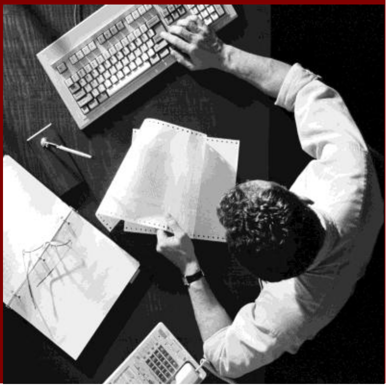


User Guide: Build Tool

SAP® ME Build Tool 6.0

Target Audience

- Project Managers
- Quality Engineers
- Developers

Document Version 1.1 – July 22, 2011

THE BEST-RUN BUSINESSES RUN SAP

THE BEST-RUN BUSINESSES RUN SAP

www.sap.com

© Copyright 2012 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft, Windows, Outlook, and PowerPoint are registered trademarks of Microsoft Corporation.
IBM, DB2, DB2 Universal Database, OS/2, Parallel Sysplex, MVS/ESA, AIX, S/390, AS/400, OS/390, OS/400, iSeries, pSeries, xSeries, zSeries, z/OS, AFP, Intelligent Miner, WebSphere, Netfinity, Tivoli, Informix, i5/OS, POWER, POWER5, OpenPower and PowerPC are trademarks or registered trademarks of IBM Corporation.

Adobe, the Adobe logo, Acrobat, PostScript, and Reader are either trademarks or registered trademarks of Adobe Systems Incorporated in the United States and/or other countries.
Oracle is a registered trademark of Oracle Corporation.

UNIX, X/Open, OSF/1, and Motif are registered trademarks of the Open Group.
Citrix, ICA, Program Neighborhood, MetaFrame, WinFrame, VideoFrame, and MultiWin are trademarks or registered trademarks of Citrix Systems, Inc.

HTML, XML, XHTML and W3C are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

Java is a registered trademark of Sun Microsystems, Inc.

JavaScript is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

MaxDB is a trademark of MySQL AB, Sweden.

SAP, R/3, mySAP, mySAP.com, xApps, xApp, SAP NetWeaver, 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. All other product and service names mentioned are the trademarks of their respective companies. Data contained in this document serves informational purposes only. National product specifications may vary.

These materials are subject to change without notice. These materials are provided by SAP AG and its affiliated companies ("SAP Group") for informational purposes only, without representation or warranty of any kind, and SAP Group shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP Group products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

Disclaimer

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

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

Documentation in SAP Developer Network (SDN)

You can find this documentation in SAP Developer Network (SDN) titled
Simple Use Cases for Advanced Adapter Engine Extended.

THE BEST-RUN BUSINESSES RUN SAP

Typographic Conventions

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

Icons

Icon	Meaning
⚠	Caution
✿	Example
💡	Note
⬆	Recommendation
◇	Syntax

Contents

Introduction

The SAP ME Build Tool (MBT) is a build system used to compile and package extensions for the SAP Manufacturing Execution application. Key features include:

- Support for simple to complex project organization
- Integration into the SDK development environment
- Extension versioning

This guide begins by looking at the big picture and how the MBT fits into the overall development landscape. Here we also introduce key terminology and concepts used in subsequent sections. The remainder of the guide provides task oriented information about how to use the MBT to build your extensions.

Development Landscape

The development landscape is the summation of all software systems and users that collaborate together to produce an extended version of the SAP Manufacturing Execution (ME) application. This landscape can vary depending on how the overall software development project is organized.

Project organization is the grouping of development teams, or so called software vendors, that ultimately contribute extensions to the application. In the simplest case there is only one vendor that is responsible for all development activities, but in many cases the project may add one or more external vendors depending on factors such as staffing availability and skill set. For example, the customer (e.g. the manufacturer that owns the ME application) may use its own internal IT staff for building simple extensions while contracting with an SAP partner for building extensions that require technical skills that they do not possess in-house. Additionally, the customer may also involve SAP Custom Development to create complex back-end business logic that requires more extensive knowledge of the inner workings of the base ME application.

Theoretically the customer can extend the project organization indefinitely by adding more and more external vendors to complement their own development team. As a user of the build system the key point to understand is how your own team's extensions are combined with those created by external vendors. To help illustrate, let's start by looking at a simple scenario where all extensions are developed by a single development team:

Figure 1: Single Vendor Landscape

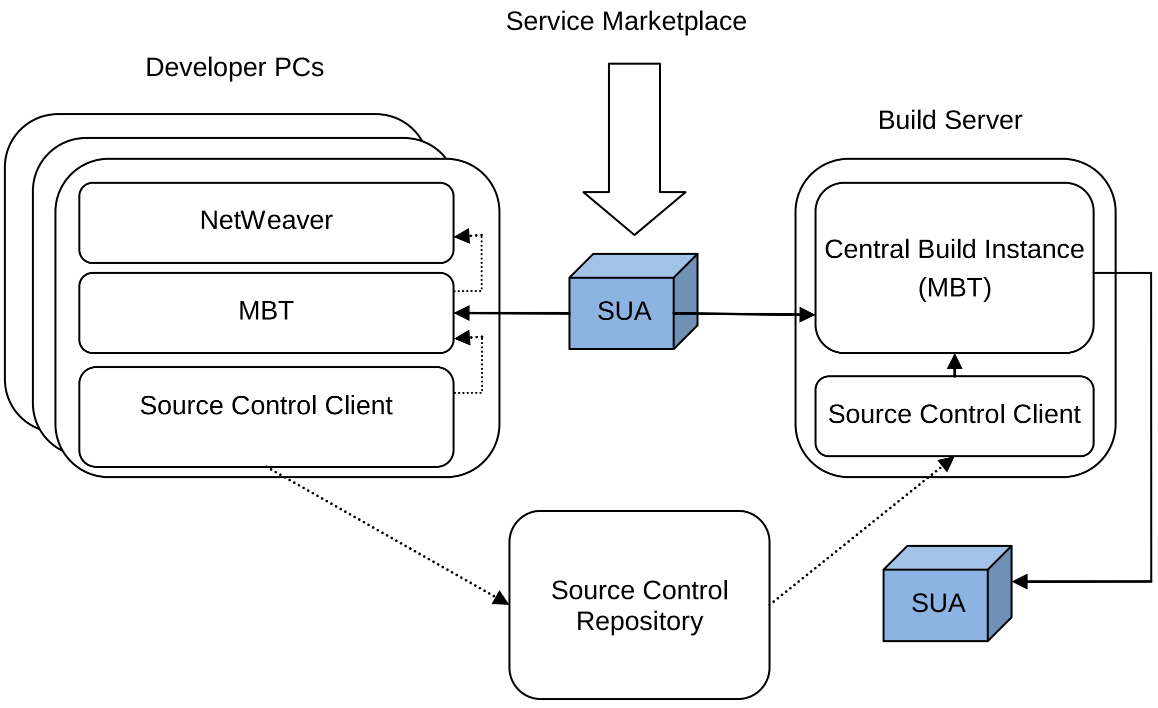


Figure 1 illustrates several important points about the build process within a single development team:

- **The Software Update Archive (SUA) is the unit of shipment for the ME application**

  The base application is downloaded from Service Marketplace and imported into the build system. In actuality there are two types of SUAs, one for client extensions, and one for server extensions, but we won't go into details here. The SUA and other archive types are described in the section Build Achives later in this guide.

- **The final build shipment is created using a stand-alone MBT instance**

Each developer creates and builds extensions in their own development environment using a local MBT instance installed with the ME SDK. Build results are deployed to their local NetWeaver instance. The development build instance is not used to create the final SUA that is delivered to test and productive systems. This function is performed by a stand-alone instance known as the central build instance.

- **The base application must be imported by each build instance**

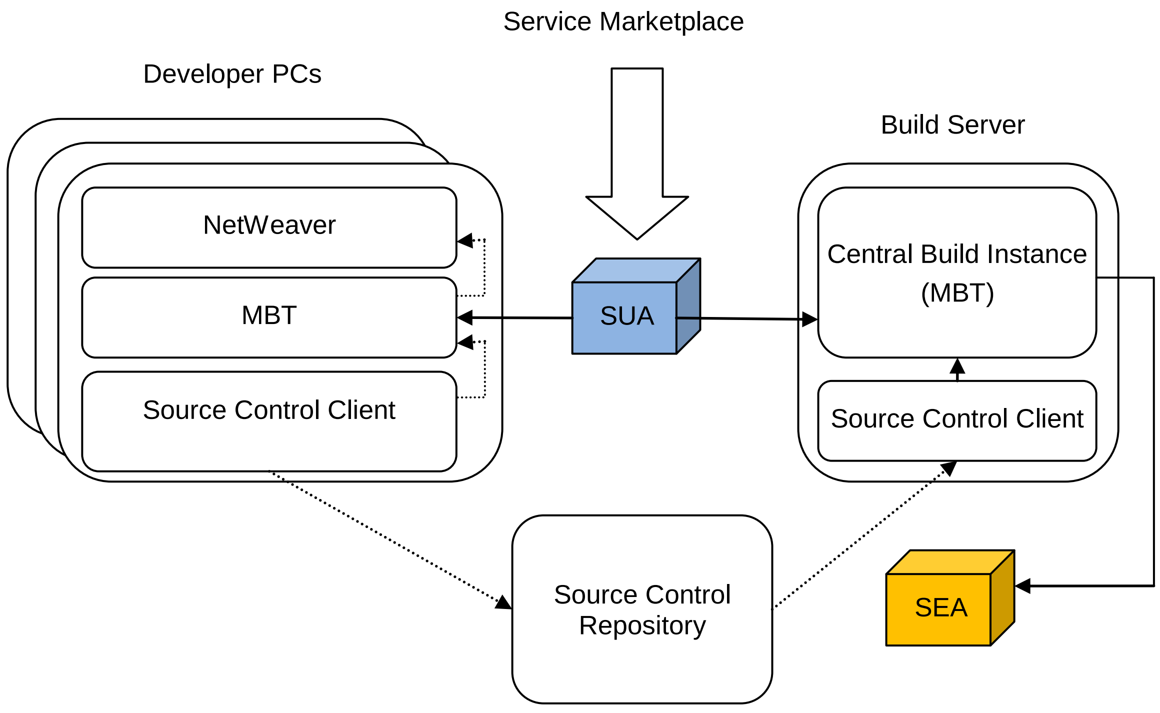
  Before extensions can be built, the build tool must be initialized by importing the base application SUA. The particular version of the base application must be the same for the central build instance and all developer instances. Version coordination should be performed by the project manager to insure that extensions are developed and built using the same ME version that is deployed in the test and production systems.

- **Extension sources are transported via the source control system**

  The source control system acts as the mechanism for synchronizing sources between developers and the central build instance. While sources could be manually copied between systems, this approach is highly discouraged as it is prone to error and misses other important benefits of source control such as file revisioning and change synchronization.

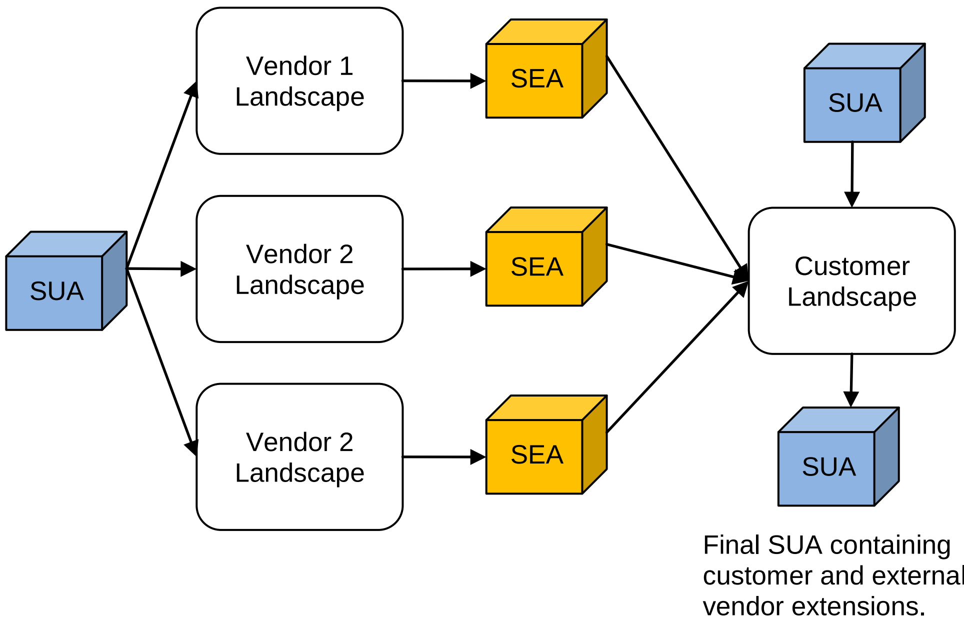
With one important exception, the landscape used by external vendors is identical to that presented in Figure 1. External vendors use their central build instance to create a Software Extension Archive (SEA) instead an SUA. Unlike the SUA, the SEA does not contain the base application, but rather packages only extension binaries and resources for import by another build instance. Thus, SEA export and import provide the mechanisms that allow external software vendors to contribute extensions to the project. Figure 2 illustrates a development landscape of an external software vendor:

Figure 2: External Software Vendor Landscape



Combining the customer landscape with one or more external vendor landscapes gives us the global landscape seen below:

Figure 3: Global Development Landscape



As mentioned before, all MBT instances should use the same base ME version, and this should be coordinated by the project manager. Another important aspect that requires coordination is the assignment of a vendor ID to each software vendor. The vendor ID is a globally unique identifier that is used by the build tool to enforce uniqueness of extensions and the physical artifacts that comprise them. See the *SDK Implementation Guide* for more information about vendor ID assignment.

Build System Operation

Ant

Ant is an open source project developed under the Apache Software Foundation and forms the foundation of the build tool. The Ant architecture is composed of a build runtime for executing the build (itself a Java program executed by a JVM), and a set of pre-defined build functions called "tasks", which are exposed in XML format and serve as an API for constructing high level build targets. Examples of tasks include tasks for copying files, making directories, compiling Java source, and testing for the existence of a file.

While tasks come prepackaged with Ant and form the fundamental build API, targets are defined by the build system developer to handle specific product build requirements. As with tasks, targets are also defined in XML format, and are essentially named functions for orchestrating low level tasks and other targets.

The build tool provides targets that support both development and central build requirements, ranging from the import of the base application to final packaging in preparation for delivery to the test or production system. For more information about MBT targets, see Appendix A: Build Target Reference. For more information about Ant, see the offline documentation installed with the build tool under the /`build/apache-ant-X.X.X/docs` directory.

The Distinction Between "Client" and "Server"

Throughout the build tool and SDK documentation you will see the terms "client" and "server" used often, so here we briefly clarify these terms.

The term "client" is meant to indicate SAP ME client scripts, which are a set of shell scripts and Java classes used to execute system maintenance tasks such as ODS rollup and alarm cleanup.

The term "server" is meant to describe software components that are deployed into the application server, including EJBs, web applications, web services, and supporting libraries and resources.

In general, these terms are used to qualify whether something is specific to the client scripts or the server-side JEE application. For example, the SUA build archive can contain either client or server software components while the SEA build archive may contain both types of components (more on build archives in the following section). As another example, "client" is used in the names of several build targets to indicate that they are specific to client script extensions.

Build Archives

In order to work with the build tool, you should have a good understanding of the various types of build archives that are integral to the build process. The following sections provide a description of the essential build archives imported and produced by the build system.

See Build Phases for more information about how these build archives are used within the build process.

Software Update Archive (SUA)

A Software Update Archive is the single unit of shipment for updating the SAP ME application. There is a different SUA for each of the following update types:

- Full client update
- Full server update

- API server update

Full Updates

Full updates always contain the base client or server SAP ME application and may optionally contain a support package, patch, or a set of extensions (client and server parts are always packaged into separate SUAs). This does not mean that support packages, patches, or extensions are themselves packaged into separate archives within the SUA. In all three cases the actual physical artifacts are co-mingled within the base application.

Client and server SUA files are named according to the following convention:

ME<[Client]>_Base_<ME version>_<NetWeaver version>_Update.zip

Note: “Client” is included in the archive name to distinguish client scripts archives from server archives.

API Updates

API server updates contain new server-side business APIs and supporting resources. New business APIs may be requested if it is determined that the currently available API is not adequate to support extension requirements.

API SUA files are named according to the following convention:

ME_API_<ME version>_Update.zip

See your SAP Client Partner, Customer Engagement Manager, or Alliance Manager for more information about requesting enhancements to the SAP ME API.

Software Extension Archive (SEA)

A Software Extension Archive is the single unit of shipment for delivering SAP ME extensions to another build system instance and does not include base application components. An SEA contains both client and server extensions in a single archive.

SEA files are named according to the following convention:

<vendorID>_SEA.zip

The vendor ID of the software vendor is prepended to the archive file name so that it is clear which vendor published the extensions.

Software Component Archive (SCA)

A software component archive is the unit of shipment and maintenance for delivering software development components for the NetWeaver platform. SCAs are deployable using the Java Support Package Manager (JSPM) tool, or in the case of the SDK development environment, using the Eclipse Deployment perspective.

The SCA is packaged within the SUA in exploaded directory format.

The SAP ME base application and SAP ME web extension application are packaged within separate SCAs. See the *SDK Implementation Guide* for more information about the web extension application.

Software Development Archive (SDA)

A software development archive contains all the artifacts (Java classes, resources, etc) for a single development component. SAP ME defines one development component for the SAP ME EAR and another development component for the SAP ME login module. Each development component is packaged into a separate SDA, and these are packaged within the SAP ME SCA.

Vendor Utility Library

The vendor utility library is a JAR file that contains all Java extension artifacts (Java class files, resources) associated with the following extension types:

- Activity Hooks
- Collaboration Plug-ins
- Print Plug-ins
- Service Plug-ins
- EJB (business interfaces only)
- General supporting classes
- Resources, such as properties bundles or XML files

The vendor utility library is packaged within the SAP ME SDA as an application level library.

Build Phases

The high level execution of the build system consists of the following phases:

- Import
- Build
- Package

The import phase is executed by the `import` build target, and is always executed separately from the build and package phases. Build and package phases are executed together by the `run.production.build` target.

Import Phase

Import is the first build phase that must be run before all other build phases. In the import phase all build archives found under the `/import/sua` and `/import/sea` directories are imported and extracted into a special staging area within the build system. Import removes any previously imported archives, effectively initializing the build system with a new base application and optionally any SEAs contributed by other software vendors. You must run a new import if you want to change the base application version, such as upgrading to a new patch or support package release, or remove, add, or update one or more SEAs.

Note: You must restart eclipse when running import from the development environment.

Build Phase

In the build phase, extensions are compiled, packaged, and copied to the staging area created by the import phase. In the development environment, extensions are also copied to the application server deployment location so that they are available for local testing.

The packaging performed in the build phase should not be confused with packaging performed in the “package” phase. In the build phase, packaging consists of creating the appropriate JAR file depending on the type of extension being built. For example, the development of a new EJB would result in the creation of an EJB JAR containing the EJB implementation class, interfaces, and deployment descriptors.

Once extensions are built, they are ready to be packaged into SCA and SUA archives in the package phase.

Package Phase

In the package phase extensions contributed by all software vendors are assembled into the final SUA. This includes locally developed extensions as well as SEAs contributed from other software vendors.

Extension Export

It is also possible to build and export only extensions (without the base application) by executing the `export.sea` build target. Export is not included as one of the build phases listed since it is only done as an intermediate step when multiple development teams are involved in creating extensions.

During export the build output produces an SEA file containing pre-built extensions, third party libraries, and various resources such as deployment descriptors. The SEA can later be imported into another build tool where its contents will be added to the base application. SEAs are imported as part of the package phase.

All archives produced by the package phase are versioned according to the version information specified in the build configuration. See Configuring the Build for more information about version related configuration files.

Versioning

Every release of SAP ME is versioned according to a four digit versioning strategy used commonly throughout the software industry. Version information is defined using the following format:

```
[major].[minor].[support package].[patch]
```

For supportability reasons, it is important to separately track the base version of SAP ME and the version of each set of extensions contributed by separate software vendors. For example, when a support issue is submitted this version information is recorded by support personnel so that it is available during root cause analysis.

Someone within the development team, usually the project manager, is responsible for maintaining version information for extensions (note that the version of the base application is maintained separately and cannot be changed). This means making decisions about when to change the version number for a given product release. The following rules are provided to serve as a guideline when deciding if and how the version information should be updated:

- **Major Version**: The major version should be incremented when many significant functional changes have been made to the product. The extension major version should be incremented when the base major version of SAP ME is changed.
- **Minor Version**: The minor version should be incremented when a few focused functional changes have been made. The extension minor version should be incremented when the base minor version of SAP ME is changed.
- **Support Package**: The support package should be incremented when there have been no functional changes and a group of patches are available for release. This is commonly referred to as a maintenance release.
- **Patch**: The patch version should be incremented when there have been no functional changes and one or more fixes have been applied to the product. Typically patch releases are used to address critical issues that cannot be postponed.

There is also additional version information reported by the SAP ME application that is taken from the build tool. The full set of version information includes the following:

- The software vendor name
- The four digit version
- The vendor ID
- The build counter (automatically incremented each time the build is run)
- An open field that can be used by software vendors to specify additional information (this information is optional)

If extensions are contributed by separate software vendors, then each set of extensions will receive its own set of version information.

SAP ME displays version information in a summarized format directly on the welcome screen. Detailed version information can be accessed by clicking the *About* menu in the Activity Manager.

For supportability reasons, version information is also logged to the application server log each time the application is deployed.

Packaging Third Party Libraries

Extensions may use libraries created by third party sources such as open source projects, commercial software providers, or even those created by SAP and installed as part of the application server runtime. Libraries that are not included with the server runtime are called "required" libraries and must be packaged with the SAP ME application. Libraries that are made available by the application server runtime are called "provided" libraries since they are already provided at runtime. These libraries are not packaged with the SAP ME application.

Required libraries must be copied to the `/build/lib/ext` directory so that they are available at compile time and are packaged with the application. These libraries are also included when exporting extensions to an SEA file. They are added to the base application when the SEA is imported.

Provided libraries must be copied to the `/build/lib/ext/provided` directory so that they are available at compile time. These libraries are not included when extensions are packaged with the application since they are provided by the applicaton server runtime.

Configuring the Build

Build configuration is contained in two configuration files. The first file is the `custom.properties` file and is located in the `/build/scripts` directory. This file contains several key properties used by the build, most of which are set when the build system is installed.

See Appendix B: Build Setting Reference for a description of all build system properties.

The second file is the `buildID.xml` file, which is located in the /`build` directory. This file contains fields for specifying version information that will be incorporated into various build artifacts.

The following table describes `buildID.xml` elements:

Name	Description
customer	The *customer* field should be set according to the name of the customer that the extensions are being built for. This field is initialized with the value specified during the build installation.
revision	The *revision* field represents the four digit version of all extensions. This value is initialized to 1.0.0.0 when the build is installed. See Versioning for guidelines on when to adjust the version.
Build	This *build* field defines the build count and is automatically incremented each time the **`run.production.build`** target completes successfully. This value is initialized to 0 when the build is installed and will grow indefinitely unless manually modified. This should be reset whenever a new version of extensions is released.

Running the Build

The MBT installation provides a script called `build.cmd` that is used for running the build tool from the command line.

The *build.cmd* is typically used only by users of the central build instance. While this script is delivered as part of the SDK development environment installation, developers run build targets directly from within the IDE.

This script hides the details of running Ant directly, eliminating the need to provide details such as the Java runtime, Ant executable, or Ant build file to run. Typically you will only provide the name of the build script itself followed by the name of one or more targets to run:

`build` *target1 target2 …*

Additional arguments can be passed to the Ant runtime if desired, such as when you want a listing of all public targets (only targets defined with a description are shown in the project listing), or when you want to debug the build by enabling verbose output.

Execute the following command to list the public build targets:

```
build -projecthelp
```

Execute the following command to display debug information while running a build target:

`build -debug` *`target`*

Note: In most cases developers will run the build directly from the IDE. Command line use is intended primarily for use with the central build instance.

Building a Release Candidate

You use a central build instance of the MBT to build a release candidate (SUA) that is later used to update the SAP ME installation.

The central build instance can only be executed from the command line using the `build.cmd` script located in the `/build` directory.

The primary production build target is the `run.production.build` target. This is the only build target you need to run to create the release candidate SUA.

Perform the following steps to create a release candidate:

1. Copy base application client and server SUA files to the `/import/sua` directory. Verify that you are using the correct version of the base application. Both client and server SUAs should be the same version.
2. Copy SEAs contributed by external software vendors to the `/import/sea` directory.
3. Open an operating system shell window and change directory to the `/build` directory.
4. Run the `import` build target. This will import all SUA and SEAs in preparation for building extensions.
5. Synchronize local sources with the build system. Use your source control client to synchronize source code located under the `/extension` directory. This step can be skipped if there are no local extensions.
6. Copy third party libraries used by local extensions to the `/build/lib/ext` directory. This step can be skipped if there are no local extensions.
7. If required, update the `revision` element information in `build.xml`. See Versioning for details about setting version information.
8. Set optional version information in `custom.properties`. This includes the `scs.build.ID` and `ext.info` properties. See Appendix B: Build Settings Reference for more information about build configuration settings.
9. Run the `run.production.build` build target. The resulting client and server SUA files will be written to `/export`. SUA files are used in conjunction with the SAP ME client and server installers to update the SAP ME installation.

Troubleshooting Build Problems

In general there are two types of problems that may occur when running a build. The first type of problem can be categorized as a “system” level problem and is not related to the actual content being built. Examples of system level problems would be the inability to access some file because it is locked by another process (such as the development application server), or running out of memory due to insufficient memory arguments set for the Java compiler. Because system level problems are related to the execution environment of the build system, they can vary widely and are not covered in this section.

The second type of problem can be categorized as a content driven problem. These are problems related to the actual artifact being built, and are seen much more often than system level problems.

Troubleshooting content driven problems is the main focus of this section. We begin by describing the primary means of debugging build issues; the build log.

The Build Log

By default Ant tasks write trace messages to standard out. While this may be sufficient for development environments, it is desirable to have a persistent log of build events when building a release candidate. This allows build results to be communicated easily to development team members should an error occur.

Build results are logged when running the `import`, `run.production.build` or `export.sea` build targets. The log file is named `sdk_build.log` and is located in the /`build` directory. Note that the log file is appended each time the build is run, so you will need to monitor its size and rename it if it becomes large.

Common Issues

Java Compile Errors

Java compile errors are typically the most common type of error encountered. The following illustrates a compile error thrown while compiling a class from Web Application project:

```
C:\SDK\2.0.0.0\Release Candidate\Dev\Oracle\build>build compile

Buildfile: C:\SDK\2.0.0.0\Release Candidate\Dev\Oracle\build\script\build.xml

validate.env:

     [echo] [2009-08-17 08:43:15] Key environment properties validated.

compile:

    [mkdir] Created dir: C:\SDK\2.0.0.0\Release
Candidate\Dev\Oracle\build\results\work\classes

    [javac] Compiling 8 source files to C:\SDK\2.0.0.0\Release
Candidate\Dev\Oracle\build\results\work\classes

    [javac] C:/SDK/2.0.0.0/Release
Candidate/Dev/Oracle/extension/web.war/src/com/vendor/productdefinition/web/con
troller/OperationBrowserController.java:153: ';' expected

    [javac]             model.put("vendor.operationbrowser.reportingStep",
textBundle.getString("vendor.operationbrowser.reportingStep"));

    [javac]                  ^

    [javac] 1 error

BUILD FAILED

C:\SDK\2.0.0.0\Release Candidate\Dev\Oracle\build\script\build.xml:1294:
Compile failed; see the compiler error output for details.
```

Compile errors are the result of programming mistakes made during development and usually require one or more source code modifications to fix. Fixes should always be tested in the development environment before applying to the production build.

Incorrect Keystore Configuration

The build tool configuration file contains several properties for configuring access to the SDK keystore. If any of these properties is not set to the correct value, a build error will result. For example, if the keystore property `security.storepass` was set to an incorrect value, the following build error would result:

```
[signjar] jarsigner error: java.lang.RuntimeException: keystore load:
Keystore was tampered with, or password was incorrect
```

Vendor ID Conflict

Each software vendor must use a unique vendor ID to insure namespace uniqueness among build artifacts. If you attempt to import an SEA file that uses the same vendor ID as your build system or if two SEAs that use the same vendor ID, a build error similar to the following will result:

```
BUILD FAILED

C:\SDK\2.0.0.0\Release
Candidate\Dev\Oracle\build\script\build.xml:3739: Duplicate vendor name
found.  All extension packages must have unique vendor ID assignments.
```

Appendix A: Build Target Reference

The build tool is used differently depending on the type of user that you are. As a developer you will mainly use build targets for importing and building your extensions, and will not use packaging targets very often. As a central build instance user you will primarily use “import”, “run.production.build”, and “export.sea” since these targets execute all build phases needed for creating production build artifacts.

The following table lists all build targets that can be used by both types of users. Any target not listed is considered private and should not be used.

Target Name	**Description**
build	Builds all extensions, if they exist. If there are not extensions found for a particular extension type, then it will be skipped. Build results are written to the appropriate directory under `/build/results/server/sda` (exact location depends on the extension type). Build results are copied to the server runtime deployment location when run in the development environment.
build.client.scripts	Builds only client script extensions. This target is provided as a convenience so that you don’t have to run a full build, including server extensions, just to build client extensions. Build results are written to the `/build/results/client` directory.
build.ejb	Builds all EJBs and packages them in a single EJB jar file. This target automatically executes the “build.lib” target. Build results are written to `/build/results/server/sda/<`*`vendor ID`*`>.ejb.jar`. Build results are also copied to the server runtime deployment location when run in the development environment. Application restart is required for changes to take affect.
build.idat	Copies custom IDAT files to `/build/results/server/sda/idat/custom`. This target is provided as a convenience so that you don’t have to run a full build just to update IDAT files.
build.lib	Builds all extensions that are deployed as an application library. This includes the following extension types: • Activity hooks • Service plug-ins (and `service-config.xml`) • Collaboration plug-ins • EJB business and home interfaces • XSL transform files • IDAT files • Supporting resource files (such as property files) Build results are written to `/build/results/server/sda/APP-INF/lib/<`*`vendor`*

Target Name	Description
	ID>`Lib.jar`. Build results are also copied to the server runtime deployment location when run in the development environment. Application restart is required for changes to take affect.
build.web	Builds all web tier extensions (including the `web.xml` descriptor file) and packages them within the SAP ME web application WAR file. This target automatically executes the `build.lib` target. Build results are also copied to the server runtime deployment location when run in the development environment. Application restart is required for changes to take affect.
build.web.classes	Builds web tier Java sources and packages the resulting classes in the SAP ME web application WAR file. The JSF descriptor `faces-config.xml` is also packaged. Build results are written to: `/build/results/server/sda/me.common.web.war/WEB-INF/lib/<vendor ID>WebLib.jar` Build results are also copied to the server runtime deployment location when run in the development environment. The JSF descriptor `faces-config.xml` is also updated. Application restart is required for changes to take affect.
build.web.config	Builds only web configuration resources and packages them in the SAP ME web application WAR file. The JSF descriptor `faces-config.xml` is not processes by this target. Build results are also copied to the server runtime deployment location when run in the development environment. Application restart is required for changes to take affect.
build.web.content	Builds only web content resources and packages them in the SAP ME web application WAR file. Build results are also copied to the server runtime deployment location when run in the development environment. JSPs will be recompiled automatically upon next access if they are not declared as servlets in web.xml.
build.webservice	Builds web service endpoints, including the generation of classes from WSDL schema declarations, and packages them into the web service WAR file. This target automatically executes the `build.lib` target. Build results are written to `/build/results/server/sda/<`*vendorID*`>-webservice.war`. Build results are also copied to the server runtime deployment location when run in the development environment. Application restart is required for changes to take affect.

Target Name	Description
build.webservice.client.in	Compiles Java web service clients and the proxy classes generated from WSDL files. Build results are written to `/export/<vendorID>-ws-client.jar`.
build.webservice.client.out	Compiles Java web service clients and the proxy classes generated from WSDL files. Build results are written to `/build/results/server/sda/APP-INF/lib/<vendorID>-webservice-client.jar`.
clean	Deletes all build output for all extensions types.
clean.client.scripts	Deletes build output created for client scripts extensions.
clean.compile	Deletes Java compiler output for all extension types.
clean.server	Deletes build output for all extension types except for client scripts build output.
clean.web-ext	Deletes build output for the web extension application.
clean.webservice	Deletes all web service endpoint build output.
clean.webservice.client.in	Deletes all inbound client web service build output.
clean.webservice.client.out	Deletes all outbound client web service build output.
compile	Compiles all server Java source files, including generated source files created for web services. Build results are written to `/build/results/server/classes`.
compile.client.scripts	Compiles all client Java source files. Build results are written to `/build/results/client/classes`.
display.env	Displays key build system properties in the build console.
export.sea	Runs the “build” phase and packages extensions into an SEA file. Note that only extensions are packaged. Base application artifacts are not included. Use this target for exporting extensions so that they can be later imported by the final production build system. Build results are written to `/export` directory. Note that you may need to adjust the build version according to the rules defined in Versioning.
import	Imports all build archives found under the `/import/sua` and `/import/sea` directories into the build system. All previously imported archives are deleted, effectively initializing the build system. It is not considered an error if one or more update archives are not found (messages will be logged indicating that the archive could not be found and that the import will be skipped). Build results are written to the

Target Name	Description
	`/build/results/server/sda` and /`build/results/client` directories.
import.client.scripts	Imports the client scripts SUA from the `/import/sua` directory. This target is provided as a convenience so that you don't have to run a full server SUA import just to import a new client SUA. Build results are written to `/build/results/client`.
package	Packages all extension types into client and server SUA files. A client SUA file will be created only if client extensions exist. SUA files are written to `/export`. The SAP ME SCA is also copied to `/export` when run in development mode.
package.client.scripts	Packages only client scripts extensions into the client SUA file. The SUA will be created only if client extensions exist. Build results are written to `/export`.
package.sca	Packages the base application, server extensions, and extensions contributed by other software vendors into a deployable SCA file. The SCA is written to `/export` when run in development mode.
package.web.ext	Packages the web extension application into an SCA file, including resources contributed from SEA files (must have already been imported). This target does not include a "build" phase since the web extension application does not contain any buildable artifacts (only static web content). The SCA is versioned with the same major, minor, support package, and patch version as ME SCA. However, a separate build counter is used.
package.xslt	Packages XSL files into an archive named `<`*`vendorID`*`>XSLT.zip`. Build results are written to /export.
run.production.build	This is the primary build target used to create the final production build artifacts. This target will execute both "build" and "package" build phases, and will also import all SEAs found in the `/import/sea` directory. This target is intended to be used primarily by users of the central build instance. You must separately import the base ME application before running this target. An import is not required if the base application has already been imported and you will continue to use the same base version. You must import a new base application if you will be delivering a new major, minor, support package, or patch base application release. Build results are copied to the `/export` directory (this location is mapped under the Build Resources Eclipse project). Results can include the ME SCA, the web extension SCA, and client and server SUA files. The web extension SCA and client SUA will be created only if extensions are found for

Target Name	Description
	either. SUA and SCA files are all versioned with the same version, including build counter. Note that you may need to adjust the build version according to the rules defined in Versioning. All build output is logged to `/build/sdk_build.log`.
update.data-sources	Allows you to change ME data source settings based on property defined in the build configuration file (custom.properties) without re-importing the core application. Relevant data source properties include all db.* properties. You must redeploy the SCA for changes to take affect. See Build Settings Reference for more information about individual settings.

Appendix B: Build Settings Reference

The following table describes the set of available build settings:

Name	Description
`vendor.name`	The name of the software vendor organization responsible for the SAP ME extensions. This is typically the company name of the software vendor. The vendor name is added to version information created by the build.
`vendor.id`	The vendor ID is a value that uniquely identifies a single software vendor responsible for the SAP ME extensions. This value is used by the build system to insure the uniqueness of extension artifacts so that extensions from multiple software vendors can coexist in their own name space. For example, in the case of Java classes the vendor ID must be included as part of the class package name. The vendor ID is also added to version information created by the build. The same vendor ID must be used by the production build instance and all development build instances for a given software vendor. Different software vendors must use unique vendor ID values.
`web.context.path`	The web context path used to access the base SAP ME web application. This is used by the build to set the context-root value in `application.xml` after an update has been imported. This value cannot be the same as the value set for the xml`.context.path` property. This property is used only in the development environment.
`xml.context.path`	The web context path used to access the production XML interface web application. This is used by the build to set the context-root value in `application.xml` after an update has been imported. This value cannot be the same as the value set for the `web.context.path` property. This property is used only in the development environment.
`db.drivername`	The NetWeaver configuration name of the SAP ME database driver. This property is used only in the development environment.
`db.vendor`	The target database server platform. This value is used when packaging custom database scripts. Valid values are: • oracle • sqlserver
`db.wip.driverclassname`	The JDBC driver class name used to connect to the WIP database. This property is used only in the development environment.

Name	Description
`db.wip.driver.url`	The partial JDBC URL for the WIP database driver. The value should be complete up to, but not including the name of the database host. SQL Server: jdbc:sqlserver:// Oracle: jdbc:oracle:thin:@
`db.wip.host`	The WIP database host name.
`db.wip.port`	The WIP database port number. For SQL Server if this property is not defined then the default port 1433 will be used.
`db.wip.sid`	The WIP database system ID. For SQL Server, this is the database name. If no value is specified SQL Server will connect to the default database.
`db.wip.instance`	The WIP database instance name for SQL Server. If no value is specified then SQL Server will connect to the default instance.
`db.wip.user`	The WIP user name.
`db.wip.password`	The WIP user password.
`db.ods.driverclassname`	The JDBC driver class name used to connect to the WIP database. This property is used only in the development environment.
`db.ods.driver.url`	The partial JDBC URL for the ODS database driver. The value should be complete up to, but not including the name of the database host. SQL Server: jdbc:sqlserver:// Oracle: jdbc:oracle:thin:@
`db.ods.host`	The ODS database host name.
`db.ods.port`	The ODS database port number. For SQL Server if this property is not defined then the default port 1433 will be used.
`db.ods.sid`	The ODS database system ID. For SQL Server, this is the database name. For SQL Server, this is the database name. If no value is specified SQL Server will connect to the default database.
`db.ods.instance`	The ODS database instance name for SQL Server. If no value is specified then SQL Server will connect to the default instance.
`db.ods.user`	The ODS user name.
`db.ods.password`	The ODS user password.

Name	Description
`dpmo.nc.codes`	Flag indicating whether to add DPMO NC codes to NC initial data load files when the base application is imported. Valid values are: • true • false
`default.locale`	The default locale to be used when creating default language properties files. Valid values depend on the locales supported by the base SAP Manufacturing Execution release and any additional language support added by the customer.
`jdk.home`	The home directory of the Java SDK installation to be used to run the build system. This property is used only in the development environment. **Note:** Paths must be separated by a forward slash "/".
`security.storepass`	The password used to access the SDK keystore. This property is valid only in the production build environment. A default value is used in the development environment.
`security.storetype`	The SDK keystore type. Valid values are jks, jceks, and pkcs12. This property is defaulted to jks and should not be changed unless the keystore type is changed.
`security.keypass`	The SDK keystore key password. This property is valid only in the production build environment. A default value is used in the development environment.
`scs.build.ID`	An identifier created by the source control system that can be used to synchronize sources to a given point in time. For example, this can be a time stamp, change list number, or file label value. The value of this property is displayed in the version information for each software vendor. This value can be reported back to the software vendor in the event that a support issue is found. The identifier can be used by the source control system to synchronize sources to the same version deployed on the production system. This property should be set before creating a final production build if you want to correlate
`ext.info`	This property represents a place holder for adding any other information that you would like reported along with other version information.
`compile.debug`	Enable debug symbols when compiling Java source. Valid values are "true" or "false".