# JSF Usage

## Table of Contents

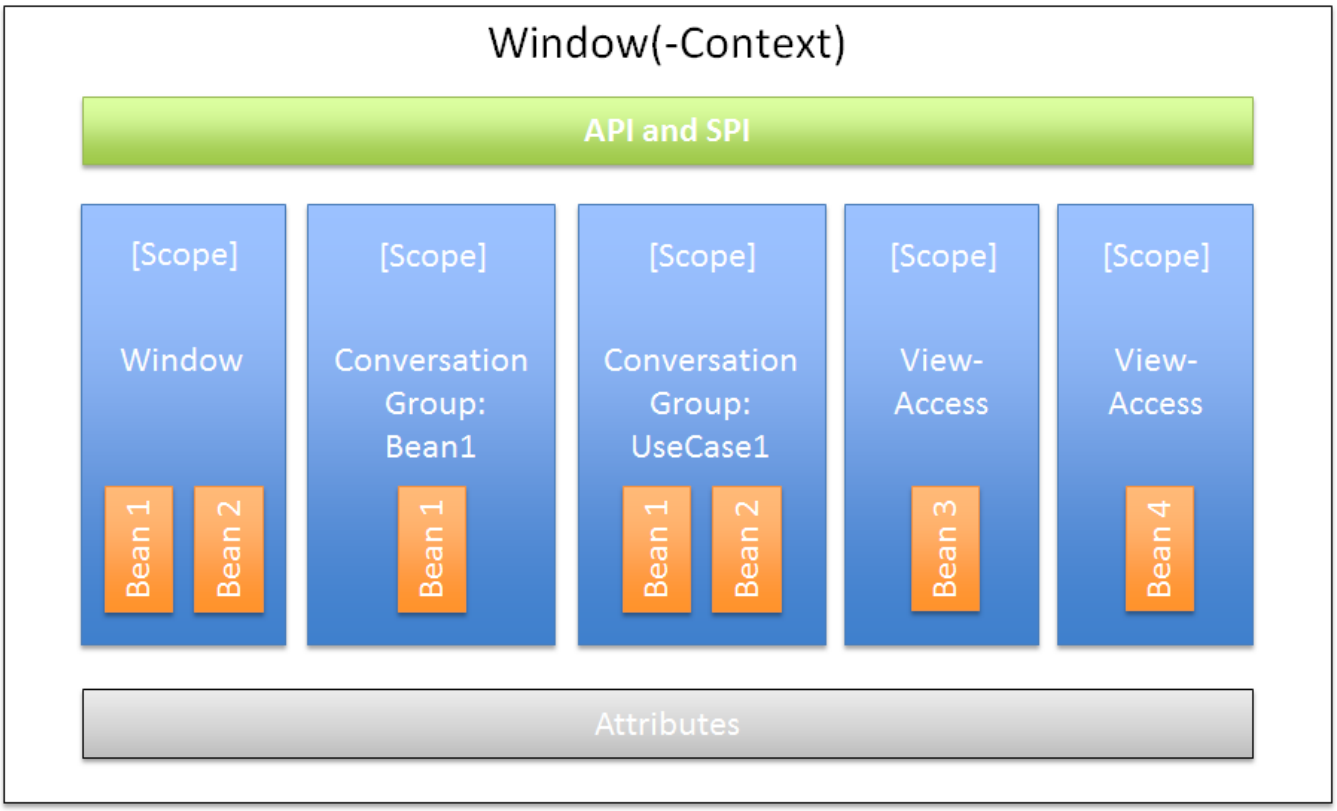
## Intro

The Intro page provides an overview, the setup of this module and describes the motivation for the features described below. This page explains the most important APIs and mechanisms of the JSF module provided by CODI. Please note that this page doesn't show all possibilities. **If you have any question, please contact the community!**

## Scopes

All CODI scopes have in common that they are bound to a window. A window is represented by the `WindowContext` which stores all scopes and allows to control the window and its scopes in a fine-grained manner.

Besides this documentation http://www.slideshare.net/os890/myfaces-codi-conversations provides a basic overview.




### Conversation Scope

First of all it's **important** to mention that CODI starts (grouped) conversations automatically as soon as you access conversation scoped beans. Furthermore, the invocation of `Conversation#close` leads to an immediate termination of the conversation.

**A CODI conversation scoped bean**

```
import org.apache.myfaces.extensions.cdi.core.api.scope.conversation.ConversationGroup;

@ConversationScoped
public class DemoBean1 implements Serializable
{
  //...
}
```

... leads to a conversation which contains just one bean with the group DemoBean1.

That's it!

As soon as you access the bean, the conversation gets started!
Hint

If you would like to use the bean within your JSF pages, you have to add `@Named` (`javax.inject.Named` ).
Further details

Currently you will find also some further information here. We are going to merge both pages.

## Grouped Conversations

(In case of **CDI std. conversations** there is just one big conversation which contains all conversation scoped beans.)
The grouped conversations provided by CODI are completely different. By default every conversation scoped bean exists in an "isolated" conversation. That means there are several parallel conversations within the same window.

Example:

**Separated CODI conversations**

```
@ConversationScoped
public class DemoBean2 implements Serializable
{
  //...
}

@ConversationScoped
public class DemoBean3 implements Serializable
{
  //...
}
```

... leads to two independent conversations in the same window (context).
If you close the conversation of `DemoBean2`, the conversation of `DemoBean3` is still active.

If you have an use-case (e.g. a wizard) which uses multiple beans which are linked together very tightly, you can create a type-safe conversation group.

**Grouped conversation scoped beans**

```
interface Wizard1 {}

@ConversationScoped
@ConversationGroup(Wizard1.class)
public class DemoBean4 implements Serializable
{
  //...
}

@ConversationScoped
@ConversationGroup(Wizard1.class)
public class DemoBean5 implements Serializable
{
  //...
}
```

You can use `@ConversationGroup` to tell CODI that there is a logical group of beans. Technically `@ConversationGroup` is just a CDI qualifier. Internally CODI uses this information to identify a conversation. In the previous example both beans exist in the same conversation (group). If you terminate the conversation group, both beans will be destroyed. **If you don't use `@ConversationGroup` explicitly, CODI uses the class of the bean as conversation group.**

**Injecting a conversation scoped bean with an explicit group**

```
//...
public class CustomBean1
{
  @Inject
  @ConversationGroup(Group1.class)
  private CustomBean2 demoBean;

  @Inject
  @ConversationGroup(Group2.class)
  private CustomBean2 demoBean;
}
```

Since `@ConversationGroup` is a std. CDI qualifier you have to use it at the injection point. You have to do that esp. because it's possible to create beans of the same type which exist in different groups (e.g. via producer methods).

Example:

**Producer methods which produce conversation scoped beans with different groups**

```
interface Group1 {}
interface Group2 {}

public class CustomBean2
{
  @Produces
  @ConversationScoped
  @ConversationGroup(Group1.class)
  public CustomBean2 createInstanceForGroup1()
  {
    return new CustomBean2();
  }

  @Produces
  @ConversationScoped
  @ConversationGroup(Group2.class)
  public CustomBean2 createInstanceForGroup2()
  {
    return new CustomBean2();
  }
}
```

## Terminating Conversations

You can inject the conversation via `@Inject` and use it to terminate the conversation immediately (see *) or you inject the current `WindowContext` which can be used to terminate a given conversation group.

**Injecting and using the current conversation**

```
import org.apache.myfaces.extensions.cdi.core.api.scope.conversation.ConversationScoped;
import org.apache.myfaces.extensions.cdi.core.api.scope.conversation.Conversation;

@ConversationScoped
public class DemoBean6 implements Serializable
{
  @Inject
  private Conversation conversation; //injects the conversation of DemoBean6 (!= conversation of DemoBean7)

  //...

  public void finish()
  {
    this.conversation.close();
  }
}

@ConversationScoped
public class DemoBean7 implements Serializable
{
  @Inject
  private Conversation conversation; //injects the conversation of DemoBean7 (!= conversation of DemoBean6)

  //...

  public void finish()
  {
    this.conversation.close();
  }
}
```

**Injecting and using the explicitly grouped conversation**

```
interface Wizard2 {}

@ConversationScoped
@ConversationGroup(Wizard2.class)
public class DemoBean8 implements Serializable
{
  @Inject
  private Conversation conversation; //injects the conversation of Wizard2 (contains DemoBean8 and DemoBean9)

  //...

  public void finish()
  {
    this.conversation.close();
  }
}

@ConversationScoped
@ConversationGroup(Wizard2.class)
public class DemoBean9 implements Serializable
{
  @Inject
  private Conversation conversation; //injects the conversation of Wizard2 (contains DemoBean8 and DemoBean9)

  //...

  public void finish()
  {
    this.conversation.close();
  }
}
```

**Terminating a grouped conversation outside of the conversation**

```
//...
public class DemoBean10 implements Serializable
{
  @Inject
  private WindowContext windowContext; //injects the whole window context (of the current window)

  //...

  public void finish()
  {
    this.windowContext.closeConversationGroup(Wizard2.class);  //closes the conversation of group Wizard2.class
  }
}
```

**Alternative approach for terminating a conversation group**

```
//...
public class DemoBean10 implements Serializable
{
  //...

  @CloseConversationGroup(group = Wizard2.class)
  public void finish()
  {
    //...
  }
}
```

**Alternative approach for terminating a conversation group in case of an exception**

```
//...
public class DemoBean10 implements Serializable
{
  //...

  @CloseConversationGroup(group = Wizard2.class, on = MyRuntimeException.class)
  public void finish()
  {
    //...
  }
}
```

These two alternative approaches can be used for simple use-cases.

**Terminate all conversations**

```
//...
public class DemoBean11 implements Serializable
{
  @Inject
  private WindowContext windowContext;

  //...

  public void finish()
  {
    this.windowContext.closeConversations();  //closes all existing conversations within the current window
(context)
  }
}
```

Hint

Since the View-Access scope is just a different kind of a conversation `#closeConversations` also terminates all view-access scoped beans. There will be a SPI to customize this behavior. Usually you will need `#closeConversations` e.g. if the user triggers a navigation via the main-menu and in such a case you usually exit the current use-case. So it makes sense that all kinds of conversations will be closed.

Hint

CODI conversations get closed/restarted immediately instead of keeping them until the end of the request like std. conversations do, because the behaviour of std. conversations breaks a lot of use-cases. However, if you really need to keep them until the end of the request, you can use EnhancedConversation#end which is provided by the https://bitbucket.org/os890/codi-addons/src/d2e11ac5e941/enhanced_conversations/.

## Restarting conversations

Instead of destroying the whole conversation the conversation stays active and only the scoped instances are destroyed. (The conversation will be marked as accessed.) As soon as an instance of a bean is requested, the instance will be created based on the original bean descriptor. This approach allows a better performance, if the conversation is needed immediately (e.g. if you know in your action method that the next page will/might use the same conversation again).

**Restarting a conversation**

```
@ConversationScoped
public class DemoBean12 implements Serializable
{
  @Inject
  private Conversation conversation;

  //...

  public void finish()
  {
    this.conversation.restart();
  }
}
```

Hint

Compared to std. CDI conversations CODI provides completely different conversation concepts. "Just the name is the same."
So please don't try to use the same implementation patterns which you might have learned for std. conversations.
CDI conversations are comparable to MyFaces Orchestra conversations.

## Sub-Conversation-Groups (since CODI v1.0.1)

Due to the parallel conversation concept of CODI there is no need of something like nested conversations. Just use them in parallel and terminate them in a fine-granular way as soon as you don't need them any longer. As described above, you can terminate a whole conversation-group. However, sometimes it's essential to have subgroups if you need to end just a part of an use-case instead of all beans related to an use-case. However, that isn't a replacement of sub-conversations, because a replacement isn't needed.

A sub-group is just a class or an interface used to identify a bunch of beans within a group. To terminate such a sub-group, it's just needed to pass the class/interface to the corresponding API for terminating a conversation. The sub-group gets detected autom. and instead of terminating a whole conversation-group, the beans of the sub-group get un-scoped.

**Explicitly listing beans of a sub-group**

```
public class MyGroup{}

@ConversationScoped
@ConversationGroup(MyGroup.class)
public class BeanA {}

@ConversationScoped
@ConversationGroup(MyGroup.class)
public class BeanB {}

@ConversationScoped
@ConversationGroup(MyGroup.class)
public class BeanC {}

@ConversationSubGroup(subGroup = {BeanA.class, BeanB.class})
public class MySubGroup extends MyGroup {}

or

@ConversationSubGroup(of = MyGroup.class, subGroup = {BeanA.class, BeanB.class})
public class MySubGroup {}
```

**Terminating a sub-group**

```
this.windowContext.closeConversation(MySubGroup.class)
```

As you see the class/interface of the sub-group has to extend/implement the group or you specify it via the `@ConversationSubGroup#of`. With `@ConversationSubGroup#subGroup` you can list all beans which belong to the sub-group. If you have a lot of such beans or you would like to form (sub-)use-case oriented groups, you can use implicit groups:

**Implicit sub-group**

```
public interface Wizard {}

@ConversationSubGroup(of = MyGroup.class, subGroup = Wizard.class)
public class ImplicitSubGroup
{
}

@Named("myWizard")
@ConversationScoped
@ConversationGroup(MyGroup.class)
public class WizardController implements Serializable, Wizard
{
    //...
}

this.windowContext.closeConversationGroup(ImplicitSubGroup.class);
```

In the listing above all beans which implement the `Wizard` interface will be closed as soon as you close the `ImplicitSubGroup` .

As mentioned before the concept of sub-groups is no replacement for sub-conversations, because they aren't needed with the parallel conversation concept of CODI. In most cases you won't face the need to use sub-groups. However, in some cases they allow to handle the conversation-management in an easier way.

## Window Scope

The window-scope is like a session per window. That means that the data is bound to a window/tab and it not shared between windows (like the session scope does). Usually you need the window-scope instead of the session-scope. There aren't a lot of use-cases which need shared data between windows.

The usage of this scope is very similar to normal conversations. Only the cleanup strategy is different and the concept itself doesn't need/support the usage of `@ConversationGroup`.

**Window scoped bean**

```
@WindowScoped
public class PreferencesBean implements Serializable
{
  //...
}
```

### Terminating the Window Scope/ window scoped Beans

Since `WindowContext#closeConversations` doesn't affect window scoped beans we need a special API for terminating all window scoped beans. If you don't use qualifiers for your window scoped beans, you can just inject the conversation into a window scoped bean and invoke the methods discussed above. If you don't have this constellation, you can use the `WindowContext` to terminate the window scoped beans or the whole window context. **If you terminate the whole window, you also destroy all conversation and view-access scoped beans automatically.**

**Terminate the whole content of the window**

```
//...
public class CustomWindowControllerBean1
{
  @Inject
  private WindowContext windowContext;

  //...

  public void closeWindow()
  {
    this.windowContext.close();
  }
}
```

**Terminate all window scoped beans**

```
//...
public class CustomWindowControllerBean2
{
  @Inject
  private WindowContext windowContext;

  //...

  public void finish()
  {
    this.windowContext.closeConversationGroup(WindowScoped.class);
  }
}
```

## View Access Scope

In case of conversations you have to un-scope beans manually (or they we be terminated automatically after a timeout). However, sometimes you need beans with a lifetime which is as long as needed and as short as possible - which are terminated automatically (as soon as possible). In such an use-case you can use this scope. The simple rule is, as long as the bean is referenced by a page - the bean will be available for the next page (if it's used again the bean will be forwarded again). It is important that it's based on the view-id of a page (it isn't based on the request) so e.g. Ajax requests **don't** trigger a cleanup if the request doesn't access all view-access scoped beans of the page. That's also the reason for the name @*View*AccessScoped.

**Access scoped bean**

```
@ViewAccessScoped
public class WizardBean implements Serializable
{
  //...
}
```

The usage of this scope is very similar to normal conversations. Only the cleanup strategy is different and the concept itself doesn't need/support the usage of `@ConversationGroup`.

Hint

@ViewAccessScoped beans are best used in conjunction with the CODI ClientSideWindowHandler, which ensures a clean browser-tab separation without touching the old windowId. Otherwise a 'open in new tab' on a page with a @ViewAccessScoped bean might cause the termination (and re-initialization) of that bean.

## Rest Scope

Our Rest Scope is a Conversation which intended for GET pages. On the first access to such a bean on a view which gets invoked via GET, all the f:viewParam will be parsed and stored internally. This RestScoped conversation automatically expires once the bean gets accessed via GET on another view or with a different set of f:viewParam.

**RestScoped bean**

```
@RestScoped
public class CarBean implements Serializable
{
  //...
}
```

The usage of this scope is very similar to normal conversations. Only the cleanup strategy is different and the concept itself doesn't need/support the usage of `@ConversationGroup`.

Hint

Please note that the usage of Post-Redirect-GET (PRG), e.g. via faces-redirect=true, might lead to ending the conversation and thus will delete/re-initialize the bean.

## JSF 2.0 Scope Annotations

JSF 2.0 introduced new annotations as well as a new scope - the View Scope. CODI allows to use all the CDI mechanisms in beans annotated with:

- `javax.faces.bean.ApplicationScoped`
- `javax.faces.bean.SessionScoped`
- `javax.faces.bean.RequestScoped`
- `javax.faces.bean.ViewScoped`

Furthermore, the managed-bean annotation ( `javax.faces.bean.ManagedBean`) is mapped to `@Named` from CDI.

All these annotations are mapped automatically. So you won't face issues, if you import a JSF 2 annotation instead of the corresponding CDI annotation.

## Flash Scope

CODI provides a fine grained conversation scope with multiple parallel and isolated/independent conversations within a single window as well as a view-access scope (see above). So we (currently) don't think that we need a flash scope. **Please contact us, if you find an use-case which needs the flash scope and you can't use the other CODI scopes.** Other portable extensions (like Seam 3 btw. Seam-Faces) just provide this scope because they don't have such fine grained conversations.

## GET-Requests

All CODI scopes also support navigation via GET-Requests.

With JSF2 you can use GET-Requests with `h:link` . It's **recommended** to use it instead of a plain HTML link.

**JSF2 component for GET-Requests**

```
<h:link value="My Page" outcome="myPage"/>
```

However, if you have a good reason for it and you **really** have to use a plain HTML link instead, you have to use something like:

**HTML link as an alternative to the JSF2 component for GET-Requests**

```
<a href="#{facesContext.externalContext.request.contextPath}/myPage.xhtml?windowId=#{currentWindow.id}">My
Page< /a >
```

# Dependency Injection

CODI allows using @Inject within the following JSF (1.2 and 2.0) artifacts:

- Converter
- Validator
- PhaseListener

As soon as a converter or a validator is annotated with @Advanced it's possible to use @Inject.

Example for a validator:

**@Inject within a JSF validator**

```
@Advanced
@FacesValidator("...") //use it just in case of JSF 2.0
public class DependencyInjectionAwareValidator implements Validator
{
  @Inject
  private CustomValidationService customValidationService;

  public void validate(FacesContext facesContext, UIComponent uiComponent, Object value) throws
ValidatorException
  {
    Violation violation = this.customValidationService.validate(value);
    //...
  }
}
```

Example for a converter:

**@Inject within a JSF converter**

```
@Advanced
@FacesConverter("...") //use it just in case of JSF 2.0
public class DependencyInjectionAwareConverter implements Converter
{
  @Inject
  private OrderService orderService;

  public Object getAsObject(FacesContext facesContext, UIComponent uiComponent, String value)
      throws ConverterException
  {
    if (value != null && value.length() > 0)
    {
      return this.orderService.loadByOrderNumber(value);
    }
    return null;
  }

  //...
}
```

# Life-cycle Annotations

## Phase-Listener Methods

As an alternative to a full phase-listener CODI allows to use observers as phase-listener methods.

Example:

**Global observer method for phase-events**

```
protected void observePreRenderView(@Observes @BeforePhase(RENDER_RESPONSE) PhaseEvent phaseEvent)
{
  //...
}
```

If you would like to restrict the invocation to a specific view, it's possible to use the optional `@View` annotation.

**Observer method for phase-events for a specific view**

```
@View(DemoPage.class)
public void observePostInvokeApplication(@Observes @AfterPhase(JsfPhaseId.INVOKE_APPLICATION) PhaseEvent event)
{
  //...
}
```

For further details about `DemoPage.class` please have a look at the view-config section.

Hint

If you don't need the `PhaseEvent` as parameter, you can just annotate your methods with `@BeforePhase(...)` and `@AfterPhase(...)`.

Hint

`@View` is an interceptor. The disadvantage is that intercepted beans introduce an overhead.
If you would like to use this mechanism for implementing pre-render view logic, you should think about using the `@PageBean` annotation.
Further details are available in the view-config section.

Since v0.9.1 `@View` used as class-level annotation won't lead to an interceptor and allows to use it instead of `@PageBean`.

## Phase-Listener

CODI provides an annotation for phase-listeners:

**PhaseListener configured via annotation**

```
@JsfPhaseListener
public class DebugPhaseListener implements PhaseListener
{
  private static final Log LOG = LogFactory.getLog(DebugPhaseListener.class);

  private static final long serialVersionUID = -3128296286005877801L;

  public void beforePhase(PhaseEvent phaseEvent)
  {
      if(LOG.isDebugEnabled())
      {
        LOG.debug("before: " + phaseEvent.getPhaseId());
      }
  }

  public void afterPhase(PhaseEvent phaseEvent)
  {
    if(LOG.isDebugEnabled())
    {
      LOG.debug("after: " + phaseEvent.getPhaseId());
    }
  }

  public PhaseId getPhaseId()
  {
    return PhaseId.ANY_PHASE;
  }
}
```

If you have to specify the order of phase-listeners you can us the optional **`@InvocationOrder`** annotation.
In combination with `@Advanced` it's possible to use dependency injection.

Example:

**@Inject within a JSF phase-listener**

```
@Advanced
@JsfPhaseListener
@InvocationOrder(1) //optional
public class DebugPhaseListener implements PhaseListener
{
  @Inject
  private DebugService debugService;

  public void beforePhase(PhaseEvent phaseEvent)
  {
    this.debugService.log(phaseEvent);
  }

  public void afterPhase(PhaseEvent phaseEvent)
  {
    this.debugService.log(phaseEvent);
  }

  public PhaseId getPhaseId()
  {
    return PhaseId.ANY_PHASE;
  }
}
```

Why @JsfPhaseListener instead of @PhaseListener?

It's easier to use the annotation because there isn't an overlap with the name of the interface. So it isn't required to use the fully qualified name for one of them.

### Faces-Request-Listener

Sometimes it's essential to perform logic directly after the `FacesContext` started and/or directly before the `FacesContext` gets destroyed. In such a case CODI provides the annotations `@BeforeFacesRequest` and `@AfterFacesRequest`.

Example:

**Observer method with @BeforeFacesRequest**

```
protected void initFacesRequest(@Observes @BeforeFacesRequest FacesContext facesContext)
{
  //...
}
```

## Producers

CODI offers a bunch of producers for JSF artifacts.

Example:

**Injection of the current FacesContext**

```
@Inject
private FacesContext facesContext;
```

## JsfLifecyclePhaseInformation

Helper for detecting the current phase of the JSF request lifecycle.

Example:

**Detecting the current phase**

```
public class MyBean
{
    @Inject
    private JsfLifecyclePhaseInformation phaseInformation;

    public void execute()
    {
        if (this.phaseInformation.isProcessValidationsPhase() || this.phaseInformation.
isUpdateModelValuesPhase())
        {
            //...
        }
    }
}
```

## Type-safe View Config

In some projects users are using enums which allow e.g. a type-safe navigation. CODI provides a mechanism which goes beyond that. You can use classes which hosts a bunch of meta-data. One use-case is to use these special classes for type-safe navigation. Beyond that CODI allows to provide further meta-data e.g. page-beans which are loaded before the rendering process starts, type-safe security and it's planned to add further features to this mechanism.

The following example shows a simple view-config.

**View config for /home.xhtml**

```
@Page
public final class Home implements ViewConfig
{
}
```

This mechanism works due to the naming convention. Instead of the convention it's possible to specify the name of the page manually. This feature is described at the end of this section.

Important hint

Use classes for your pages and for everything else interfaces!

If you would like to group pages (you will learn some reasons for that later on), you can nest the classes.

**Grouping pages**

```
public interface Wizard extends ViewConfig
{
    @Page
    public final class Page1 implements Wizard
    {
    }

    @Page
    public final class Page2 implements Wizard
    {
    }
}
```

Such a grouping allows to reduce the number of class files in your workspace. Furthermore modern IDEs allow to show the logical hierarchy out-of-the-box (in Intellij it's called "Type Hierarchy").

Hint

At `@Page(basePath = "")` we have to override the default with an empty string to have a virtual view config which doesn't affect the final name.

## Organizing your pages

View configs allow to reflect your folder structure in the meta-classes.

**Grouping pages and folder structures**

```
public interface Wizard extends ViewConfig
{
    @Page
    public final class Step1 implements Wizard
    {
    }

    @Page
    public final class Step2 implements Wizard
    {
    }
}
```

... leads to the following view-ids: `/wizard/step1.xhtml` and `/wizard/step2.xhtml`.
**That means - if you rename the folder wizard, you just have to rename a single class and everything is in sync again.**

Hint

While the nested classes lead to the final path of the file, the inheritance allows to configure a bunch of pages.

**Grouping pages and folder structures**

```
@Page(navigation = REDIRECT)
public interface Wizard extends ViewConfig
{
    @Page
    public final class Step1 implements Wizard
    {
    }

    @Page
    public final class Step2 implements Wizard
    {
    }
}
```

... leads to redirects as soon as navigation is triggered and `Step1` or `Step2` are the targets. You can centralize all available configs. Some of them can be replaced at a more concrete level (in the example above it would be possible to override the redirect mode e.g. for `Step1` or `Step2`) whereas others will be aggregated (like `AccessDecisionVoter`s).

Hint

Besides this naming convention you can use `basePath` to force a different name. Furthermore, `@Page` allows to adjust the navigation mode (forward (= default) vs. redirct) and the file extension (default: xhtml) as well as the name of the page itself.

## Custom view meta-data

It's possible to use custom annotations in view-config classes. Just annotate the custom annotation with `@ViewMetaData`.

**Implementation of a custom view meta-data annotation**

```
@Target({TYPE})
@Retention(RUNTIME)
@Documented

@ViewMetaData
public @interface ViewMode
{
    //...
}
```

**Usage of a custom view meta-data annotation**

```
@Page
@ViewMode
public final class Home implements ViewConfig
{
}
```

Optionally you can specify if a nested class (e.g. `UseCase1.Step1`) overrides the meta-data. That means there will be just one instance per such a custom annotation per page. Per default all annotations are collected independent of the already found types and independent of the instance count of an annotation type.

**Implementation of a custom view meta-data annotation which allows to override meta-data**

```
@Target({TYPE})
@Retention(RUNTIME)
@Documented

@ViewMetaData(override = true)
public @interface PageIcon
{
    //...
}
```

**Override view meta-data annotation**

```
@PageIcon(...)
public interface UseCase1 extends ViewConfig
{
    @Page
    //inherits the page-icon of the use-case
    public final class Step1 implements Wizard
    {
    }

    @Page
    @PageIcon(...) //overrides the page-icon of the use-case
    public final class Step2 implements Wizard
    {
    }
}
```

**Resolve view meta-data annotations**

```
@Inject
private ViewConfigResolver viewConfigResolver;

this.viewConfigResolver.getViewConfigDescriptor(UseCase1.Step2.class).getMetaData();

//or something like:
this.viewConfigResolver.getViewConfigDescriptor(this.facesContext.getViewRoot().getViewId()).getMetaData();
```

## Page ranges

Type-safe view configs also allow to easily specify the valid page range.

**Specified page range**

```
public interface PublicView extends ViewConfig {}

@Page
public class ReportPage implements PublicView {}

//...

@Secured(LoginAccessDecisionVoter.class)
public interface Internal extends ViewConfig
{
    @Page
    public class ReportPage implements Internal {}
}

//...

public Class<? extends Internal> showInternalReport()
{
    //...
    return Internal.ReportPage.class;
}

//...

public Class<? extends PublicView> showPublicReport()
{
    //...
    return ReportPage.class;
}
```

In this example it's easy to ensure that the correct target page is used. It isn't possible to expose the internal report (page) as public report by accident (due to a wrong import or a bad refactoring).

## Type-safe Navigation

In the previous section you have learned some details about the view-config mechanism provided by CODI. You can use these meta-classes for the navigation.

Example:

**Action method with type-safe navigation**

```
public Class<? extends ViewConfig> navigateToHomeScreen()
{
    return Home.class;
}
```

Hint

Some EL implementations like JUEL check the allowed return type explicitly. In combination with early implementations of Facelets you might see an exception which tells that action methods have to return strings. In such a case you can use `Home.class.getName()`.

Support of getNavigationCases

The new API of JSF 2 `ConfigurableNavigationHandler#getNavigationCases` doesn't support implicit navigation. That's not the case with type-safe navigation. CODI combines the best of both. You don't need navigation rules and you will get the information with calling `ConfigurableNavigationHandler#getNavigationCases` if you need to query these information. (If you don't need it, you can deactivate it via the type-safe CODI config.)

## Navigation via GET Requests (JSF 2.0+) (since v1.0.2)

Since JSF 2.0 it's possible to use GET requests for navigating between pages. With MyFaces CODI you can use type-safe navigation also in combination with GET-Requests.

**JSF-Navigation via GET requests**

```
<!-- Std. approach with JSF 2.0: -->

<h:button value="..." outcome="/pages/myPage.xhtml">
    <f:param name="param1" value="v1"/>
</h:button>

<!-- CODI allows to use: -->

<h:button value="..." outcome="#{myController.myPage}">
    <f:param name="param1" value="v1"/>
</h:button>

<!-- or if needed (not recommended, because it isn't type-safe): -->
<h:button value="..." outcome="class myPackage.Pages.MyPage">
    <f:param name="param1" value="v1"/>
</h:button>
```

**The corresponding code in MyController**

```
public Class<? extends ViewConfig> getMyPage()
{
    return Pages.MyPage.class;
}
```

## PreViewConfigNavigateEvent

In case of type-safe navigation it's possible to observe navigations which have a view-config for the source-view as well as the target-view.

**Observe type-safe navigation event**

```
protected void onViewConfigNavigation(@Observes PreViewConfigNavigateEvent navigateEvent)
{
    //...
}
```

Furthermore, it's possible to change the navigation target.

**Observe type-safe navigation event**

```
protected void onViewConfigNavigation(@Observes PreViewConfigNavigateEvent navigateEvent)
{
    if(Wizard.Page1.class.equals(navigateEvent.getFromView()) &&
        !Wizard.Page2.class.equals(navigateEvent.getToView()))
    {
        navigateEvent.navigateTo(DemoPages.HelloMyFacesCodi2.class);
    }
}
```

## View-Configs and Parameter (since v1.0.2)

Sometimes it's needed to add parameters. If you are using e.g. the implicit navigation feature of JSF 2.0+ you would just add them to the string you are using for the navigation. In case of type-safe view-configs it isn't possible to add such parameters directly. To add such parameters, it's required to use the `@PageParameter` annotation. It's possible to annotate action-methods **or** view-config classes which represent a page with this annotation. So it's possible to enforce parameters for all JSF based navigations to a view or to add parameters just in case of special actions. Furthermore, it's possible to add multiple parameters with `@PageParameter.List`. The usage for action methods is the same as the usage for view-configs. The following example shows a simple parameter.

```
@Page
@PageParameter(key="customParam", value="#{customBean.value1}")
public class Index implements ViewConfig {}
```

## Type-safe Security

**Action method with type-safe navigation**

```
@Page
@Secured(OrderVoter.class)
public final class OrderWizard implements ViewConfig
{
}
```

**Example for an AccessDecisionVoter**

```
@ApplicationScoped
public class OrderVoter extends AbstractAccessDecisionVoter
{
    @Inject
    private UserService userService;

    @Inject
    private User currentUser;

    public void checkPermission(InvocationContext invocationContext, Set<SecurityViolation> violations)
    {
        if(!this.userService.isActiveUser(this.currentUser))
        {
            violations.add(newSecurityViolation("{inactive_user_violation}"));
        }
    }
}
```

The message-key of the previous example will be passed to the `MessageContext` with the `Jsf` qualifier. You can also use a hardcoded inline message. If you would like to use a different `MessageContext` you can just inject it (see the following example).

**Example for an AccessDecisionVoter with a custom MessageContext**

```
@ApplicationScoped
public class OrderVoter extends AbstractAccessDecisionVoter
{
    @Inject
    private UserService userService;

    @Inject
    private User currentUser;

    @Inject
    @Custom
    private MessageContext messageContext;

    public void checkPermission(InvocationContext invocationContext, Set<SecurityViolation> violations)
    {
        if(!this.userService.isActiveUser(this.currentUser))
        {
            String reason = this.messageContext.message().text("{inactive_user_violation}").toText();
            violations.add(newSecurityViolation(reason));
        }
    }
}
```

## Customizing violation-handling (since v1.0.2)

Per default the created violation message gets added as faces-message. However, sometimes users don't have to see such messages (e.g. in case of autom. navigation to a login page). For such cases it's possible to introduce a custom `SecurityViolationHandler`. As soon as a project contains a bean which implements this interface, the bean will be used for handling the violations (instead of adding them autom. as faces-message).

## (Security) Error pages

The following example shows how to create a default error page. It's just allowed to provide one default error page per application.
Instead of implementing `ViewConfig` it's required to implement the `DefaultErrorView` interface.

**Default error page**

```
@Page
public final class Login extends DefaultErrorView
{
}
```

Hint

`@Secured` allows to override the default error page for a specific page or a group of pages.

If there isn't an error page, CODI will throw an `AccessDeniedException`.

# Page-Beans

It's a common pattern in JSF applications to create beans which act as page-controllers (aka Page-Beans). Such beans are mapped to 1-n views. Usually there is just one concrete implementation.

CODI allows to specify the page-bean as optional meta-data via the view-config mechanism.
You can use this approach to load the page-bean before the rendering process starts. So the post-construct method (= methods annotated with `@PostConstruct`) will be invoked if it is needed. Furthermore, it's possible to use `@BeforePhase(...)` and `@AfterPhase(...)` without observer syntax of CDI.

**View-config with Page-Bean**

```
@Page
@PageBean(LoginPage.class)
public final class Login implements ViewConfig
{
}
```

**Page-Bean**

```
//...
public final class LoginPage implements Serializable
{
  @PostConstruct
  protected void initBean()
  {
    //...
  }

  @AfterPhase(INVOKE_APPLICATION)
  protected void postPageAction()
  {
    //...
  }

  @BeforePhase(RENDER_RESPONSE)
  protected void preRenderView()
  {
    //...
  }
}
```

... you can use `@BeforePhase` and `@AfterPhase` in the same way like the Phase-Listener Methods described above (just without the need of the `PhaseEvent`).

**Page-Bean with view-controller annotations**

```
//...
public final class LoginPage implements Serializable
{
  @PostConstruct
  protected void initBean()
  {
    //...
  }

  @InitView
  protected void initView()
  {
    //...
  }

  @PrePageAction
  protected void prePageAction()
  {
    //...
  }

  @PreRenderView
  protected void preRenderView()
  {
    //...
  }
}
```

`@PostConstruct` in a view-controller bean this lifecycle-callback is invoked before the rendering phase at latest. However, it's just called at the first time the instance of the bean is used (as view-controller bean - depending on the scope of the bean). If it's required to process logic every time before the page gets rendered, it's possible to use `@PreRenderView`.

`@InitView` is invoked after a view was created.

Example:

```
viewA created -> @InitView callback for viewA called
viewB created -> @InitView callback for viewB called
viewB created -> logic already initilized -> no callback
viewA created -> @InitView callback for viewA called
viewB created -> @InitView callback for viewB called
```

-> Such methods get called after the corresponding view was created.
The evaluation happens after every request-lifecycle-phase to avoid that other features or frameworks lead to unexpected calls if they have to create views temporarily (e.g. security frameworks).

`@PrePageAction` is invoked directly before the action-method. In comparison to `@BeforePhase(INVOKE_APPLICATION)`, `@PrePageAction` also works for immediate actions. If you have to use a bean like the `RequestTypeResolver` just inject it into the bean and use it.

`@PreRenderView` is invoked before the page gets rendered. It allows e.g. to load data and change the target-view in case of an unexpected error.

`@PostRenderView` is invoked after the rendering process and allows to do special clean-up.

Hint

For normal pre-render view logic you can use phase-listener methods in combination with `@View`.
Attention: the performance depends on the interceptor-performance of your CDI implementation.

Sometimes it's required to use multiple page controllers for a page (e.g. in case of two very different parts in the page which should be handled by different view-controllers). Such a use-case isn't very common, however, via `@PageBean.List` it's possible to attach multiple pages-beans to a view-config.

**View-config with multiple Page-Beans**

```
@Page
@PageBean.List({
  @PageBean(Bean1.class),
  @PageBean(Bean2.class)
})
public final class UseCase1 implements ViewConfig
{
}
```

# Alternative to @PageBean (since v0.9.1)

If you don't like the idea of a central type-safe config but you would like to use type-safe navigation, it's possible to use `@View` as alternative to `@PageBean` directly in the class of the bean.

the `@PageBean` example above would look like:

**View controller example without @PageBean**

```
@Page
public final class UseCase1 implements ViewConfig
{
}

@Model
@View(UseCase1.class)
public class Bean1 implements ViewConfig
{
    @PreRenderView
    protected void preRenderView()
    {
        //...
    }
}


@Model
@View(UseCase1.class)
public class Bean2 implements ViewConfig
{
    @PreRenderView
    protected void preRenderView()
    {
        //...
    }
}
```

# Inline-View-Configs (since v0.9.3)

Esp. at the beginning the full blown approach which is available with type-safe View-Configs might look a bit heavy. Several users see the need for the approach later on as soon as their applications become larger.
That was the reason for providing an approach which is easier to use at the beginning. It allows to provide the View-Config at the page-bean implementations. So it's called Inline-View-Config.

However, there are clear restrictions e.g. for using it in combination with type-safe navigation, one page-bean per page is required. That means if you are using a page-bean e.g. per wizard, you have to switch to the full View-Config approach or you will need really one bean per page.

**This approach uses the package structure for creating the View-IDs.**
Due to this approach a marker is needed which marks the root path. Since there are quite different approaches to structure the folders for your pages, there are several styles for using this marker.

If you have a dedicated root-folder for all your pages (and sub-folders) you can reflect it in your package structure. The following example shows a class marked with `@InlineViewConfigRoot` in the package *.pages. So the root folder has to have the name 'pages'. Every sub-package will be mapped to a sub-folder. In case of Inline-View-Configs the page-bean has to implement the `ViewConfig` in-/directly and has to be annotated with `@Page`. You can use the same features of the normal View-Config including type-safe navigation, lifecycle callback annotations,...

**Inline-View-Config example 1**

```
package my.pkg.pages;

@InlineViewConfigRoot
public final class RootMarker
{
}

package my.pkg.pages.registration;
//...

@Named
@RequestScoped
@Page
public class RegistrationStep1 implements ViewConfig
{
  public Class<? extends ViewConfig> confirm()
  {
    //...
    return RegistrationStep2Page.class;
  }
}

//will be interpreted as /pages/registration/registrationStep1.xhtml
```

Esp. at the beginning you maybe don't have a folder for your pages. That means if you start with /page1.xhtml instead of /pages/page1.xhtml, you have to specify it explicitely with `/*`.

Furthermore, it's possible to specify a so called `pageBeanPostfix` for allowing to use a name convention for your pages beans which won't be reflected by the xhtml file name.

**Inline-View-Config example 2**

```
package my.pkg.pages;

@InlineViewConfigRoot(basePath = "/*", pageBeanPostfix = "Controller")
public final class RootMarker
{
}

package my.pkg.pages.registration;
//...

@Named
@RequestScoped
@Page
public class RegistrationStep3Controller implements ViewConfig
{
  //...
}

//will be interpreted as /registration/registrationStep3.xhtml
```

If you have a fine grained package structure which isn't reflected in the folder-structure of your pages (or a different name has to be used), it's possible to specify a `basePath`. Without a `*` at the end, all sub-packages are ignored.

**Inline-View-Config example 3**

```
package my.pkg.pages;

@InlineViewConfigRoot(basePath = "/pages/", pageBeanPostfix = "Page")
public final class RootMarker
{
}

package my.pkg.pages.registration;
//...

@Named
@RequestScoped
@Page
public class RegistrationStep2Page implements ViewConfig
{
  //...
}

//will be interpreted as /pages/registrationStep2.xhtml
```

Compared to the previous example the next example shows a custom `basePath` and all sub-packages will be mapped to sub-folders.

**Inline-View-Config example 4**

```
package my.pkg.pages;

@InlineViewConfigRoot(basePath = "/views/*")
public final class RootMarker
{
}

package my.pkg.pages.registration;
//...

@Named
@RequestScoped
@Page
public class RegistrationStep4 implements ViewConfig
{
  //...
}

//will be interpreted as /views/registration/registrationStep4.xhtml
```