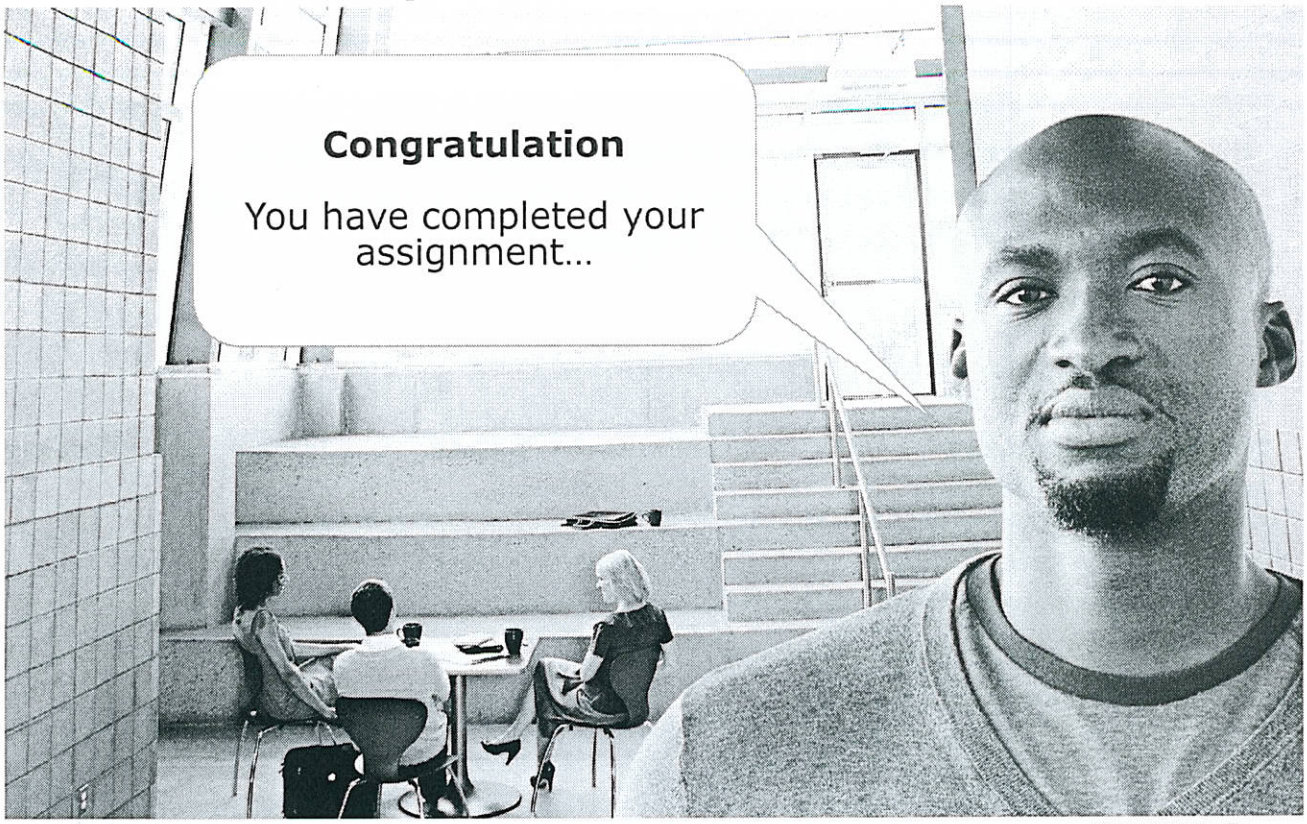
Mission accomplished



Reflection

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

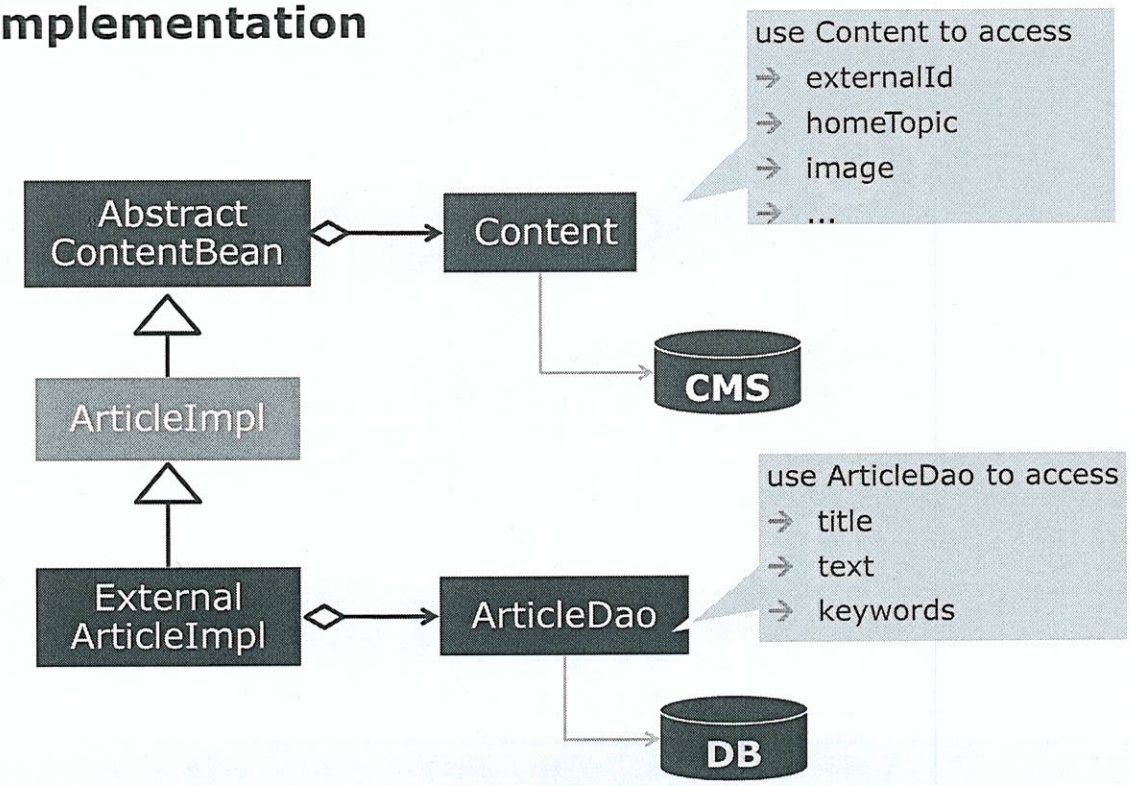
```
<DocType Name="ExternalArticle" Parent="Article">
    <StringProperty Length="20" Name="externalId"/>
</DocType>
```

- 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



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

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!**

CoreMedia

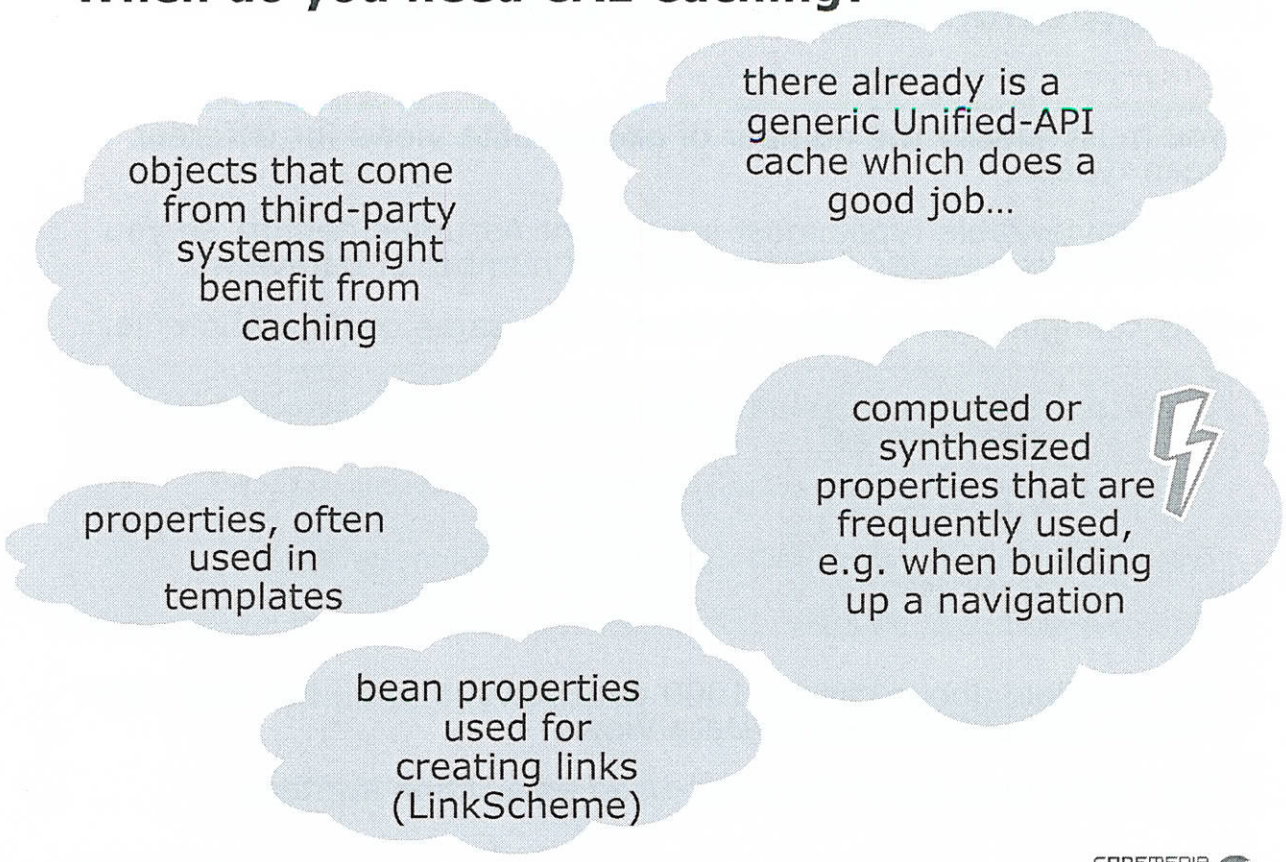
Association types are all about the relation between *TWO* data views

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

When do you need CAE Caching?



COREMEDIA

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

Caching

caching is enabled by configuration

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

cached objects reside in memory

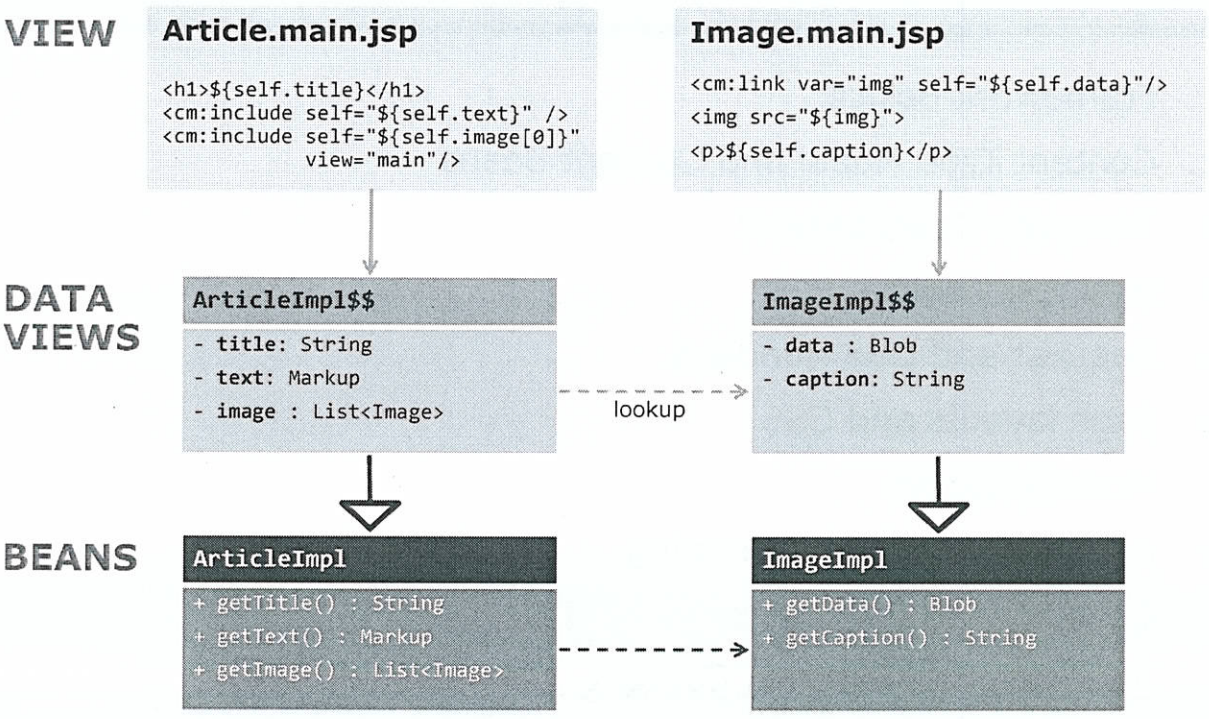
caching does not require any coding

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

CAE *does* cache *content beans*

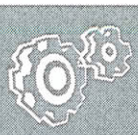
data views must not be confused with (template) views

CAE Data views acting as Cache



COREMEDIA

Write your own view variant extension



COREMEDIA

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.

COREMEDIA

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;
  }
}
```

Extension Points of the ViewResolver

com.coremedia.objectserver.view.ViewDecorator

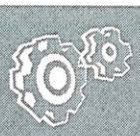
- Is used to decorate a view (e.g. add debug information to output)
- Method: `View decorate(View originalView);`
- Extension Point (Spring): `viewDecorators : List<ViewDecorator>`

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>`

COREMEDIA

Add a Servlet View to render pages as PDF

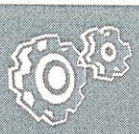


COREMEDIA

COREMEDIA

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

COREMEDIA



Display Articles as Teasers with a non JSP view

COREMEDIA

COREMEDIA

Aim: Display your Articles as PDF

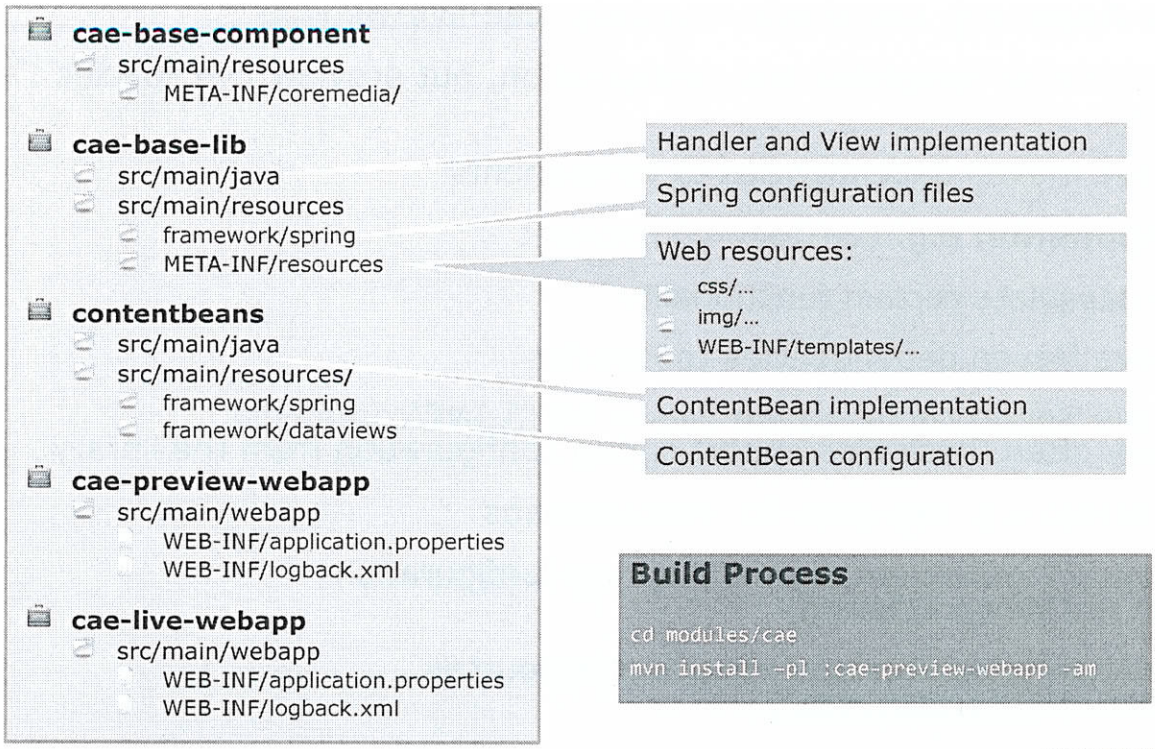
COREMEDIA

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...

Folder Structure in CAE module



Build Process

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


Exercise 19

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

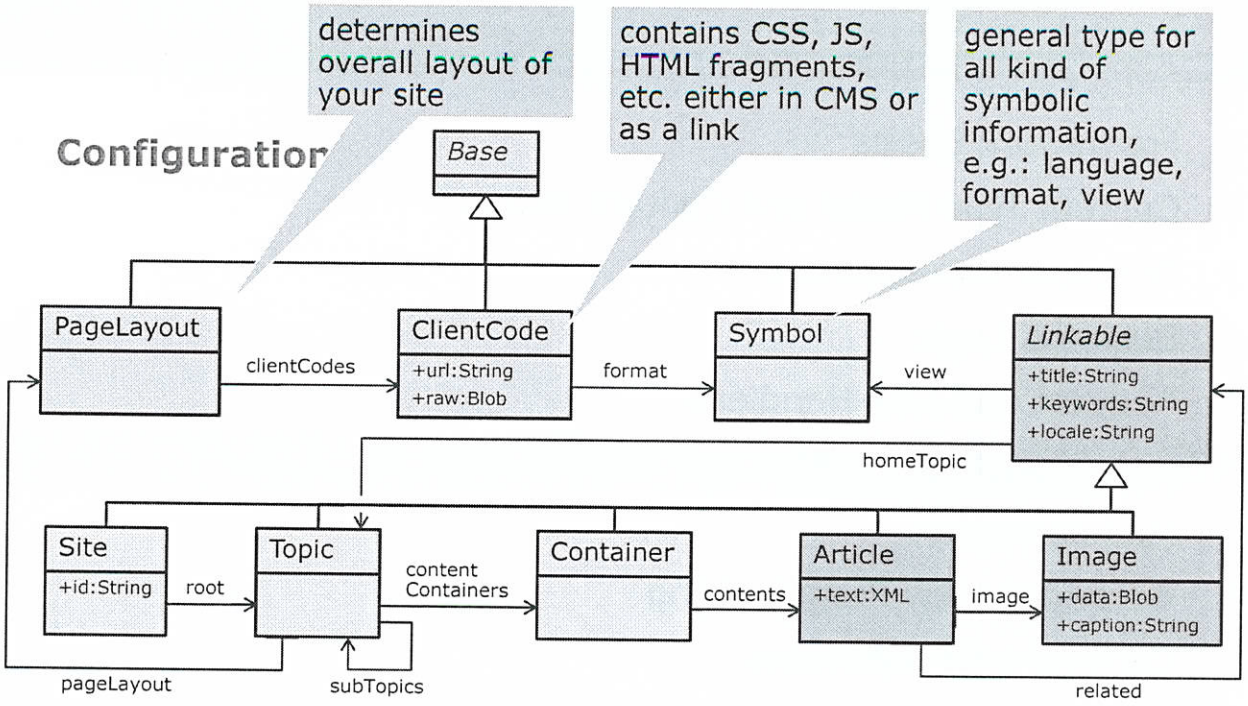
→ Web-resources need to be placed in classpath below "META-INF/resources/"

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

Core.Dining Content Type Model



COREMEDIA

Exercise 18

COREMEDIA

Add dynamic page formatting based on CMS content

COREMEDIA

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

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)
  {…}

  ....

}
```


CoreMedia

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

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...

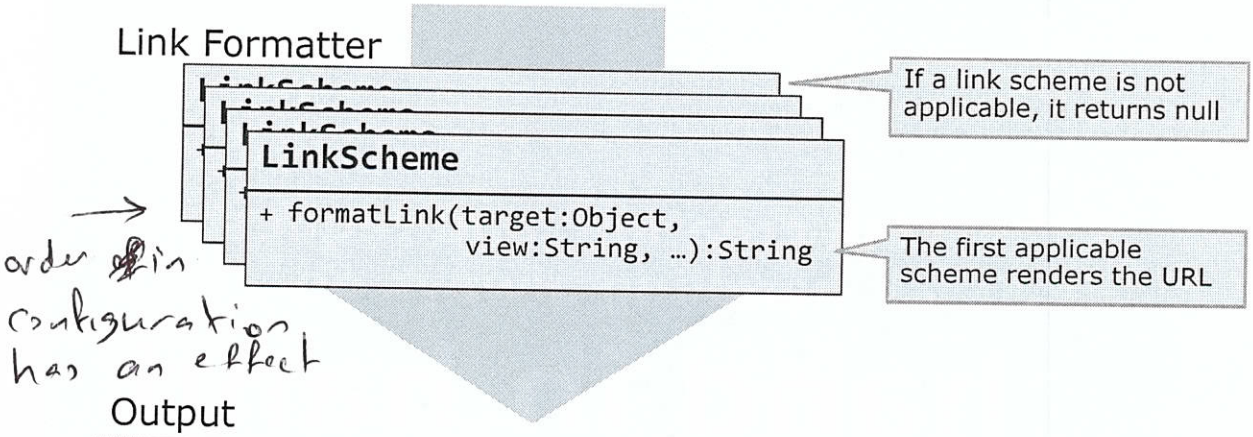
```
UriComponentsBuilder builder = UriComponentsBuilder.newInstance();

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

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

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

What does a LinkScheme do?

Template

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

Link Formatter



Output

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


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!

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)

✓ Core media

- 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

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.

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)

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

Agenda

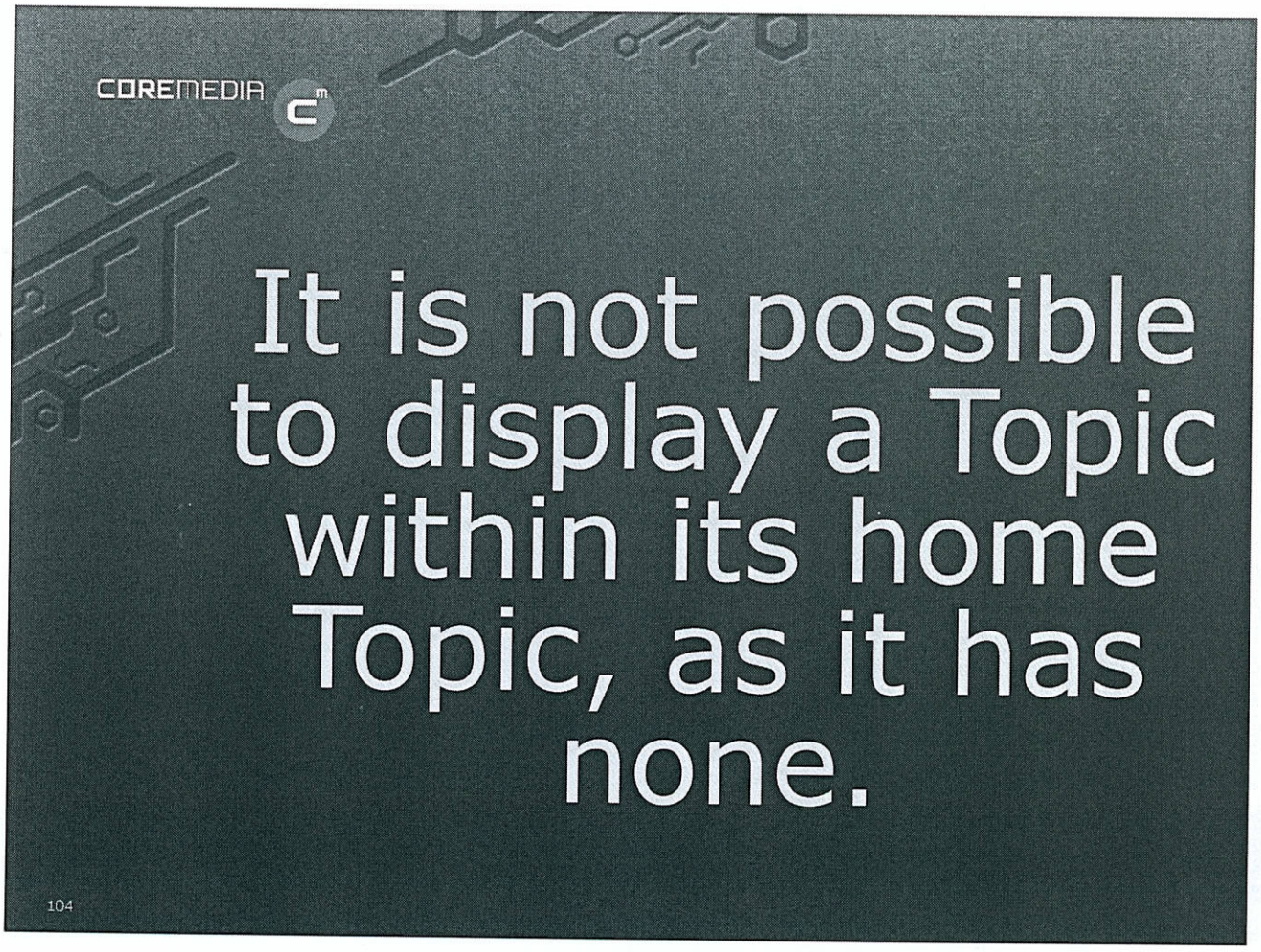
1. Framework Basics
2. Content Type Model and Content Beans
3. Template Development
4. Business Logic
5. Request and Link Handlers
6. Page layouts and CSS
7. Modularization
8. Advanced View Programming
9. Caching with Dataviews
10. Integration of External Content

5. Request and Link Handlers

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

Exercises 15

Create templates and code for rendering the navigation



COREMEDIA

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

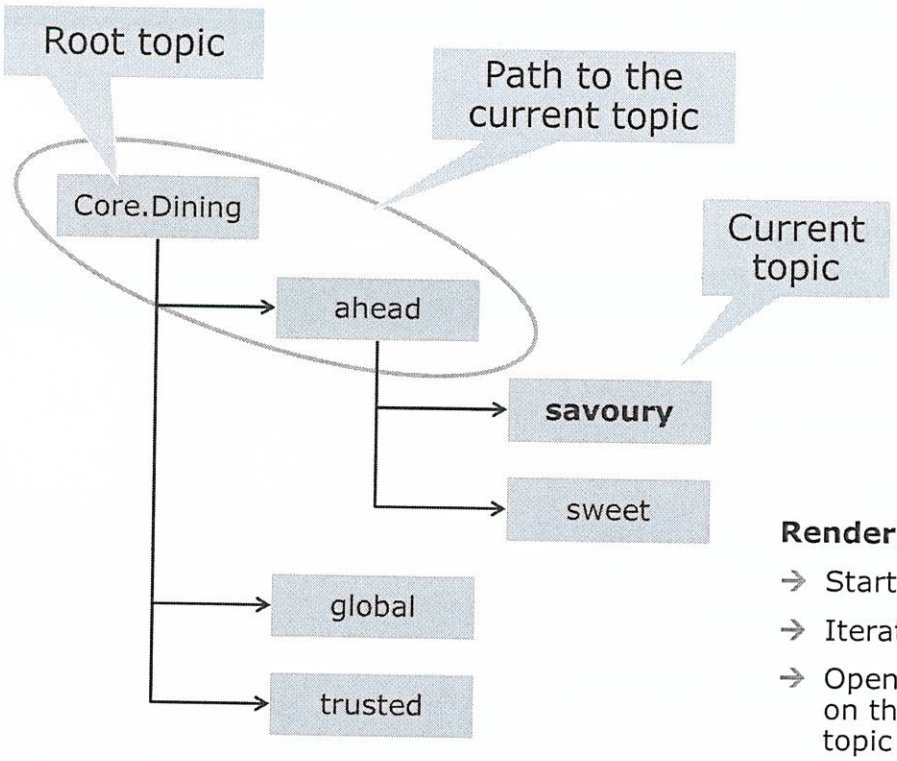
COREMEDIA

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

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();
```

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)

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

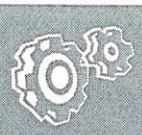
- you start with the root Topic
- it will not be displayed as link
- but all its sub topics will...

```
<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

Create Templates for Overview Pages and Article Teasers



COREMEDIA

Proposed steps towards the solutions

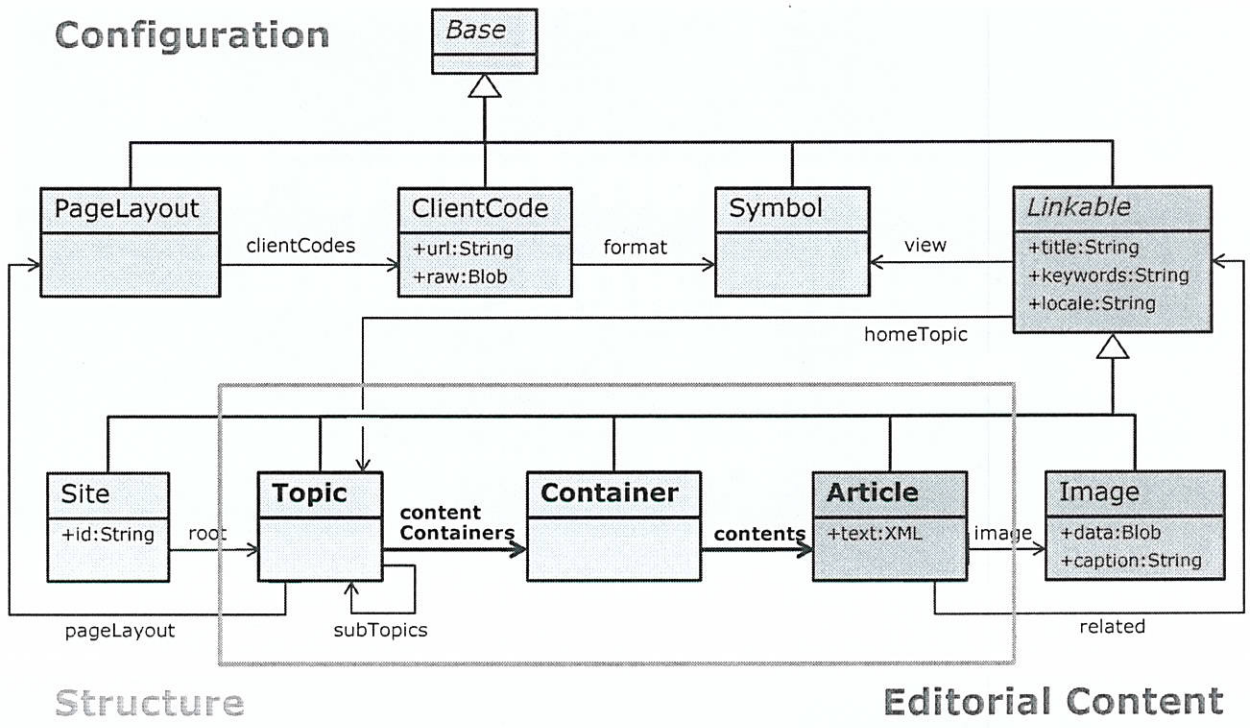
Task	Technical Solution	
Step #1 Content type model	→ Use the pre-built content type model already provided	☑
Step #2 Basic rendering and layout	→ Basic Content bean generation and configuration → Basic JSP programming	☑
Step #3 Rendering the complex navigation	→ Extending content beans with business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

COREMEDIA

CORE MEDIA

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

Core.Dining Content Type Model



CORE MEDIA

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

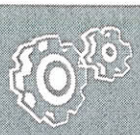
```
LinkableImpl extends LinkableBase implements Linkable
```

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

Exercises 10

Add a different view for an article

COREMEDIA



Adding checks and loops...

COREMEDIA

COREMEDIA

Stepping up one level: Understand how the View Dispatcher works

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.

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"

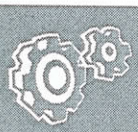
→ the value is given in "value"

→ 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

COREMEDIA



Develop the first version of your application

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

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

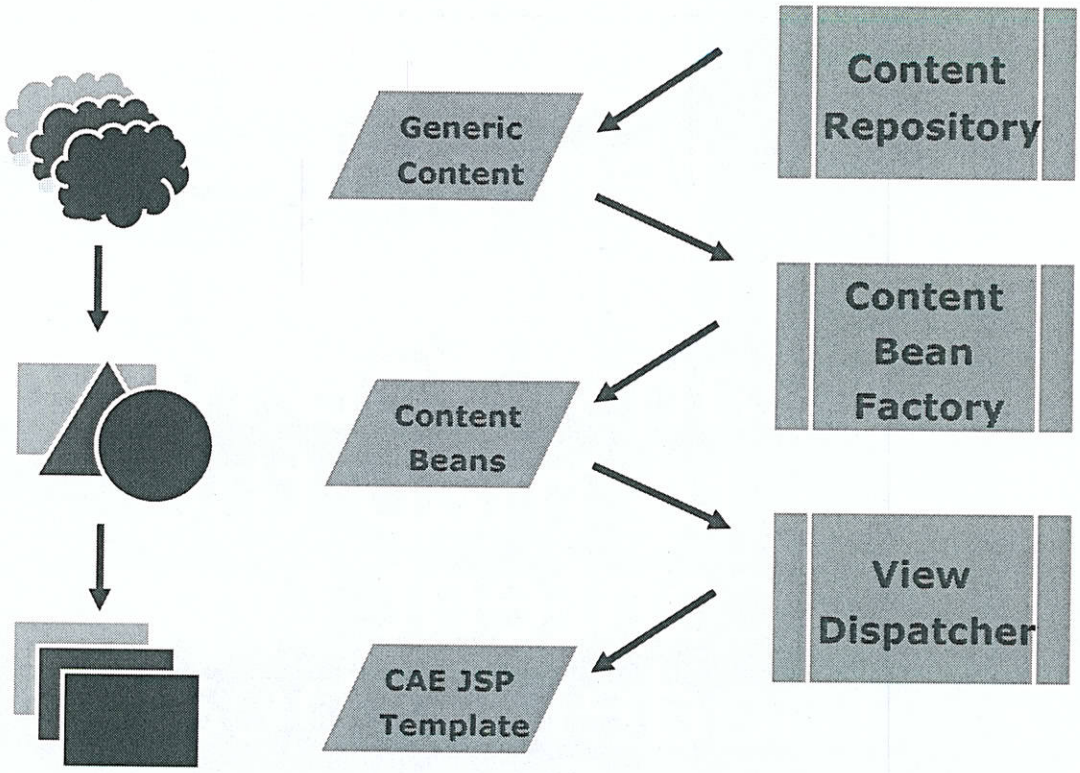
```
<bean name="contentBeanFactory:Article"
  scope="prototype"
  class="com.coremedia.coredining.contentbeans.ArticleImpl">
</bean>
```


Exercises 4 + 5

COREMEDIA

Let a tool generate your content beans and define a mapping from content type to bean type

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

COREMEDIA

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

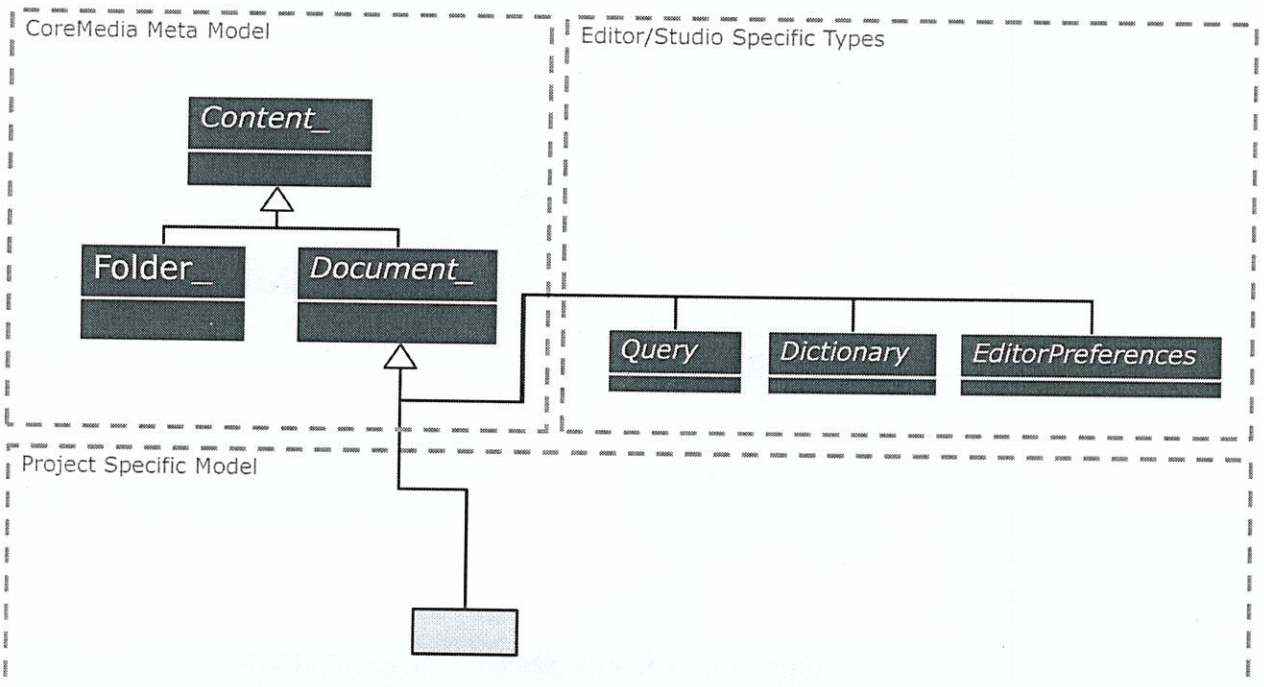
COREMEDIA

You can use it as a reference throughout the whole course

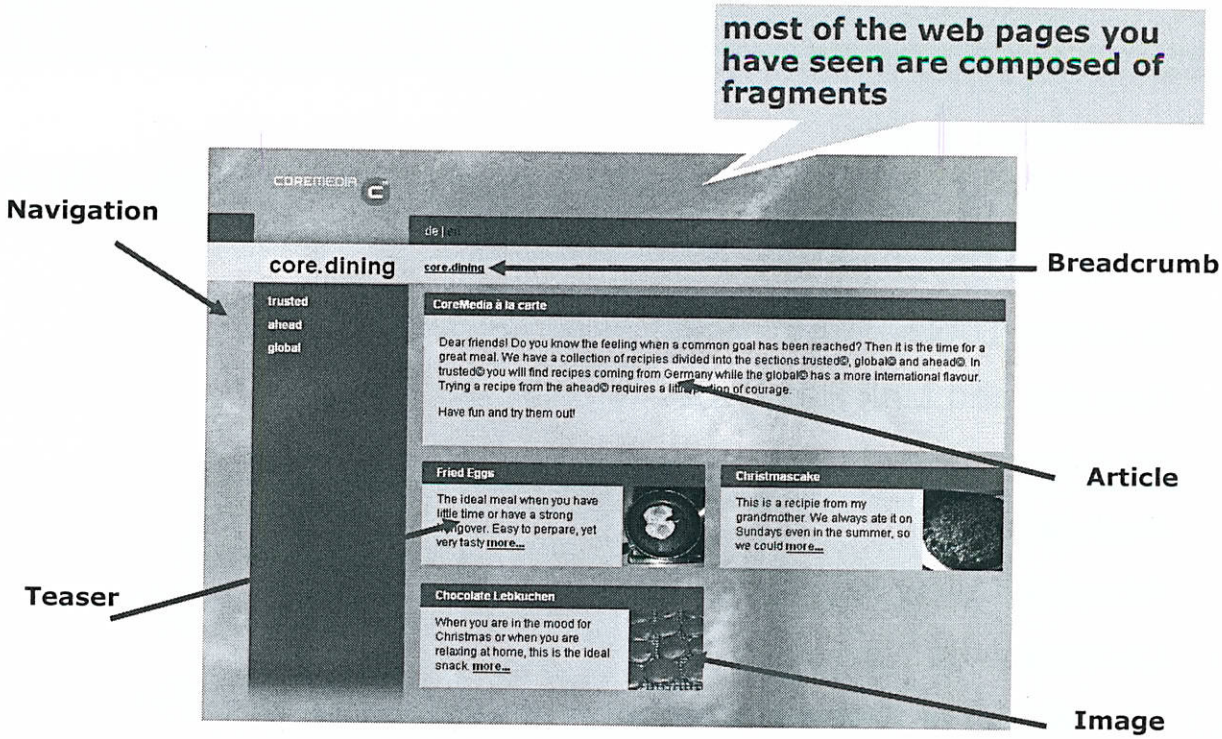
2. Content Type Model and Content Beans

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

CoreMedia Content Model



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

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>
```

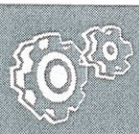
- expressions in the JSTL expression language (EL) are enclosed in ${ ... }
- they can be directly embedded into JSP pages

- 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

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

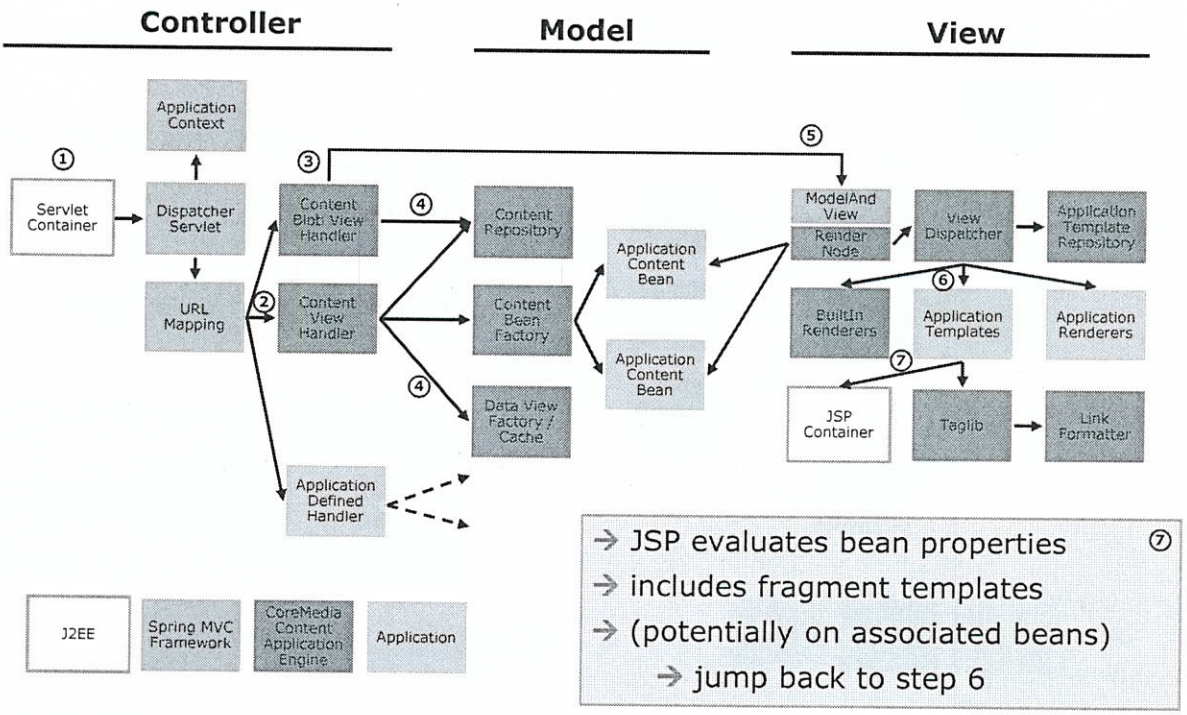
COREMEDIA

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

A typical CAE request

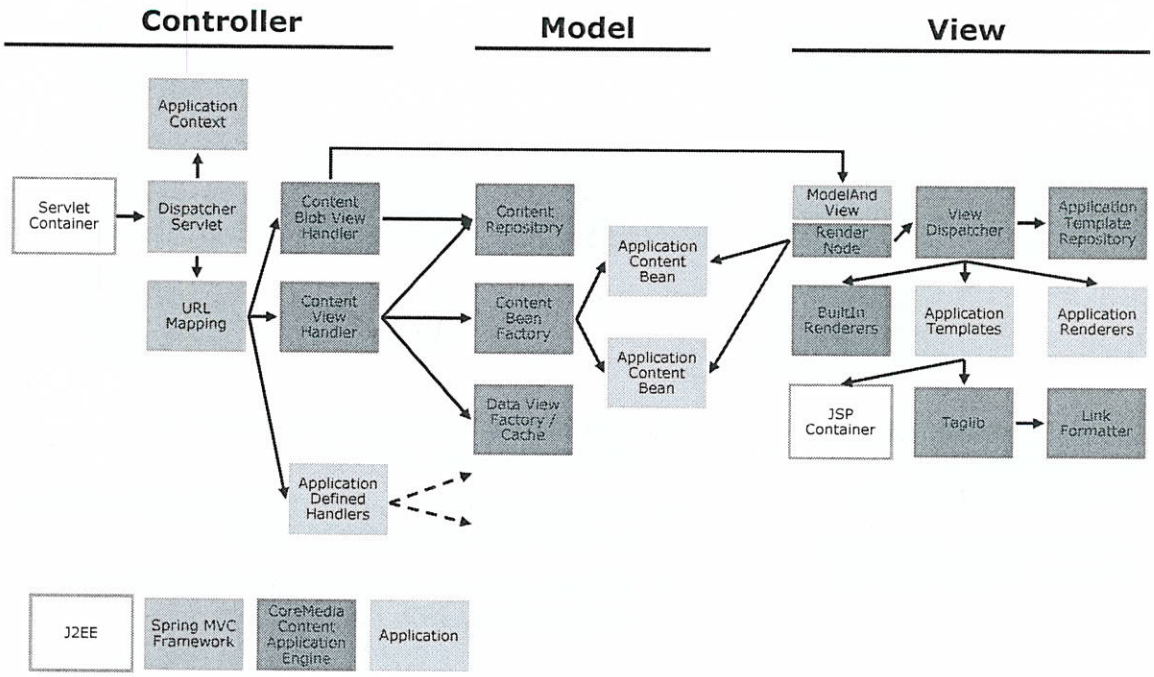


COREMEDIA

COREMEDIA

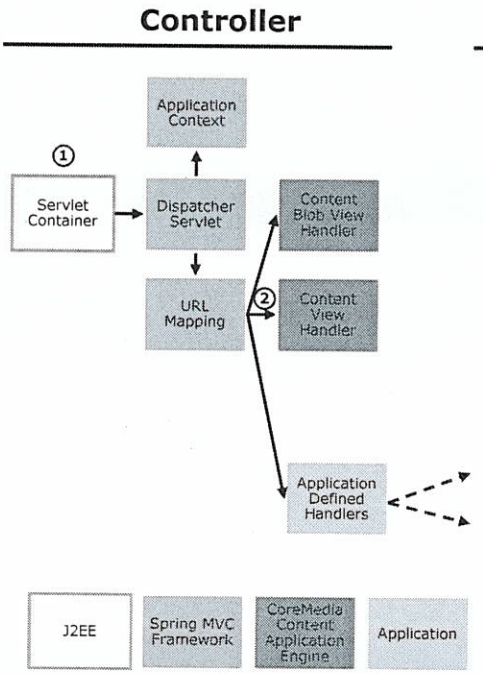
You should receive a color printout of this overview **now**

CAE architecture (fine grain)



CONTENTMEDIA

A typical CAE request



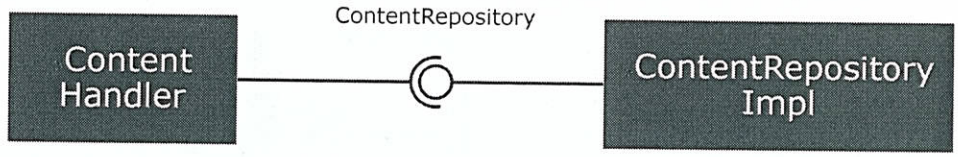
request is passed from servlet container to Spring dispatcher servlet ①

dispatcher servlet dispatches request to a handler using URL mapping ②

CONTENTMEDIA

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

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

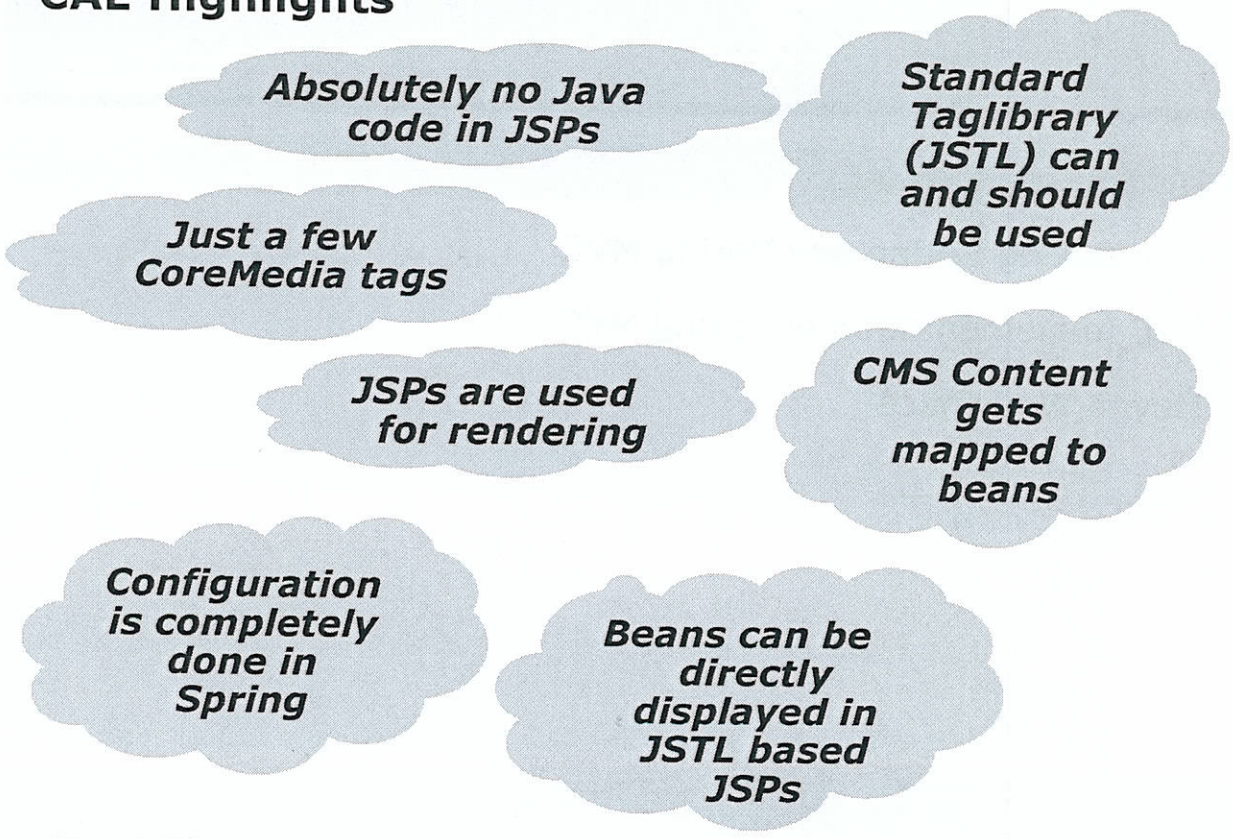
Features

→independent layers

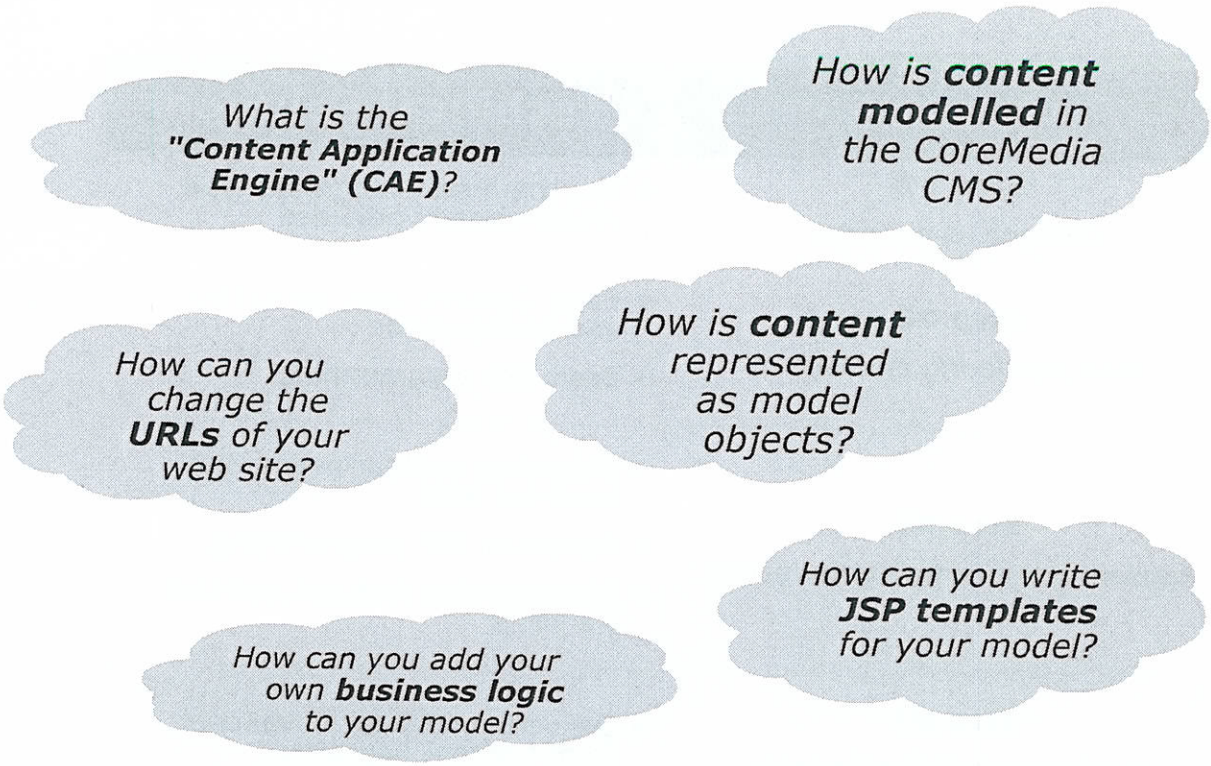
→highly effective and configurable caching

→can render any bean

CAE Highlights



What can this course teach you?

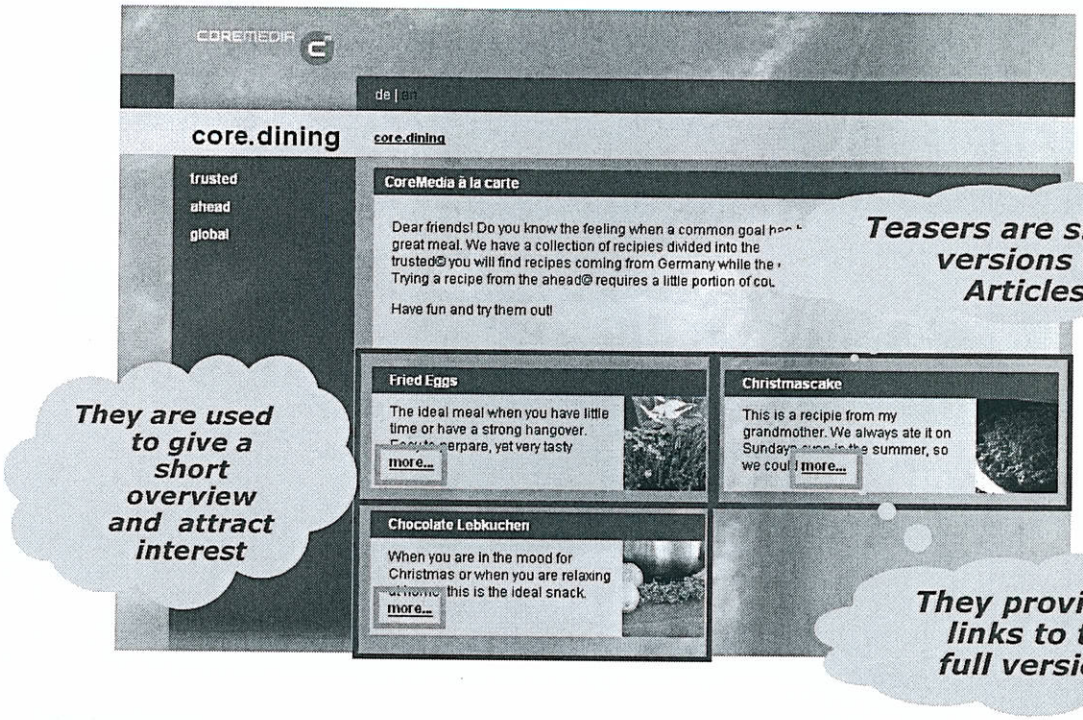


Agenda

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***

core.dining: Teasers



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

COREMEDIA

COREMEDIA

You might need some planning now

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