
Step #4 Making Layout Configurable	→ Delivering CSS and JavaScript from Content Repository	✓

Agenda

1. Framework Basics
2. Content Type Model and Content Beans
3. Template Development
4. Business Logic
5. Request and Link Handlers
6. Page layouts and CSS
7. Modularization
8. Advanced View Programming
9. Caching with Dataviews
10. Integration of External Content

Why Modularization?

Current Situation

→ We are focussing on cae-preview-webapp

→ All implementations, spring configuration, templates and web-resources have been added to the cae-preview-webapp module.

But:

→ cae-preview-webapp is not the only web application!

→ Templates, spring-configuration and java resources are reused in cae-live-webapp.

→ ContentBean classes and ContentBean mappings are also used by the CAE Feeders.

Therefore we have to care about modularization!

Modularization in CoreMedia

Main approach:
Use Maven dependency mechanism to create re-usable modules. Every re-usable component should be a JAR Artifact!

Java classes

→ Classpath is defined by Maven dependencies
(This is Maven standard)

Web Resources

→ Servlet 3.0 allows placing of web resources in classpath

→ Web-resources need to be placed in classpath
below "META-INF/resources/"

Spring Configuration

→ CoreMedia Extension Mechanism

→ Load all spring files from the classpath which comply the following naming convention:
  → META-INF/coremedia/component-*.xml
  → META-INF/coremedia/component-*.properties

Three Types of Artifacts
Web-Apps, Components and Libraries

Web-Apps: e.g. cae-preview-webapp, cae-live-webapp

→ Contain no templates and configuration, but preview/live specific ones

→ Have maven dependencies to components

Components: e.g. cae-base-component

→ Encapsulate certain functionality

→ Have Maven dependencies to libraries

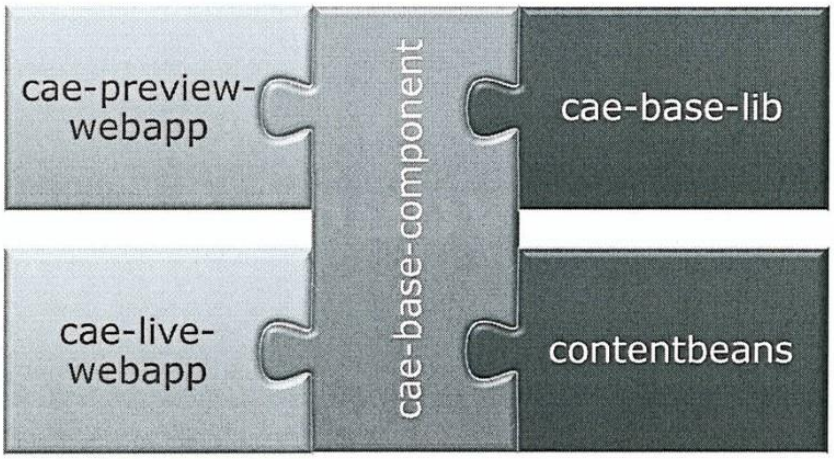
→ Have Spring configuration files in META-INF/coremedia very often these files import spring configuration from the library

Libraries: e.g. cae-base-lib, contentbeans

→ Contain implementation and spring configuration (in /framework/spring)

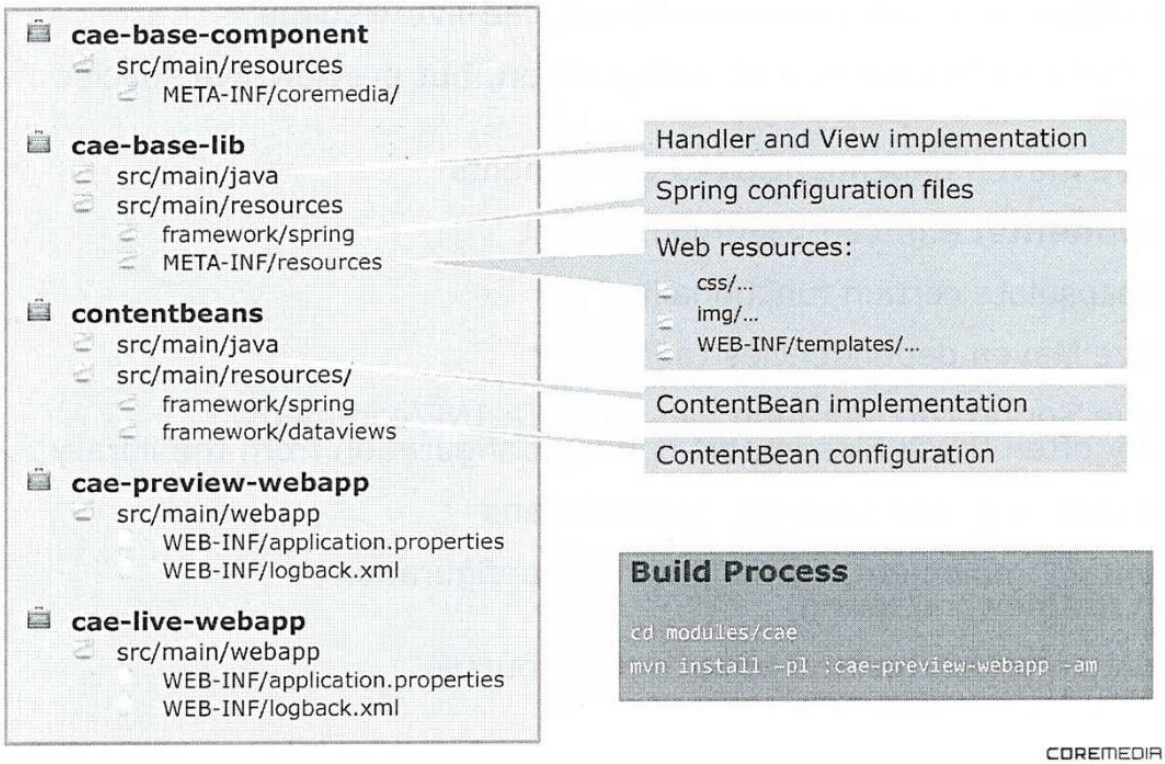
→ Contain templates and other web resources. (/META-INF/resources/)

The Role of the cae-base-component



→ The **cae-base-component** module is the **glue** between the web applications and the libraries

Folder Structure in CAE module



Build Process

```
cd modules/cae
mvn install -pl :cae-preview-webapp -am
```


Exercise 19

Build process

Because of the modular structure of the Maven workspace we will have to compile four different modules in order to start the cae-preview-webapp:

→ modules/cae/contentbeans

→ modules/cae/cae-base-lib

→ modules/cae/cae-base-component

→ modules/cae/cae-preview-webapp

Hint: Use maven to build the webapp with all dependencies:

```
cd C:\training\workspace\modules\cae
mvn clean install -pl :cae-preview-webapp -am
C:\training\workspace\modules\cae\cae-preview-webapp
mvn tomcat7:run -DskipTests
```


Agenda

COREMEDIA

JSPs are not the only way to render a content bean ...

COREMEDIA

Your teasers for Articles are not finished, yet. Use Java view programming to display their first words only...

Programmed view configuration in Spring

non-JSP (=Java) views are defined in cae-views.xml

```
<bean id="markupTeaserView"
  class="com.coremedia.corebase.view.MarkupTeaserView"/>

<customize:append id="programmedViewsCustomizer"
                  bean="programmedViews" order="100">
  <map>
    <entry key="com.coremedia.xml.Markup"
           value-ref="richtextMarkupView"/>
    <entry key="com.coremedia.cap.common.Blob"
                       value-ref="blobView"/>
    <entry key="com.coremedia.xml.Markup#teaserText"
           value-ref="markupTeaserView"/>
  </map>
</customize:append>
```

add your custom view to the map of the programmedViews bean

view class implementing a special interface

optional view name

each entry contains the class to specify a view for

→ The view, configured in the example above is equivalent to the following template:

templates/com.coremedia.xml/Markup.teaserText.jsp

The general TextView Interface

for your non-JSP (=Java) view you can either implement TextView or XmlView or both

```
public class PlainXmlView implements TextView {

  public void render(
      Object bean,
      String view,
      Writer out,
      HttpServletRequest request,
      HttpServletResponse response)
  {
    Markup markup = (Markup) bean;
    markup.writeOn(out);
  }
}
```

when you use TextView you need to implement this render method

parameters always include the bean to render and the chosen view

The writer of the current response. XmlView has ContentHandler as out

you can usually safely ignore these two last parameters as they are for advanced purposes only

writes plain XML with declaration

COREMEDIA

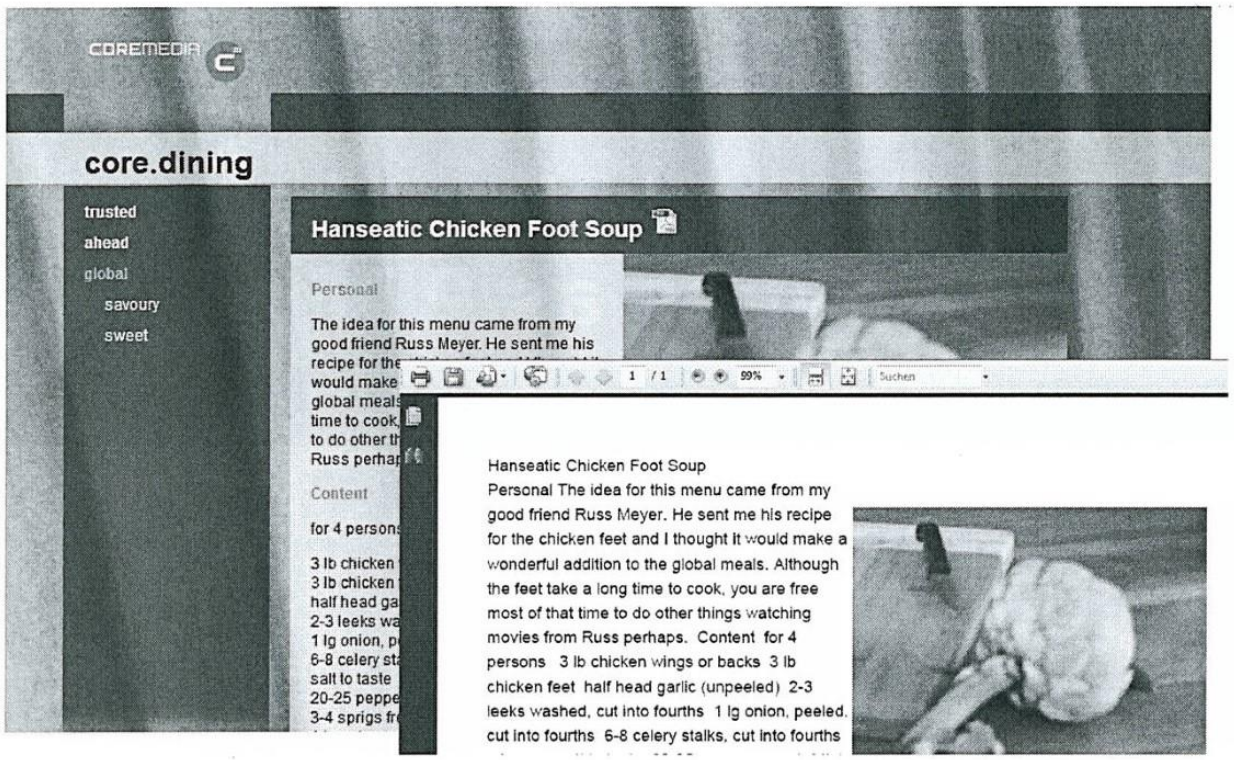
Display Articles as Teasers with a non JSP view

COREMEDIA

COREMEDIA

Aim: Display your Articles as PDF

Article download as PDF



The ServletView Interface

```
public class MyView implements ServletView {

  public void render(
        Object bean,
        String view,
        HttpServletRequest request,
        HttpServletResponse response) {
    byte[] bytes = …;
    response.setContentType(“text/text");
    …
    response.getOutputStream().write(bytes);
  }
}
```

ServletViews do not have a response writer 'out'.

Therefore you cannot use ServletViews for <cm:includes />

ServletViews are designed to generate complete responses.

COREMEDIA

Add a Servlet View to render pages as PDF

COREMEDIA

COREMEDIA

Aim: Decide which view to use guided by the "view" property

View Variants

Remember: In exercise 18 we already used the name of a symbol to determine the view for a ClientCode include:

```
<c:forEach items="${self.clientCodes}" var="clientCode">
   <cm:include self="${clientCode}" view="${clientCode.format[0].content.name}" />
</c:forEach>
```

In this module we are using a more generic approach called "View Types" or "Layout Variants"

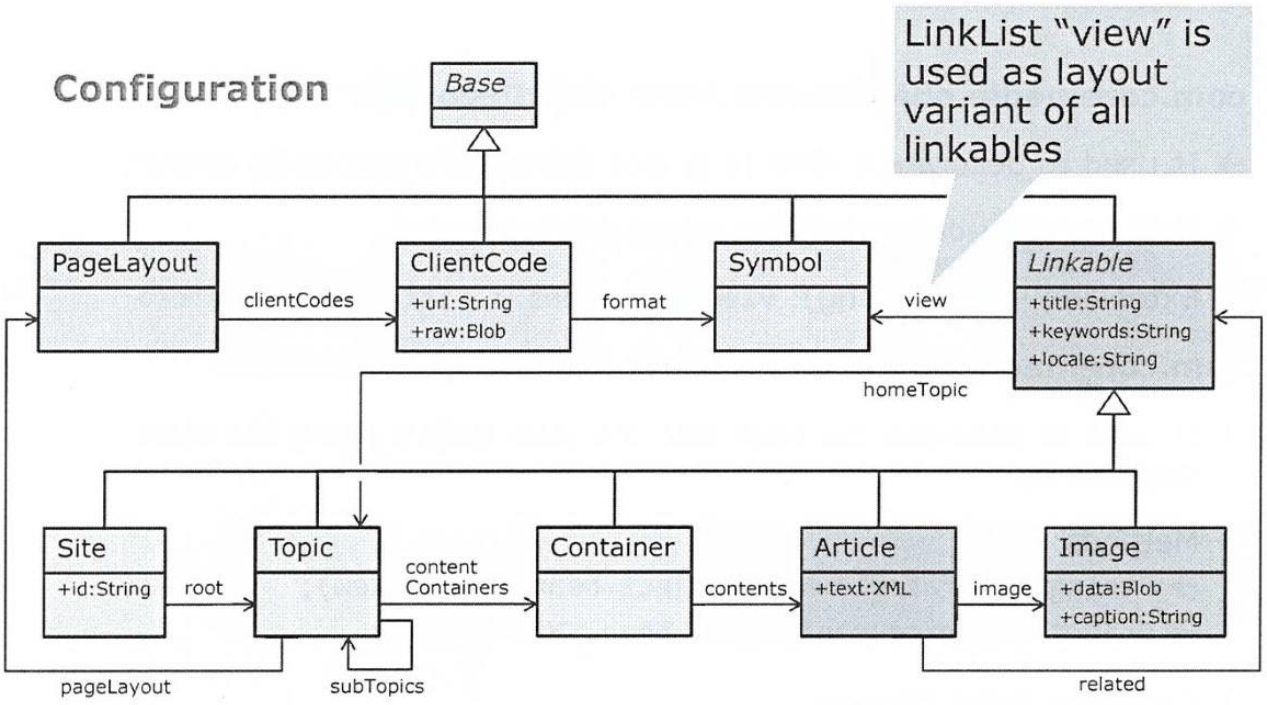
In Blueprint:

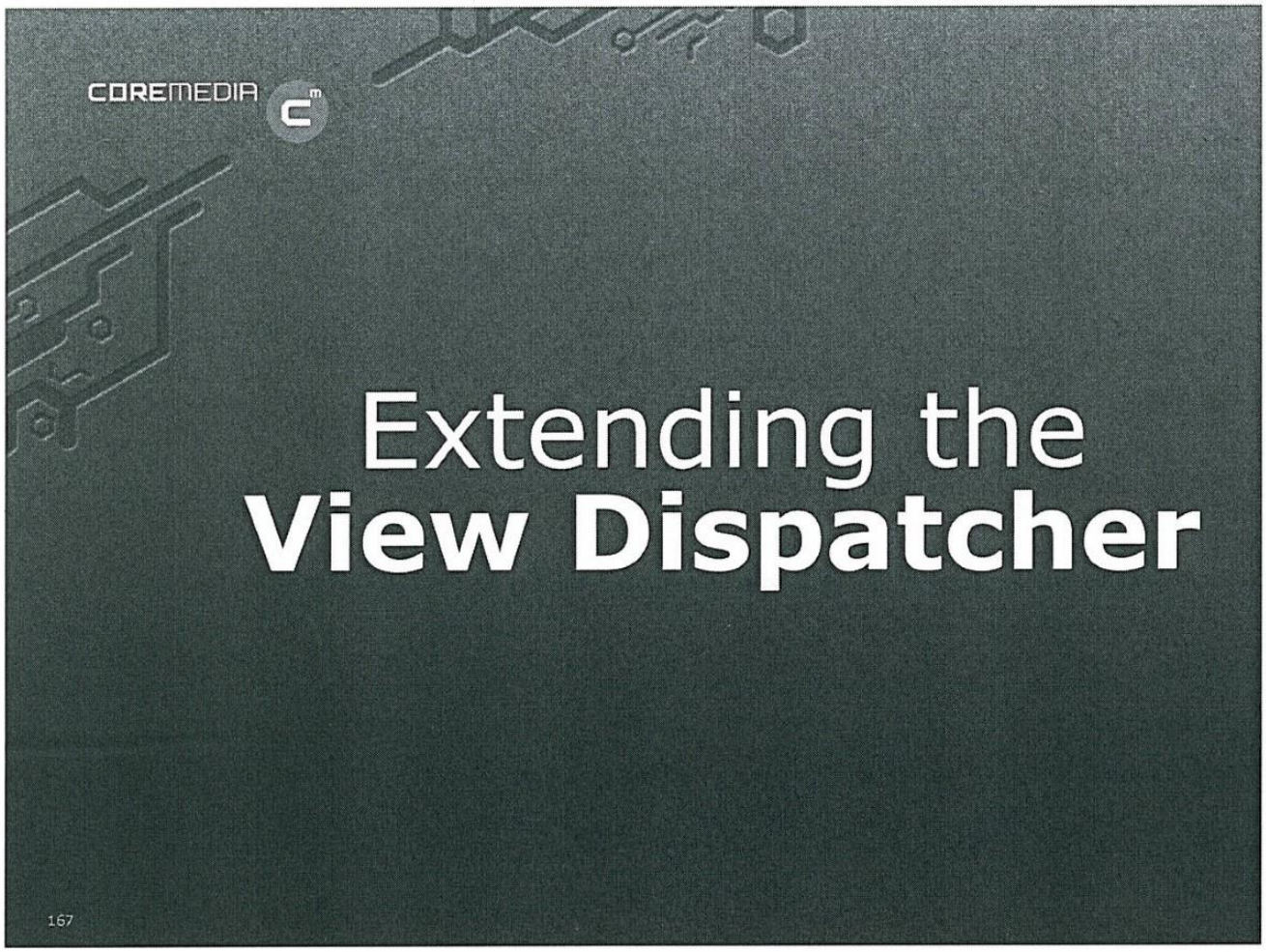
→ Every Linkable can have an associated Layout Variant (modeled as link list)

→ The Blueprint **View Dispatcher** has been improved to look-up templates based on the following naming convention:
TypeOfBean.viewName**[layoutVariant]**.jsp

We will now implement the same behaviour in Core.Dining.

Core.Dining Content Type Model





Extension Points of the ViewResolver

com.coremedia.objectserver.view.ViewDecorator

- Is used to decorate a view (e.g. add debug information to output)
- Method: `View decorate(View originalView);`
- Extension Point (Spring): `viewDecorators : List<ViewDecorator>`

com.coremedia.objectserver.view.RenderNodeDecorator

- Is used to decorate the bean and the view before doing the view dispatching
- Methods:
  - `String decorateViewName(Object bean, View view);`
  - `Object decorateBean(Object bean, View view);`
- Extension Point (Spring): `renderNodeDecoratorProviders:List<RenderNodeDecoratorProvider>`

More Extension Points of the ViewResolver

com.coremedia.objectserver.view.resolver.ViewLookupTraversal

- implements the view lookup strategy
- Default: `RepositoriesFirstViewLookupTraversal`
- Extension Point (in Spring)
  `viewResolver#viewLookUpTraversal : ViewLookupTraversal`

More extension points can be found in com/coremedia/cae/view-services.xml located in cap-objectserver-<version>.jar

Implementing a RenderNodeDecorator

For our task, an implementation of the method RenderNodeDecorator#decorateViewName() is sufficient:

```
public class ViewVariantRenderNodeDecorator implements RenderNodeDecorator {
  public String decorateViewName(Object bean, String view) {
    String viewVariant = … use the bean to evaluate the view variant
    if (viewVariant!=null) {
      if (view==null) {
        return "[" + viewVariant + "]";
      } else {
        return view + "[" + viewVariant + "]";
    }
    return view;
  }
  public Object decorateBean(Object bean, String view) {
    return bean; // no decorating required…
  }
}
```

How would you implement the evaluation of the viewVariant?

Implementing a RenderNodeDecorator

Keep in mind:
Your View Dispatcher should always work for all kind of beans!

→ The View Variant feature should also be available for Non-ContentBeans!

→ Therefore it is not a good idea to access the method Linkable#getView() : List<Symbol>.

→ Better way: Have an interface HasViewVariant and let LinkableImpl implement it:

```
public interface HasViewVariant {
  String getViewVariant();
}
```

→ Your RenderNodeDecorator should test for this interface, not for Linkable.

Registering the RenderNodeDecorator

First, we need a Provider for our RenderNodeDecorator

```
public class ViewVariantRenderNodeDecoratorProvider
                        implements RenderNodeDecoratorProvider {
  private static RenderNodeDecorator INSTANCE =
                                  new ViewVariantRenderNodeDecorator();
  @Override
  public RenderNodeDecorator getDecorator(String view, Map model, …) {
        return INSTANCE;
  }
}
```

Default ViewDispatcher Configuration

```
<import resource="classpath:/com/coremedia/cae/view-services.xml"/>

<bean id="viewVariantRenderNodeDecoratorProvider"
      class="c.c.c.view.ViewVariantRenderNodeDecoratorProvider" />

<customize:append id="corediningRenderNodeDecoratorProvidersCustomizer"
                  bean="renderNodeDecoratorProviders">
  <description>
    Append your custom RenderNodeDecoratorProviders here.
    By default the list is empty.
  </description>
  <list>
    <ref bean="viewVariantRenderNodeDecoratorProvider" />
  </list>
</customize:append>
```

View dispatcher extension points are defined here

Our extension

The extension point

Simplify your Templates

→ Simplify Topic.main.jsp as follows:

→ Replace the <c:choose> tag with a simple <c:forEach>

→ Include every container with the main view. Let the view dispatcher make the rest...

Topic.main.jsp:

```
<c:forEach items="${self.contentContainers}" var="container">
  <cm:include self="${container}" view="main" />
</c:forEach>
```

→ Rename the collection templates:
Container.tableOfTeasers.jsp -> Container.main[tableOfTeasers].jsp
Container.listOfTeasers.jsp -> Container.main[listOfTeasers].jsp

Write your own view variant extension

Completing View Variants

The current implementation of View Variants differs from the implementation in the CoreMedia Blueprint in one important aspect:

→ **In CoreDining**
If you want to render a bean which has a view variant, but you don't have a view variant template, the view dispatcher will throw an Exception (No view found)

→ **In CoreMedia Blueprint**
If no view variant template is available, blueprint uses the default view template. Example:
If `Container.main[listOfTeasers].jsp` is missing, use `Container.main.jsp` instead.

This fallback mechanism is implemented by a custom implementation of `ViewLookupTraversal` in CoreMedia Blueprint.

Fallback ViewLookupTraversal for view variants

```
public class FallbackViewLookupTraversal
                              extends RepositoriesFirstViewLookupTraversal {

  public View lookup(List<ViewRepository> repos, Type type, String viewName)
  {
    View v = super.lookup(repos, type, viewName);
    if (v == null) { // no view found...
      int p = viewName.indexOf('['); // has view variant in view name?
      if (p>=0) {
        String defaultViewName = viewName.substring(0,p);
        if (defaultViewName.length()==0) defaultViewName = null;
        return super.lookup(repos, type, defaultViewName); // try again...
      }
    }
    return v;
  }
}
```


CoreMedia

Agenda

1. Framework Basics
2. Content Type Model and Content Beans
3. Template Development
4. Business Logic
5. Request and Link Handlers
6. Page layouts and CSS
7. Modularization
8. Advanced View Programming
9. Caching with Dataviews
10. Integration of External Content

CoreMedia

Caching

caching is enabled by configuration

CAE does *not* cache the rendered results of the (template) views

cached objects reside in memory

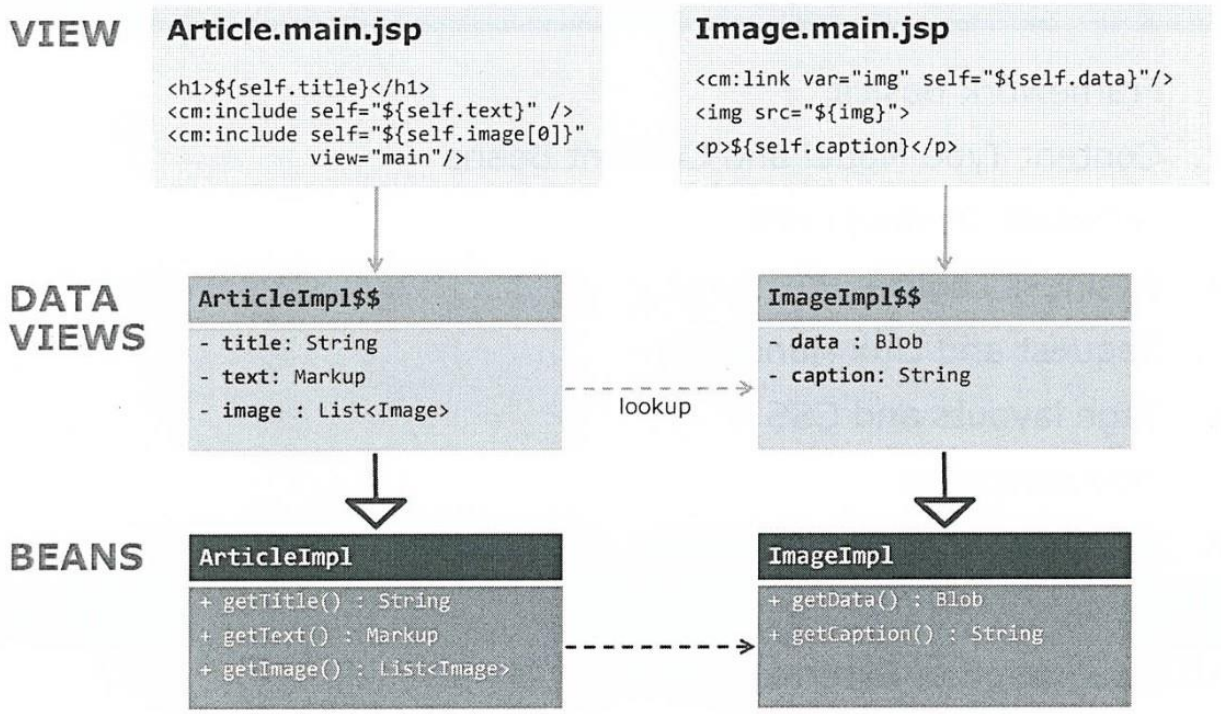
caching does not require any coding

cached objects are called **data views** as they provide a local view to an object

CAE *does* cache *content beans*

data views must not be confused with (template) views

CAE Data views acting as Cache



Configuration of Data View Cache Sizes

- You must specify the number of cached data views for different bean types.
- All dataview'able types must implement AssumesIdentity, so you can use this type for defining the total number of dataviews.
- This configuration is typically done in the same configuration file.

```
<cachesize class="com.coremedia.objectserver.dataviews.AssumesIdentity"
           size="1000"/>
<cachesize class="com.coremedia.coredining.contentbeans.Article"
           size="200" />
<cachesize class="com.coremedia.coredining.contentbeans.Image"
           size="200" />
```

- The example above caches 1000 dataviews in total, but only 200 images and 200 articles as Data Views
- The Cache uses *Least-Recently-Used* as eviction strategy.

Loading of Data Views in Controllers

- In order to benefit from Data View Caching, you need to load a data view for the bean, you want to cache.
- There might be more than one dataview definitions for the same bean type (optional attribute name on <dataview>), but typically you can ignore that.
- This should be initially done in the Controllers/Handlers
- This is automatically done for handler parameters of a type implementing AssumesIdentity (like LinkableImpl!)
- During rendering, Data Views can load other Data Views depending on the specified *association types*.

loads the (default) data view from cache.

```
Topic currentTopic = resolveCurrentTopic(currentTopicPath);
currentTopic = dataViewFactory.loadCached(currentTopic, null);
```

When do you need CAE Caching?

there already is a generic Unified-API cache which does a good job...

objects that come from third-party systems might benefit from caching

computed or synthesized properties that are frequently used, e.g. when building up a navigation

properties, often used in templates

bean properties used for creating links (LinkScheme)

Data views configuration

- caching requires the definition of data views which is located in classpath at /framework/dataviews/dataviews.xml in module **contentbeans** (default)
- The file name and the module can differ in projects.

```
<dataviews xmlns="http://www.coremedia.com/2004/objectserver/dataviewfactory">
  <dataview appliesTo="com.coremedia.coredining.contentbeans.ArticleImpl">
    <property name="title"/>
    <property name="text"/>
    <property name="image" associationType="static" />
    <property name="related" associationType="static" />
    <property name="homeTopic" />
  </dataview>
  <dataview appliesTo="com.coremedia.coredining.contentbeans.ImageImpl">
    <property name="title" />
    <property name="data" />
    <property name="caption" />
  </dataviews>
  …
</dataviews>
```

references to other data views require an association type

you configure a set of properties for a certain content bean that are cached

COREMEDIA

Be careful: Caching can sometimes reveal subtle programming bugs in your beans

Exercise 23

COREMEDIA

Add caching to your navigation

COREMEDIA

CORE MEDIA

Association types are all about the relation between ***TWO*** data views

DataView without Association Types Pseudo Code

```
class ArticleImpl$$ extends ArticleImpl {
   private String title;
   private Image image;

   public void $$load() {
     this.title = super.getTitle();
     this.image = super.getImage();
   }

   public String getTitle() {
       return this.title;
   }

   public Image getImage() {
       return this.image;
   }
}
```

- The properties "*title*" and "*image*" are cached by this dataview
- The Method "getImage()" returns a ContentBean
- **Problem:** The dataview of the "*image*" is never loaded

Overview: Association types for links to other content beans

- **Static**
  - holds ***reference to bean***
  - queries cache upon every request to property
  - returns data view
- **Dynamic**
  - ***does not hold a reference*** to neither bean nor data view
  - queries cache upon every request to property
  - returns data view
- **Aggregation**
  - holds ***reference to data view***
  - data view is ***shared*** with all other data views
- **Composition**
  - holds ***reference to data view***
  - data view is **not *shared*** with all other data views

Association Type "*composition*" Pseudo Code

```
class ArticleImpl$$ extends ArticleImpl {

  private ImageImpl$$ image;

  public void $$load() {
    Image imageContentBean = super.getImage();
    this.image = new ImageImpl$$(imageContentBean);
  }

  public Image getImage() {
    return this.image;
  }
}
```

- both data views are strongly coupled.
- rootTopic can not be shared by other topic.

Association Type "*aggregation*" Pseudo Code

```
class ArticleImpl$$ extends ArticleImpl {

  private ImageImpl$$ image;

  public void $$load() {
    Image imageContentBean = super.getImage();
    this.image = cache.load( imageContentBean );
  }

  public Image getImage() {
    return this.image;
  }
}
```

- both data views are strongly coupled.
- rootTopic DV is loaded from cache, can be shared!

Association Type "*static*" Pseudo Code

```
class ArticleImpl$$ extends ArticleImpl {

  private ImageImpl image;

  public void $$load() {
      this.image = super.getImage();
  }

  public Image getImage() {
    return cache.load( this.image );
  }
}
```

- both data views are loosely coupled.
- invalidation of image does not fire invalidation of the article.
- **most common association type!**

Association Type "*dynamic*" *Pseudo Code*

```
class ArticleImpl$$ extends ArticleImpl {

  // no attribute

  // no load method

  public Image getImage() {
     Image imageContentBean = super.getImage();
     return cache.load( imageContentBean );
  }
}
```

- both data views are not coupled at all.
- assures, that the return value of getRootTopic() is also a Data View.
- **should be used for dynamic calculated properties! (e.g. personalization)**

Typical Use Cases for Association Types

- **Static**
  - Static is the standard association type.
  - All kind of link lists between related, but independent content beans
- **Aggregation**
  - Links to content beans which are not changing very often
  - Symbols, languages, ...
- **Composition**
  - For links between two objects, which are dependent of each other.
  - Article with a picture which is only used there, e. g. for a specific machine part.
- **Dynamic**
  - Permanently changing banner ad.
  - Rotating contents.

Association type: Comparison

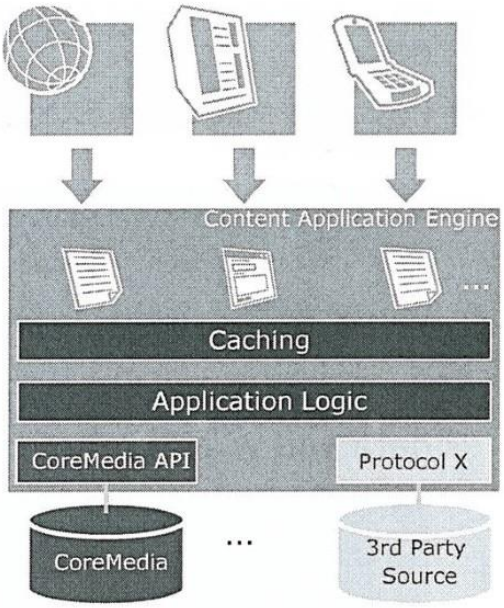
Association Type	Loads data view from Cache	Holds reference to	Implies Cache dependency to
Static	Yes	Content Bean	Content Bean
Dynamic	Yes	Nothing	None
Aggregation	Yes	Data View	Content Bean and Data View
Composition	No	Data View	Content Bean (and Data View)

Agenda

1. Framework Basics
2. Content Type Model and Content Beans
3. Template Development
4. Business Logic
5. Request and Link Handlers
6. Page layouts and CSS
7. Modularization
8. Advanced View Programming
9. Caching with Dataviews
10. Integration of External Content

Third party integration: Layers

- **Layer I: Data Sources**
  - CoreMedia Content Server
  - 3rd Party Systems (e.g. RDBMS)
- **Layer II: API**
  - CoreMedia Unified API
  - 3rd Party API (e.g. SQL/JDBC)
- **Layer III: Application Logic**
  - encapsulates application and business logic
  - separates logic from rendering
- **Layer IV: Data Views and Caching**
  - automatic dependency tracking
  - time- and event-based invalidations for 3rd party components
- **Layer V: Rendering and Delivery**



You will always need a **link** between internal and external data

Steps to integrate third party data

You will always need a **link** between internal and external data

→ This link will need something like a **primary key** for external data

→ Possible solution: Extend your document model and supply this information

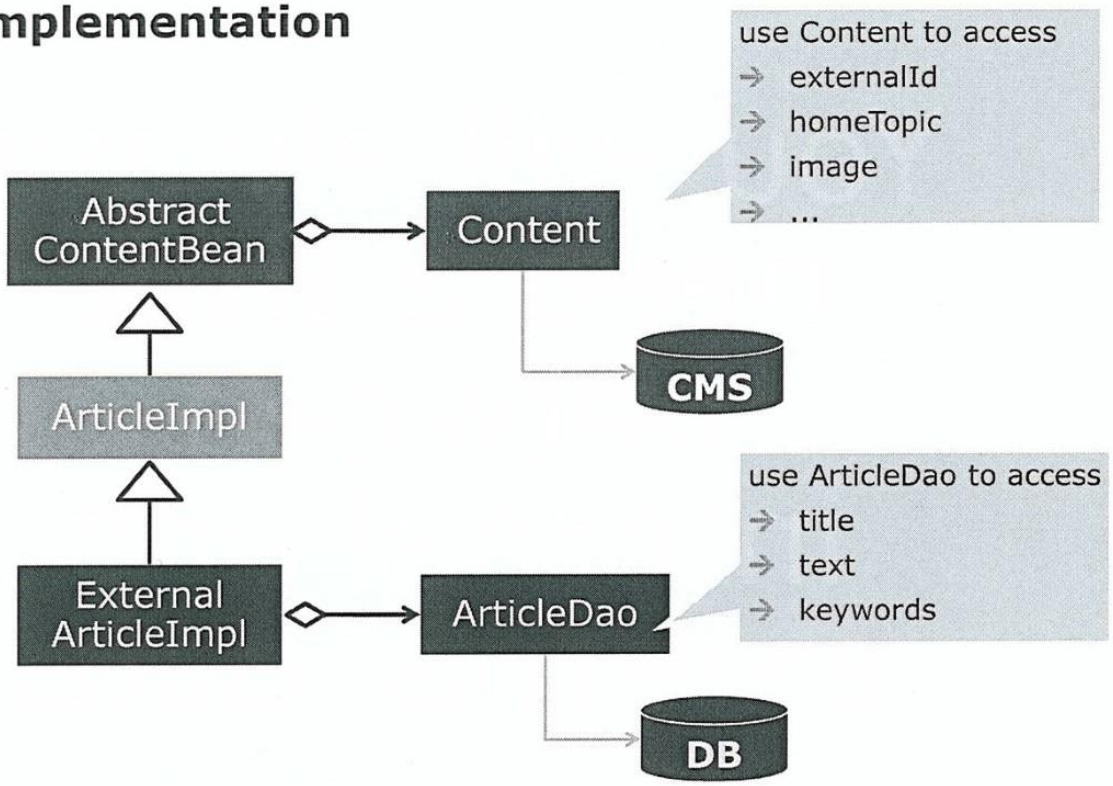
```
<DocType Name="ExternalArticle" Parent="Article">
    <StringProperty Length="20" Name="externalId"/>
</DocType>
```

→ In the bean implementation (ExternalArticleImpl) for external data you will have to make use of this primary key

Best Practice:

→ Use a DAO
→ Let the bean work against a DAO interface
→ Let the DAO return a composition of all relevant data
→ Configure the external bean plus the DAO in the project specific contentbeans.xml

External Articles Implementation



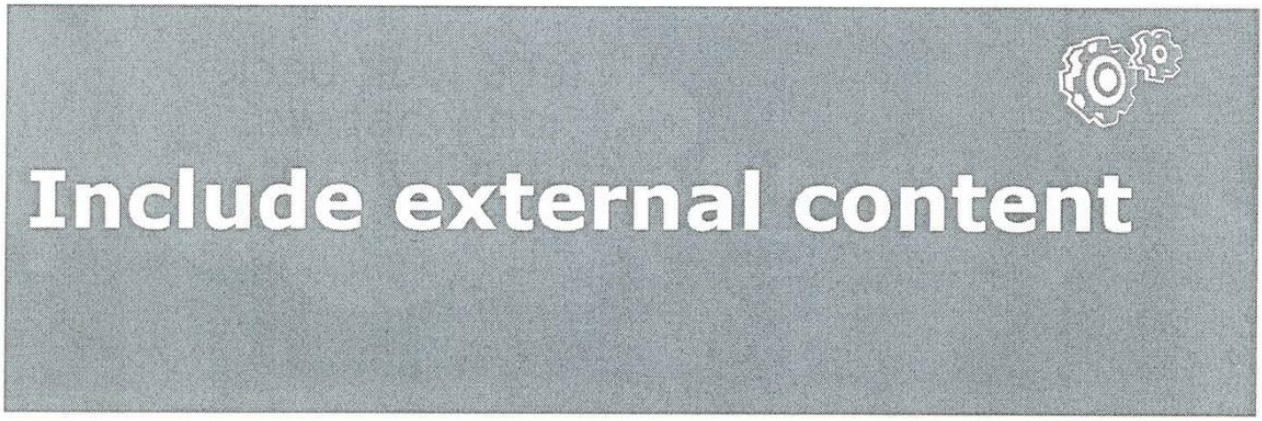
Registering the External Article

```
<bean name="contentBeanFactory:ExternalArticle" scope="prototype"
      class="com.coremedia.coredining.contentbeans.ExternalArticleImpl>
  <property name="articleDAO" ref="articleDAO" />
</bean>

<!-- the data access object for articles -->

<bean id="articleDAO" class="com.coremedia.coredining.dao.JdbcArticleDAO">
  <property name="url"
            value="jdbc:postgresql://localhost:5432/coredining"/>
  <property name="driver" value="org.postgresql.Driver"/>
  <property name="login" value="external" />
  <property name="password" value="external" />
</bean>
```


COREMEDIA

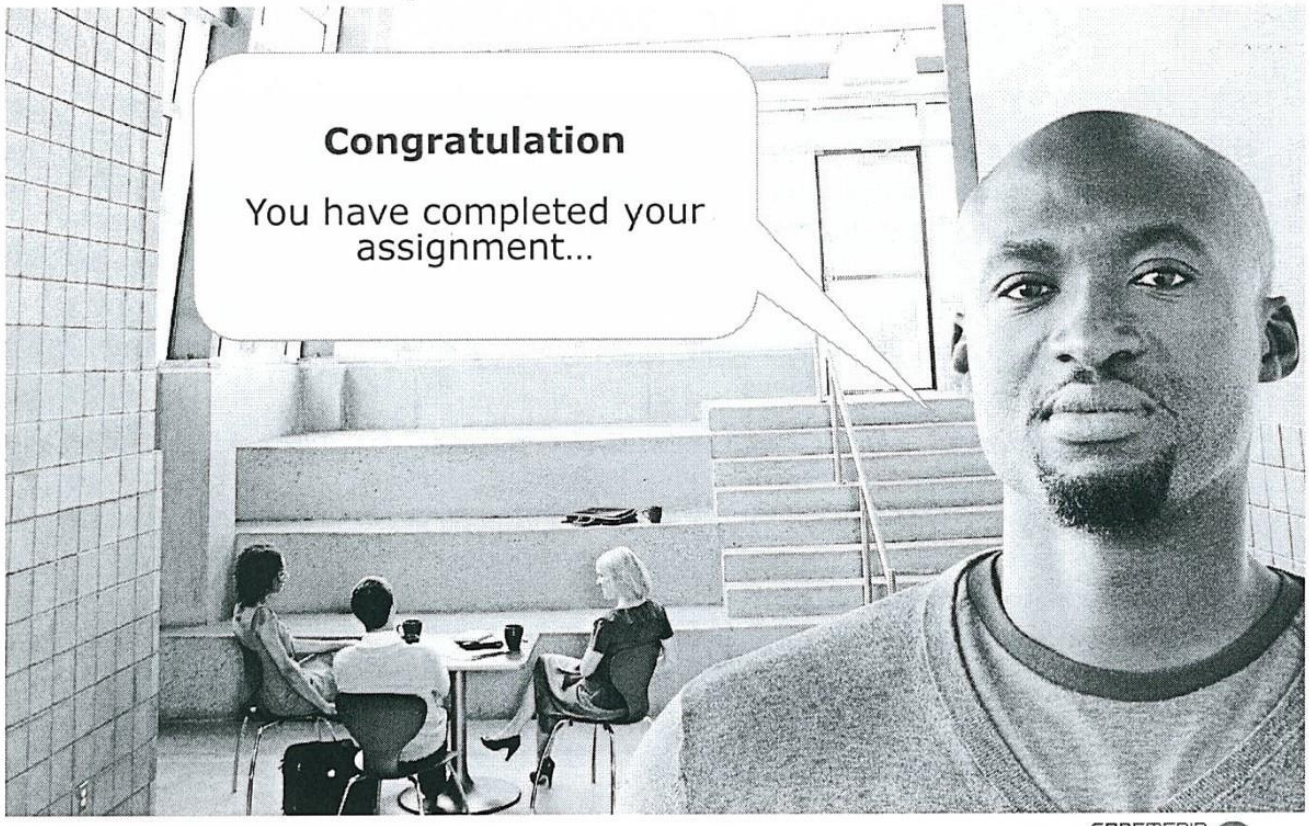
Exercise 24

COREMEDIA



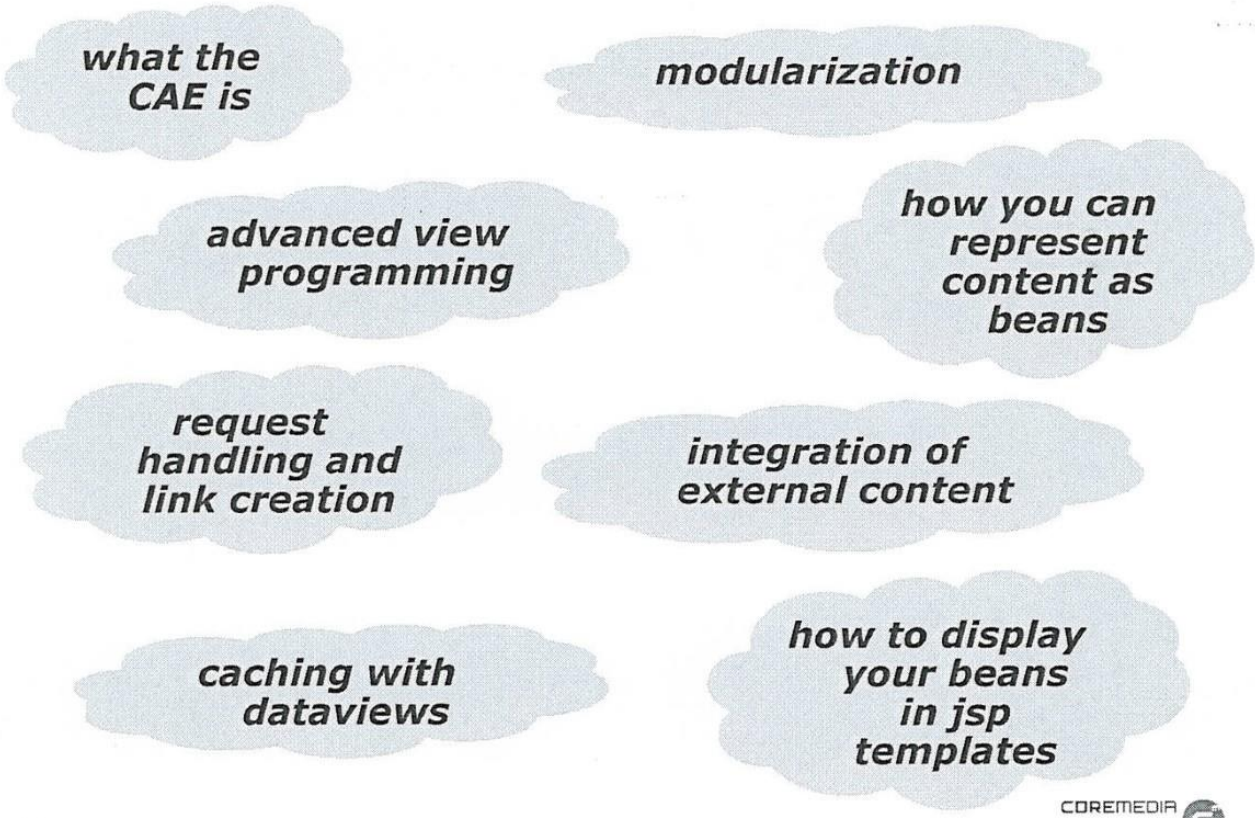
COREMEDIA

Mission accomplished



Reflection

What have you learned?



CoreMedia Training Program