



Web Application Development Training

Workbook

Exercise 1

Task: Install your system

1. Open your windows explorer and go to the directory C:\training\
This folder contains the maven workspace of the Core.Dining CMS project, a folder for exercise material, some start scripts and a GettingStarted.html.
2. Open the file GettingStarted.html in your browser. This file will give your information how to build and start the Core.Dining server components.
3. Follow the instructions of your trainer to install and start your server components.

After a few minutes your will have a running Content Management Server and a running Preview CAE web application.

Verify that everything is up and running using the Service URLs shown in the GettingStarted.html

Checkpoint: Your browser should display the application

Service URLs

4. Service URLs

Content Management Environment

Content Management Server

IOR URL: <http://localhost:41080/coremedia/ior>

Workflow Server *

IOR URL: <http://localhost:43080/coremedia/ior>

Preview CAE

Overview Page: <http://localhost:40081/coredining/servlet/content/92>

Article Page: <http://localhost:40081/coredining/servlet/content/22>

Studio

<http://localhost:40080/>

SolR Search Engine

<http://localhost:44080/solr>

Content Delivery Environment

Master Live Server *

<http://localhost:42080/coremedia/ior>

Replication Live Server *

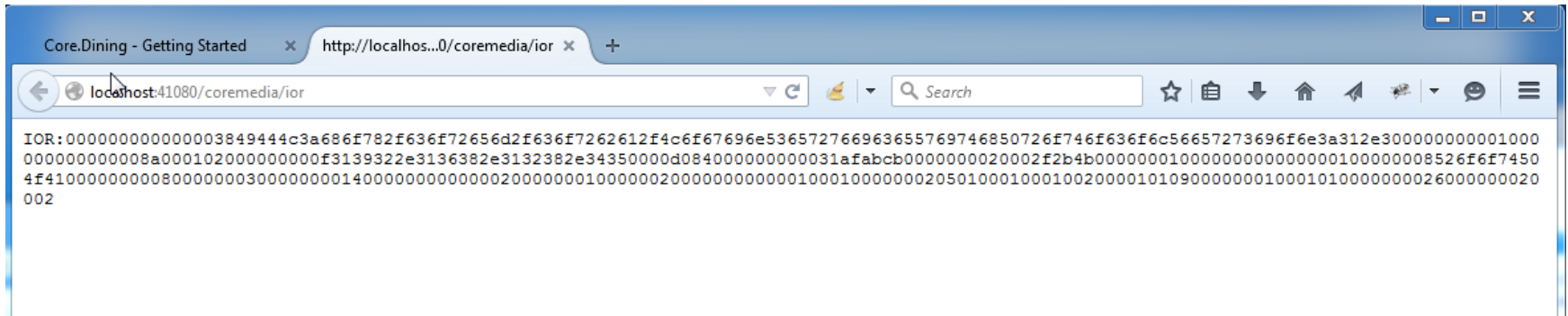
<http://localhost:48080/coremedia/ior>

Live CAE *

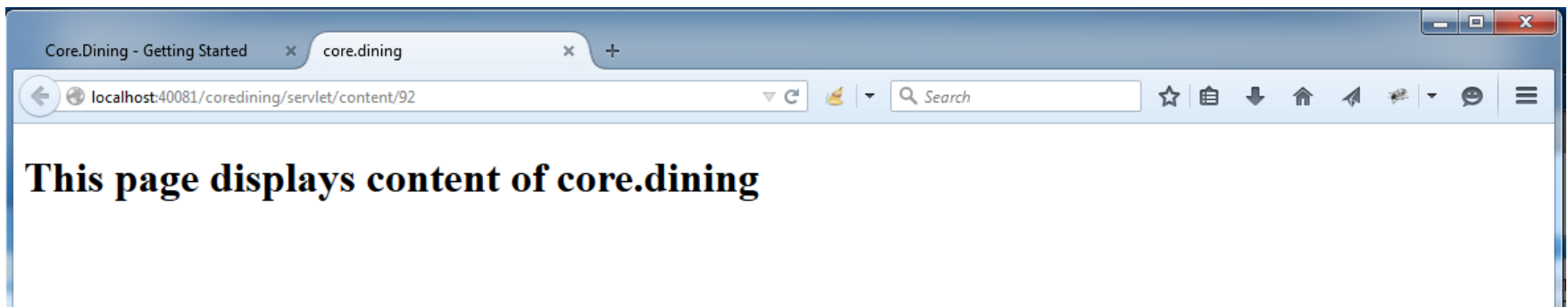
<http://localhost:49080/coredining/servlet/content/92>

Checkpoint:

→ Content Management Server is running:



→ Preview CAE is running



Exercise 2

Task: Make a first change to the application

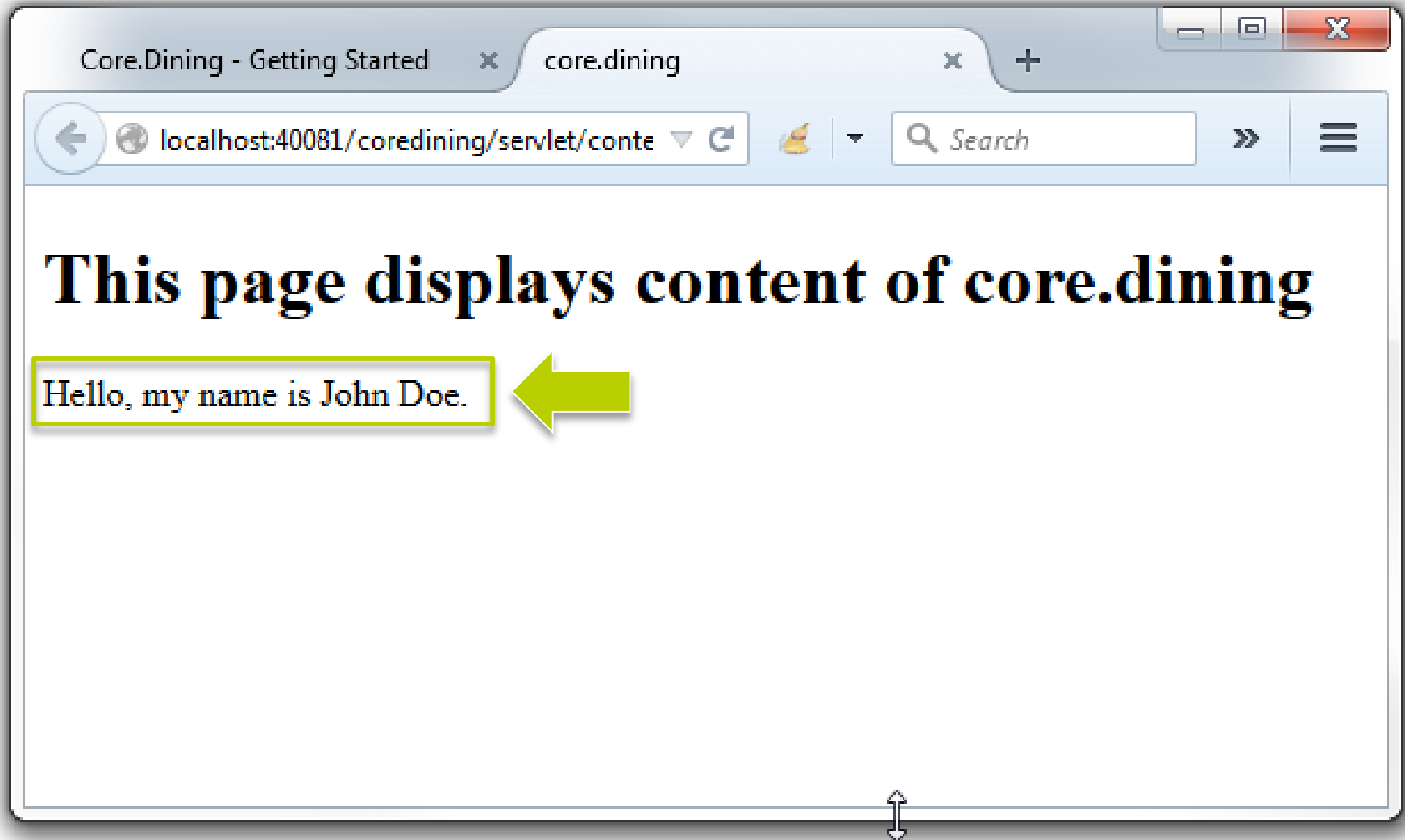
1. Start the Eclipse IDE
2. If not already done, import the module 'cae-preview-webapp' as Existing Maven Project.
3. Locate the application templates:
src/main/webapp/WEB-INF/templates
4. In folder "com.coremedia.coredining.contentbeans" modify the Linkable.jsp to display your name in the head and title of the page
5. Reload your browser. Your changes will be visible immediately

Note: If you only change JSP files, you don't need to restart your web application. The Maven Jetty Plugin is able to apply these kind of changes.

Tipp: Use the directory src/main/webapp as source folder (context menu: Build Path > Use as Source Folder)

Checkpoint: Refresh your browser to see the changes

Checkpoint: After changing the JSP template



Exercise 3

Task: Add a new JavaBean property and display it in a JSP template

1. Open the Java class file
com.coremedia.coredining.contentbeans.LinkableImpl
2. Add a method *getText()* which returns a String literal.
3. Display this property in the corresponding JSP (Linkable.jsp)
4. If you reload your browser, you will see an Error stating that the *getText()* method is missing.
5. Stop your web application and start it again. Now your changes will become visible.

Note: A restart of the web application is required in the following situations:

- Changes on Java classes
- Changes on Spring configuration files
- Adding or removing JSP templates

Checkpoint: Refresh your browser to see the your text

Checkpoint: getText() is missing

HTTP Status 500 - Request processing failed; nested exception is javax.el.PropertyNotFoundException: Property 'text' not found on type com.coremedia.coredining.contentbeans.LinkableImpl

type Exception report

message Request processing failed; nested exception is javax.el.PropertyNotFoundException: Property 'text' not found on type com.coremedia.coredining.contentbeans.LinkableImpl

description The server encountered an internal error that prevented it from fulfilling this request.

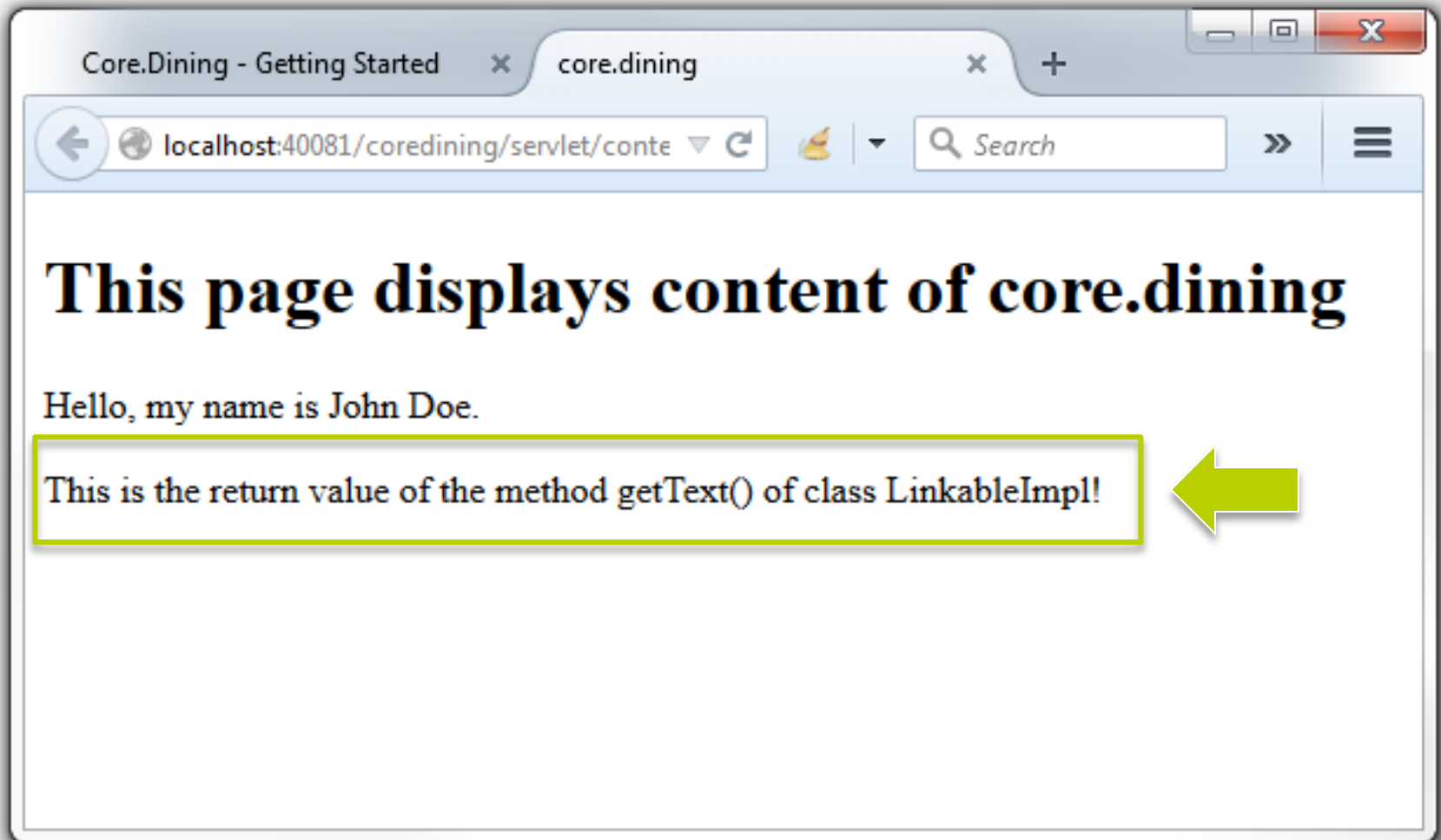
exception

```
org.springframework.web.util.NestedServletException: Request processing failed; nested exception is javax.el.PropertyNotFoundException: Property 'text' not found on type com.coremedia.coredining.contentbeans.LinkableImpl
    org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:894)
    org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:778)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:621)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:722)
    org.springframework.web.filter.CharacterEncodingFilter.doFilterInternal(CharacterEncodingFilter.java:88)
    org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:76)
```

root cause

```
javax.el.PropertyNotFoundException: Property 'text' not found on type com.coremedia.coredining.contentbeans.LinkableImpl
    javax.el.BeanELResolver$BeanProperties.get(BeanELResolver.java:237)
    javax.el.BeanELResolver$BeanProperties.access$400(BeanELResolver.java:214)
    javax.el.BeanELResolver.property(BeanELResolver.java:325)
    javax.el.BeanELResolver.getValue(BeanELResolver.java:85)
    javax.el.CompositeELResolver.getValue(CompositeELResolver.java:67)
    org.apache.el.parser.AstValue.getValue(AstValue.java:183)
    org.apache.el.ValueExpressionImpl.getValue(ValueExpressionImpl.java:185)
    org.apache.jasper.runtime.PageContextImpl.evaluate(PageContextImpl.java:896)
    org.apache.jsp.WEB_002dINF.templates.com_coremedia_coredining_contentbeans.Linkable_jsp._jspService(Linkable_jsp.java:81)
    org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:79)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:722)
```


Checkpoint: Your text is displayed



Exercise 4

Task: Generate a default set of content beans

1. Delete all existing Java classes and interfaces in the package `'com.coremedia.coredining.contentbeans'`.
2. Let the BeanGenerator create new content beans for the document types of Core.Dining. You will find an Ant script build.xml in your workspace with a target `"generate-content-beans"`.

Note: Existing files will not be overwritten! Delete all files you want to regenerate

3. Refresh your Eclipse project
4. In the "contentbeans" package you will find the generated beans
5. Have a look at the generated Base classes. How are the access methods for the different property types implemented?



1. What is the general structure of the generated files?
2. What could be the idea of three files per content type?
3. Where would your own extensions go to?

Checkpoint: You should see the generated beans in package "contentbeans"

Checkpoint: The generated beans

The screenshot shows the Eclipse IDE with the following components:

- Package Explorer:** Displays the project structure. The package `com.coremedia.coredining.contentbeans` is expanded, showing a list of classes. `LinkableBase.java` is highlighted with a yellow circle.
- Main Editor:** Displays the source code of `LinkableBase.java`. The code is as follows:

```
1 package com.coremedia.coredining.contentbeans;
2
3 import com.coremedia.cap.content.Content;
4
5 /**
6  * Generated base class for beans of document type "linkable".
7  */
8 public abstract class LinkableBase extends BaseImpl {
9
10     /**
11      * DEVELOPER NOTE
12      * Change {@link com.coremedia.coredining.contentbeans.LinkableImpl} instead of this class.
13      */
14
15     /**
16      * Returns the value of the document property "title"
17      * @return the value of the document property "title"
18      */
19     public String getTitle() {
20         return getContent().getString("title");
21     }
22
23     /**
24      * Returns the value of the document property "keywords"
25      * @return the value of the document property "keywords"
26      */
27     public String getKeywords() {
28         return getContent().getString("keywords");
29     }
30
31     /**
32      * Returns the value of the document property "homeTopic"
33      * @return the value of the document property "homeTopic"
34      */
35     public List<? extends Topic> getHomeTopic() {
36         List<?<Content>*/ contents = getContent().getLinks("homeTopic");
37         List<? extends Topic> contentBeans = (List<? extends Topic>).createBeansFor(contents);
38         return contentBeans;
39     }
40 }
```
- Callout:** A yellow circle highlights the `getTitle()` method in the code. A callout box points to it with the text "generated property access method".

Exercise 5

Task: Configure a document type to bean mapping

1. Locate the Spring configuration file for the mapping from document type to bean class:
framework/spring/coredining-contentbeans.xml
2. For each document type provide a mapping entry to one of the generated beans.
3. Use the "...Impl" bean classes as mapping targets
4. Delete the default mapping for document type "Linkable"
5. Restart your web application (because we made changes on Java classes and spring configuration!)



- Reconsider the document model: Do you need a mapping for every document type?

Checkpoint: When you refresh your browser you should see an error (<Unable to find value for "text" in object of class...>).

Checkpoint:

➔ Overview Pages will fail! (Do you know why?)

```
HTTP Status 500 - Request processing failed; nested exception is
javax.el.PropertyNotFoundException: Property 'text' not found on type
com.coremedia.coredining.contentbeans.TopicImpl

Type Exception report
Message Request processing failed; nested exception is javax.el.PropertyNotFoundException: Property 'text' not found on type
com.coremedia.coredining.contentbeans.TopicImpl
Description The server encountered an internal error that prevented it from fulfilling this request.
Exception
org.springframework.web.util.NestedServletException: Request processing failed; nested exception is javax.el.PropertyNotFoundException:
Property 'text' not found on type com.coremedia.coredining.contentbeans.TopicImpl
    org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:894)
    org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:778)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:621)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:722)
    org.springframework.web.filter.CharacterEncodingFilter.doFilterInternal(CharacterEncodingFilter.java:88)
    org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:76)
Root cause
javax.el.PropertyNotFoundException: Property 'text' not found on type com.coremedia.coredining.contentbeans.TopicImpl
    javax.el.BeanELResolver$BeanProperties.get(BeanELResolver.java:237)
    javax.el.BeanELResolver$BeanProperties.access$400(BeanELResolver.java:214)
```

➔ Article Pages will show article text! (Do you know why?)

This page displays content of Spicy Duck with Ravioli and Broccoli

Hello, my name is John Doe.

Personal

This is a wonderful and tasty dish. I first tasted it during one cool fall evening in southern France. Whenever I cook it, the memories of the beautiful mountainside and the very special scent of the landscape come to my mind. By the way, the dish is perfect for a dinner party (if you prepare the oven a little in advance).

Content

makes 4 servings

2 cups red wine

2 5-pound ducks

1 teaspoons salt

1/2 teaspoon black pepper

1/2 teaspoon ginger

1 tablespoon olive oil

Exercise 6

Task: Render an Article

1. Open the Article Page (You will find the link to article pages in the GettingStarted.html)
2. You will see no error as an Article actually has a text property
3. Adapt Linkable.jsp to display title and text in a reasonable way.
Use the **<cm:include>** tag to include the text property, because this is the appropriate way to display XML properties.
4. Add some code to display the image associated to the Article
5. You will need the **<cm:link>** tag from the CoreMedia tag library
6. Explore the Eclipse JSP editor and how it can help you
7. Reload your browser to see your changes.

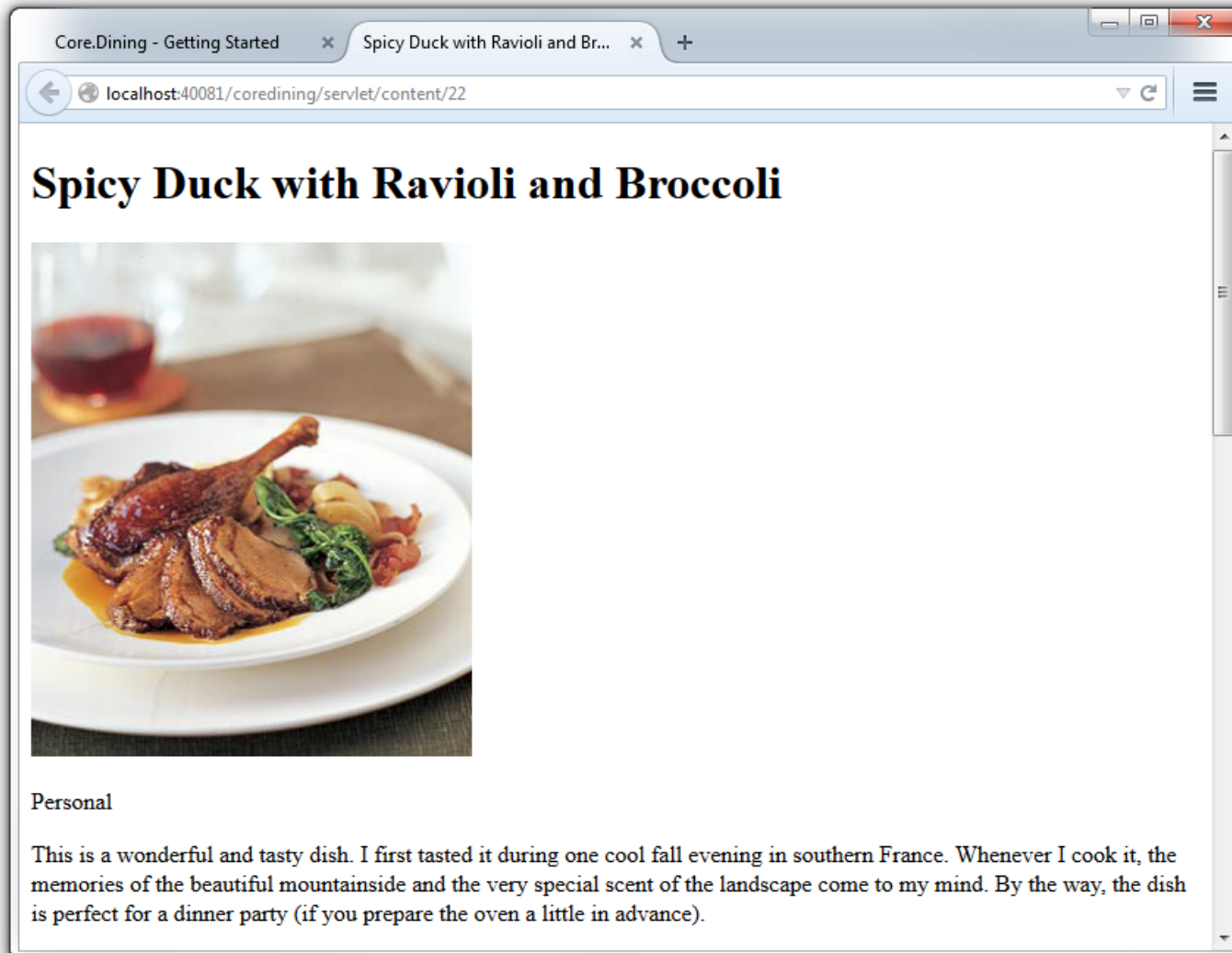


- Why is the template "Linkable.jsp" used even though our content bean is of type "Article"? Is it a good idea to change "Linkable.jsp" in this way in the first place?

Checkpoint: When you refresh your browser you should see a complete article including an image.

Checkpoint:

Article page with image



Exercise 7

Task: Create new view templates

1. Rename "Linkable.jsp" to "Article.jsp". Articles will now be displayed using this JSP
2. Create a new template for the image of the article. What must be the name of the template?
3. Move the code to display the image from "Article.jsp" to the image template and adapt the expression in the `<cm:link>` tag accordingly.
4. Use **`<cm:include>`** to call the new image template from Article.jsp
5. Restart your web application



1. What happens if there is no image defined for that Article? What if there is no title? No text?
2. How could you check that?

Checkpoint: When you refresh your browser you should still see your application without change. Additionally, you should now be able to preview images from the editor.

Checkpoint: If you see something like that, maybe you have forgotten the curly braces: \${...}

HTTP Status 500 - Request processing failed; nested exception is com.coremedia.objectserver.view.ViewException: The view found for self.image[0]#null does not support rendering on a writer: NoViewFound

type Exception report

message Request processing failed; nested exception is com.coremedia.objectserver.view.ViewException: The view found for self.image[0]#null does not support rendering on a writer: NoViewFound

description The server encountered an internal error that prevented it from fulfilling this request.

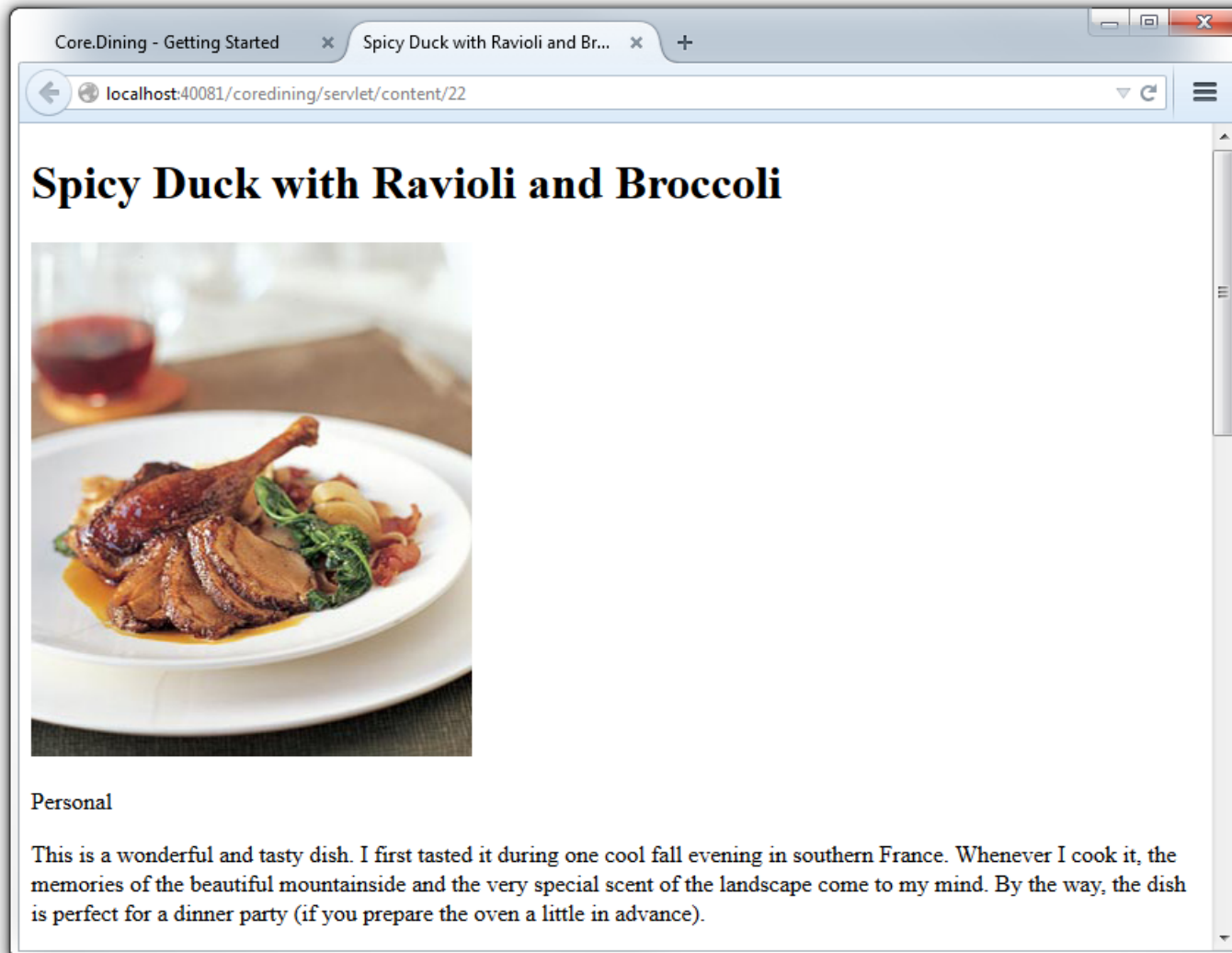
exception

```
org.springframework.web.util.NestedServletException: Request processing failed; nested exception is com.coremedia.objectserver.view.ViewException: The view found for self.image[0]#null does not support rendering on a writer: NoViewFound
    org.springframework.web.servlet.FrameworkServlet.processRequest (FrameworkServlet.java:894)
    org.springframework.web.servlet.FrameworkServlet.doGet (FrameworkServlet.java:778)
    javax.servlet.http.HttpServlet.service (HttpServlet.java:621)
    javax.servlet.http.HttpServlet.service (HttpServlet.java:722)
    org.springframework.web.filter.CharacterEncodingFilter.doFilterInternal (CharacterEncodingFilter.java:88)
    org.springframework.web.filter.OncePerRequestFilter.doFilter (OncePerRequestFilter.java:76)
```

root cause

```
com.coremedia.objectserver.view.ViewException: The view found for self.image[0]#null does not support rendering on a writer: NoViewFound
    com.coremedia.objectserver.view.ViewUtils.render (ViewUtils.java:159)
    com.coremedia.objectserver.view.ViewUtils.render (ViewUtils.java:100)
```

Checkpoint: You should not see any changes in the looks of your page.



Exercise 8

Task: Add checks to your templates

Not all articles have images or a title or even text. In order to make the templates more robust, we want to add some checks:

1. Modify "Article.jsp" to check for existence of all properties
2. Use the **<c:if>** tag from the JSTL tag library
3. Eclipse will guide you via code completion when you type **<c:if** and press [Ctrl] + [Space]
4. Enter something similar to "\${**not empty** self.text}" or "\${**not empty** self.image}" into the test attribute.
5. Do the same for the Image.jsp.
6. Use **<c:out>** to encode all string properties (e.g. \${self.title}).
7. Reload your browser.



Why is it important to use the **<c:out>** tag for strings?

Checkpoint: When you refresh your browser you should not see any difference. But start the Site Manager and add some "
" to the title of an article. Check what happens with and without **<c:out>** tag.

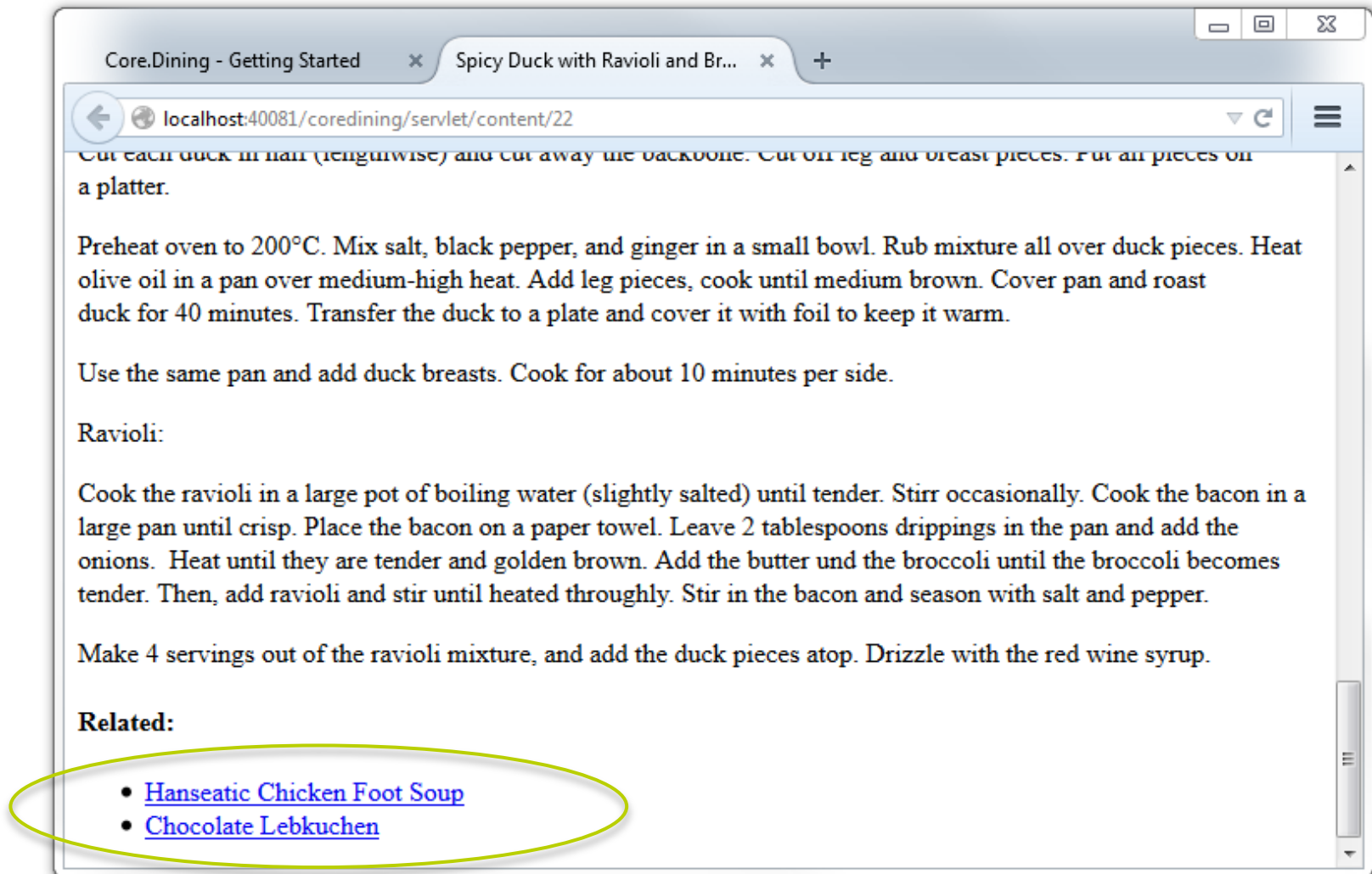
Exercise 9

Task: Add loops to your templates

1. An Article also contains a linked list of related articles. We want to render these related articles list of hyperlinks.
2. Add a “**c:forEach**” tag from the JSTL tag library to your Article.jsp to display all related content
3. Add a link from the title to the related content
4. Eclipse will guide you via code completion when you type “**<c:forEach**” and press [Ctrl] + [Space]
5. Reload your browser.

Checkpoint: When you refresh your browser you should see a list of links to the related content. The links should actually work.

Checkpoint: All related content is displayed as links at the bottom of the page (scroll down)



Exercise 10

Task: Add a template for rendering a textual link for Linkables

All textual hyperlinks to articles on our website look exactly the same: an ``-tag with the title property as link text. All the properties we need to render a link are defined in the abstract type `Linkable`. So it is a good idea to use this type for creating our new template.

1. Add a new view for a `Linkable` called "link" (`Linkable.link.jsp`)
2. Take over the code from the original article template to display an article as a link to itself
3. Refactor the original `Article.jsp` to include this template:

```
<cm:include self="${rel}" view="link"/>
```

NOTE: There is no `Article.link.jsp`, but because of the object orientation of the `ViewDispatcher`, the `Linkable.link.jsp` will be sufficient.

4. Restart your webapp.

NOTE: You can check if your new view is working by adding `"?view=link"` to your article URL.



- ➔ Should `"Linkable.link.jsp"` be a complete HTML page?
- ➔ Think ahead: Where else would you use links to `Linkables` on the final version of the `core.dining` web-site?

Checkpoint: When you refresh your browser you should still see your application without change.

Exercise 11

Task: Use a common frame for all linkables

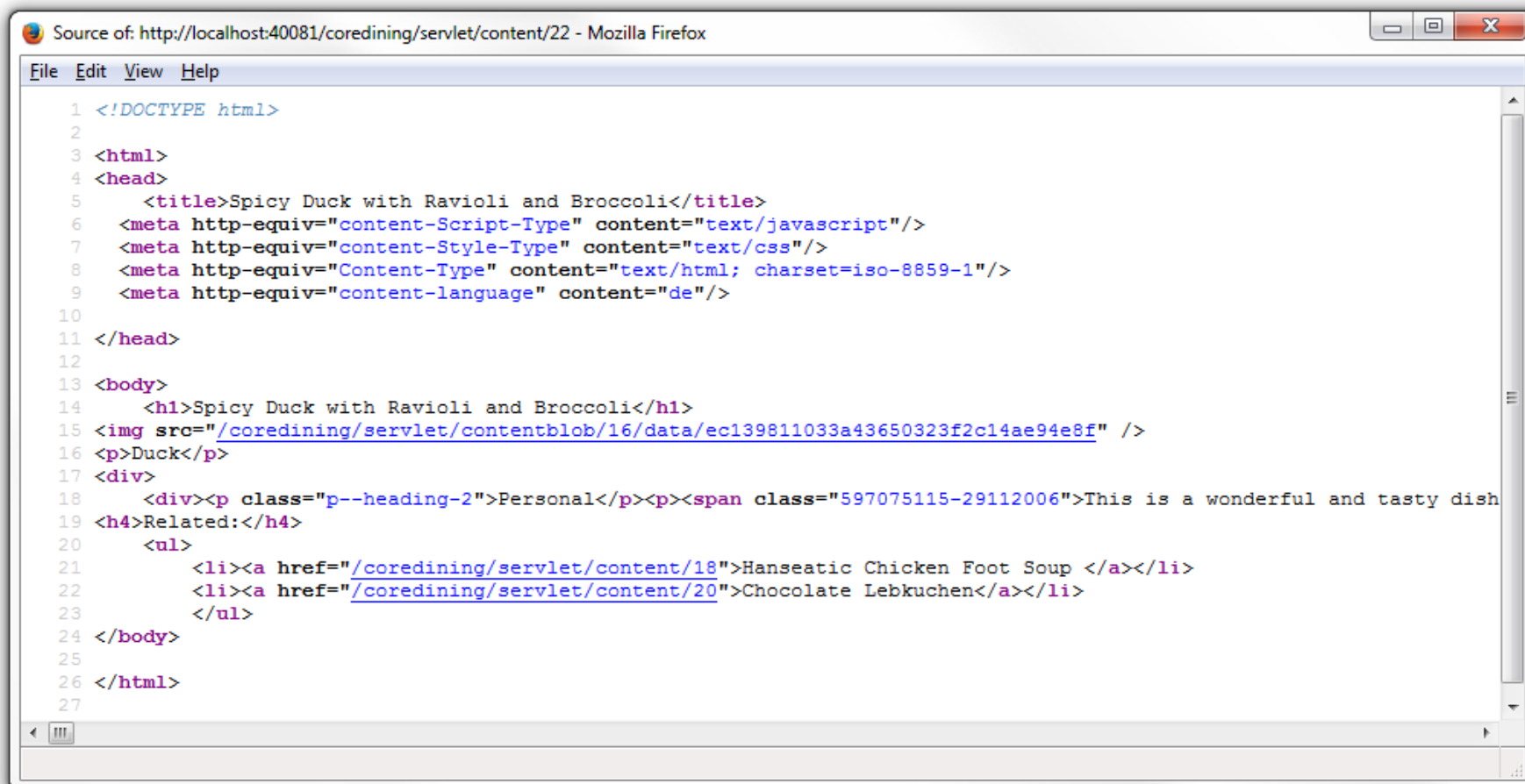
1. Copy (!) Article.jsp to Linkable.jsp (you have both files afterwards!)
2. Edit Linkable.jsp to contain `<cm:include self="${self}" view="main"/>` as only content of the body element
3. Rename Article.jsp to Article.main.jsp
4. Remove everything from Article.main.jsp, but what is inside the body element
5. Rename Image.jsp to Image.main.jsp (and modify Article.main.jsp to reflect that change)
6. Restart your webapp.
7. Add a parameter `view=main` to the URL in your browser to result in something like:
<http://localhost:40081/coredining/servlet/content/22?view=main>



- Compare the HTML sources (with and without view parameter) and explain the difference!
- What other benefits might such a construction have?

Checkpoint: Your browser should be able to display both the version with a view and the one without.

Checkpoint: Without view you have a complete HTML page, as the common frame (Linkable.jsp) adds header and footer



```
1 <!DOCTYPE html>
2
3 <html>
4 <head>
5   <title>Spicy Duck with Ravioli and Broccoli</title>
6   <meta http-equiv="content-Script-Type" content="text/javascript"/>
7   <meta http-equiv="content-Style-Type" content="text/css"/>
8   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
9   <meta http-equiv="content-language" content="de"/>
10
11 </head>
12
13 <body>
14   <h1>Spicy Duck with Ravioli and Broccoli</h1>
15   
16   <p>Duck</p>
17   <div>
18     <div><p class="p--heading-2">Personal</p><p><span class="597075115-29112006">This is a wonderful and tasty dish
19   <h4>Related:</h4>
20     <ul>
21       <li><a href="/coredining/servlet/content/18">Hanseatic Chicken Foot Soup </a></li>
22       <li><a href="/coredining/servlet/content/20">Chocolate Lebkuchen</a></li>
23     </ul>
24   </div>
25
26 </body>
27 </html>
```


Checkpoint: The same URL with the view parameter will return just a fragment...



```
Source of: http://localhost:40081/coredining/servlet/content/22?view=main - Mozilla Firefox
File Edit View Help
1 <h1>Spicy Duck with Ravioli and Broccoli</h1>
2 
3 <p>Duck</p>
4 <div>
5     <div><p class="p--heading-2">Personal</p><p><span class="597075115-29112006">This is a wonderful and
6 <h4>Related:</h4>
7     <ul>
8         <li><a href="/coredining/servlet/content/18">Hanseatic Chicken Foot Soup </a></li>
9         <li><a href="/coredining/servlet/content/20">Chocolate Lebkuchen</a></li>
10     </ul>
11
```

Exercise 12

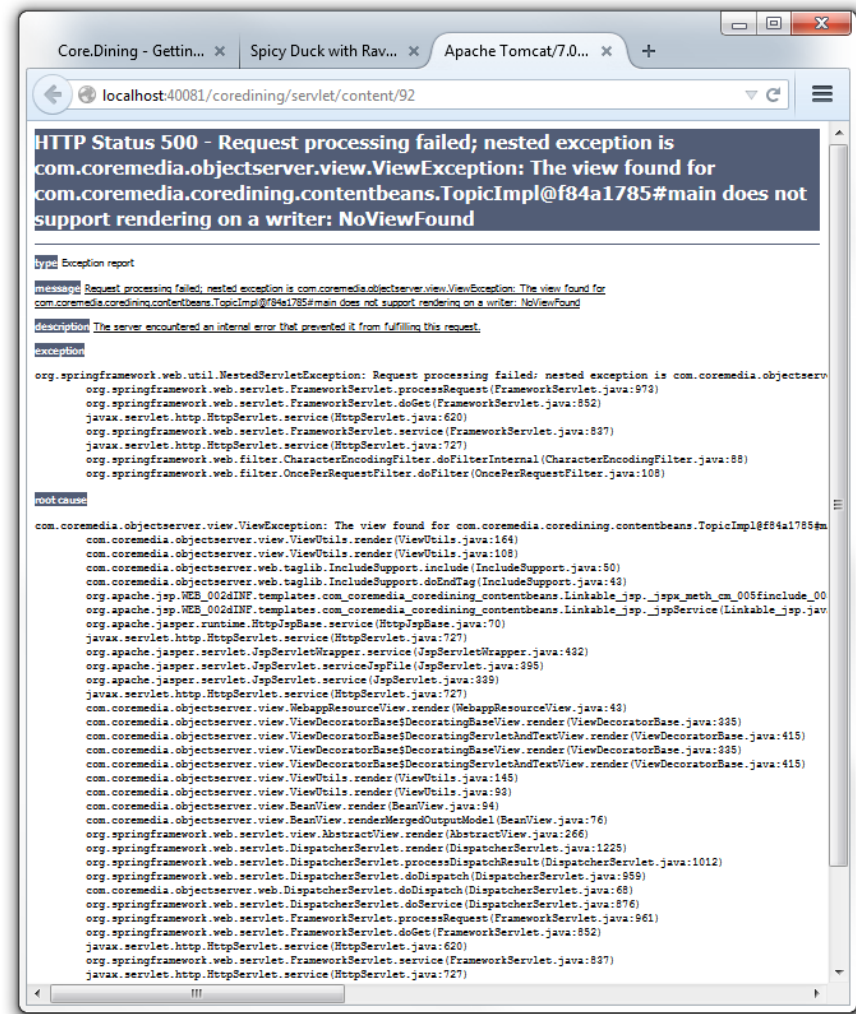
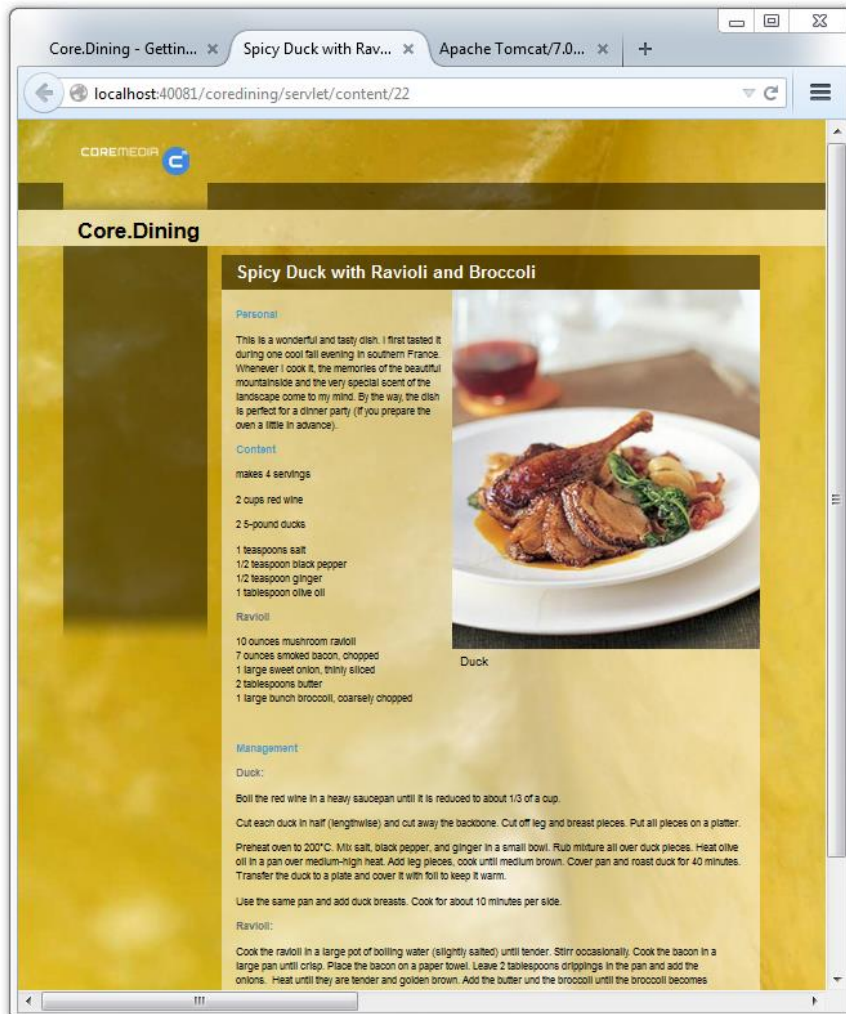
Task: Overview pages – Layout beautified

Apart from the technical level, the looks of an application can make a big difference as well. Give your application an even better look by adding CSS. The `Linkable.jsp` is a good place for adding CSS to all your pages.

1. For the beautification of your site:
Copy `Article.main.jsp`, `Image.main.jsp`, and `Linkable.jsp` from the directory `exercise-material/12` to your template directory - they simply add `<div>` tags with IDs and classes to activate the CSS.
2. Explore these new templates.
3. In `Linkable.jsp` change the line containing the reference to the background style sheet from `"background.css"` to `"background-beans.css"`.
4. Refresh your browser.
5. Try the other options: `background-banana.css`, `background-starfruit.css`, and `background-orange.css`.

Checkpoint: Your articles are rendered correctly. The overview pages are still throwing an exception. What template is missing?

Checkpoint: Article pages look nice. Overview pages still show a NoViewFound error.



Exercise 13

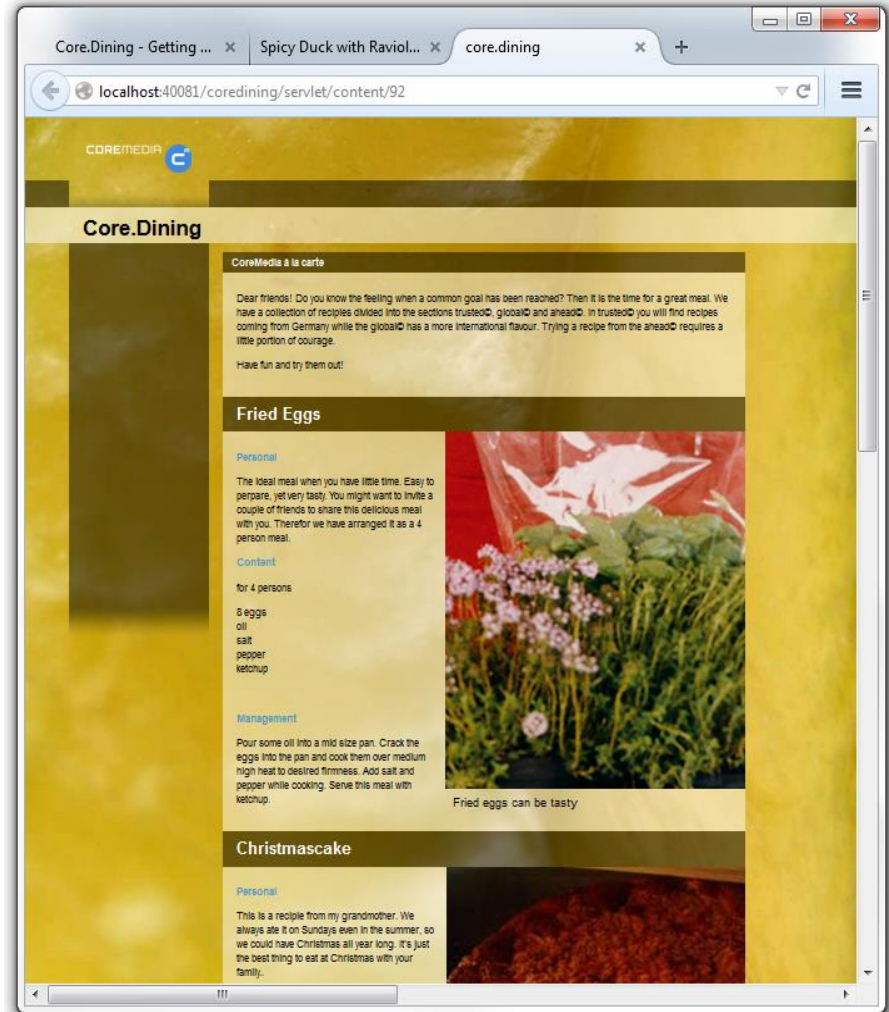
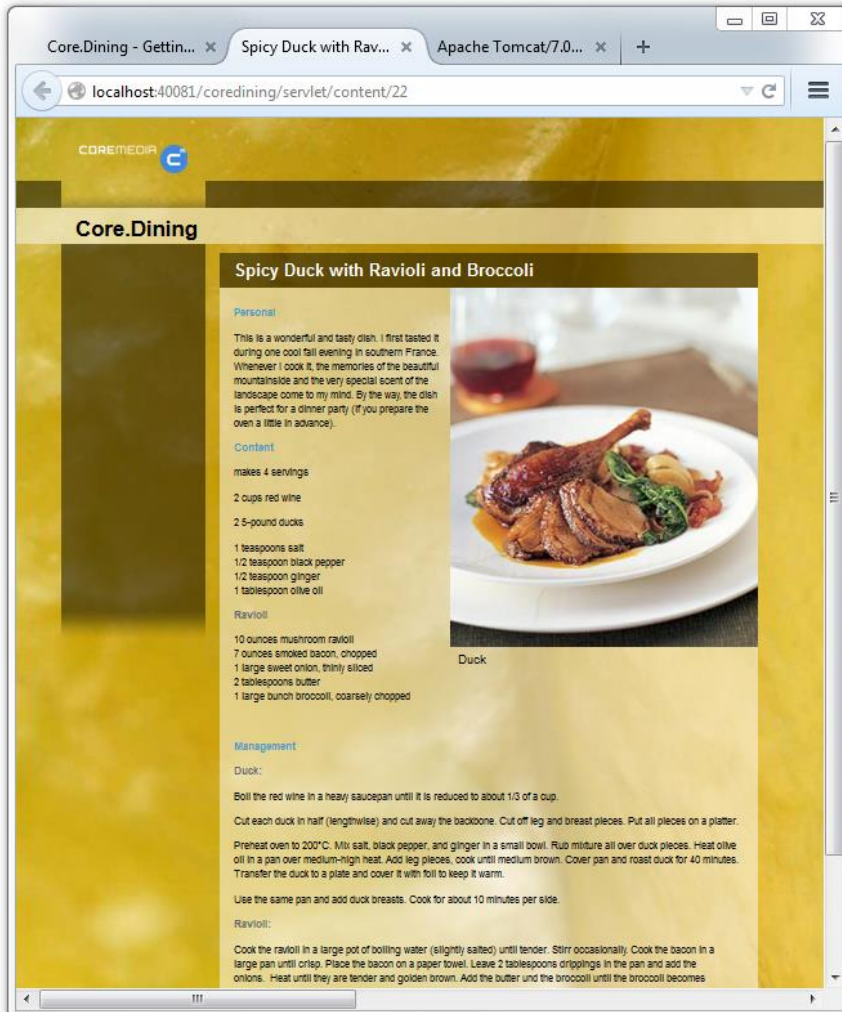
Task: A new "main" view for topics and containers

Now it is time to fix our problem with the overview pages. The "main" view for Topics will iterate the link list "contentContainers" which contains Containers. These Containers will then iterate their contents-lists which contain Articles.

1. Now we need a "main" view that displays all the containers linked to a topic. This requires a "main" view for the container as well.
Copy the prepared Topic.main.jsp, and Container.main.jsp from the exercise-material to your template directory.
2. Complete the code in Container.main.jsp. In here you should iterate the list property "contents" using **<c:forEach>** and include the "main" view for every item in the list.
3. Check the Topic.main.jsp: there you will find an include of containers with the view "tableOfTeasers".
4. Make a copy of your Container.main.jsp and rename this copy to Container.tableOfTeasers.jsp. (We will refactor this template in a later exercise.)
5. Restart your web application.

Checkpoint: Article and Topics are now rendered in a nice layout, but the overview pages can be improved. What would it take to render overview pages with article teasers?

Checkpoint: Overview pages will no longer throw an exception...

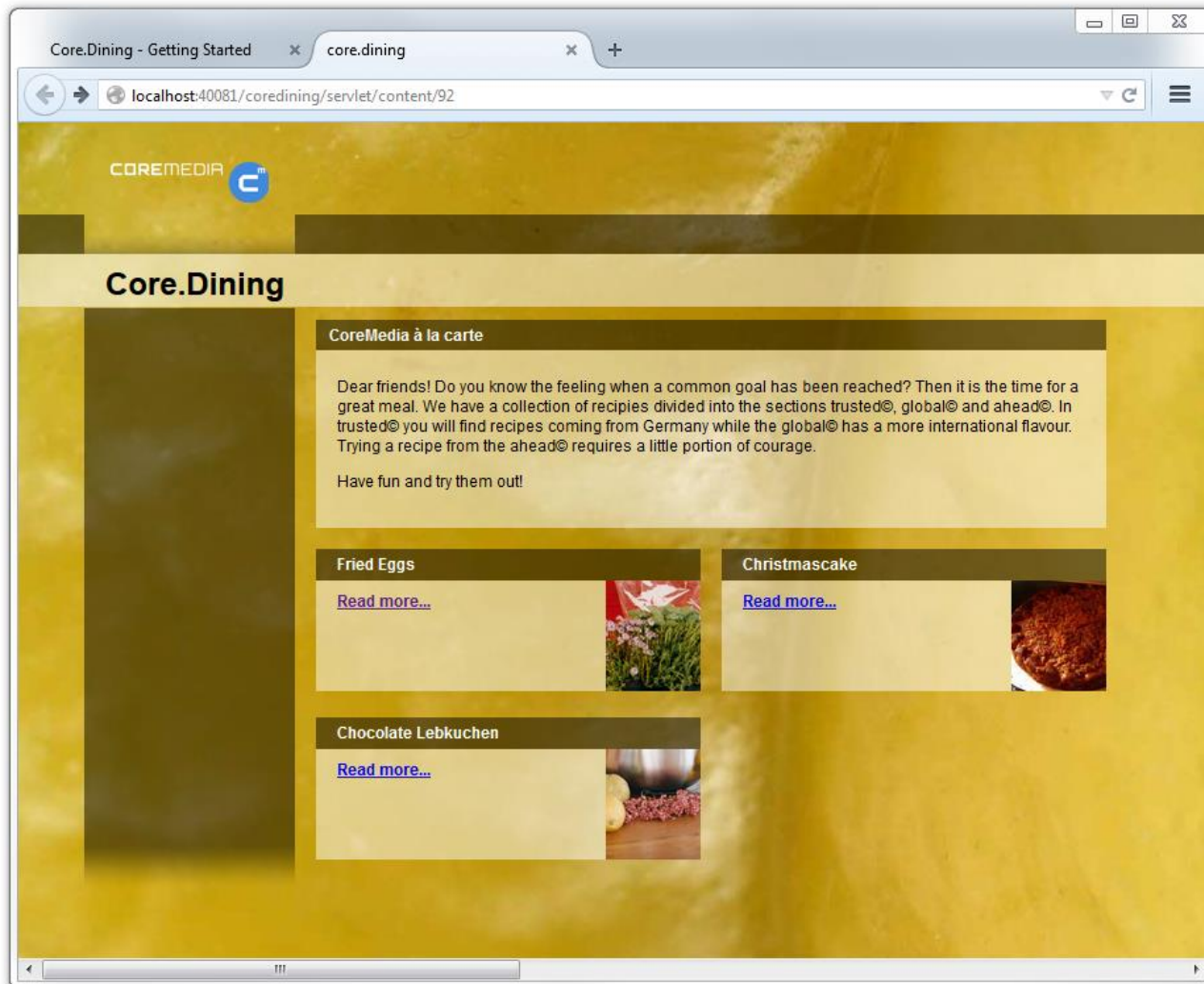


Exercise 14

Task: A teaser view for Article

1. In our overview pages articles should not be displayed in their full version. Write JSP templates to neatly arrange them in a short teaser version.
2. To do that copy Article.teaser.jsp and Container.tableOfTeasers.jsp from the exercises material folder to your template directory.
3. Examine both templates.
4. The Article.teaser.jsp requires a "*thumbnail*" view for Images. Therefore create a copy of the existing template Image.main.jsp and call it Image.thumbnail.jsp.
5. Edit Image.thumbnail.jsp to remove the caption and add the attribute *class="thumbnail"* to the tag ``.
6. Restart your web application.

Checkpoint: Articles now display neatly as teasers in overview pages.



Exercise 15

Task: Add the navigation

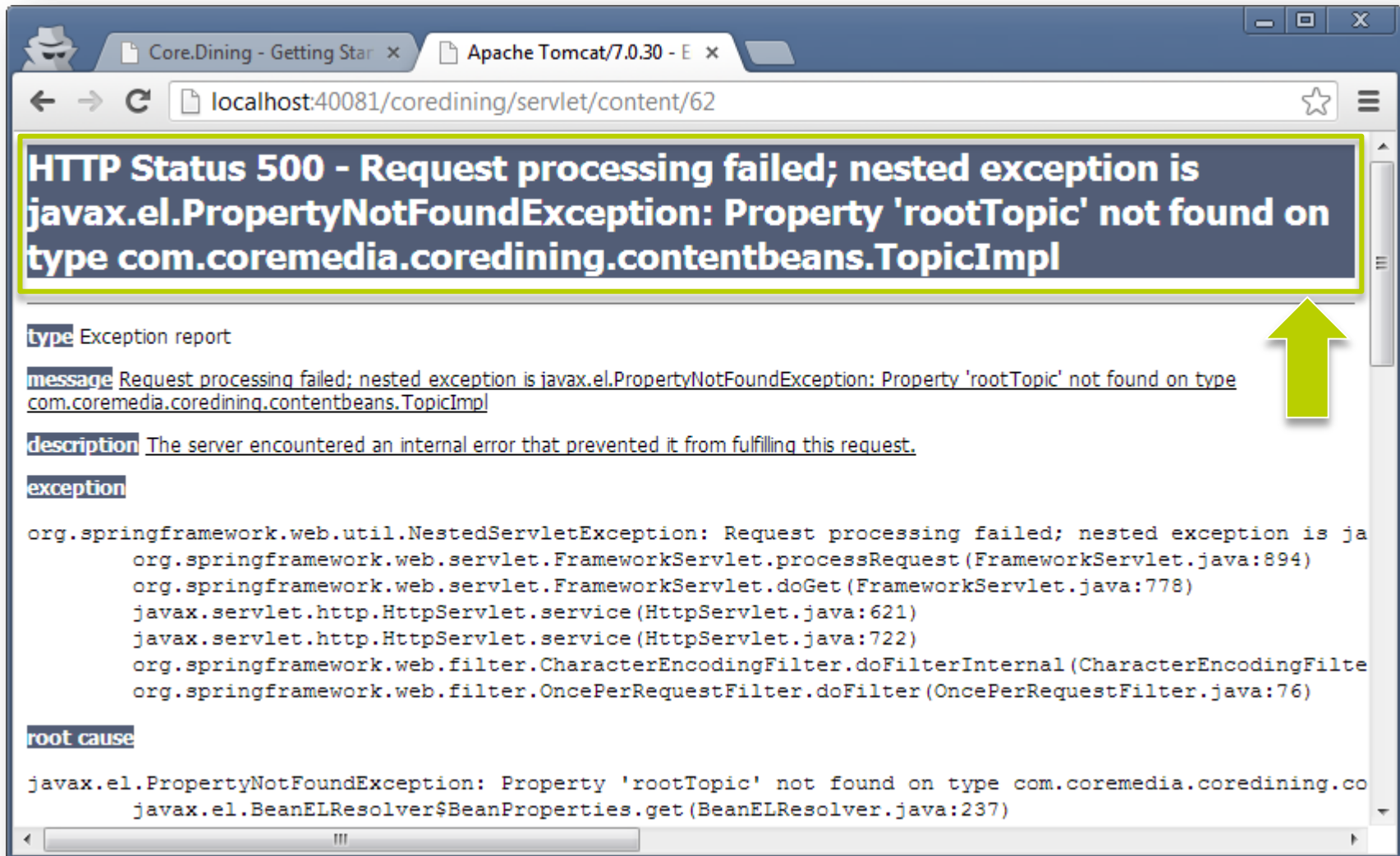
1. Copy Linkable.jsp, Topic.navigation.jsp, and Topic.navigationRecursively.jsp from the exercises material folder to your template directory.
2. Restart your web application to apply the changes.
3. When you refresh your browser you will see that a number of properties (e.g. "rootTopic") are missing in your TopicImpl bean.
4. To add them copy the prepared java sources Topic.java and TopicImpl.java to your contentbean package as a starting point
5. Fill in the method bodies as described in that file.
6. Restart your web application to apply the changes.
7. Click on the Article link to see the navigation
8. Click on the navigation to see an error.



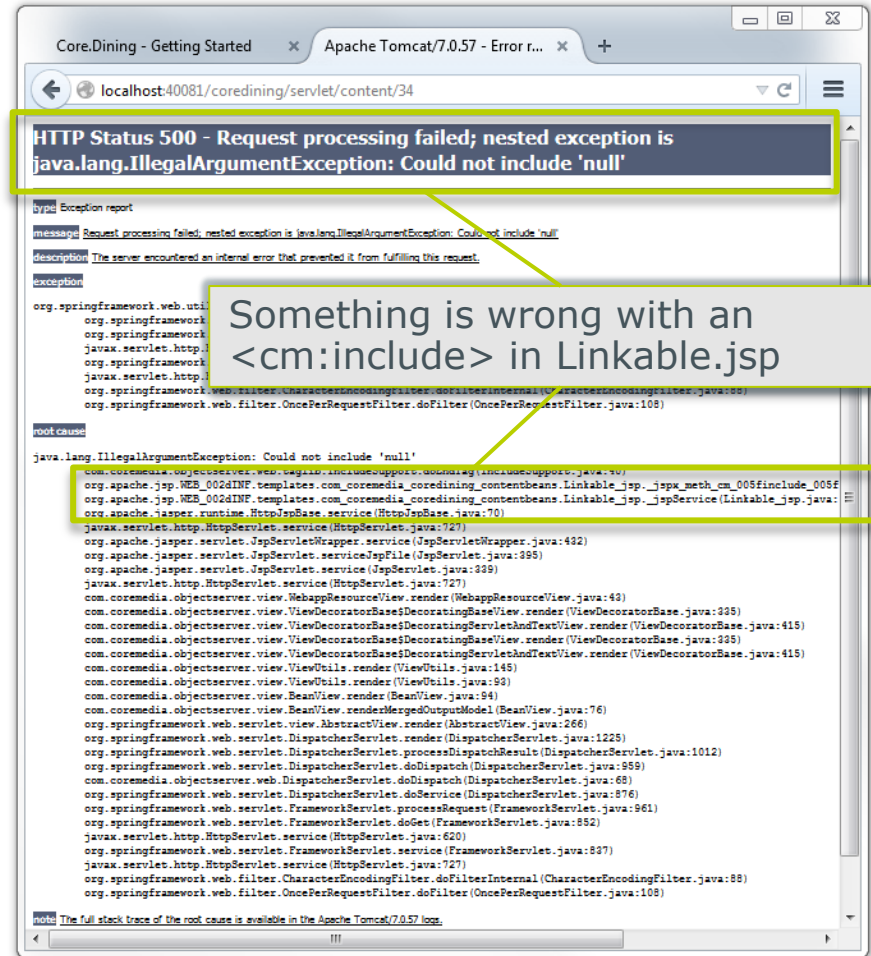
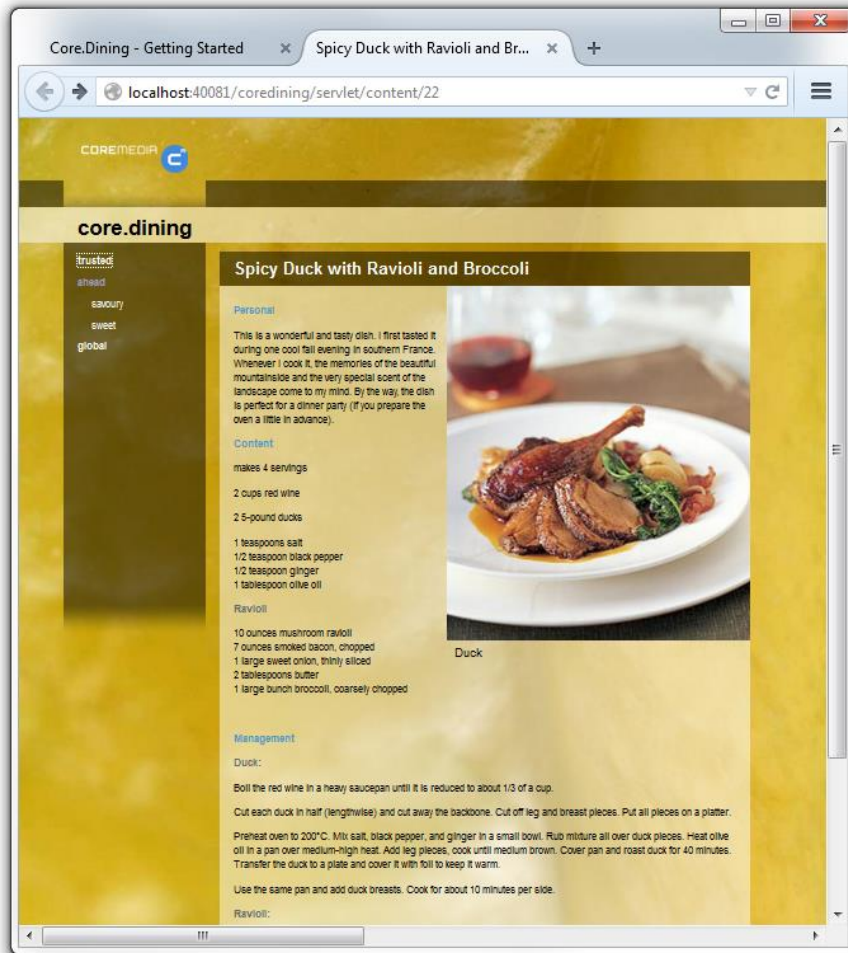
- Why does your navigation does not work for overview pages (Topics)?
- Can you identify the problem from the stacktrace?

Checkpoint: You should see a navigation in your article pages, but an error for overview pages.

Checkpoint: After copying the templates, some properties are missing...



Checkpoint: After implementation of the required methods in Topic: Articles work, Topics don't



Exercise 16

Task: Add a basic navigation handler

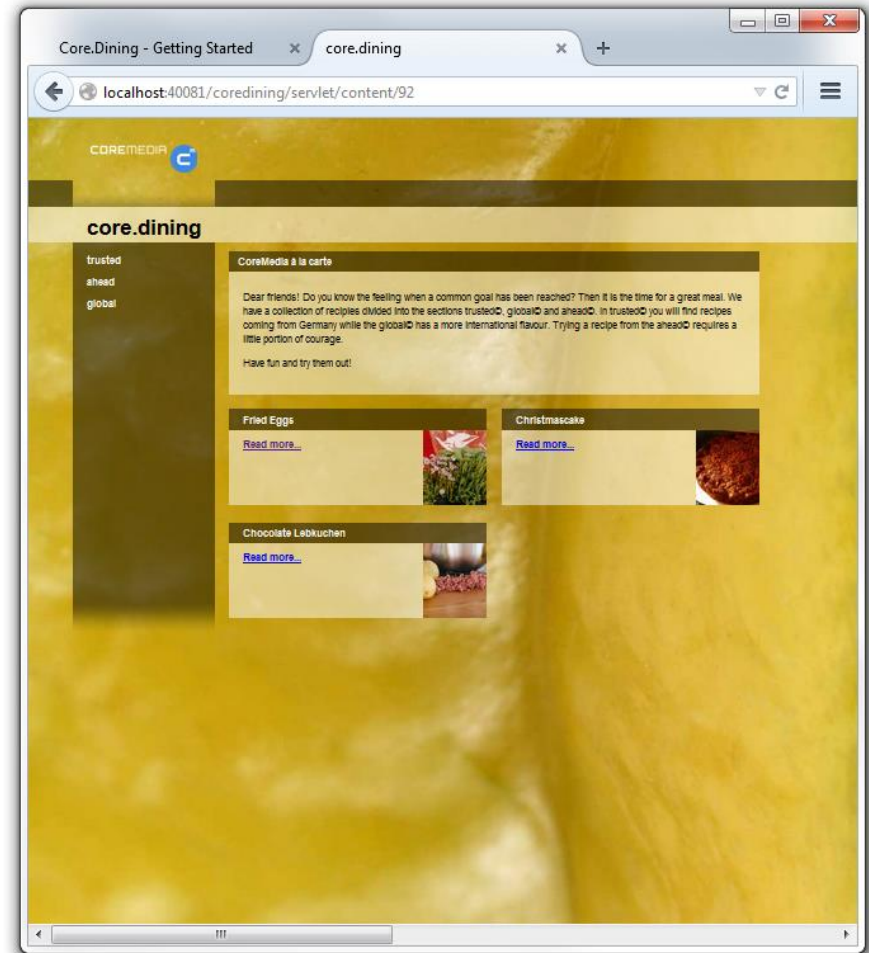
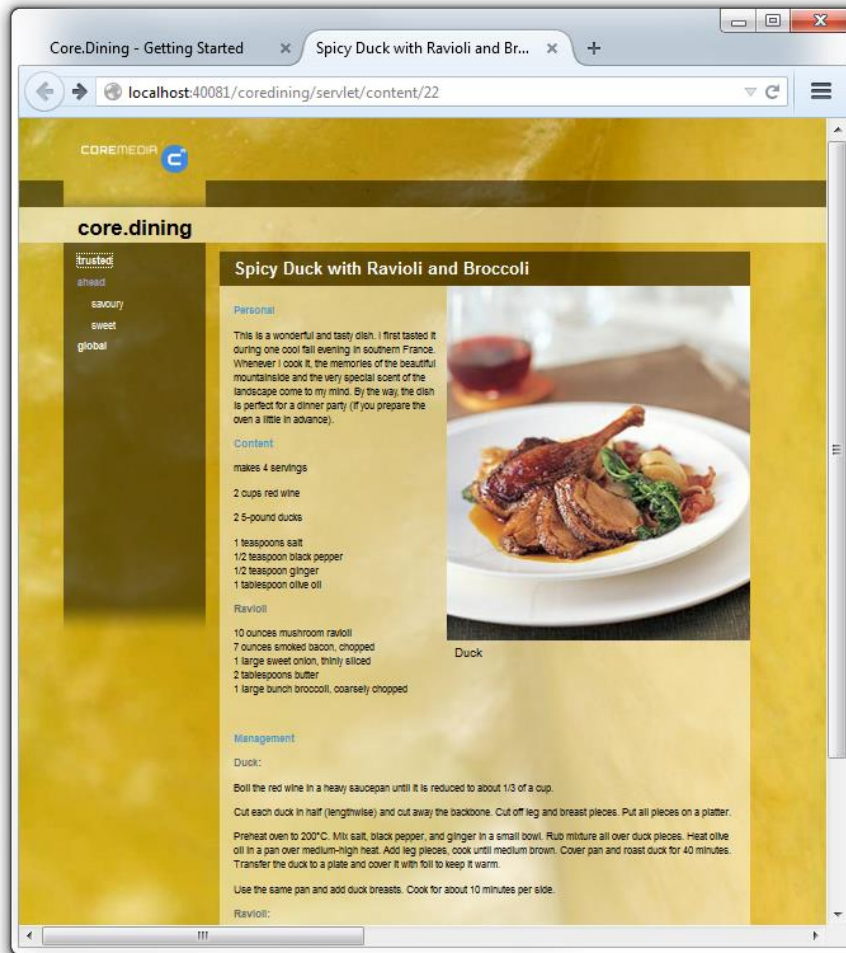
Write a handler that always chooses a reasonable Topic for the navigation no matter what type the content bean to be displayed is.

1. To do that copy the class *CustomContentHandler* from the exercises material folder to the package *com.coremedia.coredining.handlers*.
2. Add the correct pattern of the annotation `@RequestMapping` of the handler method
3. Finish the code in method *handleContent(..)* to properly set the variable *"currentTopic"*
4. Activate your handler by adding a bean definition to framework/spring/coredining-handlers.xml: replace the current bean definition with `id="contentViewHandler"` by a bean of your *CustomContentHandler* class.
5. In *Linkable.jsp* replace `"${self.homeTopic[0]}"` with `"${currentTopic}"` to make use of your changes
6. Restart your web application.



- ➔ Try out the URL `"/content/SpicyDuck.html"` – would this handler handle the request?
- ➔ If true, how could you prevent it?

Checkpoint: Using the "currentTopic" for including the navigation solved our problem.



Exercise 17

Task: Write a handler for search engine optimized URLs

In this exercise we are implementing a search engine optimized URL format for our website. Your work will be based on the `OptimizedContentHandler` class which you can find in the exercise-material folder.

The `RequestMapping` part of this handler is already implemented. Have a look to the code in order to see how it works.

1. Implement the `buildLink()` method. This method returns a map of path variables which will be used in the `URI_PATTERN`. You need to evaluate these variables.

For your convenience the `OptimizedContentHandler` already contains a set of prepared helper methods which you should use in your implementation. Follow the instructions in the `TODO`-comment of the `buildLink()` method.

2. Examine the prepared methods that you have used in your implementation.
 1. Where does the `currentTopic` come from?
 2. How is the Site document derived from the `currentTopic`?
3. Compare your `buildLink()` implementation with its counterpart `handleContent()`.
4. Add the handler class to your spring application context. The preferred spring configuration file is framework/spring/coredining-handlers.xml.

You will find the spring configuration on the next page.

Handler Configuration

```
<context:annotation-config/>
```

```
...
```

```
<bean id="optimizedContentHandler"  
      class="com.coremedia.coredining.handlers.OptimizedContentHandler">  
  <property name="contentRepository" ref="contentRepository" />  
  <property name="contentBeanFactory" ref="contentBeanFactory" />  
</bean>
```

Exercise 18

Task: Replace static inclusion of style sheets in Linkable.jsp with a dynamic list of sheets from the CMS

1. Start with a set of prepared JSPs from the exercises material folder. Copy them to your template directory.
2. They use types "*PageLayout*", "*ClientCode*", and "*Symbol*" to allow an individually configurable layout per "Topic"
3. Explore the use of these content types in the core.dining document model.
4. Examine the templates. Make sure you understand what happens inside these JSPs
5. Restart your web application to apply the changes.
6. You will notice that only the top level Topics have page layouts configured. This is not a bug! It is common to web applications, that configuration settings should be inherited from top level topics to their sub topics.
7. Override *TopicBase#getPageLayout()* in *TopicImpl.java* to use the page layout of the parent topic in case no page layout has been set for that specific topic
8. Restart your web application.
9. Use the CMS editor to change the page layouts of a topic.

Exercise 19

Task: Refactor your application for cae-live-webapp.

1. Import the following additional maven projects into your workspace:
 - modules/cae/cae-base-component
 - modules/cae/cae-base-lib
 - modules/cae/contentbeans
2. Refactor your workspace. You will find an ant target "refactor-workspace" in the build.xml inside the module "cae-preview-webapp". This target will move all resources to their correct locations. Make sure, that your preview CAE is stopped before executing this target.
3. After refactoring you will find your cae-preview-webapp sources directory almost empty. Have a look at the modules "cae-base-lib", "contentbeans" and "cae-base-components".

Find out:

1. Where are your templates?
 2. Where is the spring configuration
 3. Where are the content beans and the other CAE specific java sources?
4. Open a console in modules/cae and rebuild the complete cae aggregator:
`mvn clean install -DskipTests`

Continued on next page...

Exercise 19, continued

Task: Test your cae-live-webapp

1. Start the master live server.
Use the start script `workspace/modules/server/master-live-server-webapp/run.bat`
2. Check that the master live server is running. The IOR of the master live server is <http://localhost:42080/coremedia/ior>
3. Make sure that all your content is checked in and published. For approval and publication you can use the script `05. approve and publish.bat` at the root folder of your training installation.
4. Start the `cae-live-webapp` using the command "mvn tomcat7:run" from within the cae-live-webapp module directory. You can also use the run.bat script.
5. Check the live web application in your browser:
<http://localhost:49080/coredining/servlet/content/220>

Checkpoint: Your Live CAE should display the published content of the core.dining website.

Since we distributed our sources over several modules, we need a different build strategy in order to compile all sources.

- Move into the directory `workspace/modules/cae`
- Execute "mvn clean install -DskipTests -pl :cae-preview-webapp -am"

Exercise 20

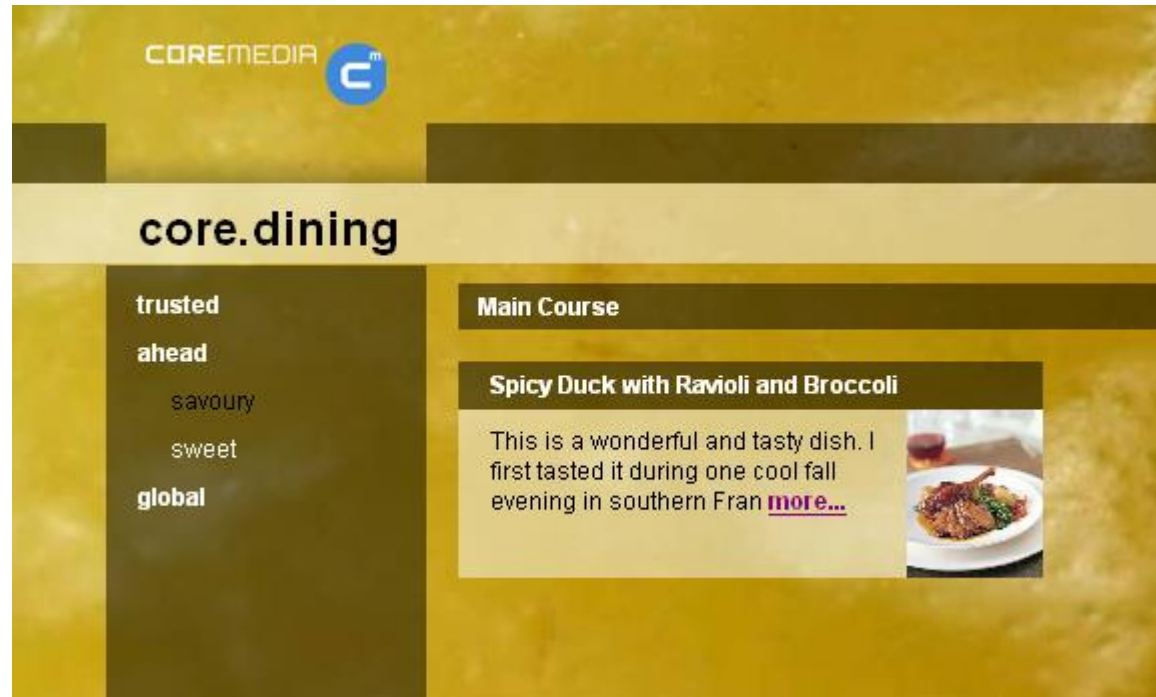
A teaserText-View for CoreMedia Richtext

We are going to write a programmed view that truncates the text of an article (Markup, CoreMedia Richtext).

1. To do that copy MarkupTeaserTextView.java into the right package of your project
2. Properly implement method "render" to only display the first 100 characters of the markup object. Your code should use the Markup#writeOn(ContentHandler) method. The SAX-ContentHandler is already implemented as an inner class.
3. Change your Article.teaser.jsp template to render a short version of its text using the view "teaserText".
4. Open the spring configuration file framework/spring/coredining-views.xml
5. Add a bean for your programmed view class to this file.
6. Adapt the customizer for the "programmedViews" bean. Add an entry for Markup objects that are viewed with the view "teaserText".
7. Restart your web application.

Checkpoint: When you refresh your browser the teasers should display with the first words of the article.

Checkpoint: This teaser view now has the final look. The first letters of the Article without formatting and headings are displayed.



Exercise 21

Provide PDF rendering for your Articles

1. Use the prepared ServletView [ArticlePdfView.java](#) as starting point of your implementation. This class uses the Open-Source library "itext" to create PDF output.
2. Configure your implementation as the view "*pdf*" for *Articles*.
3. Add a link to the "*main*" view of *Article* to generate PDF
4. You can use the prepared PDF-image as link-image - you might want to copy it to [src/main/resources/META-INF/resources/img](#) in order to have it deployed along with your application. Use the tag

```
<c:url value="/img/pdf-logo.png" />
```

to get the URL for the image.

5. After all restart your web application

Exercise 22

Write your own View Variant extension

1. Have a look at the prepared files in the exercises material folder. Use them if you like or try to do it without them.
2. First add the interface `HasViewVariant` to the module `contentbeans`. Let `LinkableImpl` implement this interface. Use link list property "view" of `Linkable` to find out the name of the View Variant.
3. Implement the interface `RenderNodeDecorator`. Use the interface `HasViewVariant` to get the view variant for the given bean.
4. Write a simple `RenderNodeDecoratorProvider` which returns you `RenderNodeDecorator` and register it in the `coredining-views.xml`.
5. Simplify the `Topic.main.jsp`: instead of using a `<c:choose>` tag, simply iterate the `contentContainers` of the topic and include every container with the view "main".
6. Rename the `Container.tableOfTeasers.jsp` to `Container.main[tableOfTeasers].jsp`. Copy the template `Container.main[listOfTeasers].jsp` from the material folder to your workspace.
7. Rebuild and restart the web-application and test it. You can use the Site Manager to change the view variant of your containers. Check if this has an effect in your web application.

Exercise 23

Enabling Data View Caching

1. The dataviews configuration file at "contentbeans/src/main/resources/framework/dataviews/dataviews.xml"
2. Add a dataview configuration for the following types and properties:
 - ArticleImpl: title, text, image, related, homeTopic
 - ImageImpl: title, data, caption
 - TopicImpl: title, subTopics, rootTopic, pathElements
3. Set cachesizes for these types to a meaningful value. Configure a total number of 1000 dataviews.
4. Set the associationType to "static" for all those properties, that returns Objects for which a dataviews configuration is also available.
1. The method OptimizedContentHandler#handleContent() resolves the variable currentTopic based on the URL. This bean is not yet a dataview. Add some lines of code, that load the dataview of the calculated currentTopic. You need to add a property of type DataViewFactory and a corresponding setter-method to your handler class. Don't forget to set the DataViewFactory property in your spring application context. (e.g. by adding `<property name="dataViewFactory" ref="dataViewFactory" />` to your bean).

Continued on next page...

Exercise 23

Check your DataView Caching

1. Rebuild (make) your cae-base-webapp and start it with the eclipse debugger.
2. Set a breakpoint to the following code lines:
 - Start of CustomContentHandler#handleContent()
 - Start of OptimizedContentHandler#handleContent()
 - ImageBase#getData(), ArticleBase#getText()
3. Open an article page in your browser and step through the handler code.
4. Observe the following behaviour:
 - The get methods of the ContentBeans will be only called once. Why?
 - What is the type of the parameter self in the handler methods?
 - What properties are the properties and values of the bean self?
 - What is the type of the variable "currentTopic" before and after you load the dataview for it.
 - What happens to the article if you test your dataviews with a topic page?

QUIZ: Would it be a good idea to load the dataview of the variable "site" in OptimizedContentHandler#handleContent() instead of loading the dataview of the currentTopic?

Checkpoint: Your method gets called from the data view subtype upon initialization.

The screenshot shows an IDE with a breakpoint set in the `TopicImpl.getRootTopic()` method at line 21. The call stack on the left shows the execution path starting from the thread suspension point, through `TopicImpl.loadrootTopic()`, `AbstractDataViewFactory$Key.doInstantiate()`, `AbstractDataViewFactory$Key.evaluate(Cache)`, `Cache.toValue(Object)`, `Cache.doCompute(Object)`, `Cache.compute(Object)`, `Cache.make(Object, boolean)`, `Cache.getOrPeek(Object, boolean)`, `Cache.get(Object)`, `CodeGeneratingDataViewFactory(AbstractDataViewFactory).load(Object, String, boolean)`, `TopicImpl.getParent()`, `TopicImpl.getPathElements()`, and finally `TopicImpl.loadpathElements()`. The variable inspector on the right shows the state of the `this` object, including `dataViewFactory`, `initialized`, `content`, `contentContainers`, `definition`, `i18n`, `pageLayout`, `parent`, `pathElements`, `relatedContainers`, and `rootTopic`.

```
13 public class TopicImpl extends TopicBase implements Topic {
14
15     /**
16      * Traverses the topic hierarchy to find the root of the topic tree.
17      *
18      * @return the root topic of this topic (might be the topic itself)
19      */
20     public TopicImpl getRootTopic() {
21         TopicImpl toReturn = this;
22
23         TopicImpl current = toReturn.getParent();
```


Exercise 24

Display non-CMS content

1. Extend the document model with the type "ExternalArticle" having string property "externalId" that inherits from "Article". The property "externalId" denotes the primary key of the external content
2. The document type model of core.dining is located in the module "contentserver-component" at the following location:
[src/main/resources/framework/doctypes/core.dining/](#)
3. Stop all running content server instances
4. Rebuild the server aggregator module using "mvn clean install"
5. Restart the content management server webapp.
6. Create a new DB scheme with name "external" and also a login role with name and password "external".
7. Use the task "create-external-article" of the ant script "[build.xml](#)" located in the [cae-preview-webapp](#) directory to create some external recipes. You might want to enter some more meaningful recipes in the ant build file...

Continued on next page...

Exercise 24

Display non-CMS content

8. Implement the content bean `ExternalArticleImpl` manually, there is just one method `"getId()"` to be implemented. (You don't need base classes and interfaces, because your `External Article` will extend from `ArticleImpl`.)
9. Copy `"ArticleDAO.java"` and `"DbJdbcArticleDAO.java"` and the `coredining-externalcontent.xml` to you project workspace. Examine the code of `DbJdbcArticleDAO`.
10. Add a setter for the Article DAO in `"ExternalArticleImpl.java"`. Configure the DAO in the `coredining-contentbeans.xml`.
11. Overwrite the getter methods `"getTitle()"`, `"getText()"`, `"getKeywords()"` in `ExternalArticleImpl`. Use the DAO object to receive the title, text and keywords from the database instead of the content server.
12. Notice that no additional JSPs are required!
13. Add some documents of type `"ExternalArticle"` and link them to the external content – make sure that you select a home topic and type in a valid `externalId` (one of these: 1, 2, 3, 4). You might need to restart the editor to notify it of the new document type.
14. Preview these documents to check your code actually works