



**Developer Guide**

---

# **Sybase Brand Mobiliser 1.3**

DOCUMENT ID: DC01690-01-0130-02

LAST REVISED: June 2013

Copyright © 2013 by Sybase, Inc. All rights reserved.

This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase trademarks can be viewed at the Sybase trademarks page at <http://www.sybase.com/detail?id=1011207>. Sybase and the marks listed are trademarks of Sybase, Inc. ® indicates registration in the United States of America.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world.

Java and all Java-based marks are trademarks or registered trademarks of Oracle and/or its affiliates in the U.S. and other countries.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

# Contents

<b>Developing Brand Mobiliser Applications .....</b>	<b>1</b>
Advanced Interactive Messaging Server .....	1
Application States .....	2
Base States .....	3
Subscriber States .....	3
USSD States .....	4
Custom States .....	4
Input and Output Parameters .....	4
State Machine .....	5
Application Composer .....	5
State Transitions .....	6
Controlling State Transitions with Regular Expressions .....	7
Testing Regular Expressions .....	8
State Editor .....	9
Adding States to Applications .....	10
Editing State Properties .....	10
Removing States .....	11
Removing State Transitions .....	11
Developing Interactive Applications .....	11
Adding Keywords to Applications .....	12
Searching for a Keyword .....	13
Designing Application Task Flows .....	13
Short Codes, Long Codes, and Keywords .....	14
Developing Event Applications .....	15
Events .....	16
Creating Events .....	17
Creating One-Time Event Windows .....	17
Creating Recurring Event Windows .....	18
Assigning Events to Applications .....	19
Activation .....	19

Application Mode Transitions .....	20
Activating Applications .....	21
Activating Events .....	21
Deactivating Applications .....	22
Deactivating Events .....	22
Testing Applications .....	22
Testing Interactive Applications .....	23
Testing Event Applications .....	25
Importing Applications .....	26
Importing Application XML Files .....	26
Creating Applications from Templates .....	27
Exporting Applications .....	27
Exporting a Single Application .....	28
Exporting a Group of Applications .....	28
Sample Applications .....	28
Cash-Out Interactive Application .....	28
Mobiliser Counter Interactive Application .....	30
Utility Notification Event Application .....	31
<b>Developing Custom Application States .....</b>	<b>33</b>
Application Life Cycle .....	33
Developing and Deploying Custom States .....	35
Extending the SmappStatePlugin Class .....	35
Extending the AbstractDynamicMenu Class .....	39
Implementing State Logic .....	41
Custom State Information .....	43
Custom State Variables .....	45
Setting Up Apache Maven .....	53
Custom State Bundles .....	56
Custom State Samples .....	78
Sample GetMyWeather State .....	78
Sample Custom State .....	80
Sample Custom-Menu State .....	82
State SDK Core Components .....	84
<b>States Catalog .....</b>	<b>87</b>
Add Subscriber State .....	87

Application Call State .....	89
Application Call Return State .....	92
Compare Typed Variables State .....	94
Compare Variables State .....	96
Copy Variables State .....	98
Counter State .....	100
Get Subscriber State .....	101
Goto Application State .....	104
Process Subscriber State .....	106
Send SMS State .....	109
Send USSD Input State .....	111
Send USSD Menu State .....	112
Sample USSD Menu Code .....	116
Send USSD Text State .....	118
Set Variable State .....	120
Start Application State .....	122
Update Subscriber State .....	124
<b>Index .....</b>	<b>127</b>

# Contents

# Developing Brand Mobiliser Applications

In Sybase® Brand Mobiliser, states are basic building blocks that you can link sequentially to model application task flows. Applications are executed by the Brand Mobiliser Processing Engine at runtime.

Two application types, interactive and event, differ by both how they are invoked and how they perform. Interactive applications provide rich, user-interactive mobile services, and are typically invoked when mobile customers send a keyword to a preassigned short or long code. Event applications work non-interactively, such as batch processes that send campaign messages, and are typically invoked by events, such as scheduled times or triggers.

You can create applications:

- From scratch
- Using provided application templates
- By importing application files from another computer

Brand Mobiliser provides tools that let you visually compose a mobile-messaging application, test it using a built-in simulator, and deploy it, ready to be used by mobile consumers.

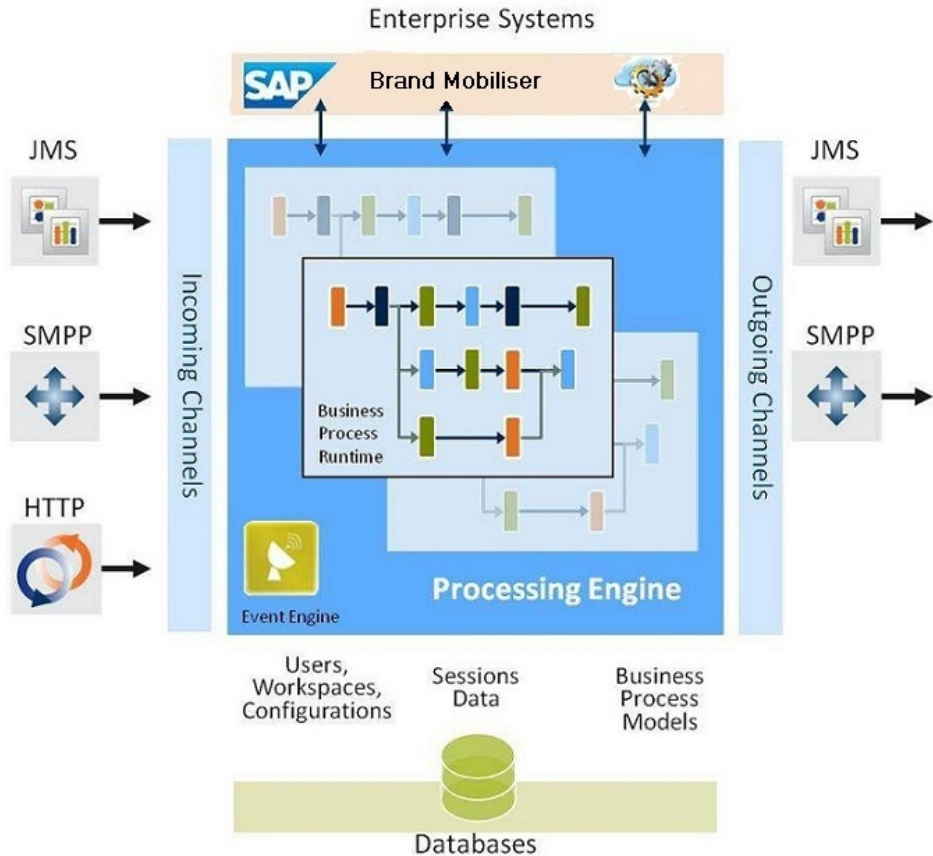
## **Advanced Interactive Messaging Server**

---

The core of Sybase® Brand Mobiliser is the Advanced Interactive Messaging Server (AIMS), also known as the messaging server.

The server components include:

- Brand Mobiliser Processing Engine (processing engine) – manages application life cycles, and provides the runtime environment.
- Event engine – invokes applications based on scheduled events.
- Session manager – tracks active sessions and terminates expired sessions.
- Channel manager – manages incoming and outgoing communication channels.



## Application States

In Brand Mobiliser, states are basic building blocks that you can link sequentially to model application-process flows.

Brand Mobiliser states are either:

- Standalone – implemented natively.
- Service – proxy to a Web service or aggregated Web services that are exposed through the service-oriented architecture (SOA) layer.

Brand Mobiliser provides:

- Base states – for composing task flows.
- Subscriber states – for performing operations on subscriber storage.



You can meet customer requirements by developing custom states using the Brand Mobiliser State SDK. You can add custom states dynamically using the plug-in mechanism that is enabled by the OSGi services registry.

Create Brand Mobiliser applications using the Application Composer Web tool. Application types include:

- Interactive – provide a user-interactive mobile service; typically invoked when mobile consumers send a keyword to a preassigned short code.
- Event – designed for batch processing; invoked by events, such as scheduled times, system triggers, or external triggers.

Most states can be used in either application type. However, there are a few states that are available only to a specific application type. For example, you can use the Process Subscriber state only in event applications, because it relies on the callback mechanism provided by the processing engine. You can use Application Call and Application Call Return states only for interactive applications, because these states do not support the callback mechanism.

Application Composer prevents you from adding invalid states to an application.

### **Base States**

Brand Mobiliser base states provide standalone functionality, without dependency on or interaction with external services. You commonly use base states to construct process flows.

Base states perform functions such as calling applications, comparing and copying variables, incrementing counters, sending SMS messages, and setting session variable values.

#### **See also**

- *Application Call State* on page 89
- *Application Call Return State* on page 92
- *Compare Typed Variables State* on page 94
- *Compare Variables State* on page 96
- *Copy Variables State* on page 98
- *Counter State* on page 100
- *Goto Application State* on page 104
- *Send SMS State* on page 109
- *Set Variable State* on page 120
- *Start Application State* on page 122

### **Subscriber States**

Applications that contain subscriber states have access to subscriber storage, which stores attributes that are useful in push campaigns.

Subscriber storage is nondurable storage for staging, or in-transit storage, pending batch transfer to the system of record. The database schema is designed to be generic, and is not fully optimized for large scale or more domain-specific purposes.

### **See also**

- *Add Subscriber State* on page 87
- *Get Subscriber State* on page 101
- *Process Subscriber State* on page 106
- *Update Subscriber State* on page 124

## **USSD States**

Brand Mobiliser delivers Unstructured Supplementary Service Data (USSD) states via Java Messaging Service (JMS) to external USSD channels.

Brand Mobiliser USSD states prompt subscribers for input, send text notifications and menu-based requests.

### **See also**

- *Send USSD Input State* on page 111
- *Send USSD Menu State* on page 112
- *Send USSD Text State* on page 118

## **Custom States**

You can develop custom application states to extend the functionality of Brand Mobiliser, and to meet client-specific requirements.

Custom states are typically developed by:

- SAP® personnel to implement client-specific requirements.
- Third parties for plug-in applications to meet client requirements.

To integrate new custom states, develop Java components using the provided APIs, and customize the product by installing custom-state bundles.

## **Input and Output Parameters**

Application states can have input and output parameters. Input parameters allow states to receive input from consumers, other states, and applications. Output parameters allow states to save values in session variables that can be used by other states or applications.

Input parameters contain the information a state requires to perform its task. Input parameters can be constant values, or values copied from a variable in the current user session.

Output parameters allow states to return values. All output parameters are available as variables.

### **See also**

- *Custom State Variables* on page 45
- *Defining Input Variables* on page 47

- *Defining Output Variables* on page 49
- *Accessing Input Variables* on page 50
- *List Variables* on page 51

### **State Machine**

A state machine defines an application process flow at runtime. During development, you can compose an application task flow visually using the Application Composer. When you activate the application, the process flow is converted to a state machine.

States are elements of a state-machine system. An application usually has many states, and can include different types of states. Each state has a previous state and a following state, unless it is the initial state or the final state. There can be only one initial state, but, depending on user interaction, there can be many final states.

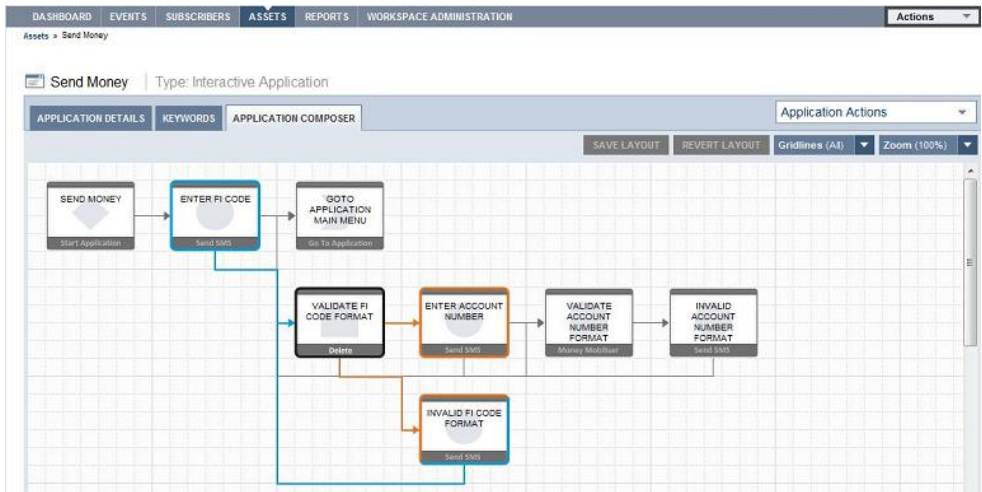
An initial state is the first state in an application, and only handles state transitions to follow-up states, based on transition rules. The initial-state, which is a base state, is Start Application. The initial state is created automatically when you create an application, and cannot be deleted. By default, the name of the initial state is the same as the name of the application.

### **Application Composer**

To visually develop applications, use the Application Composer.

The Application Composer state layout view lets you visualize the processing steps of the application task flow. You can create states and draw transitions between them. The Application Composer enables application developers to:

- Visualize states in the application using an automatic layout
- Drag and drop states to rearrange the layout
- Highlight the context, dependencies, and transitions of states
- Zoom in and out to see a complete or partial application layout
- Set the grid line type



The Layout Canvas shows the application flow, from left to right, on a grid line background. The flow consists of states (shown in boxes) and transitions that connect two states (depicted as lines with arrows). State boxes include the name of the state instance, the type, and a watermark pattern that define the state type. In complex applications, transition lines may overlap.

When you highlight a state, all of its transition lines and states they connect to are highlighted. To highlight a state, move the cursor over the state icon and left-click. The dependent states and transition lines display in different colors:

- The selected state displays a dark gray border; for example, the Validate FI Code Format state in the screen above. When you select a state, the text at the bottom of the state icon changes to **Delete**.
- States that transition to the highlighted state display a blue border and a blue transition line.
- States to which the highlighted state transitions display an orange border and an orange transition line.
- States that transition both to and from the highlighted state have borders that are half blue and half orange (dual mode); for example, the Invalid FI Code Format state in the screen above.

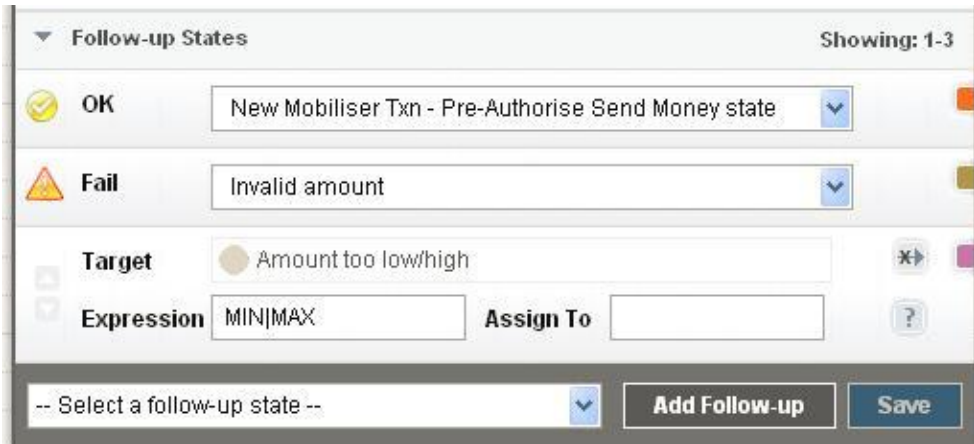
## State Transitions

Some state transitions are determined by matching regular expressions with text supplied by consumers. Other states have specific transitions that define follow-up states, which state developers define in the code.

The OK and Fail transitions do not use pattern matching; such transitions are based on states' code, and validation provided by, or events in, back-end systems. Some states do not require OK or Fail transitions. If a state does require one of these transitions, and you do not specify a follow-up state, the application terminates.

For dynamic transitions, a state's code has the option to return an expression, which provides the input to the pattern-matching mechanism. Dynamic transitions also provide a way to transition to success or failure outcomes, and may replace the OK and Fail transitions. Dynamic transitions can communicate information back to applications about certain validation problems.

This example includes an OK transition, a Fail transition, and a dynamic transition that uses the expression MIN|MAX.



**See also**

- *Controlling State Transitions with Regular Expressions* on page 7

**Controlling State Transitions with Regular Expressions**

You can control state transitions by defining regular expressions. When expressions match user-input strings, the state transitions to the follow-up state.

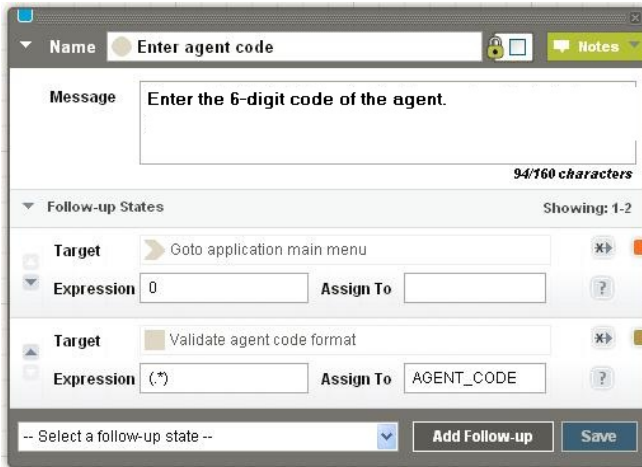
Some states expect user input to control the transition to follow-up states. Input can be provided either by consumers in response to the Send SMS state, or as dynamic output from either a Brand Mobiliser state, or a third-party custom state. Dynamic values allow external systems to communicate specific context information back to the application.

A regular expression can contain any combination of characters. The Brand Mobiliser expression tester enables you to test regular expressions during application development. Sample regular expressions are:

Regular Expression	Matches
. *	Any value in the Expression field.
(. *)	Any value in the Expression field; assigns the expression to a session variable.

In more complex cases, you can break a regular expression into multiple regular-expression groups and assign them to separate session variables.

For a complete description of regular expressions, see: <http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html>.



In the state editor, Target identifies the state that follows the current state if its Expression value matches the input. If the input matches more than one Expression value, a list of matches is created. The first entry in the list is the first matching pattern, continuing with other states in the order in which they appear in the state editor. For example, if the input is 0, the follow-up state is Goto Application Main Menu, even though 0 also matches the second expression. If the input is anything other than 0, it matches the second expression, and the value is assigned to the session variable AGENT\_CODE, because the value of Expression is surrounded by parentheses. To move an expression up or down in the Follow-up States list, use the arrows on the left side of the editor.

### Testing Regular Expressions

As you develop applications in the Application Composer, you can test regular expressions to determine whether they match alphanumeric strings.

1. In the Application Composer, select a state.
2. Click the ? icon to the right of the Assign To field for a follow-up state.  
The expression tester opens and populates Expression and Assign To fields with follow-up state values from the state editor.
3. Enter the value to test in the **Text to Test** field, and click **Test**.

The result is either:

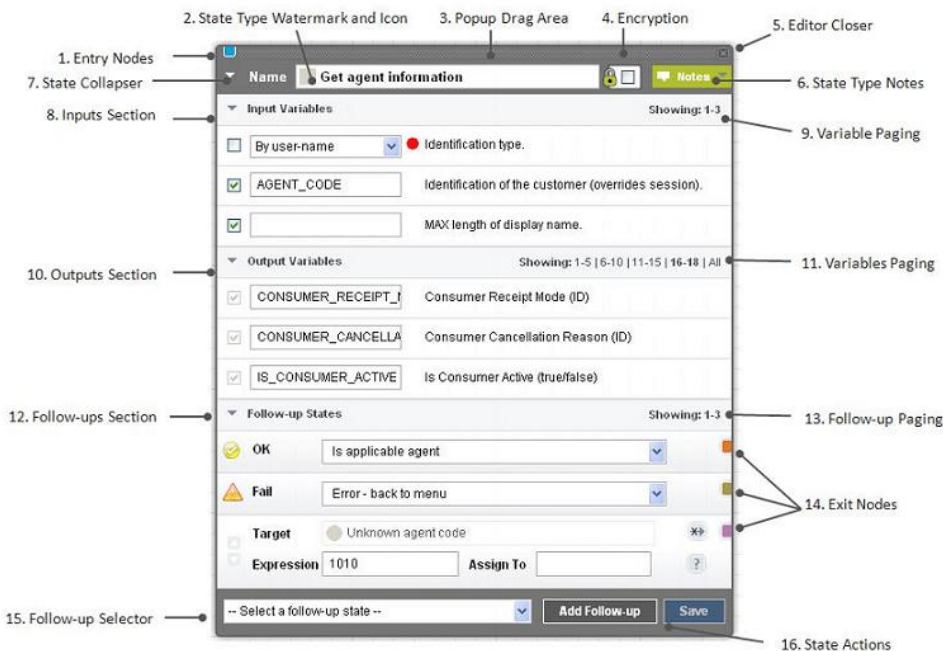
- **Match** – value in Expression field matches the value in Text to Test field.

- No Match – expression value does not match Text to Test value.

## State Editor

In the state editor, you can edit state properties, define follow-up states, test regular expressions for follow-up transitions, and detach the current state from follow-up states.

The state editor window opens automatically when you select a state in the Application Composer. Depending on the state type, the state editor displays various options, context-sensitive links, and entry fields.



State editor fields and controls are:

1. Entry Nodes – identify links to other states that transitions to this state. If you click an entry node, a state editor opens for the corresponding state. If you hover over an entry node, you see the state name with which it is associated.
2. State Type Watermark and Icon – icon associated with the state type. The watermark allows you to quickly recognize state types in the editor and in the layout view.
3. Pop-up Drag Area – you can move the state editor anywhere within the Application Composer by clicking the header and dragging.
4. Encryption – encrypts incoming and outgoing messages, which are saved in message logs.
5. Editor Closer – closes the state editor. If you have pending changes that have not been saved, you are prompted to either save or discard these changes.

6. State Type Notes – to view or edit notes that describe a state's function, input and output variables, and follow-up state transitions, click the down arrow.
7. State Collapser – shrink or enlarge the state editor.
8. Inputs Section – input variable names and values. Click the down arrow to close this section.
9. Variables Paging for Inputs Section – if a state contains more than five input variables, you can page through the others by selecting the relevant page set. To display all input variables, click **All**.
10. Outputs Section – output variable names and values. Click the down arrow to close this section.
11. Variables Paging for Outputs Section – if a state contains more than five output variables, you can page through the others by selecting the relevant page set. To display all output variables, click **All**.
12. Follow-up Section – configure, change, and test follow-up states. To collapse this section, click the down arrow.
13. Follow-up Paging – three follow-up states appear on each page. To see more follow-up states, select the relevant page set.
14. Exit Nodes – identify links to other states that this state transitions to. If you select an exit node, a state editor opens for the next state. If you hover over an exit node, you see the state name with which it is associated.
15. Follow-up Selector – select the follow-up state. All states, except the Goto Application state, allow you to add a follow-up state.
16. State Actions – **Add Follow-up State** and **Save**.

### Adding States to Applications

You can add new states in the Application Composer. When you create a new application, a Start Application state is created automatically, as the initial application state.

1. In the Application Composer, select an existing state.
2. In the state editor, expand the list of follow-up states, and select a state.
3. Click **Add Follow-up**.

The new state appears in the Application Composer. A transition line connects the current state to the new state.

A new state is automatically assigned the name New **State Type** State. Change the name, because state names must be unique.

### Editing State Properties

You can edit state properties and state transitions in the Application Composer.

1. In the Application Composer, select the state you want to edit.
2. In the state editor, configure state properties.

These changes are immediately saved to the database:



- Adding a new follow-up state
  - Adding a transition to an existing state
  - Removing a transition from an existing state
  - Moving a transition up or down in the list of follow-up states
3. For other changes, click **Save**.

### **Removing States**

In the Application Composer, you can remove states from an application. Removing a state permanently deletes the state and transition lines that are connected to it from the application.

1. In the Application Composer, select the state to remove.
2. Click **Delete**.

If you remove a state that has follow-up states, these states may be orphaned.

### **Removing State Transitions**

Removing a state transition permanently deletes the transition, but does not remove any follow-up states to which it is connected.

1. In the Application Composer, select the state with the follow-up transition you want to delete.
2. In the state editor, to the right of the Target State field, click the asterisk-arrowhead icon:



#### **Next**

To reattach orphaned states, add a new transition using the follow-up selector.

## **Developing Interactive Applications**

Brand Mobiliser interactive applications provide rich, user-interactive mobile services, and are typically invoked when mobile customers send a keyword to a preassigned short or long code.

1. In the Dashboard screen, at the bottom of the My Applications module, select **Create Interactive Application**.
2. On the Application Details tab, enter:
  - Name – the main identifier for an application. SAP recommends that you do not use duplicate names within a workspace.
  - Category – (optional) select the application category from the list. You can use categories to group applications together for managing and reporting.

## Developing Brand Mobiliser Applications

- Active From – the date and time the application becomes active, based on the server date and time.
- Active To – the date and time the application ceases to be active, based on the server date and time.
- Timeout (secs) – an interactive application establishes conversations with mobile subscribers. When a conversation starts, a unique session is established for the conversation. The session terminates (or times out) when there is no conversation for more than the number of seconds you enter here. The default value is 450 seconds (7 minutes and 30 seconds).
- Session Limit Response – the message that is sent to mobile subscribers when the application cannot start or carry on a conversation for various reasons; the most common reason being too many conversations are already taking place, exceeding the system capacity. In this case, the default message is sent to mobile subscribers. For example, the message may say “Service busy, try again in few minutes.”

3. Click **Save**.

4. (Optional) To save the application to the local file system, click **Export**.

The application is exported to a Brand Mobiliser application XML file. You can transfer the XML file to other Brand Mobiliser workspaces or instances. You can also use the file to back up the application, or store it in the source control management system.

---

**Note:** The Export button is disabled until you save an application the first time.

---

5. Add a keyword to the application.

6. Design the application task flow.

7. Activate the application.

8. Test the application.

### See also

- *Activating Applications* on page 21
- *Testing Interactive Applications* on page 23

## Adding Keywords to Applications

A keyword identifies an application within a workspace. Create at least one keyword for each interactive application.

1. Select the **Keywords** tab, and enter values for these fields.

- Add New Keyword – enter plain text or regular expressions. SAP recommends that a keyword be unique for each application in the same workspace.
- Active From – the date and time the keyword becomes active, based on the server date and time.
- Active To – the date and time the keyword ceases to be active, based on the server date and time.

2. To save the keyword, click the diskette icon.

After you save a keyword, another Add New Keyword field appears, allowing you to add another keyword.

### See also

- *Searching for a Keyword* on page 13
- *Short Codes, Long Codes, and Keywords* on page 14

## Searching for a Keyword

Keywords should be unique within a Brand Mobiliser workspace. The keyword-search tool enables application developers to see if a keyword is assigned to any applications.

If you use a regular expression to define a keyword, the keyword search tool cannot detect duplicates.

1. In the Interactive Applications window, select the **Keywords** tab.
2. Enter the keyword for which to search, and click **Search**.

If any applications in the workspace already use the keyword, this information appears on the screen:

- Used by – the application name.
- Approved – indicates whether the application is active. False means that either the application is inactive, or the application has never been activated, so the status is draft.

### See also

- *Adding Keywords to Applications* on page 12
- *Short Codes, Long Codes, and Keywords* on page 14

## Designing Application Task Flows

The key to effective application development is defining the task flows involved in modeling business processes. In the Application Composer, you can graphically design an application task flow.

The first time you open the Application Composer, you see the Start Application state. If you select the state, the state editor opens, which allows you to add follow-up states.

You can rearrange a layout by dragging and dropping state icons. To get a better view of state transitions, you may want to rearrange the layout, particularly when transition lines overlap. You can drag and drop state icons into fixed-grid positions on the canvas. The canvas does not allow free-form positions. Transition lines are automatically positioned, and you cannot move them.

- To move a state, select it, and drag it to an alternate grid position.

While moving, the state icon appears transparent, and the target grid positions are highlighted when the mouse enters the grid area.

- To delete a state, select the state, and click **Delete**.

When you delete a state, all transitions to and from other states are deleted. However, corresponding states and all of their downstream flows are not deleted. States that are not connected to other states become orphans, but they are still accessible from the follow-up state list, and you can connect them to other states.

- To save a rearranged layout to the database, click **Save Layout**.
- To revert the application layout to the last one saved in the database, click **Revert Layout**.
- To change the grid lines, expand the **Gridlines** list, and select All, Partial, or None.
- To zoom in or out, expand the **Zoom** list, and select the magnification you want to see, relative to the initial display.

If you zoom out from the default 100% view, you must reset the zoom level back to 100% before you can make any layout changes.

### See also

- *Developing Event Applications* on page 15
- *Activating Applications* on page 21
- *Creating Events* on page 17
- *Assigning Events to Applications* on page 19
- *Activating Events* on page 21
- *Testing Event Applications* on page 25

## Short Codes, Long Codes, and Keywords

A short code or long code plus a keyword identifies an interactive application within a Brand Mobiliser workspace.

Each Brand Mobiliser workspace has a unique short or long code. For incoming messages, the processing engine compares the destination MSISDN with the short or long code list to find a matching workspace. Once a matching workspace is identified, the processing engine compares the message content with keywords assigned to applications in the workspace. A workspace can contain many applications, which should all have unique keywords. At runtime, the processing engine stops when it finds the first matching keyword, and calls the corresponding application.

A short code is a special telephone number, significantly shorter than a full telephone number that can be used to address SMS and MMS messages from some mobile phones or fixed phones, and is limited to national borders. A long code is a longer number and is available internationally.

Brand Mobiliser uses short codes and long codes differently from how they are used in the mobile-operator world. Short codes are often associated with mobile services, such as Brand

Mobiliser interactive applications, and they are assigned by the mobile operator to the owner of the service.

For example, company XYZ wants to provide a mobile service for paying street-parking fines in the financial district of San Francisco. XYZ applies for an assigned short code from a mobile operator. Typically, the short code (9999) is advertised on billboards in the financial district area: “To pay parking fines with your mobile phone, text “SFpay to 9999.” When a mobile subscriber texts SFpay to 9999, the message first reaches the mobile operator. The operator, in turn, routes it to Brand Mobiliser. When Brand Mobiliser receives the message, the Brand Mobiliser Processing Engine maps the destination MSISDN (9999) to a workspace. Once the workspace is identified, the engine looks at the keyword SFpay and maps it to the corresponding interactive application in that workspace. The first matching application is chosen.

A keyword can be a simple string like “coupon,” or a regular expression. Optionally, you can associate a date range with a keyword, which controls the length of time a keyword remains active. A keyword's date range takes precedence over an application's date range: if an application's date range expires, but the keyword date range is still active, the application remains active until the keyword dates expire. When keyword dates are empty, the application defines the date range.

Best practices:

- Verify that an interactive application acting as an entry point has at least one assigned keyword.
- Use the keyword-search tool to verify that a keyword is assigned to only one application in the workspace.
- If you define a regular expression as a keyword, verify that the regular expression does not overlap with keywords that are already in use by other applications. The keyword-search tool does not work for regular expressions.

### See also

- *Adding Keywords to Applications* on page 12
- *Searching for a Keyword* on page 13

## Developing Event Applications

---

Event applications work non-interactively, such as batch processes that send campaign messages, and are typically invoked by events, such as scheduled times, system triggers, or external triggers. An event application can send outbound messages but has no user-interactive capability.

After you create and activate an event application, you can assign an event to it. You can assign an event to only one event application.

1. On the Dashboard screen, at the bottom of the My Applications module, select **Create Event Application**.
2. On the Application Details tab, enter:
  - Name – the main identifier for an application. SAP recommends that you do not use duplicate names within a workspace.
  - Category – (optional) select the application category from the list. You can use categories to group applications together for managing and reporting.
  - Active From – the date and time the application becomes active, based on the server date and time.
  - Active To – the date and time the application ceases to be active, based on the server date and time.
3. To save the application, expand Advanced Settings, and click **Save**.
4. Select the **Application Composer** tab, and define the application states and the task flow.
5. Activate the application.
6. Create an event and assign it to the application.
7. Activate the event.
8. Test the application.
9. (Optional) To export the application, expand Advanced Settings, and click **Export**.

---

**Note:** The Export button is disabled until you save an application the first time.

---

The application is exported to a Brand Mobiliser application XML file, and saved to the local file system. You can transfer the XML file to other Brand Mobiliser workspaces or instances. You can also use the file to back up the application, or store the XML in the source control management system.

### See also

- *Designing Application Task Flows* on page 13
- *Activating Applications* on page 21
- *Creating Events* on page 17
- *Assigning Events to Applications* on page 19
- *Activating Events* on page 21
- *Testing Event Applications* on page 25

## Events

A Brand Mobiliser event triggers an event application. Event applications are designed for batch processing, and are triggered by events, such as scheduled times.

You assign an event to an event application, so that when the event occurs, the application is invoked. For example, you can create a promotional event that is scheduled between November 1 and November 30. Within this event runtime, you can define event windows that specify when to invoke the event application. You can define event windows by setting start

and stop dates and times. You can also define recurring windows, for example, to occur daily, by setting start and stop times.

The event model is a container for storing configuration details and relationships, including active runtime, event windows (manual or recurring), the event application to trigger when an event window is current, and all related interactive applications.

If you assign an event to an interactive application, no one can delete the application.

## **Creating Events**

Create an event to trigger an event application.

1. In the Dashboard screen, at the bottom of the My Events module, select **Create New Event**.
2. On the Event Details tab, enter:
  - Name – the main identifier for an event. Duplicate names within a workspace are allowed, but not recommended.
  - Category – (optional) select a category from the list. You can use categories to filter events.
  - Runtime From – the date and time the event becomes active, based on the server date and time.
  - Runtime To – the date and time the event ceases to be active, based on the server date and time.
  - Description – (optional) a description of the event's purpose.
3. Click **Save**.

### **Next**

1. Set up event windows.
2. Assign the event to an active event application.
3. Activate the event.

### **See also**

- *Developing Event Applications* on page 15
- *Designing Application Task Flows* on page 13
- *Activating Applications* on page 21
- *Assigning Events to Applications* on page 19
- *Activating Events* on page 21
- *Testing Event Applications* on page 25

## **Creating One-Time Event Windows**

Create a one-time event window to define when to start and stop time an event application. At the event-window start time, the event starts its corresponding event application; the event

application stops either when it has finished processing its data, or at the event-window stop time, whichever comes first.

1. On the Events screen, select the **Event Windows** tab.
2. Click **Add New Window**, and enter:
  - Start date and time – time and date at which to start the event application.
  - Stop date and time – time and date at which to stop the event application.
  - Limit – maximum number of loopbacks to process. When used with a throttle, specify as a multiple of throttle. For example, if throttle = 60 messages per minute, specify a limit of 60, 120, or 180.
  - Throttle – maximum processing rate: number of messages per minute.
  - Resume – select to resume from the last processed item; leave unselected to restart from the beginning of the list. This is useful for states that process lists, such as the Process Subscriber state.
3. Save your settings.
4. (Optional) Create another event window, if necessary.

### Creating Recurring Event Windows

Create recurring event windows to start event applications at the same time every day, week, or month.

1. On the Events screen, select the **Event Windows** tab.
2. Click **Add New Window**.
3. Select **Switch to Recurring Mode**, and select:
  - Recurring Start Date – the date at which to start the event application.
  - Recurring Interval – the frequency at which to start the application: Daily, Weekly, or Monthly.
4. Click **Add New Window**, and enter:
  - Start time – time at which to start the event application.
  - Stop time – time at which to stop the event application.
  - Limit – maximum number of loopbacks to process. When used with a throttle, specify as a multiple of throttle. For example, if throttle = 60 messages per minute, specify a limit of 60, 120, or 180.
  - Throttle – maximum processing rate: number of messages per minute.
  - Resume – select to resume from the last processed item; leave unselected to restart from the beginning of the list.
5. Save your settings.
6. (Optional) Define additional recurring event windows, if required.



## Assigning Events to Applications

Assign an event to an event application. The event invokes the event application.

### Prerequisites

Activate the event application.

### Task

1. In the main Brand UI window, select **Events**.
2. Select the event, then select either the **Event Applications** tab or the **Interactive Applications** tab.

---

**Note:** You can assign an event to only one event application. If an assignment already exists, you can remove it. If you assign an event to an interactive application, it prevents it from being inadvertently deleted. You can assign an event to an unlimited number of interactive applications.

---

3. Click **Assign Applications**.
4. To narrow the list of applications that appear, do one of the following, and click **Search**:
  - Select **Event Applications** or **Interactive Applications**.
  - Enter the application name.
  - Expand the **Advanced** list, and select a category.
5. Select the application to assign to the event.
6. To save the assignment, select:
  - **Add to Event** – remains on the current screen.
  - **Add and Return to Event** – returns to the Events screen, and displays the Event Applications tab.

### See also

- *Developing Event Applications* on page 15
- *Designing Application Task Flows* on page 13
- *Activating Applications* on page 21
- *Creating Events* on page 17
- *Activating Events* on page 21
- *Testing Event Applications* on page 25

## Activation

Before you can run Brand Mobiliser applications and events, you must activate them.

The processing engine executes applications and events when they are in active mode. If you edit the active version of an application or an event in the Brand UI, changes are saved to an

## Developing Brand Mobiliser Applications

in-review version. Changing an in-review version does not impact the active version, until you reactivate the application or event.

Initially, the mode of activated applications and events is on-deck, and changes to active when the active from-date and time are the same as the current-date and time. Artifacts in active mode are rolled back to on-deck mode, if the active from-date and time are moved into the future.

To run some newly created artifacts—default menus, applications, and events—you must activate them. If you make any changes to one of these artifacts, you must reactivate them.

Once artifacts are activated, changes are committed and cannot be rolled back. If applications or events contain mistakes, deactivate them. For information about default menus, see *Brand Mobiliser System Administration*.

### Application Mode Transitions

After you create an application, it transitions through a series of modes. A running application is in active mode.

Initial Mode	Event/Condition	New Mode
None	Create an application	Draft
Draft	<ul style="list-style-type: none"><li>• Activate the application</li><li>• Start date is earlier than current date</li></ul>	Active
Draft	<ul style="list-style-type: none"><li>• Activate the application</li><li>• Start date is later than current date</li></ul>	On-deck
On-deck	Start date is earlier than current date	Active
On-deck	Modify the application	On-deck
On-deck	<ul style="list-style-type: none"><li>• Modify the application</li><li>• Start date is earlier than current date</li></ul>	Active in-review
Active	Modify the application	Active in-review
Active	End date is earlier than current date	Ended
Active In-Review	End date is earlier than current date	Ended

## Activating Applications

You must activate applications before you can test or run them. If you modify an active application and save changes, you must reactivate the application before changes are applied to the active version.

Applications that are currently running are in active mode. If you activate an application, but its active start time is in the future, the application mode is on-deck, and cannot be tested.

1. On the Brand UI navigation bar, select **Assets**.
2. On the Assets screen, select **Activate Applications**.
3. Click **Load Applications for Activation**.  
Applications that are in-review appear.
4. Choose either:
  - To activate a single application, select **Actions > Activate**.
  - To activate all in-review applications, select **Activate All**.

### **See also**

- *Developing Event Applications* on page 15
- *Designing Application Task Flows* on page 13
- *Creating Events* on page 17
- *Assigning Events to Applications* on page 19
- *Activating Events* on page 21
- *Testing Event Applications* on page 25
- *Developing Interactive Applications* on page 11
- *Testing Interactive Applications* on page 23

## Activating Events

Activate an event to trigger an event application.

### **Prerequisites**

Assign the event to an active event application.

### **Task**

1. In the Brand UI navigation bar, select **Events**.
2. For the event you want to activate, select **Actions > Activate**.

### **See also**

- *Developing Event Applications* on page 15
- *Designing Application Task Flows* on page 13

## Developing Brand Mobiliser Applications

- *Activating Applications* on page 21
- *Creating Events* on page 17
- *Assigning Events to Applications* on page 19
- *Testing Event Applications* on page 25

### **Deactivating Applications**

If necessary, you can deactivate or delete an application.

- To deactivate the application until a specified future date, change the active from-date to a future date, and reactivate.
- (Interactive applications only) To prevent an application from being invoked, remove the keywords, and reactivate.
- To delete an application:
  - a) Export the application.
  - b) Delete the application.

### **Deactivating Events**

If necessary, you can deactivate an event. If the event has a current event window, change the window start date to a future date, before deactivating the event.

1. In the Brand UI navigation bar, select **Events**.
2. Select the event you want to deactivate.
3. On the **Event Details** tab, change the active from-date to a future date.
4. (If necessary) Reset the event window start date and time.
5. Save your changes and reactivate the event.

The event remains inactive until the specified future date.

## **Testing Applications**

---

Test Brand Mobiliser applications using the built-in application simulator.

To access the Simulation page, expand the Actions list on the right side of the navigation bar, and select **Simulate Application**. You can test interactive applications and event applications. Select the tab that corresponds to the application type you want to test.

You can also test applications using either a Short Message Peer-to-Peer (SMPP) test harness or a Java Message Service (JMS) test harness; these methods are typically used by custom-state developers and advanced system administrators.

## Testing Interactive Applications

Test an interactive application in the current workspace by simulating incoming and outgoing messages.

### Prerequisites

Activate the application.

### Task

1. On the **Interactive Application** tab of the Simulation page, enter:
  - Customer MSISDN – numeric value. Brand Mobiliser uses the MSISDN to either create a new session or find the existing session. If the application being tested has states that interface with a back-end system, such as Money Mobiliser, enter an MSISDN that identifies a customer in that system.
  - Workspace Short | Long Code – select from the list.
  - Message Encoding – accept the default, or select Unicode.
  - Message Text – a valid keyword for the application.

2. Click **Send to Brand Mobiliser**.
3. To see Brand Mobiliser responses, click **Reload Message Log**.

If the application calls an external Web service, responses may take longer than the page-refresh time.

### See also

- *Developing Interactive Applications* on page 11
- *Activating Applications* on page 21

### Sample Interactive Message Log

An interactive-application message log shows a sequence of consumer interactions with Brand Mobiliser.

For each message, the logs displays:

- Send Date – the date and time the message was sent.
- ACK and ACK Date – whether an acknowledgment is requested from the short message service center (SMSC) or the SMS gateway, and the date and time the acknowledgment was received.
- Direction – message direction, IN or OUT; IN messages come from customers; OUT messages are Brand Mobiliser responses.
- Sender – sender's identification number. For IN messages, the number is the customer's MSISDN; for OUT messages, it is the workspace short or long code.
- Application – name of the application that processed the message. A Brand Mobiliser application can call other applications, which are identified in the log.
- Receiver – receiver's identification number. For IN messages, the number is the workspace short or long code; for OUT messages, it is the customer's MSISDN.

Simulation

INTERACTIVE APPLICATION
EVENT APPLICATION

Customer MSISDN

Workspace Short | Long Code

Message Encoding

Message Text

Send to Brand Mobiliser
Send to Customer

Reload Message Log

Send Date	ACK	ACK Date	Direction	Sender	Application	Receiver	Message
Fri Sep 14 19:27:18 GMT 2012	requested	N/A	OUT	44778	1 Secured Main Menu	+1925	TXMS: 0 entry found. [0: Menu]
Fri Sep 14 19:27:18 GMT 2012	N/A	N/A	IN	+1925	1 Secured Main Menu	4477f	1
Fri Sep 14 19:27:04 GMT 2012	requested	N/A	OUT	4477f	0.4 SUB - Credential Verification (...)	+1925	WALLET MAIN. Reply: 1: View Transactions 2: Transfer Money 3: Request Money 4: Airtime Topup 5: Pay Bills 6: Manage Account 9: Bye
Fri Sep 14 19:27:03 GMT 2012	N/A	N/A	IN	+1925	0.4 SUB - Credential Verification (...)	4477f	12
Fri Sep 14 19:26:51 GMT 2012	requested	N/A	OUT	4477f	0 START - Customer Verification	+1925	WALLET. Please send MPIN. [0. Menu]
Fri Sep 14 19:26:51 GMT 2012	N/A	N/A	IN	+19253	0 START - Customer Verification	4477f	wallet

## **Testing Event Applications**

To test event applications, invoke the triggering event. Event applications are linked to events that occur at times defined by their event windows.

1. On the Simulation page, select the **Event Application** tab:
  - Event Name – select from the list.
  - Resume From Last – accept the default value, false. If set to true, and if the previous test did not exhaust the subscriber list, the application resumes from the last subscriber.
  - Throttle – enter the maximum processing rate: number of messages per minute.
  - Limit – enter the maximum number of loopbacks to process. When used with a throttle, specify as a multiple of throttle. For example, if throttle = 60 messages per minute, specify a limit of 60, 120, or 180.
  - Event Threads – specify the number of threads to use to run the simulation. Change this value to test performance with different numbers of threads.
  - End Date – specify to keep the application from overrunning.
2. Click **Simulate Event**.
3. To see messages, click **Reload Message Log**.

Depending on the number of subscribers, you may need to reload the log multiple times to see all the messages.

### **See also**

- *Developing Event Applications* on page 15
- *Designing Application Task Flows* on page 13
- *Activating Applications* on page 21
- *Creating Events* on page 17
- *Assigning Events to Applications* on page 19
- *Activating Events* on page 21

### Sample Event Message Log

The Utility Notification event application generates messages that appear in the message log.

The screenshot shows a 'Simulation' window with two tabs: 'INTERACTIVE APPLICATION' and 'EVENT APPLICATION'. The 'EVENT APPLICATION' tab is active. It contains several configuration fields: 'Event Name' (Service Outage Event), 'Resume From Last' (false), 'Throttle [messages/min]' (empty), 'Limit' (5), 'Event Threads' (empty), and 'End Date' (9/14/12 23:50). A 'Simulate Event' button is present. Below the configuration is a 'Reload Message Log' button and a table with the following data:

Send Date	ACK	ACK Date	Direction	Sender	Application	Receiver	Message
Fri Sep 14 23:46:57 GMT 2012	requested	N/A	OUT	8899	Service Outage Notification	+19253371758	RI +19253371758: Half-hour service outage in 2 hours.
Fri Sep 14 23:46:57 GMT 2012	requested	N/A	OUT	8899	Service Outage Notification	+19253371757	RI +19253371757: Half-hour service outage in 2 hours.
Fri Sep 14 23:46:57 GMT 2012	requested	N/A	OUT	8899	Service Outage Notification	+19253371756	RI +19253371756: Half-hour service outage in 2 hours.

**See also**

- *Utility Notification Event Application* on page 31

## Importing Applications

You can import application XML files that were previously exported from Brand Mobiliser, and you can create applications from Quick-Start template files that are installed with Brand Mobiliser.

**See also**

- *Exporting Applications* on page 27

### Importing Application XML Files

Import a Brand Mobiliser application by uploading the XML file that contains the application configuration. XML configuration files are created when you export applications from Brand Mobiliser.

If you import a single application that links to other applications, create the linked-to applications before you import. If you import a single application that contains circular references, which are common in menu-based systems, you must manually relink applications before you can run them.



To import a group of dependent applications, first export them as a group, so all the dependent applications are in one export file. When you import a group of applications from a single export file, all interdependent links and references are maintained.

1. In the Brand UI, select **Assets**, then select **Create Asset**.
2. Under Upload Applications From Existing Files, click **Browse**, and select the application file.
3. Enter a name for the application.
  - If the file contains a single application, the application name is replaced.
  - If the file contains more than one application, the new application name is prepended to all applications. For example, if the file contains two applications, Test1 and Test2, and you enter `NewName` as the new application name, the uploaded applications are named `NewName-Test1` and `NewName-Test2`.
4. Click **Upload**.
5. To edit application details, select **View Application Details**.

### Creating Applications from Templates

Brand Mobiliser includes a set of application templates that you can upload and run.

1. In the Brand UI navigation bar, select **Assets**.
2. Select **Create Asset**.
3. Choose a template from the list, and click **Create**.  
The template is installed, and names of the template applications appear.
4. Select **Application Details**.

After you create an application, you can run it or modify its details.

#### **See also**

- *Developing Quick-Start Templates* on page 75

### Exporting Applications

You can export applications to make backup copies, or to move applications to other Brand Mobiliser installations. If you export an application, it is saved in an XML file.

#### **See also**

- *Importing Applications* on page 26

## Exporting a Single Application

Exporting a single application creates an XML file that contains the application configuration.

1. In the Brand UI, navigate to the **Application Details** tab for the application you want to export.
2. Click **Export**.  
The application is exported to a file called `appFlow.xml` in the `Downloads` directory.

If the application you export contains references to other applications through either the Goto Application state or the Application Call state, details of the called applications are included in `appFlow.xml`; however, interapplication links may not be reestablished when you import the file. To maintain links and dependencies between applications, export them as a group.

## Exporting a Group of Applications

Exporting a group of applications maintains links and dependencies between applications.

1. In the Brand UI, navigate to the **Assets** page.
2. Select the check box to the left of each application you want to export.
3. Click **Group Export Applications**.  
A file called `groupedFlow.xml`, which contains all the exported application configurations is created in the `Downloads` directory.

## Sample Applications

---

Mobiliser Platform offers a customizable way to more efficiently manage financial services. It allows customers to redeem vouchers on any phone, remit money domestically, pay bills automatically, and manage their accounts remotely.

### Cash-Out Interactive Application

Use SMS to interact with the Cash-Out application. Brand Mobiliser manages a unique user session that maintains the context of the conversation between the user and the application.

The Cash-Out application comprises multiple interactive applications. The applications are linked by either Goto Application states, in which control is passed to referenced applications, or Application Call states, in which case control moves temporarily to the referenced application, before returning to the application that called it.

A complete mobile service is created from multiple interactive applications that are validated with a customer's MSISDN. Although there is no Brand Mobiliser internal customer list, back-end systems—such as Money Mobiliser—can validate customers. The Cash-Out application assumes a valid customer session exists.

Once an application has validated a customer, it is typical to offer a series of SMS menus, from which customers can select. By default, the Cash-Out application contains one menu option that is related to the mobile financial services that are offered to customers.

The Cash-Out application:

1. Requests the account from which to withdraw cash.
2. Requests the code of the customer support agent with whom to perform the transaction.
3. Requests the transaction amount.
4. Validates and preauthorizes the transaction by verifying sufficient funds in the account, amount limits, and the agent's SVA.
5. Requests an account PIN, or transaction confirmation.
6. Sends money to the agent.
7. If a transaction fails, requests a solution to validation problems.

## Cash-Out Application State Editor

In the Cash-Out application, the Get Wallet Menu state sends a menu to customers via SMS.

The screenshot displays the 'Cash Out Process' state editor. The main canvas shows a flowchart with three states: 'CASH OUT PROCESS' (Start Application), 'GET WALLET MENU' (Delete), and 'ENTER AGENT CODE' (Send SMS). An 'ERROR - BACK TO MENU' (Send SMS) state is also visible. A detailed configuration panel for the 'Get Wallet Menu' state is open, showing message content, input variables, output variables, and follow-up states.

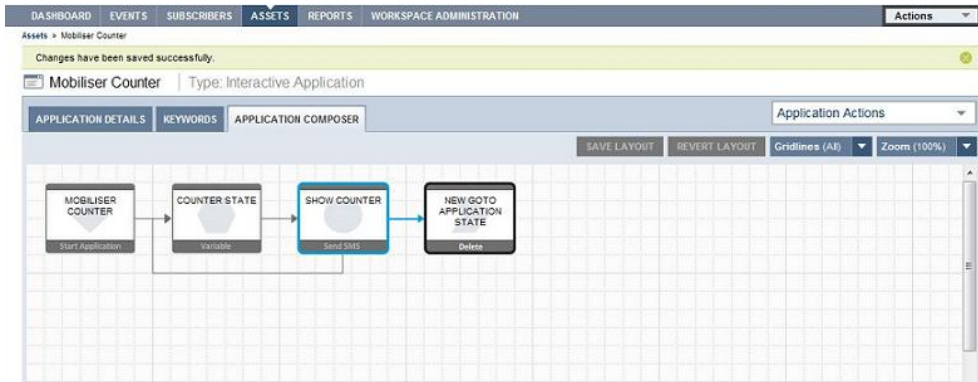
**State Configuration: Get Wallet Menu**

- Name:** Get Wallet Menu
- Message:** Please select the source account: (33/760 characters)
- Input Variables:**
  - Select an entry -- (Identification type)
  - Identification of the customer (overrides session)
  - 0,41,42 (List of PI types. Example: 12,123,45)
  - (List of PI classes. Example: 1,2,3)
  - (Max count of PIs)
- Output Variables:**
- Follow-up States:**
  - OK: Enter agent code
  - Fail: Error - back to menu
- Target:** Enter agent code
- Expression:** \*

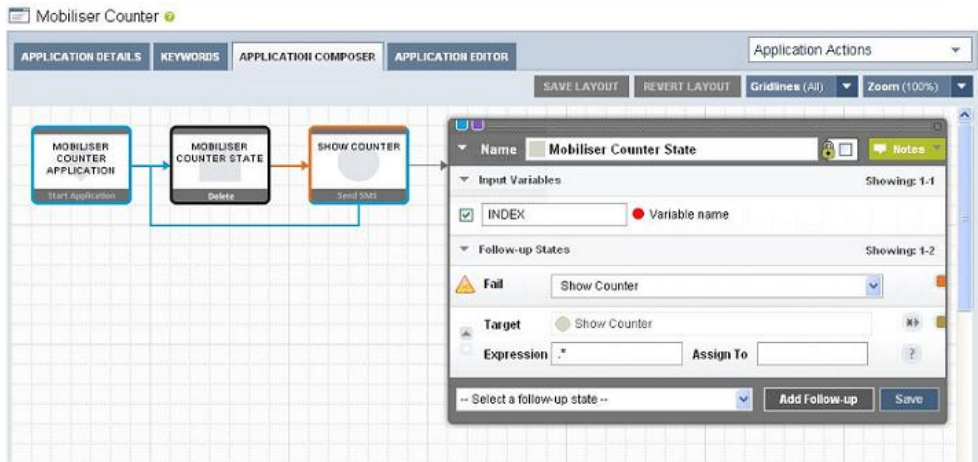
## Mobiliser Counter Interactive Application

The Mobiliser Counter sample application increments a session variable, displays the value, then either increments the value again, or exits.

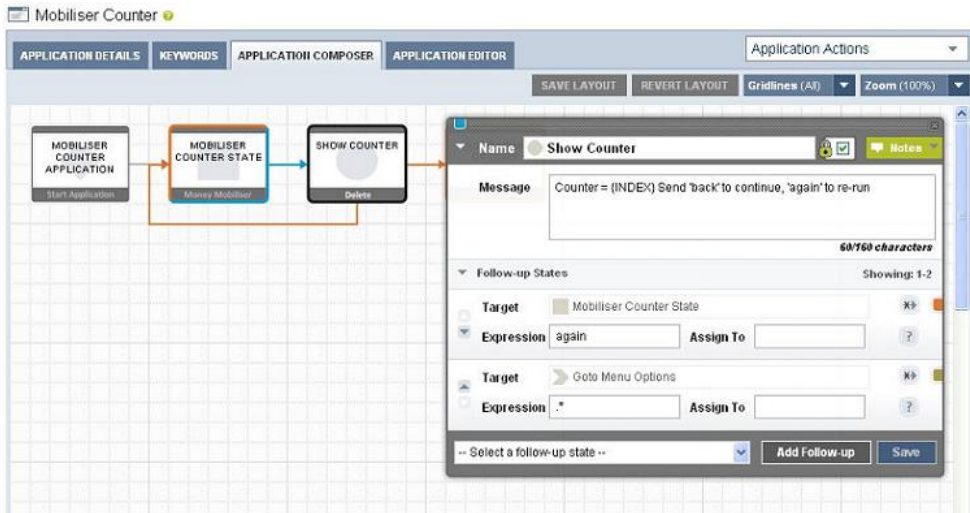
You can develop the Mobiliser Counter application in the Application Composer.



The session variable **INDEX** is used as the counter variable. This variable is dynamically substituted into the text sent to mobile consumers.



If consumers respond with the keyword "again," the application loops back to the Mobiliser Counter state. Any other input causes the application to exit.



## Utility Notification Event Application

Event applications are designed for task flow or batch processing, and are typically invoked by events, such as scheduled times, system triggers, or external triggers.

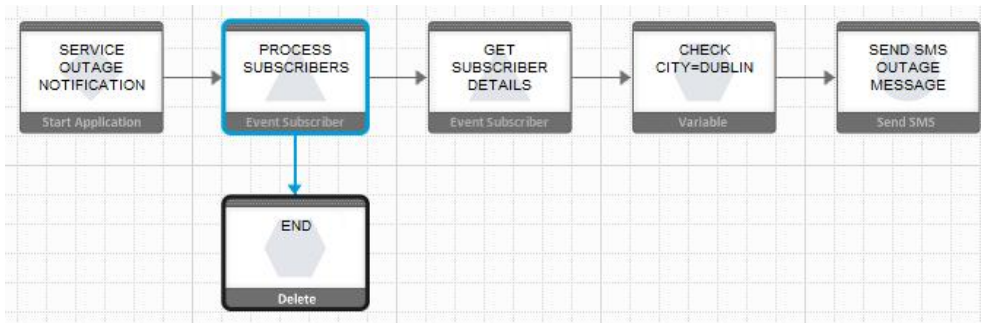
For example, Brand Mobiliser applications can provide end-to-end solutions for utility companies. A common use case includes:

- Self-registration – register telephone numbers using SMS; for customers who did not provide their number when signing up with the company.
- Self-services – such as looking up usage history and status of move-in activation, reporting issues, and finding offices.
- Notifications – set up notifications for overdue payments, high usage, service-outage alerts, summer-savings awareness, and so on.
- Engagement – enables customers who receive notifications to reply. For example, if customers respond to overdue-payment notifications, they automatically receive 1–2 days extension; they can also authorize automatic payments.

In this example, the company's customer relationship management (CRM) system generates a list of subscribers who have opted to receive outage notifications. The list contains customer telephone numbers (MSISDNs) and cities for which an outage-notification service is provided. This list is uploaded to Brand Mobiliser subscriber storage. When a service outage is planned for the city of Dublin, the Process Subscriber state retrieves subscribers from the list. For each subscriber:

1. Get Subscriber Details retrieves subscriber attributes (city).
2. Check City=Dublin filters out customers who are not in Dublin.
3. Send SMS Outage Message sends a message to Dublin customers.

## Developing Brand Mobiliser Applications



Invoke the application, by assigning it to an active event, and creating an event window. Event windows can be one-time or recurring. This application has a one-time event window.



An alternative to manually uploading subscribers to the database is to use an event application to fetch subscribers from the system of record, and use batch processing to upload and store them in the database.

### See also

- *Sample Event Message Log* on page 26

# Developing Custom Application States

Custom state development using the State SDK is a Java development task you can perform with or without a development IDE, such as Eclipse or NetBeans. After you develop and deploy custom states, you can use them to develop applications.

Before proceeding with custom state development, verify that:

- The development environment meets system requirements in the *Brand Mobiliser Release Bulletin*.
- Brand Mobiliser is installed on the development machine. Brand Mobiliser is required to access State SDK bundles for custom state development, and to deploy and test custom states through the development process.

Third-party software mechanisms that custom states can use include:

- Spring Framework – for application context and dependency injection.
- Spring Dynamic Modules (Spring DM) – for abstracting OSGi mechanisms.
- OSGi Services – for software-service publication and consumption.
- OSGi Configuration Admin – for container-based configuration of services and components.

## Application Life Cycle

---

Applications run in the Brand Mobiliser Processing Engine (processing engine) runtime container and are managed by the processing engine. Once deployed to the runtime container, applications can be invoked by either incoming messages or events. Events can be generated by the system, a scheduled time, or a call from an external Web service.

### *Starting or Restarting an Application*

For a newly started application, a new session is created, and the Application Start state is executed. Sessions are based on a consumer's MSISDN, which is typically the mobile telephone number from which the message is sent. The Application Start state is created automatically for new applications, and cannot be removed. This state performs initialization prior to executing the application. The Application Start state is typically followed by at least one state. For example, if an interactive application is invoked by an incoming message, the Application Start state processes the incoming message, and routes it to the appropriate follow-up state, based on the message value. The Application Start state can also filter messages, and save incoming message values in session attributes.

If you restart an application, the existing session is reactivated, and all session attributes are available to the application. The application continues from the last active state.

### *Executing the Current Application State*

The processing engine executes the current application state, calling either `processMessage` or `processState`; these methods contain state-specific logic.

The processing engine calls:

- `processMessage` to reactivate a state, when an external event occurs for which the state is waiting.
- `processState` when another state activates the current state through a follow-up transition.

### *Processing an Incoming Message*

If a state is reactivated by a call to its `processMessage` method, the state processes the incoming message.

For example, State 1 —> Send SMS state —> State 3. When the flow reaches the Send SMS state, a message is sent out and the flow waits for a response. When the response arrives, the processing engine calls the Send SMS state's `processMessage` method to reactivate the state. The state processes the message, finds the follow-up transition that matches the incoming message, and returns the follow-up transition state. For example, if the follow-up state is State 3, the processing engine sets the current state to State 3, and begins executing it.

### *Processing State Logic*

When a state is activated by a follow-up transition, the processing engine calls the `processState` method, which contains the core logic of the state. If the state needs to call an external Web service, you implement the call in the `processState` method.

States do not return objects from the `processState` method. Instead, they set flags using the helper object `SmappStateProcessingAction`, which is an input parameter to the method. For example, if the state-logic processing is successful, the state calls `continueProcessing(followUpState)`, passing the name of the follow-up state as `followUpState`.

The processing engine sets the current state to the value of **followUpState**, and executes the current state.

To determine the follow-up state, you can call either of two methods provided by the utility class `StateUtils`, which is included in the State SDK:

- `determineFollowingSmappStateFromPattern`
- `determineFollowingSmappStateFromTransitionType`

In addition to calling `continueProcessing`, states can call:



- `terminateProcessing` – if a severe error occurs and the application must be terminated.
- `waitForMessage` – if the state sends a message and must wait for the response.

### *Terminating Conditions*

The processing engine continues through the application flow until it meets one of these terminating conditions:

- No follow-up transition
- Call to `terminateProcessing`
- Call to `waitForMessage`

The first two conditions terminate the application. A call to `waitForMessage` pauses the application until a response is received, and the session hibernates. When the response message arrives, the life cycle restarts.

For event applications, if the processing engine encounters no follow-up transition, it checks the preconfigured terminating criteria to determine whether to stop, or keep the session alive and generate a callback to repeat from the Application Start state.

## **Developing and Deploying Custom States**

---

Develop and deploy custom states to extend the functionality of Brand Mobiliser, and to meet client-specific requirements.

1. Develop a custom state by extending either:
  - `SmappStatePlugin` class – for most states.
  - `AbstractDynamicMenu` class – for menu states.
2. (Classes that extend `SmappStatePlugin` only) Implement the state logic.
3. Add custom state information.
4. Define custom state variables.
5. Set up Apache Maven.
6. Build and deploy a custom state bundle.

### **Extending the SmappStatePlugin Class**

You can simplify custom-state development by extending the `SmappStatePlugin` class.

If you develop a custom state by extending the `SmappStatePlugin` class, you must:

- Implement the state logic.
- Provide the state information: ID, name, revision number, and usage notes.
- Specify the input attributes.

## Developing Custom Application States

- Specify the output attributes.
- Customize the state follow-up transitions, if they are different from the default transitions.

### See also

- *Sample Custom State* on page 80
- *Sample GetMyWeather State* on page 78
- *Implementing State Logic* on page 41

### **StatePlugin Interface**

You can use the `StatePlugin` interface to develop Brand Mobiliser application states.

The `SmappStatePlugin` class is a base abstract class that implements the `StatePlugin` interface. Most custom states should extend `SmappStatePlugin`, which provides basic implementations that are common to most custom states, as well as helper methods that are commonly used in state implementations.

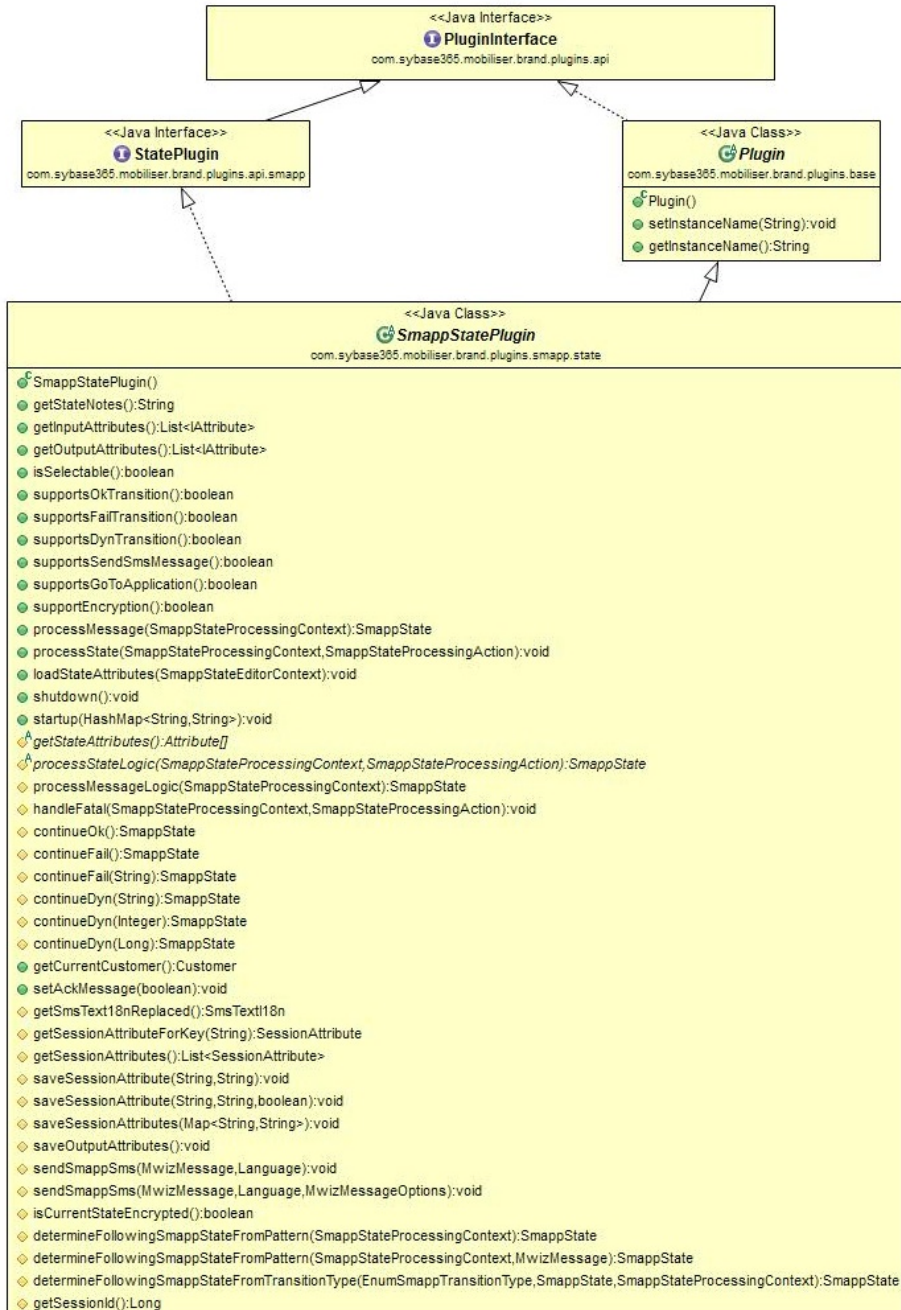
Two important methods in the `StatePlugin` interface are `processMessage` and `processState`, which are integral parts of application life cycles. Some of the methods in the `StatePlugin` interface customize the state editor, for example, `supportsOkTransition` and `getStateNotes`.

If a custom state extends the `SmappStatePlugin` class, implementing the class is simplified significantly. Instead of implementing both `processMessage` and `processState` methods, you can focus on adding business logic to the `processStateLogic` method. This is sufficient in most custom-state implementations.

---

**Note:** Do not extend the abstract class `Plugin`. Instead, extend `SmappStatePlugin`.

---



### **PluginInterface Interface**

If you develop a custom state by extending the `SmappStatePlugin` class, it implements the `PluginInterface` interface.

Plug-in components must have at least one class that implements the `PluginInterface`. Components that implement `PluginInterface` are automatically loaded into the messaging server and started. During start-up, the server calls the `startup` method of the implementing class, which allows the class to perform any necessary setup.

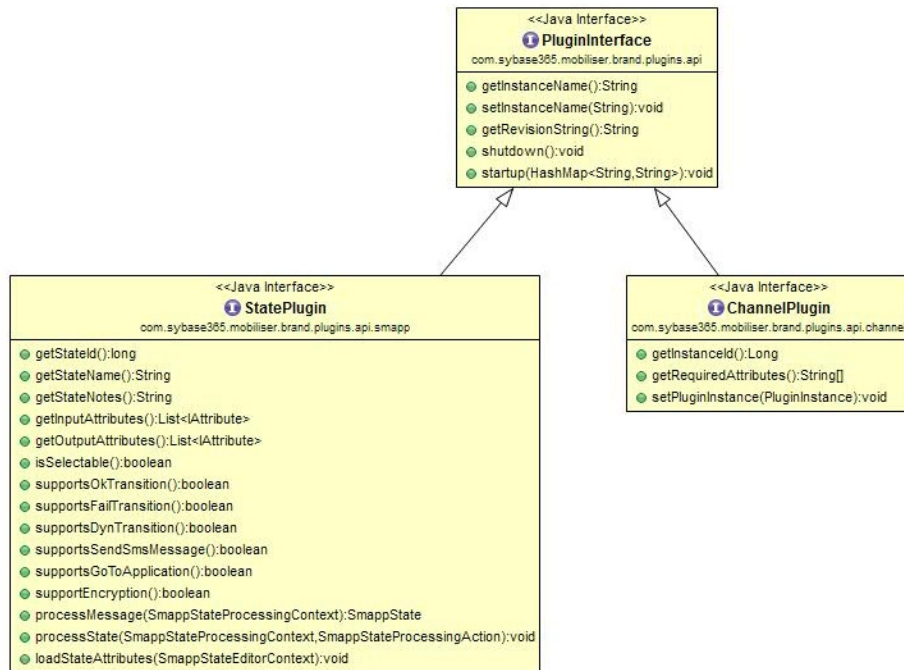
`PluginInterface` methods are:

- `getInstanceName():String`
- `setInstanceName(String):void`
- `getRevisedString():String`
- `shutdown():void`
- `startup(HashMap<String,String>):void`

The `shutdown` method is called when the server is shutting down, giving the implementation a chance to perform housecleaning, such as persisting cache data.

`getInstanceName`, `setInstanceName`, and `getRevisedString` are placeholders only. The component must implement the appropriate functionality.

`StatePlugin` and `ChannelPlugin` implementations extend `PluginInterface` and define their specific interfaces. You can use `StatePlugin` APIs to develop custom states. The `ChannelPlugin` interface is reserved for SAP internal development only.



## Extending the AbstractDynamicMenu Class

Many SMS and Unstructured Supplementary Service Data (USSD) applications rely on menus to receive consumer responses. Menus reduce the potential for response errors, because they are numbered lists.

The `AbstractDynamicMenu` class simplifies the development of custom-menu states that extend the class, because they inherit:

- A list of menu items
- Menus and indexes that are generated automatically and recalculated on each page
- Methods to send menus as SMS messages
- These variables:
  - **Show Exit Menu** – an input variable that specifies whether to allow recipients to exit the menu.
  - **Variable Name of the Selected Key** – an output variable representing the menu selection, which is stored as a key-value pair object. Key is the unique key of the menu item, which may be used later in the application.
  - **Variable Name of the Selected Value** – an output variable that represents the value of the selected key.

Custom states that extend the `AbstractDynamicMenu` class must implement these methods:

- `constructMenuList()` – gets the menu list.
- `init()` – initializes the state.
- `getStateAttributeList()` – gets the list of attributes.
- `saveSessionVariables()` – explicitly saves session variables.

Message recipients can select from lists, and reply using index numbers. If a menu has more than four items, it includes a pagination option, which displays the next four items in the list. On the last page, selecting the pagination option returns to the first page. Selecting the exit option abandons a list without a response; the application task flow determines the follow-up transition. To force recipients to choose an item from the list, you can disable the exit option.

In a typical custom-state implementation that extends the `SmappStatePlugin` class, you implement state logic in the `processStateLogic` method. However, when you extend the `AbstractDynamicMenu` class, both `processStateLogic` and `processMessageLogic` methods are implemented by the abstract class. These methods contain the menu processing logic, and are declared as `final`, so they cannot be overridden.

### See also

- *List Variables* on page 51
- *Sample Custom-Menu State* on page 82

### **AbstractDynamicMenu Life Cycle**

The life cycle of the `AbstractDynamicMenu` class is based on the life cycle of the `SmappStatePlugin` class; however, there are slight differences in menu functionality.

If you extend the `AbstractDynamicMenu` class, it implements the `processMessageLogic` method and the `processStateLogic` method.

1. The `processStateLogic` method calls the `init` method.
2. `processStateLogic` calls both the `constructMenuList` and `saveSessionVariables` methods.
3. The `SmappStatePlugin::getStateAttributes` method calls `getStateAttributeList`, which aggregates the attributes returned by the method with attributes defined in the `AbstractDynamicMenu` class, such as the input exit-menu item and the output key-value pair.
4. An `AbstractDynamicMenu` state is initially activated as a follow-up transition from a previous state, so the processing engine calls its `processStateLogic` method. The `init` and `constructMenuList` methods are called sequentially to initialize and construct the menu. Eventually, the menu is sent as an SMS message, and the processing engine waits for the response. The consumer selects a menu item.

5. If `constructMenuList` returns only a single item, the state immediately calls `saveSessionVariables`, and proceeds with the default dynamic follow-up transition. You can customize the state's default behavior by overriding the `continueWhenSingleEntry` method.
6. When a response arrives, the processing engine calls the state's `processMessageLogic` method, which calls `constructMenuList` to assemble the menu and interpret the selected menu item. If the selection is a valid menu item, `saveSessionVariables` is called. The state prepares the selected-item details for output, and proceeds with the follow-up transition, as returned by the `saveSessionVariables` method. If null is returned, the default OK follow-up transition is used.

## **Implementing State Logic**

If you extend the `SmappStatePlugin` class, implement state logic in the `processStateLogic` method. If you extend the `AbstractDynamicMenu` class, the abstract class implements the state logic.

At runtime, the processing engine calls a state's `processState` method, which in turn calls `processStateLogic`. The `processState` method is implemented by the `SmappStatePlugin` abstract class.

The `processStateLogic` method signature is:

```
protected SmappState processStateLogic(
    SmappStateProcessingContext context,
    SmappStateProcessingAction action)
    throws MwizProcessingException, DBException;
```

The `processStateLogic` input parameters are:

- `SmappStateProcessingContext` – provides access to resources, such as data-access objects for session variables.
- `SmappStateProcessingAction` – signals to the processing engine that there is to be additional processing.

### **See also**

- *Extending the `SmappStatePlugin` Class* on page 35

### **SmappStateProcessingContext**

The processing engine `SmappStateProcessingContext` provides access to resources, such as session variables and the subscribers data store.

You can use the `SmappStateProcessingContext` to share resources between the processing engine and the state; however, in most state implementations, this is unnecessary.

---

**Note:** Do not alter `SmappStateProcessingContext`.

---

## Developing Custom Application States

You can use these `SmappStateProcessingContext` methods:

- `getStateDao` – inserts, updates, or deletes session variables.
- `getSubscriberDao` – accesses the subscribers data store. Also used by some Brand Mobiliser built-in states.
- `isAckMessageRequested` – queries whether an acknowledgment is requested.
- `setAckMessageRequest` – specifies whether an acknowledgment is requested.
- `isCurrentStateEncrypted` – queries whether state data is encrypted.

The following resources are available for read-only access, and include no API support. Do not access these resources directly, or make any changes. If you have special requirements, consult with SAP support services.

- `client`
- `session`
- `clientMsisdn`
- `currentState`
- `customer`
- `langDefault`
- `matchingPattern`
- `mr`
- `msg`
- `newSession`

Do not use the following methods or resources; doing so may result in errors or unexpected application behavior:

- `getlangRequest`
- `updateSession`
- `cacheMgr`
- `outgoingQueue`

### **SmappStateProcessingAction**

The `SmappStateProcessingAction` class controls state and application processing. Use it to signal the processing engine that further processing is intended.

The processing engine recognizes three signaling actions: continue, wait, and terminate, which you can send by calling:

- `continueProcessing (SmappState)` – continues execution to the specified follow-up state. Causes an infinite loop if the follow-up state is the same as the calling state. Termination must be handled within the state.
- `waitForMessage ()` – pauses execution and waits for a response, then continues execution to the specified follow-up state.



- `terminateProcessing ()` – terminates the application.

States that extend the `SmappStatePlugin` class, implementing logic inside the `processStateLogic` method need not explicitly call `continueProcessing` or `terminateProcessing`. The same functionality is accomplished by returning the follow-up state from the `processStateLogic` method. For example, instead of calling `continueProcessing`, return the follow-up state using one of the helper methods:

- `continueOk()`
- `continueFail()`
- `continueDyn()`

To terminate processing, states should call `continueFail`, and let the state-editor configuration determine what to do. If the state is not configured to forward `continueFail` calls to a follow-up state, the application automatically terminates.

---

**Note:** If a state calls `waitForMessage` before it returns null from the `processStateLogic` method, the application does not terminate, because the state is waiting for a response. For this reason, SAP recommends that you do not let states return null.

---

To enable states to send messages and wait for replies before they continue processing, call `waitForMessage`.

To display a message control in the state editor, call `supportsSendMessage`.

## Custom State Information

State information includes an ID, a name, a revision number, and usage notes. The name and usage notes are metadata that the state editor shows in the Application Composer.

For a custom state, you can explain its purpose and functionality as state notes, which appear in the state editor.

```
@Override
public String getStateNotes() {
    StringBuilder sb = new StringBuilder();
    sb.append("A sample state. When executed, it checks for ");
    sb.append("an entered Postal/ZIP Code, and returns the ");
    sb.append("weather report for that area.\n\n");
    sb.append("Use the following follow up states:\n ");
    sb.append("- OK: Weather report for the area was found\n ");
    sb.append("- FAIL: Unexpected error\n ");
    sb.append("- Dyn -1: Area code entered was not valid\n ");
    sb.append("- Dyn -2: No weather report for the area\n ");
    return sb.toString();
}
```



The revision number is a prerequisite for any plug-in component, as specified in the `PluginInterface` class. It identifies a version, and sets the plug-in number. `getRevisionString()` can return any String value.

```
@Override
public String getRevisionString() {
    return "1.0.0";
}
```

The state ID is a unique identifier for the state. Each state must have a unique ID stored in the database for each installation in which the state is used. This unique value allows the state to be resolved to the same type across installations.

```
private static long STATE_ID = 600000L;

@Override
public long getStateId() {
    return STATE_ID;
}
```

For custom states, assign unique ID values between 600,000 and 999,999. Values between 0 and 599,999 are reserved for Brand Mobiliser use.

## Custom State Variables

You can define input and output variables for custom states. Variables are used as both metadata in the state editor, and as runtime objects for storing session variables.

In the `GetMyWeather` sample custom state, one input variable (Zip or Postal Code) and one output variable (Your Weather Synopsis) are defined in the code, and appear in the state editor view.

```
// Define input variable

private static final TextBoxAttribute inPostCode =
    new TextBoxAttribute("POSTCODE", "Zip or Postal Code", false);

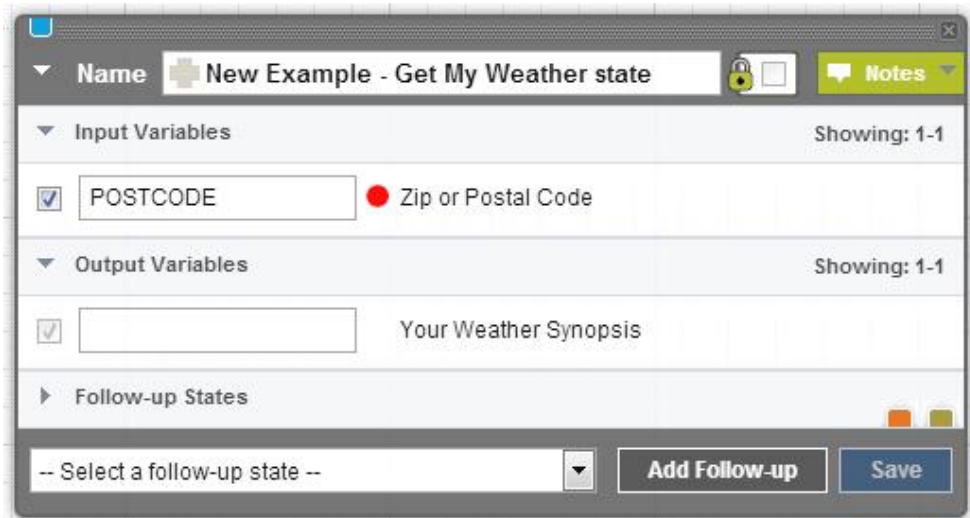
// Define output variable

private static final OutputAttribute outWeather =
    new OutputAttribute("WEATHER", "Your Weather Synopsis");

private static Attribute[] stateAttr;

static {
    stateAttr = new Attribute[] {inPostCode, outWeather};
}

@Override
protected Attribute[] getStateAttributes() {
    return stateAttr.clone();
}
```



`getStateAttributes` is an abstract helper method that the `SmappStatePlugin` class implements. It aggregates both input and output variables. The base class derives the required `getInputAttributes` and `getOutputAttributes` methods from `getStateAttributes`, based on the attribute-type class. The state editor uses the attribute array that the `getStateAttributes` method returns to render input and output variables. The `saveOutputAttributes` method saves output attributes from the attribute array.

All variables (input and output) have input controls that appear on the state editor. The `public String getText()` method returns the text from input controls.

### See also

- *Input and Output Parameters* on page 4
- *Defining Input Variables* on page 47
- *Defining Output Variables* on page 49
- *Accessing Input Variables* on page 50
- *List Variables* on page 51
- *Sample GetMyWeather State* on page 78

### Variables for Troubleshooting

When you develop custom states, include error output variables that can help you troubleshoot problems in the production environment.

To facilitate debugging, include output variables in the state code for an error message, a unique error ID, and a service code. If the state calls an external Web service, for example, the Web service can return a code in the service-code output variable.

```
// Define output variables

private static final OutputAttribute outErrMsg =
    new OutputAttribute("ERR_MSG", "Error Message");
private static final OutputAttribute outErrUUID =
    new OutputAttribute("ERR_UUID", "Error Unique ID");

private static final OutputAttribute outSvcCode =
    new OutputAttribute("SVC_CODE", "Service Code");

// some code omitted here...

@Override
protected SmappState processStateLogic(...)
{
    // Logic implementation

    try {
        // Reset the error output variable
        outErrMsg.setHoldValue("");
        outErrUUID.setHoldValue("");
        saveOutputAttributes();
    }
}
```

```

    return continueOk();
}
catch (Exception ex) {
    String uuid = UUID.randomUUID().toString();
    log.error(ex.getMessage() + " [UUID={}]", uuid);
    outErrMsg.setHoldValue(message);
    outErrUUID.setHoldValue(uuid);
    saveOutputAttributes();
    return continueFail();
}
}
}

```

UUID is a unique user ID that you can use to report errors. For example, if an error occurs, an SMS message can be sent to the consumer, who is identified by the UUID. Consumers can call customer support to report issues, using their UUID. UUIDs are logged so they can be correlated with reported issues.

### **Defining Input Variables**

States use input variables to get input values, either from a session variable or as a constant. You can configure the behavior in the state editor. The `InputAttribute` class manages input variables.

In addition to the basic properties, input variables have an **isOptional** property. If set to true, the input variable is optional; false indicates it is mandatory.

The input variable constructor is:

```
InputAttribute (String id, String description, boolean isOptional)
```

Two types of input variables exist, text box input controls and selection input controls.

#### *Text Box Input Controls*

Text boxes manage either a single constant value or a value that is accessed from a session variable.



You can create the input variable in the example above using this constructor:

```
TextBoxAttribute( " POSTCODE " , " Zip or Postal Code " , false);
```

By default, the variable ID is automatically assigned to the `TextBoxAttribute` control. In this case, the ID is `POSTCODE`. The description, `Zip or Postal Code`, appears to the right. The red dot indicates that the input variable is mandatory.

**Note:** If input is mandatory and a session variable name is specified, a runtime error is thrown if the session variable does not exist. The processing engine terminates the application, unless

the state implementation handles `RequiredParameterMissingException`, with either `continueFail` or `continueDyn` follow-up transitions.

---

The state of the check box tells the processing engine how to process an input variable:

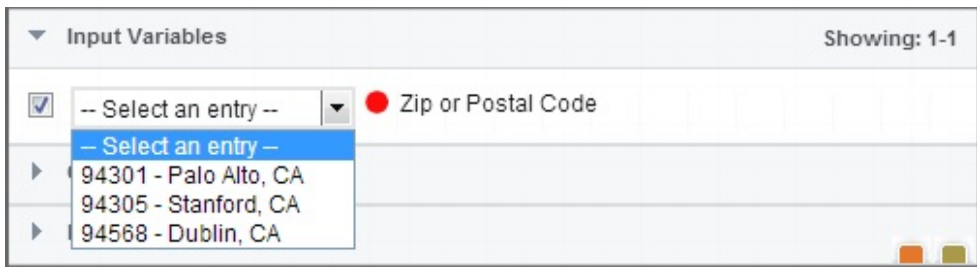
- Selected – retrieve the value from the named session variable.
- Not selected – use the constant value.

If you use a state twice in the same application, and if the state saves a value in a session variable, change the session-variable name in the second instance, so it does not overwrite the value.

To find the session-variable name, hover the mouse over the description text; pop-up text includes the variable description and the variable name.

### *Selection Input Controls*

Selection input controls manage constant values that are selected from a list of options. Lists are populated in the state code.



Unique IDs are automatically assigned as the session-variable name; you cannot change them, and they do not appear in the state editor. To find the session-variable name, hover the mouse over the description text; pop-up text includes the variable description and the variable name.

To use a state twice in the same application, and save the value of the session variable, you can call the `Copy Variables` state to copy the session variable to another variable.

The check box performs the same function as it does for text box controls. The red dot indicates that an input selection is mandatory.

### **See also**

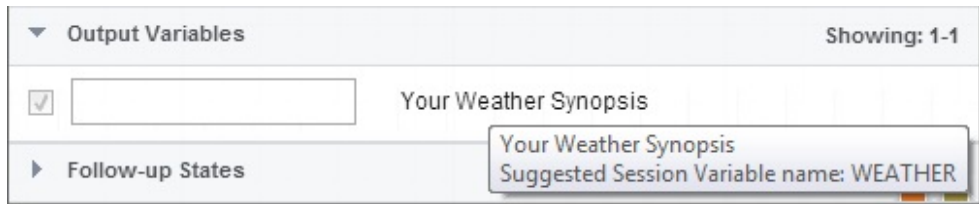
- *Input and Output Parameters* on page 4
- *Custom State Variables* on page 45
- *Defining Output Variables* on page 49
- *Accessing Input Variables* on page 50
- *List Variables* on page 51

## Defining Output Variables

States return results as output variables, which are always session variables. Only states can set output variables, and only at runtime. Output-variable check boxes are always selected and cannot be modified.

To create an output variable, use the `OutputAttribute` constructor:

```
OutputAttribute("WEATHER", "Your Weather Synopsis")
```



By default, output session-variable names are not set, so text boxes are empty. You can set values by calling either of these two methods:

- `setValue` – creates a session variable (if none exists), and saves the value immediately in the database, or,
- `setHoldValue` – temporarily holds the value in the cache, until you explicitly call the `SmappStatePlugin::saveOutputAttributes` method.

The `saveOutputAttributes` method saves multiple session variables with a single database connection. If the state has only a few output variables, call the `setValue` method. If there are many output variables, call `setHoldValue`; this may impact the efficiency of the state at runtime.

To set output variables, call one of the methods in the `OutputAttribute` class:

- `public void setValue (String val)`
- `public void setValue (Long val)`
- `public void setValue (Integer val)`
- `public void setValue (Boolean val)`
- `public void setHoldValue (String val)`
- `public void setHoldValue (Long val)`
- `public void setHoldValue (Integer val)`
- `public void setHoldValue (Boolean val)`

## See also

- *Input and Output Parameters* on page 4
- *Custom State Variables* on page 45
- *Defining Input Variables* on page 47
- *Accessing Input Variables* on page 50

- *List Variables* on page 51

### **Accessing Input Variables**

You can access input variables that are in a custom state using either the `getInputValue` method or the `getInputValueWithWarning` method.

The signatures of the methods you can call to access input variables are:

```
public InputValue getInputValue()
    throws DBException;
public InputValue getInputValueWithWarning()
    throws DBException, RequiredParameterMissingException;
```

To retrieve optional input variables, call `getInputValue`. A null value is returned if either an input variable is not provided, or if the session variable that the input variable is assigned to does not exist.

```
InputValue iv = optionalVar.getInputValue();

if (iv != null) {
    retrieve the value
}
```

To retrieve mandatory input variables, call `getInputValueWithWarning`. The exception `RequiredParameterMissingException` is raised if either an input variable is not provided, or if the session variable that the input variable is assigned to does not exist. You can retrieve all mandatory input variables in the same `try/catch` block.

```
try {
    Long id = mandatoryIdVar.getInputValueWithWarning().getLong();
    Integer count =
mandatoryCountVar.getInputValueWithWarning().getInt();
}
catch (RequiredParameterMissingException rex) {
    log.error(rex.getMessage());
    return continueFail();
}
```

---

**Note:** The `RequiredParameterMissingException::getMessage` method indicates the mandatory variable that is missing.

---

Both methods that access input variables return the `InputValue` class. `InputValue` methods return values that you enter in the state editor when you configure an input attribute; return values can be either constants or session-variable names:

- `InputValue.getString()`;
- `InputValue.getString(int size)`;
- `InputValue.getLong()`;
- `InputValue.getInt()`;
- `InputValue.getBoolean()`;
- `InputValue.getDouble()`;



- `InputValue.getMsisdn()`;

### See also

- *Input and Output Parameters* on page 4
- *Custom State Variables* on page 45
- *Defining Input Variables* on page 47
- *Defining Output Variables* on page 49
- *List Variables* on page 51

### List Variables

List variables do not appear in the state editor. You can use list variables to save lists of the `BeanConverterInterface` type to session variables.

As an example, the `AbstractDynamicMenu` class uses a list variable to persist an SMS menu. The `BeanConverterInterface` specifies that a bean must provide string serialization and deserialization logic. Each `BeanConverterInterface` item is saved as a session variable with a unique name.

```
package com.sybase365.mobiliser.brand.plugins.smapp.beans;
public interface BeanConverterInterface<T> {
    T convert(String value);
    String convert(T object);
}
```

---

**Note:** Strings returned by the `convert(T object)` method must be less than 1000 characters.

---

The `SessionVariableAttribute` class has two methods: `getList` and `setList`. The `getList` method retrieves a list from the database. When `setList` is called, the list is saved to a session variable, which requires a database connection.

---

**Note:** Lists are saved outside of transactions. Therefore, if an exception occurs, the method throws a `DBException`, and a partial list may be saved. It is up to the state implementation that uses this attribute to retry.

---

Most state implementations do not need list variables. They are needed only if a state can transition into an internal waiting condition by calling `waitForMessage`. For example, list variables are most commonly used when sending SMS messages. Calling `waitForMessage` causes the application to hibernate until the response arrives. The list variable is saved to a session variable, so it is available when the application is reactivated.

### See also

- *Input and Output Parameters* on page 4
- *Custom State Variables* on page 45
- *Defining Input Variables* on page 47
- *Defining Output Variables* on page 49

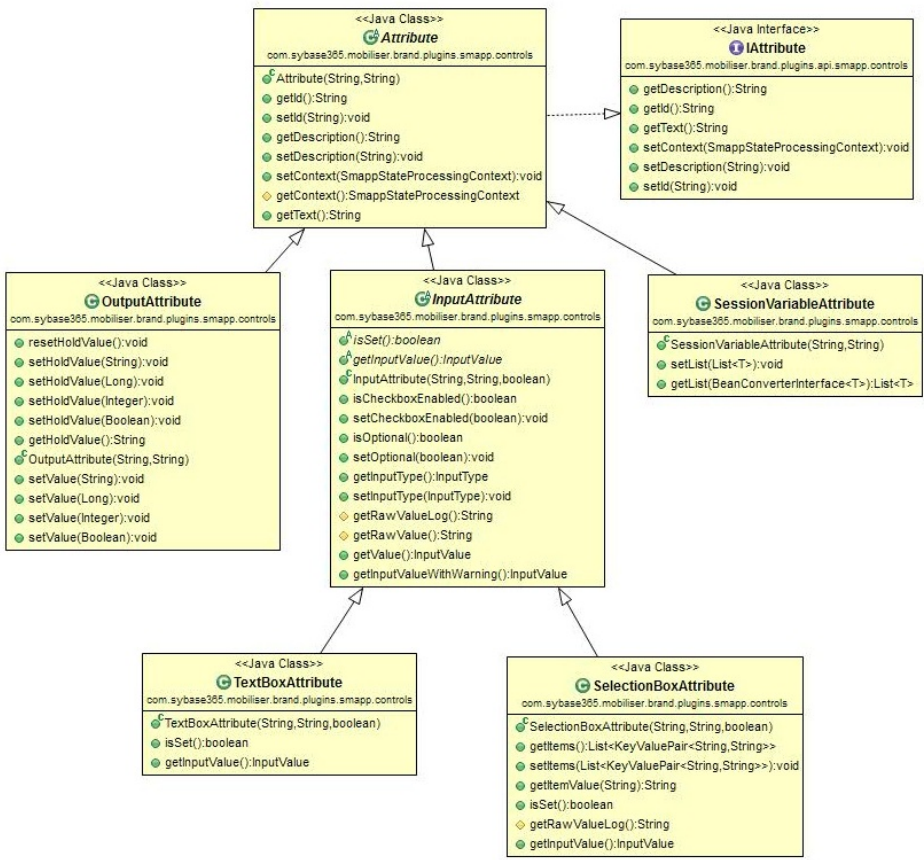
## Developing Custom Application States

- *Accessing Input Variables* on page 50
- *Extending the AbstractDynamicMenu Class* on page 39

### State Attributes Class Hierarchy

All state variables that are derived from the `Attribute` class are identified by an ID and a description, which are defined in the constructor `Attribute(String ID, String Description)`. ID is a unique identifier of the attribute; for `InputAttribute`, ID defaults to the session variable name. The value of the Description variable appears in the Application Composer.

The diagram below illustrates the attribute class hierarchy.



These methods are reserved for use by the processing engine:

- `public void setContext (SmappStateProcessingContext context)`
- `protected SmappStateProcessingContext getContext ()`

`SmappStateProcessingContext` is the running context of the application, set by the processing engine using the `setContext` method.

`SmappStateProcessingContext` provides access to the data source that stores session variables.

## **Setting Up Apache Maven**

Apache Maven is a software project management tool that is based on a project object model (POM). You can use Maven to manage a project's build, reporting, and documentation from a central piece of information.

Install and configure Apache Maven, and deploy the State SDK bundles, so you can build custom-state bundles and deploy them to Brand Mobiliser.

### **Installing Apache Maven**

You can download Apache Maven from the Apache Maven Project Web site. Apache Maven version 3.0.4 has been tested and certified with Brand Mobiliser.

1. Navigate to <http://maven.apache.org/download.cgi>, and download Apache Maven.
2. To verify that your Apache Maven installation is successful, on the command line, run:

```
mvn -version
```

The output looks similar to:

```
Apache Maven 3.0.4 (r1232337; 2012-01-17 00:44:56-0800)
Maven home: C:\ZPrograms\apache-maven-3.0.4 Java version: 1.6.0_35,
vendor: Sun Microsystems Inc.
Java home: C:\Program Files\Java\jdk1.6.0_35\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 7", version: "6.1", arch: "amd64", family: "windows"
```

### **Next**

Configure Apache Maven.

### **Configuring Apache Maven**

You can customize where Maven looks for dependencies by editing the Maven configuration file.

### **Prerequisites**

Install Apache Maven.

### **Task**

By default, Maven looks for dependencies in its central repository; however, in some cases, it may need additional repositories. For example, some companies have their own internal Maven repositories, and you, as a developer, must find these dependencies. The central Maven repository is open to the public, and its libraries are either open source or available for public

## Developing Custom Application States

use. Brand Mobiliser SDK libraries are not hosted in the central Maven repository, nor in any publicly accessible Maven repository.

1. Navigate to your Apache Maven installation directory, and open the `conf\setting.xml` file.
2. Enter these lines:

```
<settings>
<profiles>
  <profile>
    <id>brand_state_development</id>
    <repositories>
      <repository>
        <id>EclipseLink</id>
        <name>Eclipse Link</name>
        <url>http://download.eclipse.org/rt/eclipselink/maven.repo</url>
      </repository>
    </repositories>
  </profile>
</profiles>

<activeProfiles>
  <activeProfile>brand_state_development</activeProfile>
</activeProfiles>
</settings>
```

3. To add a Maven dependency location, between the `<repositories></repositories>` elements, add a `<repository></repository>` element pair.
4. For the new repository, define:
  - `id` – repository ID.
  - `name` – name of the repository.
  - `url` – Internet location of the repository.

Maven creates a default-user local cache repository in `${user.home}/.m2/repository`, where *user.home* depends on the operating system. For example, on a Windows 7 machine, the *user.home* location is `C:\Users\userName`. During the build process, this is the first location Maven searches for dependency libraries. Initially, the local repository is empty. During the first build, Maven does not find libraries in the local repository, so it looks in the Maven central repository, which is, by default, <http://search.maven.org/#browse>. Maven downloads any dependency libraries to the local repository, then uses them in the build. Subsequent builds are faster, because dependency libraries have been downloaded to the local repository.

### Next

Deploy State SDK bundles to Maven repositories.

## **Deploying State SDK Bundles to a Maven Repository**

You can deploy State SDK bundles to the local Maven repository (also known as the `.m2`). Deploy bundles to local repositories on each development machine.

### **Prerequisites**

Install and configure Apache Maven.

### **Task**

In Brand Mobiliser version 1.3, the State SDK consists of five bundles:

- mobiliser-brandplugin-api-1.3.1.jar
- mobiliser-brandstate-sdk-1.3.1.jar
- mobiliser-brandplugin-security-1.3.1.jar
- mobiliser-brandplugin-core-1.3.1.jar
- mobiliser-brandplugin-jpa-1.3.1.jar

Deploy these bundles to the Maven repository so they are accessible as dependencies to state-development projects. Bundles are in the `BRAND_HOME\bundle\application` directory. To deploy the bundles, run a script for each bundle, or copy all five scripts to a single script file, and run it once.

---

**Note:** Scripts are for Windows only; to run on Linux, modify the `-Dfile` path.

---

1. Change to the `BRAND_HOME` directory.

2. Run:

```
mvn install:install-file -Dfile=bundle\application\mobiliser-brand-
plugin-api-1.3.1.jar
-DgroupId=com.sybase365.mobiliser.brand.plugins -DartifactId=mobiliser-
brand-plugin-api
-Dversion=1.3.1 -Dpackaging=jar

mvn install:install-file -Dfile=bundle\application\mobiliser-brand-
state-sdk-1.3.1.jar
-DgroupId=com.sybase365.mobiliser.brand.plugins -DartifactId=mobiliser-
brand-state-sdk
-Dversion=1.3.1 -Dpackaging=jar

mvn install:install-file -Dfile=bundle\application\mobiliser-brand-
security-1.3.1.jar
-DgroupId=com.sybase365.mobiliser.brand.security -
DartifactId=mobiliser-brand-security
-Dversion=1.3.1 -Dpackaging=jar

mvn install:install-file -Dfile=bundle\application\mobiliser-brand-
core-1.3.1.jar
-DgroupId=com.sybase365.mobiliser.brand.core -DartifactId=mobiliser-
brand-core -Dversion=1.3.1
-Dpackaging=jar
```

```
mvn install:install-file -Dfile=bundle\application\mobiliser-brand-  
jpa-1.3.1.jar  
-DgroupId=com.sybase365.mobiliser.brand.database -  
DartifactId=mobiliser-brand-jpa  
-Dversion=1.3.1 -Dpackaging=jar
```

### **Custom State Bundles**

Package custom states as OSGi bundles, which you can deploy to Brand Mobiliser.

An OSGi bundle is a JAR file with extra manifest headers that can be deployed in the OSGi container. A custom-state bundle can contain one or more custom states, and it must be packaged as an OSGi bundle before you can deploy it to Brand Mobiliser.

#### **Building Custom State Bundles**

After you develop custom states, and set up Apache Maven, build OSGi bundles that you can deploy to Brand Mobiliser.

##### **1. *Creating Maven Projects***

The main artifacts of a Maven project are the project object model (POM) file, and folders that contain source-code files.

##### **2. *Customizing Maven POM Files***

Customize a Maven project object model (POM) file to create and build custom-state OSGi bundles to deploy to Brand Mobiliser.

##### **3. *Creating Maven Project Artifacts***

After you create a Maven project, create project artifacts to use in a custom-state bundle.

##### **4. *Building Maven Projects***

You can build Maven projects on the command line, or you can use Maven build and unit test projects in an IDE.

##### **5. *Declaring States as Spring Beans***

Developing a custom-state bundle requires that you declare each state as a Spring Framework bean in the beans-context.xml file. A state is any Java class that either directly or indirectly extends the SmappStatePlugin abstract class.

##### **6. *Configuring Bean Properties***

The bean properties file, properties-context.xml, declares all properties that must be retrieved from the OSGi configuration administration service during runtime; properties are stored in the service so they can be configured dynamically at runtime.

##### **7. *Registering States as OSGi Services***

To enable Brand Mobiliser to discover states at runtime, register them as OSGi services, by declaring them in the services-context.xml file.

## **Creating Maven Projects**

The main artifacts of a Maven project are the project object model (POM) file, and folders that contain source-code files.

You can create a new Maven project on the command line, or in any IDE that supports Maven. To create a Maven project on the command line:

```
mvn archetype:create -DgroupId=com.sap.example -DartifactId=customState
```

where:

- *groupId* – names the package.
- *artifactId* – names the project and the project folder.

As the project is created, you see progress messages. For example:

```
[INFO] Scanning for projects...
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/
plugins/
maven-clean-plugin/2.4.1/maven-clean-plugin-2.4.1.pom
Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/
plugins/
maven-clean-plugin/2.4.1/maven-clean-plugin-2.4.1.pom (5 KB at 6.8 KB/
sec)
[...]
[INFO]
[INFO] -----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
[INFO]
[INFO] --- maven-archetype-plugin:2.2:create (default-cli) @ standalone-
pom ---
[...]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 41.155s
[INFO] Finished at: Mon Oct 22 17:00:49 PDT 2012
[INFO] Final Memory: 8M/245M
```

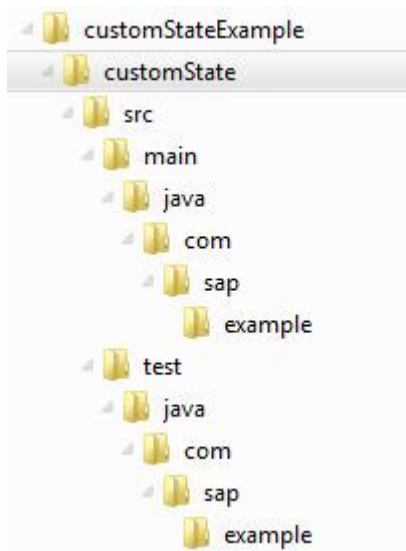
### **See also**

- *Creating Maven Project Artifacts* on page 63
- *Sample Maven POM File* on page 60
- *Maven Project Structure* on page 58
- *Customizing Maven POM Files* on page 59

### Maven Project Structure

When you create a Maven project, the directory structure that is created includes the project object model (POM) file.

In this sample project, the *groupId* is set to `com.sap.example`. This directory structure is created automatically for a new project:



Two Java files, `App.java` and `AppTest.java`, are created in the `example` folders, under `main` and `test`, respectively. The POM file, which contains the initial project configuration, is created in the `customState` folder. You can use this POM file as a starting point for custom-state development.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.sap.example</groupId>
  <artifactId>customState</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>customState</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```



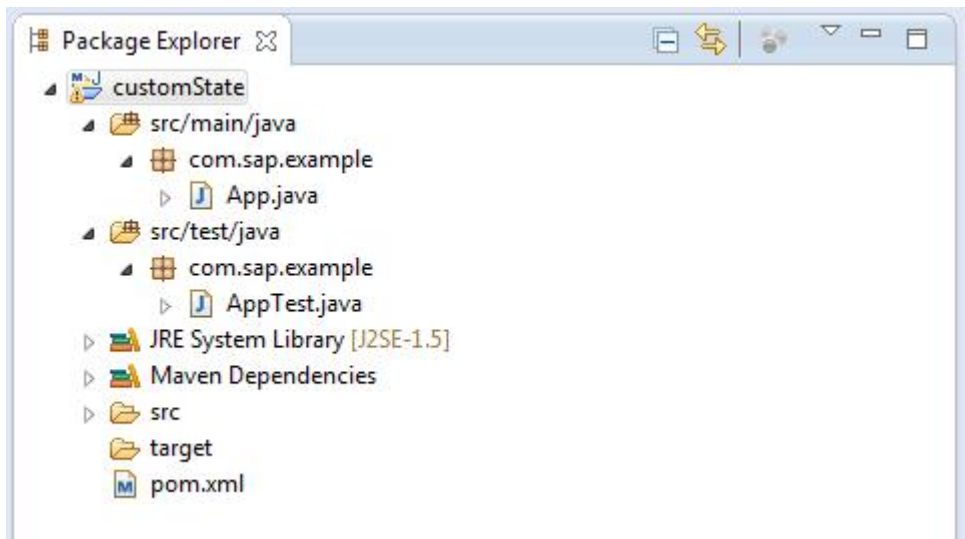
```

</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

You can open or import a newly created Maven project into your IDE. Eclipse and NetBeans both support Maven. The image below shows the sample project structure in Eclipse.



Once you are familiar with the structure and the content of POM files, you can create them manually. You can also create a new project in any IDE that supports Maven.

### See also

- *Sample Maven POM File* on page 60
- *Creating Maven Projects* on page 57
- *Creating Maven Project Artifacts* on page 63

### **Customizing Maven POM Files**

Customize a Maven project object model (POM) file to create and build custom-state OSGi bundles to deploy to Brand Mobiliser.

Edit the Maven `pom.xml` file for your project to define:

- `groupId` – package name.

## Developing Custom Application States

- `artifactId` – name of the project.
- `version` – version number of the project.
- `packaging` – bundle.
- `name` – name of the state.

For example:

```
<groupId>com.sap.example</groupId>
<artifactId>customState</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>bundle</packaging>
<name>Custom State</name>
```

### See also

- *Creating Maven Projects* on page 57
- *Creating Maven Project Artifacts* on page 63

### Sample Maven POM File

A Maven project object model (POM) file contains all the required information for Maven to create and build OSGi bundles that you can deploy to Brand Mobiliser.

This POM file (`pom.xml`) illustrates the basic configuration for a custom-state bundle. The state implementation does not need libraries other than those provided by the SDK. The SDK libraries are shown as dependencies. The contents of the original POM are shown in **bold**:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.sap.example</groupId>
  <artifactId>customState</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>bundle</packaging>
  <name>Custom State</name>
  <url>http://www.sap.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <bundle.namespace>${project.groupId}</bundle.namespace>
    <bundle.symbolicName>${bundle.namespace}.${project.artifactId}</
bundle.symbolicName>
    <brand.version>1.3.1</brand.version>
  </properties>

  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
```

```

<configuration>
  <source>1.6</source>
  <target>1.6</target>
</configuration>
</plugin>

<!-- Create an OSGi Bundle Manifest -->
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <version>2.3.7</version>
  <extensions>>true</extensions>
  <configuration>
    <instructions>
      <manifestLocation>META-INF</manifestLocation>
      <Bundle-Category>object</Bundle-Category>
      <Bundle-SymbolicName>${bundle.symbolicName}</Bundle-SymbolicName>

      <Bundle-Version>${project.version}</Bundle-Version>
      <Embed-Dependency></Embed-Dependency>

      <!--
      Note: When you develop additional classes within this object
      bundle, include the package names of the classes in either the
      Export-Package, or the Private-Package, otherwise it will not
      be included in the bundle.
      -->

      <Export-Package>
      </Export-Package>

      <Private-Package>
        com.sap.example
      </Private-Package>

      <DynamicImport-Package>
      </DynamicImport-Package>

      <!--
      Note: If you use other only referenced from spring context then
      include them in the Import-Package instruction here. The *
      instruction ensures that any directly imported packages in
      supporting classes are included automatically, but the Spring
      context referenced ones need explicit reference.
      -->
      <Import-Package>
        *
      </Import-Package>

      <!--
      Each module can override these defaults in an
      optional osgi.bnd file
      -->
      <_include>-osgi.bnd</_include>

      <!--

```

## Developing Custom Application States

```
Enable viewing of the properties file content from telnet console
-->
<ARF-Bundle-Template>/META-INF/config</ARF-Bundle-Template>

</instructions>
<obrRepository>NONE</obrRepository>
</configuration>
</plugin>
</plugins>
</build>

<dependencies>
<dependency>
<groupId>com.sybase365.mobiliser.brand.plugins</groupId>
<artifactId>mobiliser-brand-plugin-api</artifactId>
<version>${brand.version}</version>
</dependency>
<dependency>
<groupId>com.sybase365.mobiliser.brand.plugins</groupId>
<artifactId>mobiliser-brand-state-sdk</artifactId>
<version>${brand.version}</version>
</dependency>
<dependency>
<groupId>com.sybase365.mobiliser.brand.security</groupId>
<artifactId>mobiliser-brand-security</artifactId>
<version>1.3.1</version>
</dependency>
<dependency>
<groupId>com.sybase365.mobiliser.brand.core</groupId>
<artifactId>mobiliser-brand-core</artifactId>
<version>${brand.version}</version>
</dependency>
<dependency>
<groupId>com.sybase365.mobiliser.brand.database</groupId>
<artifactId>mobiliser-brand-jpa</artifactId>
<version>${brand.version}</version>
</dependency>

<!-- Logging -->
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>slf4j-api</artifactId>
<version>1.6.6</version>
</dependency>

<!-- Optional for Unit Test -->
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>

<!--
```

```

Required Javax Persistence dependencies not available
from Maven central repository
-->
<profiles>
  <profile>
    <activation>
      <jdk>[1.5, 1.7)</jdk>
    </activation>
    <dependencies>
      <dependency>
        <groupId>org.eclipse.persistence</groupId>
        <artifactId>javax.persistence</artifactId>
        <version>2.0.4.v201112161009</version>
        <scope>provided</scope>
      </dependency>
    </dependencies>
    <repositories>
      <repository>
        <id>EclipseLink</id>
        <url>http://download.eclipse.org/rt/eclipselink/maven.repo</url>
      </repository>
    </repositories>
  </profile>
</profiles>
</project>

```

**See also**

- *Maven Project Structure* on page 58
- *Creating Maven Projects* on page 57
- *Creating Maven Project Artifacts* on page 63

**Creating Maven Project Artifacts**

After you create a Maven project, create project artifacts to use in a custom-state bundle.

**Prerequisites**

Create a Maven project.

**Task**

1. In the example subdirectory under main, delete the `App.java` file.
2. In the example subdirectory, under test, delete the `AppTest.java` file.
3. In the main directory, create a subdirectory called `resources`.  
The `resources` directory stores configuration files that Brand Mobiliser needs when it loads state bundles.
4. In the `resources` directory, create these subdirectories:
  - `META-INF` – contents are packaged in the state bundle.

## Developing Custom Application States

- META-INF/spring – stores a configuration file that the Spring Framework uses.
- META-INF/sample/conf – stores sample configuration property files; if you copy these files to *BRAND\_HOME/conf/cfgload*, Brand Mobiliser can load them dynamically.

Configuration files are specific to a bundle. They tell Brand Mobiliser what states and configurations to load, and how to link them together.

5. In the `test` directory, create these subdirectories:

- `java`
- `resources`

### See also

- *Creating Maven Projects* on page 57
- *Sample Maven POM File* on page 60
- *Maven Project Structure* on page 58
- *Customizing Maven POM Files* on page 59

### **Building Maven Projects**

You can build Maven projects on the command line, or you can use Maven build and unit test projects in an IDE.

For information about building projects using Maven in the Eclipse IDE, see <http://maven.apache.org/eclipse-plugin.html>.

On the command line, run:

```
mvn clean install
```

As the project builds, you see progress messages:

```
[INFO] Scanning for projects...
Downloading: http://repo.maven.apache.org/maven2/org/apache/felix/maven-
bundle-plugin/
2.3.7/maven-bundle-plugin-2.3.7.pom
Downloaded: http://repo.maven.apache.org/maven2/org/apache/felix/maven-
bundle-plugin/
2.3.7/maven-bundle-plugin-2.3.7.pom
(4 KB at 15.0 KB/sec)
[...]
[INFO] Installing C:\ZMobiliser\customStateExample\customState\target
\customState-1.0-SNAPSHOT.jar
to C:\Users\I824993\.m2\repository\com\sap\example\customState\1.0-
SNAPSHOT\customState-1.0-SNAPSHOT.jar
[INFO] Installing C:\ZMobiliser\customStateExample\customState\pom.xml
to
C:\Users\I824993\.m2\repository\com\sap\example\customState\1.0-SNAPSHOT
\customState-1.0-SNAPSHOT.pom
[INFO]
[INFO] --- maven-bundle-plugin:2.3.7:install (default-install) @
customState ---
[INFO] Local OBR update disabled (enable with -DobrRepository)
```

```
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 36.332s
[INFO] Finished at: Mon Oct 29 10:48:50 PDT 2012
[INFO] Final Memory: 11M/242M
[INFO]
-----
```

The bundle JAR file is saved in the `/className/target` directory; its name is derived from the Maven project *artifactId* and version. For this example, the filename is `customState-1.0-SNAPSHOT.jar`.

### **Declaring States as Spring Beans**

Developing a custom-state bundle requires that you declare each state as a Spring Framework bean in the `beans-context.xml` file. A state is any Java class that either directly or indirectly extends the `SmappStatePlugin` abstract class.

You can configure Spring beans by setting properties, or by creating other beans that support state operations.

1. Edit the `beans-context.xml` file to add a `<bean>` element for each state. Define:

- `id` – name of the state.
- `class` – name of the Java class that implements the state.

For example:

```
<bean id="SampleState" class="com.sap.example.SampleState">
  <property name="country" value="${sample.country}"/>
</bean>
...
```

2. (Optional) Declare state properties, and assign either constant values or references to the values that are defined in the `properties-context.xml` file.

The value of the *country* property is a reference to the *sample.country* property defined in `properties-context.xml`.

### **Configuring Bean Properties**

The bean properties file, `properties-context.xml`, declares all properties that must be retrieved from the OSGi configuration administration service during runtime; properties are stored in the service so they can be configured dynamically at runtime.

You can reconfigure states at runtime, without reloading state bundles or restarting Brand Mobiliser. However, state developers must implement dynamic reconfiguration, by defining state properties in the code.

Edit the `properties-context.xml` file to configure bean properties:

## Developing Custom Application States

- a) Set `osgix:cm-properties id` to the name of the OSGi configuration administration service property that is identified by the value of `persistent-id`.  
Brand Mobiliser initializes the property, and loads the property file identified by the value of `persistent-id`.
- b) For each property, enter a `<prop key>` element and default value.  
Properties are initialized with values from the OSGi configuration administration service. If a property does not exist in the service, the default value is used.
- c) Set the value of `ctx:property-placeholder properties-ref` to the value of `osgix:cm-properties id`.  
The value identifies a list of properties that are available for the Spring Framework to use during state initialization.

For example:

```
<osgix:cm-properties id="sampleState-cfg" persistent-  
id="service.sampleState">  
  <prop key="sample.country">US</prop>  
</osgix:cm-properties>  
  
<ctx:property-placeholder properties-ref="sampleState-cfg"/>
```

---

**Note:** SAP recommends that you store a copy of the `properties-context.xml` file in the `META-INF/sample/conf` directory.

---

### **Registering States as OSGi Services**

To enable Brand Mobiliser to discover states at runtime, register them as OSGi services, by declaring them in the `services-context.xml` file.

Registered states are discoverable by the `StatePlugin` interface:

```
com.sybase365.mobiliser.brand.plugins.api.smapp.StatePlugin
```

Edit `services-context.xml`, and set OSGi service properties:

- `id` – name of the service.
- `ref` – name of the state.
- `interface` – name of the class that implements the `StatePlugin` interface.

For example:

```
<osgi:service id="SampleStateService" ref="SampleState"  
interface="com.sybase365.mobiliser.brand.plugins.api.smapp.StatePlugin"/>
```



## **Deploying State Bundles**

To deploy custom-state bundles, make the files available to Brand Mobiliser at runtime, and configure the states to start automatically.

1. Copy the bundle `.jar` files to `BRAND_HOME/bundle/application`.

This directory contains all the bundles that are deployed to the Brand Mobiliser runtime environment.

---

**Note:** Brand Mobiliser system bundles are installed in `BRAND_HOME/bundle`.

---

2. Edit the `BRAND_HOME/conf/config.properties` file to add the new custom-state file to the list of bundles that are started automatically.

```
felix.auto.start.15 = ${aims.app.dir}/customState-1.0-SNAPSHOT.jar
```

All state bundles are listed in the `config.properties` file. Brand Mobiliser reinitializes its bundle cache each time it starts.

3. Restart the Brand Mobiliser server.

To verify that no errors occurred, check these log files:

- `brand.log`
- `felix.log`
- `spring.log`
- `persist.log`

If there are errors, check the Spring configuration and the `import/private/dynamic` package specifications.

## **Next**

To verify that bundles resolve and start, use either Telnet or the AIMS System Web console (both require access to localhost).

### **Verifying Deployment Using Telnet**

Use Telnet to verify that custom-state bundles resolve and start. The Telnet interface listens only on the localhost port, which ensures runtime environment security.

1. On the command line, run:

```
telnet localhost 5365
```

2. At the Telnet prompt, run:

```
felix:lb
```

You see output similar to the following; the state of the bundle, in this case `customState`, is `Active`:

```
START LEVEL 20
ID|State      |      Level|Name
0|Active      |          0|System Bundle (4.0.3)
1|Active      |          14|activemq-core (5.5.1)
```

## Developing Custom Application States

```
2|Active      |      14|activemq-pool (5.5.1)
3|Active      |      14|activemq-ra (5.5.1)
4|Active      |      14|activemq-spring (5.5.1)
5|Active      |      14|ARF :: System :: arf-sys (0.3.4)
6|Active      |      14|ARF :: System :: arf-util-commands (0.3.2)
7|Active      |      14|ARF :: System :: cm-bridge (0.3.4)
8|Active      |      14|Java Activation API (1.1.1)
9|Active      |      14|Java Messaging System API (1.1.0)
10|Active     |      14|CGLIB Code Generation Library (2.2.0)
11|Active     |      14|AOP Alliance API (1.0.0)
12|Active     |      14|Commons Pool (1.5.6)
...
108|Active    |       1|ARF :: System :: cm-loader (0.3.4)
109|Resolved   |       1|AIMS :: Object :: Brand Mobiliser Felix JRE
System Package Support (1.3.1)
110|Installed  |      10|AIMS :: Object :: Brand Mobiliser Quartz OSGi
Support (1.3.1)
111|Active     |      17|Restlet API (2.0.13.0)
112|Active     |      17|Restlet Extension - Servlet (2.0.13.0)
113|Active     |      17|Restlet Extension - Spring Framework (2.0.13.0)

114|Active     |      17|Restlet Extension - JSON (2.0.13.0)
115|Active     |      17|AIMS :: Service :: Brand Mobiliser Core REST
Services (1.3.1)
116|Active     |      16|AIMS :: Object :: Web Core (0.1.9)
117|Active     |      16|AIMS :: Object :: Web API and Model (0.1.9)
118|Active     |      16|AIMS :: Process :: Brand Mobiliser Webadmin UI
(1.3.1)
119|Active     |      15|customState (1.0.0.SNAPSHOT)
```

### Verifying Deployment Using the AIMS Web Console

In a development environment, you can use the AIMS System Web console to verify that custom-state bundles resolve and start. To ensure runtime environment security, the console restricts access, based on a list of allowable IP addresses. By default, only localhost is accessible.

### Prerequisites

Enable the AIMS System Web console.


### Task

1. (Optional) To add IP addresses that the console can access:
  - a) Edit the `org.apache.felix.webconsole.internal.servlet.OsgiManager.properties` file.
  - b) Add IP addresses to the `allowed.ip.list`, as a comma-separated list.
2. In a Web browser, connect to `http://localhost:8080/system/console`.  
If you added other IP addresses, you can connect using one of them.

3. In the AIMS System Web console, enter these credentials:

- User name – `sybase365`
- Password – `fr4nt1c`

The **Bundles** tab lists all installed bundles. The Status of the `customState` bundle is Active.



**AIMS System Web Console Bundles**

Bundle information: 122 bundles in total, 119 bundles active, 2 active fragments, 0 bundles resolved, 1 bundles installed.

Id	Name	Version	Category	Status	Actions
121	customState (com.sap.example.customState)	1.0.0.SNAPSHOT	object	Active	[Refresh] [Reload] [Install/Update...]

Bundle information: 122 bundles in total, 119 bundles active, 2 active fragments, 0 bundles resolved, 1 bundles installed.

4. To view details about a bundle, click the bundle name.

The console displays metadata, created by the Maven Bundle Plug-in (from the bundle's manifest file), package wiring, and services information.

### Enabling the AIMS System Web Console

During development, you can use the AIMS System Web console to inspect deployed bundles, registered configurations, and the OSGi container. By default, the Web console is disabled.

1. Edit the `BRAND_HOME/conf/config.properties` file, and uncomment these lines:

```
# Uncomment to aid in debugging container issues.
#felix.auto.start.6 = \
#${aims.app.dir}/aims-felix-webconsole-1.0.2.jar \
#${aims.app.dir}/event-webconsole-1.0.3-SNAPSHOT.jar
```

2. Copy the

```
org.apache.felix.webconsole.internal.servlet.OsgiManager.p
properties
```

file to the `conf/cfgbackup` folder.

### Next

See <http://felix.apache.org/site/apache-felix-web-console.html>.

### Configuring State Bundles

You can configure state bundles in the `service.bundle.properties` file, where `bundle` is the name of the state bundle.

### Prerequisites

Deploy the state bundle.

### Task

1. Edit the `service.bundle.properties` file.
2. Copy the file to the `BRAND_HOME/conf/cfgload` directory.  
When the Brand Mobiliser server restarts, the files in the `/conf/cfgload` directory are moved to `/conf/cfgbackup`, and all properties are reconfigured.

### Next

Verify the new configuration using either Telnet or the AIMS Web System console.

### Verifying Bundle Configuration Using Telnet

You can use Telnet to verify that state bundle configuration changes are in effect.

1. On the command line, run:

```
telnet localhost 5365
```

2. At the Telnet prompt, run:

```
aims:cmlist
```

You see:

Configuration list:

```
org.apache.felix.webconsole.internal.servlet.OsgiManager
  file:bundle/application/aims-felix-webconsole-1.0.2.jar
service.event.quartz
  file:bundle/application/event-scheduler-quartz-1.0.3.jar
org.ops4j.pax.logging
  file:bin/pax-logging-service-1.6.9.jar
service.webui.security
  file:bundle/application/web-core-0.1.9.jar
service.sampleState
  file:bundle/application/customState-1.0-SNAPSHOT.jar
service.brand_webapp
  file:bundle/application/mobiliser-brand-webadmin-ui-1.3.1.war
service.mobiliserCustomer.states.plugin null
service.mobiliserCustomer.client.plugin null
service.dsprovider
  file:bundle/application/dbcp-osgi-service-1.3.1.jar
service.coreprocessing
  file:bundle/application/mobiliser-brand-processing-1.3.1.jar
org.ops4j.pax.web
  file:bundle/application/pax-web-jetty-bundle-1.1.4.jar
service.event.core
  file:bundle/application/event-core-1.0.3.jar
```

In the output above, the service process ID (PID) for the `customState-1.0-SNAPSHOT.jar` is `service.sampleState`.

3. To see the `customState-1.0-SNAPSHOT.jar` configuration, run:

```
aims:cmget service.sampleState
```

You see:

```
Configuration for service (pid) "service.sampleState"
(bundle location = file:bundle/application/customState-1.0-
SNAPSHOT.jar)
```

key	value
service.pid	service.sampleState
sample.country	US
arf.filename	service.sampleState.properties

If you set the `<ARF-Bundle-Template>` property in the Maven POM file, you can view the sample properties file that is packaged in the state bundle. Sample property files generally contain documentation for each property.

**4. To find all state bundles that have sample property templates, run:**

```
aims:template
```

You see:

```
Bundles with configuration templates:
ID: 39 Bundle:com.sybase365.mobiliser.thirdparty.smpapi
ID: 49 Bundle:com.sybase365.mobiliser.brand.processing.mobiliser-brand-
processing
ID: 51 Bundle:com.sybase365.mobiliser.brand.database.mobiliser-brand-
jpa
ID: 52 Bundle:com.sybase365.mobiliser.brand.database.mobiliser-brand-
jpa-eclipselink
ID: 56 Bundle:com.sybase365.mobiliser.framework.event-store-db-
provider
ID: 57 Bundle:com.sybase365.mobiliser.framework.event-store-jpa
ID: 58 Bundle:com.sybase365.mobiliser.framework.event-store-
eclipselink
ID: 60 Bundle:com.sybase365.mobiliser.brand.osgi.dbcp-osgi-service
ID: 117 Bundle:com.sybase365.mobiliser.brand.service.mobiliser-brand-
rest-core
ID: 118 Bundle:com.sybase365.aims.webui.web-core
ID: 120 Bundle:com.sybase365.mobiliser.brand.webadmin.mobiliser-brand-
webadmin-ui
ID: 121 Bundle:com.sap.example.customState
```

**5. To see more information about the `com.sap.example.customState` bundle, run:**

```
aims:template 121
```

**Verifying Bundle Configuration Using the AIMS Web Console**

You can use the AIMS System Web console to verify that state bundle configuration changes are in effect.

**Prerequisites**

Enable the AIMS System Web console.

### Task

1. In a Web browser, connect to `http://localhost:8080/system/console`.
2. In the AIMS System Web console, enter these credentials:
  - User name – `sybase365`
  - Password – `fr4nt1c`
3. Select the **Configuration Status** tab, then select the **Configuration** tab.  
You see all state-bundle configurations.

### Custom State Bundle Samples

Many custom-state implementations are based on a service-oriented architecture, in which the custom states consume existing Web services, either SOAP or Representational State Transfer (REST)ful types. States can either get results from one Web service, or they can aggregate results from multiple Web service calls.

#### Consuming SOAP Web Service Sample

A custom state can consume an external SOAP Web service.

The Web service provider in this sample is the United States Consumer Product Safety Commission. The WSDL file (`CPSCUpcSvc.wsdl`) is embedded with the bundle.

Alternately, you can retrieve the WSDL file in real time using the `<wsdlUrls>` configuration. The JAX-WS Maven plug-in reads the WSDL file and generates all the required artifacts for Web service development, deployment, and invocation.

#### *pom.xml*

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">

<build>
[...]
<!-- Create an OSGi Bundle Manifest -->
<plugins>
<plugin>
[...]
  <configuration>
  [...]
    <Private-Package>
      com.sap.example
      ,org.tempuri
    </Private-Package>
  [...]
  </configuration>
</plugin>
</plugins>
</build>

<profiles>
```

```

<!-- Required Javax Persistence dependency -->
<profile>
  <activation>
    <jdk>[1.5, 1.7)</jdk>
  </activation>
  <dependencies>
    <dependency>
      <groupId>org.eclipse.persistence</groupId>
      <artifactId>javax.persistence</artifactId>
      <version>2.0.4.v201112161009</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <repositories>
    <repository>
      <id>EclipseLink</id>
      <url>http://download.eclipse.org/rt/eclipselink/maven.repo</url>

    </repository>
  </repositories>
</profile>

<!-- Required SOAP Web Service JAX-WS only on JDK 6 -->
<profile>
  <id>jdk6</id>
  <activation>
    <jdk>1.6</jdk>
  </activation>
  <build>
    <plugins>
      <plugin>
        <groupId>org.jvnet.jax-ws-commons</groupId>
        <artifactId>jaxws-maven-plugin</artifactId>
        <version>2.1</version>
        <executions>
          <execution>
            <id>import-wsdl</id>
            <phase>generate-sources</phase>
            <goals>
              <goal>wsimport</goal>
            </goals>
            <configuration>
              <wsdlFiles>
                <wsdlFile>CPSCUpSvc.wsdl</wsdlFile>
              </wsdlFiles>
              <extension>true</extension>
              <xdebug>true</xdebug>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</profile>

```

## Developing Custom Application States

```
<!-- Required SOAP Web Service JAX-WS only on JDK 7 -->
<profile>
  <id>jdk7</id>
  <activation>
    <jdk>1.7</jdk>
  </activation>
  <build>
    <plugins>
      <plugin>
        <groupId>org.jvnet.jax-ws-commons</groupId>
        <artifactId>jaxws-maven-plugin</artifactId>
        <version>2.2</version>
        <executions>
          <execution>
            <id>import-wsdl</id>
            <phase>generate-sources</phase>
            <goals>
              <goal>wsimport</goal>
            </goals>
            <configuration>
              <wsdlFiles>
                <wsdlFile>CPSCUpcSvc.wsdl</wsdlFile>
              </wsdlFiles>
              <extension>true</extension>
              <xdebug>true</xdebug>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</profile>
</profiles>
</project>
```

### *SampleSOAPState.java*

```
package com.sap.example;
[...]
import org.tempuri.CPSCUpcSvc;
import org.tempuri.GetRecallByWordResponse.GetRecallByWordResult;

public class SampleSOAPState extends SmappStatePlugin {

    @Override
    protected SmappState processStateLogic(SmappStateProcessingContext
context,
                                           SmappStateProcessingAction action)

        throws MwizProcessingException, DBException {
        CPSCUpcSvc recallService = null;
        String serviceUrl = "http://www.cpsc.gov/cgi-bin/CPSCUpcWS/
CPSCUpcSvc.asmx?WSDL";
        try {
            recallService = new CPSCUpcSvc(new URL(serviceUrl),
```



```

        new QName("http://tempuri.org/", "CPSCUpSvc"));
    } catch (MalformedURLException mfue) {
        [...]
    }
    if (null == recallService) {
        return continueFail();
    }
    String keyword = "booster";
    GetRecallByWordResult recallServiceResult =
recallService.getCPSCUpSvcSoap12().getRecallByWord(keyword, "", "");

    if (null == recallServiceResult) {
        return continueDyn(1);
    }
    return continueOk();
}
}

```

### Consuming RESTful Services

Custom states that consume external RESTful Web services can use the Restlet API.

These Restlet bundles are included with Brand Mobiliser, so you need not copy them when you install bundles. For information about using the Restlet API, see [www.restlet.org](http://www.restlet.org).

#### *org.restlet-2.10.13.jar*

```

<groupId>org.restlet.jee</groupId>
<artifactId>org.restlet</artifactId>
<version>2.0.13</version>

```

#### *org.restlet.ext.servlet-2.0.13.jar*

```

<groupId>org.restlet.jee</groupId>
<artifactId>org.restlet.ext.servlet</artifactId>
<version>2.0.13</version>

```

#### *org.restlet.ext.spring-2.0.13.jar*

```

<groupId>org.restlet.jee</groupId>
<artifactId>org.restlet.ext.spring</artifactId>
<version>2.0.13</version>

```

#### *org.restlet.ext.json-2.0.13.jar*

```

<groupId>org.restlet.jee</groupId>
<artifactId>org.restlet.ext.json</artifactId>
<version>2.0.13</version>

```

### Developing Quick-Start Templates

You can develop custom states to enhance application capabilities, such as integration with existing enterprise systems or cloud services. To demonstrate functionality, include sample

applications in state bundles, which appear in the Brand UI as quick-start templates that you can import into Brand Mobiliser.

### Prerequisites

1. Develop custom states and deploy them to Brand Mobiliser.
2. Develop one or more sample applications that use the custom states.
3. Export applications to an XML file. An XML file can contain multiple applications.

---

**Note:** Each XML file creates one quick-start template. Each custom-state bundle can contain multiple quick-start templates.

---

### Task

Quick-start templates provide commonly used applications that you can customize to meet specific customer needs. You can also create a quick-start template that includes a group of applications to meet a specific functionality, for example, Mobile Wallet.

1. Copy application XML files to META-INF/sample/template.
2. For each XML file, create a dynamic template plug-in.
3. Redeploy the custom-states bundle to Brand Mobiliser.

The Quick-Start Templates component appears on the Brand Mobiliser Web UI Dashboard.

### See also

- *Creating Applications from Templates* on page 27

### Creating Dynamic Template Plug-Ins

To create a dynamic template that you can plug in to a custom-state bundle, configure the State SDK `SmappTemplateProvider` class as a Spring bean.

This example configures the `SmappTemplateProvider` class for the `GetDate.xml` file, which contains an application that demonstrates how to use the custom state `Get Date`. To configure the `SmappTemplateProvider` class, edit both the `beans-context.xml` and the `services-context.xml` files.

#### *beans-context.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

<!--
*****
```

```

        Beans Configuration
*****
-->
<bean id="SampleState" class="com.sap.example.SampleState">
  <property name="country" value="\${sample.country}"/>
</bean>

<!-- Template -->
<bean id="SampleApplication" class=

"com.sybase365.mobiliser.brand.template.SmappTemplateProvider">
  <property name="name" value="Sample Get Date Application" />
  <property name="description" value="Type: Training.
      A sample application to demonstrate the Get Date
state." />
  <property name="resource" value="classpath:META-INF/template/
GetDate.xml" />
</bean>
</beans>
    
```

### *services-context.xml*

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:osgi="http://www.eclipse.org/gemini/blueprint/schema/
blueprint"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
  http://www.eclipse.org/gemini/blueprint/schema/blueprint
  http://www.eclipse.org/gemini/blueprint/schema/blueprint/gemini-
blueprint-1.0.xsd">

  <!--
  *****
  Register state as OSGi Service
  *****
  -->
  <osgi:service id="SampleStateService"
    ref="SampleState"
    interface=

"com.sybase365.mobiliser.brand.plugins.api.smapp.StatePlugin"/>

  <!--
  Template Service
  -->
  <osgi:service id="SampleApplicationService"
    ref="SampleApplication"
    interface=

"com.sybase365.mobiliser.brand.plugins.api.smapp.SmappTemplate"
    context-class-loader="service-provider"/>
</beans>
    
```

## Custom State Samples

---

Custom state samples illustrate how to implement a service state, a standalone state, and a menu state.

### Sample GetMyWeather State

The `GetMyWeather` sample illustrates a typical custom-state implementation. This type of state is called a service state, because its function is to call a specific Web service (in this case a weather service), and store the results for the application to use. This type of state is commonly integrated with enterprise systems.

```
public class GetMyWeather extends SmappStatePlugin {
    private static final Logger LOG =
        LoggerFactory.getLogger(GetMyWeather.class);

    // Define Input attributes

    private static final TextBoxAttribute inPostCode =
        new TextBoxAttribute("POSTCODE", "Zip or Postal Code", false);

    // Define Output attributes

    private static final OutputAttribute outWeather =
        new OutputAttribute("WEATHER", "Your Weather Synopsis");

    private static Attribute[] stateAttr;

    static {
        stateAttr = new Attribute[] {inPostCode, outWeather};
    }
    private static long STATE_ID = 600000L;

    @Override
    public long getStateId() {
        return STATE_ID;
    }

    @Override
    public String getStateName() {
        return "Example - Get My Weather";
    }

    @Override
    public String getRevisionString() {
        return "1.0.0";
    }

    @Override
```

```

public String getStateNotes() {
    StringBuilder sb = new StringBuilder();
    sb.append("A sample state. When executed, it checks for a ");
    sb.append("Postal/ZIP Code, and returns the weather report for ");
    sb.append(" that area.\n\n Use the following follow up states:\n ");
    sb.append("- OK: Weather report for the area was found\n ");
    sb.append("- FAIL: Unexpected error\n ");
    sb.append("- Dyn -1: Area code entered was not valid\n ");
    sb.append("- Dyn -2: No weather report for the area\n ");
    return sb.toString();
}

@Override
public boolean supportsFailTransition() {
    return true;
}

@Override
protected Attribute[] getStateAttributes() {
    return stateAttr.clone();
}

@Override
protected SmappState processStateLogic(
    SmappStateProcessingContext context,
    SmappStateProcessingAction action)
    throws MwizProcessingException, DBException {

    WeatherResult result = null;

    try {
        // Call the weather Web service
        // Details are Web service specific and therefore
        // are encapsulated in the callWeatherService method

        result = callWeatherService();

        if (result == null)
            return continueFail();

        if (result.status == -1)
            return continueDyn(-1);

        if (result.status == -2)
            return continueDyn(-2);

        // Output attribute

        outWeather.setValue(result.text);
        return continueOk();
    }
    catch (DBException dbex) {
        // Database exception can occur while saving session attributes
        LOG.error("error");
        return continueFail();
    }
}

```

```
}  
}  
}
```

### See also

- *Sample Custom State* on page 80
- *Extending the SmappStatePlugin Class* on page 35
- *Custom State Variables* on page 45

## Sample Custom State

A simple custom state, named `SampleState`, formats the current date.

You can modify the date format in the `properties-context.xml` file. The formatted date is stored in an output variable.

### *SampleState.java*

```
package com.sap.example;  
import java.text.Format;  
import java.text.SimpleDateFormat;  
import java.util.Date;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import com.sybase365.mobiliser.brand.dao.DBException;  
import com.sybase365.mobiliser.brand.jpa.SmappState;  
import  
com.sybase365.mobiliser.brand.plugins.api.smapp.SmappStateProcessingActi  
on;  
import  
com.sybase365.mobiliser.brand.plugins.api.smapp.SmappStateProcessingCont  
ext;  
import  
com.sybase365.mobiliser.brand.plugins.smapp.controls.Attribute;  
import  
com.sybase365.mobiliser.brand.plugins.smapp.controls.OutputAttribute;  
import  
com.sybase365.mobiliser.brand.plugins.smapp.state.SmappStatePlugin;  
import  
com.sybase365.mobiliser.brand.processing.exceptions.MwizProcessingExcept  
ion;  
  
public class SampleState extends SmappStatePlugin {  
    private static final Logger LOG =  
        LoggerFactory.getLogger(SampleState.class);  
    protected static final OutputAttribute outDate =  
        new OutputAttribute("DATE", "Current Date");  
    private static Attribute[] stateAttr;  
    private String country = "";  
  
    public void setCountry(String value) {  
        LOG.debug("Country = " + value);  
        this.country = value;  
    }  
}
```

```

static {
    stateAttr = new Attribute[] {outDate};
}

private static long STATE_ID = 600000L;

@Override
public String getStateNotes() {
    return "A sample state. When executed, it returns the current \n"

        + " date in the format of the configured country.\n\n"
        + "Use the following follow up states:\n"
        + "- OK: date and time in the output variable.\n"
        + "- FAIL: If an error occurs during processing.\n";
}

@Override
public boolean supportsFailTransition() {
    return true;
}

@Override
protected Attribute[] getStateAttributes() {
    return stateAttr.clone();
}

public String getRevisionString() {
    return "1.0.0";
}

public long getStateId() {
    return STATE_ID;
}

public String getStateName() {
    return "Example - Get Date";
}

@Override
protected SmappState processStateLogic(
    SmappStateProcessingContext context,
    SmappStateProcessingAction action)
    throws MwizProcessingException, DBException {

    Format formatter = new SimpleDateFormat("MM dd yyyy");

    if (!country.equalsIgnoreCase("US"))
        formatter = new SimpleDateFormat("dd MM yyyy");

    outDate.setValue(formatter.format(new Date()));
    return continueOk();
}
}

```

### See also

- *Sample GetMyWeather State* on page 78
- *Extending the SmappStatePlugin Class* on page 35

## Sample Custom-Menu State

The contents of `SendSampleMenu.java` and `SampleBean.java` illustrate how to create a custom-menu state.

### *SendSampleMenu.java*

Some details from this sample have been omitted, because they are similar to those in nonmenu custom-state implementations.

```
// Package name and imports have been omitted for clarity
public class SendSampleMenu extends AbstractStateMenuImpl {

    // Other omissions include input and output variable declarations,
    // getRevisionString, getStateId, getStateName, and getStateNotes

    @Override
    protected int getMaxMenuItems () {
        return 4;
    }

    // Similar implementation as getStateAttributes

    @Override
    protected Attribute[] getStateAttributeList() {

        // Assume stateAttr has been defined
        return stateAttr.clone();
    }

    @Override
    protected SmappState init(SmappStateProcessingAction action)
        throws DBException {
        try {
            // Get the menu list from the source: database or service
            // Convert it to the SampleBean list
            // See SampleBean class below

            List<SampleBean> sampleList = getSampleMenuList();

            // Store the list in the session variable
            setMenuListToSession(sampleList);
        }
        catch (DBException dbex) {
            return continueFail();
        }
        catch (Exception ex) {
            return continueFail();
        }
        return null;
    }
}
```



```

    }

    @Override
    protected List<KeyValuePair<String, String>> constructMenuList()
        throws DBException {
        List<KeyValuePair<String, String>> menuList =
            new ArrayList<KeyValuePair<String, String>>();

        for (SampleBean sb : getMenuListFromSession(new SampleBean()))
        {
            keyValuePair = new KeyValuePair<String, String>();
            keyValuePair.setKey(sb.getId());
            keyValuePair.setValue(sb.getStatus());
            menuList.add(keyValuePair);
        }
        return menuList;
    }

    @Override
    protected SmappState saveSessionVariables(String key, String value)
        throws DBException {
        int selectedKey = Integer.parseInt(key);
    }
}

```

### *SampleBean.java*

```

// Package name and imports have been omitted for clarity

public class SampleBean implements BeanConverterInterface<SampleBean> {

    protected String id;
    protected String status;

    public static SampleBean parse (String id, String status) {
        SampleBean sb = new SampleBean();
        sb.id = id;
        sb.status = status;
    }

    @Override
    public String convert(SampleBean sb) {
        StringBuilder sb = new StringBuilder();
        sb.append(sb.getId());
        sb.append("|");
        sb.append(sb.getStatus());
        return sb.toString();
    }

    @Override
    public SampleBean convert(String value) {
        String[] values = value.split("\\|");
        Return SampleBean.parse(values[0], values[1]);
    }

    public String getId() {

```

```
    return id;
}

public String getStatus() {
    return status;
}
}
```

### See also

- *Extending the AbstractDynamicMenu Class* on page 39

## State SDK Core Components

---

You can use State SDK core components when developing custom states. Each component is an OSGi bundle. These components are deployed with Brand Mobiliser, so you need not redeploy them with custom-state components.

### *Plug-in APIs*

The Plug-in APIs include APIs for states, state attributes, and data access objects.

Apache Maven:

```
<groupId>com.sybase365.mobiliser.brand.plugins</groupId>
<artifactId> mobiliser-brand-plugin-api</artifactId>
<name>AIMS :: Object :: Brand Mobiliser Plugin - API</name>
```

File name: mobiliser-brand-plugin-api-1.3.1.jar

### *State SDK*

The State SDK contains state implementation base classes, state input and output controls, and helper classes.

Apache Maven:

```
<groupId>com.sybase365.mobiliser.brand.plugins</groupId>
<artifactId> mobiliser-brand-state-sdk</artifactId>
<name>AIMS :: Object :: Brand Mobiliser Plugin - State SDK</name>
```

File name: mobiliser-brand-state-sdk-1.3.1.jar

### *Security*

The Security APIs support encryption functionality that states use.

Apache Maven:

```
<groupId>com com.sybase365.mobiliser.brand.security</groupId>
<artifactId> mobiliser-brand-security</artifactId>
<name>AIMS :: Object :: Brand Mobiliser Security</name>
```

File name: mobiliser-brand-security-1.3.1.jar

### *Core Objects*

Apache Maven:

```
<groupId>com.sybase365.mobiliser.brand.core</groupId>  
<artifactId> mobiliser-brand-core</artifactId>  
<name>AIMS :: Object :: Brand Mobiliser Core Objects</name>
```

File name: mobiliser-brand-core-1.3.1.jar

### *Persistence APIs and Models*

Apache Maven:

```
<groupId>com.sybase365.mobiliser.brand.database</groupId>  
<artifactId> mobiliser-brand-jpa</artifactId>  
<name>AIMS :: Object :: Brand Mobiliser Persistence</name>
```

File name: mobiliser-brand-jpa-1.3.1.jar



# States Catalog

You can use predefined Brand Mobiliser states to build interactive and event applications.

Each state definition includes:

- Input variables – constant values, or values copied from a variable in the current user session.
- Output variables – allow states to return values.
- Follow-up state OK – the condition that constitutes success.
- Follow-up state OK – the condition that constitutes failure, and possible reasons for the failure.
- Follow-up state dynamic – dynamic conditions that transition to follow-up states.
- State editor – example of the state configuration.
- Notes – additional information about the state.
- Usage – Application Composer screen shot that contains the state.

## Add Subscriber State

---

Adds a subscriber and attributes to the selected subscriber list. You can retrieve a subscriber's MSISDN from a session variable, and set as many as 20 attributes.

### *Input Variables*

- **Subscriber Set** – select a subscriber set from a list.
- **Subscriber MSISDN** – unique key for retrieving a subscriber's attributes.
- **Attribute 1, Attribute 2, ... Attribute 20** – subscriber attributes.

### *Output Variables*

**SUBSCRIBER\_COUNT** – total number of subscribers in the subscriber set, after adding the current one.

### *Follow-up State – OK*

Subscriber was added successfully.

### *Follow-up State – Fail*

Error while adding the subscriber, possibly because:

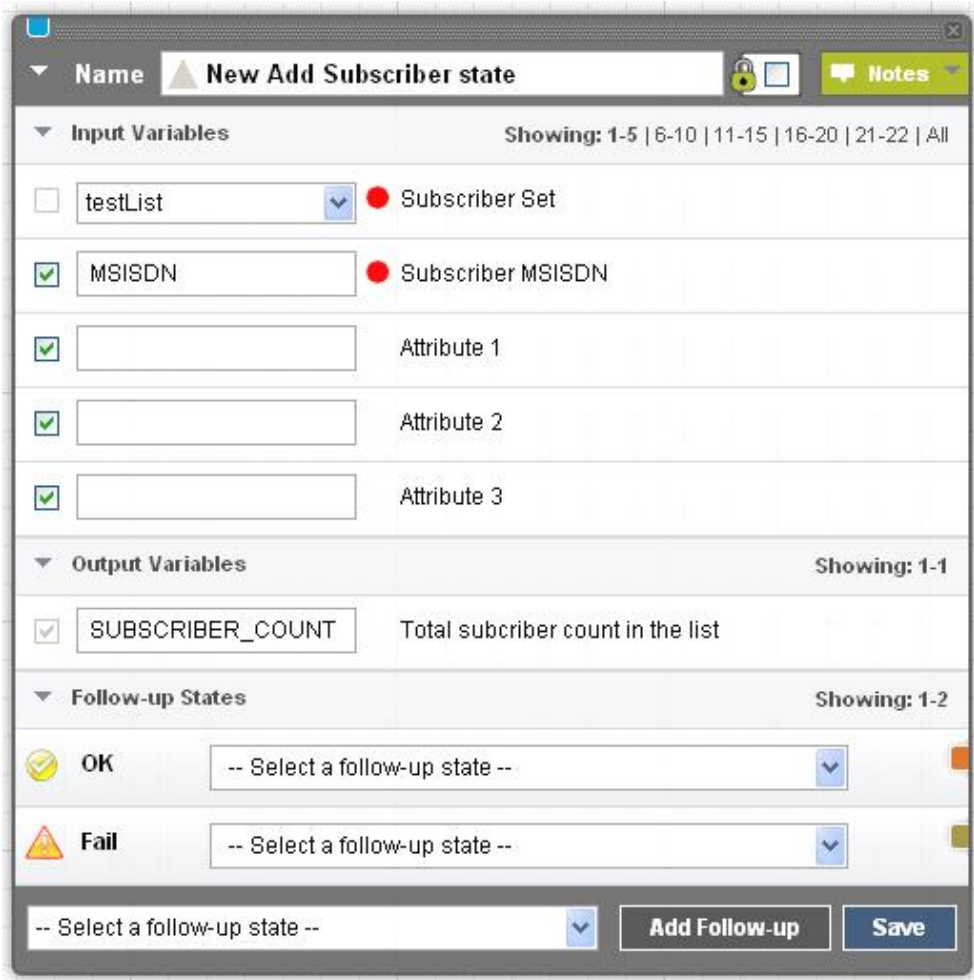
- MSISDN already exists
- Unrecoverable system error, such as a database-connection failure

*Follow-up State – Dynamic*

Not applicable.

*State Editor*

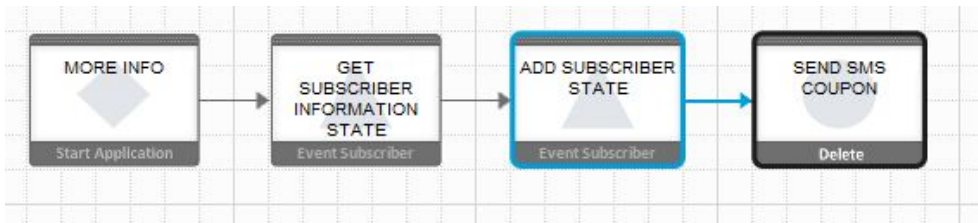
In this example, the New Add Subscriber state adds a subscriber to the testList subscriber set.



*Usage*

A common use for the Add Subscriber state is to store subscribers who opt to receive messages or coupons. For example, in the More Info application, a message is sent to subscribers, and the message contains a reply keyword for interested subscribers. When a subscriber replies

with the keyword, the application retrieves the subscriber's information from the list used in the campaign (Get Subscriber Information state), adds the subscriber to the Opt-In list (Add Subscriber state), and sends a discount coupon to the subscriber.



### See also

- *Get Subscriber State* on page 101
- *Process Subscriber State* on page 106
- *Update Subscriber State* on page 124

## Application Call State

---

Calls another application as a subroutine. The called application has access to session variables, and returns control to the current (calling) application.

### *Input Variables*

**Application** – select an application in the list. All applications in the list are active in the current workspace.

### *Output Variables*

None.

### *Follow-up State – OK*

Not applicable.

### *Follow-up State – Fail*

Not applicable.

### *Follow-up State – Dynamic*

Uses the return value from the Application Call Return state to select which transition to follow.

### *State Editor*

The return value from the called application determines the follow-up state. In the example below:

## States Catalog

- SUCCESS calls Get Agent Information.
- FAILURE calls Invalid Agent Code Format.

The screenshot shows a configuration window for a state named "Validate agent code format". The window has a title bar with standard OS window controls and a "Notes" button. Below the title bar, there is a dropdown menu for "Application" set to "Cash Out Process - Validate Agent Code Format". A section titled "Follow-up States" is expanded, showing two entries. The first entry has a "Target" of "Get agent information", an "Expression" of "SUCCESS", and an empty "Assign To" field. The second entry has a "Target" of "Invalid agent code format", an "Expression" of "FAILURE", and an empty "Assign To" field. At the bottom of the window, there is a dropdown menu for selecting a follow-up state, currently set to "-- Select a follow-up state --", and two buttons: "Add Follow-up" and "Save".

### Notes

Interactive applications only.

### Usage

In this example, customers enter a 6-digit code that identifies an agent, and the code is validated. Because this is a common task, you may want to write the validation procedure as a separate application that returns a status code. Using multiple follow-up states, you can link the return value to the appropriate follow-up state.

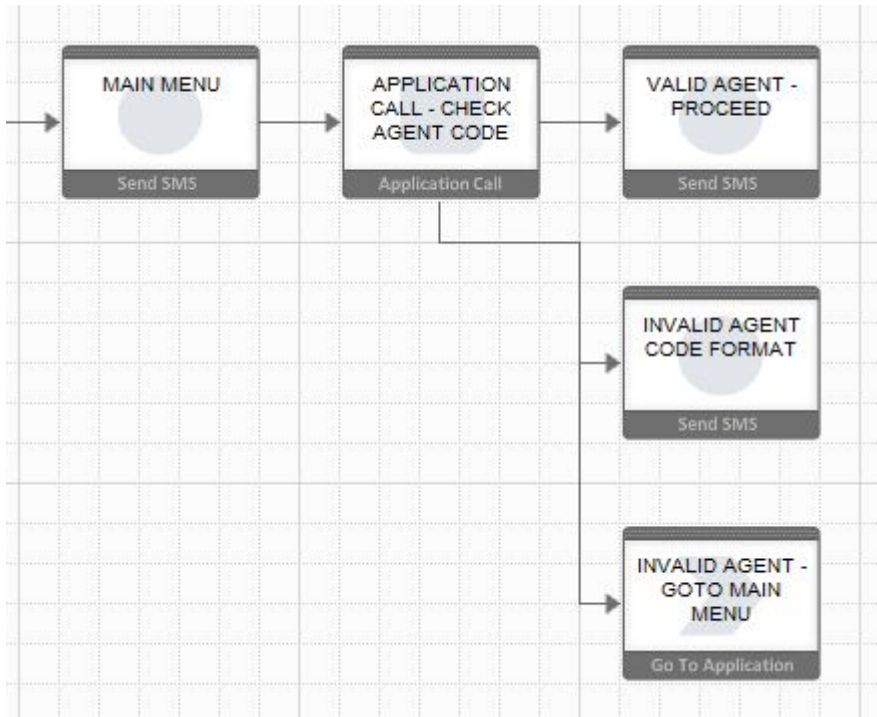


The screenshot shows a window titled "Application Call - Check Agent Code". At the top, there is a "Name" field with the value "Application Call - Check Agent Code" and a "Notes" button. Below this is an "Application" dropdown menu set to "A Manual Mob Sub".

The main section is titled "Follow-up States" and shows "Showing: 1-3". It contains three rows of configuration:

- Row 1:** Target: Valid Agent - Proceed; Expression: 0; Assign To: [empty]; Action: [?]
- Row 2:** Target: Invalid Agent Code Format; Expression: -1; Assign To: [empty]; Action: [?]
- Row 3:** Target: Invalid Agent - Goto Main Menu; Expression: -2; Assign To: [empty]; Action: [?]

At the bottom, there is a dropdown menu with "-- Select a follow-up state --", an "Add Follow-up" button, and a "Save" button.



**See also**

- *Application Call Return State* on page 92
- *Goto Application State* on page 104

## Application Call Return State

---

The final state of applications that are called by other applications. This state returns a value to the calling application.

*Input Variables*

**Return Value** – value returned to the calling application.

*Output Variables*

None.

*Follow-up State – OK*

Not applicable.

*Follow-up State – Fail*

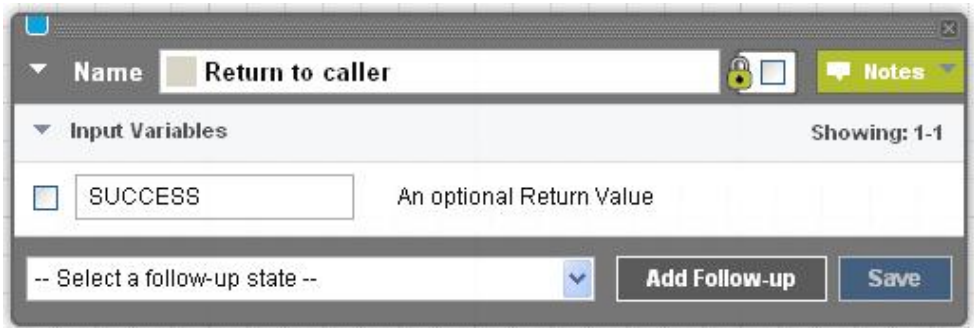
Not applicable.

### Follow-up State – Dynamic

Not applicable.

### State Editor

This state returns the constant value SUCCESS to the calling application.

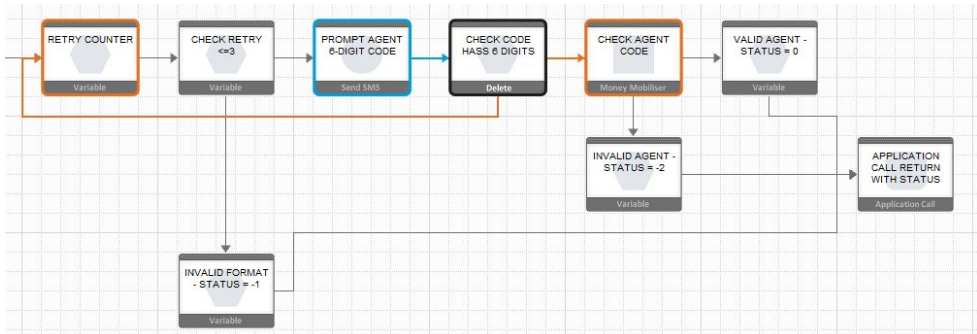


### Notes

Interactive applications only.

### Usage

This application attempts to validate an agent code, and returns three possible values to the calling application.



### See also

- *Application Call State* on page 89

## Compare Typed Variables State

---

Compares two variables of the same type: text, integer, double, or date.

### *Input Variables*

- **Variable Type** – type to compare: text, integer, double, or date.
- **Text Case Sensitive** – whether text comparison is case-sensitive, yes or no; the default is no.
- **Left Variable** – name of the variable on left side of operator. If the corresponding check box is selected, the application assumes **Left Variable** is the name of a session variable; otherwise, the application assumes **Left Variable** is a constant.
- **Operator** – comparison operator; variable type determines valid operators:

Variable Type	Valid Operators
text	=, !=, =REGEX If =REGEX is selected, enter the regular expression as the <b>Right Variable</b> .
integer, double, or date	=, !=, <=, <, >=, >

- **Right Variable** – name of variable on right side of operator (or regular expression). If the corresponding check box is selected, the application assumes **Right Variable** is the name of a session variable, otherwise, a constant.

---

**Note:** If you enter the name of a session variable that does not exist, the state fails.

---

### *Output Variables*

None.

### *Follow-up State – OK*

**Left Variable** equals **Right Variable**.

### *Follow-up State – Fail*

- The values of **Left Variable** and **Right Variable** are not equal, or
- Either **Left Variable** or **Right Variable** is the name of a session variable that does not exist.

### *Follow-up State – Dynamic*

Not applicable.

*State Editor*

In this example, a case-sensitive text comparison is performed for the session variables **TEMP** and **VAR2**. If equal, the follow-up state is Send Variable Values - Equal; if unequal, or either session variable does not exist, the follow-up state is Send Variable Values - Not Equal.

The screenshot shows the 'Compare Variables - Advanced' state editor. The title bar includes a name field with a plus icon, a lock icon, and a 'Notes' button. The main area is divided into two sections: 'Input Variables' and 'Follow-up States'.

**Input Variables Section:**

- TEXT (Variable Type)
- YES (Text Case Sensitive [Default: No])
- TEMP (Left Variable)
- = (Operator)
- VAR2 (Right Variable)

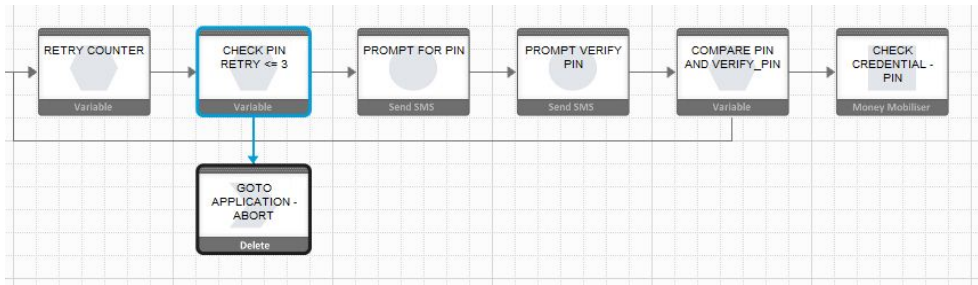
**Follow-up States Section:**

- OK (Send Variable Values - Equal)
- Fail (Send Variable Values - Not Equal)

At the bottom, there is a dropdown menu with the text '-- Select a follow-up state --', an 'Add Follow-up' button, and a 'Save' button.

*Usage*

A common use of the Compare Typed Variables state is in an application that prompts for a PIN, and limits the number of incorrect entries.



### See also

- *Compare Variables State* on page 96

## Compare Variables State

---

Compares the values of two variables, for string equality.

### *Input Variables*

For both input variables, if the corresponding check box is selected, the application assumes the value is the name of a session variable; otherwise, the value is treated as a constant.

- **Variable 1** – name of a session variable, or a constant value.
- **Variable 2** – name of a session variable, or a constant value.

### *Output Variables*

None.

### *Follow-up State – OK*

The values of **Variable 1** and **Variable 2** are equal.

### *Follow-up State – Fail*

- The values of **Variable 1** and **Variable 2** are not equal, or
- Either **Variable 1** or **Variable 2** is the name of a session variable that does not exist.

### *Follow-up State – Dynamic*

Not applicable.

### *State Editor*

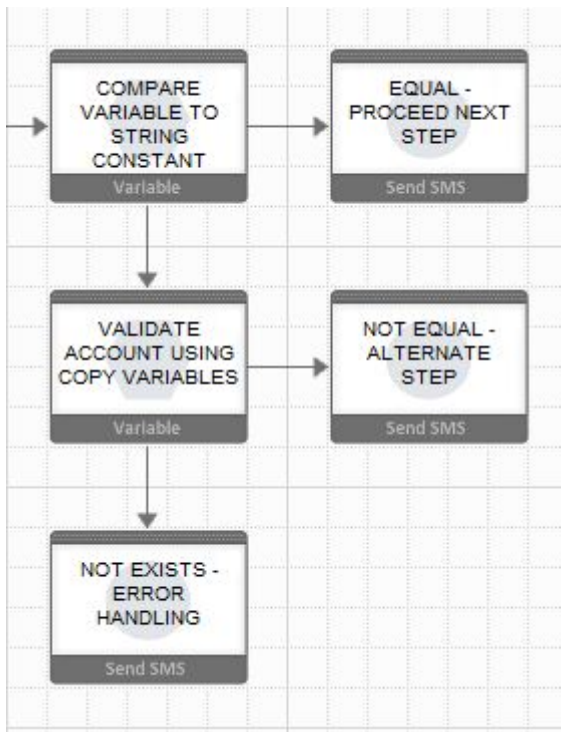
In this example, if the values of **TEMP** and **VAR2** are equal, the application proceeds to the Send Variable Values - Equal state; if unequal, or either session variable does not exist, proceeds to the Send Variable Values - Not Equal state.

*Notes*

This state compares only for string equality. For comparing other types, use the Compare Typed Variables state.

*Usage*

The sample application below compares the session variable **ACCOUNT** to a constant value. If the two values are unequal, the Validate Account Using Copy Variable state is called to copy the **ACCOUNT** session variable to a dummy session variable. If copying fails, the **ACCOUNT** session variable does not exist.



**See also**

- *Compare Typed Variables State* on page 94

## Copy Variables State

---

Copies a constant or the value of a source variable to a session variable.

*Input Variables*

**Source** – the source from which to copy. If source is the name of a session variable, select the check box. Otherwise, the application assumes the value of source is a constant.

---

**Note:** If you specify a session variable that does not exist, the state fails.

---

*Output Variables*

**Destination** – name of the destination session variable. If the session variable does not already exist, it is created.

*Follow-up State – OK*

Successfully copied the source to the destination variable.



*Follow-up State – Fail*

Failed to copy the source to the destination variable, usually because the source variable does not exist.

*Follow-up State – Dynamic*

Not applicable.

*State Editor*

This example copies the value of the session variable **CUST\_BALANCE** into the session variable **PRE\_REMIT\_BALANCE**.

The screenshot shows the State Editor interface for a state named "Copy Customer Balance". The interface is divided into several sections:

- Name:** Copy Customer Balance
- Input Variables:** Showing: 1-1. A single variable, **CUST\_BALANCE**, is listed with a checked checkbox and a red circle labeled "Source".
- Output Variables:** Showing: 1-1. A single variable, **PRE\_REMIT\_BALANCE**, is listed with a checked checkbox and the label "Destination".
- Follow-up States:** Showing: 1-2. Two states are listed:
  - OK:** Txn - Authorise Transaction
  - Fail:** Get Customer Balance
- Bottom:** A dropdown menu with "-- Select a follow-up state --", an "Add Follow-up" button, and a "Save" button.

*Notes*

Session variables are also set in these circumstances:

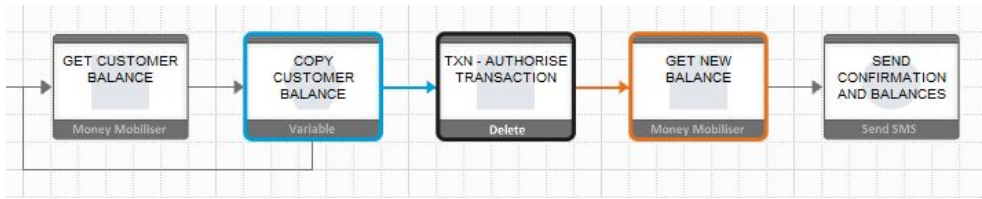
- If you specify a value surrounded by parentheses in the Expression field for a follow-up state, and specify the session variable name in the Assign To field.
- If a state returns values, they are copied to session variables, so they are accessible by follow-up states.

*Usage*

In the sample application below, the customer balance is retrieved twice, before and after calling the transaction. The customer balance is stored in a session variable called Balance. To

## States Catalog

prevent overwriting the pretransaction balance with the posttransaction balance, the application copies the pre-transaction balance into another session variable before calling Get New Balance. If Copy Customer Balance fails, Get Customer Balance is called again.



### See also

- *Set Variable State* on page 120

## Counter State

---

Creates a variable that is incremented by one each time the state is called.

### *Input Variables*

**Variable Name** – name of the session variable to increment. You must select the corresponding check box, or the state fails.

### *Output Variables*

None.

### *Follow-up State – OK*

Not applicable.

### *Follow-up State – Fail*

Fails if variable check box is not selected.

### *Follow-up State – Dynamic*

Determined by the integer **N**, the updated counter.

### *State Editor*

In this example, the Counter state increments the INDEX session variable.

### Notes

The Counter state increments session variables only.

### Usage

You can use the Counter state as an index in a loop; commonly used to allow customers a limited number of retry attempts.

## Get Subscriber State

---

Gets subscriber information from a selected subscriber list. The subscriber's MSISDN is retrieved from the session variable MSISDN. Up to 20 subscriber attributes can be retrieved and assigned to session variables.

### Input Variables

- *Subscriber Set* – select a subscriber set from a list.
- *Subscriber MSISDN* – unique key for retrieving a subscriber's attributes.

### Output Variables

*Attribute 1, Attribute 2, ... Attribute 20* – up to 20 subscriber attributes can be assigned to these session variables.

### Follow-up State – OK

Subscriber attributes successfully retrieved.

## States Catalog

### *Follow-up State – Fail*

Error while retrieving attributes, possibly because:

- MSISDN does not exist.
- Unrecoverable system error, such as database-connection failure.

### *Follow-up State – Dynamic*

Not applicable.

### *State Editor*

This Get Subscriber state retrieves the attributes for the subscriber identified by MSISDN, from the testList subscriber set, and saves attribute values in the output variables.

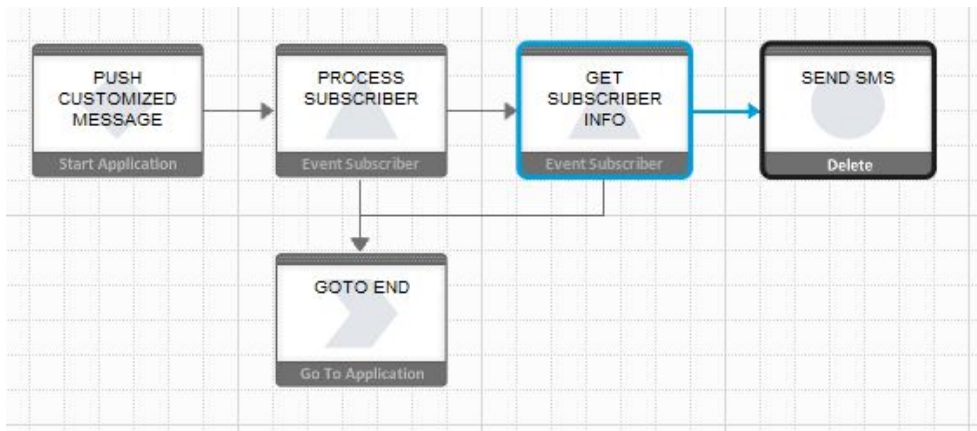
The screenshot displays a configuration window for a state named "New Get Subscriber state". The window is divided into several sections:

- Name:** "New Get Subscriber state" with a lock icon and a "Notes" button.
- Input Variables:** Shows two variables: "testList" (unchecked) and "MSISDN" (checked). Each has a red dot icon and a label: "Subscriber Set" and "Subscriber MSISDN" respectively.
- Output Variables:** Shows five variables: "ATTRIB1" through "ATTRIB5", all checked. Each has a label: "Attribute 1" through "Attribute 5".
- Follow-up States:** Shows two states: "OK" (yellow checkmark icon) and "Fail" (yellow triangle icon). Each has a dropdown menu with the text "-- Select a follow-up state --".

At the bottom of the window, there is a dropdown menu with "-- Select a follow-up state --", an "Add Follow-up" button, and a "Save" button.

### Usage

The Get Subscriber state is typically used with the Process Subscriber state.



### See also

- *Add Subscriber State* on page 87
- *Process Subscriber State* on page 106
- *Update Subscriber State* on page 124

## Goto Application State

---

The final state of an application that transfers control to another application. Session variables are available to the next application.

### *Input Variables*

**Application** – select an application from the list. All applications in the list are active in the current workspace.

### *Output Variables*

None.

### *Follow-up State – OK*

Not applicable.

### *Follow-up State – Fail*

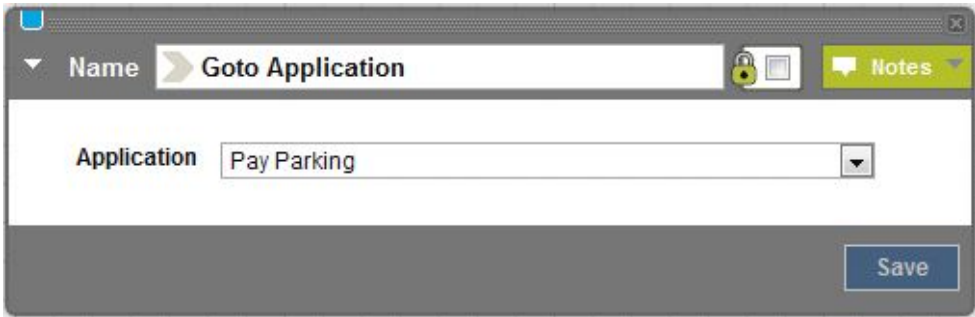
Not applicable.

### *Follow-up State – Dynamic*

Not applicable.

### *State Editor*

This Goto Application state calls the Pay Parking application.



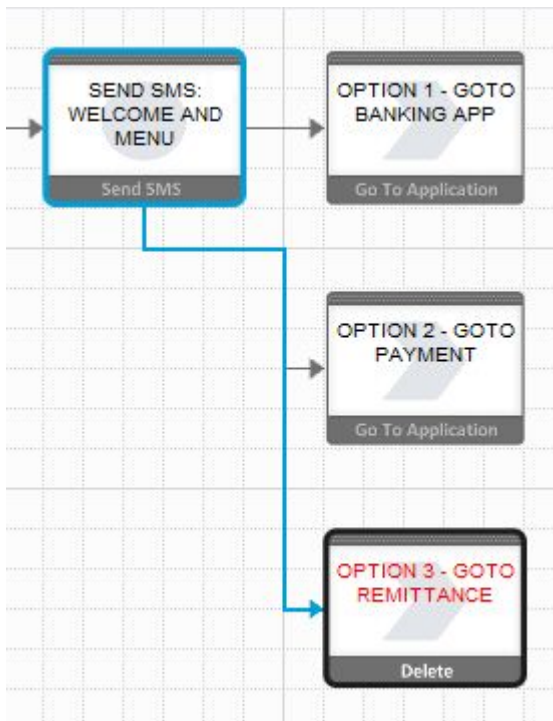
### Notes

The called (Goto) application must be in the same workspace as the calling application.

In event applications, the Goto Application state cannot follow the Process Subscriber state, because the Goto Application state discontinues the loopback mechanism provided by the engine.

### Usage

In this example, the Send SMS state sends a menu to customers, whose selections determine the next (Goto) application.



**See also**

- *Application Call State* on page 89

## Process Subscriber State

---

In event applications, the Process Subscriber state typically retrieves a subscriber from a subscriber set, passes the subscriber information to the Send SMS state, then either returns to get the next subscriber, or ends the application.

*Input Variables*

**Subscriber Set** – select a subscriber set from the list.

*Output Variables*

None.

*Follow-up State – OK*

A subscriber is available to process.

*Follow-up State – Fail*

The event-window processing terminates, because of database connection errors, or other unexpected errors.

*Follow-up State – Dynamic*

- END – the end date for the event window has been reached.
- FINISH – processing terminates because the event window ends.
- COMPLETE – no unprocessed subscribers remain in the list.

---

**Note:** If the state does not handle END, FINISH, and COMPLETE dynamic transitions, the follow-up state is the same as OK.

---

*State Editor*

This sample state processes subscribers in the testList subscriber set. When it successfully retrieves a subscriber from the set, it calls Send Event Message.



The screenshot shows a configuration window for a state named "New Process Subscriber state". It includes a "Name" field, a "Notes" button, and sections for "Input Variables" and "Follow-up States".

- Name:** New Process Subscriber state
- Input Variables:** Showing: 1-1. Includes a checkbox for "testList" and a red circle icon for "Subscriber Set".
- Follow-up States:** Showing: 1-2.
  - OK:** Send Event Message
  - Fail:** -- Select a follow-up state --

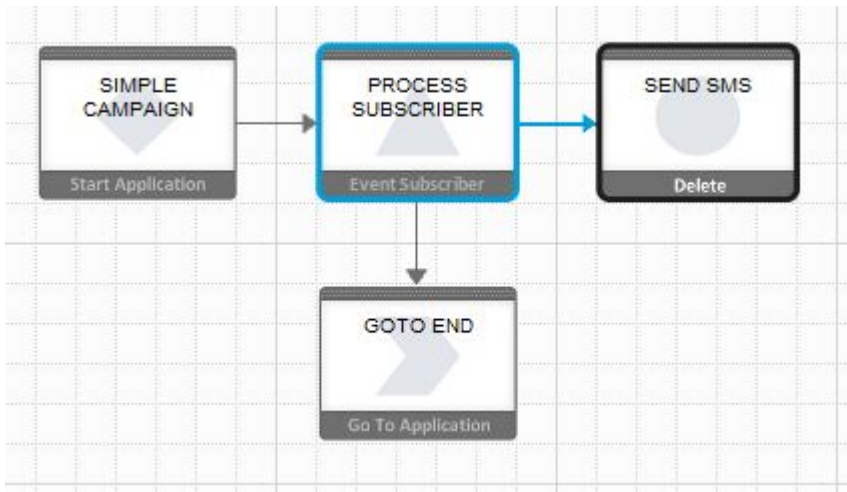
At the bottom, there is a dropdown menu for "Follow-up States" (currently showing "-- Select a follow-up state --"), an "Add Follow-up" button, and a "Save" button.


**Notes**

Event applications only.


**Usage**

This example shows how a simple static-message push campaign gets a subscriber from a set, and sends a message.






**Name**    **Notes**




**Input Variables** Showing: 1-1

  Subscriber Set

**Follow-up States** Showing: 1-3



 **OK**  

 **Fail**  

**Target**    


**Expression**  **Assign To**


-- Select a follow-up state --  **Add Follow-up** **Save**

**Name**    **Notes**

**Message**  **119/160 characters**

**Input Variables** Showing: 1-1

 Request SMPP Acknowledgement

-- Select a follow-up state --  **Add Follow-up** **Save**

**See also**

- *Add Subscriber State* on page 87
- *Get Subscriber State* on page 101
- *Update Subscriber State* on page 124

## Send SMS State

---

Sends short message service (SMS) messages to mobile subscribers. If there is at least one follow-up state, the application waits for a subscriber response; otherwise, the application terminates.

### *Input Variables*

**Message** – text to send via SMS. If the text is more than 160 characters, the text is divided and sent in multiple messages.

To embed the value of a session variable into the text, enter the name of the variable, surrounded by curly braces. For example, if you enter { INDEX }, it is replaced by the value of the session variable **INDEX**. If no such variable exists, { INDEX } is sent as a literal.

In event applications, the Request SMPP Acknowledgement flag appears in the message, requesting acknowledgement from the short message peer-to-peer (SMPP) gateway.

### *Output Variables*

None.

### *Follow-up State – OK*

Not applicable.

### *Follow-up State – Fail*

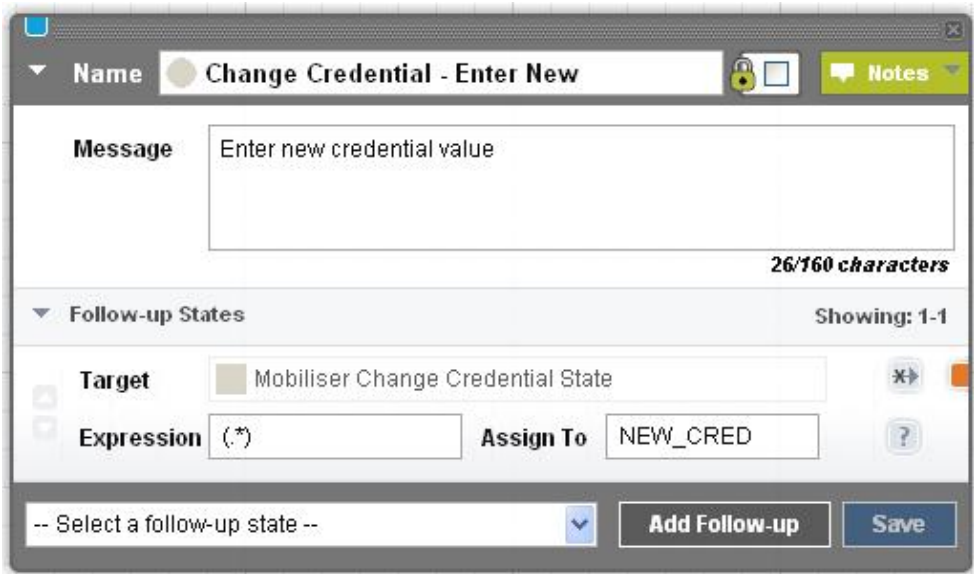
Not applicable.

### *Follow-up State – Dynamic*

Continue the application when a response is received. To determine the follow-up state, compare the response to the values of Expression for follow-up states.

### *State Editor*

This example specifies one follow-up state, the Mobiliser Change Credential state. The value of Expression matches any response, and assigns the response to the **NEW\_CRED** session variable, which can be used later in the task flow.



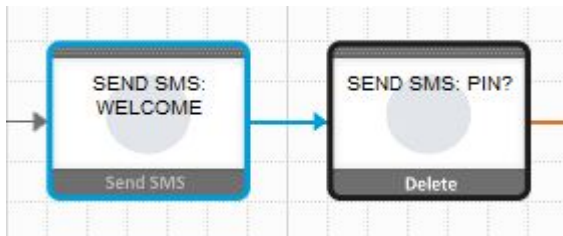
*Notes*

If session variables are embedded in a message, it may be impossible to determine the number of characters in the message prior to runtime.

At runtime, the Send SMS state temporarily suspends the application flow and waits for a response. By default, the wait (also known as session timeout) lasts 7.5 minutes (450 seconds). Once a session times out, responses are ignored. Depending on the setup, subscribers may receive a guidance message or a menu. You can alter the length of the session timeout for each application, on the Application Details screen.

*Usage*

In the scenario illustrated below, the Send SMS state sends a message asking for the subscriber's PIN.



## Send USSD Input State

---

Sends a prompt for input to subscribers using Unstructured Supplementary Service Data (USSD).

### *Input Variables*

All input variables are optional.

- **Input Validation String** – value that can validate expected response values.
- **Input Validation Handler URL** – URL to validate expected response values.
- **Mask the Response** – select Yes or No to mask input on the telephone.

### *Output Variables*

None.

### *Follow-up State – OK*

Not applicable.

### *Follow-up State – Fail*

If an internal problem occurs formatting the state text.

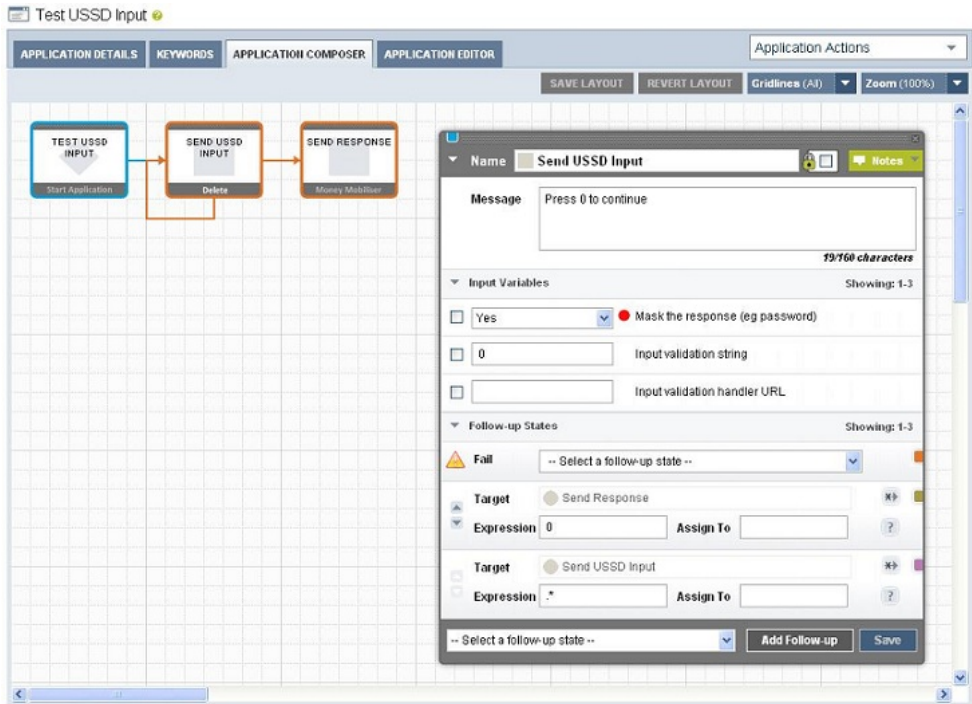
### *Follow-up State – Dynamic*

Continue the application when a response is received. To determine the follow-up state, compare the response to the values of Expression for follow-up states.

### *State Editor*

This example specifies two follow-up states; if the input value is 0, the Send Response state is called; if the input value is anything else, the Send USSD Input state is called again.

# States Catalog



## See also

- *Send USSD Menu State* on page 112
- *Send USSD Text State* on page 118

## Send USSD Menu State

Sends a menu to subscribers via Unstructured Supplementary Service Data (USSD), and expects menu-option responses. This is an abstract state type, which you can extend to develop dynamic menus.

### Input Variables

**Show Exit Menu Item** – enter:

- 1 for yes; this is the default.
- 0 for no.

### Output Variables

- Variable for selected key – name of the session variable in which to store the selected option key.

- Variable for selected value – name of the session variable in which to store the selected option value.

*Follow-up State – OK*

Typically used when the menu is created successfully, and the user sends a valid response.

*Follow-up State – Fail*

Used only if there is an internal error processing the dynamic menu.

*Follow-up State – Dynamic*

To process dynamic transitions, they must be implemented in the state's code.

*State Editor*

In this example, if users send a valid response, another application is called to process the response. If an error occurs, control is passed to an application that terminates processing. The selected option key is stored in the session variable **VAR\_KEY**, and the selected option value is stored in the session variable **VAR\_VALUE**.

The screenshot displays the 'APPLICATION COMPOSER' interface. At the top, there are three tabs: 'APPLICATION DETAILS', 'KEYWORDS', and 'APPLICATION COMPOSER'. Below the tabs is a state flow diagram on a grid background. The flow starts with a state box labeled 'TEST USSD MENU' (Start Application) with a blue border, which points to 'SEND USSD MENU' (Delete) with a black border. From 'SEND USSD MENU', an arrow points to 'APPLICATION CALL YES' (Application Call) with an orange border, and another arrow points down to 'GOTO APPLICATION END' (Go To Application) with an orange border.

Below the diagram is a configuration window for the 'Send USSD Menu' state. The window has a title bar with a close button and a 'Notes' button. The main content is organized into sections:

- Name:** Send USSD Menu
- Input Variables (Showing: 1-1):** A checkbox labeled 'No' is checked, with a dropdown arrow to its right. The label 'Show Exit menu' is to the right of the dropdown.
- Output Variables (Showing: 1-2):**
  - VAR\_KEY Variable name of the selected key
  - VAR\_VALUE Variable name of the selected value
- Follow-up States (Showing: 1-2):**
  - OK Application Call Yes
  - Fail Goto Application End

At the bottom of the window, there is a dropdown menu with the text '-- Select a follow-up state --', an 'Add Follow-up' button, and a 'Save' button.

**Notes**

This state enables you to create a dynamic menu, and present the menu to subscribers as a series of options with relevant responses. The menu items are:



- Header text – enter in the Message input field, as the message header.
- Options – provided programmatically in instances of this state type, by a state developer.
- Paging Options – this state type automatically adds Next and Previous options to a menu list if there are more options than fit on a single page.
- End Option – an option that you can add to end or exit the menu.

### Usage

To implement a dynamic menu, create a subclass that extends this abstract class:

```
com.sybase365.mobiliser.brand.plugins.ussd.impl.AbstractDynamicUssdMenu
```

This abstract superclass creates and structures messages. Subclasses must override and implement abstract methods to provide the required functionality.

```
/* The state attribute list is already set */
protected abstract Attribute[] getStateAttributeList();

/*
 * Initialize the dynamic list, possibly based on subscriber information
 */
protected abstract SmappState init(SmappStateProcessingAction action)
    throws MwizProcessingException, DBException, JAXBException,
    IOException,
    ServiceException, RequiredParameterMissingException;

/*
 * Return the list of options in a format [[key,text],...]
 */
protected abstract List<KeyValuePair<String, String>> getMenuList()
    throws NumberFormatException, DBException,
    RequiredParameterMissingException;

/*
 * Allow the branching of processes based on selected key.
 * If you want to use the configured dynamic follow-up
 * transitions, override this method and return continueDyn(key);
 * otherwise, override this method and return null to follow the
 * OK transition when the user selects an option.
 */
protected abstract SmappState
saveSessionVariables(SmappStateProcessingContext context,
    String key, String value)
    throws MwizProcessingException, DBException,
    RequiredParameterMissingException;
...
```

### See also

- *Send USSD Input State* on page 111
- *Send USSD Text State* on page 118

## **Sample USSD Menu Code**

The code for a sample implementation of the Send USSD Menu state produces a menu with four options: Option 1, Option 2, Option 3, and Option 4.

The `SmappStateSendUssdMenu` class implements the sample USSD menu. The fully qualified class name is:

```
com.sybase365.mobiliser.brand.plugins.ussd.impl.SmappStateSendUssdMenu
```

`SmappStateSendUssdMenu` is a subclass of the `AbstractDynamicUssdMenu` abstract class.

```
package com.sybase365.mobiliser.brand.plugins.ussd.impl;

import com.sybase365.mobiliser.brand.dao.DBException;
import com.sybase365.mobiliser.brand.jpa.SmappState;
import
com.sybase365.mobiliser.brand.plugins.api.smapp.SmappStateProcessingActi
on;
import
com.sybase365.mobiliser.brand.plugins.smapp.controls.Attribute;
import com.sybase365.mobiliser.brand.plugins.useful.KeyValuePair;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * Loads all available languages and puts them into a menu
 *
 */
public class SmappStateSendUssdMenu extends AbstractDynamicUssdMenu
{
    protected static final Logger LOG =
        LoggerFactory.getLogger(SmappStateSendUssdMenu.class);

    private static final String[] OPTIONS =
        { "Option 1", "Option 2", "Option 3", "Option 4" };

    private List<String> listOfOptions = Arrays.asList(OPTIONS);

    private static Attribute[] stateAttr;

    static {
        stateAttr = new Attribute[]{};
    }

    @Override
    protected Attribute[] getStateAttributeList() {
        return stateAttr.clone();
    }
}
```

```

@Override
public long getStateId() {
    return 485002;
}

@Override
public String getStateName() {
    return "Send USSD Menu";
}

@Override
public String getStateNotes() {
    return "This state generates a sample USSD Menu.\n" +
        "Use these follow-up states:\n" +
        "- OK: If user selected a menu item.\n" +
        "- FAIL: If an error occurs.";
}

@Override
public boolean supportsOkTransition() {
    return true;
}

@Override
public String getRevisionString() {
    return "$Revision:28128 $";
}

@Override
protected SmappState init(SmappStateProcessingAction action)
    throws DBException {

    if (listOfOptions == null) {
        return continueFail();
    }

    return null;
}

@Override
protected int getMaxMenuItems() {
    return this.listOfOptions.size();
}

@Override
protected List<KeyValuePair<String, String>> constructMenuList()
    throws DBException {

    List<KeyValuePair<String, String>> list =
        new ArrayList<KeyValuePair<String, String>>();

    int optionNumber = 1;

    for (String option : listOfOptions) {
        KeyValuePair<String, String> keyVal = new KeyValuePair<String,
String>();

```

```
        keyVal.setKey(Integer.toString(optionNumber));
        keyVal.setValue(option);
        list.add(keyVal);
        optionNumber++;
    }

    return list;
}

@Override
protected SmappState saveSessionVariables(SmappStateProcessingContext
context,
                                           String key, String value)
    throws MwizProcessingException, DBException,
RequiredParameterMissingException {
    return null;
}
}
```

## Send USSD Text State

---

Sends a text notification to subscribers via Unstructured Supplementary Service Data (USSD). When subscribers send confirmations, the channel manager passes the messages to the processing engine.

### *Input Variables*

**USSD Session Handling** – select how USSD sessions are managed by the channel manager.

---

**Note:** This option is relevant only when the channel manager is configured to manage USSD session information.

---

The session handling options are:

- None – used when no other option is selected; no specific handling is performed.
- Default – session handling is based on the follow-up state transitions.
- Continue – overrides the default behavior; the channel manager instructs the USSD Gateway with which it is interfacing to continue the USSD session for this user, regardless of whether there are follow-up transitions.
- End – overrides the default behavior; the channel manager instructs the USSD Gateway with which it is interfacing to terminate the USSD session for this user, regardless of whether there are follow-up transitions.

### *Output Variables*

None.

### *Follow-up State – OK*

Not applicable.

*Follow-up State – Fail*

Not applicable.

*Follow-up State – Dynamic*

To determine the follow-up state, compare responses to values of Expression for follow-up states.

*State Editor*

In this example, you specify the text to send to subscribers in the Message field. Notes describe the state functionality and how to use it.

The screenshot shows the State Editor for a state named "Send USSD Text". The interface includes a title bar with the state name, a lock icon, and a "Notes" button. Below the title bar is a green box containing the state's description: "Send USSD Text. This state should be used when interfacing with USSD services through the Channel Manager and JMS. For this configuration, it will send the text message to the JMS queue, with a type of MvMzMtUssdText. Follow up states are used to branch the application based on the customer response. Session variables may be added to the message using syntax of {name}. Use the 'Ussd Session Handling' option to control the outcome of the USSD session after sending the message." Below this is a "Message" field containing the text "This is a USSD text message! <end>" with a character count of "34/160 characters". Underneath is an "Input Variables" section showing "Showing: 1-1" and a dropdown menu for "USSD Session Handling" with options: "-- Select an entry --", "None", "Default", "Continue", and "End". At the bottom right are "Add Follow-up" and "Save" buttons.

*Notes*

To tell the channel manager to end the USSD session, the state appends [`[$ [End] $`] to the message text. The channel manager strips off this text before sending the message to the USSD Gateway.

**See also**

- *Send USSD Input State* on page 111
- *Send USSD Menu State* on page 112

## Set Variable State

---

Sets a session variable with a specified string value. If you specify a numeric value, it is saved as a string.

### *Input Variables*

- **Variable** – name of the session variable to set.
- **Value** – value to save in the session variable. To set **Variable** with the value of another session variable, specify the session variable name as { **sessionVariable** } where **sessionVariable** contains the value to copy.

### *Output Variables*

None.

### *Follow-up State – OK*

The name of the follow-up state after successful processing.

This process always succeeds and moves to the next state.

---

**Note:** This state performs no error checking. Even if the input variables are empty, it proceeds to the follow-up state. SAP recommends that you use the Copy Variables state to set session variables, because it performs input validations, and uses the Fail follow-up state for error handling and debugging.

---

### *Follow-up State – Fail*

Not applicable.

### *Follow-up State – Dynamic*

Not applicable.

### *State Editor*

This example sets the session variable **CREDIT** to 1000. The variable can be accessed by any state in the application.

The screenshot shows a configuration window for a state named "Set MAX Allowable Credit Limit". The window has a title bar with a close button and a "Notes" button. The main area contains two input fields: "Variable" with the value "CREDIT" and "Value" with the value "1000". Below these is a section titled "Follow-up States" which is expanded to show one state: "OK" with a dropdown menu containing "Get Customer Information". A "Save" button is located at the bottom right of the window.

### Notes

Session variables are also set in these circumstances:

- If you specify a value surrounded by parentheses in the Expression field for a follow-up state, and specify the session variable name in the Assign To field.
- If a state returns values, they are copied to session variables, so they are accessible by follow-up states.

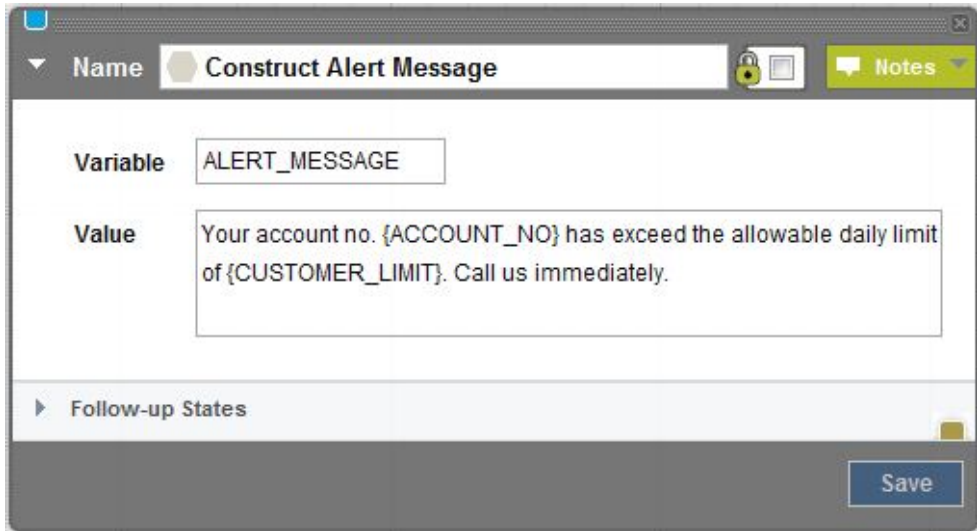
---

**Note:** Setting session variables overwrites any values that are already set for them. For example, if a state returns a value in the session variable X, and the follow-up state also sets variable X, the return value is lost. To avoid this issue, use the Copy Variables state, instead of Set Variable.

---

### Usage

This example sets the session variable **ALERT\_MESSAGE** with a message sent by the Send SMS state.



**See also**

- *Copy Variables State* on page 98

## Start Application State

---

The Start Application state is the initial state in applications. It is created automatically, and cannot be deleted.

*Input Variables*

None.

*Output Variables*

None.

*Follow-up State – OK*

Not applicable.

*Follow-up State – Fail*

Not applicable.

*Follow-up State – Dynamic*

Keywords sent by subscribers initiate applications. An application can have multiple keywords. Dynamic transitions enable custom flows, and are based on incoming keywords.

*State Editor*

A Start Application state with a single follow-up state, Send SMS: Welcome and Menu.



Name **Start Application State** Notes

Follow-up States Showing: 1-1

Target

Expression  Assign To

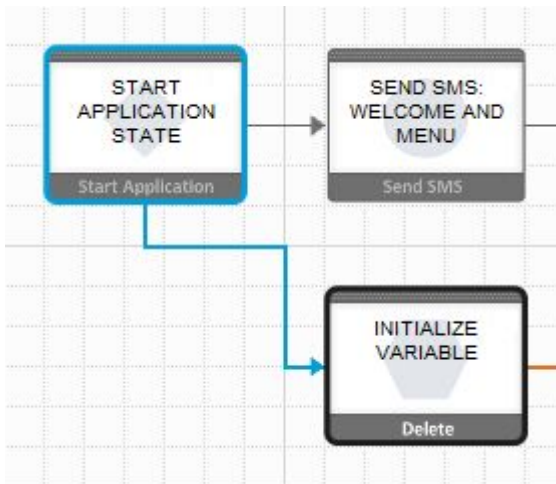
-- Select a follow-up state -- Add Follow-up Save

*Notes*

At least one follow-up state is required.

*Usage*

In this example, the Start Application state processes multiple keywords using different task flows.



## Update Subscriber State

---

Updates subscriber attributes in the selected subscriber set. Gets the subscriber's MSISDN from a session variable, and updates as many as 20 attributes.

### *Input Variables*

- **Subscriber Set** – select a subscriber set from a list.
- **Subscriber MSISDN** – unique key for retrieving a subscriber's attributes.
- **Attribute 1, Attribute 2, ... Attribute 20** – subscriber attributes.

### *Output Variables*

None.

### *Follow-up State – OK*

Subscriber updated successfully.

### *Follow-up State – Fail*

Error while updating the subscriber, possibly because:

- MSISDN already exists.
- Unrecoverable system error, such as database-connection failure.

### *Follow-up State – Dynamic*

Not applicable.

### *State Editor*

In this example, the Update Subscriber state updates attributes for subscribers in the testList subscriber set.

**Name**   **Notes**

**Input Variables** Showing: 1-5 | 6-10 | 11-15 | 16-20 | 21-22 | All

<input type="checkbox"/>	<input type="text" value="testList"/>	<input type="checkbox"/>	Subscriber Set
<input checked="" type="checkbox"/>	<input type="text" value="MSISDN"/>	<input type="checkbox"/>	Subscriber MSISDN
<input checked="" type="checkbox"/>	<input type="text"/>	<input type="checkbox"/>	Attribute 1
<input checked="" type="checkbox"/>	<input type="text"/>	<input type="checkbox"/>	Attribute 2
<input checked="" type="checkbox"/>	<input type="text"/>	<input type="checkbox"/>	Attribute 3

**Follow-up States** Showing: 1-2

<input checked="" type="checkbox"/>	<b>OK</b>	<input type="text" value="-- Select a follow-up state --"/>
<input type="checkbox"/>	<b>Fail</b>	<input type="text" value="-- Select a follow-up state --"/>

**Notes**

None.

**Usage**

One possible use for the Update Subscriber state is a voting application, in which a voter is added to the Voting Results list, and subsequently, the Update Subscriber state can insert information in other fields.

**See also**

- *Add Subscriber State* on page 87
- *Get Subscriber State* on page 101
- *Process Subscriber State* on page 106



# Index

## A

- AbstractDynamicMenu class 39
  - life cycle 40
- AbstractDynamicUssdMenu class 116
- AbstractStateMenuImpl class, extending 82
- accessing input variables 50
- activating
  - applications 21
  - events 21
- activation 19
- Add Subscriber state 87
- adding
  - states to applications 10
- advanced interactive messaging server 1
- AIMS 1
- AIMS System Web console
  - enabling 69
- Apache Maven
  - configuring 53
  - creating a project 57
  - customizing POM files 59
  - installing 53
  - project artifacts, creating 63
  - project structure 58
  - projects, building 64
  - sample POM file 60
  - setting up 53
- Application Call Return state 92
- Application Call state 89
- Application Composer 5
- application states 2
  - adding 10
  - base 3
  - configuring in Application Composer 5
  - custom, developing 33
  - custom, developing and deploying 35
  - developing with PluginInterface 38
  - developing with StatePlugin interface 36
  - dynamic menu 39
  - editing 9
  - editing properties 10
  - extending SmappStatePlugin 35
  - implementing state logic 41
  - removing 11
  - Send SMS 109

- Send USSD Input 111
- Send USSD Menu 112
- Send USSD Text 118
- transitions, removing 11
- troubleshooting, variables for 46
- applications
  - activating 21
  - adding keywords to 12
  - assigning events to 19
  - Cash-Out interactive sample 28
  - deactivating 22
  - deleting 22
  - designing task flows 13
  - event, developing 15
  - event, testing 25
  - exporting 27, 28
  - exporting a group 28
  - importing 26
  - importing XML files 26
  - interactive, developing 11
  - interactive, testing 23
  - life cycle 33
  - mode transitions 20
  - samples 28
  - testing 22
  - uploading templates 27
- applications, samples
  - Utility Notification event 31
- assigning events to applications 19
- Attribute class hierarchy 52

## B

- base states 2, 3
  - Application Call 89
  - Application Call Return 92
  - Compare Typed Variables 94
  - Compare Variables 96
  - Copy Variables 98
  - Counter 100
  - Goto Application 104
  - Send SMS 109
  - Set Variable 120
  - Start Application 122
- bean properties
  - configuring 65

## Index

- BeanConverterInterface 51
- beans-context.xml 65
- Brand Mobiliser
  - overview 1
- building
  - custom-state bundles 56
  - Maven projects 64
- bundles
  - custom states 56
  - verify deployment using Telnet 67
  - verify deployment using Web console 68
  - verifying configuration using Web console 71

## C

- Cash-Out interactive application 28
  - state editor 29
- channel manager 1
- ChannelPlugin interface 38
- class hierarchy, Attribute 52
- classes
  - AbstractDynamicMenu 39, 51
  - AbstractDynamicUssdMenu 116
  - AbstractStateMenuImpl 82
  - InputAttribute 47
  - OutputAttribute 45, 49
  - SampleSOAPState 72
  - SampleState 80
  - SendSampleMenu 82
  - SessionVariableAttribute 51
  - SmappStatePlugin 35, 36, 41, 80
  - SmappStateProcessingAction 42
  - SmappStateSendUssdMenu 116
  - SmappTemplateProvider 76
  - StateUtils 33
  - TextBoxAttribute 45, 47
- code samples
  - USSD menu 116
- Compare Typed Variables state 94
- Compare Variables state 96
- components
  - custom states 56
  - State SDK 84
- configuring
  - Apache Maven 53
  - bean properties 65
  - custom-state bundles 69
  - Spring beans 65
- constructMenuList method 39, 40
- consuming RESTful Web services 75

- consuming SOAP Web services 72
- continueProcessing method 42
- continueWhenSingleEntry method 40
- controlling state transitions 7
- Copy Variables state 98
- Counter state 100
- creating
  - Apache Maven project 57
  - applications from templates 27
  - custom-state bundles 56
  - event applications 15
  - events 17
  - interactive applications 11
  - Maven project artifacts 63
- creating applications
  - Application Composer 5
- custom application states, developing 33
- custom state information 43
- custom states 4
  - developing and deploying 35
  - dynamic menu 39
  - GetMyWeather 78
  - implementing logic 41
  - variables 45
- custom-menu state, sample 82
- custom-state bundles 56
  - adding quick-start templates to 75
  - building 56
  - configuring 69
  - deploying 67
  - verifying configuration using Telnet 70
- customizing
  - Maven POM files 59

## D

- deactivating
  - applications 22
  - events 22
- defining
  - input variables 47
  - output variables 49
  - state variables 45
- deploying
  - custom-state bundles 67
  - State SDK bundles to Maven repository 55
- developing
  - applications, overview 1
  - custom application states 33
  - custom states with PluginInterface 38

- custom states, troubleshooting variables 46
- event applications 15
- interactive applications 11
- states by extending `SmappStatePlugin` 35
- states with `StatePlugin` interface 36
- developing and deploying custom states 35
- dynamic menu state 39
- dynamic template plug-ins
  - creating 76

## E

- editing
  - state properties 10
  - states 9
- enabling
  - AIMS System Web console 69
- event applications
  - developing 15
  - sample message log 26
  - testing 25
  - Utility Notification sample 31
- event engine 1
- event windows
  - one time 17
  - recurring 18
- events 16
  - activating 21
  - assigning to applications 19
  - creating 17
  - deactivating 22
- exporting
  - applications 27
  - group of applications 28
  - single application 28
- extending `SmappStatePlugin` class 35

## G

- Get Subscriber state 101
- `getInputValue` method 50
- `getInputValueWithWarning` method 50
- `getList` method 51
- GetMyWeather sample state 78
- `getStateAttributeList` method 39, 40
- `getStateAttributes` method 40
- `getStateDao` method 41
- `getSubscriberDao` method 41
- Goto Application state 104

## I

- implementing state logic 41
  - `SmappStateProcessingAction` 42
  - `SmappStateProcessingContext` 41
- importing
  - application XML files 26
  - applications 26
- `init` method 39
- input parameters 4
- input variables
  - accessing 50
  - defining 47
- `InputAttribute` class 47
- installing
  - Apache Maven 53
- interactive applications
  - Cash-Out sample 28
  - developing 11
  - initial state 122
  - Mobiliser Counter sample 30
  - sample message log 24
  - testing 23
- interfaces
  - `BeanConverterInterface` 51
  - `ChannelPlugin` interface 38
  - `PluginInterface` 38
  - `StatePlugin` 36
  - `StatePlugin` interface 38
- `isAckMessageRequested` method 41
- `isCurrentStateEncrypted` method 41

## K

- keywords 14
  - adding to applications 12
  - searching for 13

## L

- life cycle, application 33
- list variables 51
- long codes 14

## M

- Maven
  - configuring 53

## Index

- customizing POM files 59
- project artifacts, creating 63
- projects, building 64
- sample POM file 60
- Maven projects
  - structure 58
- Maven repository
  - deploying State SDK bundles to 55
- messaging server 1
- methods
  - constructMenuList 39, 40
  - continueProcessing 42
  - continueWhenSingleEntry 40
  - getInputValue 50
  - getInputValueWithWarning 50
  - getList 51
  - getStateAttributeList 39, 40
  - getStateAttributes 40
  - getStateDao 41
  - getSubscriberDao 41
  - init 39
  - isAckMessageRequested 41
  - isCurrentStateEncrypted 41
  - processMessage 33, 36
  - processMessageLogic 40
  - processState 33, 36, 41
  - processStateLogic 36, 40–42
  - saveOutputAttributes 49
  - saveSessionVariables 39, 40
  - setHoldValue 49
  - setList 51
  - setValue 49
  - supportsSendSmsMessage 42
  - terminateProcessing 33, 42
  - waitForMessage 33, 42, 51
- Mobiliser Counter sample application
  - testing 30
- mode transitions 20
- O**
- OSGi services
  - registering states as 66
- output parameters 4
- output variables 49
- OutputAttribute class 49
- OutputAttribute class, example 45
- P**
- parameters 4

- PluginInterface 38
- POM file, sample 60
- Process Subscriber state 106
- processMessage method 33, 36
- processMessageLogic method 40
- processState method 33, 36, 41
- processStateLogic method 36, 40–42
- properties-context.xml 65

## Q

- quick-start templates 27
  - adding to custom-state bundles 75

## R

- registering states
  - as OSGi services 66
- regular expressions
  - controlling state transitions 7
  - testing 8
- removing
  - state transitions 11
  - states from applications 11
- repositories, Maven 53

## S

- sample applications 28
  - Cash-Out interactive 28
  - event message log 26
  - interactive message log 24
  - Mobiliser Counter 30
  - Utility Notification event 31
- samples
  - custom-menu state 82
  - date formatter 80
  - GetMyWeather state 78
  - Maven POM file 60
  - SOAP Web service 72
  - USSD menu code 116
- SampleSOAPState class 72
- SampleState.java 80
- saveOutputAttributes method 49
- saveSessionVariables method 39, 40
- searching for keywords 13
- Send SMS state 109
- Send USSD Input state 111
- Send USSD Menu state 112



- Send USSD Text state 118
  - SendSampleMenu class, sample 82
  - service states 2
  - services-context.xml 66
  - session manager 1
  - SessionVariableAttribute class 51
  - Set Variable state 120
  - setHoldValue method 49
  - setList method 51
  - settings.xml file 53
  - setValue method 49
  - short codes 14
  - SmappStatePlugin abstract class 41
  - SmappStatePlugin class 35
    - extending 78
  - SmappStateProcessingAction 42
  - SmappStateProcessingAction class 42
  - SmappStateProcessingContext 41
  - SmappStateSendUssdMenu class 116
  - SmappTemplateProvider class
    - configuring as a Spring bean 76
  - SOAP Web service sample 72
  - Spring beans
    - configuring 65
    - SmappTemplateProvider class 76
  - standalone states
    - base states 2
    - subscriber states 2
  - Start Application state 122
  - state attributes
    - class heirarchy 52
  - state bundle samples 72
    - RESTful Web service 75
    - SOAP Web service 72
  - state editor 9
  - state machine 5
  - State SDK bundles
    - deploying to Maven repository 55
  - State SDK core components 84
  - state transitions 6
    - controlling with regular expressions 7
  - state variables
    - defining 45
  - StatePlugin interface 36, 38
  - states
    - Add Subscriber 87
    - adding to applications 10
    - application 2
    - Application Call 89
    - Application Call Return 92
    - base 3
    - Compare Typed Variables 94
    - Compare Variables 96
    - Copy Variables 98
    - Counter 100
    - custom 4
    - defining 43
    - Get Subscriber 101
    - Goto Application 104
    - Process Subscriber 106
    - properties, editing 10
    - removing from an application 11
    - Send SMS 109
    - Send USSD Input 111
    - Send USSD Menu 112
    - Send USSD Text 118
    - service 2
    - Set Variable 120
    - standalone 2
    - Start Application 122
    - subscriber 3
    - transitions, removing 11
    - Update Subscriber 124
    - USSD 2, 4
  - StateUtils class 33
  - subscriber states 2, 3
    - Add Subscriber 87
    - Get Subscriber 101
    - Process Subscriber 106
    - Update Subscriber 124
  - supportsSendSmsMessage method 42
- ## T
- task flows
    - applications, designing 13
  - templates, quick start 27
  - terminateProcessing method 33, 42
  - testing
    - applications 22
    - event applications 25
    - interactive applications 23
    - regular expressions 8
  - TextBoxAttribute class, sample 45, 47
  - transitions
    - application modes 20

## Index

### U

Update Subscriber state 124

uploading application templates 27

USSD menu

sample code 116

USSD states 2, 4

Send USSD Input 111

Send USSD Menu 112

Send USSD Text 118

Utility Notification event application 31

### V

variables

input, accessing 50

input, defining 47

list 51

output, defining 49

variables for troubleshooting 46

verifying

bundle configuration using Telnet 70

bundle configuration using Web console 71

deployment using Telnet 67

deployment using Web console 68

### W

waitForMessage method 33, 42, 51

windows, event

one time 17

recurring 18