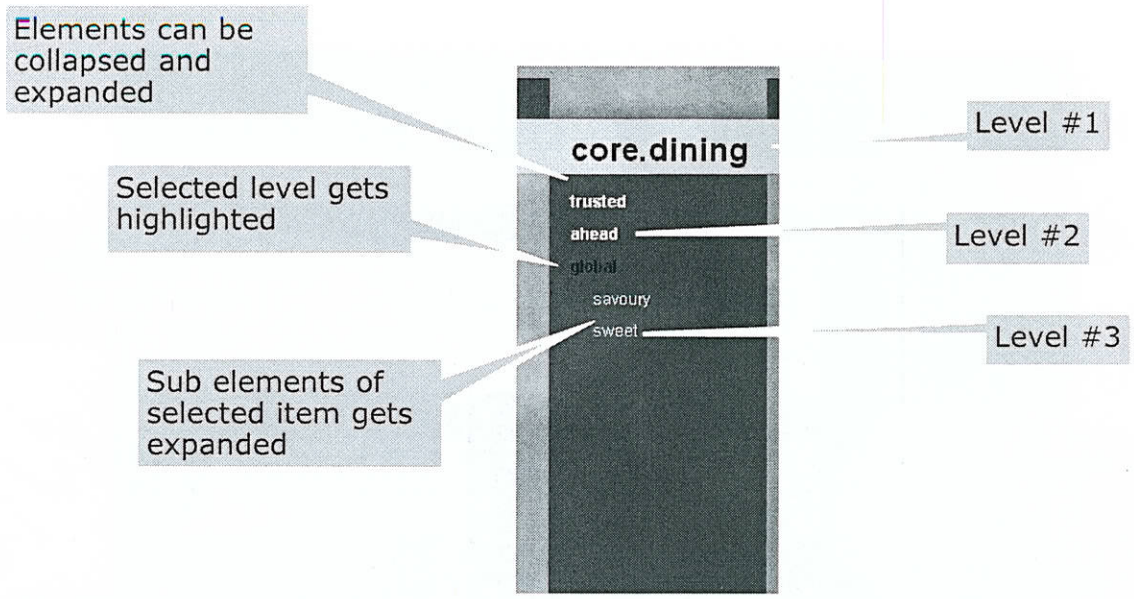
Web Application Development Training

Developing Web Applications with the Content Application Engine

Your Assignment: Create a site guided by a click dummy

core.dining: Navigation

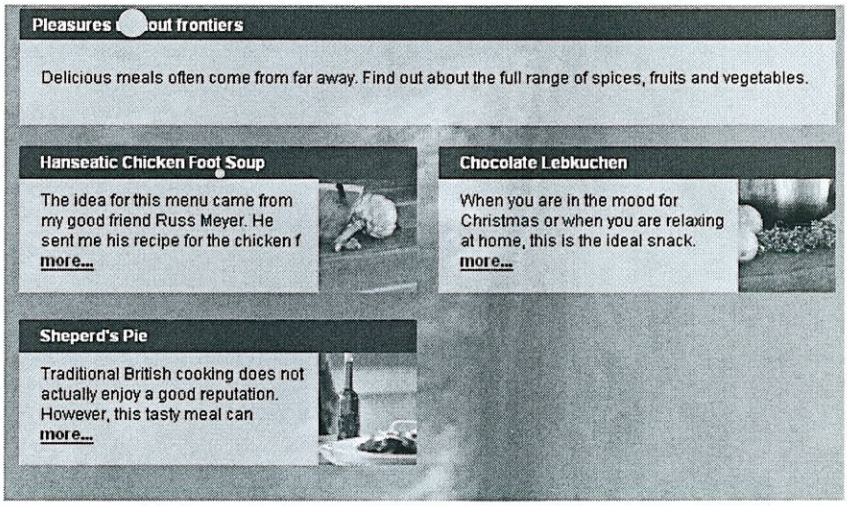


COREMEDIA

core.dining: Main area

a two column list of teasers

composed of a large size intro and ...



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

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...

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

1. Framework basics

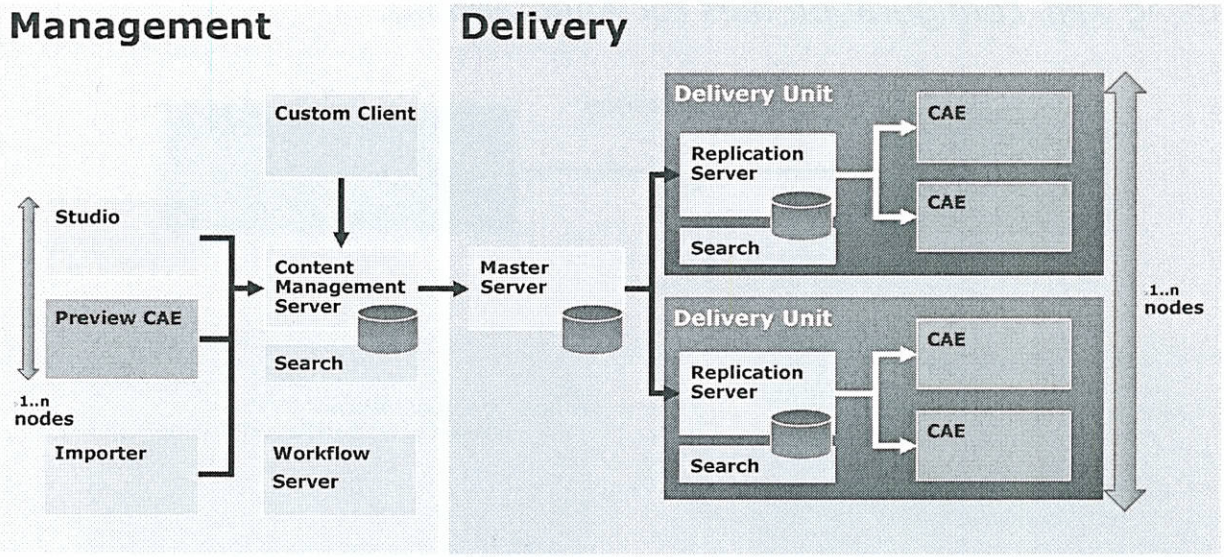
→ Introducing the CAE

→ Introducing Spring and Spring MVC

→ CAE Implementation of Spring MVC

→ Maven Workspace

CoreMedia Architecture (coarse grain)



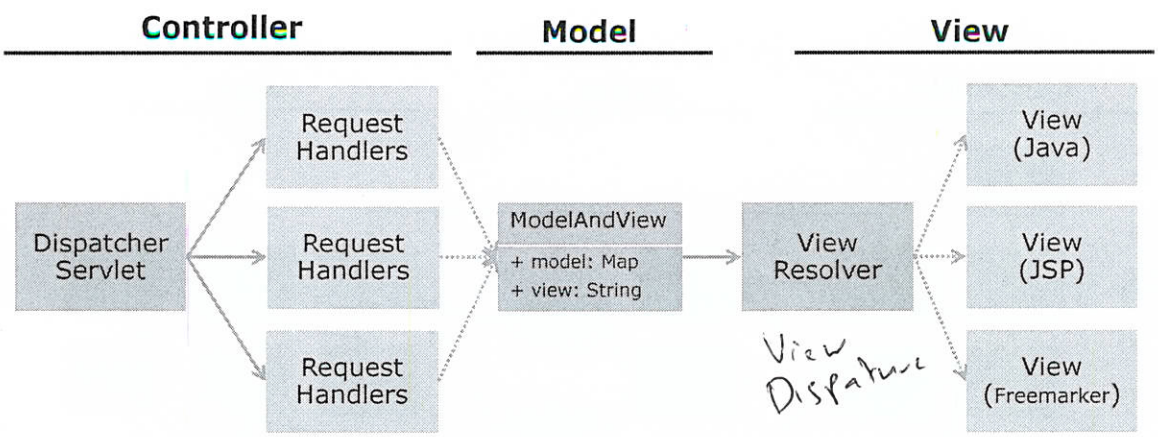
CoreMedia

1. Framework basics

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

CoreMedia

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, ...

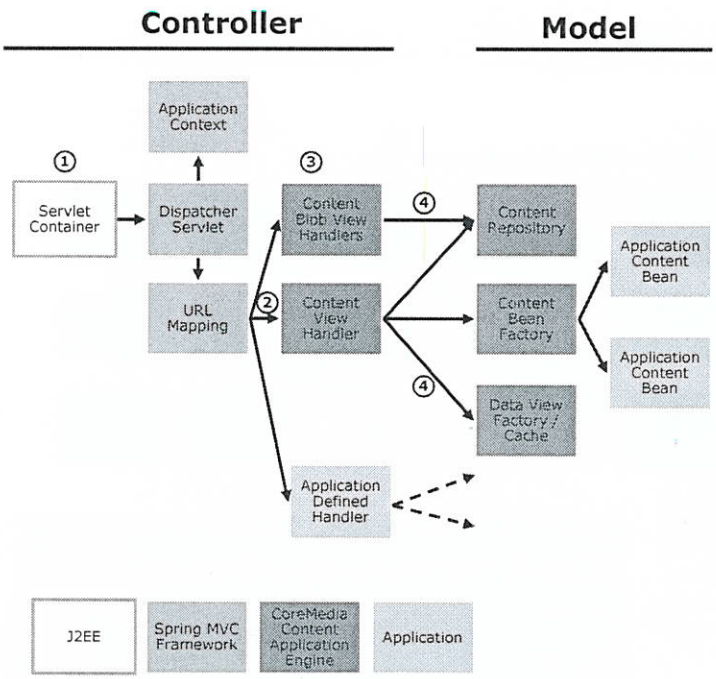
CoreMedia

1. Framework basics

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

CoreMedia

A typical CAE request



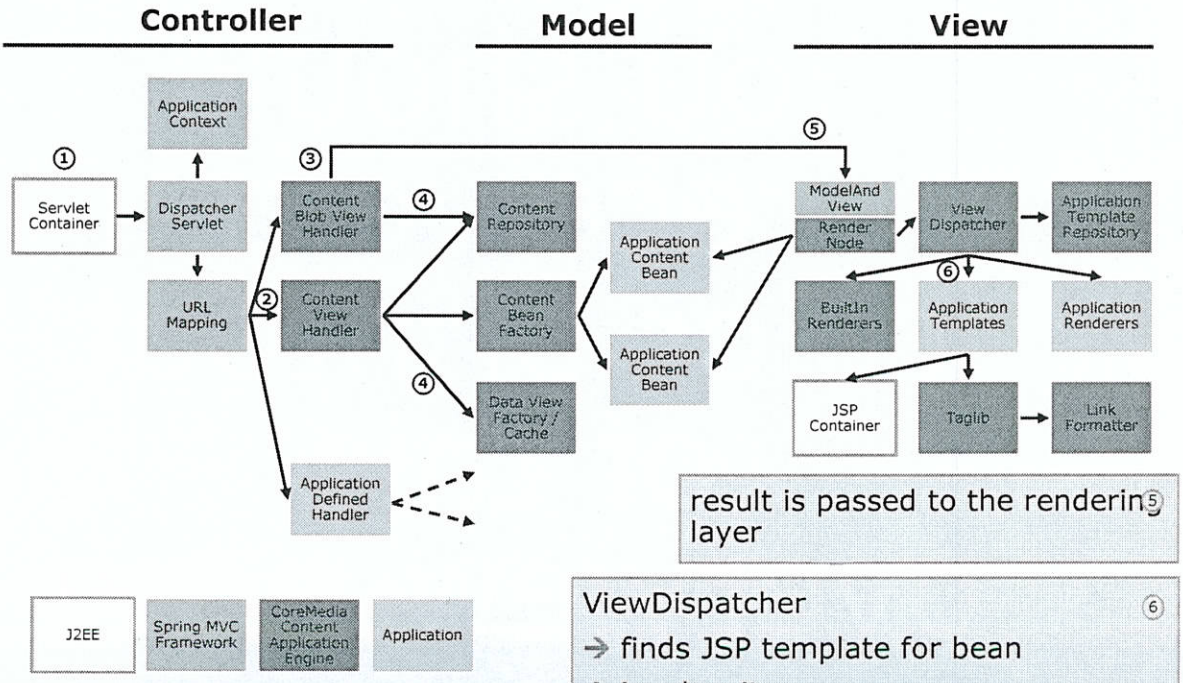
controller parses the request uri and extracts ③

- → content id,
- → view name, and
- → possibly other parameters

- → controller maps content id to content bean ④
- → if configured it also tries to look up a cached data view

CoreMedia

A typical CAE request



result is passed to the rendering layer ⑤

ViewDispatcher ⑥

- → finds JSP template for bean
- → invokes it

CoreMedia

COREMEDIA

You can use it as a reference throughout the whole course

COREMEDIA

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

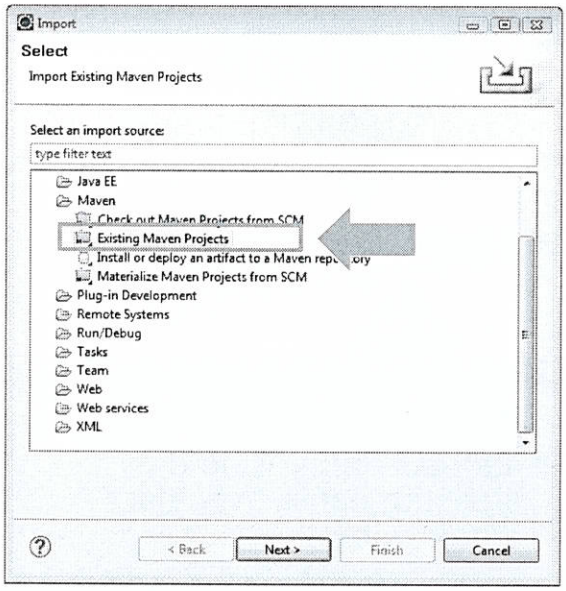
CoreMedia Project Workspace

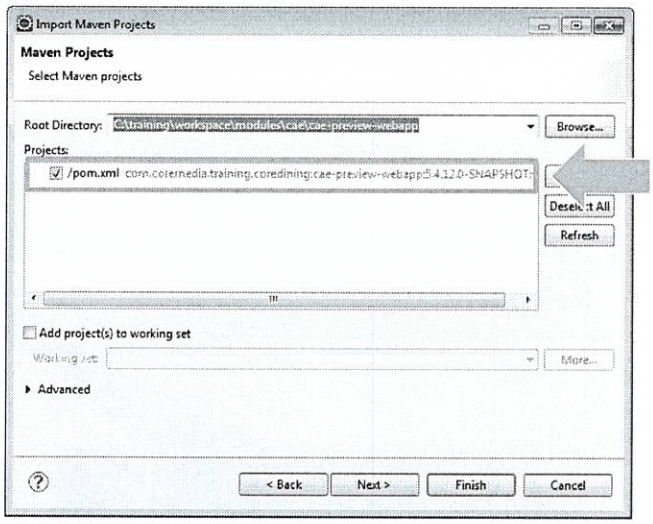
- **workspace**
  - **modules**
    The development workspace of the project.
    This is where you develop your CM application
  - **packages**
    The deployment workspace of the project.
    This is where you configure your RPMs. We are not going to use this module in this course.
  - **boxes**
    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

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
```

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

all modules

```
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

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

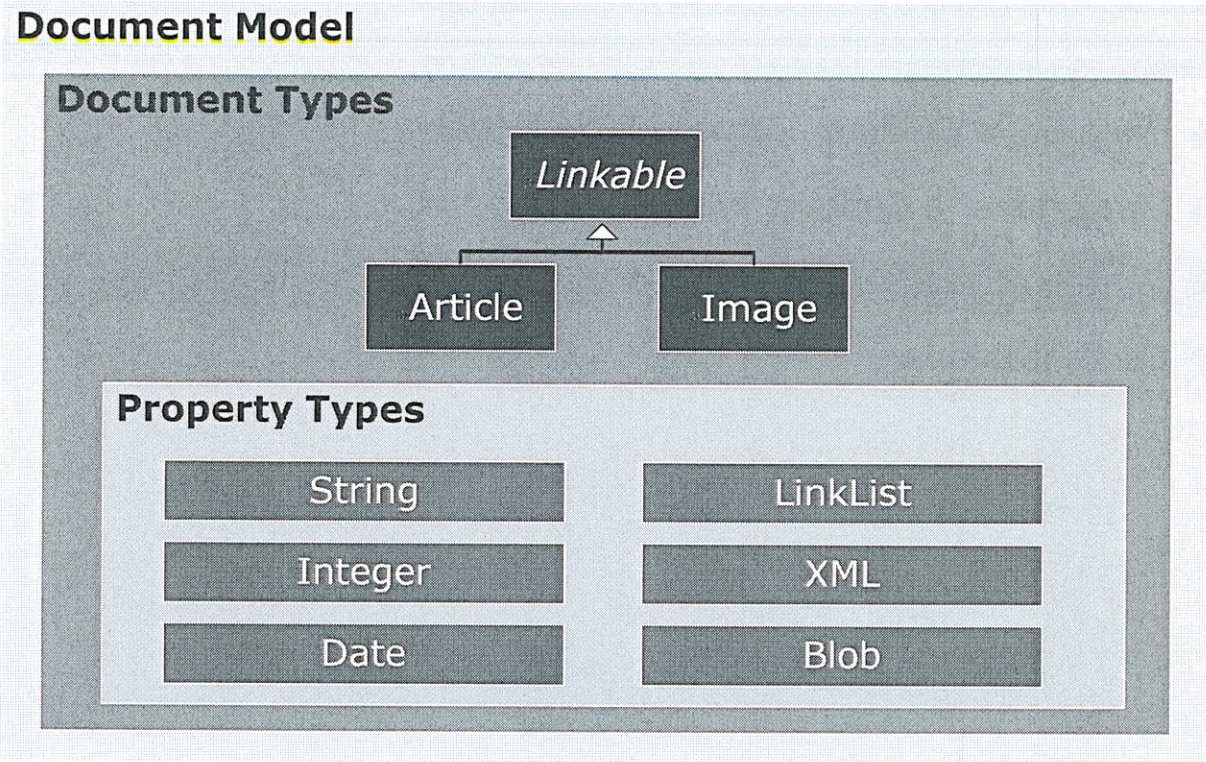
→ From Style Guide to Content Type Model

→ Elements of a Content Type Model

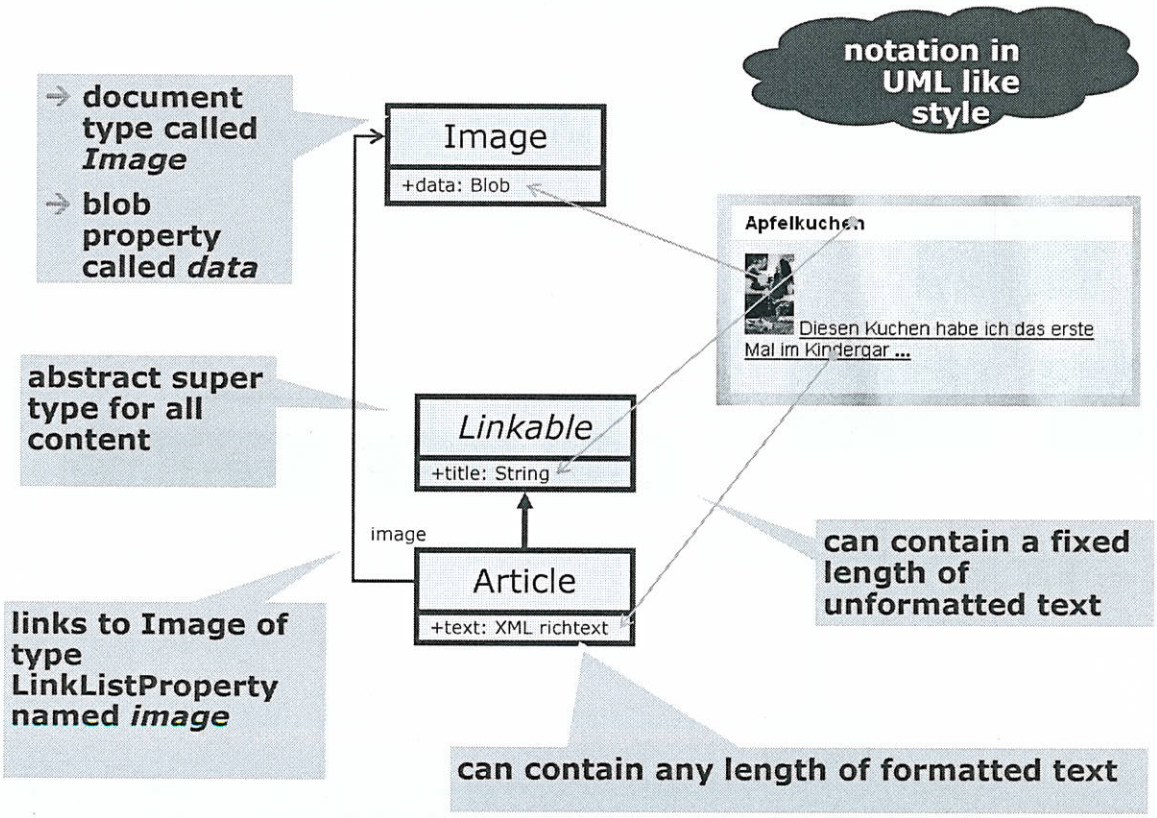
→ The Core.Dining Content Type Model

→ Content Beans

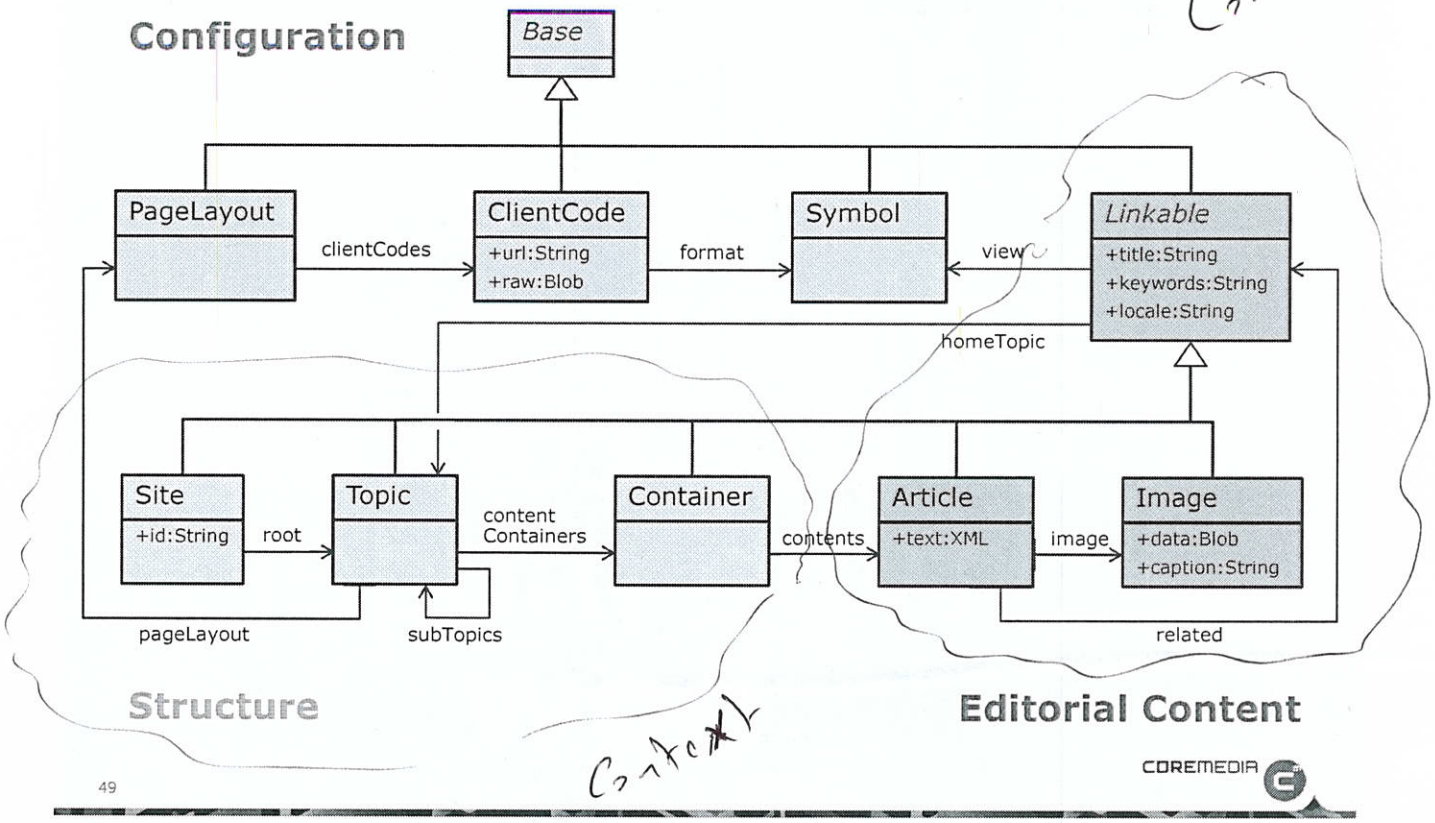
Components of a document model



All content data is stored in properties

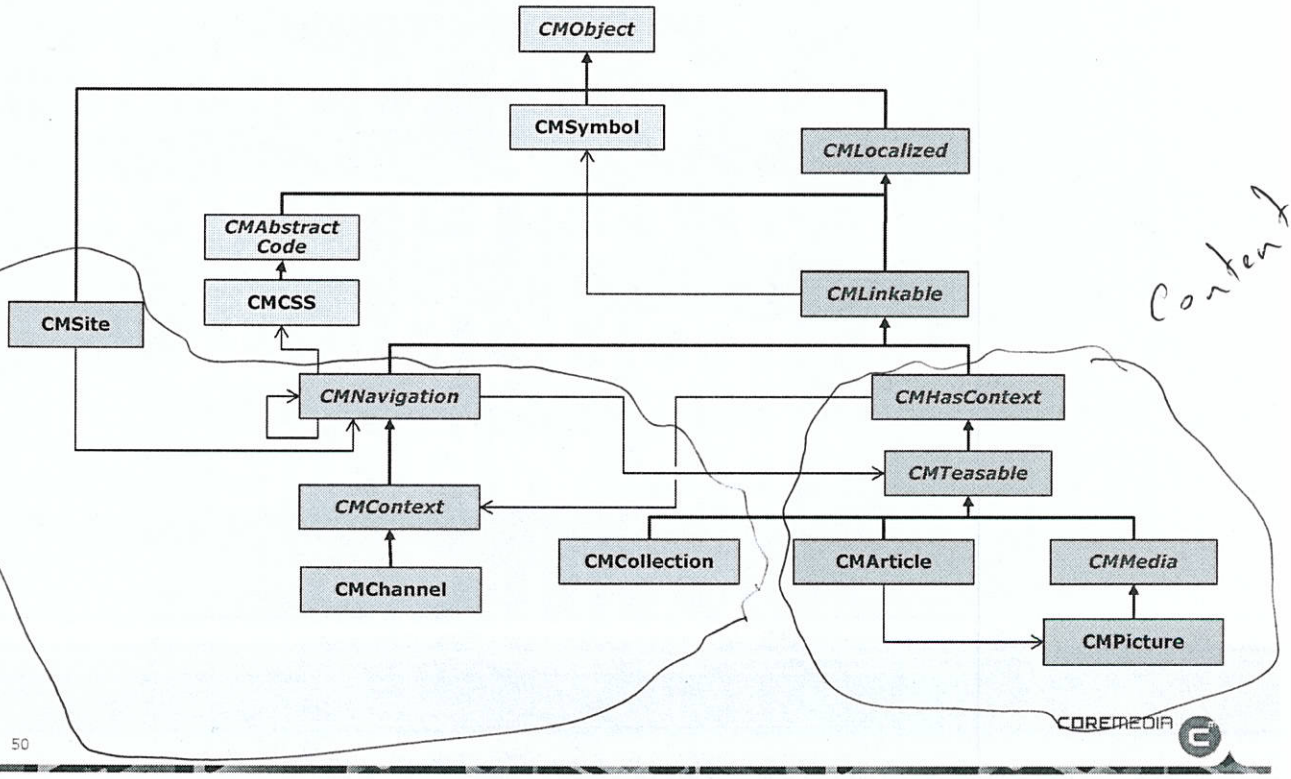


Core.Dining Content Type Model



COREMEDIA

Blueprint Content Type Model (Extract)



COREMEDIA

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

Bean Generator

→ Content Beans can be generated using a tool called BeanGenerator

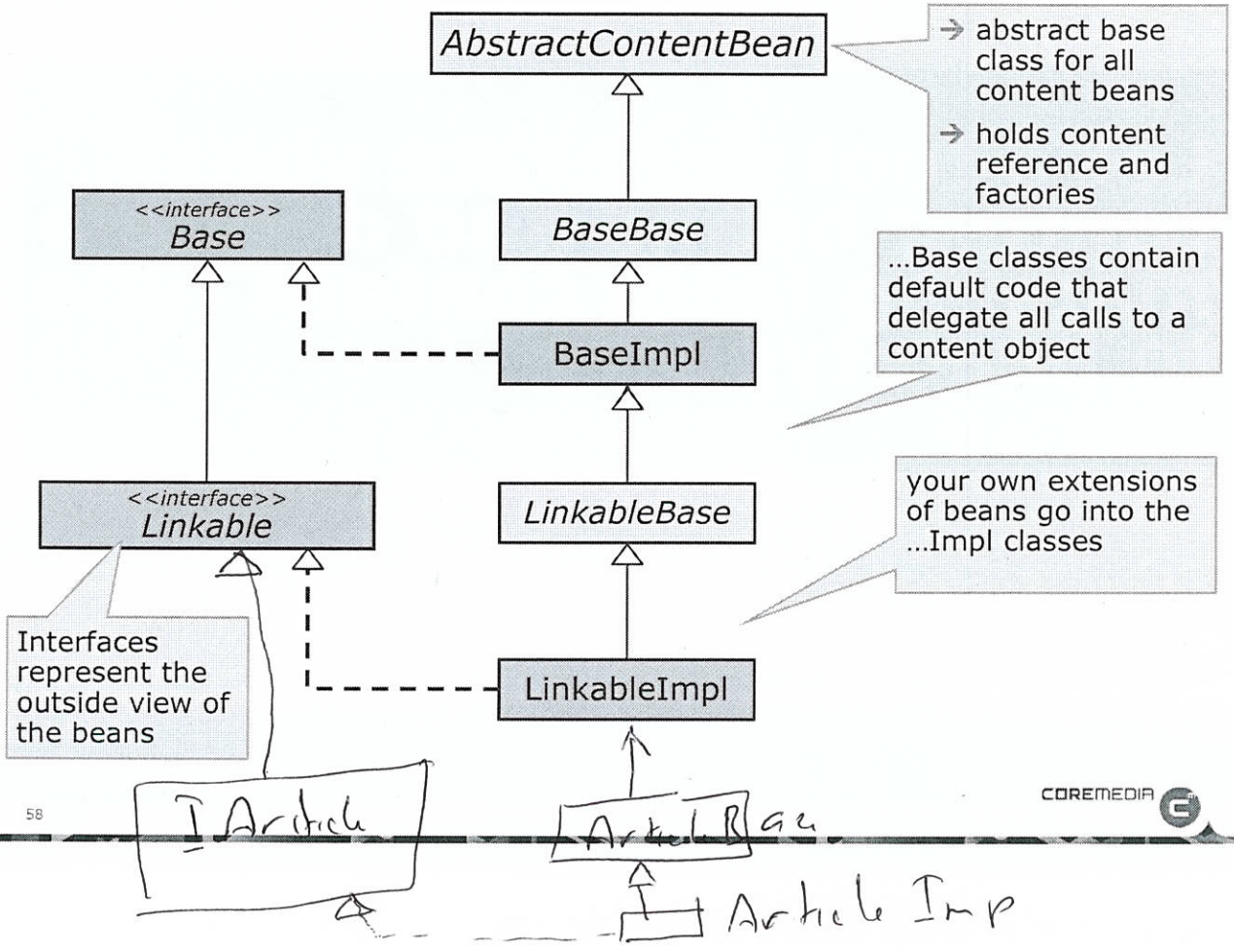
→ The Bean Generator is available as Maven artifact:

```
<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
```


Content beans: Sample type hierarchy



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

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:

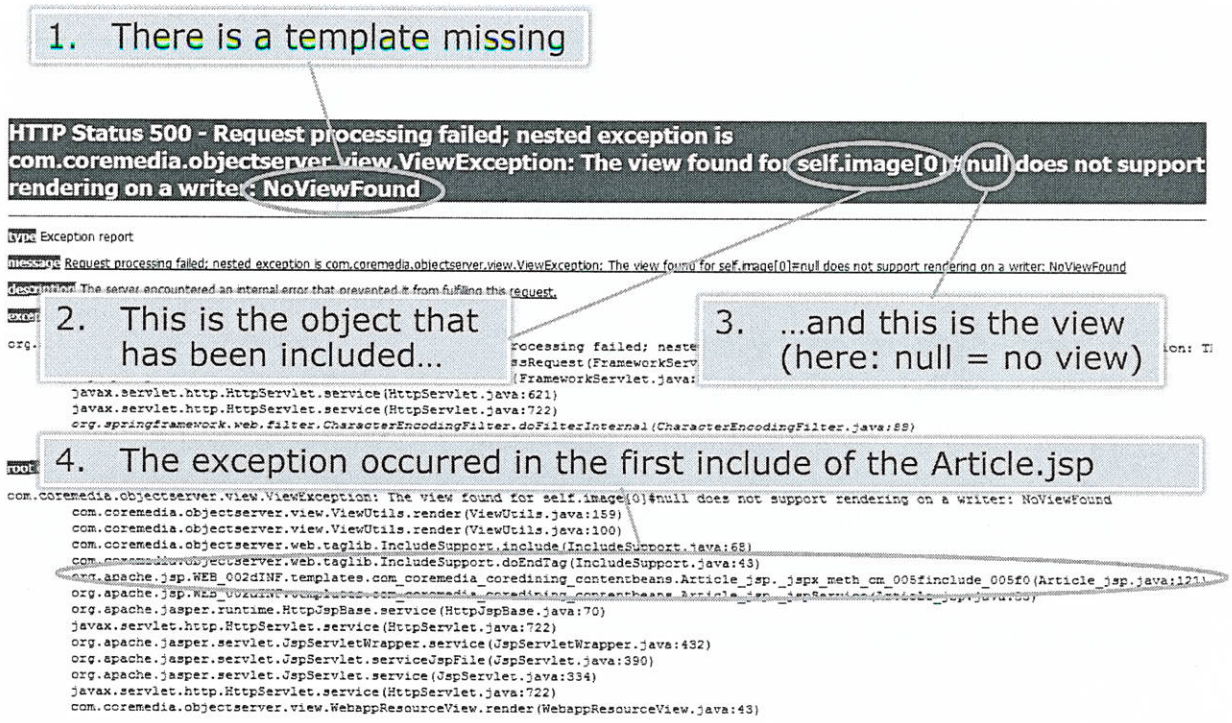
→ **self**: the object to be rendered

→ **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

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: 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.

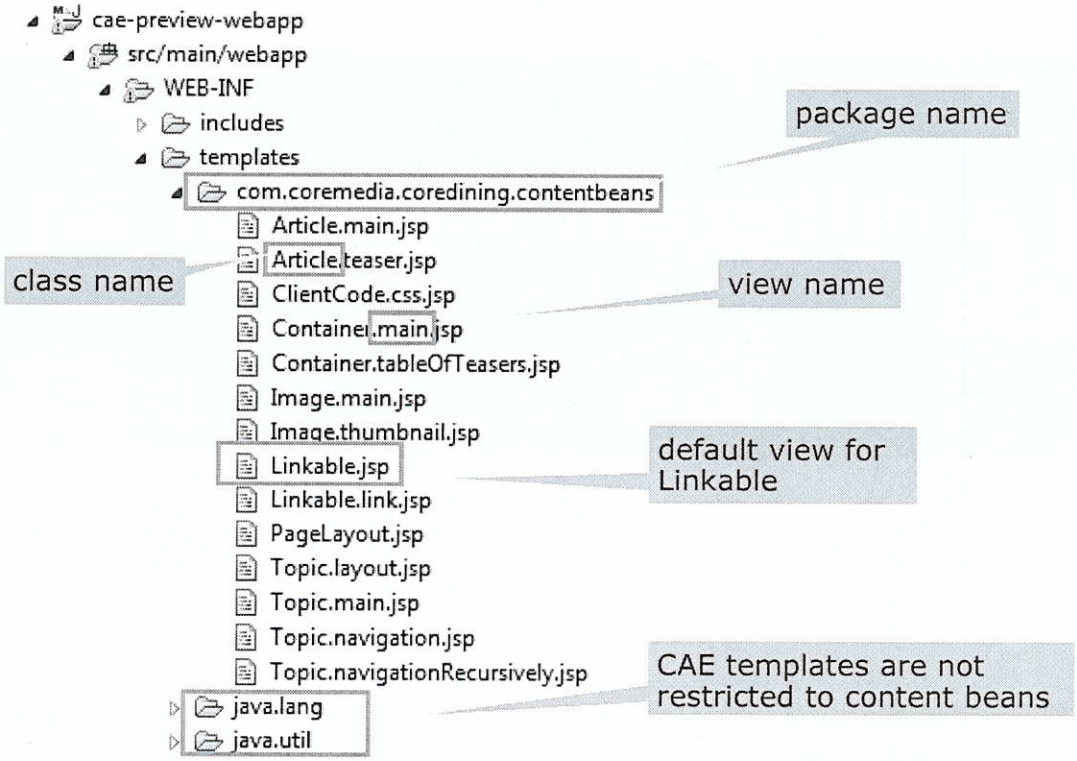
here: the output will be: /coredining/css/structure.css

CoreMedia View Dispatcher

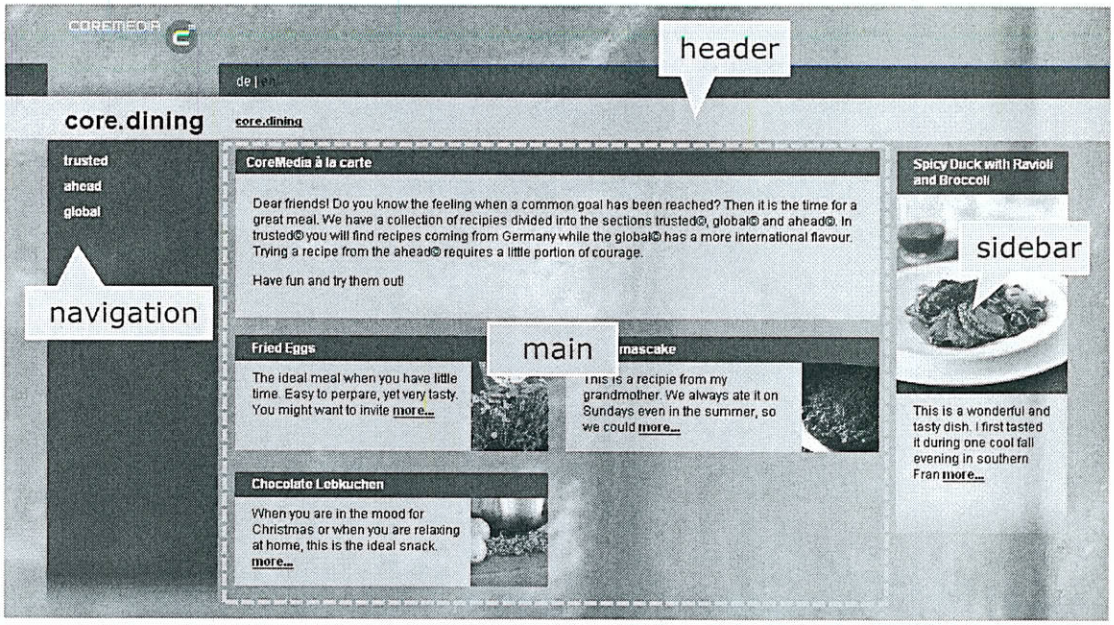
- JSP templates reside in a directory named after the content bean package
- template names are composed of
  - the content bean class name
  - optionally a view name separated by a "."
  - "JSP" as suffix
- in case there is no template named after the content bean the view dispatcher tries to find a 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



Core.Dining Page Structure



→ The Core.Dining website has a common frame.

→ Only the main area is different for articles and topics

COREMEDIA

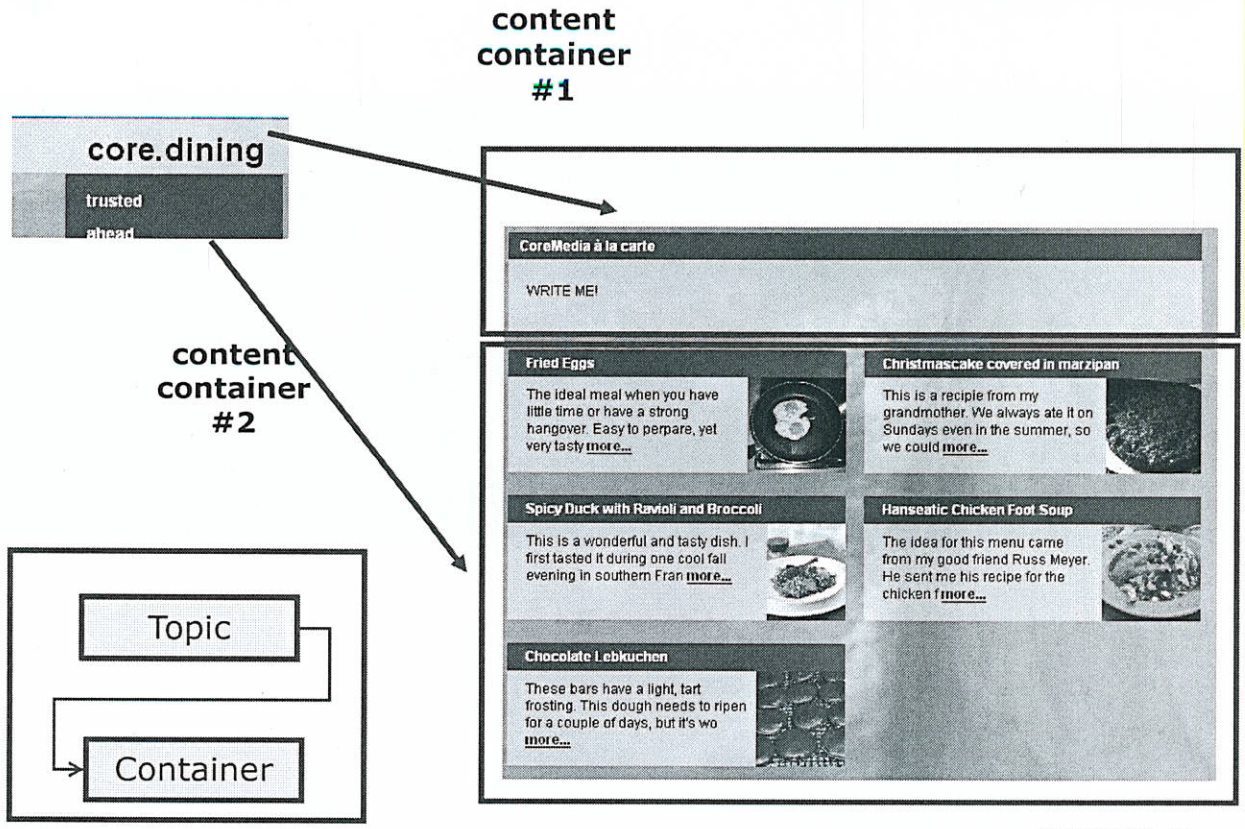
Exercises 11 + 12

COREMEDIA

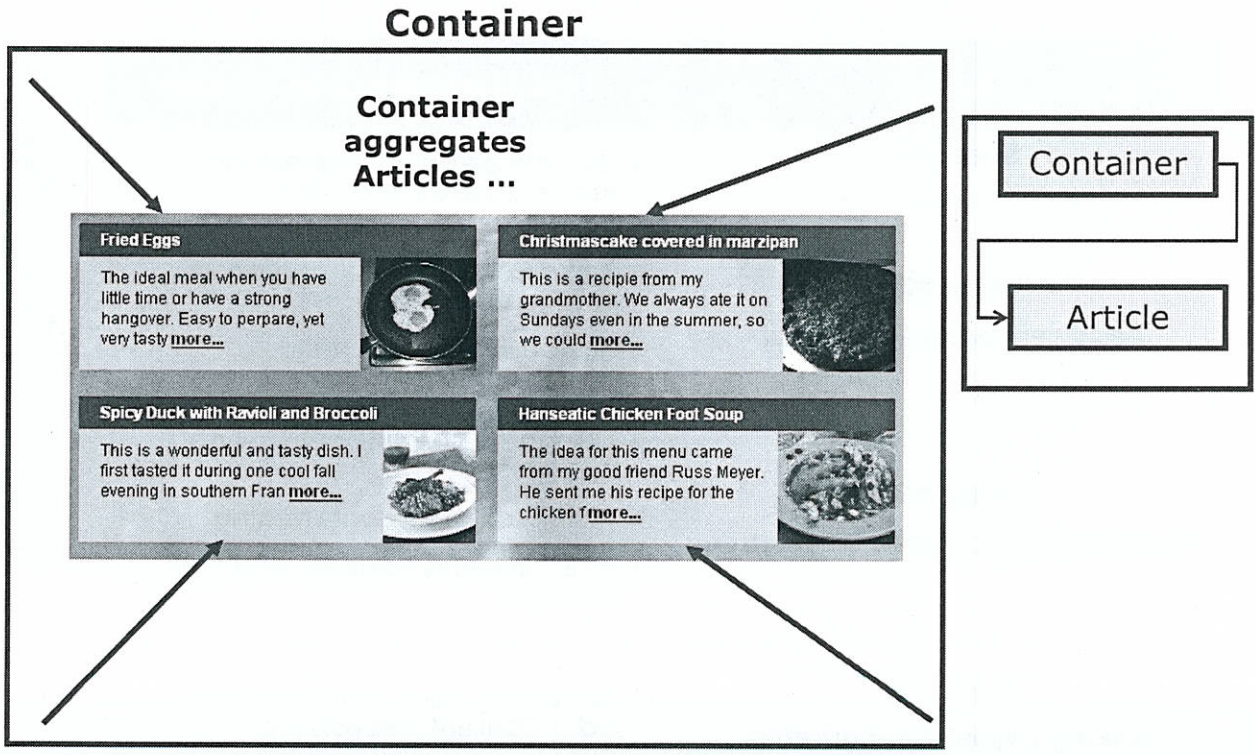
Use a common frame for all linkables and make it look nice...

COREMEDIA

Topic.main.jsp



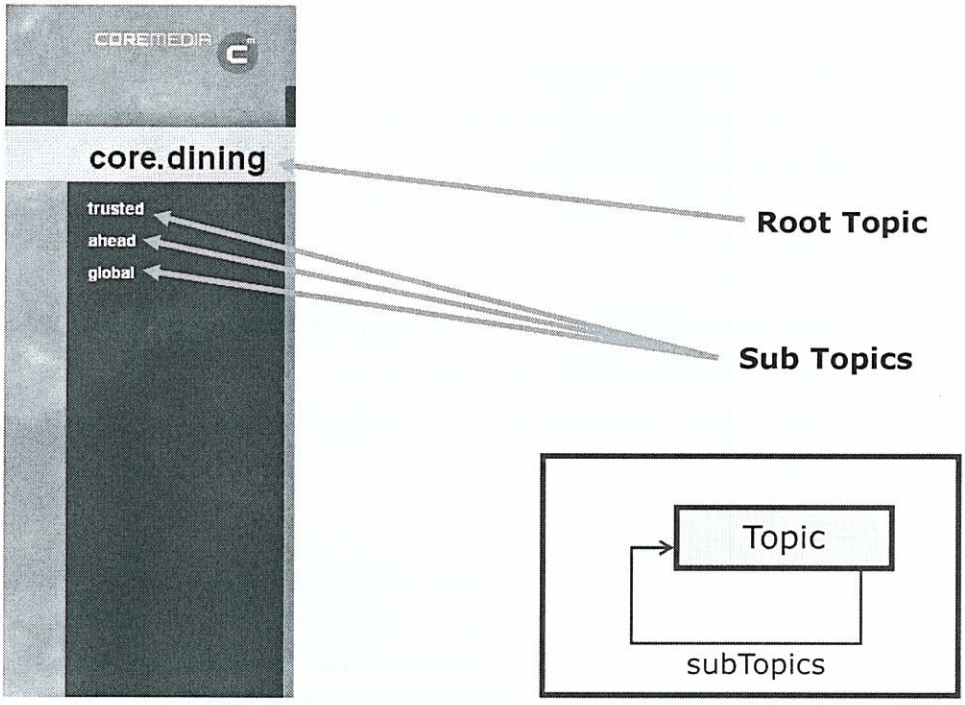
Container.main.jsp



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



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

... 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"

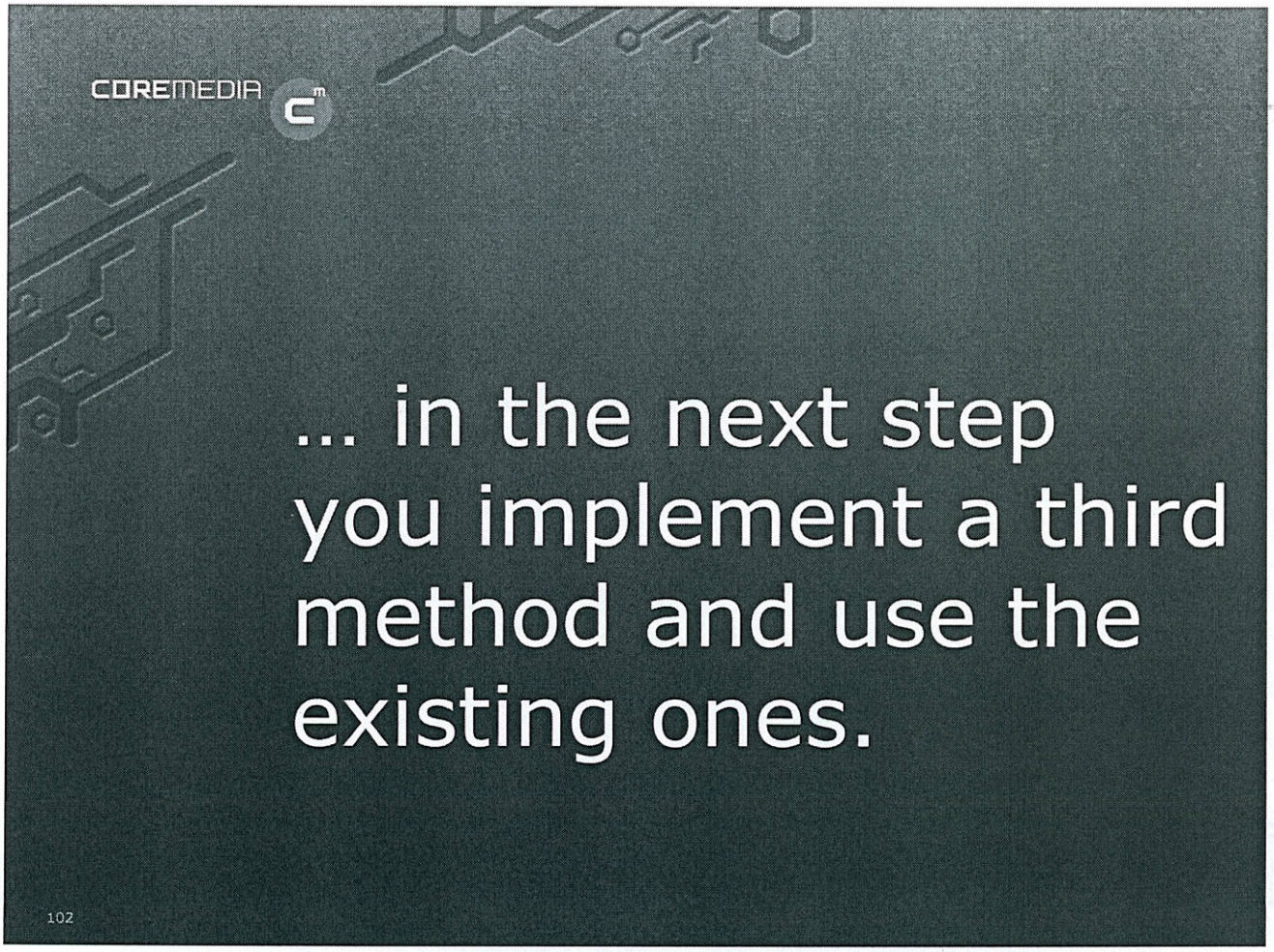
if there are no such links, this topic does not have a parent

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*

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
}
```

COREMEDIA

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

COREMEDIA

... like your variable for the current Topic!

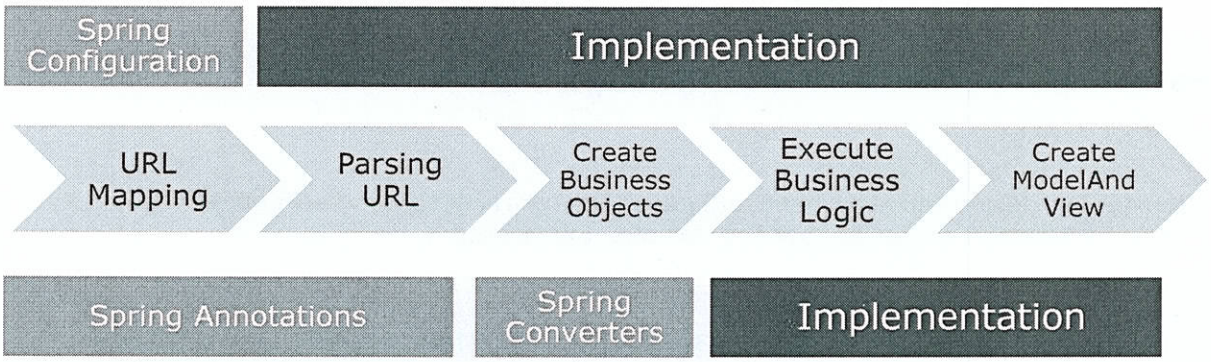
Controllers vs. Handlers

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

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

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

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

Content Handler
Registration of Spring Converters

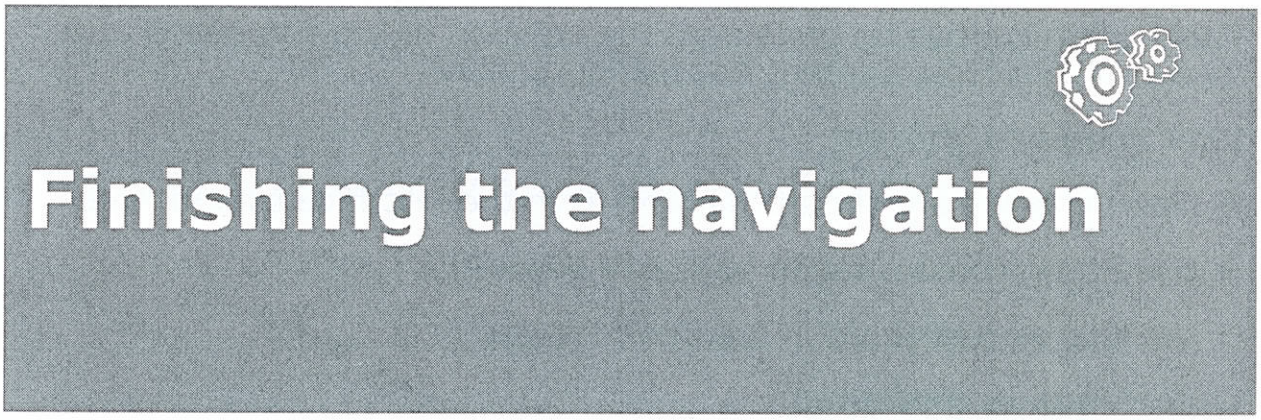
```
<import resource="classpath:/com/coremedia/cae/handler-services.xml"/>

...

<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



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)

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

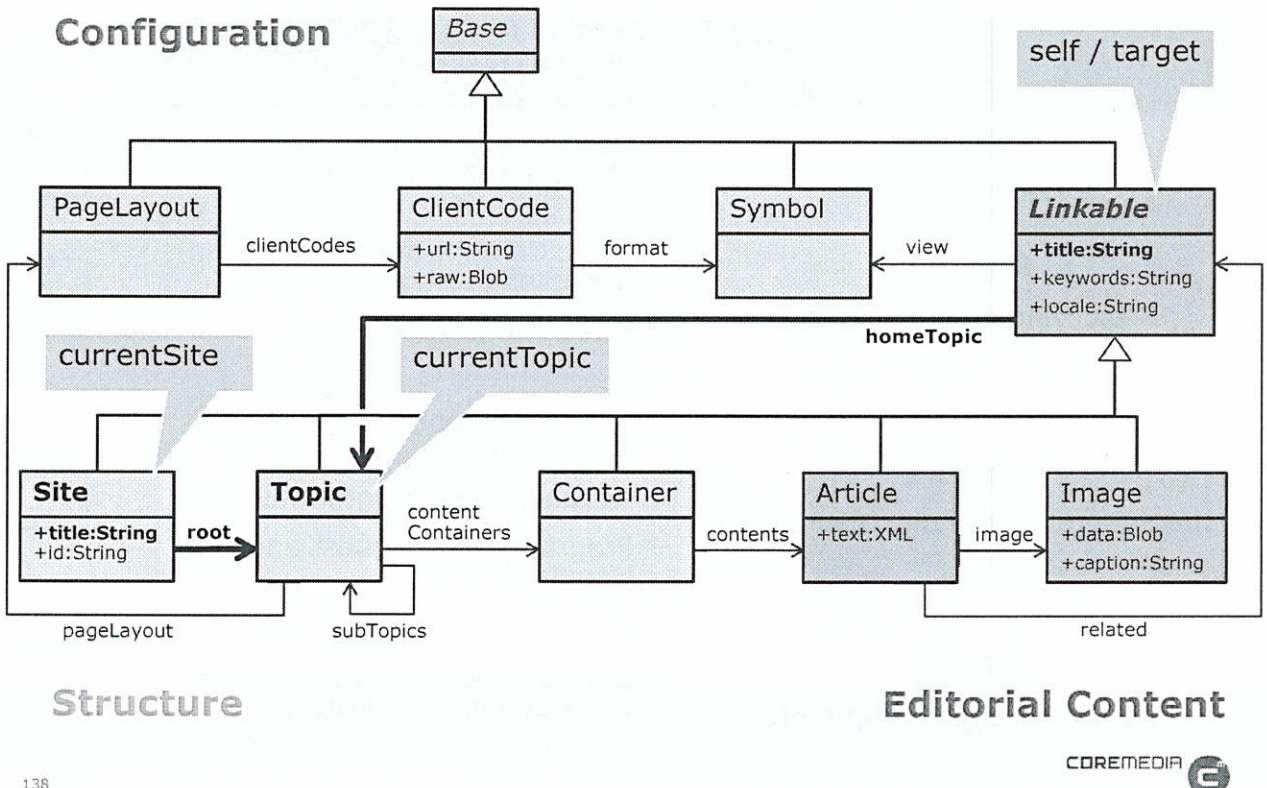
```
@LinkPostProcessor
public Object prependBaseUri(UriComponents originalUri, HttpServletRequest request) {

  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

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



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

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

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

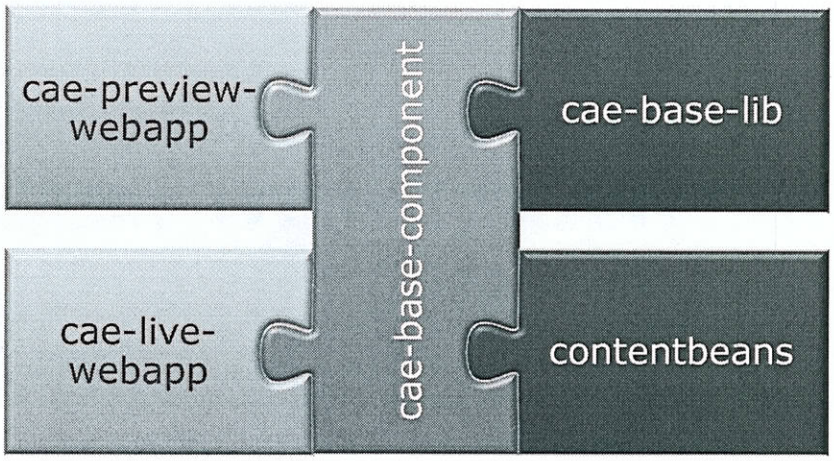
→ Have Spring configuration files in META-INF/coremedia very often these files import spring configuration from the library

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

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

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

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

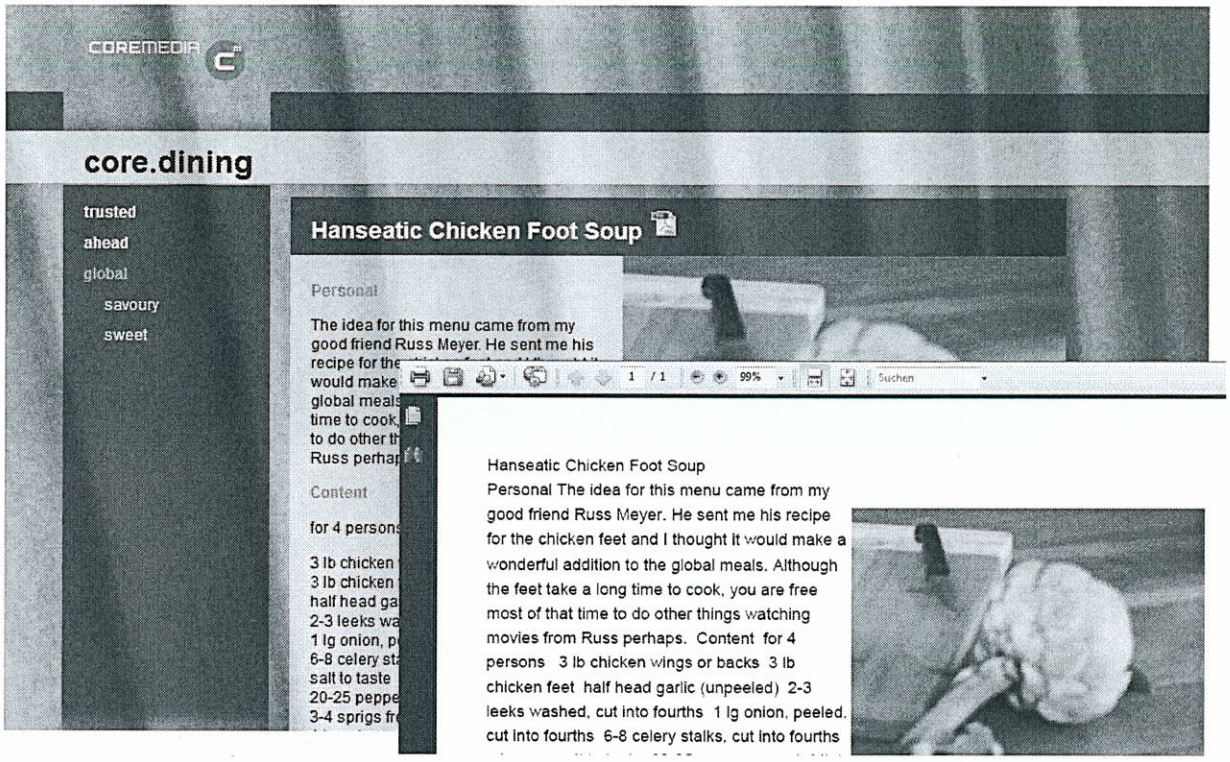
parameters always include the bean to render and the chosen view

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

Article download as PDF



CoreMedia

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

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"

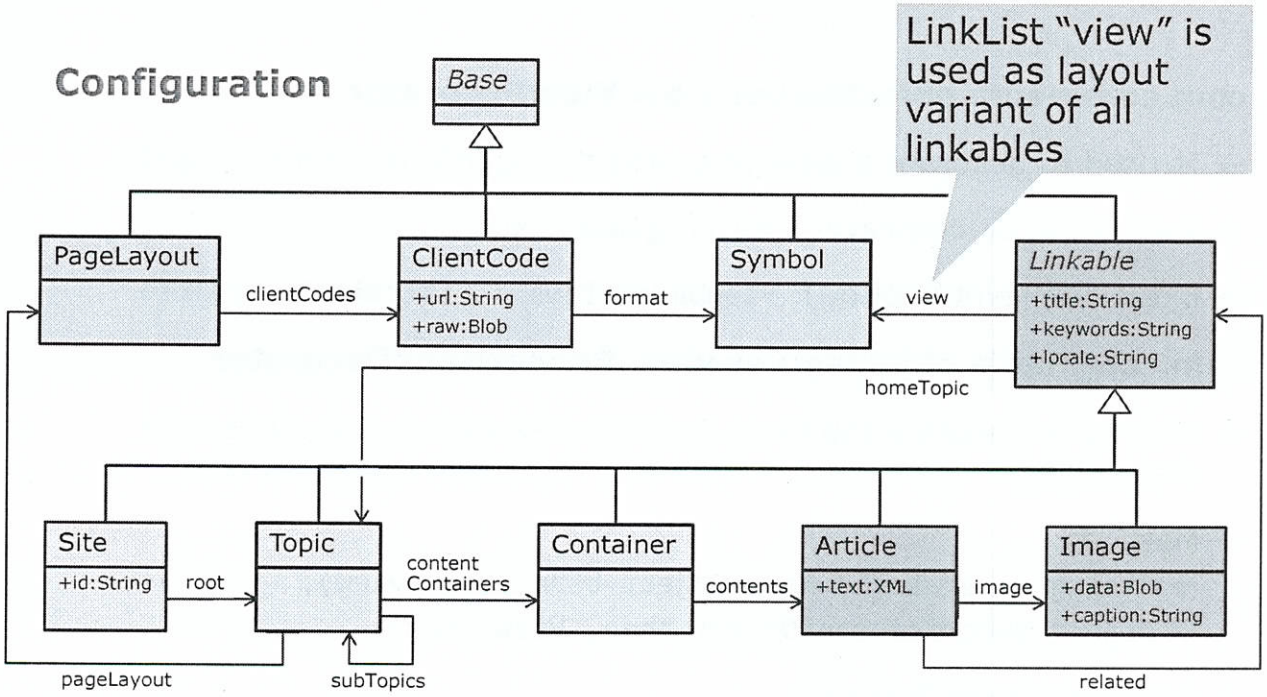
In Blueprint:

→ 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



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?

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

CoreMedia

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

CoreMedia

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

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);
```

COREMEDIA

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

Exercise 23

COREMEDIA

Add caching to your navigation

COREMEDIA

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 "*dynamic*"
Pseudo Code

```
class ArticleImpl$$ extends ArticleImpl {

   // no attribute

   // no load method

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

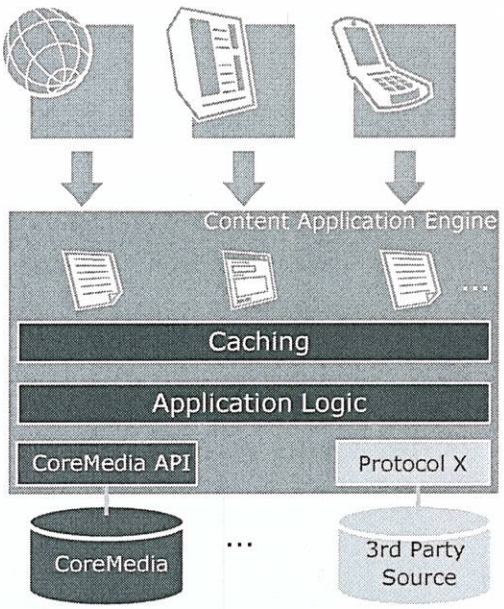
- both data views are not coupled at all.
- assures, that the return value of getRootTopic() is also a Data View.
- **should be used for dynamic calculated properties! (e.g. personalization)**

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.

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**



CoreMedia

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

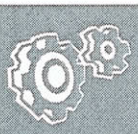
Registering the External Article

```
<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

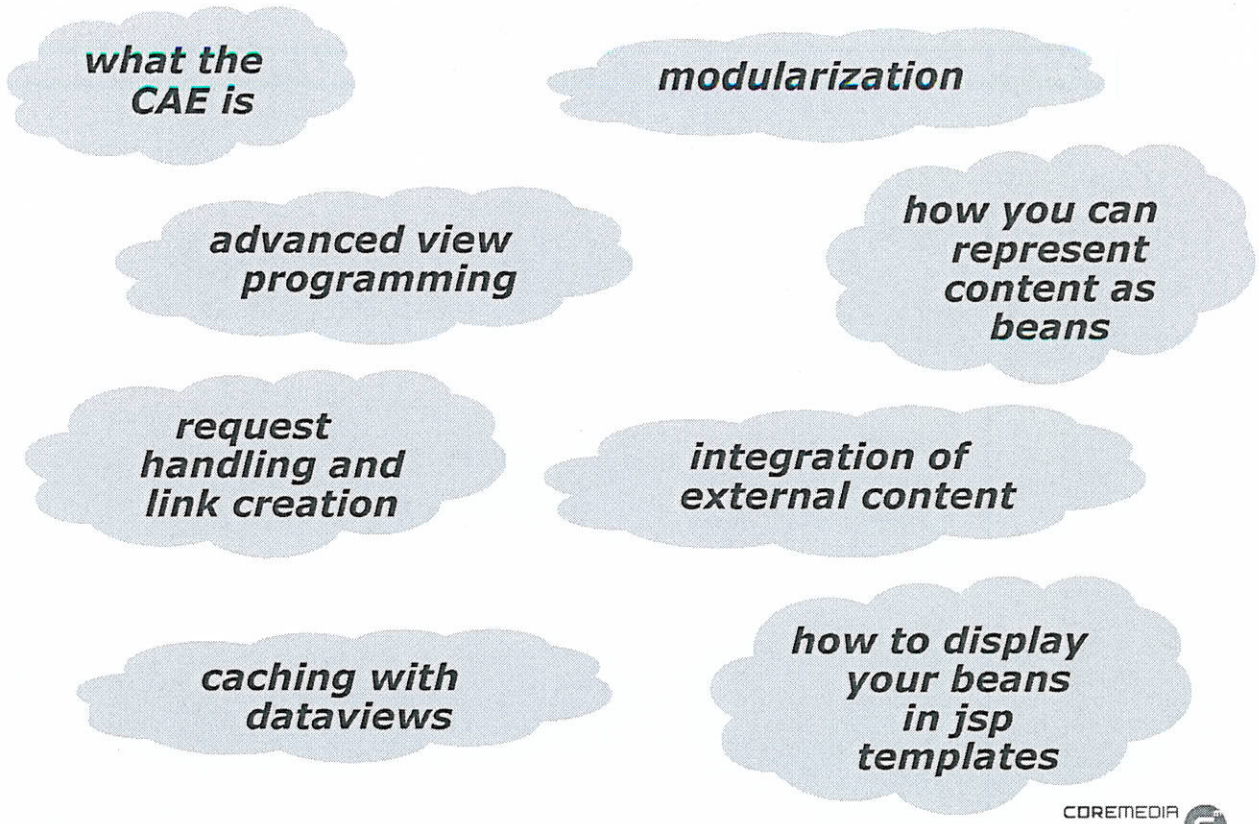
COREMEDIA



Include external content

COREMEDIA

What have you learned?



CoreMedia Training Program