SAP NetWeaver CE 7.1

Web Dynpro Java - Reference

Web Dynpro Java Controller and Interface Concept

Document Version 1.00 – Juli 2007

SAP AG
Neurottstraße 16
69190 Walldorf
Germany
T +49/18 05/34 34 24
F +49/18 05/34 34 20
www.sap.com

Disclaimer

Some components of this product are based on Java™. Any code change in these components may cause unpredictable and severe malfunctions and is therefore expressively prohibited, as is any decompilation of these components.

Any Java™ Source Code delivered with this product is only to be used by SAP's Support Services and may not be modified or altered in any way.

Documentation on SAP Service Marketplace

You can find this documentation at
`service.sap.com/instguidesNW04`

Typographic Conventions

Type Style	Represents
Example Text	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options.
	Cross-references to other documentation.
Example text	Emphasized words or phrases in body text, graphic titles, and table titles.
EXAMPLE TEXT	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
`Example text`	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
`Example text`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
`EXAMPLE TEXT`	Keys on the keyboard, for example, F2 or ENTER.

Icons

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Contents

Document Version 1.00

Controllers and their Interfaces

Primarily Web Dynpro follows principles of declarative, model-based UI development which minimize coding and maximize design. Nevertheless the implementation of a Web Dynpro controller class demands a good understanding of the underlying controller class and interface architecture. This architecture is based on extensive code-generation which highly simplifies your application coding.

To implement Web Dynpro controller code an application developer must know about the following issues:

- How do the controller declarations, which are done by the application developer, affect the generated controller classes and interfaces?
- Which interfaces can be used in a Web Dynpro controller class to invoke inbuilt, or generated controller functions?
- Which public controller interfaces can be used to call one controller from another.
- Which controller interfaces, implemented by the Web Dynpro Java Runtime, can be used to invoke *generic* controller functions?

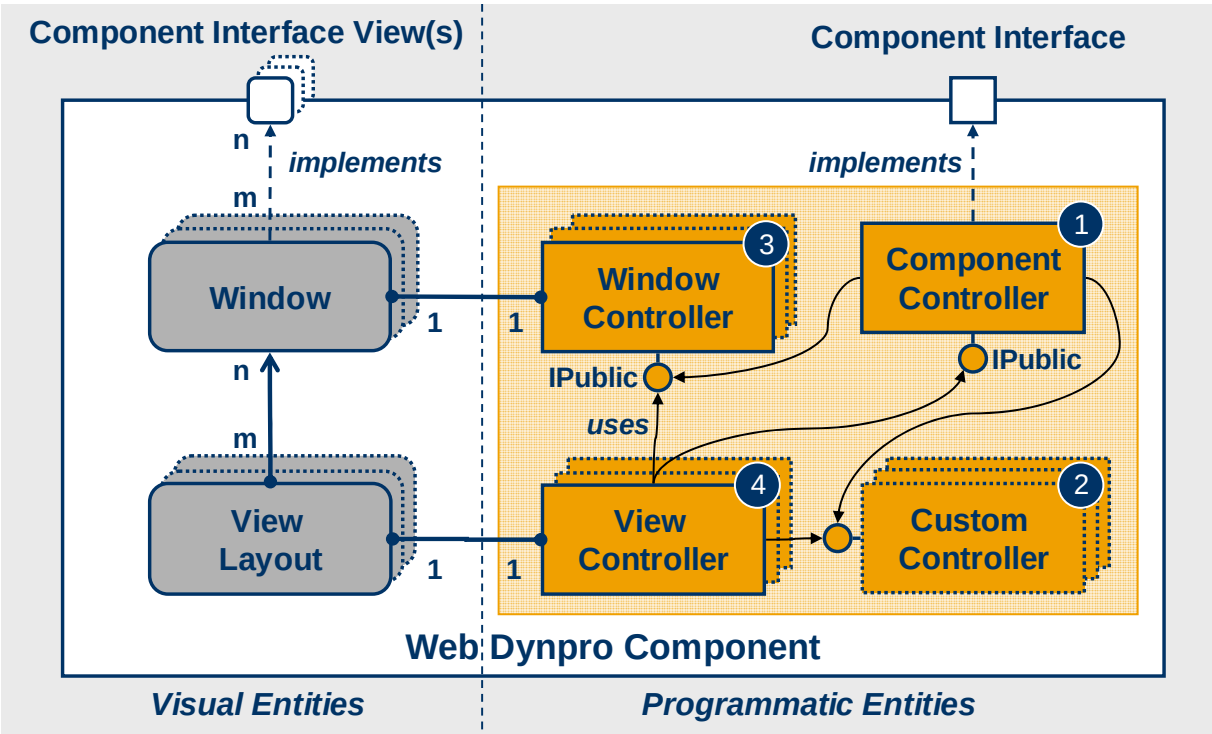
This section comprises a detailed description of the Web Dynpro controller and interface concept and answers the above questions. It is separated into the following three sections:

1. **Concepts**: Two sections on the controller concept [Page 1] and the controller interface concept [Page 5] explain the technical principles behind all Web Dynpro controllers and behind the layered controller interface architecture (`IPrivate`, `IPublic` and *generic* controller interfaces).
2. **Example**: Based on a concrete sample component [Page 14] with three controllers you learn how different declarations affect the generated controller classes and interfaces. Various programming tasks are described and solved by invoking generated or generic controller interfaces.
3. **Reference**: The Controller Class and Interface Reference [Page 21] comprises a detailed description of all generated Web Dynpro controllers and interfaces. It documents how the specific controller definitions like *controller usages, public methods, events, event handlers* or *navigation plugs* affect the automatically generated Web Dynpro controller classes and interfaces. In contrast to the *generic* Web Dynpro controller APIs (`IWDController`, `IWDComponent`, `IWDViewController` and `IWDWindowController`) the generated classes and interfaces cannot generally be documented with *JavaDoc*. This gap is filled by the Controller Class and Interface Reference.

This section is focusing on a single Web Dynpro component and does not cover scenarios with multiple components. Therefore the controller-specific aspects of implementing local and standalone component interface definitions are not described in this section.

Controller Concept

Controller Classes inside a Web Dynpro Component

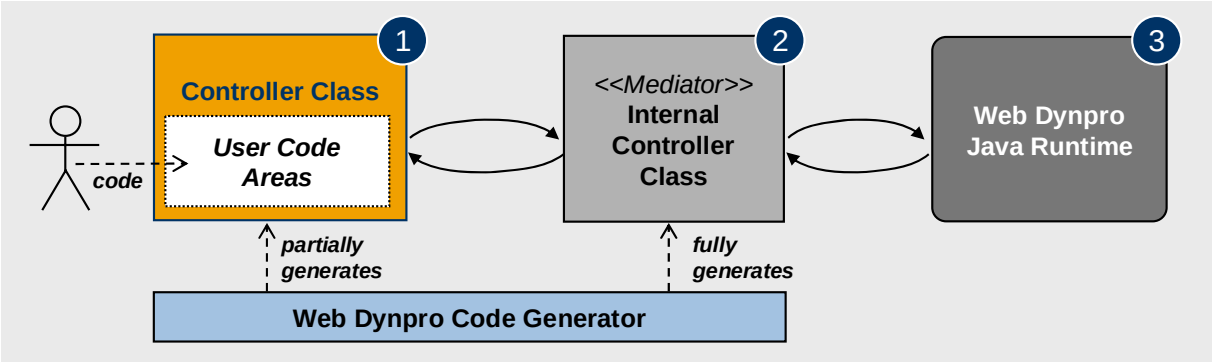


The following four controller types can be contained in a Web Dynpro Component (see figure above):

4. *Component Controller*: Every component contains a single component controller by default. It has no visual interface and can be seen as a component's master controller.
5. *Custom Controller*: A custom controller is an optional controller (0..n) inside a component having no visual interface. It is there for implementing specific application logic concerns (value help listeners, configuration) inside a separate controller and outside the component controller.
6. *Window Controller*: A window controller implements the presentation logic of its related window (0..n, visual interface). The presentation logic mainly deals with handling or firing navigation events (inbound and outbound plugs, startup and exit plugs, suspend and resume plugs).
   A window controller can additionally implement one or many component interface view controllers defined in local or standalone component interface definitions.
7. *View Controller:* A view controller implements the presentation logic of its related view (0..n, visual interface). The presentation logic of a view mainly deals withs firing navigation events (inbound and outbound plugs) or dynamically modifying view layouts in `wdDoModifyView()` and handles action events, validates user input or simply delegates method calls to window-, custom- or component controller.

Controller Implementation

The complete implementation of a Web Dynpro controller is distributed across three different places (see next figure):



8. **Controller class**: Partially generated class comprising the application defined (*self-implemented*) logic or source code within *user code areas*. This is the only controller class an application developer implements source code.
9. **Internal controller class**: Every Web Dynpro controller class is associated with an *internal controller class* named `Internal<controller name>.java`. This fully generated class acts as a *mediator* between the controller class and the Web Dynpro Java Runtime or other controller classes. It does not comprise any code implemented by the application developer but delegates to it.
10. **Web Dynpro Java Runtime implementation**: The Web Dynpro Java Runtime implements generic controller functionality which can by accessed via special *IWD<Controller>*-APIs

Controller Class with User Code Areas

For every listed controller type a corresponding Java class file `<controller name>.java` is created by the Web Dynpro Tools (figure above, point 1). Within the *User Code Areas* of this controller class the application developer can implement the required custom code. All code and JavaDoc areas are delimited with the lines `//@@begin` and `//@@end`:

Controller Class with user code

```
  //@@begin javadoc:wdDoInit()
  /** Hook method called to initialize controller. */
  //@@end
  public void wdDoInit()
  {
    //@@begin wdDoInit()
    //@@end
  }

  //@@begin others
  //@@end
```

Additional Custom Code Block

At the end of a controller class file the `//@@begin others` - `//@@end` code section can be used for any Java code that is not to be visible to other controllers/views or that contains constructs currently not supported directly by Web Dynpro such as inner classes, private methods or member variables etc.

The content of this section is in no way managed or controlled by the Web Dynpro Designtime or the Web Dynpro Runtime.

It is important to understand, that the full functionality of a Web Dynpro controller is only partially implemented within the visible controller class comprising the application defined logic (*User Code Areas*). Many additional functions are implemented within the internal controller class and within the Web Dynpro Java Runtime itself. You can access these functions in your custom code via the special shortcut variables **`wdThis`** and **`wdControllerAPI`**.

Using other Java Classes and Interfaces

You can easily use additional Java classes or interfaces within a Web Dynpro controller class. These Java classes are not generated by the Web Dynpro Generation Framework and must completely be implemented by the application developer:

- **Java Classes stored in the same Web Dynpro Development Component (or Project)**: Custom Java classes or interface files must be stored under the folder **`src/packages`**. This source folder is on the build path of every Web Dynpro DC/Project by default.
- **Java Classes stored in another Web Dynpro Development Component**: Custom Java classes and interfaces can be exposed within the public part of another DC. By adding a corresponding DC dependency to your Web Dynpro DC you can use these Java classes within all controller classes.

See also

- Controller Interface Concept [Page 5]
- Example: Implementing a Component with its Controllers and Interfaces [Page 14]
- Controller Class and Interface Reference [Page 21]

Controller Interface Concept

Introduction

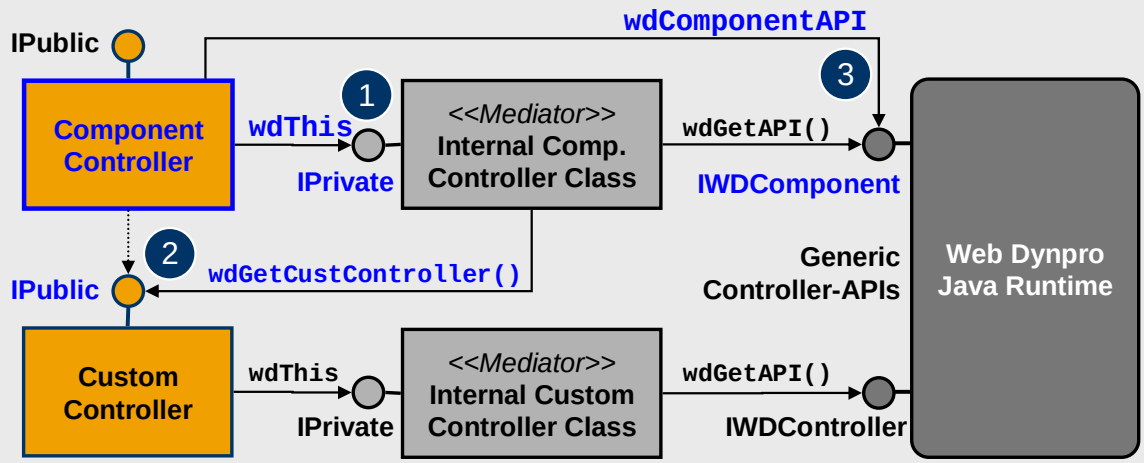
The Web Dynpro controller concept which is realized within a component consequently adheres to the *separation of concern design principle*. This means that the full logic and computation of a Web Dynpro controller is distributed among several classes:

- **Self-implemented controller classes:** implement custom code. User code areas contain the *self-implemented* controller logic.
  - Application logic is implemented within *component* and *custom* controllers having no visual interface.
  - Presentation logic is implemented within *view* and *window* controllers having a visual interface.
- **Internal controller classes:** act as *mediator* classes between the Web Dynpro Java Runtime and the controller classes implementing custom code delegate additional functions like firing events, managing navigation or messages to the Web Dynpro Java Runtime. Its implementation gets automatically adapted to the related controller declarations like public methods, event subscriptions, events or the definition of view layouts.
- **Generic controller classes**: The Web Dynpro Java Runtime implements several *generic controller APIs* like `IWDController`, `IWDComponent`, `IWDViewController` and `IWDWindowController`. The generic controller APIs are required to access controller metadata (*IWD…Info*-APIs) or to modify controllers at runtime (e.g. dynamic event subscription). Furthermore they are needed to access some generic service-APIs like `IWDMessageManager`, `IWDWindowManager` or `IWDTextAccessor` in your controller code.

To implement custom code in a Web Dynpro controller class an application developer must often access *externally* implemented controller classes providing additional services. These services are exposed to the self-implemented controller class as different types of fully generated *interfaces* called *Web Dynpro controller interfaces*.

Sample Scenario - Utilizing Controller Interfaces

The following diagram illustrates the roles of different Web Dynpro controller interfaces which are used to access externally implemented controller code. The related technical principles will successively be explained within the next sections.



The application developer implements code within the Component controller and the Custom controller classes. He now wants to access externally implemented custom controller code within the component controller.

- **IPrivate-API**: This interface exposes logic of the internal controller class to the *self-implemented* controller class. It can be accessed with the private member variable `wdThis` of type `IPrivate` in every Web Dynpro controller (1). The component controller can only use the `IPrivate`-API of its own internal controller class. It is not visible outside.
- **IPublic-API**: The `IPublic`-interface exposes logic of the custom controller class to other controllers in the same component. To use this public controller interface within the component controller a *usage relation* to the custom controller must be defined first. Afterwards the `IPrivate`-API of the component controller gets extended with a new method `wdGet<custom controller name>Controller()` returning the `IPublic`-API of the custom controller class. It can then be accessed with `wdThis.wdGet< custom controller name >Controller()` (2).
- **Generic Controller API**: To access the generic controller interface `IWDComponent` within the component controller clase the shortcut variable `wdComponent = wdThis.wdGetAPI()` can be used (3).

* The names `IPrivate` - and `IPublic`-API are the general terms for all generated controller interfaces with the prefixes `IPrivate` and `IPublic`. The qualified names of these interfaces are `IPrivate<controller name>`.

Here you can see how the internal controller class acts a mediator between the self-implemented controller class and the Web Dynpro Java Runtime. The controller class sends messages to the internal controller mediator class when needed and the mediator passes them on to other Web Dynpro controller classes or to the Web Dynpro Java Runtime itself that need to be informed.

See also

- IPrivate Interface - Accessing own Controller Functions [Page 7]
- IPublic Interface - Accessing other Controller Functions [Page 8]
- Shortcut Variables - Accessing Generic Controller Interfaces [Page 11]
- Example: Implementing a Component with its Controllers and Interfaces [Page 14]
- Controller Class and Interface Reference [Page 21]

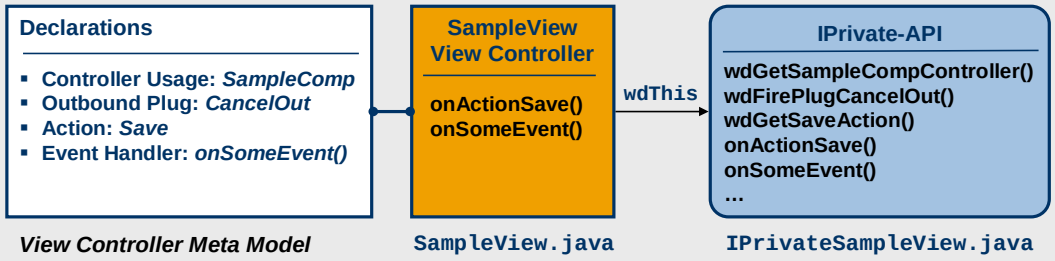
IPrivate Interface - Accessing own Controller Functions

Based on its declarative, model-driven development concept Web Dynpro automatically adapts the generated controller classes and interfaces to the user (application developer) definitions.

Whereas the controller class itself only implements custom code, the internal controller class implements additional generic code. The `IPrivate`-API is the glue between these two classes. It exposes the internal controller class to the self-implemented controller class so that the generated *internal* controller class and its related interfaces can be invoked by the application developer.

The `IPrivate`-API is generated for every Web Dynpro controller type.

The next diagram illustrates the role of the `IPrivate`-API within a practical example:



* class and interface diagram only comprises added methods based on declaration

Within the view *SampleView* the application developer makes some declarations like the usage of the component controller, the outbound plug *CancelOut*, the action *Save* or the event handler `onSomeEvent()`.

After this modification of the view's meta model data the Web Dynpro Code Generator re-generates and adapts the view controller class and its `IPrivate` interface:

Action handler and event handler methods are added to the controller class and to the `IPrivate`-API of the internal controller class.

The `IPrivate`-API is additionally extended by the methods:

- `wdGetSampleCompController()` to access the `IPublic`-API of the used component controller,
- `wdFirePlugCancelOut()` to fire the navigation event of outbound plug *CancelOut*,
- `wdGetSaveAction()` to access the object representing action *Save*.

To access the fully generated `IPrivate`-API within the self-implemented controller class the shortcut variable `wdThis` can be used. This private member variable is of type `IPrivateSampleView`.

Besides the declaration-based methods the `IPrivate` interface also exposes the generic methods `wdGetAPI()` and `wdGetContext()` to the component controller. The technical details on these functions will be described in the next section. There you will see, that the `IPrivate` interface *extends* the `IPublic` interface, if that one exists. This is the case for all *non-view* controllers like the window, component and custom controllers.

See also

- Example: Implementing a Component with its Controllers and Interfaces [Page 14]

IPublic Interface - Accessing other Controller Functions

The next Web Dynpro controller interface type we consider is the `IPublic` interface. It is exposed by every *non-view* Web Dynpro controller and can be invoked by all controllers in the same component. With this interface it is easily possible to share logic across several controllers so that no code redundancies occur.

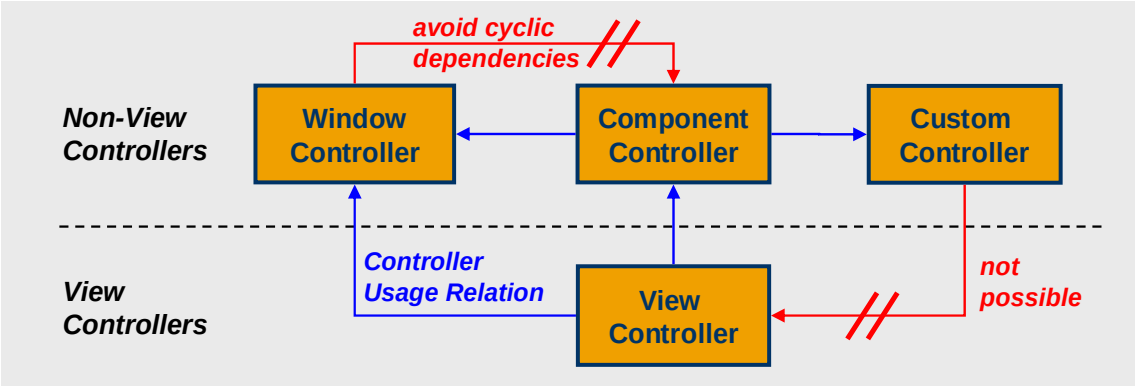
Like the `IPrivate` interface the `IPublic` interface gets fully generated by the Web Dynpro Tools based on the controller declarations made by the application developer.

A simple example is the definition of a new public method `executeSevice()` in the component controller. This method is automatically added to the component controller's `IPublic`-API so that it can be invoked within a view controller's action event handler:
`wdThis.wdGetSampleCompController().executeService().`

Before going into the details of the `IPublic`-API we first consider the *controller visibility concept* the different Web Dynpro controllers adhere to.

Controller Usage Relations and Visibility

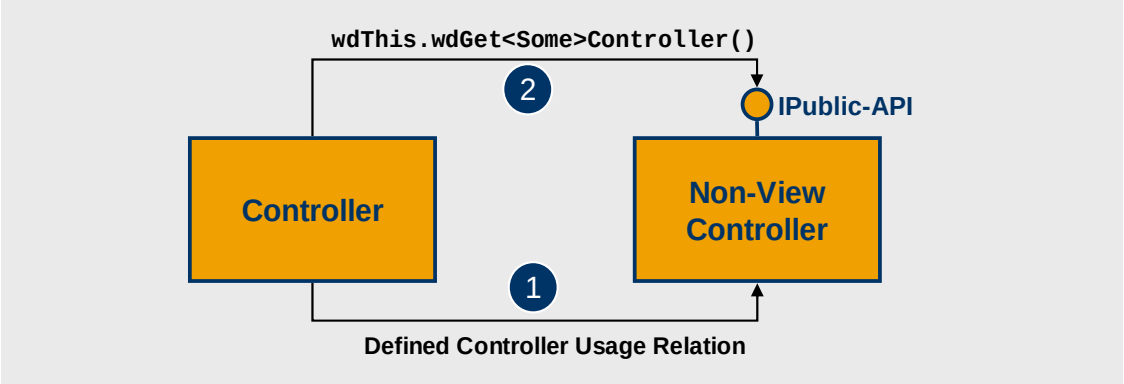
In case a Web Dynpro controller A requires a function implemented in another controller B a *controller usage relation* must explicitly be defined first within the Web Dynpro Tools.



When defining controller usage relations the following aspects must be considered.

- **By default no controller usage relations are defined**: By default every Web Dynpro controller is *not* related with any other controller in the same component which means that no controller is *visible* for other controllers first. Whith this strategy the Web Dynpro controller model minimizes the dependencies between separate controller classes. It lies in the responsibility of the application developer to explicitly define the required controller dependencies *or controller usage relations.*
- **View controllers cannot be used**: With respect to the other controllers in a component, a view controller is always considered to be a *consumer* of data and a *caller* of logic; therefore it can never be nominated as a required or used controller by any other Web Dynpro controller. As a consequence a view controller does not expose an `IPublic` interface to other controllers.
- **Cyclic controller usages should be avoided**: The definition of a cyclic controller usage relation – like a component controller uses a window controller and vice versa – should be avoided because this may lead to an unexpected behavior at runtime.

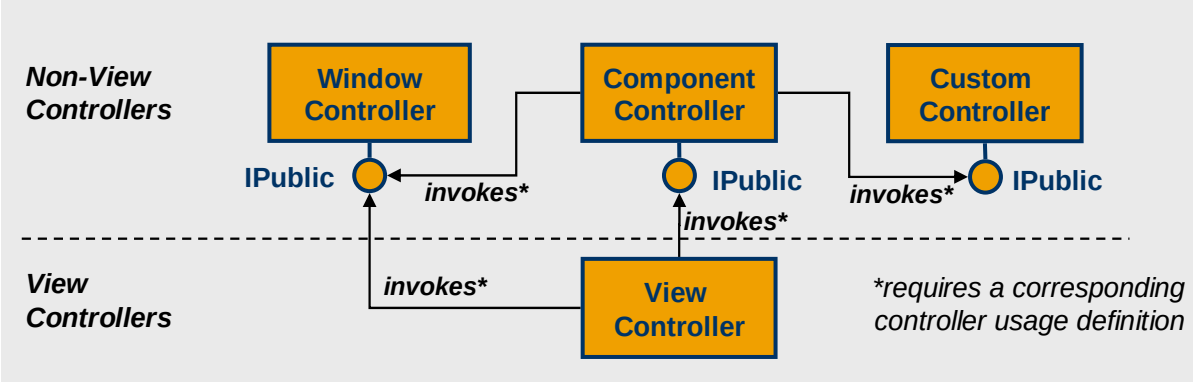
Invoking the IPublic-API of a Used Controller



To invoke the `IPublic` interface exposed by a non-view controller a usage relation to this controller must be defined first (figure above, point 1).

With this controller usage definition the `IPrivate`-API of the consuming controller is automatically extended by the method `wdGet<used controller name>Controller()` which returns the `IPublic` interface of the used controller (2).

The next diagram illustrates a typical controller scenarion in which the `IPublic` interfaces of non-view controllers are invoked by other controllers in the same component.



As view controllers cannot be used by other controllers they principally do expose an `IPublic`-API to other controllers.

IPrivate extends IPublic

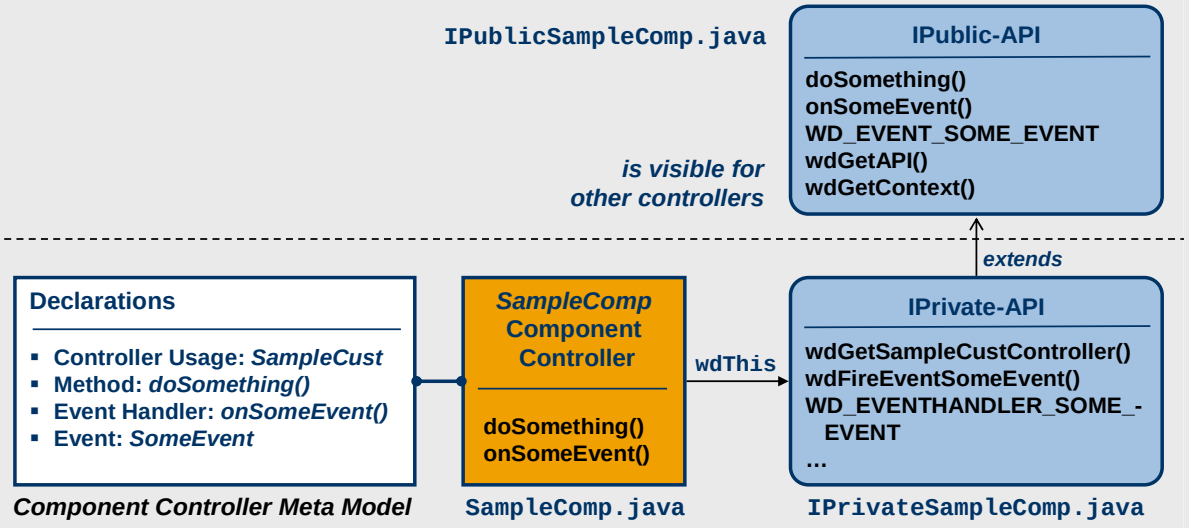
An important relationship between the two generated Web Dynpro controller interface types is, that the `IPrivate` extends the `IPublic` interface. This means, that all methods and constants the `IPublic`-API exposes to other controllers can be accessed within the controller class itself by invoking the `IPrivate`-API.

Depending on which declaration you make in a controller the `IPublic` or the `IPrivate` API is adapted correspondingly. Some definitions are exposed to other controllers within the IPublic interface, others do only affect the IPrivate interface so that they are not visible outside:

- **IPublic-API**: public controller methods, event handlers, event id of type `IWDEventId` for the dynamic event subscription in another controller.
- **IPrivate-API**: accessor method to invoke the `IPublic`-API of a used controller, accessor method to invoke the `IWDComponentUsage`-API for a used component, fire event methods, event handler id of type `IWDEventHandlerId` for the dynamic event subscription in the same controller.

Based on these rules it is clear why a declared event cannot directly be fired within another controller. As the method `wdFireEvent<event name>()` is added to the `IPrivate`- but not to the `IPublic`-API you must explicitly define a public method which can be invoked outside and which itself calls the fire method.

Example



Within the component controller *SampleComp* the application developer makes some declarations like the usage of the custom controller *SampleCust*, the event *SomeEvent,* the event handler `onSomeEvent()` and the method `doSomething()`.

After this modification of the component controller's meta model the Web Dynpro Code Generator re-generates and adapts the controller class and its related `IPrivate` and `IPublic` interfaces:

IPublic-API

The `IPublic-API` is extended by two methods and one constant:

- public method `doSomething()`
- event handler `onSomeEvent()`
- `WD_EVENT_SOME_EVENT` to dynamically subscribe an event handler to the event SomeEvent with the `IWDComponent-API`

The declared public method `doSomething()` and the event handler `onSomeEvent()` are added to the `IPrivate-API` and to the controller class `SampleComp.java`.

Besides the automatically added methods and constant the `IPublic-API` generically contonains the methods `wdGetAPI()` and `wdGetContext()` which can be invoked to access the generic controller API or the root context node of a used controller.

IPrivate-API

The `IPrivate-API` extends the above `IPrivate-API`. It is extended by two additional methods and one constant:

- `wdGetSampleCustController()` to access the `IPublic-API` of the used custom controller *SampleCust*.
- `wdFireEventSomeEvent()` to fire the serverside controller event *SomeEvent*,
- `WD_EVENTHANDLER_SOME_EVENT` to dynamically subscribe the event handler `onSomeEvent()` to an event at runtime.

To access the `IPrivate-API` within the controller class the shortcut variable `wdThis` of type `IPrivateSampleComp` has to be used.

See also

- Example: Implementing a Component with its Controllers and Interfaces [Page 14]

Shortcut Variables – Accessing Generic Controller Interfaces

Generic Controller APIs

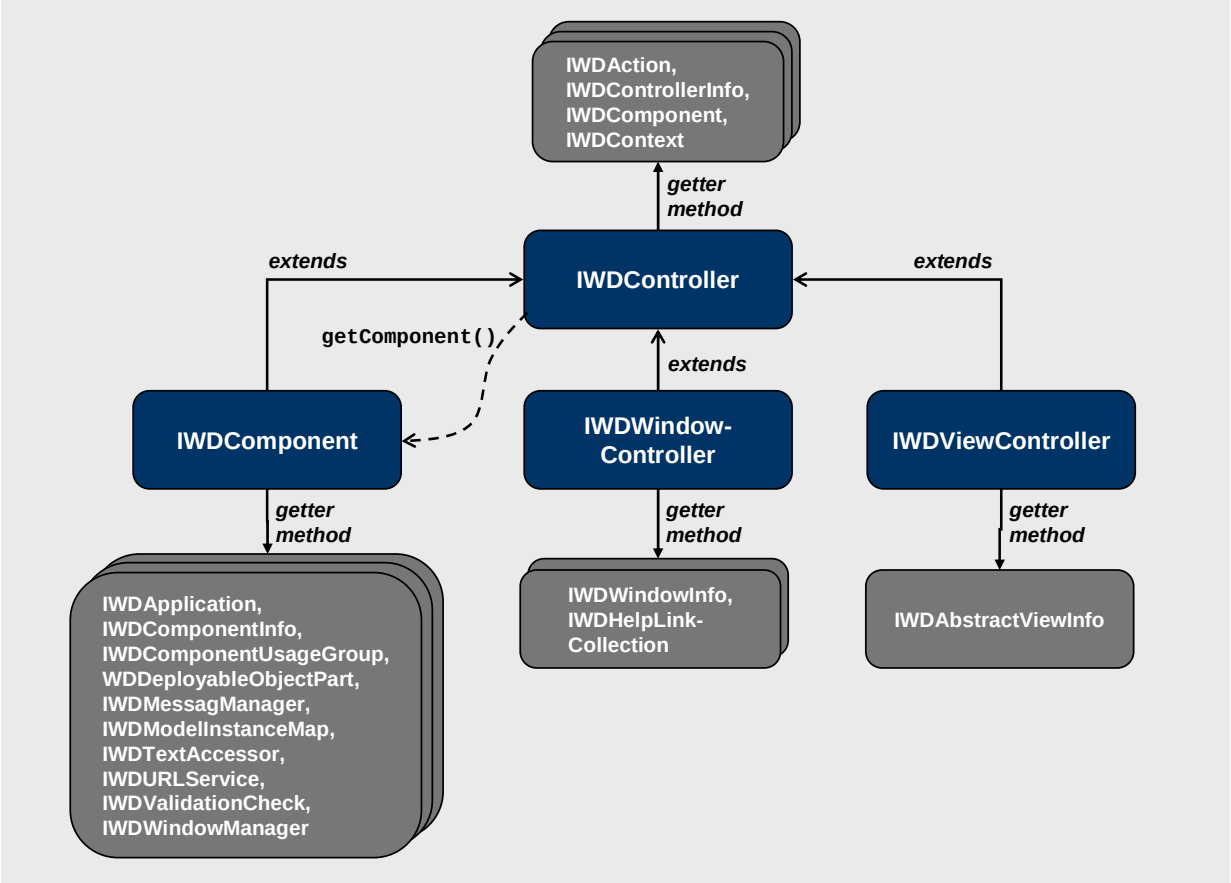
The *generic controller APIs* `IWDController`, `IWDComponent`, `IWDViewController` and `IWDWindowController` are implemented by the Web Dynpro Runtime itself.

These generic controller interfaces must be invoked by the application developer to fulfill special requirements:

- **Accessing or modifying controller metadata**: To access or dynamically modify the metadata information of a controller at runtime within your custom code you must invoke the corresponding *IWD…Info*-APIs. These APIs are exposed by getter methods of the generic controller APIs.
- **Accessing or modifying metadata of associated objects**: In some cases the metadata or the state of associated objects like *windows*, *applications*, *actions* or *component usage groups* have to be modified at runtime. The related generic *IWD**-APIs are again accessible via the generic controller APIs.
  Examples for this sceanrio are: dynamic event subscription, dynamic view composition, addition or removal of plugs, requesting focus in view layouts.

- **Accessing generic services**: Many generic service APIs like `IWDMessageManager`, `IWDWindowManager` or `IWDTextAccessor` can only be invoked via the generic controller APIs. In contrast to other services they are not based on a standalone service class like *WDURLGenerator*, *WDResourceHandler*, *WDWebResource* *WDSystemLandscape* or *WDProtocolAdapter*.

The next diagram illustrates which Web Dynpro Runtime APIs are associated with the generic controller APIs `IWDController`, `IWDComponent`, `IWDViewController` and `IWDWindowController`:



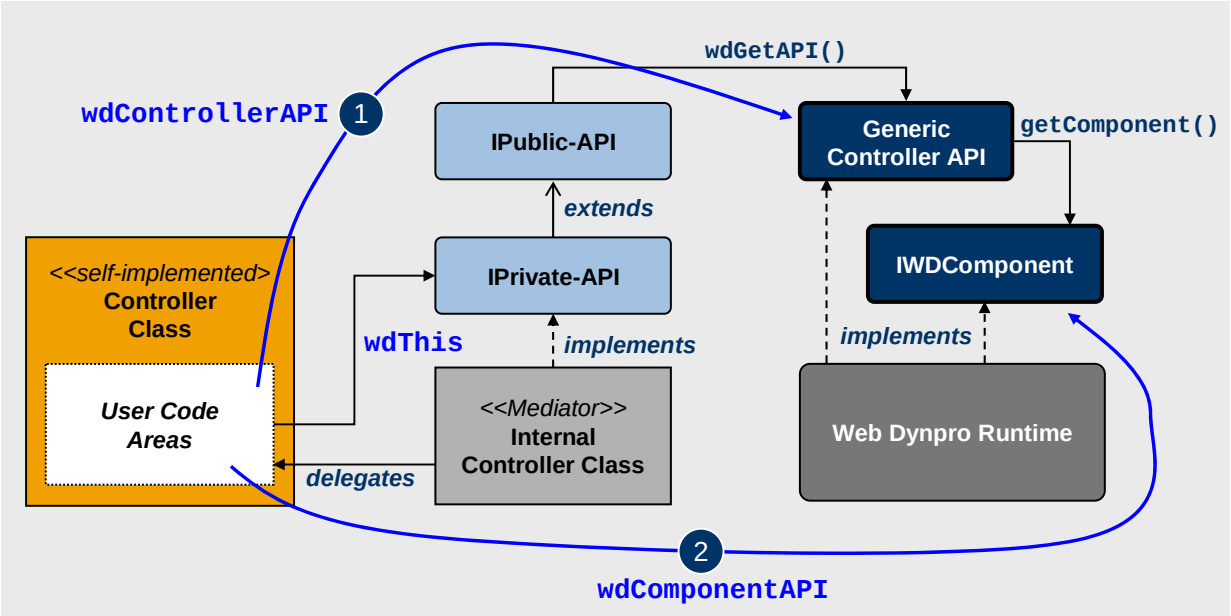
Controller Shortcut Variables

To easily access the generic controller APIs within your custom application code every Web Dynpro controller class has two private member variables `wdControllerAPI` and `wdComponentAPI`.

- `wdControllerAPI = wdThis.wdGetAPI()`
  Depending on the given controller type the variable `wdControllerAPI` is of type …
  - `IWDComponent` within a component controller
  - `IWDController` within a custom controller
  - `IWDWindowController` within a window controller
  - `IWDViewController` within a view controller.

- `wdComponentAPI = wdThis.wdGetAPI().getComponent()`
  The variable `wdComponentAPI` is of type `IWDComponent` within all controller classes.

The following diagram shows the relationships between the self-implemented controller class, the internal controller class, the generated controller interfaces `IPublic` and `IPrivate` and the generic controller APIs. The name controller *shortcut* variables for `wdControllerAPI` (1) and `wdComponentAPI` (2) is based on the fact, that they replace the method calls `wdGetAPI()`and `getComponent()`on the variable `wdThis`.



See also

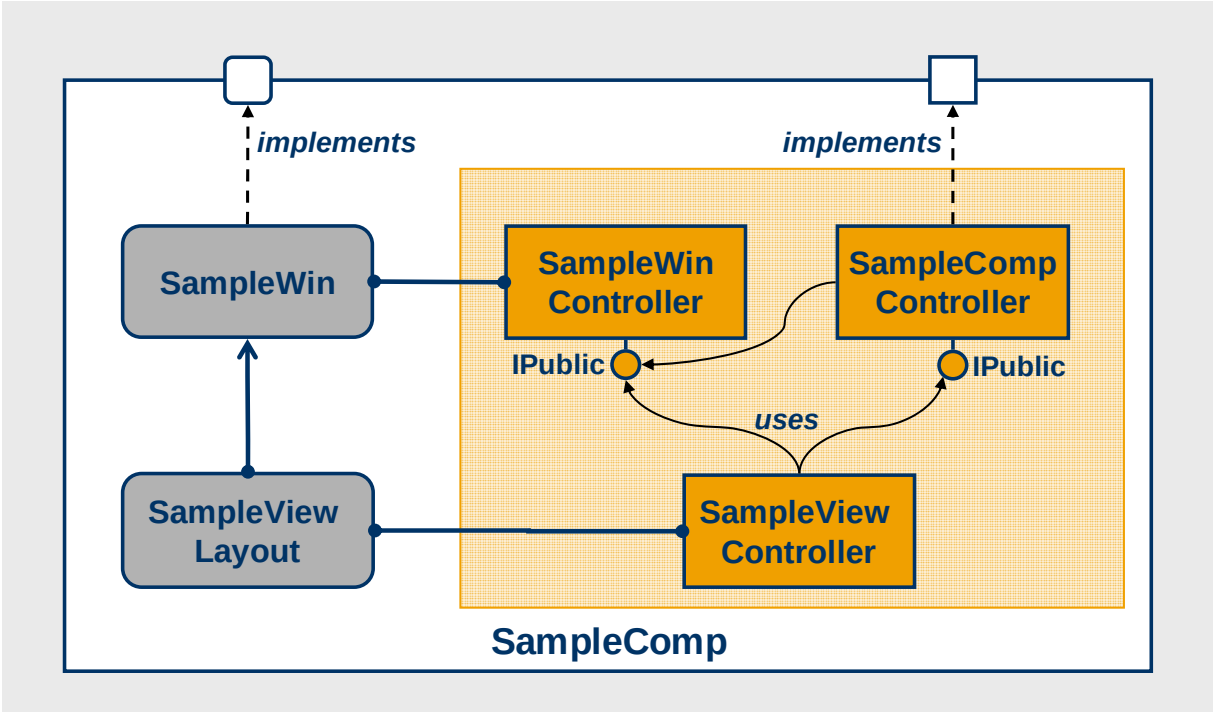
- Example: Implementing a Component with its Controllers and Interfaces [Page 14]
- Controller Class and Interface Reference [Page 21]

Implementing a Component with its Controllers and Interfaces

In this section the Web Dynpro controller and interface concept is illustrated in a concrete sample component. Based on some controller-specific declarations the `IPrivate`- and `IPublic`-APIs are extended by the Code Generation Framework of the Web Dynpro Tools.

The sample component comprises three controllers:

- Component controller *SampleComp*
- Window controller *SampleWin*
- View controller *SampleView*



Programming Tasks

The sample code blocks of the next three sections demonstrate how to solve typical programming tasks by invoking the generated controller interfaces `IPublic`- and `IPrivate` in the application code. These interfaces are based on the declarations (like methods, events, controller usages, actions or plugs) done by the application developer.

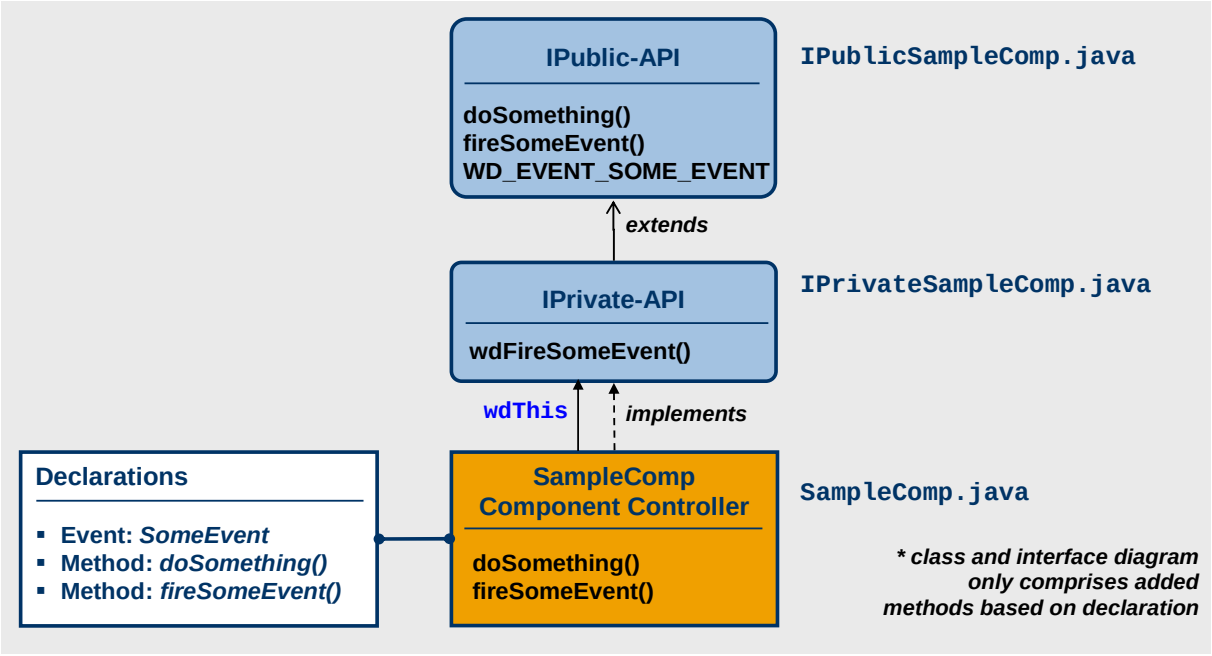
- Implementing the Component Controller [Page 15]
- Implementing the Window Controller [Page 16]
- Implementing the View Controller [Page 17]

See also

- Controller Concept [Page 1]
- Controller Interface Concept [Page 5]
- Controller Class and Interface Reference [Page 21]

Implementing the Component Controller

Overview



Using the IPrivate API

Task 1: Calling public Method doSomething()

The defined public method `doSomething()` can be invoked within the component controller class by using the member variable `wdThis` of type `IPrivate`. The declared methods are exposed by the `IPublic`-API which is extended by the `IPrivate`-API.

SampleComp.java

```
wdThis.doSomething();
```

Task 2: Firing the event SomeEvent

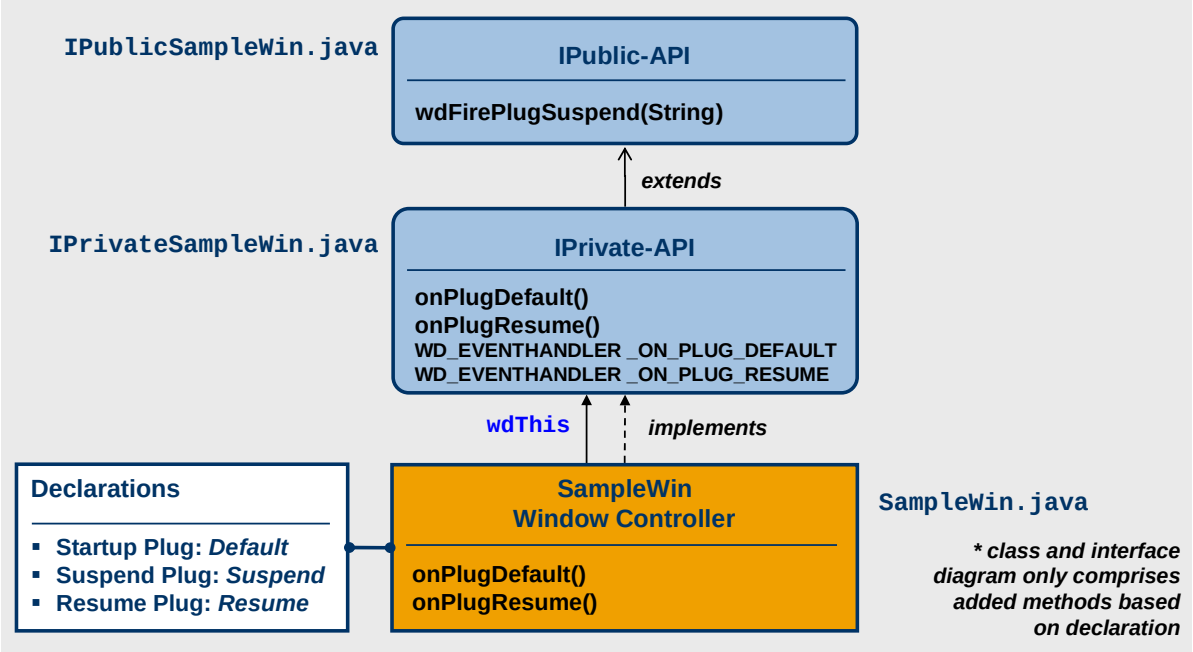
The event *SomeEvent* can be fired by invoking a corresponding `wdFireXY()` method defined in the `IPrivate`-API of the component controller.

SampleWin.java

```
wdThis.wdFireSomeEvent();
```

Implementing the Window Controller

Overview



Using the IPublic API

Task 1: Firing a Suspend Plug

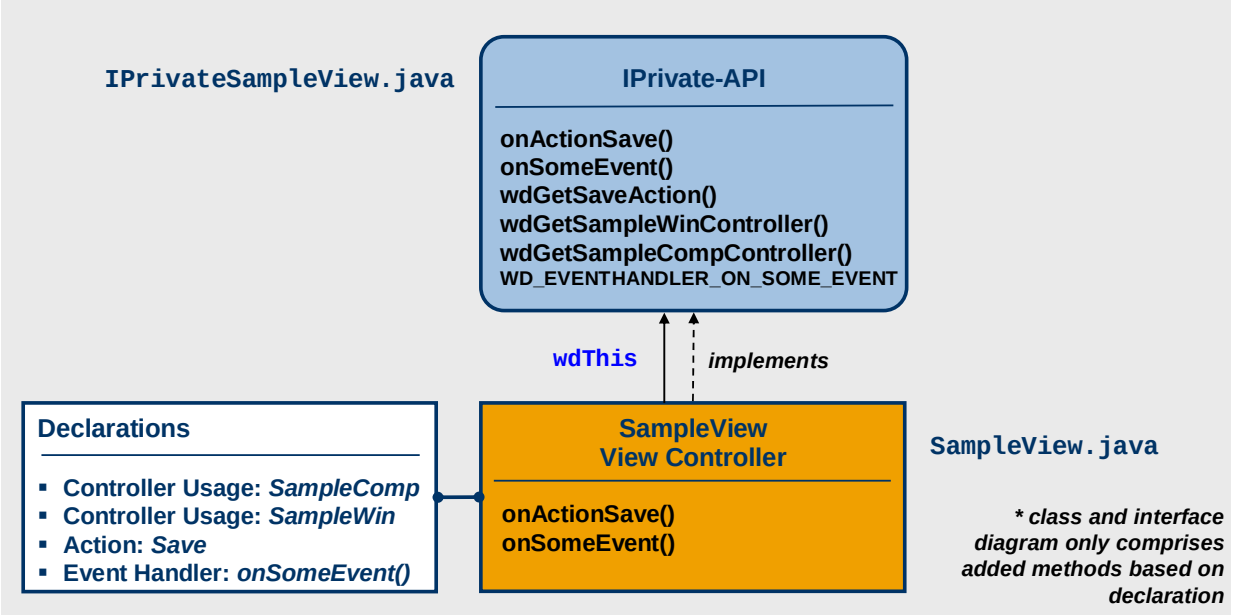
The suspend Plug *Suspend* can be fired by invoking the `IPublic`-API of the window controller. As the `IPrivate`-API extends the `IPublic`-API the generated method `wdFirePlugSuspend()` can be invoked on the member variable wdThis (of type `IPrivate`-API).

SampleWin.java

```
wdThis.wdFirePlugSuspend(someTargetAppURL);
```

Implementing the View Controller

Overview



Implementing Hook Methods

Task 1: Implementing the Action Event Handler onActionSave()

The action *Save* is associated with the event handler `onActionSave()`. In this method the application logic for the action *Save* can be implemented.

SampleView.java

```
//@@begin javadoc:onActionSave(ServerEvent)
/**
 * Declared validating event handler.
 *
 * @param wdEvent generic event object provided by framework
 */
 //@@end
public void onActionSave(
  com.sap.tc.webdynpro.progmodel.api.IWDCustomEvent wdEvent )
{
  //@@begin onActionSave(ServerEvent)
  ...
  //@@end
}
```

Using the IPrivate-API

Task 2: Dynamically subsribing an event handler to an event

You can dynamically subsribe the view controller's event handler `onSomeEvent()` to the event *SomeEvent* (exposed by the component controller) at runtime with the generic component controller API `IWDComponent`. For this purpose the generated event- and event handler constants must be used.

SampleView.java

```
//@@begin javadoc:wdDoInit()
/** Hook method called to initialize controller. */
//@@end
public void wdDoInit()
{
  //@@begin wdDoInit()
  IWDEventId eventId =
    wdThis.wdGetSampleCompController().WD_EVENT_SOME_EVENT;
  IWDEventHandlerId eventHandlerId =
    wdThis.WD_EVENTHANDLER_ON_SOME_EVENT;

  wdComponentAPI.addEventHandler(eventId,eventHandlerId);
  //@@end
}
```

Task 3: Disabling the Save Action Object

The object instance of the defined action *Save* can be accessed in the `IPrivate`-API of the view controller.

SampleView.java

```
wdThis.wdGetSaveAction().setEnabled(false);
```

Using the IPublic-API of another controller

Task 4: Firing the suspend plug of a window conroller

You can directly fire the suspend plug *Suspend* defined in the window controller *SomeWin* in a view controller. To invoke the `IPublic`-API of the window controller a corresponding controller usage relation must be defined.

SampleView.java

```
IPublicSampleWin winPublicAPI = wdThis.wdGetSampleWinController();
```

```
winPublicAPI.wdFirePlugSuspend(someSuspendURL);

// direct access
wdThis.wdGetSampleWinController().wdFirePlugSuspend(someSuspendURL);
```

Task 5: Firing event SomeEvent defined in the component controller

Events which are defined in non-view controllers cannot be fired by other controllers because the `IPublic`-API does not expose a corresponding `wdFireEventXY()`-method by default. Consequently you must declare a public method `fireSomeEvent()` in the component controller which fires the event *SomeEvent* and which can be invoked by the view controller. To invoke the `IPublic`-API of the component controller a corresponding controller usage relation must be defined.

SampleView.java

```
wdThis.wdGetSampleCompController().fireSomeEvent()
```

SampleComp.java

```
//@@begin javadoc:fireSomeEvent()
/**
 * Method declared by application.
 */
//@@end
public void fireSomeEvent( )  {
  //@@begin fireSomeEvent()
  wdThis.wdFireEventSomeEvent();
  //@@end
}
```

Using the Generic Controller APIs

Task 6: Changing the keyboard input focus on the client

Change the keyboard input focus to the UI element whose primary purpose is to raise an event bound to the action Save.

SampleView.java

```
//Assumption: Only one single UI element is bound to action Save
wdControllerAPI.requestFocus(wdThis.wdGetSaveAction());
```

Task 7: Accessing the Message Manager

The `IWDMessageManager`-API is exposed to all controllers in a Web Dynpro component via the generic component controller API `IWDComponent`. This interface can directly be accessed with the shortcut variable `wdComponentAPI`.

SampleView.java

```
IWDMessageManager msgMgr = wdComponentAPI.getMessageManager();
```

Task 8: Accessing the Generic Controller API of the embedding Window

Invoke the generic `IWDWindowController`-API of the window controller instance which embeds this view you .

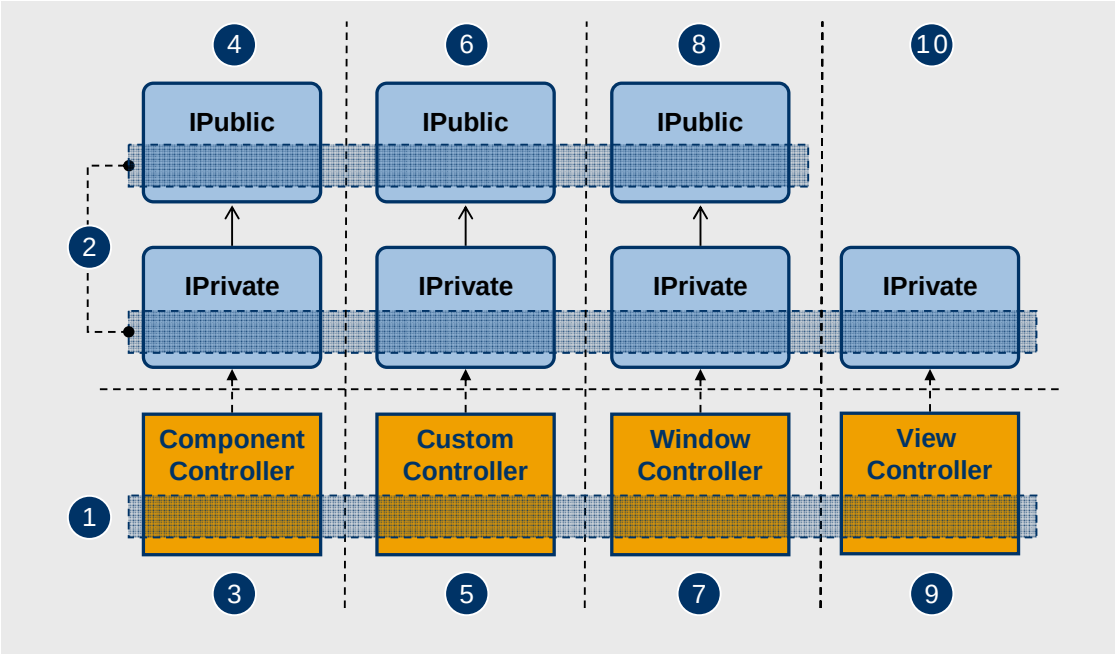
SampleView.java

```
IWDWindowController wdwController = wdControllerAPI.getWindowController();
```

Controller Class and Interface Reference

The class and interface reference for all four Web Dynpro controller types (*Component*, *Custom*, *Window* and *View* controllers) is based on the following structure:

- All redundant parts which are common for all controller classes or controller interfaces are documented in two separate sections.
- The controller-specific additions to the common controller reference are documented within additional sections.
- Controller classes and controller interfaces are described separately.



Reference

- All Controllers:
  - Common Controller Class Reference (1) [Page 22]
  - Common Controller Interface Reference (2) [Page 31]
- Component Controller:
  - Component Controller Class Reference (3) [Page 24]
  - Component Controller Interface Reference (4) [Page 37]
- Custom Controller:
  - Custom Controller Class Reference (5) [Page 26]
  - Custom Controller Interface Reference (6) [Page 36]
- Window Controller:
  - Window Controller Class Reference (7) [Page 27]
  - Window Controller Interface Reference (8) [Page 38]

- View Controller:
  - View Controller Class Reference (9) [Page 29]
  - View Controller Interface Reference (10) [Page 40]

Common Controller Class Reference

The Common Controller Class Reference comprises those parts all Web Dynpro controller classes have in common.

The additional parts, which depend on a specific controller type, are documented within separate sections on the four Web Dynpro controller types *Component*, *Custom*, *Window* and *View* controller.

Methods

Predefined Hook Methods

```
public void wdDoExit()
```

Called by the Web Dynpro Runtime before destroying a controller instance to clean it up.

```
public void wdDoInit()
```

Called by the Web Dynpro Runtime to initialize a controller instance.

Additional public Methods based on Declaration

```
public <type> <method name>([parameter { "," parameter }])
  [throws {checked exception class type}]
```

Controller method with arbitrary signature (parameters and return type).

Checked or *compiler-enforced* exceptions can be defined for all controller methods, which are declared by an application developer. In the Web Dynpro Tools checked exceptions are added to a method definition similar to the addition of method parameters. The calling public controller method must take care of these exceptions by either catching or adding them in its own method definition (declaring them in its throws clause).

Within *View Controllers* additional methods based on declaration are of type *public*. Nevertheless they cannot be called from other controllers because view controllers do not expose an `IPublic`-API.

```
public void <event handler name>( IWDCustomEvent wdEvent,
  "," parameter {"," parameter}])
```

Event handler method. The parameters must be compatible with the parameters of the subcribed event. The name of event handlers should start with the prefix **on**.

Other private Methods

```
private <type> <method name>([parameter { "," parameter }])
```

Additional private methods can be added within the final user coding area: `//@@begin others` - `//@@end` code. These methods are not added to the controller's `IPublic`-API so that they are not exposed to other controllers even when they are specified with the `public` visibility statement.

Member Variables

Predefined Shortcut Variables
`private final IPrivate<controller name> wdThis` References the generated `IPrivate` interface of the internal controller class.
`private final IPrivate<controller name>.IContextNode wdContext` References the root node within a controller context. Provides typed access not only to the elements of the root node but also to all nodes in the context (methods node<node name>()) and their currently selected element (methods `current<node name>Element()`). It also facilitates the creation of new elements for all nodes (methods `create<node name>Element()`).
`private final com.sap.tc.webdynpro.progmodel.api.IWDComponent wdComponentAPI` A shortcut for `wdThis.wdGetAPI().getComponent()`. Represents the generic API of the Web Dynpro component this controller belongs to. Can be used to access additional Web Dynpro Runtime APIs associated with the `IWDComponent`-API like the message manager, the window manager or to dynamically add/remove event handlers.
`private final <generic controller API> wdControllerAPI` A shortcut for `wdThis.wdGetAPI()`. Represents the generic controller API (`IWDController`, `IWDComponent`, `IWDViewController` and `IWDWindowController`) of the generic Web Dynpro counterpart for this controller.
Predefined Variables
`logger` Logging location
`serialVersionUID` This variable is not relevant for application code development. It is required for the SAP NetWeaver serialization framework to check the binary compatibility of serialized objects.
Private Member Variables
`private <type> <variable name>` An arbitrary number of private member variables can be added within the final user coding area: `//@@begin others - //@@end` code

See also

- Component Controller Class Reference [Page 24]
- Custom Controller Class Reference [Page 26]
- Window Controller Class Reference [Page 27]
- View Controller Class Reference [Page 29]
- Common Controller Interface Reference [Page 31]

Component Controller Class Reference

This reference for component controller classes comprises specific additions to the Common Controller Class Reference [Page 22].

Naming Convention

The name of a component controller class is based on the component name. Component names should end with the suffix **`Comp`**.

Component Interface Relation

The component controller class implements the following component interface controllers:

- *Local Component Interface:* It is defined within the local component interface of the component, the component controller belongs to.
- *Standalone Component Interface.* standalone component Interfaces are defined outside an implementing component. The component, the component controller belongs to, can define several implementation relations to standalone component interfaces. All component controller interfaces (its context, methods and events) which are defined in these related standalone component Interfaces must be implemented by the component controller class.

Lifespan

A component controller has the same lifespan as the component it belongs to. The creation or the destruction of a component instance implies the creation or the destruction of the related component controller instance.

For component usages with the *lifecycle*-property **`createOnDemand`** the component creation is automatically done by the Web Dynpro Java Runtime when the `IExternal` API of the component interface controller is invoked within custom application code.

Special Methods

Additional Predefined Hook Methods
`public void wdDoApplicationStateChange( IWDApplicationStateChangeInfo stateChangeInfo, IWDApplicationStateChangeReturn stateChangeReturn)` Hook that informs the application about a state change. This hook is called e.g. to tell the application that it will be • left via a suspend plug and therefore should go into a suspend/sleep mode with minimal need of resources. • left due to a timeout and could write it's state to a data base if the user comes back later on. The concrete reason is available via the `IWDApplicationStateChangeInfo`-API. **Important**: *This hook is called for the top level component only!*
`public void wdDoBeforeNavigation(boolean isCurrentRoot)` Hook before the navigation phase starts. This hook allows you to flush the model queue and handle any errors that occur. Firing outbound plugs is allowed in this hook. Using *preorder depth-first traversal*, this hook is called for all component controllers starting with the current

root component.
The parameter `isCurrentRoot` is *true* if this is the root of the current request.

```
public void wdDoPostProcessing(boolean isCurrentRoot)
```

Hook called to handle data retrieval errors before rendering. After `wdDoModifyView()`, the Web Dynpro Framework gets all context data needed for rendering by validating the contexts (which in turn calls the supply functions and supplying relation roles). In this hook, the application should handle the errors which occurred during validation of the contexts. Using preorder depth-first traversal, this hook is called for all component controllers starting with the current root component.
The parameter `isCurrentRoot` is *true* if this is the root of the current request.

Permitted operations: Flushing model queue, creating messages, reading context and model data

Forbidden operations: Invalidating model data, manipulating the context, firing outbound plugs, creating components

Shortcut Variables

Predefined Shortcut Variables

```
private final IWDComponent wdControllerAPI
```

A shortcut for `wdThis.wdGetAPI()`. Represents the generic controller API `IWDComponent` of the generic Web Dynpro counterpart for this controller.

`wdThis, wdComponentAPI, wdContext`

See common controller class reference

See also

- Common Controller Interface Reference [Page 31]
- Component Controller Interface Reference [Page 24]

Custom Controller Class Reference

This reference for custom controller classes comprises specific additions to the Common Controller Class Reference [Page 22].

Naming Convention

Custom controller names should end with the suffix **`Cust`**.

Lifespan

The lifespan of a custom controller instance is automatically managed by the Web Dynpro Runtime. It is created *on-demand*, that means when the `IPublic`-API of the custom controller is invoked by another controller in the same component or when its context is referenced via context mapping by another controller context.

Shortcut Variables

Predefined Shortcut Variables
`private final IWDController wdControllerAPI` A shortcut for `wdThis.wdGetAPI()`. Represents the generic controller API `IWDController` of the generic Web Dynpro counterpart for this controller.
`wdThis, wdComponentAPI, wdContext` See common controller class reference

See also

- Common Controller Class Reference [Page 22]
- Custom Controller Interface Reference [Page 36]

Window Controller Class Reference

This reference for window controller classes comprises specific additions to the Common Controller Class Reference [Page 22].

Naming Convention

The name of a window controller class is based on the window name. Window names should end with the suffix **`Win`**.

Component Interface View Relation

A window can optionally implement one or many component interface views. Component interface views can be defined within *local* or *standalone component interfaces.*

- *Local component interface:* The interface views which are defined within a local component interface definition must be implemented by at least one window in the same component. A window implements a component interface view, when it has the same *inbound*, *outbound*, *startup*, *exit*, *suspend* and *resume* plugs with the same parameters as defined in the component interface view.
- *Standalone component interface:* A component which defines an implementation relation to a standalone component interface must implement it. A component can implement several standalone component interfaces. The interface views which are defined within a related standalone component interface must be implemented by at least one window in the same component. A window can also implement several standalone or local component interfaces.

When a window implements a component interface view its window controller implements the plug event execution (`wdFirePlug<plug name>()`) and the plug event handling (`onPlug<plug name>()`).

Lifespan

Creation

A window controller instance is automatically created by the Web Dynpro Java Runtime as soon as the associated window is part of the current view assembly. Otherwise (the window does not belong to the current view assembly) the window controller is **not** being created even in case the component, this window belongs to, is created within its embedding component.

Take into account that a window controller is **not** created *on-demand* in case the window is not part of the current view assembly. This means that the invocation of the `IPublic`-API by another controller in the same component or the existance of another mapped controller context does not induce the creation of the window controller instance (like it is the case for custom controllers). In this case an exception is fired by the Web Dynpro Java Runtime.

Destruction

When a component instance is destroyed all contained window controllers naturally get destroyed too.

In case the embedding component instance is not destroyed the destruction of a window controller depends on the *Lifespan* property definition:

- *Lifespan* = **`Framework controlled`**: When the window disappears from the view compostion based on a navigation step the Web Dynpro Runtime does *not* destroy the window controller. When the window afterwards reappears in the view assembly the window controller is still alive and gets not initialized again. Consequently the `wdDoInit()` hook method is not called by the Web Dynpro Runtime again.
- *Lifespan* = **`When visible`**: When the window disappears from the view compostion based on a navigation step the Web Dynpro Runtime automatically destroys the window controller. When the window afterwards reappears in the view assembly a new window controller instance is created and the `wdDoInit()` hook method is called by the Web Dynpro Runtime again.

Special Methods

Predefined Event Handlers
`public void onPlugDefault( IWDCustomEvent wdEvent ["," parameter {"," parameter}] )` Event handler for startup plug **'Default'**. Additional event parameters can be declared. The value of a URL-parameter with the same name like a declared event parameter of type `String` is automatically passed by the Web Dynpro Java Runtime.

Plug Event Handlers based on Declaration
`public void onPlug<inbound plug name>( IWDCustomEvent wdEvent ["," parameter {"," parameter}])` Event handler for inbound plug. Additional event parameters can be declared.
`public void onPlug<startup plug name> ( IWDCustomEvent wdEvent ["," parameter {"," parameter}] )` Event handler for additional startup plug. Additional event parameters can be declared.
`public void onPlug<resume plug name>( IWDCustomEvent wdEvent ["," parameter {"," parameter}] )` Event handler for window resume plug. Additional event parameters can be declared.

A window can implement one or many component interface views. In this case all plugs of a component interface view must also be defined within the window. The associated plug event handlers are implemented in the window controller. Normally a plug event handler is only invoked by the Web Dynpro Runtime when the associated plug is the target of a navigation link. Only in case a plug is explicitly defined as the **Default Plug** of an embedded component interface view the Web Dynpro Runtime invokes its event handler independent from a defined navigation link. By default the *Default Plug* property of an embedded (component interface) view is set **none**.

Shortcut Variables

Predefined Shortcut Variables	
	`private final IWDWindowController wdControllerAPI` A shortcut for `wdThis.wdGetAPI()`. Represents the generic controller API `IWDWindowController` of the generic Web Dynpro counterpart for this controller.
	`wdThis, wdComponentAPI, wdContext` See common controller class reference

See also

- Common Controller Class Reference [Page 22]
- Window Controller Interface Reference [Page 38]

View Controller Class Reference

This reference for view controller classes comprises specific additions to the Common Controller Class Reference [Page 22].

Naming Convention

The name of a view controller class is based on the view name. View names should end with the suffix **`View`**.

Lifespan

Creation

A view controller instance is automatically created by the Web Dynpro Java Runtime as soon as the associated view is part of the current view assembly.

Destruction

When a component instance is destroyed all contained view controllers naturally get destroyed too.

In case the embedding component instance is not destroyed the destruction of a view controller depends on the *Lifespan* property definition:

- *Lifespan* = **`Framework controlled`**: When the view disappears from the view compostion based on a navigation step the Web Dynpro Runtime does *not* destroy the view controller. When the view afterwards reappears in the view assembly the view controller is still alive and gets not initialized again. Consequently the `wdDoInit()` hook method is not called by the Web Dynpro Runtime again.
- *Lifespan* = **`When visible`**: When the view disappears from the view compostion based on a navigation step the Web Dynpro Runtime automatically destroys the view controller. When the view afterwards reappears in the view assembly a new view controller instance is created and the `wdDoInit()` hook method is called by the Web Dynpro Runtime again.

Special Methods

Predefined Hook Methods

```
public void wdDoBeforeAction(
  com.sap.tc.webdynpro.progmodel.api.IWDBeforeAction validation)
```

Hook method called for additional validation of user input. The parameter `validation` is an interface for reporting validation errors.

```
public static void wdDoModifyView(
  IPrivateSampleView wdThis,
  IPrivateSampleView.IContextNode wdContext,
  com.sap.tc.webdynpro.progmodel.api.IWDView view,
  boolean firstTime)
```

Hook method called to modify a view layout just before rendering. It is designed for the creation of a UI tree or UI sub-tree at runtime in case it is not possible to declare the UI at design time.

The Web Dynpro programming model recommends that UI elements can only be accessed by code executed within the call to this hook method.

The method is neither intended for fine grain UI manipulations nor for context manipulations. Fine grained UI manipulations are achieve via *data binding* that means via setting context attributes. Context manipulations are done in `wdDoInit`, action event handlers or event handlers.

Predefined Context Menu Event Handler

```
public void wdOnContextMenu(
  final IWDContextMenuManager contextMenuManager,
  final IWDContextMenuEvent event)
```

Service event handler for context menu events. It is called by the Web Dynpro Runtime when the user triggers the event for opening a context menu.

The handler method has two input parameters for invoking the following Web Dynpro APIs:

The `IWDContextMenuManager`-API passed to the context menu event handler provides methods to create, destroy and access menus, to access the context menu provided for the current event and to create menu items without using the generic factory method createElement() from the `IWDView`-API.

The `IWDContextMenuEvent`-API provides information about the *context* of the menu: the *menu originator* of the event (view element where the user has clicked), the *menu provider* (view element that provides a context menu, if any) and the node element describing the scope of the originator e.g. inside a table row.

The `IWDView`-API is not passed into this service event handler method to discourage manipulation of the view layout inside this method.

Inbound Plug Event Handlers based on Declaration

```
public void onPlug<inbound plug name>( IWDCustomEvent wdEvent
  ["," parameter {"," parameter}])
```

Event handler for inbound plug. Additional event parameters can be declared.

An inbound plug event handler is invoked by the Web Dynpro Runtime when the associated inbound plug is the target of a navigation link.

In case an inbound plug is explicitly defined as the **Default Plug** of a *view usage* (defined in a window) the Web Dynpro Runtime invokes its event handler independent from a defined navigation link. By default the *Default Plug* property of a defined view usage is set **none**.

Action Event Handlers based on Declaration

```
public void onAction<action name>( IWDCustomEvent wdEvent
  ["," parameter {"," parameter}])
```

Action event handler. Additional action parameters can be declared.

Shortcut Variables

Predefined Shortcut Variables

```
private final IWDViewController wdControllerAPI
```

A shortcut for `wdThis.wdGetAPI()`. Represents the generic controller API `IWDViewController` of the generic Web Dynpro counterpart for this controller.

`wdThis, wdComponentAPI, wdContext`

See common controller class reference

See also

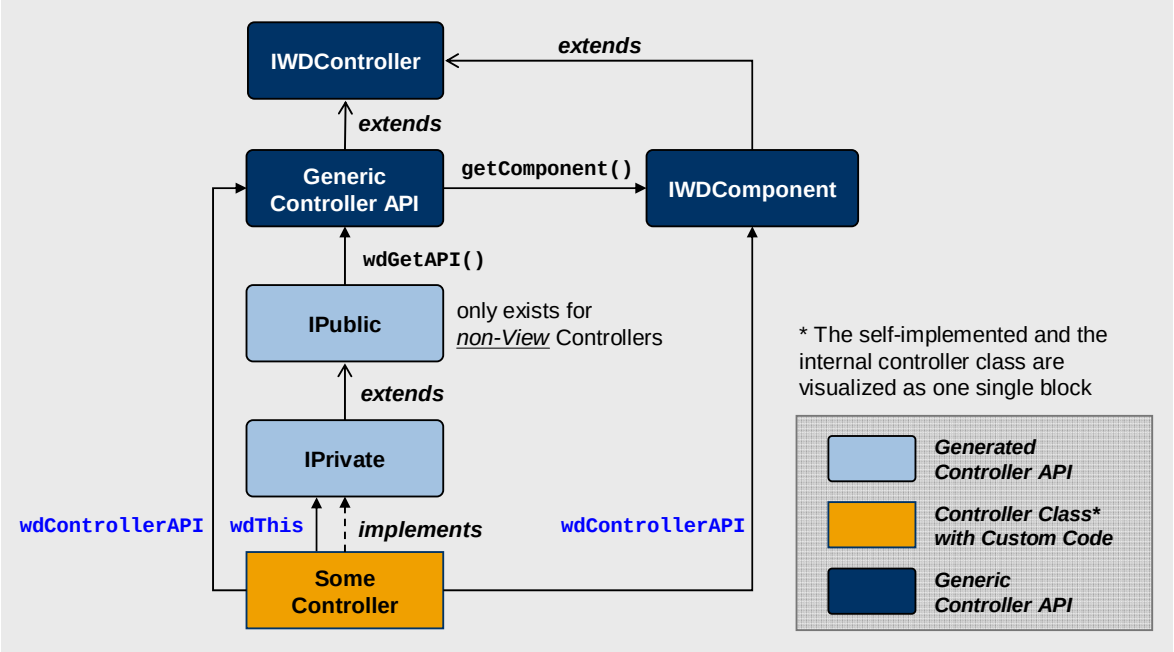
- Common Controller Interface Reference [Page 31]
- View Controller Interface Reference [Page 40]

Common Controller Interface Reference

The Common Controller Interface Reference comprises those parts all Web Dynpro controller interfaces have in *common.* Please note that there is no special Web Dynpro controller type called Common Controller. The adjective *common* refers to those parts all controllers have in common but not to a specific controller type.

The additional parts, which depend on a specific controller type, are documented within separate sections on the four Web Dynpro controller types *Component*, *Custom*, *Window* and *View*.

API Overview



IPrivate API

The `IPrivate` interface extends the `IPublic`-API. It is only visible within the self-implemented controller class (application class) and can be invoked there with the shortcut variable **`wdThis`**. It is not visible for other controllers.

Additional Methods based on Declaration
`public <type> <method name>([parameter { "," parameter }])` Controller method with arbitrary signature (parameters and return type). The return type depends on the declaration.
`void wdFireEvent<event name>( [parameter { "," parameter }] )` Fires a serverside controller event and passes definded optional parameters to the event listeners.  Events can only be defined in *non-view* controllers.  As the method `wdFireEvent<event name>()` is not exposed in the `IPublic`-API

the event cannot directly be fired by another controller in the same component. You must explicitly declare a public method which itself fires the defined *private* event.

`IPublic<controller name>`
`  wdGet<non-view controller name>Controller()`
 	
Gets `IPublic` interface of a controller belonging to the same component.

`wdGet<component usage name>ComponentUsage()` Gets `IWDComponentUsage` API for the defined component usage. This is for example used to create and destroy component instances at runtime (*lifecycle* property of the component usage is defined as **`manual`**).

`IExternal<component usage name>`
`  wdGet<component usage name>Interface()`
 	
Gets `IExternal` interface of the used component interface controller.

Additional Constants for declared Event Handlers
`public static` `IWDEventHandlerId` `  WD_EVENTHANDLER_<event handlername>` Constant for a declared event handler method. Can be used for dynamic event subscription and un-subscription. See `IWDComponent`-API.

IPublic API

The `IPublic`-API of *non-view* controllers is visible for other controllers as soon as a corresponding controller usage relation is defined. It can be invoked there with the accessor method `wdThis.wdGet<non-view controller name>Controller()`.

View Controllers do not expose an `IPublic`-API to other controllers. The predefined methods `wdGetAPI()` and `wdGetContext()` and the declared methods are consequently added to the `IPrivate`-API.

Predefined Methods
`<generic controller API> wdGetAPI()` See controller interface reference of the four special Web Dynpro controller types.
`IContextNode wdGetContext();` Returns the context root node. See section *Inner Context APIs* below.
Additional public Methods based on Declaration
`public <type>` `<method name>([parameter { "," parameter }])` Controller method with arbitrary signature (parameters and return type). The return type depends on the declaration.
`public void` `<event handler name>( IWDCustomEvent wdEvent,` `  ["," parameter {"," parameter}])` Event handler method. The parameters must be compatible with the parameters of the subcribed event. The name of event handlers should start with the prefix **on**.
Additional Constants for declared Events

```
public static IWDEventId WD_EVENT_<event name>
```

Event constant for a declared event of a non-view controller. Can be used for dynamic event subscription and un-subscription. See `IWDComponent`-API.

Events can only be defined in *non-view* controllers.

Inner Context APIs

Every Web Dynpro controller context comprises a root context node with no attributes by default:

- Name: **Context**
- Cardinality: *1..1*
- Singleton: *true*
- Selection cardinality: *1..1*

For this context node the two interfaces `IContextNode` and `IContextElement` are generated to allow context programming and dynamic context modification (adding attributes or inner nodes) in your application code.

IContextNode

Interface for the root node **'Context'**.

```
public IPrivate<controller name>.IContextElement
  currentContextElement()
```

Returns the element at the lead selection. Returns `null` if the lead selection is not set.

```
public IWDContext wdGetAPI()
```

Provides access to the generic context API as described by `IWDContext`.

```
createContextElement()
createAndAddContextElement()
bind(IPrivate<controller name>.IContextElement element)
```

As the root context node has the cardinality *1..1* these methods are not needed within your application code.

You should never call these methods within your application code to avoid undesired effects. The method `createAndAddContextElement()` throws a runtime exception. Creating and binding a new context element to the root node deletes the whole context structure what is most probably not intended.

IContextElement

Interface for the elements of the root node **'Context'**

Generic Controller API

IWDController

```
public interface IWDController
```

Generic API of all Web Dynpro controllers. All controllers created by the Web Dynpro toolset implement this interface. Unless explicitly stated, the methods listed for this interface are not available to parent components through the component usage interface.

	Access within controller class: Use the shortcut variable **`wdControllerAPI`** to invoke the generic `IWDController`-API within your application code.
Generic Controller API (Controller Type Specific)	
	`public interface <generic controller API> extends IWDController` The generic controller API (`IWDController`, `IWDComponent`, `IWDViewController` and `IWDWindowController`) depends on the concrete controller type (custom, component, view and window controller) and represents the generic Web Dynpro counterpart for this controller. **Access within controller class:** Use the shortcut variable **`wdControllerAPI`** to invoke the generic controller API within your application code.
IWDComponent	

	`public interface IWDComponent extends IWDController`
	The `IWDComponent`-API is the generic API of all Web Dynpro component controllers. It can be invoked in all controllers of a component to use central services and Web Dynpro Runtime APIs which are relevant for all controllers in a component.
	Exposed Web Dynpro APIs:
	`IWDApplication, IWDComponentInfo, IWDApplication, IWDComponentInfo, IWDComponentUsageGroup, WDDeployableObjectPart, IWDMessagManager, IWDModelInstanceMap, IWDTextAccessor, IWDURLService, IWDValidationCheck, IWDWindowManager`
	Access within controller class:
	Use the shortcut variable **`wdComponentAPI`** to invoke the generic component controller API within your application code.

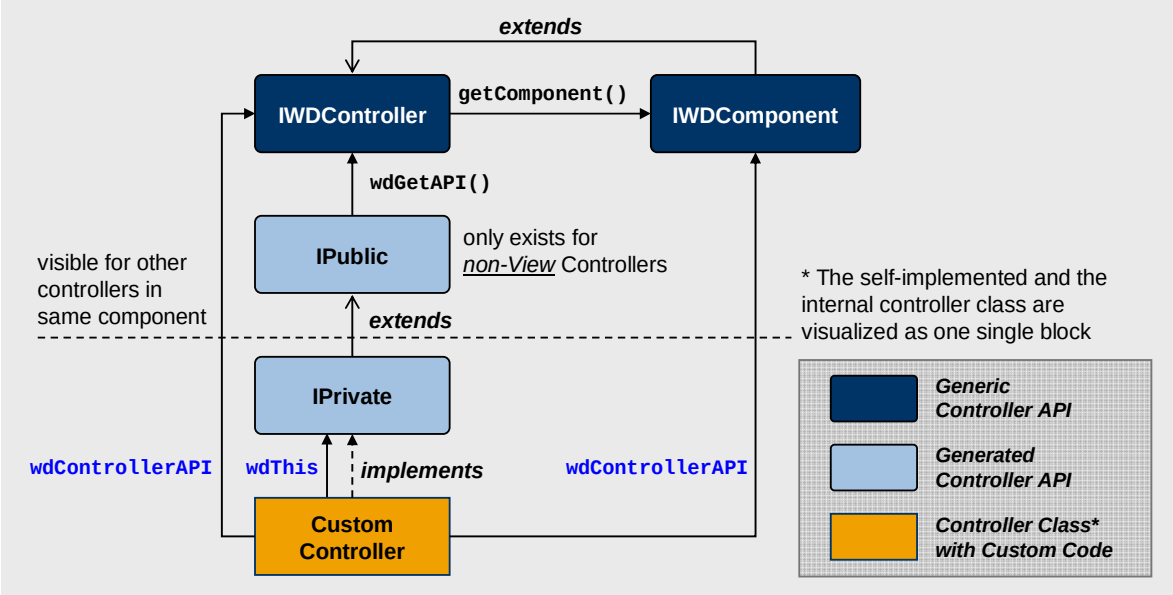
See also

- Component Controller Interface Reference [Page 24]
- Custom Controller Interface Reference [Page 36]
- Window Controller Interface Reference [Page 38]
- View Controller Interface Reference [Page 40]

Custom Controller Interface Reference

This reference for custom controller interfaces only comprises specific additions to the Common Controller Interface Reference [Page 31].

API Overview



IPrivate API

Same as described in the Common Controller Interface Reference.

IPublic API

Predefined Methods

`IWDController wdGetAPI()`

Provides access to the generic controller API of this custom controller.

Generic Controller API

IWDController

```
public interface IWDController
```

Access within controller class:

Use the shortcut variable **`wdControllerAPI`** to invoke the generic controller API within your custom controller code.

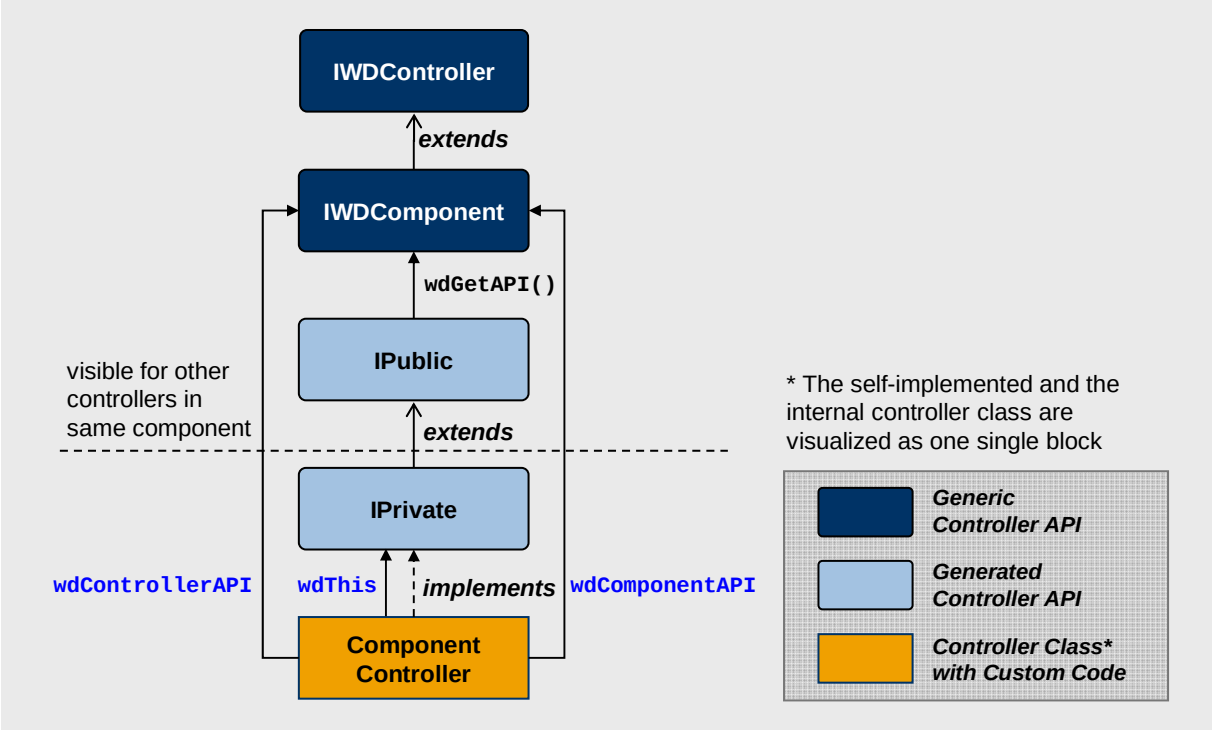
See also

- Common Controller Interface Reference [Page 31]
- Custom Controller Class Reference [Page 26]

Component Controller Interface Reference

This reference for component controller interfaces comprises specific additions to the Common Controller Interface Reference [Page 31].

API Overview



IPrivate API

Same as described in the Common Controller Interface Reference.

IPublic API

Predefined Methods

`IWDComponent wdGetAPI()`

Provides access to the generic controller API of this component controller.

Generic Controller API

IWDComponent

```
public interface IWDComponent extends IWDController
```

Access within controller class:

The shortcut variables **`wdComponentAPI`** wd and **`wdControllerAPI`** are both of the same type `IWDComponent`.

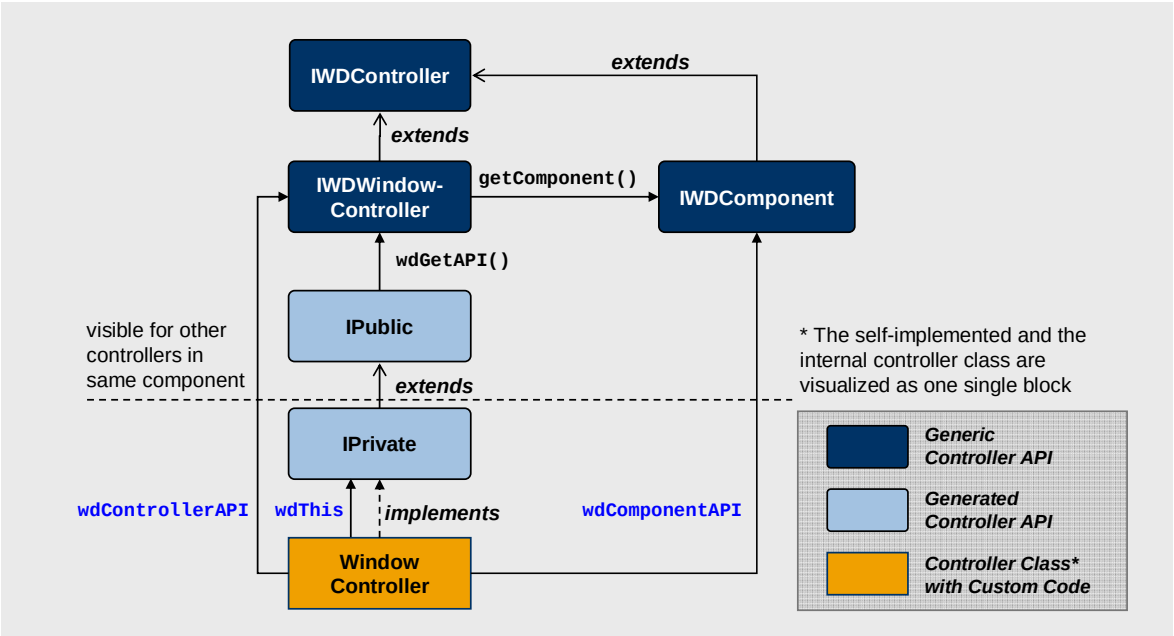
See also

- Common Controller Interface Reference [Page 31]
- Component Controller Class Reference [Page 24]

Window Controller Interface Reference

This reference for window controller interfaces comprises the differences to the Common Controller Interface Reference [Page 31].

API Overview



IPrivate API

Plug Event Handlers based on Declaration
`public void onPlug<inbound plug name>( IWDCustomEvent wdEvent ["," parameter {"," parameter}])` Event handler for inbound plug. Additional event parameters can be declared.
`public void onPlug<startup plug name> ( IWDCustomEvent wdEvent ["," parameter {"," parameter}] )` Event handler for additional startup plug. Additional event parameters can be declared.
`public void onPlug<resume plug name>( IWDCustomEvent wdEvent ["," parameter {"," parameter}] )` Event handler for window resume plug. Additional event parameters can be declared.
Additional Constants for declared Plugs
`public static IWDEventHandlerId WD_EVENTHANDLER_ON_PLUG_<INBOUND PLUG NAME>` Inbound plug event handler constant

```
public static IWDEventHandlerId
  WD_EVENTHANDLER_ON_PLUG_<STARTUP PLUG NAME>
```

Startup plug event handler constant

```
public static IWDEventHandlerId
  WD_EVENTHANDLER_ON_PLUG_<RESUMUE PLUG NAME>
```

Resume plug event handler constant

IPublic API

Additional Methods based on Declaration

```
void wdFirePlug<outbound plug name>([parameter {"," parameter}])
```

Fires outbound plug.

```
void wdFirePlug<suspend plug name>([parameter {"," parameter}])
```

Fires suspend plug.

```
void wdFirePlug<exit plug name>([parameter {"," parameter}])
```

Fires exit plug.

Generic Controller API

IWDWindowController

```
public interface IWDWindowController extends IWDController
```

Access within controller class:

Use the shortcut variable **`wdControllerAPI`** to invoke the generic window controller API within your window controller code.

See also

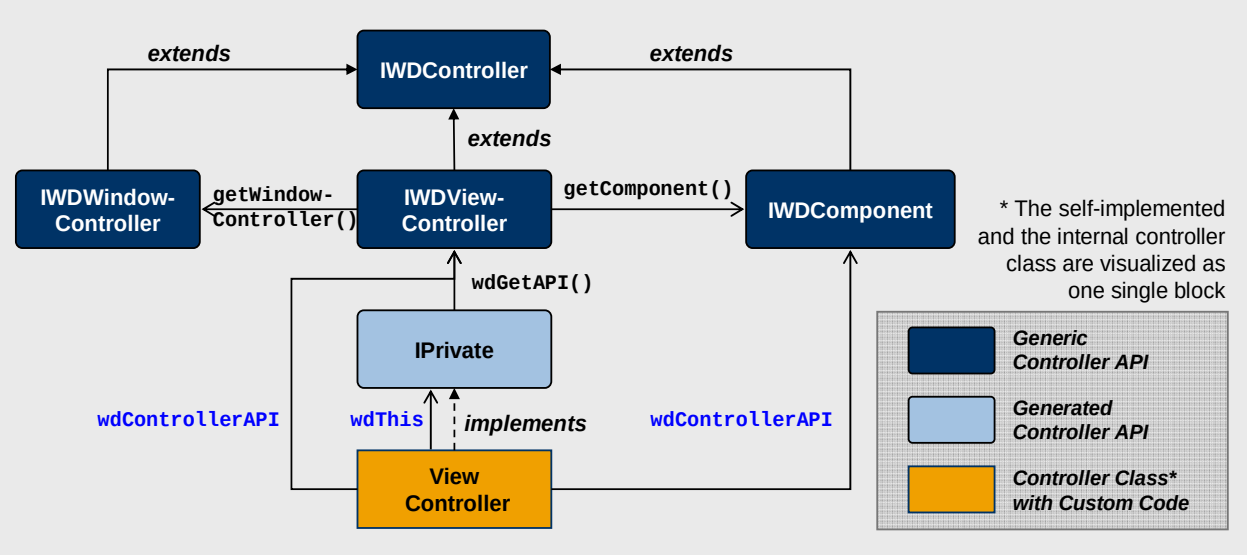
- Common Controller Interface Reference [Page 31]
- Window Controller Interfaces [Page 38]

View Controller Interface Reference

This reference for view controller interfaces comprises the additions to the Common Controller Interface Reference [Page 31].

View Controllers do not expose an `IPublic`-API to other controllers.

API Overview



IPrivate API

Predefined Methods
`IWDAction wdCreateAction( WDActionEventHandler eventHandler, String text);` Creates a new action for this view controller. The parameter `eventHandler` is the action's event handler with an appropriate signature. The parameter `text` is the text displayed in the UI element triggering this action.
`IWDAction wdCreateNamedAction( WDActionEventHandler eventHandler, String name, String text);` Creates a new named action for this view controller. The parameter `eventHandler` is the action's event handler with an appropriate signature. The parameter `text` is the text displayed in the UI element triggering this action. The parameter `name` is the action's name.
Additional Methods based on Declaration
`void wdFirePlug<outbound plug name>([parameter {“,” parameter}])` Fires outbound plug.
`IWDAction wdGet<action name>Action();` Gets action object for a defined action.

Event Handlers based on Declaration
`public void onAction<action name>( IWDCustomEvent wdEvent [“,” parameter {“,” parameter}])` Action event handler. Additional action parameters can be declared.
`public void onPlug<inbound plug name>( IWDCustomEvent wdEvent [“,” parameter {“,” parameter}])` Inbound plug event handler. Additional event parameters can be declared. See controller class reference.

Enumeration of all available Action Event Handlers
`public final class WDActionEventHandler.<ACTION NAME>` Constant for a defined action event handler. Used to dynamically create new action objects at runtime.

Additional Constants based on Declaration
`public static IWDEventHandlerId WD_EVENTHANDLER_ON_ACTION_<ACTION NAME>` Action event handler constant.
`public static IWDEventHandlerId WD_EVENTHANDLER_ON_PLUG_<INBOUND PLUG NAME>` Inbound plug event handler constant.

Generic Controller API

IWDViewController
`public interface IWDViewController extends IWDController` **Access within controller class:** Use the shortcut variable **`wdControllerAPI`** to invoke the generic view controller API within your view controller code.

See also

- Common Controller Interface Reference [Page 31]
- View Controller Class Reference [Page 29]