JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche Fakultät

# azDeploy: Remote Deployment of .NET Applications and Database Schemas

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Software Engineering

Eingereicht von:
Rainer Pichler BSc

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
o. Univ.-Prof. Dr. Dr. h.c. Hanspeter Mössenböck

Mitwirkung:
Dr. Reinhard Wolfinger

Linz, Dezember 2011

Maintaining software on appliances installed at remote sites is a time-consuming task. This thesis presents azDeploy, a solution for remotely deploying .NET application and database schema upgrades. Compared to manual operations, it aims to reduce the time needed for these tasks and the error probability through automation. Furthermore, it interacts with the maintained applications, allowing seamless and non-disturbing upgrades. Besides elaborating its concept, a prototype was implemented.

Die Wartung der Software von Appliance-Lösungen an entfernten Standorten ist zeitaufwändig. Diese Masterarbeit stellt eine Lösung zur Fernaktualisierung von .NET-Anwendungen und Datenbankschemata namens azDeploy vor. Sie soll den Zeitaufwand sowie die Fehlerwahrscheinlichkeit gegenüber dem händischen Einspielen dieser Aktualisierungen mittels Automatisierung verringern. Um den Betrieb durch die Wartungsaktivitäten möglichst wenig zu beeinträchtigen, interagiert sie mit den betroffenen Anwendungen. Neben der Ausarbeitung des Konzepts wurde ein Prototyp von azDeploy entwickelt.

# Table of Contents

# Aufgabenstellung

### Installation und Update von Softwarekomponenten und Datenbankschemata

Für die Installation und das Update von komponentenbasierten .NET-Programmen mit SQL-Datenbank soll ein Werkzeug entwickelt werden. Das Werkzeug soll Änderungen am Datenbankschema und geänderte .NET-Komponenten beim Kunden über das Internet installieren. Die Installation wird von Mitarbeitern des Herstellers ausgelöst, d.h. ein Mitarbeiter prüft mit dem Werkzeug welche Updates beim Kunden installierbar sind und startet manuell das Update.

Das Werkzeug soll aus drei Teilen bestehen: Ein Teil empfängt und installiert das Update am Zielcomputer. Vom Hersteller aus steuert ein zweiter Teil die Updates. Da es beim Kunden mehrere Zielcomputer geben kann und diese nicht direkt aus dem Internet erreichbar sind, empfängt ein dritter Teil am Server des Kunden die Aktualisierungen und leitet diese über das Intranet an die Arbeitsplätze weiter.

Das Werkzeug soll mit C# und dem .NET-Framework entwickelt werden. Dabei muss Windows Presentation Foundation, Windows Communication Foundation und Microsoft SQL-Server 2008 verwendet werden.

### Task: Installation and Update of Software Components and Database Schemas (Translation by Rainer Pichler)

The task is to develop a tool allowing the installation and update of component-based .NET applications and SQL databases. The tool should install database schema changes and changed .NET components at customer sites via the Internet. The installation is triggered by the vendor staff. Thus, an employee uses the tool to determine the applicable updates and starts the update manually.

The tool should consist of three components: The first component receives and installs the update on the target computer. At the vendor site, the second component controls the update process. Since there may exist multiple target computers which do not have Internet access, the third component runs on the customer's server, receives the updates and forwards them to the target computer via the local network.

The tool should be developed with C# and the .NET Framework. Also, Windows Presentation Foundation, Windows Communication Foundation, and Microsoft SQL-Server 2008 must be used.

Nähere Auskünfte/Further Information: Dr. Reinhard Wolfinger
Beginn/Issue Date: November 2009

# 1 Introduction

Software vendors often have to maintain appliances installed at remote customer sites. In this work, *appliance* refers to an embedded system that many different users use for one specific purpose. Such an appliance runs a single .NET application and hosts a database.

Maintaining appliances includes deploying feature enhancements, bug fixes, and security fixes. This work subsumes them under the term *upgrades.* Such upgrades can affect the application running on the appliance or the database it uses. Deploying upgrades manually has several drawbacks:

- If the vendor sends his operators to the remote site, this can cause traveling expenses. To avoid this, the vendor could use remote desktop software to maintain the appliances remotely. However, this is still manual work and the effort multiplies with the number of appliances at the remote site, because the whole procedure must be repeated for each appliance.
- Users cannot use the appliance during the maintenance procedure. Thus, the longer the procedure takes, the longer the appliance cannot be used.
- Tasks that are carried out manually are error-prone. Especially repeating the same upgrade procedure on several appliances under time pressure can lead to reduced concentration. This entails a higher risk of making mistakes and rendering appliances unusable.
- It is a cumbersome task for the vendor to keep track of the rolled out application and database schema versions. Manually recording changes is error prone and time-costly.

Therefore, this thesis elaborates azDeploy, a deployment solution which addresses these issues. azDeploy provides:

**Reduced Upgrade Time:** azDeploy reduces the time an upgrade takes. Firstly, automation is faster than human interaction. Secondly, azDeploy can carry out the same task on many appliances simultaneously.

**Reduced Risk:** Through automation, azDeploy decreases error probability. If an upgrade fails, it restores the previous state of the appliance.

**Reliable Information:** azDeploy keeps track of the installed upgrades.

| Feature | Remote Desktop | Customer-Managed | AZDEPLOY |
|---|---|---|---|
| Operator | Vendor | Customer | Vendor |
| Scope | Universal | Applications | Applications and Databases |
| Scalability | low | high | high |
| Vendor Adoption Effort | low | high | moderate |
| Flexibility | high | low | moderate |
| Error Rate | high | low | low |
| Customer Involvement | low | high | low |

Table 1.1: Distinguishing Features of AZDEPLOY

Table 1.1 compares AZDEPLOY with the Remote Desktop approach and with using existing deployment systems managed by the customers.

Using remote desktop software gives the highest flexibility, but does not scale well and is error prone. In contrast, the two other approaches take less time but also sacrifice flexibility. The main disadvantage of a customer-managed deployment system is that its availability and implementation vary between customers. Thus, this approach incurs a high customer involvement and a high adoption effort as the vendor needs to support different deployment systems.

When using AZDEPLOY, the deployment system is under the control of the vendor, thus minimizing the customer involvement. Also, compared to the customer-managed deployment system, the vendor's adoption effort is lower as the vendor needs to support only AZDEPLOY. Finally, AZDEPLOY is flexible, since vendor applications can hook into its operations.

**Platform and Technology** The prototype presented in this thesis was implemented for .NET 3.5 applications and Microsoft SQL Server 2008 running under Windows XP. However, as the concept is universally applicable, AZDEPLOY could be ported to other platforms.

## 1.1 Application Scenario

The requirements for AZDEPLOY are derived from the following scenario, which resembles features of real world applications: A software vendor has sold a time recording solution called JORNADA to a manufacturing company. JORNADA consists of a server application and of a client application. The server machine hosts the server application and the *main database*. The server application only accesses the *main database*. The client application accesses the *main database* on the server machine and the appliance's *local database*. The appliances cannot directly access the Internet.

Every employee at the manufacturing company uses JORNADA, but not all of them are computer experts. Therefore, JORNADA is designed as an appliance: The client application runs in full screen mode all the time on a dedicated embedded system. Several appliances are installed in the company's buildings and are used by numerous different employees throughout the day. Since the employees have flexible working hours, they expect JORNADA to be available around the clock. Thus, there are no exclusive time intervals for system maintenance. Instead, the vendor must try to keep the downtime of the system as short as possible.

## 1.2 Requirements

To meet the requirements of the application scenario (see Section 1.1), the software vendor wants a solution that

- upgrades all JORNADA applications and databases on an arbitrary number of appliances in parallel. The software vendor ships the JORNADA applications as Windows Installer packages and the database schema upgrades as incremental SQL scripts. Also, the vendor wants to back-up and restore multiple databases in parallel. The solution ensures that the applications and databases stay in a valid configuration at all times. As a consequence, whenever an upgrade fails, the solution rolls back all changes.
- provides information about the installed applications and databases as well as their upgrade history.
- interacts with the JORNADA applications: For example, it ensures that the JORNADA client application is shut down properly before an application upgrade. To avoid interrupting operations, the solution obtains approval from the JORNADA client application that it is in an idle state before the upgrade is started.
- displays debug information generated by the JORNADA applications for troubleshooting. As it is stored in the databases, the software vendor wants to be able to display data for arbitrary SQL queries.
- the software vendor can control remotely via a secure web interface.

## 1.3 Thesis Structure

Chapter 2 derives the specification for AZDEPLOY from the requirements in Section 1.2. Chapter 3 introduces the system architecture. Chapter 4 describes how the individual parts of AZDEPLOY communicate with each other. Chapter 5 describes the solution's components and the features of the individual applications. Chapter 6 will highlight several implementation details. Chapter 7 and 8 explain how to set-up and work with AZDEPLOY. Chapter 9 discusses the result of this work and suggests further improvements.

# 2 Specification

This chapter describes the specification for azDeploy, that is the actors and the use cases, and the database and application management capabilities.

## 2.1 Actors and Use Cases

azDeploy distinguishes between five actor types:

**Operators:** They work for the software vendor and use azDeploy actively.

**Developers:** They work for the software vendor and prepare the packages and scripts to deploy via azDeploy.

**Vendor Applications:** These .NET applications are administered by azDeploy and can influence azDeploy's operations.

**Customer Administrators:** They work for the customer and manage its IT infrastructure. Thus, they demand that azDeploy is secure, that it does not require additional maintenance and uses a limited number of Internet protocol ports to allow efficient firewall configuration.

**Users:** They work for the customer and use the vendor applications administered by azDeploy. Users cannot see azDeploy, as it runs as a service in the background.

Since customer administrators and users do not directly interact with azDeploy, they are passive actors.

**Use Cases** azDeploy has several use cases:

- The operator is curious about which version of an application is installed on an appliance. S/he uses azDeploy to retrieve this information.
- The operator wants to upgrade an application at a customer site. Therefore the operator upgrades the application and the database schema via azDeploy. Users are affected by this upgrade since azDeploy restarts the upgraded application.
- The operator retrieves the log of an application from the appliance, because a user reported that the application does not work as expected.

- The operator wants to back up all databases at a customer site. S/he uses azDeploy to back up all databases in one step.
- Users want to finish their work when using Jornada. Therefore, azDeploy will not install upgrades while an appliance is in use.

## 2.2 Database Management

azDeploy introduces version control for databases schemas. A *database schema* describes the structure of a database's objects. azDeploy allows the operators to put one schema per database under version control. Developers provide upgrades for a database schema as annotated SQL scripts.

### Features

The system supports the following functions for managing databases:

**Manage Databases:** The operator can create, delete, back-up and restore databases. Also, s/he can delete database backups.

**Initialize Database:** The operator can put a database under version control by defining two properties:

**Database Schema:** This property determines which upgrade scripts can be run on the database.

**Initial Version:** This property defines the current version of the database schema. Empty databases should be set to version 0.0.0.0. Setting a deviating version number is useful when putting existing databases under version control.

**Upgrade Database Schema:** The operator can only apply this operation to databases that are under version control. The operator upgrades a database schema by choosing an upgrade path. Such upgrade path consists of one or more SQL upgrade scripts, which are executed one by one. For each SQL upgrade script, the target schema version of the preceding script matches the required schema version of the current script.
azDeploy offers the operator only the shortest upgrade path to each schema version: For example, if a database with schema version 1.0 should be upgraded to version 1.2 and there are three upgrade scripts available - version 1.0 to 1.1, version 1.1 to 1.2 and version 1.0 to 1.2 - azDeploy will only offer the upgrade path consisting of the cumulative upgrade script from version 1.0 to 1.2.
Since the SQL upgrade scripts are run within a transaction, the upgrade only turns effective if all upgrade scripts execute without errors.

**View Custom Data:** The operator may run arbitrary SQL queries on a database. S/he can store a SQL query as a *view* that is available for all databases that have specific schema. The operator can define a schema version range for which the view is available.

All but the last operation can also be applied to a set of databases at the customer site. Because azDeploy manages the database backups too, the databases must be installed locally on the appliance. For upgrades, the schema and version of the selected databases must match.

## Version Format

azDeploy uses the .NET version format (see [1]). Database schemas should use its four numbers as follows:

**Major and Minor:** These numbers designate the major and minor version number of the database schema. A schema upgrade where one or both of these numbers increase may not be compatible with applications requiring the previous schema version. Such upgrades include introducing new mandatory columns or renaming database objects accessed by the applications.

**Build:** A change of this number indicates that new optional database objects are introduced. Such upgrade could be the insertion of a new nullable column or a new table. Therefore, such an upgrade does not break application compatibility.

**Revision:** A change of this number indicates that the structure of database objects accessed by applications have not changed. Such upgrade could be the insertion of new rows in list of values tables or changing database object settings to improve performance. Therefore, such an upgrade does not break application compatibility.

azDeploy does not require to obey this scheme. However, vendor applications interacting with azDeploy can take advantage of it: For example, instead of being restarted, Jornada Client keeps running and merely needs to refresh cached data if only the *build* or *revision* numbers of the database schema changed after a database upgrade.

## SQL Upgrade Scripts

Developers publish database schema upgrades as annotated SQL scripts (Listing 2.1). An upgrade script starts with metadata that azDeploy interprets, afterwards the SQL statements modifying the database schema follow. Since the metadata is defined in the commentary section, developers can run upgrade scripts without azDeploy throughout development.

Developers must set all attributes and tags shown in Listing 2.1. azDeploy allows to run the script only on databases that have both a schema name matching the *schema* attribute value and a schema version matching the *requiresVersion* attribute value. Once the script has been run without errors, azDeploy sets the database schema version to the value of the attribute *providesVersion*.

```
/*<databaseScript
    author="Rainer Pichler"
    date="2011-07-08"
    schema="JornadaClient"
    requiresVersion="1.0.1.0"
    providesVersion="1.2.0.0">
    <description>
        introduce security log
    </description>
</databaseScript>*/

CREATE TABLE SecurityLog (
[...]
```

Listing 2.1: A Database Schema Upgrade Script

## 2.3 Application Management

azDeploy can deploy software distributed as Microsoft Windows Installer (MSI) packages that comply with the restrictions mentioned on page 58. Developers use Microsoft Visual Studio 2008 and the *package tool* application to create such packages.

An operator can use azDeploy to install or upgrade an application on multiple appliances in parallel. Also, s/he can uninstall arbitrary applications installed by Windows Installer on multiple appliances in parallel. Because upgraded applications may render configuration files unusable for the previously installed application version, azDeploy does not allow to downgrade an application directly. However, a downgrade is possible through uninstalling the newer application version first and then installing the previous application version.

Finally, vendor applications can track azDeploy's operations and prevent them, for instance while they are in use (see Section 4.6 on page 37).

# 3 Architecture

This chapter gives an overview of the components of azDeploy and addresses the key design considerations.

## 3.1 System Overview

This section introduces the six applications of azDeploy.

**Server Application** There is a single installation of the *server application* which resides on a machine called server at the vendor site. The server has direct Internet access and listens for incoming connections from the customer sites.

**Administration Center** The *administration center* browser application enables operators to control azDeploy. It is hosted on the server via a web server and communicates with the server application. Because multiple operators can use azDeploy in parallel, several instances of the administration center may run at the same time.

**Gateway Agent** The *gateway agent* resides on a machine called *gateway* at the customer site. There is one gateway agent installation per customer and the gateway must have Internet connectivity. The gateway agent communicates with the server application at the vendor site via the Internet. It distributes control commands within the customer's network.

**Client Agent** An instance of the *client agent* runs on each appliance and carries out the deployment tasks. Thus, it runs under a privileged user session. Because the client agent communicates with the server application indirectly via the gateway agent, it does not require Internet access.

**Package Tool** The *package tool* resides on the developer workstations and enables developers to prepare software installation packages for deployment.

**Application Starter** The *application starter* runs on each appliance and allows to start installed or restart upgraded applications. It interacts with azDeploy like a vendor application.

Table 3.1 lists the different machine types and their properties.

| Machine | Instances | Internet Access | DBMS | Installed Applications |
|---|---|---|---|---|
| Server | 1 | Direct | Yes | Server Application<br>Administration Center |
| Operator Workstation | many | Needs to access Server | No | Web Browser |
| Developer Workstation | many | No | No | Package Tool |
| Gateway | 1 per site | Yes | No | Gateway Agent |
| Appliance | many | No | Yes | Client Agent<br>Application Starter |

Table 3.1: Machine Types

**Network Topology** Figure 3.1 depicts the network topology of the Jornada scenario. Its left side shows the vendor site network, consisting of the server and two workstations. The server hosts the azDeploy server database and the azDeploy server application, and is connected to the Internet. The workstations run the administration center within a web browser. The right side of Figure 3.1 shows a customer site network, consisting of a gateway, a Jornada server machine and three appliances. The gateway is connected to the Internet and runs the azDeploy gateway agent. The Jornada server runs the Jornada server application whereas the appliances run the Jornada client application. Additionally, each of them hosts a database and runs the azDeploy client agent. azDeploy considers the Jornada server an appliance that differs from the other appliances only in the installed applications and the database schema.

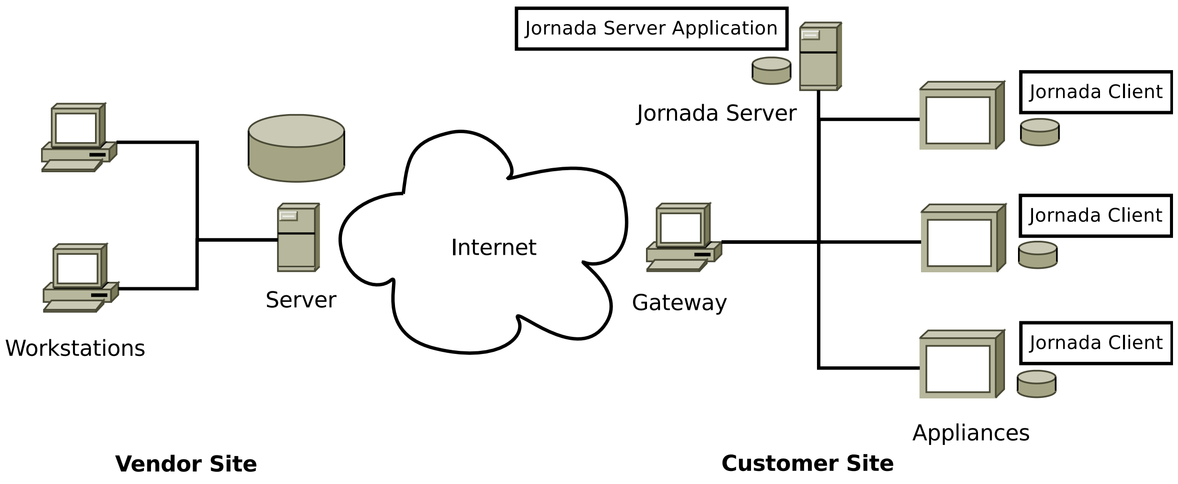


Figure 3.1: Network Topology

**Example** Figure 3.2 visualizes how azDeploy works. It is based on the scenario shown in Figure 3.1, but does not show the second workstation and the network links. Instead, it depicts an operator and the basic steps that take place when upgrading two databases (upgrading vendor applications works analogous). While any communication between the workstation and the customer site involves the server and the gateway, Figure 3.2 does not show these details for the sake of clarity. Upgrading the databases on two appliances consists of the following logical steps:

1. The operator logs onto the administration center.
2. The administration center queries the server application for the customer sites.
3. The operator selects the customer site s/he wants to administer.
4. The administration center queries the gateway agent of the administered site for the available databases. In turn, the gateway agent queries the client agents of all appliances and the Jornada server for the hosted databases. Then, the administration center displays the available databases.
5. The administration center queries the server application for applicable upgrades.
6. The operator selects the two databases that should be upgraded and the upgrade.
7. The administration center initiates the upgrade operations. The upgrade scripts, that are stored on the server, are transferred to the affected appliances.
8. Both affected appliances request permission from the vendor applications to upgrade their databases (see Section 4.6 on page 37).
9. Both affected appliances run the upgrade scripts on their databases independently and report the result to the administration center and to the vendor applications. Thus, failed upgrades do not prevent the upgrade of the other databases.
10. The administration center displays the results of the upgrade operations.

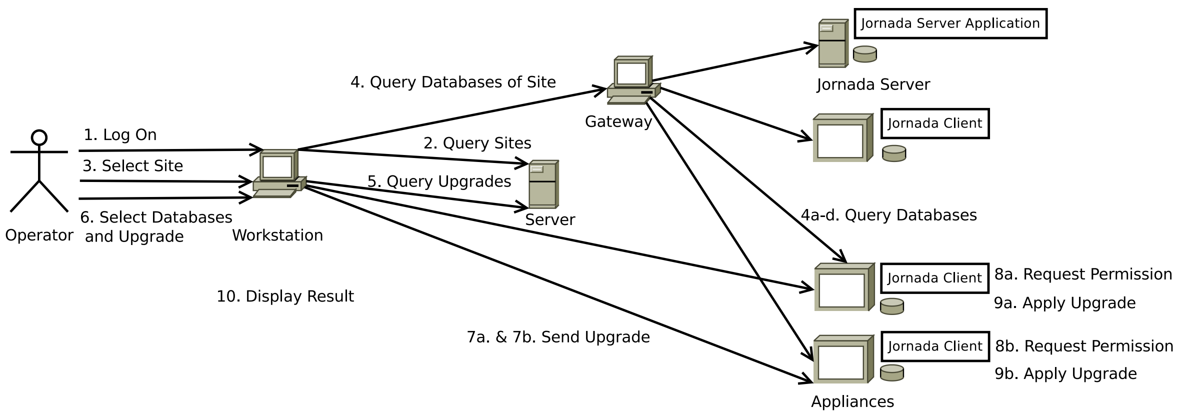


Figure 3.2: Upgrading Two Databases

## 3.2 Design Considerations

**Deployment and Maintenance** One goal of azDeploy is to make deployment more efficient. Therefore, the benefits must exceed the additional deployment effort of azDeploy itself. Deployment effort can be divided into two phases:

1. Initial deployment effort is needed once for each customer site: It consists of setting up the gateway agent on the gateway machine and setting up the client agent on each appliance. Because this initial task must be carried out manually, the effort scales with the number of appliances.
   To ease initial deployment, azDeploy's applications are installable through Windows Installer and have no dependencies other than the .NET Framework. Also, the configuration effort is held minimal: Through the use of port sharing, only a single port must be forwarded to the gateway machine if using network address translation. The operator must enter the site's credentials in the gateway agent configuration file and enter the gateway's hostname in the client agents' configuration files. To give the client agent access to the administered databases, a plug-in that extracts the database connection string from the vendor application's configuration file is added.

2. Maintenance deployment effort: To adapt to new scenarios and to fix bugs, the system is able to update itself without user interaction. Because the core applications of azDeploy (server application, gateway agent and client agent) are tightly coupled, each application must patch itself to the newest release before connecting. The browser application can be updated centrally by publishing a new version on the web server. The azDeploy prototype cannot update the application starter.

**Portability** azDeploy targets a Microsoft Windows environment. Even the administration center requires the .NET Framework. However, since the administration center uses an interoperable protocol to communicate with the server application, developers could implement a version that runs on other platforms.

**Security** azDeploy spreads across three network zones: Firstly, the vendor site includes the server and the operators' workstations. Secondly, the customer site includes the gateway machine and the appliances. Thirdly, as the Internet is untrusted, azDeploy encrypts all communication between the vendor site and the customer site. By default, encryption is not enabled within the customer site. At the vendor site, all internet-accessible WCF services use encryption. Thus, operators can use the administration center securely at remote locations such as customer sites. Section 6.4 on page 70 will discuss the security aspects in more detail.

# 4 Communication

azDeploy uses Windows Communication Foundation (abbreviated WCF) for networking. Therefore, the first section introduces the basic concepts of WCF. The remaining sections describe the WCF services azDeploy consists of and their interaction with each other.

azDeploy employs four classes of WCF services (Figure 4.1):

**Agent Upgrade Services:** azDeploy uses these WCF services to remotely upgrade its own components, excluding the application starter. It should not be confused with the ability to upgrade vendor applications, as it only affects the gateway agent and the client agent. These WCF services involve the server application, the gateway agent and the client agent.

**Operation Control Services:** azDeploy uses these WCF services to control the individual components and processes. They involve the administration center and the server application at the vendor site, as well as the gateway agent and the client agent at the customer sites.

**File Transfer Services:** The operation control services rely on these services to transfer files, such as application packages or database upgrade scripts, to the appliances. These WCF services involve the server application, the gateway agent and the client agent.

**Notification Services:** These WCF services allow vendor applications to track and deny configuration changes.

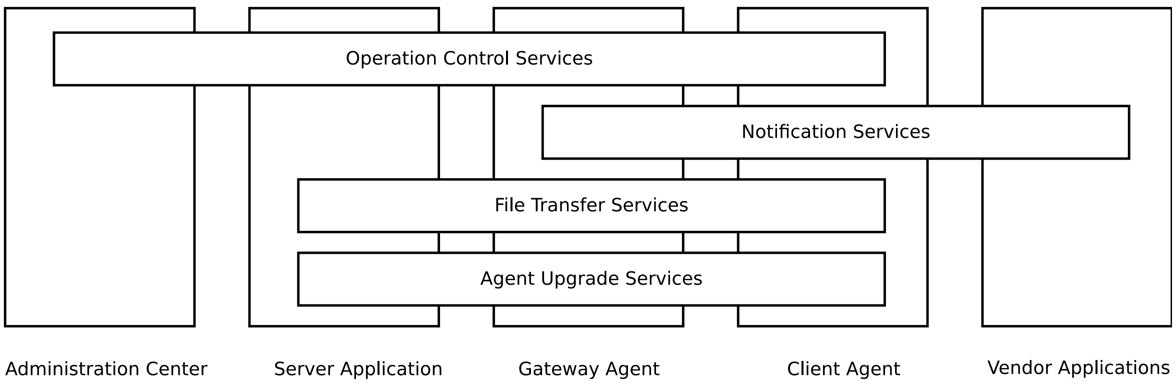


Figure 4.1: Distribution of the Service Classes

## 4.1 Windows Communication Foundation

Windows Communication Foundation (WCF) is a framework for creating network services and clients in .NET. To avoid confusion with daemon-like Windows services, these services will be termed WCF services throughout this work.

### Service Definition

**Service Contracts and Endpoints** *Service contracts* specify the methods a certain type of WCF service exposes. A service contract is defined through a .NET interface with the exposed methods. The actual WCF service class implements the methods of the service contract interface. Because a WCF service class can implement multiple service contracts, there is the concept of endpoints. Each endpoint satisfies a certain service contract and exposes its methods. Clients can address an endpoint by a URL [2,3].

**Bindings** A *binding* defines the underlying communication protocol and its properties like for instance security options for an endpoint. WCF assigns exactly one binding to an endpoint [2]. azDeploy uses four binding types:

The proprietary *netTcpBinding* has limited interoperability, but offers many features like for example client callbacks. It is therefore used for the communication between the server application, the gateway agent and the client agent. In contrast, the communication between the server application and the administration center is based on the *wsHttpBinding*, which has less features, but provides cross-platform interoperability. To facilitate this interoperability, *mexHttpBinding* publishes standardized metadata about the service. *netNamedPipeBinding* is used to interconnect the WCF services within the server application, as it is only capable of and optimized for local machine communication [4,5].

**Binding Configuration** Bindings can be configured both via the application configuration file and in code [6]. azDeploy uses an application configuration file for both the server application and the administration center. Thus, the operators can adjust the binding configuration without having to change the application. In contrast, the applications installed at the customer site configure the bindings in code. Therefore, customer administrators cannot inspect the binding configuration. Instead, the variable parameters like the log-on credentials are stored as settings in the respective application configuration file. Section 6.4 on page 70 discusses the fundamental binding configurations of the server application.

**Example** The server application's core WCF service illustrates these concepts: The WCF service class *MainService* implements the service contracts *IAdminService* and *IControlService*. For *IAdminService*, it exposes an endpoint via a *wsHttpBinding* under the URL *https://localhost:8002/AdminService*, which is consumed by the administration center. For *IControlService*, it exposes another endpoint via a *netTcpBinding*

```
[ServiceContract]
public interface IAdminService {
    [OperationContract]
    string[] GetSites();
    [OperationContract]
    Host[] GetHosts(Site site);
    ...
}
```

Listing 4.1: Service Contract for the *IAdminService* Endpoint (simplified and shortened)

```
[DataContract]
public class Host {
    [DataMember]
    public string SiteName { get; set; }
    [DataMember]
    public string HostName { get; set; }
}
```

Listing 4.2: *Host* Data Contract

under the URL *net.tcp://localhost:8001/ControlService*, which is consumed by the gateway agent. Additionally, the service offers a metadata exchange endpoint via a *mexHttpBinding* for the *IAdminService* endpoint.

Listing 4.1 shows two simplified method definitions from the *IAdminService* service contract: *GetSites* returns all customer sites connected to the server. Likewise, *GetHosts* returns the on-line appliances of the specified site. Each WCF service method must be annotated with an *OperationContract* attribute [3].

**Data Contracts** Analogous to service contracts, there are *data contracts*: These enable the developer to use complex data structures as both parameters and return values of service methods. Data contracts are defined through a class annotated with the *DataContractAttribute*. Furthermore, each property of the class must be annotated with a *DataMember* attribute [7].

azDeploy uses the data contract *Host* (Listing 4.2) to address a machine at a customer site. It contains the name of the customer site and the host name of the appliance.

## Advanced Concepts

To explain the more advanced WCF techniques that azDeploy uses, the rest of the section refers to the service contract *IControlService* (Listing 4.3).

Consuming the endpoint *IControlService* allows the gateway agent to log on and off the server as well as to inform the server about events. In contrast to the previously discussed service contract example, it uses the concepts of fault contracts, callback contracts, sessions, instancing, concurrency and one-way operations.

```
[ServiceContract(SessionMode = SessionMode.Required, CallbackContract = typeof(IGatewayCallback))]
public interface IControlService {
    [OperationContract(IsOneWay = false, IsInitiating = true, IsTerminating = false)]
    [FaultContract(typeof(IdentifierInUseFault))]
    void LogOn();
    [OperationContract(IsOneWay = false, IsInitiating = false, IsTerminating = true)]
    void LogOff();
    [OperationContract(IsOneWay = true)]
    void NotifyEvent(RemoteEvent ev);
}
```

Listing 4.3: *IControlService* Service Contract

**Fault Contracts** By default, if a service method like *LogOn* throws an exception, no exception details are disclosed to the consumer of the service for security reasons. To transport information about certain errors, developers use *fault contracts*. A fault contract is a data contract carrying error information. The *FaultContract* attribute of the method *LogOn* indicates that the method may return a fault of the type *IdentifierInUseFault*. This happens in case a gateway agent tries to log on with a site name already used by another gateway agent. To propagate this fault, *LogOn* throws an exception of the generic type *FaultException*, passing along the fault contract object (Listing 4.4, line 10) [8, 9].

Note that this example merely uses the fault contract object to indicate the type of error as it does not contain further error information. The gateway agent handles this error by surrounding the service call with a try-catch block and catching an exception of the type *FaultException<IdentifierInUseFault>* [9].

**Callbacks** To issue commands to customer sites, *MainService* invokes operations on gateway agents through a callback contract. A callback contract is an interface similar to a service contract which the calling client must implement. The *CallbackContract* property of the *ServiceContract* attribute indicates the interface to implement [10].

Listing 4.5 shows the callback contract for the gateway agent: The method *GetSiteHosts* retrieves all appliances connected to the gateway agent. The method *RunTask* is used by the task system (see Section 4.5 on page 28), which allows the service and callback contracts to stay lean.

*MainService* can call these methods on the respective gateway agent's communication channel object, which implements *IGatewayCallback* [10].

But how can *MainService* obtain a reference to such an object for a specific gateway agent? This question will be answered through a look at the session concept in the next paragraph.

**Sessions** Because several customer sites may log onto the server, *MainService* maintains a session for each connected gateway agent to distinguish between them. Therefore, the property *SessionMode* is set to *Required* in the service contract attribute

```
 1 private ClientManager<string, IGatewayCallback, SiteSessionData> clientManager;
 2
 3 public void LogOn() {
 4     var callback = OperationContext.Current.GetCallbackChannel<IGatewayCallback>();
 5     ((IDuplexContextChannel)callback).Closed += GatewayCallback_Closed;
 6     var siteName = CallingSiteName;
 7    var success = clientManager.LogOnClient(siteName, callback, OperationContext.
        Current.SessionId, new SiteSessionData());
 8
 9     if (!success)
10         throw new FaultException<IdentifierInUseFault>(new IdentifierInUseFault(),
             "A site with the given name is already logged on.");
11
12     AdminSessionManager.DispatchPublicEvent(new SiteConnectivityChangedEvent(
         siteName, true));
13     Trace.WriteLine(string.Format("Site {0} logged on.",siteName));
14 }
15
16 private string CallingSiteName {
17     get {
18         return OperationContext.Current.ServiceSecurityContext.WindowsIdentity.Name
             .Split('\\')[1]; // omit Windows domain
19     }
20 }
```

Listing 4.4: The *LogOn* Method of *MainService*

```
public interface IGatewayCallback {
    [OperationContract]
    Host[] GetSiteHosts();
    [OperationContract(IsOneWay = true)]
    void RunTask(Task task);
}
```

Listing 4.5: Callback Contract for the Gateway Agent

(Listing 4.3). Also, the properties *IsInitiating* and *IsTerminating* are specified in the *OperationContract* attributes for the methods *LogOn* and *LogOff*. This tells the WCF runtime to establish a session when *LogOn* is called and terminate it once *LogOff* is called. Because sessions are mandatory, no operations can be called by the gateway agent without an established session [11].

The service implementation obtains the *IGatewayCallback* channel object for the calling gateway agent through the WCF *OperationContext* (Listing 4.4, line 4) [10]. It extracts the site identifier from the gateway's credentials in the *CallingSiteName* property. Since WCF does not provide a separate storage for session specific data [12], the service uses the generic class *ClientManager* to store the gateway agent's callback object and session data (Listing 4.4, line 7).

**Concurrency and Instancing** As multiple gateway agents and multiple instances of the administration center connect to the server application, it has to deal with parallelism. To allow scaling, WCF provides service object instancing and multithreading: WCF offers to instantiate a service object per service, per session or per operation. Concerning multithreading, a service implementation can basically allow or disallow

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Reentrant,InstanceContextMode=
    InstanceContextMode.Single)]
public class MainService : IAdminService, IControlService {
    public void LogOn() { ... }
    ...
}
```

Listing 4.6: Configuration of Concurrency and Instancing of *MainService*

concurrent calls [12].

As the concurrency and instancing behavior is specified in the actual service implementation of *MainService* (Listing 4.6), it applies to both endpoints of the service.

*MainService* uses one instance and opts to avoid concurrent threads. Therefore the property *InstanceContextMode* of the attribute *ServiceBehavior* is set to *Single*. However, for the property *ConcurrencyMode*, it uses the value *Reentrant* instead of *Single*: This loosens WCF's service object locking behavior [12].

In the mode *Single*, the WCF runtime only allows to process one message at a time. Suppose that the administration center wants to retrieve the list of online appliances for a site. Therefore, it calls the method *GetHosts* which is exposed through the service contract *IAdminService*. Its implementation by *MainService* is shown in Listing 4.7. To call *GetHosts*, the administration center sends a request message to *MainService*. In turn, the WCF runtime receives the message and locks *MainService* so that it cannot process further requests. Then it interprets the message and invokes *GetHosts* on the service thread. In line 2 of *GetHosts*, *MainService* calls the *GetSiteHosts* method on the callback channel of the corresponding gateway agent. To do so, *MainService* sends a request message to the gateway agent. Then *MainService* blocks until it receives a reply message. At the gateway agent, the WCF runtime interprets the message, calls the *GetSiteHosts* method and sends a reply message containing the connected appliances to *MainService*. However, since *MainService* waits for the reply message in line 2, processing of the initial message to call the *GetHosts* method has not yet finished. Therefore, the service is still locked and cannot process the reply message [10, 12].

All in all, a deadlock situation occurs as *MainService* waits for the reply message from the gateway agent, while the reply message waits for the service lock to disappear.

In contrast, in the concurrency mode *Reentrant*, the WCF runtime unlocks the service each time it makes an outgoing call. Therefore, in the previous example, the service is unlocked when *MainService* waits for the reply message from the gateway agent, implying that a deadlock situation cannot occur. Because the service accepts further calls while waiting for the reply message from the callback invocation, any client can invoke service methods meanwhile. Therefore, the service has to ensure that it is in a consistent state before invoking a callback [10, 12].

The third concurreny mode is *Multiple*, where WCF does no locking and the service has to ensure thread safety itself. It is used in the agent upgrade service (see Section 4.3 on page 21).

```
public Host[] GetHosts(string siteName) {
    return clientManager.GetCallback(siteName).GetSiteHosts();
}
```

Listing 4.7: The *GetHosts* Method of *MainService*

**One-way operations** One-way operations do not return data nor report the operation execution success. The service does so by not sending a reply message to the client. Depending on the service configuration, the client often does not block when calling a one-way operation, because the service can receive and buffer several messages before processing them. One-way operations must be marked by setting the *IsOneWay* property of the *OperationContract* attribute to *true* [10]. The method *NotifyEvent* shown in Listing 4.3 is marked as one-way, as the notifying gateway agents do neither care about when the service processes the remote event nor expect a result.

## 4.2 Identifiers

This section explains the most important identifiers azDeploy uses.

**Sites** azDeploy uses a *site name* string to refer to a specific customer site. The gateway agents authenticate themselves to the server with the site name and a password.

**Hosts** To address a customer site's gateway and appliances, azDeploy uses a *host identifier* (data contract in Listing 4.2 on page 15). It has the properties *SiteName* and *HostName*. For an appliance, *SiteName* is the site identifier whereas *HostName* is the appliance's machine name. To refer to the gateway, *HostName* is set to the value *"@"*. Additionally, the task system uses *Host* to address certain WCF services hosted by the server application. In this case, *SiteName* has the value *"@"* and *HostName* contains the addressed service's name. The string representation of host identifiers is *sitename/hostname*. Table 4.1 shows examples for the three types of host identifiers.

| SiteName | HostName | Refers To | Description |
|---|---|---|---|
| demosite | EXAMPLEAPPLIANCE | Appliance | Addresses the appliance *EXAMPLEAPPLIANCE* of the site *demosite*. |
| demosite | @ | Gateway | Addresses the gateway of the site *demosite*. |
| @ | ServerTaskProcessor | Service | Addresses the service *ServerTaskProcessor* hosted by the server application. |

Table 4.1: Different Types of Host Identifiers

**Connection Strings** azDeploy refers to database connection strings via an identifier. The database connection strings are stored on the appliances and not transmitted over the network, but referenced by a string like *"JornadaLocal"*. As the locally stored connection strings may differ between appliances for a connection string identifier, a specific connection string can only be referenced by specifying the appliance and the connection string identifier. This is useful in the presented scenario: Although each local database of a Jornada appliance requires different log-in credentials stored in the connection string, the administration center abstracts this and shows only one connection string identifier called *"JornadaLocal"*.

**Databases** Multiple databases may exist on an appliance. To uniquely address a database, azDeploy uses the class *Database* that contains three properties:

**Host:** The host identifier of the appliance hosting the database.

**ConnectionStringIdentifier:** The identifier which refers to the connection string. It is required as the referenced connection string contains the log-in credentials and the database server instance.

**Name:** The name of the database.

Table 4.2 shows how to address the database *JornadaLocalDB* on the appliance *demosite/EXAMPLEAPPLIANCE*, using the credentials stored in the connection string *JornadaLocal*.

| Property | Value |
|---|---|
| Host | `new Host("demosite","EXAMPLEAPPLIANCE")` |
| ConnectionStringIdentifier | JornadaLocal |
| Name | JornadaLocalDB |

Table 4.2: Database Identifier Example

**Database Backups** To refer to database backups, azDeploy uses *backup points*. A backup point identifies a set of database backups that were created at the same time. It can be used to undo a database schema upgrade that was applied to multiple databases by restoring the individual backups in a single step. Its class *BackupPoint* has a *Name* and a *Date* property.

**Upgrade Paths** Database schema upgrade scripts are referred to indirectly by an *upgrade path*. An upgrade path includes one or more upgrade scripts that are executed sequentially. It is represented through the class *UpgradePath* which contains one or more *DatabaseScriptInfo* objects.

## 4.3 Agent Upgrade Services

These WCF services upgrade the gateway agent and the client agent to the newest version.

**Protocol** The upgrade protocol works as follows: Before the gateway agent connects to the other WCF services hosted by the server application, it connects to *AgentUpgradeService* and retrieves the version information about the newest gateway agent release. If the version matches, it disconnects from the service and proceeds normally by connecting to the other services. In case of a version mismatch, the gateway agent upgrades itself: It repeatedly fetches blocks of data from *AgentUpgradeService* and writes them into a file until the service provides no more data. Then it disconnects from *AgentUpgradeService*. The transmitted file is a Windows Installer package. The gateway agent invokes a command sequence within a shell session that installs the package. Then the agent shuts down. Once Windows Installer has upgraded the agent installation, it starts the new agent version as a Windows service. Because now the newest version is already installed, the agent starts up normally.

For the client agent, the protocol works similar. However, it connects to the WCF service *AgentUpgradeProxy* running on the gateway agent. This proxy service forwards the requests from and responses to the client agent.

Therefore, both *AgentUpgradeService* and *AgentUpgradeProxy* fulfill the same service contract *IAgentUpgradeService* (Listing 4.8). To ensure compatibility with any agent version, the service contract uses basic data types and should not change.

```
[ServiceContract]
public interface IAgentUpgradeService {
    [OperationContract]
    string GetContractVersion();
    [OperationContract]
    byte[] GetClientChunk(string clientType, int index);
}
```

Listing 4.8: Service Contract for the Agent Upgrade Services

**Implementation** These services do not establish sessions. Instead of having a server side state, the client increases the *index* parameter whenever it calls the method *GetClientChunk* (Listing 4.9). It also indicates its client type, that is *Gateway* for the gateway agent and *Client* for the client agent. This stateless approach allows *AgentUpgradeService* to handle multiple requests at the same time with a single service object instance, as shown in the service behavior configuration (Listing 4.9, line 1). Due to the small filesize of the setup packages, *AgentUpgradeService* loads them completely into a plain byte array. Therefore no concurrency issues occur. *AgentUpgradeProxy*, which is hosted by the gateway agent, also has a single service object instance and allows concurrent service calls.

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple, InstanceContextMode =
    InstanceContextMode.Single)]
public class AgentUpgradeService : IAgentUpgradeService {
    private byte[] gatewayAgent = null;
    private byte[] clientAgent = null;
    private const int CHUNK_SIZE = 1024;

    public AgentUpgradeService() { ... } // load files into memory

    public string GetContractVersion() {
        return ContractMetadata.ContractVersion;
    }

    public byte[] GetClientChunk(string clientType, int index) {
        byte[] src = null;

        if (clientType == "Gateway")
            src = gatewayAgent;
        else src = clientAgent;

        if (index < 0 || index * CHUNK_SIZE >= src.Length)
            return new byte[] { };

        var length = CHUNK_SIZE;
        if (index * CHUNK_SIZE + CHUNK_SIZE >= src.Length)
            length = src.Length - index * CHUNK_SIZE;
        var res = new byte[length];
        for (int i = 0; i < length; i++)
            res[i] = src[CHUNK_SIZE * index + i];
        return res;
    }
}
```

Listing 4.9: *AgentUpgradeService* implementation

## 4.4 Operation Control Services

The operation control services represent the core WCF services of azDeploy. The server application hosts *MainService*, which offers an *IAdminService* endpoint and an *IControlService* endpoint. To control the operations, the administration center connects to the *IAdminService* endpoint. Each gateway agent hosts *GatewayService* which offers an *IGatewayService* endpoint. *GatewayService* connects to the *IControlService* endpoint of *MainService* and all client agents of a site connect to the *IGatewayService* endpoint of *GatewayService*.

### 4.4.1 Main Service

This section describes the two endpoints of *MainService*.

#### IAdminService Endpoint

The administration center uses the methods of the *IAdminService* endpoint (Listing 4.10 on page 25) to invoke operations on the server application, the gateway agents and the client agents. Before calling any other methods, the administration center calls the *LogOnAdmin* method. In this method, *MainService* establishes a session for the operator. Likewise, before disconnecting from *MainService*, the administration center calls the *LogOffAdmin* method. These methods do not take parameters as they extract the credentials from the underlying WCF authentication mechanisms. Before invoking operations on a site, the administration center calls the method *LockSite* for the respective site. This ensures that one site is administered by only one operator at a time. To unlock the site for other operators, administration center can lock another site, supply *null* as parameter to *LockSite* or log off. Since administration sessions do not expire in the current implementation, terminating the administration center without logging off keeps the site locked for other operators.

**Remote Operations** Invoking remote operations works in two manners, depending on the type of operation:

There are strictly synchronous calls like the *GetSites* method which returns all sites connected to the server application. In this case, the administration center is blocked until *MainService* has conducted the operation and returns the result.

In contrast, for longer running operations, the task system is used (see Section 4.5 on page 28). Such methods like *CreateDatabase* invoke the desired operation with the supplied parameters and return a task handle immediately. Although the administration center is blocked during the invocation call, it is not blocked while the operation is running. Once the operation has completed, the administration center is notified asynchronously about its result.

How does this notification work? *MainService* dispatches events to do so. Since the binding used by the *IAdminService* endpoint does not support callback contracts, *MainService* cannot push events to the administration center. Instead, the administration center periodically queries *MainService* for new events via the *FetchEvents* method. These events inform the administration center about the connection and disconnection of sites as well as about the completion of operations.

**Common Parameters** Several service methods employ parameters with the names *comment* or *backupPointDate*. The latter is only available in selected database operations. Whenever a method requires a *comment* parameter, the administration center must supply a string describing the reason for the operation. azDeploy stores this text into the database and product installation history tables.

In contrast, the *backupPointDate* parameter can be null. If the administration center supplies a *DateTime* object, azDeploy creates a database backup of the target database before applying the database operation. When the administration center applies the same operation, such as upgrading the database schema, to several databases at once, it supplies the same *DateTime* value for each method call. This way, azDeploy aggregates all database backups under a single *backup point* with the name *automatic* for the supplied date. In turn, the operator can undo the operation for all databases in a single step by restoring all aggregated database backups at once.

**Basic Query Operations** *GetSites* returns all sites connected to the server. *GetHosts* returns all appliances connected to the gateway of the specified site.

**Basic Database Operations** *CreateDatabase* creates a database. The *Database* object contains the name of the new database, the connection string identifier, and the target appliance. *DropDatabase* deletes the specified database. *BackupDatabase* creates a backup of the specified database. The *BackupPoint* object specifies the name and date of the backup point. *DeleteDatabaseBackup* deletes a database backup on the specified appliance. The *backupPath* parameter is the relative path of the backup file. The administration center extracts it from the *BackupInfo* object that describes the backup. *RestoreDatabase* restores a database backup to the specified database. Again, *backupPath* is used to reference the backup.

**Database Deployment Operations** *InitializeDatabase* puts the specified database under version control. The administration center has to specify the name of its database schema and the initial schema version. *UpgradeDatabase* upgrades the database schema of the specified database. The administration center must supply an upgrade path describing the SQL upgrade scripts to apply. *StoreDatabaseQuery* saves a database query on the server. It returns an ID which can be used to delete the query with *DeleteDatabaseQuery*.

**Database Query Operations** *GetDatabaseSchemas* returns the names of all available database schemas. *GetConnectionStringIdentifiers* retrieves the identifiers of all connection strings available on an appliance. *GetDatabaseQueries* returns all stored

```
[ServiceContract]
public interface IAdminService {
// Session Management
void LogOnAdmin();
void LogOffAdmin();
bool LockSite(string siteName);

// Basic Query Operations
string[] GetSites();
Host[] GetHosts(string siteName);
RemoteEvent[] FetchEvents();

// Basic Database Operations
CreateDatabaseTask CreateDatabase(Database database);
DropDatabaseTask DropDatabase(Database database, DateTime? backupPointDate);
BackupDatabaseTask BackupDatabase(Database database, BackupPoint backupPoint,
    string comment);
DeleteDatabaseBackupTask DeleteDatabaseBackup(Host host, string backupPath);
RestoreDatabaseTask RestoreDatabase(Database database, string backupPath, string
    comment, DateTime? backupPointDate);

// Database Deployment Operations
InitializeDatabaseTask InitializeDatabase(Database database, string schema,
    MVersion version, string comment, DateTime? backupPointDate);
UpgradeDatabaseTask UpgradeDatabase(Database database, DatabaseUpgradePath
    upgradePath, string comment, DateTime? backupPointDate);
int StoreDatabaseQuery(DatabaseQuery query);
void DeleteDatabaseQuery(int queryId);

// Database Query Operations
string[] GetDatabaseSchemas();
GetConnectionStringIdentifiersTask GetConnectionStringIdentifiers(Host host);
DatabaseQuery[] GetDatabaseQueries(string schema, MVersion version);
QueryDatabaseTask RunDatabaseQuery(Database database, string sqlText);
GetDatabaseBackupsTask GetDatabaseBackups(string siteName);
GetHostDatabasesTask GetHostDatabases(Host host);
GetSiteDatabasesTask GetSiteDatabases(string siteName);
GetDatabaseHistoryTask GetDatabaseHistory(Database database);
DatabaseUpgradePath[] GetDatabaseUpgradePaths(string schema, MVersion
    requiresVersion);

// Software Deployment Operations
ChangeSoftwareTask InstallProduct(Host host, Guid packageCode, string comment);
ChangeSoftwareTask UninstallProduct(Host host, Guid productCode, string comment);

// Software Query Operations
GetInstalledSoftwareTask GetInstalledSoftware(Host host);
GetInstalledSoftwareTask GetInstalledManagedSoftware(Host host);
GetProductHistoryTask GetProductHistory(Host host);
ProductPackageInfo[] GetProducts();
}
```

Listing 4.10: *IAdminService* Service Contract (shortened)

database queries suitable for the specified database schema and version. *RunDatabase-Query* then executes the supplied SQL query statement on the specified database. *Get-DatabaseBackups* queries all backups on the specified site. *GetHostDatabases* queries all databases available on the specified appliance. *GetSiteDatabases* queries all databases available within the specified site. *GetDatabaseHistory* queries the upgrade history of the specified database. *GetDatabaseUpgradePaths* returns all available upgrade paths for the specified database schema and starting version.

**Software Deployment Operations** *InstallProduct* installs or upgrades an application. The administration center must specify the target appliance and the package code of the installation package. *UninstallProduct* removes an application. The administration center must specify the target appliance and the product code of the application to remove.

**Software Query Operations** *GetInstalledSoftware* and *GetInstalledManagedSoftware* retrieve the applications installed on an appliance. *GetInstalledManagedSoftware* only returns vendor applications. *GetProductHistory* queries the history of software installations and uninstallations on an appliance. *GetProducts* returns all software installation packages available in the server repository.

#### IControlService Endpoint

The *IControlService* endpoint offers three methods (Listing 4.3 on page 16). The gateway agent uses the methods *LogOn* to connect to and *LogOff* to disconnect from *MainService*. These methods take no parameters as they extract the credentials from the WCF authentication mechanisms. Further, the gateway agent calls the method *NotifyEvent* to inform the server application about completed operations and newly connected or disconnected appliances. Finally, *IControlService* employs the callback contract *IGatewayCallback* (Listing 4.5 on page 17). It allows *MainService* to query which appliances are connected to the gateway agent and to invoke remote operations via the *RunTasks* method.

### 4.4.2 Gateway Service

*GatewayService* offers an *IGatewayService* endpoint (Listing 4.11). All appliances of a particular site connect to it. Its service contract is very similar to *IControlService*. However, because *GatewayService* does not employ WCF authentication mechanisms, client agents supply their hostname when they call the method *LogOn*. In turn, this method returns them the site name. Appliances can also inform their gateway about completed operations via the method *NotifyEvent*.

*IGatewayService* employs the callback contract *IHostCallback* (Listing 4.12). Through it, *GatewayService* can query the databases and database backups available on an appliance. Finally, it can invoke operations on an appliance via the method *RunTask*.

```
[ServiceContract(SessionMode = SessionMode.Required, CallbackContract = typeof(
    IHostCallback))]
public interface IGatewayService {
    [OperationContract(IsInitiating = true)]
    [FaultContract(typeof(IdentifierInUseFault))]
    string LogOn(string hostName);
    [OperationContract(IsInitiating = false, IsTerminating = true)]
    void LogOff();
    [OperationContract(IsOneWay = true)]
    void NotifyEvent(RemoteEvent ev);
}
```

Listing 4.11: *IGatewayService* Service Contract

```
public interface IHostCallback {
    [OperationContract]
    DatabaseInfo[] GetDatabases();
    [OperationContract]
    BackupInfo[] GetDatabaseBackups();
    [OperationContract]
    void RunTask(Task task);
}
```

Listing 4.12: *IHostCallback* Callback Contract

### 4.4.3 Remote Events

azDeploy uses remote events to notify its applications about connecting or disconnecting sites and appliances as well as completed operations.

The client agent sends remote events. The server application and the gateway agent send, process and relay remote events. The administration center only processes remote events. The gateway agent and the client agent transmit remote events via the one-way operation `void NotifyEvent(RemoteEvent ev)` of the *IControlService* and *IGatewayService* endpoints. Its one-way character implies that the event sender does not care when its counterpart processes the event (see page 19). All remote events derive from the class *RemoteEvent* and add further properties. *RemoteEvent* only contains the property *Time*, indicating when the event was created.

azDeploy employs several remote event types:

**HostConnectivityChangedEvent:** Indicates that the state of a client agent's connection to its gateway agent has changed. The property *Host* designates the affected appliance. If the property *IsOnline* has the value *true*, the client agent has connected to its gateway agent. Otherwise, the client agent has disconnected from its gateway agent.

**SiteConnectivityChangedEvent:** Indicates that the state of a gateway agent's connection to the server application has changed. The property *SiteName* designates the name of the affected site. If the property *IsOnline* has the value *true*, the gateway agent has connected to the server application. Otherwise, the gateway agent has disconnected from the server application.

**TaskCompletedEvent:** Indicates that an operation has completed and also contains its results.

**TaskFailedEvent:** Indicates that an operation has failed. It contains a handle for the affected operation and an error message.

**ClientAliveEvent:** The gateway agent and the client agent send this remote event periodically. It contains the property *Identifier*. azDeploy uses this event type as a heartbeat mechanism to detect if a communication peer has crashed.

Table 4.3 depicts the flow of the discussed remote event types. In this representation, events flow from right to left. *Send* means that the application creates events of the specified type. *Process* means that the application reacts to the specified event type. *Relay* means that the application relays a received event to the next application. The server application processes task events that come from the gateway agent or the client agent, but also sends task events to the administration center.

| Remote Event Type | Admin. Ctr. | Server | Gateway | Client |
|---|---|---|---|---|
| HostConnectivityChangedEvent | process | relay | send | - |
| SiteConnectivityChangedEvent | process | send | - | - |
| TaskCompletedEvent | process | send/process | send/relay | send |
| TaskFailedEvent | process | send/process | send/relay | send |
| ClientAliveEvent | - | process | send/process | send |

Table 4.3: Remote Event Sources and Sinks

**Event Subscription** Subscribing to remote events works implicitly: *MainService* intercepts events in its *NotifyEvent* service method. Depending on the event type, it notifies either one specific or all administration center instances. *HostConnectivity-ChangedEvent* and *SiteConnectivityChangedEvent* are of general interest and therefore all administration center instances receive them. *TaskCompletedEvent* and *Task-FailedEvent* affect specific operations and therefore only the initiator of the operation receives them. Finally, *MainService* does not dispatch the *ClientAliveEvent* to the administration center at all. Since *MainService* cannot use a callback contract to inform the administration center about events, *MainService* stores the events for all administration center sessions. The administration center fetches new events periodically via the *FetchEvents* method of the *IAdminService* endpoint. The administration center's components then can subscribe to specific events (see Section 5.4 on page 54).

## 4.5 Tasks

The task system allows azDeploy to execute operations asynchronously on the server, the gateways, and the appliances. Section 4.5.1 explains the task system from the external perspective of the administration center. Section 4.5.2 discusses the internal workings hidden from the administration center.

## 4.5.1 External View of the Task System

To run a remote operation asynchronously, the administration center first must tell the other components of azDeploy to execute it. Once the operation has completed, the administration center can interpret its result.

### Invoking an Asynchronous Remote Operation

To invoke an asynchronous remote operation, the administration center has to deal with a task object. Its properties contain the parameters for the operation and the following information:

**Host:** This describes on which machine to run the operation. Since this property's type is *Host*, the administration center can address services of the server application, a gateway or an appliance like indicated in Table 4.1 on page 19.

**TaskId:** The handle azDeploy uses to refer to the operation. Its data type is globally unique identifier (GUID). *MainService* assigns its value.

**Username:** The Windows principal of the user who wants to run the operation.

For the specific operations, azDeploy uses individual task classes deriving from the class *Task*. Since these task objects are transferred between the applications, they are all data contracts. They employ the naming convention *"<Verb><Noun>Task"*.

For instance, to create a database, the administration center uses a task object of the type *CreateDatabaseTask* (Listing 4.13). It has an additional property *Database* of the type *Database* containing the name of the database and the connection string to use.

```
[DataContract]
[Task(TaskResultType = typeof(DatabaseTaskResult))]
public class CreateDatabaseTask : Task {
    [DataMember]
    public Database Database { get; set; }

    public CreateDatabaseTask(Database database) {
            Database = database;
    }
}
```

Listing 4.13: The *CreateDatabaseTask* Task Class

However, the administration center does not need to construct a task object: The *IAdminService* endpoint of *MainService* (Listing 4.10 on page 25) offers methods which take the parameters for the operation, start the operation and return the task object.

Running a task consists of three steps:

1. Registration: The administration center submits the parameters for the task to *MainService* by calling the corresponding method on *IAdminService*. For example, to create a database, the administration center calls the method *CreateDatabase*. In turn, *MainService* assigns a *TaskId* and returns the task object of the type *CreateDatabaseTask*. This operation is invoked synchronously.

2. Execution: From the administration center's point of view, the task is executing asynchronously. The administration center continues to run meanwhile.

3. Interpretation: Once the task has terminated, *MainService* notifies the administration center via an event. In turn, the administration center can process the results of the operation.

### Interpreting the Operation's Result

In the interpretation step, the administration center has to handle the three outcomes of the task execution:

- Task completed with success
- Task completed with errors
- Task failed

**Task Completion** If a task has completed, the administration center receives a *TaskCompletedEvent*. It contains a *TaskResult* object (Listing 4.14). Like specific task classes, there are also specific task result classes deriving from the class *TaskResult*. These classes contain further information about the outcome of the task. The property *TaskResultType* of the task class' *Task* attribute denotes its task result type (Listing 4.13, line 2). The task result type for *CreateDatabaseTask* is *DatabaseTaskResult* (Listing 4.15).

The administration center determines the task the result belongs to through the *TaskId* property. Then it inspects the task result further: If the property *Success* has the value *true*, then the operation has executed without errors. Depending on the task type, the administration center may access further properties which contain the results of the operation. This does not apply to *DatabaseTaskResult*, but is used when querying data.

In contrast, if *Success* has the value *false*, the operation did not complete due to errors. The administration center can find out more about the error by inspecting the error information within the task result object. Whether and in which form such an error information is available depends on the task result type. *DatabaseTaskResult* provides an *Error* property that indicates a database error class, an error message and the line number where the error occurred.

```
[DataContract]
public abstract class TaskResult {
        [DataMember]
        public bool Success { get; set; }
        [DataMember]
        public Guid TaskId { get; set; }
        ...
}
```

Listing 4.14: The *TaskResult* Class

```
[DataContract]
public class DatabaseTaskResult : TaskResult {
    [DataMember]
    public DatabaseOperationError Error { get; set; }
    ...
}

[DataContract]
public class DatabaseOperationError {
    [DataMember]
    public byte Class { get; set; }
    [DataMember]
    public string Message { get; set; }
    [DataMember]
    public int LineNumber { get; set; }
    ...
}
```

Listing 4.15: The Task Result Type for Database Operations and its Error Information

**Task Failure** If a task has failed, the administration center receives a *TaskFailedEvent*. This means that neither the operation succeeded nor that there is structured error information available. This can happen due to two reasons: Firstly, the implementation of the task may not have returned a proper task result object, but has thrown an exception. This may indicate a bug in the task implementation. Secondly, it may be that the gateway agent or client agent went offline and thus, the operation could not be executed.

The *TaskFailedEvent* includes the property *TaskId* that indicates the corresponding task and the property *ErrorMessage* containing an error string. If a task implementation throws an exception, *ErrorMessage* contains the exception message.

Note that AZDEPLOY can distinguish between these three outcomes without knowing anything about the specific task type. The inner workings of the task system described in Section 4.5.2 take advantage of this abstraction.

### 4.5.2 Inner Workings of the Task System

This section discusses the mechanisms of the task system which are only visible to the server application.

From the perspective of the administration center and the target agent, a task starts executing a single operation and returns its result asynchronously. Therefore, the administration center needs to call a method on *IAdminService*, retain a reference to the returned task object and wait for the event which indicates the task completion or the task failure. The target agent receives a task object, runs the operation and sends a task completion or a task failure event when it finishes. Once a task has terminated, the administration center and the target agent can forget about it.

Although this *external view* is simple and desirable, it is constraining at the same time. Thus, the server application maintains this model to the outside, but employs a more flexible model internally.

For instance, the administration center uses *IAdminService's* method `DropDatabaseTask DropDatabase(Database database, DateTime? backupPointDate)` to delete an existing database. If the administration center supplies a backup point date, AZDEPLOY creates a backup before deleting the database (see page 24). To make this work with the external view (see Section 4.5.1), *DropDatabaseTask* would need an additional parameter to store the backup point date. Also, the client agent's task implementation would need to make a backup before dropping the database. The same would apply to other tasks offering a backup option, such as initializing, upgrading and restoring a database.

**Composite Tasks** Instead of implementing backup functionality within each task implementation, the server application uses *composite tasks*. A composite task consists of multiple *sub-tasks*. In the example where the administration center wishes to back up the database prior to deleting it, the server application creates a composite task with two sub-tasks of the types *BackupDatabaseTask* and *DropDatabaseTask*. Once the backup task finished, the server application starts the drop database task. Finally, it returns the task result of the drop database task to the administration center.

To adhere to the external view, the server application must consider the following:

- The administration center still starts a *single task* and receives a *single task result* at the end of all sub-task operations. Thus, the returned task result must be of the *same type*. Also, the *TaskId* of the returned task result or task failure must reference the *initial task*.
- The gateway and client agents do not need to know that the sub-tasks belong together. Therefore, they can still treat each task *separately*.

The server application uses the class *CompositeTask* to internally represent a composite task. It has three properties:

**OwnerSession:** This GUID indicates the administration center session starting the initial task. It is stored separately as no task object holds this information.

**PublicTask:** The initial task. It is the only task that the administration center knows. Therefore the composite task's final outcome must reference its *TaskId*.

**ControlFlow:** It determines the sequence of the composite task's sub-tasks.

For greater flexibility, a composite task stores its individual sub-tasks not in a collection, but uses a task control flow instead.

```
1  interface ITaskControlFlow {
2      ControlFlowDecision OnStart();
3      ControlFlowDecision OnTaskCompleted(TaskResult result);
4      ControlFlowDecision OnTaskFailed(TaskFailedEvent ev);
5  }
6
7  class ControlFlowDecision {
8      public enum CFDOutcome { Completed, Failed, RunTask, Wait };
9      public CFDOutcome Outcome { get; private set; }
10     public TaskResult TaskResult { get; private set; }
11     public string TaskFailedMessage { get; private set; }
12     public Task[] NextTasks { get; private set; }
13
14     public static ControlFlowDecision CreateTaskCompletedDecision(TaskResult result
           ) {...}
15     public static ControlFlowDecision CreateTaskFailedDecision(string message)
           {...}
16     public static ControlFlowDecision CreateRunTasksDecision(params Task[] tasks)
           {...}
17     public static ControlFlowDecision CreateWaitDecision() {...}
18     ...
19 }
```

Listing 4.16: *ITaskControlFlow* and *ControlFlowDecision*

**Task Control Flows** Depending on the previous task's outcome, a *task control flow* determines the next tasks to run. It may also terminate the composite task. A task control flow is a class implementing the interface *ITaskControlFlow* (Listing 4.16).

When a composite task starts, it calls its task control flow's *OnStart* method. Then, the task control flow reacts to the termination of each sub-task: When a task completes, the composite task calls the method *OnTaskCompleted*. In case a task fails, it calls the method *OnTaskFailed*. All three methods return a *ControlFlowDecision* (Listing 4.16).

The returned *ControlFlowDecision's* property *Outcome* determines how the composite task proceeds. It can have four values (Listing 4.16, line 8):

**Completed:** This outcome completes the composite task. The task control flow stores the task result into the *ControlFlowDecision's TaskResult* property. AZDEPLOY allows this outcome only if all sub-tasks have already terminated.

**Failed:** This outcome lets the composite task fail. The task control flow stores the error message for the *TaskFailedEvent* into the *ControlFlowDecision's TaskFailedMessage* property. Again, AZDEPLOY allows this outcome only if all sub-tasks have already terminated.

**RunTasks:** This outcome starts one or more sub-tasks. The task control flow stores their task objects in the *ControlFlowDecision's NextTasks* property.

**Wait:** This outcome leads to no action. It is only allowed when there are still sub-tasks running.

Table 4.4 lists the outcomes allowed on whether there are running sub-tasks or not. *ControlFlowDecision* works as a factory class. Instead of calling a constructor, the task control flow calls the factory method according to the desired outcome with the required parameters.

| State/Outcome | Completed | Failed | RunTasks | Wait |
|---|---|---|---|---|
| Sub-tasks running | | | x | x |
| No sub-tasks running | x | x | x | |

Table 4.4: Allowed Control Flow Decision Outcomes Depending on Composite Task State

azDeploy uses three task control flow classes:

**LinearControlFlow:** This control flow executes a sequence of tasks passed to its constructor (Listing 4.17). In case a sub-task fails or completes with errors, it terminates the composite task. If the sub-task's task result type matches the public task's one, it reports the result through a decision with outcome *Completed*. Otherwise, it returns a decision with outcome *Failed*, as it cannot return an unexpected task result type to the administration center.
All database operations offering back-up option use this control flow. Also, all composite tasks which contain a single task use it.

**InstallProductControlFlow:** This control flow installs a software product on an appliance. It copies the installation package to the appliance and starts the installation. It may be that a previous release was uninstalled during setup. If installing the new release fails, *InstallProductControlFlow* reinstalls the old release to restore the previous state.

**LoggingControlFlow:** This control flow wraps any other control flow. It forwards all created sub-tasks and their outcome to the logging service. To do so, it starts a *LogTask* for each terminated sub-task of the wrapped control flow.

**Example** Listing 4.18 shows *MainService's DropDatabase* method. Firstly, it creates the task objects for the sub-tasks. *InitTask* is a convenience method to set the task object's properties *TaskId*, *Username* and *Host*. Secondly, *DropDatabase* creates a composite task object. It contains the session, the public task and the task control flow. Note that *LoggingControlFlow* wraps *LinearControlFlow* to enable logging of all executed tasks. Fourthly, it uses the *TaskManager* class to start executing the composite task. Finally, it returns the public task object to the administration center.

Figure 4.2 on page 36 shows the interaction between the administration center, the server application and the client agent. The figure does not show the gateway agent since it merely forwards messages. In this example, the administration center wants to create a backup automatically prior to deleting the database. The bars indicate time spans in which the respective application is blocked due to the task execution. Note that the interaction between the server application and the client agent is not asynchronous, as the UML notation suggests, but is meant to represent one-way operations.

```
class LinearControlFlow : ITaskControlFlow {
    private Task[] tasks;
    private int current_task;

    public LinearControlFlow(params Task[] tasks) {
        this.tasks = tasks;
    }

    public ControlFlowDecision OnStart() {
        return ControlFlowDecision.CreateRunTasksDecision(tasks[0]);
    }

    public ControlFlowDecision OnTaskCompleted(TaskResult result) {
        if (current_task == tasks.Length - 1) // last task
            return ControlFlowDecision.CreateTaskCompletedDecision(result);
        else if (result.Success)
            return ControlFlowDecision.CreateRunTasksDecision(tasks[++current_task
                ]);
        else
        {
            var type = TaskAttribute.GetAttribute(tasks[tasks.Length - 1]).
                TaskResultType;
            if (type.IsAssignableFrom(result.GetType()))
                return ControlFlowDecision.CreateTaskCompletedDecision(result);
            else return ControlFlowDecision.CreateTaskFailedDecision(string.Format(
                "Sub-Task {0} completed with errors.", result.TaskId));
        }
    }

    public ControlFlowDecision OnTaskFailed(TaskFailedEvent ev) {
        return ControlFlowDecision.CreateTaskFailedDecision(ev.ErrorMessage);
    }
}
```

Listing 4.17: The Linear Task Control Flow

```
public DropDatabaseTask DropDatabase(Database database, DateTime? backupPointDate)
    {
    var session = AdminSessionManager.GetUserSession(OperationContext.Current.
        ServiceSecurityContext.PrimaryIdentity.Name);
    var list = new List<Task>();

    if (backupPointDate.HasValue)
    {
        list.Add(AdminSessionManager.InitTask<BackupDatabaseTask>(session, database
            .Host, new BackupDatabaseTask(database, new BackupPoint("automatic",
            backupPointDate.Value), "Automatic before database drop.")));
    }
    var dropDatabaseTask = AdminSessionManager.InitTask(session, database.Host, new
         DropDatabaseTask(database));
    list.Add(dropDatabaseTask);

    var composite = new CompositeTask(session, dropDatabaseTask, new
        LoggingControlFlow(session,new     LinearControlFlow(list.ToArray())));
    TaskManager.Start(composite);
    return dropDatabaseTask;
}
```

Listing 4.18: The *DropDatabase* Service Method of *MainService*

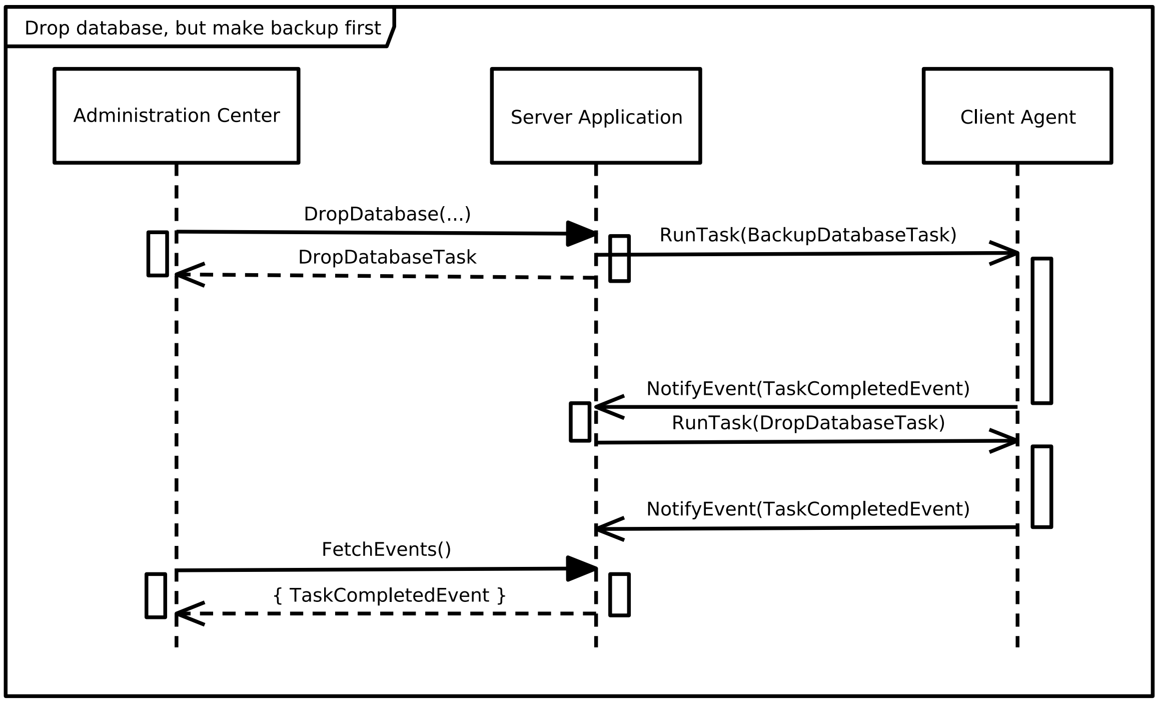

Drop database, but make backup first
Administration Center
Server Application
Client Agent
DropDatabase(...)
DropDatabaseTask
RunTask(BackupDatabaseTask)
NotifyEvent(TaskCompletedEvent)
RunTask(DropDatabaseTask)
NotifyEvent(TaskCompletedEvent)
FetchEvents()
{ TaskCompletedEvent }

Figure 4.2: Drop Database Task Sequence

**Task Scheduling** The class *TaskManager* schedules all tasks. It ensures that gateway and client agents execute only one operation at a time. *TaskManager* holds back other tasks wanting to run on the same gateway or appliance in a task queue until the prior tasks have finished. The agents run the task operations within a separate thread: This avoids setting the WCF response timeouts unnecessary high as task operations may take a long time to complete. Thus, the method call is over as soon as the actual task operation starts.

In contrast, *TaskManager* assumes that all server application services support executing multiple tasks in parallel: The server task processor service logs task executions. Therefore it may need to log a task from appliance B while it still processes the logging of a task from appliance A. The same applies to the file transfer service as there may be multiple file transfers to different appliances at a time.

**Helper Classes** The task system requires each task to be executed within a composite task. Therefore, to execute a single task, *MainService* uses the helper class *SingleTaskPattern*. Its *Execute* method allows to start a task with a single line of code. The last parameter determines whether to log the task execution. Listing 4.19 shows its use in the service method to retrieve an appliance's databases. As this query task does not change the appliance's configuration, the method turns off the task logging.

To execute the task operation, gateway and client agents use the class *TaskRunner*. It enables developers to write task implementation methods which take a task of a specific task type as parameter and return a task result object of the corresponding task result type. *TaskRunner* uses reflection to call the appropriate task implementation method within a separate thread. At startup, *TaskRunner* checks whether each task implementation method returns the appropriate task result type. Listing 4.24 on page 46 shows the implementation of the *BackupDatabaseTask*. After calling the task implementation method, *TaskRunner* sends back a *TaskCompletedEvent* containing the task result to the server application via the gateway agent. If the task implementation method throws an exception, *TaskRunner* catches it and sends back a *TaskFailedEvent*.

```
public GetHostDatabasesTask GetHostDatabases(Host host) {
    var session = AdminSessionManager.GetUserSession(OperationContext.Current.
        ServiceSecurityContext.PrimaryIdentity.Name);
    return SingleTaskPattern.Execute(session, host, new GetHostDatabasesTask(),
        false);
}
```

Listing 4.19: The *GetHostDatabases* Service Method of *MainService*

## 4.6 Notification Services

As azDeploy does not know about the internals of the deployed applications, it cannot answer questions such as:

- Can I upgrade the server's main database without affecting the Jornada appliances?
- Is it necessary to restart the Jornada client after a minor database schema upgrade?
- Can I upgrade the Jornada client application on an appliance or is it in use?

But finding answers to such questions is crucial for seamless operation. Therefore, azDeploy lets the applications decide, as they can answer these questions best.

The notification services allow an application to

- track configuration changes taking place at the customer site
- deny configuration changes affecting operations negatively

Configuration changes may affect applications in various ways:

**Create or Initialize Database:** The application may want to use this information to configure database connectivity.

**Upgrade Database Schema:** The application may be in use. If idle, it may need to disconnect from the target database.

**Backup Database:** Creating a backup may reduce database performance the application currently needs.

**Restore or Drop Database:** The application may use the target database. If idle, it should disconnect from the target database.

**Install or Uninstall Product:** The target application may be in use. If idle, it must shut down properly.

The design of the notification services takes the following into account:

- The developers of the Jornada applications do not know about the internals of azDeploy.
- The interface of the notification services must be stable.
- Working with the notification services requires no additional dependencies except the .NET-Framework.

Therefore, the notification services implement a generic protocol which relies on a few data contracts only. Section 4.6.1 discusses the application interface of the notification services, while Section 4.6.2 explains how they integrate within azDeploy.

### 4.6.1 Application Interface

Because applications decide whether a configuration change should take place, the client agent conducts a configuration change only if:

1. All applications at a customer site respond to the client agent.
2. All applications at a customer site grant the configuration change.

This approach ensures that applying the configuration change does not affect operations negatively. Because a non-responding application has an unknown state, the configuration change cannot be applied safely in this case.

An application can react to a configuration change at three points in time which are called notification stages:

**Request:** The client agent requests permission from the application to conduct a configuration change. The application can either grant or deny it. When denying it, the application can state whether the denial is temporary or permanent. Denying a database schema upgrade while the application is in use would be temporary, while an attempt to drop the productive database may result in a permanent denial.
If the application grants the configuration change, it has to ensure that it stays in a state which allows the configuration change. Thus, in case of an database schema or application upgrade, the application should switch into a "locked" state so that no user can start working with the application.

In case AZDEPLOY already knows that an application denied the configuration change, it may not send the configuration change request to the remaining applications.

**Activity:** In case all applications granted the configuration change, the client agent informs the applications that it will start the operation. As the application now knows that the configuration change will take place, it prepares for it: For example, before upgrading the application, the currently running instance must shut down.
After this notification, the client agent starts the configuration change.

**Completion:** Finally, the client agent informs the application that it completed the configuration change. It also sends this notification for denied configuration changes which did not take place. The application may switch back to normal operation.

Figure 4.3 depicts the transitions between these three notification stages.

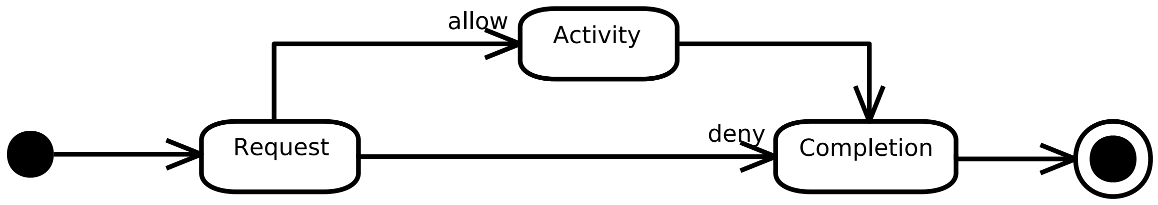


Figure 4.3: Notification Stage Transitions

Listing 4.20 shows the service definition of the notification service. When an application starts, it logs onto the service. While running, it will be notified about configuration changes via the callback contract. Finally, when an application shuts down, it logs off from the service.

The client agent calls the three methods of *INotificationCallback* (Listing 4.20) corresponding to the three notification stages discussed above. They have several similar parameters:

**hostname:** The hostname of the appliance which the configuration change affects.

**action:** This string identifies the type of the configuration change. Configuration changes affecting databases start with *Database* whereas these affecting applications start with *Product*.

**reference:** Applications use this identifier to track a configuration change across its different notification stages.
For example, when granting a database schema upgrade, the application may store this value and switch to a *"locked"* state. Then, in *NotifyCompletion*, it just needs to check whether the *reference* parameter matches the stored value. If

so, it can return to normal operations, given that it is compatible with the new schema version.

**parameters:** This dictionary of string keys and values provides further information about the configuration change.
For instance, in case of a database schema upgrade request, the key *ToVersion* contains the new version number of the database schema.

**success:** Only available in *NotifyCompletion*. Indicates whether the configuration change was applied successfully. The value *false* can also mean that an application denied the configuration change.

```
[ServiceContract(SessionMode = SessionMode.Required, CallbackContract = typeof(
    INotificationCallback))]
public interface INotificationService {
    [OperationContract(IsInitiating = true)]
    void LogOn();
    [OperationContract(IsInitiating = false, IsTerminating=true)]
    void LogOff();
}

public interface INotificationCallback {
    [OperationContract]
    void NotifyActivity(string hostname, string action, Guid reference, Dictionary<
        string, string> parameters);
    [OperationContract(IsOneWay = true)]
    void NotifyCompletion(string hostname, string action, bool success, Guid
        reference, Dictionary<string, string> parameters);
    [OperationContract]
    RequestResult Request(string hostname, string action, Guid reference,
        Dictionary<string, string> parameters);
}
```

Listing 4.20: *INotificationService* and *INotificationCallback*

### Configuration Changes and Parameters

Table 4.5 shows the configuration change types. For convenience, the class *Actions* in the assembly Client.Contracts contains these action strings as constants. Table 4.6 explains the parameters of the different actions. The notification services pass these parameters to the applications in each notification stage (request, activity and completion) of a configuration change. The *Database.** row lists the parameters which all database actions supply whereas the *Product.** row lists the parameters which all product actions supply.

**Application Upgrades** If an application wants to react to application upgrades, it listens for both the actions *Product.Install* and *Product.Uninstall*. This is because Microsoft Windows Installer distinguishes two types of application upgrades:

| Action Constant | String Value |
|---|---|
| Actions.DatabaseCreate | Database.Create |
| Actions.DatabaseInitialize | Database.Initialize |
| Actions.DatabaseUpgrade | Database.Upgrade |
| Actions.DatabaseBackup | Database.Backup |
| Actions.DatabaseRestore | Database.Restore |
| Actions.DatabaseDrop | Database.Drop |
| Actions.ProductInstall | Product.Install |
| Actions.ProductUninstall | Product.Uninstall |

Table 4.5: Configuration Change Types in Notifications

| Action | Parameter | Description |
|---|---|---|
| Database.* | Database | Name of the database |
| | ConnectionStringIdentifier | Connection string identifier |
| Database.Initialize Database.Restore | Schema | Database schema name. Null, if restored database is not initialized. |
| | Version | Database schema version. Null, if restored database is not initialized. |
| Database.Upgrade | Schema | Database schema name |
| | FromVersion | Current schema version |
| | ToVersion | Schema version after the upgrade. |
| Product.* | ProductCode | MSI product code (GUID) |
| | ProductName | MSI product name |
| | ProductVersion | MSI product version |
| | Manufacturer | MSI manufacturer |
| | PackageCode | MSI package code (GUID) |
| Product.Install | UpgradeCode | MSI upgrade code |
| | AZProperty.* | Properties in the package's *AZProperty* table are exported as prefixed parameters. |
| Product.Uninstall | IsUpgrade | If true, the product was removed because of an upgrade to a newer version. If false, it was removed normally. |

Table 4.6: Configuration Change Parameters

Firstly, Windows Installer applies certain upgrades in one step itself. In this case, the application only gets informed about the *Product.Install* action and the action's parameters contain information about the new application release.

Secondly, there are upgrades which azDeploy applies by first uninstalling the old application release and then installing the new application release. In this case, it notifies the application firstly about the *Product.Uninstall* action. The action parameters contain information about the application version to be uninstalled. Then, it notifies the application about the *Product.Install* action. This time, the action parameters contain information about the new application release.

Note that it does not suffice to react only to the *Product.Install* action: Although *Product.Install* occurs in either case, the application may need to react to *Product.Uninstall* already: Because if the listening application itself is about to be uninstalled, it needs to shut down properly before.

When an application is notified about the *Product.Uninstall* action, it can determine whether this is a permanent removal or due to an application upgrade by inspecting the parameter *IsUpgrade*. This parameter can have the value *true* or *false*.

But how can an application decide whether it is removed during an upgrade, if no *Product.Uninstall* action appears? In this case, it can inspect the parameter *UpgradeCode*. If it matches the application's upgrade code, it is affected by the upgrade. Section 6.2 will explain the details of the Windows Installer integration.

## Handling Configuration Change Requests

To handle a configuration change request, the application returns a *RequestResult* object. The application can create it by either calling the static method `RequestResult Allow()` or `RequestResult Deny(ErrorType errorType, int errorCode, string errorMessage)` on the class *RequestResult*. If the application denies the request, it supplies three parameters:

**errorType:** The application states whether the denial is temporary or permanent.

**errorCode:** An arbitrary error code.

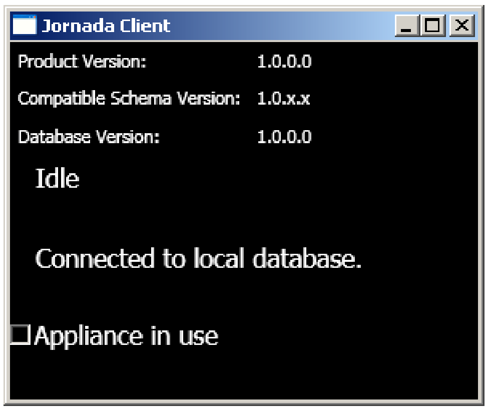
**errorMessage:** An error message describing the reason for the denial.

azDeploy only takes into account whether the application grants or denies the request. The error information is used to educate the operator about the reason for the denial.

## Example

The Jornada Client demonstrates how vendor applications interact with the notification services. Its window displays the application's state (Figure 4.4). The class *NotificationClient* interacts with the notification services and determines the application's reaction to the configuration changes.

The Jornada Client resembles the behavior of the Jornada scenario's client application (see Section 1.1 on page 3) in a simplified way: It has no business logic. Rather, the user can decide whether the application pretends that it is *idle* or *in use*. *In use* simulates that an employee currently uses the appliance. Additionally, the JORNADA Client writes log data into the appliance's database. For the sake of brevity, it does not take into account the existence of the JORNADA server machine. Thus, in a real world scenario, the JORNADA Client would also have to track configuration changes affecting the server application and the central *main database*.


Jornada Client

Product Version: 1.0.0.0
Compatible Schema Version: 1.0.x.x
Database Version: 1.0.0.0
Idle
Connected to local database.
☐ Appliance in use

Figure 4.4: Jornada Client

**Application Behavior** The JORNADA Client interacts with AZDEPLOY as follows: While it is idle, it grants to upgrade itself. During such an application upgrade, the application shuts down prior to uninstallation.

Concerning its local database, the application permanently denies the request to delete it. The application allows to upgrade the local database's schema or to restore a backup only while it is idle. When these database operations occur, the application switches to the state *locked* and disconnects from the database. In this state, the application is idle, but the user cannot interact with it. After AZDEPLOY has conducted the operation, the application checks whether it is still compatible with the changed database schema version. It considers schema versions where the first two numbers match as compatible. If the database schema is still compatible, the application reconnects to the database, switches to the state *idle* and can be used again. Otherwise, if the database schema is incompatible, the application stays in the state *locked* and waits for further configuration changes which restore compatibility. This could be a rollback to a previous database backup or an application upgrade.

Because only *NotificatonClient* is of interest, the application is abstracted through the interface *IApplicationControl* (Listing 4.21). It allows *NotificationClient* to interact with the application: Firstly, the application exports the name of the local database and its schema version. Secondly, *NotificationClient* can try to switch the application's state to *locked* and unlock it again. Finally, it can change the database connection state or shut down the application.

```
interface IApplicationControl {
    int LocalDatabaseMajorVersionNumber { get; }
    int LocalDatabaseMinorVersionNumber { get; }
    string LocalDatabaseName { get; }
    bool TryLockApplication();
    void UnlockApplication();
    void ConnectToLocalDatabase();
    void DisconnectFromLocalDatabase();
    void ShutdownApplication();
    void DisplayDatabaseVersion(Version version); // updates UI only
}
```

Listing 4.21: *IApplicationControl* Interface

```
[CallbackBehavior(ConcurrencyMode=ConcurrencyMode.Reentrant)]
class NotificationClient : INotificationCallback {
    private List<Guid> notificationFilter = new List<Guid>();
    public IApplicationControl ApplicationControl { get; set; }
    private volatile bool stopNotificationClient = false;
    ...
    private RequestResult TryLockApplication(Guid reference) {
        var success = ApplicationControl.TryLockApplication();
        if (success)
        {
            notificationFilter.Add(reference);
            wasActivityRun = false;
            return RequestResult.Allow();
        }
        else return RequestResult.Deny(ErrorType.Temporary, 2, "JornadaClient is in
             use.");
    }

    public RequestResult Request(string hostname, string action, Guid reference,
        Dictionary<string, string> parameters) {
        if (hostname.ToLowerInvariant() == Environment.MachineName.ToLowerInvariant
            () && !stopNotificationClient) // on local host
        {
            if (action.StartsWith("Database.") && parameters["Database"] ==
                ApplicationControl.LocalDatabaseName)
            {
                if (action == Actions.DatabaseDrop)
                    return RequestResult.Deny(ErrorType.Permanent, 1, "Not allowed
                        to drop production database.");
                else if (action == Actions.DatabaseRestore || action == Actions.
                    DatabaseUpgrade)
                    return TryLockApplication(reference);
                else return RequestResult.Allow();
            }
            else if ((action == Actions.ProductUninstall || action == Actions.
                ProductInstall) && parameters["ProductName"] == "Jornada Client")
            {
                return TryLockApplication(reference);
            }
        }
        return RequestResult.Allow();
    }
```

Listing 4.22: Handling Configuration Change Requests

**Configuration Change Requests** Listing 4.22 shows how *NotificationClient* handles configuration change requests. Because the configuration change action is supplied as string, it is used to aggregate all database configuration changes. Whenever the application changes its state to *locked*, it retains the configuration change's *reference*. Thus, it can determine the notifications of interest in the following stages without having to inspect all parameters.

**Reacting to Configuration Changes** Listing 4.23 contains the callback methods for the notification stages *Activity* and *Completion*. Although the *Request* method already switches the application to *"locked"* for certain operations, the actual reaction takes place in the activity stage. This is because other applications may have denied the request so that the configuration change does not happen. In this case, the configuration change would directly transist from the *Request* stage to the *Completion* stage. Finally, after upgrading the database schema or restoring a backup, the application returns to the *idle* state, given that the database schema is still compatible.

```
public void NotifyActivity(string hostname, string action, Guid reference,
    Dictionary<string, string> parameters) {
    if (!notificationFilter.Contains(reference) || stopNotificationClient)
        return;
    wasActivityRun = true;
    if (action.StartsWith("Database."))
        ApplicationControl.DisconnectFromLocalDatabase();
    else if (action.StartsWith("Product."))
    {
        Disconnect();
        ApplicationControl.ShutdownApplication();
    }
}

public void NotifyCompletion(string hostname, string action, bool success, Guid
    reference, Dictionary<string, string> parameters) {
    if (!notificationFilter.Contains(reference) || stopNotificationClient)
        return;
    notificationFilter.Remove(reference);
    if (action == Actions.DatabaseUpgrade || action == Actions.DatabaseRestore)
    {
        string newVersionStr = (action == Actions.DatabaseUpgrade) ? parameters["
            ToVersion"] : parameters["Version"];
        var newVersion = new Version(newVersionStr);
        ApplicationControl.DisplayDatabaseVersion(newVersion);
        if (ApplicationControl.LocalDatabaseMajorVersionNumber == newVersion.Major
            && ApplicationControl.LocalDatabaseMinorVersionNumber == newVersion.
            Minor)
        {
            // schema version considered as compatible
            if (wasActivityRun)
                ApplicationControl.ConnectToLocalDatabase();
            ApplicationControl.UnlockApplication();
        }
        // else wait for application upgrade
    }
}
```

Listing 4.23: Reacting to Configuration Changes

### 4.6.2 Integration within azDeploy

azDeploy informs all vendor applications at a customer site about all configuration changes taking place at the site. Therefore, the client agent disseminates notifications about all stages of a local configuration change. This happens in the task implementation. The class *Notifier* allows the client agent to send notifications and to receive request results from each vendor application running at the site. It first contacts all applications running on the local appliance. Then it contacts the gateway agent, which notifies all other client agents on the site, which in turn inform all vendor applications running on the respective appliance. In case of a configuration change request, the gateway agent also collects all request results. After the call to the gateway agent returns, all applications are informed about the configuration change.

```
public static DatabaseTaskResult RunTask(BackupDatabaseTask task) {
    var reference = Guid.NewGuid();
    var parameters = new Dictionary<string, string>();
    parameters["Database"] = task.Database.Name;
    parameters["ConnectionStringIdentifier"] = task.Database.
        ConnectionStringIdentifier;
    var requestResult = AppContext.Notifier.Request(AppContext.Client.Host.HostName
        , Actions.DatabaseBackup, reference, parameters);
    if (requestResult.ErrorType != ErrorType.Success)
    {
        AppContext.Notifier.NotifyCompletion(AppContext.Client.Host.HostName,
            Actions.DatabaseBackup, false, reference, parameters);
        return new DatabaseTaskResult(new DatabaseOperationError() { Class = 1,
            LineNumber = 0, Message = requestResult.ToString() });
    }
    AppContext.Notifier.NotifyActivity(AppContext.Client.Host.HostName, Actions.
        DatabaseBackup, reference, parameters);
    Trace.WriteLine("Backing up database " + task.Database + ".");
    var db = new ManagedDatabase();
    db.OpenConnection(task.Database.ConnectionStringIdentifier, task.Database.Name)
        ;
    var result = db.BackupDatabase(task.BackupPoint,task.Username,task.Description)
        ;
    db.CloseConnection();
    AppContext.Notifier.NotifyCompletion(AppContext.Client.Host.HostName, Actions.
        DatabaseBackup, result.Success, reference, parameters);
    return result;
}
```

Listing 4.24: The Implementation of *BackupDatabaseTask*

**Implementation** Listing 4.24 shows the task implementation for backing-up a database. Firstly, it sets up the *reference* value and the *parameters* used by each notification. For database backups, these parameters are the database name and the connection string identifier. Then, it uses the class *Notifier* to request permission from all applications for backing-up the database.

*Notifier* returns a single request result object. If all vendor applications granted to back-up the database, the object's *ErrorType* property has the value *Success*. In contrast, if at least one vendor application denied the request, the task implementation receives the request result object of the application which denied the request first.

If any vendor application denied the request, the configuration change directly transitions into the *completion* stage: *Notifier* sends a completion notification to all vendor applications. The *success* parameter with the value *false* indicates that the configuration change failed. Then it ends executing the task by returning a database task result containing the error information.

In contrast, if all vendor applications granted the request, the configuration change transitions into the *activity* stage. In turn, *Notifier* sends an activity notification to all vendor applications. Then, the task implementation initiates the backup operation through the class *ManagedDatabase*. Once the operation finishes, the configuration change enters the *completion* stage: *Notifier* sends a completion notification to all vendor applications. Its *success* parameter carries the success information of the database task result. Finally, it finishes executing the task by returning the task result.

## 4.7 File Transfer Services

The server application uses the file transfer services to transfer SQL upgrade scripts and software installation packages to the appliances. As these services do not depend on the operation control services, transferring files does not block operations.

Within the server application, the class *FileTransferService* implements file transfers. It uses per-session instancing and the concurrency mode *reentrant*. The service provides two endpoints: The *IFileTransferAdminService* endpoint allows initiating file transfers and the *IFileTransferService* endpoint actually transfers the files.

### IFileTransferAdminService

Because the *IFileTransferAdminService* endpoint (Listing 4.25) is operated locally through *MainService*, it uses a *netNamedPipeBinding* for communication. Before using it, *MainService* must establish a session. *IFileTransferAdminService* employs the task system: To start transferring files, *MainService* uses *TaskManager* to enqueue a *TransferFilesTask*. *TaskManager* then calls the *AddTransmission* method on *IFileTransferAdminService*. A *TransferFilesTask* object has two properties:

**Target:** The appliance to which the files are transferred.

**Instructions:** This array contains objects of the type *TransferFileInstruction*.

A *TransferFileInstruction* describes a single file to be copied to the appliance's download directory and has two properties:

**SourcePath:** The absolute path of the file to transfer on the server.

**DestinationPath:** The relative path within the download directory of the appliance where to store the transferred file.

Once the transfer completes, *FileTransferService* notifies *MainService* via the *NotifyLocalEvent* method.

```
[ServiceContract(SessionMode = SessionMode.Required, CallbackContract = typeof(
    IFileTransferAdminCallback))]
public interface IFileTransferAdminService {
    [OperationContract(IsInitiating=true,IsTerminating=false)]
    void LogOnAdmin();
    [OperationContract(IsOneWay=true)]
    void AddTransferTask(TransferFilesTask task);
    [OperationContract(IsInitiating=false,IsTerminating=true)]
    void LogOffAdmin();
}

public interface IFileTransferAdminCallback {
    [OperationContract(IsOneWay = true)]
    void NotifyLocalEvent(RemoteEvent ev);
}
```

Listing 4.25: Service Contract for the *FileTransferAdminService* Endpoint

### IFileTransferService

Unlike *IControlService*, where whole sites log on via their gateway agents, each client agent logs on “directly” to *IFileTransferService* (Listing 4.26): The gateway agent hosts the *FileTransferProxy* service. This service offers an *IFileTransferService* endpoint to the site's client agents. As its name suggests, *FileTransferProxy* simply forwards all service method calls to *FileTransferService* hosted by the server application. *FileTransferProxy* also implements the *IFileTransferCallback* contract (Listing 4.26). Likewise, it forwards all callback calls to the corresponding client agent. To make this transparent proxy behavior work, *FileTransferProxy* uses per-session instancing with reentrant concurrency. Therefore, there exist as many *FileTransferProxy* instances as connected client agents.

```
[ServiceContract(SessionMode=SessionMode.Required,CallbackContract=typeof(
    IFileTransferCallback))]
public interface IFileTransferService {
    [OperationContract(IsInitiating=true,IsTerminating=false)]
    void LogOn(Host host);
    [OperationContract]
    FileTransferInfo RetrieveNextFileInfo();
    [OperationContract]
    byte[] RetrieveChunk();
    [OperationContract(IsInitiating=false,IsTerminating=true)]
    void LogOff();
}

public interface IFileTransferCallback {
    [OperationContract(IsOneWay = true)]
    void NotifyFilesAdded();
}
```

Listing 4.26: Service Contract for the *FileTransferService* Endpoint

How do the file transfer services differ from the agent upgrade services? Firstly, they employ a push rather than a pull model: Whereas within the agent upgrade services the agents initiate the file transfer, within the file transfer services the server application initiates the file transfer. Secondly, they integrate into the task system. Thirdly, they allow to transfer an arbitrary number of files to different locations. Finally, they maintain state and therefore employ sessions.

**Protocol** After a client agent has logged on to the file transfer services, a file transfer works as follows: *MainService* adds a new file transfer task. Next, *FileTransferService* notifies the client agent that it wants to transfer files by calling the method *NotifyFilesAdded* on the callback channel. The client agent then uses the method *RetrieveNextFileInfo* to retrieve information about the file to be transferred. It returns an object of type *FileTransferInfo* which has three properties:

**Path:** The path where to store the transferred file within the client agent's download directory.

**Filesize:** The filesize of the transferred file.

**Checksum:** A string containing the MD5 hash of the transferred file in hexadecimal representation.

After that, the client agent repeatedly calls the method *RetrieveChunk* to obtain the file's data: Per call, the server transfers a chunk of data. Once the whole file is transmitted, the method returns null. Finally, the client agent calls the method *RetrieveNextFileInfo* again to obtain information about the next file transfer. When there are no more enqueued file transfers, this method returns null.

**Limitations** The client agent does not need to know about distinct file transfer tasks. It just assumes transfers of independent files. As a consequence, to avoid file name conflicts, each transfer task should put its files into a separate directory: This applies if a client agent applies the same database schema upgrade script to several databases at the same time. As these upgrade tasks are independent, they do not know that the required upgrade script was already transferred nor that other tasks will need it again. Therefore, the upgrade script is transferred for each database and will be deleted after the database schema was upgraded.

Furthermore, the current implementation does not cache the transferred files. Thus, when sending the same file to several appliances of the same site, the server has to send it multiple times. As files can be identified by their size and their checksum, caching can be realized with a small effort.

# 5 Project Structure and Applications

This chapter describes the project structure and the characteristics of the server application, the administration center, the gateway agent, and the client agent. The application starter and the package tool will be discussed in Section 6.2.

## 5.1 Project Structure

azDeploy consists of the assemblies listed in Table 5.1. Table 5.2 shows some software metrics for azDeploy. The applications are structured as follows:

**Server Application** The server application consists of two assemblies: *ServerServiceLibrary* contains all WCF service classes whereas the assembly *Server* merely hosts and configures the WCF services. This allows converting the server application to a Windows service by replacing the assembly *Server* with a Windows service project.

**Windows Services** Gateway and client agents can already run as Windows services. For development purposes, they can run as standalone applications. Therefore, the assemblies *Gateway* and *Client* already employ a Windows service like structure. The respective Windows service projects merely pass on the service control calls.

**Shared Assemblies** The assembly *Core* contains the functionality all applications need. The assembly *Repository* enables the server application and the client agent to read metadata from Windows Installer packages.

**Shared Contracts** Since the applications of azDeploy communicate as a closed system, they share the service and data contracts via separate assemblies. This reduces development and refactoring effort as developers do not need to generate proxy classes from service metadata if contracts change. The assembly *Common.Contracts* contains the contracts needed by all applications. All other assemblies ending in *Contracts* contain contracts used only between some applications. For instance, the contracts within *Gateway.Contracts* are used only for customer site communication between the gateway agent and the client agent.

**Extensibility** Plug-ins for the client agent use the assembly *ExtensionBase*. Applications interacting with azDeploy via the notification services use *Client.Contracts*.

| Assembly | Description |
|---|---|
| AdministrationCenter | The administration center application. |
| AgentUpgradeClient | Used to upgrade the agent applications itself. |
| ApplicationStarter | The application starter application. |
| Client | The client agent executable. |
| Client.Contracts | Data contracts used by the agents and applications interacting with azDeploy. |
| ClientService | The windows service which runs the client agent. |
| Common.Contracts | Data contracts used by the server application, the gateway agent, and the client agent. |
| Core | Classes shared by the server application, the gateway agent, and the client agent. |
| ExtensionBase | Used by plug-ins for the client agent. |
| Gateway | The gateway agent executable. |
| Gateway.Contracts | Data Contracts which are used between the agents. |
| GatewayService | The windows service which runs the gateway agent. |
| JornadaClient | The demo application interacting with azDeploy. |
| PackageTool | The package tool executable. |
| Repository | Classes shared between the client agent and the server application. |
| Server | The server application executable. |
| Server.Administration.Contracts | Data Contracts which are used between the administration center and the server application |
| Server.Control.Contracts | Data Contracts which are used between the server application and the gateway agent. |
| ServerDatabase | Provides access to the server database. |
| ServerServiceLibrary | The server services. |
| XMLConnectionStringProvider | Plug-in for the client agent which reads connection strings from a XML file. |

Table 5.1: Assemblies

| Metric | Total[a] | Contracts | Administration Ctr. | Server | Gateway | Client |
|---|---|---|---|---|---|---|
| Effective LoC[b] | 10500 | 1200 | 2700 | 1400 | 500 | 1500 |
| Number of Classes | 260 | 96 | 57 | 43 | 13 | 22 |

[a]Includes non-listed applications and shared assemblies, but excludes XAML markup
[b]Lines of Code

Table 5.2: Software Metrics

## 5.2 Server Application

The server application is the central component of AZDEPLOY. It links the administration center instances with the customer sites. A single instance of the server application runs at the vendor site. The TCP port of the *IControlService* endpoint must be accessible from the Internet.

The server application provides:

- Administration of all customer sites via the *IAdminService* endpoint
- A central log on point for all customer sites
- The agent upgrade service to keep the gateway and client agents up to date
- Access to the software and database schema upgrade script repositories
- Access to the database upgrade and product installation history
- Management of stored database views
- Logging of all configuration changes

The fundamental classes of the server application are:

**AgentUpgradeService:** The agent upgrade service implementation. It exposes the *IAgentUpgradeService* endpoint.

**FileTransferService:** The file transfer service implementation. Besides the *IFileTransferService* endpoint, it also exposes the *IFileTransferAdminService* endpoint through which *MainService* can initiate file transfers.

**MainService:** The core service of the server application. It provides both the endpoints *IAdminService* and *IControlService*.

**AdminSessionManager:** Since the underlying binding configuration of the *IAdminService* endpoint does not support WCF sessions (see Section 5.4), this class maps service calls to custom sessions. It does so by using the username of the client credentials which the administration center sends with each call as session identifier. This makes it possible to distinguish between the administration center users without supplying an additional session parameter in each service method. However, this approach does not allow one user to use multiple instances of the administration center in parallel.

**WCF Behavior Classes:** These classes intercept the service calls to *MainService* (see Section 6.3 on page 66).

**Repository Classes:** *QueryRepository*, *ScriptRepository* and *SoftwareRepository* provide access to the stored database views, the database schema upgrade scripts, and the software packages.

**Task Control Flow Classes:** They were discussed in Section 4.5.2 on page 31.

**ServerTaskProcessorService:** This service implementation exposes the endpoint *IServerTaskProcessorService*. It processes log tasks and tasks querying the database and product installation history.

**TaskManager:** This class schedules the tasks of the task system (see Section 4.5.2 on page 31).

**ServerDatabaseDataContext:** This class provides access to the server database.

Figure 5.1 shows all WCF Services that make up the server application.

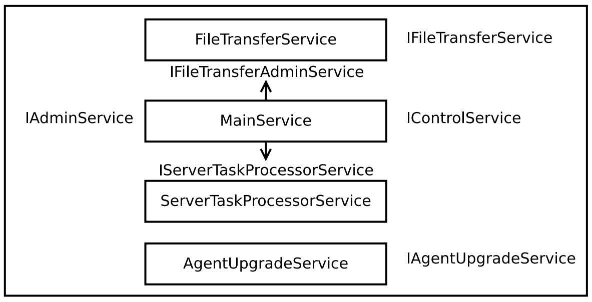


Figure 5.1: Services hosted by the Server Application

## 5.3 Gateway and Client Agents

The gateway agent acts as a proxy and links the client agents of a customer site with the server application. A single instance of the gateway agent runs at each customer site. It must have Internet access.

The gateway agent provides:

- Proxy functionality for the agent upgrade, control, and file transfer services. It is implemented in the classes *AgentUpgradeProxy*, *GatewayService* and *FileTransferProxy*.
- Task implementations to receive all databases and database backups of a site. They are implemented in the class *TaskImplementations*.
- The notification service for the site which interchanges notifications between the individual client agents. It is implemented in the class *NotificationService*.

The client agent conducts all configuration changes on an appliance and provides:

- Task implementations of all configuration changes
- The notification service for all applications interacting with azDeploy

- Plug-in support to add connection strings

The fundamental classes of the client agent are:

**CClient:** The client class which implements the callback contract *IHostCallback* required by *IGatewayService*.

**FileTransferClient:** The client class which receives files sent by the server application. It implements the callback contract *IFileTransferCallback* required by *IFileTransferService*.

**NotificationService:** The notification service implementation. It exposes the local *INotificationService* endpoint to which the vendor applications connect.

**Notifier:** This class allows the client agent to send notifications to all vendor applications of the site. Firstly, it contacts the local notification service, then the gateway agent's notification service which disseminates the notification throughout the site.

**PluginManager:** This class automatically loads all plug-ins from assemblies.

**TaskImplementations:** This class handles all tasks conducted on the appliance. It links the actual operations implemented in other classes with the notification services.

**BackupArchive:** This class can compress database backups as zip archives. It also stores metadata about the backups within the archives.

**ManagedDatabase:** This class contains the implementations of the database operations.

**MsiInstallerHelper:** This class allows to install, upgrade, and uninstall software.

## 5.4 Administration Center

The administration center is a XAML browser application (short XBAP) which allows the operators to control all operations of azDeploy. Such applications use Windows Presentation Foundation (short WPF) to implement their user interface. The operators must be able to use the administration center also from their homes over the Internet. Thus, the administration center implements security by encrypting all communication.

### Security

The .NET framework enforces a concept called code access security (short CAS). CAS ensures that the code can only conduct activities for which it has the required security permissions. CAS distinguishes two fundamental levels of trust: *Full trust*, if the code has all permissions. In contrast, if the code has only a limited set of permissions, CAS refers to it as *partial trust* [16].

**Permissions** The administration center XBAP specifies its security permissions through a ClickOnce application manifest. It is set up to use the partial trust security zone *Internet*. A security zone defines a set of permissions granted to an application. Additionally, the administration center requires the security permission *WebPermission* which is not available in the security zone *Internet*. Therefore, it requests the *WebPermission* in its ClickOnce application manifest. To actually obtain the additional security permission, the manifest must be signed with a certificate. This is because a XBAP from the web is granted only Internet zone permissions by default [15].

The administration center needs the additional *WebPermission* since WPF applications have only site-of-origin access [14]. However, in the azDeploy scenario the administration service endpoint the administration center connects to does not match the administration center's URL as it runs on a different port than the *Internet Information Services* (short IIS) web server providing the administration center XBAP.

This is the result of two restrictions: Firstly, it is possible to host a WCF service within IIS, thus allowing it to use the IIS port. However, WCF services hosted within IIS can only use HTTP bindings [17]. Therefore, this is not an option as *MainService* also exposes the control service endpoint which uses a *netTcpBinding*. Secondly, it is also not possible for the server application and IIS to use port sharing under Windows XP [13].

**WCF Constraints** CAS employs security demands: This means that all callers on the call stack must have sufficient permissions to run a certain operation. Most functionality of WCF requires the full trust permission set. Therefore, as the administration center executes under a partial trust permission set, the administration center can only choose between a few binding configurations which also support partial trust callers. To secure communication, these binding configurations can only opt for transport security mode [16]. Securing communication will be discussed in Section 6.4 on page 70.

## Binding Selection

Due to these security constraints, the possible bindings for the administration center are *BasicHTTPBinding*, *WebHTTPBinding*, and *WSHTTPBinding* [16] None of them supports duplex operations which implies that the server application cannot use a callback contract to contact the administration center. The administration center opts for *WSHTTPBinding* as it has the most features including session support [4]. However, session support for HTTP bindings requires *reliable messaging*, which partial trust callers cannot use [16, 18]. Therefore, the administration center cannot employ sessions. Instead, the server application and the administration center use a custom session replacement as described on page 52.

## Event Processing

Due to the lack of duplex messaging, the administration center polls *MainService* periodically for new remote events. The administration center uses the class *RemoteEvent-*

```
class RemoteEventDispatcher {
    public static event EventHandler<RemoteEventArgs<HostConnectivityChangedEvent>>
         HostConnectivityChanged;
    public static event EventHandler<RemoteEventArgs<SiteConnectivityChangedEvent>>
         SiteConnectivityChanged;
    public static event EventHandler<RemoteEventArgs<TaskFailedEvent>> TaskFailed;
    public static event EventHandler<TaskCompletedEventArgs<DatabaseTaskResult>>
        DatabaseTaskCompleted;
    public static event EventHandler<TaskCompletedEventArgs<
        ChangeSoftwareTaskResult>> ChangeSoftwareTaskCompleted;
    ...
}
```

Listing 5.1: The RemoteEventDispatcher Class

*Dispatcher* to allow an event-driven programming style in the rest of the application. *RemoteEventDispatcher* periodically queries *MainService* for new remote events. For each remote event, it triggers a local event with event arguments adapted to the event's type.

Listing 5.1 shows some event handlers of *RemoteEventDispatcher*. It includes an event handler for each remote event type and an event handler for each task result type. To wrap the different task result types, it uses the generic *TaskCompletedEventArgs* class. To query for remote events periodically, it uses the *DispatcherTimer* class.

**Threading** The *DispatcherTimer* runs within the WPF UI thread. This is important because WPF UI objects can only be accessed from the UI thread. Thus, the UI classes of the administration center can react to remote events like WPF UI events and need not to consider threading issues [19, 20].

# 6 Implementation Details

This chapter discusses noteworthy implementation details. It gives information about database management (Section 6.1), application management (Section 6.2), WCF Custom Behaviors (Section 6.3), and WCF Security (Section 6.4).

## 6.1 Database Management

azDeploy applies database schema upgrades only to initialized databases. Initializing a database means assigning a database schema and database schema version to a database. The client agent does this by adding the metadata table *SYS_DBProperties* to the database.

*SYS_DBProperties* stores properties as key and value pairs. It contains three properties:

**DatabaseSchema:** The name of the database schema.

**DatabaseSchemaVersion:** The version of the database schema. Each time the client agent applies a database schema upgrade, it updates this property.

**MetadataVersion:** The client agent uses this property to check whether it understands all database metadata. This value only increases with a new release of the client agent which changes the metadata table structure or adds new properties. Future releases then must upgrade the metadata structure to their supported version before working with the database.

Both *DatabaseSchemaVersion* and *MetadataVersion* employ the four-number .NET version format (see [1]). Table 6.1 shows the contents of *SYS_DBProperties* for a Jornada client local database with database schema version 1.2.

| Name | Value |
|---|---|
| DatabaseSchema | JornadaClient |
| DatabaseSchemaVersion | 1.2.0.0 |
| MetadataVersion | 1.0.0.0 |

Table 6.1: The *SYS_DBProperties* Metadata Table

In contrast, azDeploy stores the history of database configuration changes in the table *DatabaseChangeLog* of the server database. This separation has several advantages:

- Information concerning the upgrade that should not be disclosed to the customer administrators is stored in the server database.
- Higher performance: The client agent never queries the history of database configuration changes. Instead, the server application often queries it.
- Whenever azDeploy restores a backup to a database, it only needs to record the restore operation in the server database. The restored backup already carries the database schema version in *SYS_DBProperties*.

## 6.2 Application Management

azDeploy uses Microsoft Windows Installer to install, uninstall, and upgrade applications. Section 6.2.1 explains the basic concepts of Windows Installer. Section 6.2.2 shows how azDeploy integrates Windows Installer.

### 6.2.1 Windows Installer

This section introduces the features of Windows Installer, the structure of an installation package, and the installation process.

#### Windows Installer Features

Windows Installer is a software installation service that comes with Microsoft Windows. As it supports transactions, it can rollback an installation if it fails. A software product can be installed either for a specific user or for all users on a machine (*installation context*). Application developers can provide upgrades of their product as full installation packages or as a patch. Within an installation, a user can select which *features* of an application s/he wants to install, resulting in a set of components that Windows Installer installs [28–30].

azDeploy can only deploy applications which comply with several restrictions:

**Package Format:** azDeploy only supports single-file full installation packages.

**Installation Context:** As azDeploy manages applications on a per-machine basis, installation packages must install within the per-machine installation context [31].

**Features:** azDeploy manages applications as a whole. Thus, installation packages must have exactly one feature referring to all components of the application [30].

**Configuration:** Developers must author installation packages which remove previous installed versions of the product prior to upgrading. Also, the package should refuse to install if a newer product version is already installed.

To illustrate the concepts of Windows Installer, the remainder of this section refers to the Jornada Client 1.0 installation package.

### Installation Package

An installation package contains an *installer database*. This is a relational database consisting of tables which steer the setup process. Columns can also be primary keys or secondary keys which reference a row in another table [32].

azDeploy compatible installation packages employ several important installer database tables [33]:

**Component:** It contains the application components. Microsoft Visual Studio 2008 authored packages treat each .NET assembly as a separate component.

**CustomAction:** It contains custom actions which allow developers to integrate custom steps into the setup process [34].

**File:** It contains all files which can be installed and assigns each file to a component by referencing the *Component* table [35].

**InstallExecuteSequence:** It controls the setup process [36].

**Property:** It contains property names and values (Table 6.2).

**Upgrade:** Windows Installer uses this table to upgrade an installed product [37].

Windows installer distinguishes application releases by their *product code* (GUID) and by their *product version*. Both properties are stored in the *Property* table [29, 40].

An installation package also contains a *summary information stream* that stores several properties about the package. Among them, the *package code* uniquely identifies an installation package [38, 39].

| Property | Value |
|---|---|
| UpgradeCode | {66BBB50A-5DAA-4E1B-BBD2-936B2F9C720F} |
| ProductName | Jornada Client |
| ProductCode | {5A8950DD-38AB-4033-B746-8BAD1B6F97EF} |
| ProductVersion | 1.0.0 |
| Manufacturer | Rainer Pichler |
| ProductLanguage | 1033 |
| VSDVERSIONMSG | Unable to install because a newer version of this product is already installed. |

Table 6.2: *Property* Table Content (shortened)

## Installation Process

The setup process is table-driven. *Sequence tables* determine Windows Installer's actions and their order. They have three columns [41]:

**Action:** The name of the (custom) action to execute.

**Condition:** A condition determining whether Windows Installer executes the action. An empty field equals *true*.

**Sequence:** A number determining the execution order of the action entries.

Depending on the installation mode, Windows Installer uses a different set of sequence tables. Because AZDEPLOY uses the *Simple Installation* mode and disables the Windows Installer UI, it only processes the *InstallExecuteSequence* table [41, 42].

Table 6.3 shows several basic action entries from the *InstallExecuteSequence* table of the JORNADA Client 1.0 installation package authored with Microsoft Visual Studio 2008 and the package tool. The actual table contains another 60 actions which execute between the listed actions. The actions are already ordered by the *Sequence* column and thus in the execution order.

How does installing, upgrading or removing a product work? The top-level action *INSTALL* triggers the processing of the *InstallExecuteSequence* table. Windows Installer executes *INSTALL* for all installation tasks. Thus, Windows Installer processes the *InstallExecuteSequence* table also when removing an installed product due to an uninstallation or an upgrade [43].

The following paragraphs explain the fundamental actions Windows Installer executes when processing the *InstallExecuteSequence* table of the JORNADA Client 1.0 installation package.

| Action | Condition | Sequence |
|---|---|---|
| FindRelatedProducts | | 200 |
| ERRCA_CANCELNEWERVERSION | NEWERPRODUCTFOUND AND NOT Installed | 201 |
| InstallInitialize | | 1500 |
| ProcessComponents | | 1600 |
| RemoveFiles | | 3500 |
| InstallFiles | | 4000 |
| RegisterProduct | | 6100 |
| InstallExecute | | 6500 |
| RemoveExistingProducts | | 6550 |
| InstallFinalize | | 6600 |
| LaunchExecutable | NOT Installed | 7600 |

Table 6.3: *InstallExecuteSequence* Table Content (shortened)

**1. Identifying Affected Products** Only when installing a product, Windows Installer executes the action *FindRelatedProducts*. In this context, the term installing also includes upgrading an installed product. *FindRelatedProducts* uses the *Upgrade* table (Table 6.4) to find out which installed products the installation affects [44].

*FindRelatedProducts* iterates through each row and evaluates whether an installed product matches the row: This is the case whenever both the *upgrade code* and *product language* match and the *product version* lies within the specified version range. If the value of the field *Language* is null, all languages match. The fields *VersionMin* and *VersionMax* define the version range. Per default, their values do not lie within the range. The value of the field *Attributes* determines how Windows Installer interprets the columns *VersionMin*, *VersionMax* and *Language* as well as what it does with the detected products. Whenever an installed product matches a row, Windows Installer appends its product code to the property specified in the column *ActionProperty*. Also, it appends all features specified in the column *Remove* to the property *REMOVE*. In azDeploy compatible packages, this field is always null meaning that Windows Installer assigns the value *ALL* to the property *REMOVE* [37].

In the Jornada Client installation package, the *Upgrade* table contains two rows which ignore the product language of installed products. The first row matches all products with the stated upgrade code having a product version below 1.0.0. The value 0 of the *Attributes* field implies that Windows Installer will remove the matching products in a later step. Finally, Windows Installer appends their product codes to the property *PREVIOUSVERSIONSINSTALLED* [37].

In contrast, the second row matches all products with the stated upgrade code having a product version equal to or higher than 1.0.0. The value 258 of the *Attributes* field defines both that the value of *VersionMin* is part of the version range and that Windows Installer will not remove the detected products. Windows Installer appends the product codes of the matching products to the *NEWERPRODUCTFOUND* property [37].

In both cases, the value of the property *REMOVE* is set to *ALL* [37].

| UpgradeCode | VersionMin | VersionMax | Attributes | ActionProperty |
|---|---|---|---|---|
| {66BBB5...} | | 1.0.0 | 0 | PREVIOUSVERSIONSINSTALLED |
| {66BBB5...} | 1.0.0 | | 258 | NEWERPRODUCTFOUND |

Table 6.4: *Upgrade* Table Content (empty columns *Language* and *Remove* omitted)

**2. Preventing Downgrades** If the property *NEWERPRODUCTFOUND* is set and the product is not already installed, Windows Installer executes the custom action *ERRCA_CANCELNEWERVERSION*. This custom action is defined in the table *CustomAction* (Table 6.5). It displays an error message and cancels the installation [34,45,46]. Note that the previous action *FindRelatedProducts* probably set the property *NEWERPRODUCTFOUND* through appending product codes to it. Hence, Windows Installer refuses the installation of the package if a newer version of the product is already installed.

**3. Starting the Installation** The action *InstallInitialize* starts the actual setup process by beginning a transaction [48]. *ProcessComponents* updates the component configuration [49]. *RemoveFiles* instructs Windows Installer to remove files belonging to the removed components. It uses the *File* table to determine the files belonging to a component [35, 50]. For azDeploy compatible packages, this action will always and only remove all application files if an installed product is being removed.

*InstallFiles* installs all files belonging to the added components [35, 51]. Again, for azDeploy compatible packages, this action will always and only install all application files if a product is being installed.

*RegisterProduct* marks the product as installed and stores the installer database on the appliance [52]. Hence, Windows Installer can cleanly remove the product installation at a later point in time.

*InstallExecute* then forces Windows Installer to actually execute all operations since *InstallInitialize* [48].

**4. Cleaning Up** *RemoveExistingProducts* removes the products which *FindRelatedProducts* determined to be uninstalled. Like *FindRelatedProducts*, Windows Installer also only runs *RemoveExistingProducts* when installing a product. To remove the products, it starts *concurrent installations*. Hence, Windows Installer removes no longer needed files of old product installations after the installation of the new package [53].

**5. Completing the Installation** *InstallFinalize* forces Windows Installer to actually execute all operations since *InstallExecute*. Thus, to run the operations of *RemoveExistingProducts*. Also, *InstallFinalize* ends the installation transaction [54].

Finally, in case the product was installed, the Windows Installer runs the custom action *LaunchExecutable*. The next paragraphs explain its behavior and the concept of custom actions.

### Custom Actions

*Custom actions* integrate custom steps into the installation process. The table *CustomAction* (Table 6.5) contains all custom actions. Its essential columns are [34]:

**Action:** The name sequence tables use to refer to the custom action.

**Type:** This column indicates the underlying custom action type and the flags specifying its behavior.

**Source and Target:** These columns store the parameters of the custom action.

**Launch Executable** The package tool inserts the custom action *LaunchExecutable* which starts an installed executable as the last step within the setup. Therefore, Windows Installer does not end the installation process unless the executable terminates [55]. Thus, developers can integrate an executable which runs exactly once after installing the package. This executable could for instance adapt configuration files before launching the actual application after setup.

When inserting *LaunchExecutable* into the *CustomAction* table (Table 6.5), the package tool sets the *Type* field to 18, making it a custom action that launches an executable installed by the package. To refer to the executable, it sets the *Source* field to a key referencing an executable file in the file table [55].

Although the *Type* field value could be altered in a way that Windows Installer ends the installation after launching the referenced executable, azDeploy does not use *LaunchExecutable* to start the application after its installation [56]. This is because azDeploy runs under a privileged service account session and invokes Windows Installer within this session too. But as the application must be launched within the active desktop session, azDeploy takes another approach to start the application (see Section 6.2.2).

| Action | Type | Source | Target |
|---|---|---|---|
| ERRCA_CANCELNEWERVERSION | 19 | | [VSDVERSIONMSG] |
| LaunchExecutable | 18 | _02ADD... | |

Table 6.5: *CustomAction* Table Content (shortened)

## Update Types

Windows Installer distinguishes three different update types. The update type depends on whether the *product code*, the *product version* or both properties of an installed product change when installing an upgrade. Each update type implies certain installation methods [29].

Since azDeploy only supports full installation packages, this results in either a reinstallation or a simple installation of the package. azDeploy treats the update types *Small Update* and *Minor Upgrade* the same way, which is reinstalling the package (Table 6.6).

However, for *reinstallation*, Windows Installer seems to access the original installation package of the installed product too. This is not possible as azDeploy removes the installation package after the setup. Therefore, to apply a small update or a minor upgrade, azDeploy first uninstalls the installed product and in turn installs the new package. This implies that Windows Installer cannot roll back to the previous product installation, when it already uninstalled it, but installing the new package

| Update Type | Product Code | Product Version | Suggested Method |
|---|---|---|---|
| Small Update | same | same | Reinstallation |
| Minor Upgrade | same | changed | Reinstallation |
| Major Upgrade | changed | changed | Installation |

Table 6.6: Windows Installer Update Types (adapted from [29])

failed. To circumvent this deficit, AZDEPLOY resorts to the *InstallProductControlFlow* (see Section 4.5.2 on page 31): In case Windows Installer could not install the new package, *InstallProductControlFlow* installs the previously uninstalled product release again. For this to work, installation packages using the *Small Update* or *Minor Upgrade* update types should not remove their configuration files when being uninstalled.

### 6.2.2 Integrating Windows Installer within azDeploy

To interact with Windows Installer, AZDEPLOY uses the automation interface which exposes an *Installer* object via COM [57].

**Accessing Package and Product Metadata** AZDEPLOY uses the Installer object to access the installer database and the summary information stream (Listing 6.1). It queries and updates the installer database via SQL statements. To access the relevant fields of the summary information stream, it supplies the according property IDs (PIDs) [38, 58–61].

```
private const int PID_TITLE = 2; // PID for PackageTitle
private const int PID_REVNUMBER = 9; // PID for PackageCode

public static ProductPackageInfo GetProductPackageInfo(string fileName) {
    var packageInfo = new ProductPackageInfo();
    var sum = Installer.get_SummaryInformation(fileName, 0);

    packageInfo.PackageCode = new Guid((string)sum.get_Property(PID_REVNUMBER));
    packageInfo.PackageTitle = (string)sum.get_Property(PID_TITLE);

    WindowsInstaller.Database db = Installer.OpenDatabase(fileName,
        MsiOpenDatabaseMode.msiOpenDatabaseModeReadOnly);

    var view = db.OpenView("SELECT Property, Value FROM Property");
    view.Execute(null);

    var propertyName = string.Empty;
    try {
        while (true) {
            var record = view.Fetch();
            if (record != null) {
                propertyName = record.get_StringData(1);
                var value = record.get_StringData(2);
                ... // omitted: store value in respective packageInfo property
            }
            else break;
        }
    }
    catch (Exception ex) { ... }
    finally {
        view.Close();
        Marshal.FinalReleaseComObject(view);
        Marshal.FinalReleaseComObject(db);
    }
    return packageInfo;
}
```

Listing 6.1: Reading Windows Installer Package Metadata

Furthermore, some methods like *OpenDatabase* return handles which should be closed via the *MsiCloseHandle* function. However, the affected objects do not expose their handles nor does *MsiCloseHandle* exist on the .NET Installer object. Instead, azDeploy uses *Marshal.FinalReleaseComObject* to free these objects [59, 62, 63]. Omitting this step led to problems on subsequent calls.

Additionally, the Installer object provides methods to fetch the list of installed products and to access their properties [59].

**Installing, Upgrading and Uninstalling Products** Windows Installer uses the *Install* action both for installing and upgrading products. Furthermore, for reasons discussed on page 63, azDeploy replaces the installation method *reinstallation* by an uninstallation with subsequent installation.

Thus, azDeploy uses `Installer.InstallProduct(filename, "ACTION=INSTALL")` to install a product and `Installer.ConfigureProduct(productCode, 0, MsiInstallState.msiInstallStateAbsent)` to uninstall a product entirely [64, 65].

**Launching Applications** azDeploy launches an installed application within the active desktop session as follows: When preparing the installation package, the package tool adds an extra property table called *AZProperty*, which holds property and value pairs, to the package. It inserts the property *LaunchPath* containing the path to the executable of the application. When installing the package, the notification services export all properties of this table as notification parameters with the prefix *"AZProperty."*. Also, the helper application *application starter* runs in the background of the active desktop session and listens for completed application installations. Whenever there exists the parameter *AZProperty.LaunchPath*, it runs the referenced executable. Hence, applications only start automatically when they are installed via azDeploy.

## 6.3 Windows Communication Foundation Custom Behaviors

This section discusses how azDeploy uses WCF custom behaviors to separate the code for cross-cutting concerns from the operation specific code.

### Rationale

The server application enforces two general rules:

- The administration center instances and the gateway agents cannot supply non-null parameter values to service methods unless specified otherwise.
- An operator cannot administer a customer site administered by another operator.

Both rules have in common that they do not only apply to a specific operation like upgrading a database schema, but are a cross-cutting concern. However, to enforce them nevertheless, the server application would need to take care of them in each service method implementation. This would lead to code duplication and add code to service method implementations that is unrelated to the method's actual task.

Furthermore, both rules actually do not need to know the details about the operations they apply to: The first rule only needs to check whether a supplied parameter is *null*. Hence, it does neither need to know the parameter's type nor its purpose. It must only be told whether a parameter can be *null*. The second rule must know whether it has to check if a specific operator can administer a specific customer site. To do so, it only needs to know the operator and the site to administer.

Two steps are necessary to separate these two general rules from the service method implementations:

1. Implementing each rule in an operation independent form.
2. Linking these rules with the actual service method implementations.

### Custom Behaviors

To do so, the server application takes advantage of WCF *custom behaviors*. What are these?

Clients send WCF messages to a WCF service in order to invoke service methods. When a WCF message arrives, the WCF *dispatcher* component calls the corresponding service method. Developers can hook into this process at various points, for instance message deserialization or parameter inspection. To do so, they implement *custom behaviors* and *custom extensions*. Analogous, developers also can hook into the reverse process at the client side, when the WCF *proxy* component transforms method calls into WCF messages [27].

**Custom Extensions** *Custom extensions* implement the first step of separation. They actually contain the logic of the respective rule. Both rules hook into WCF in the parameter inspection stage to gain access to the parameters of interest when the WCF dispatcher calls a service method [27].

To do so, they implement the interface *IParameterInspector* (Listing 6.2). The WCF dispatcher calls the *BeforeCall* method of such parameter inspectors before invoking a service method and the *AfterCall* method afterwards. Both parameter inspectors of the server application are only interested in the *BeforeCall* method and therefore leave the *AfterCall* method implementation empty. In the *BeforeCall* method, parameter inspectors can access the service method's name and all parameters. To cancel the service method execution, they throw a fault exception [27].

Listing 6.3 shows the parameter inspector implementation ensuring that callers provide values for all non-nullable parameters. It is configured through its constructor which takes an array indicating the nullable parameter positions and thus, considers all positions not contained in the array to be non-nullable.

Listing 6.4 shows the parameter inspector implementation ensuring that only one operator can administer a site at a time. It fetches the operator's session via the class *AdminSessionManager*, and the site. The parameter inspector can also access the WCF operation context. The parameter position containing the site is configured through the constructor. Because the site's name may also be stored within a *Host* or *Database* object, the method *GetSite* extracts it from these types.

```
public interface IParameterInspector {
    void AfterCall(string operationName, object[] outputs, object returnValue,
        object correlationState);
    object BeforeCall(string operationName, object[] inputs);
}
```

Listing 6.2: *IParameterInspector* Interface

```
public class CheckNullableParameterInspector : IParameterInspector {
    private List<int> nullablePositions = new List<int>();

    public CheckNullableParameterInspector(params int[] nullablePositions) { ... }

    public object BeforeCall(string operationName, object[] inputs) {
        for (int i = 0; i < inputs.Length;i++ )
            if (inputs[i] == null && !nullablePositions.Contains(i))
                throw new FaultException(string.Format("Non-nullable parameter {0}
                    of operation '{1}' is null.",i,operationName));
            return null;
    }
    ...
}
```

Listing 6.3: *CheckNullableParameterInspector*

```
public class SiteLockParameterInspector : IParameterInspector {
    private int index;

    public SiteLockParameterInspector(int index) { this.index = index }

    public object BeforeCall(string operationName, object[] inputs) {
        try
        {
            if (index >= inputs.Length || inputs.Length == 0)
                throw new Exception("Invalid index.");
            var session = AdminSessionManager.GetUserSession(OperationContext.
                Current.ServiceSecurityContext.PrimaryIdentity.Name);
            var site = GetSite(inputs[index]);
            AdminSessionManager.CheckLock(session, site);
        }
        catch (Exception ex)
        {
            if (ex is FaultException)
                throw ex;
            else throw new FaultException("Error when locking site: " + ex.Message)
                ;
        }
        return null;
    }

    // extracts the site from siteName, Host object or Database object
    private string GetSite(object obj) { ... }
    ...
}
```

Listing 6.4: *SiteLockParameterInspector*

**Custom Behaviors** To implement the second step of separation, that is linking the parameter inspectors to the service methods, the server application uses *custom behaviors*. A custom behavior is a class that adds custom extensions to the WCF runtime. Custom behaviors have different scopes: They can affect a whole service, an endpoint, a service contract or a single service operation. The server application employs custom behaviors for all these scopes. Custom behavior classes implement the behavior interface corresponding to their scope. All these interfaces contain several method declarations, but the custom behaviors of the server application only use the method *ApplyDispatchBehavior* and leave the others empty. Depending on the scope, this method provides different parameters to access the WCF metadata for the corresponding element type (for instance a service contract or a service operation) [27].

The server application uses both an operation-scoped and a service-contract-scoped custom behavior to add the custom extension *SiteLockParameterInspector*. Listing 6.5 shows the operation-scoped custom behavior which adds the custom extension to each operation the method is called for unless the *DisableSiteLock* flag is set. Its properties *SiteParameterIndex* and *DisableSiteLock* are specified through the constructor. Listing 6.6 shows the service-contract-scoped custom behavior which adds the custom extension to all operations of a service contract. The contract it applies to must be specified externally. It only adds a new *SiteLockOperationBehavior* instance to an operation as long as such an operation behavior type has not been added already. Two similar custom behaviors exist for *CheckNullableParameterInspector*. But instead of the service-contract-scoped custom behavior, there exists a service-scoped custom behavior. This is because the server application wants to apply null-value checking to all service methods independent of the service endpoint. In contrast, it wants to add *SiteLockParameterInspector* only to the methods of the administration endpoint.

```
[AttributeUsage(AttributeTargets.Method)]
public class SiteLockOperationBehavior : Attribute, IOperationBehavior {
    public int SiteParameterIndex { get; private set; }
    public bool DisableSiteLock { get; set; }

    public SiteLockOperationBehavior(int siteParameterIndex) { ... }
    public SiteLockOperationBehavior(bool disableSiteLock) { ... }

    public void ApplyDispatchBehavior(OperationDescription operationDescription,
        DispatchOperation dispatchOperation) {
        if (!DisableSiteLock)
            dispatchOperation.ParameterInspectors.Add(new
                SiteLockParameterInspector(SiteParameterIndex));
    }
    ...
}
```

Listing 6.5: *SiteLockOperationBehavior*

```
[AttributeUsage(AttributeTargets.Class)]
public class SiteLockContractBehavior : Attribute, IContractBehavior {
    public Type ContractType { get; private set; }

    public SiteLockContractBehavior(Type contractType) { ... }

    public void ApplyDispatchBehavior(ContractDescription contractDescription,
        ServiceEndpoint endpoint, DispatchRuntime dispatchRuntime) {
        if (contractDescription.Name != ContractType.Name)
            return;
        foreach (var operation in contractDescription.Operations)
        {
            if (operation.Behaviors.Find<SiteLockOperationBehavior>() == null)
                operation.Behaviors.Add(new SiteLockOperationBehavior(0)); //
                    default site parameter position = 0
        }
    }
    ...
}
```

Listing 6.6: *SiteLockContractBehavior*

**Applying Custom Behaviors** How does the server application specify to which specific operations, service contracts, and services the WCF runtime should apply the custom behaviors? Thus far, the custom behaviors only specified the type of element they apply to, but not the specific elements themselves. Also remember that some properties of the custom behaviors still need to be set through the constructor. To set these properties and to specify the elements a custom behavior applies to, the server application uses the custom behavior classes as attributes. Therefore the custom behavior classes also derive from the class *Attribute*. This way, the server application can state declaratively which behaviors should be applied to which operations, service contracts or services through the WCF runtime [27].

Listing 6.7 shows how to apply custom behaviors declaratively. The *CheckNullableParameterInspector* is added to every service method whereas the *SiteLockParameterInspector* is only added to the methods of the administration endpoint. Furthermore, the server application can use the operation-scoped custom behavior classes as method attributes to disable *SiteLockParameterInspector* for server-local operations and to exclude method parameters from the non-null value checks. In the example, each administration center instance can invoke the method *GetSites* at any time. Also, the administration center has no duty to provide a value for *backupPointDate* when calling the method *DropDatabase*.

```
[SiteLockContractBehavior(typeof(IAdminService))]
[CheckNullableParameterServiceBehavior]
public class MainService : IAdminService, IControlService {
    [SiteLockOperationBehavior(DisableSiteLock = true)]
    public string[] GetSites() { ... }

    [CheckNullableParameterOperationBehavior(1)]
    public DropDatabaseTask DropDatabase(Database database, DateTime?
        backupPointDate) { ... }
    ...
}
```

Listing 6.7: Applying Custom Behaviors

## 6.4 Windows Communication Foundation Security

azDeploy uses WCF to implement the following security concepts [21]:

**Authentication:** Various applications of azDeploy verify the identity of each other. azDeploy implements the concept of authentication for the server application, the gateway agent, and the administration center bidirectionally. This means that both the administration center and the gateway agents make sure that they communicate with the server application. Also, the server application asks the administration center and the gateway agents for evidence of their identity when they connect to it.
In contrast, azDeploy does not implement authentication within the customer sites. Thus, neither the client agents know whether they really communicate with the gateway agent of the site nor does the gateway agent know if the peers connecting to it are genuine client agents.
However, azDeploy omits customer site security because the customer site is a trusted network. Therefore, azDeploy supposes that malicious gateway or client agents cannot be installed on the machines. In a real world application, developers could extend azDeploy's security mechanisms to include the customer site with a small effort.

**Authorization:** The server application ensures that gateway agents and administration center instances can only call service methods they are entitled to call. Hence, a gateway agent is not allowed to call service methods of the administration service endpoint whereas an administration center instance is not allowed to call service methods of the control service endpoint. The administration center and gateway agent themselves employ no authorization as there is only one instance of the server application which is entitled to call all operations. They ensure that they are connected to the genuine instance of the server application at the vendor site through authentication as described above.
Once again, azDeploy does not employ authorization within customer sites.

**Integrity and Confidentiality:** azDeploy ensures that communication between its applications that runs over the Internet is secure: A third party cannot tamper with messages undetected and encryption guarantees that a third party cannot eavesdrop communications.
Once again, azDeploy does not employ these concepts within customer sites.

### 6.4.1 Authentication, Integrity and Confidentiality

To fulfill the requirements integrity and confidentiality, the server application encrypts all communication with the administration center instances and the gateway agents.

**Transport and Message Security** WCF offers both *transport security* and *message security* for securing communication. When using transport security, WCF encrypts the whole communication channel and therefore offers point-to-point security. Also, the underlying transport protocol determines the available authentication modes. In contrast, when using message security, WCF encrypts each message separately. Also, each message carries the authentication credentials. Thus, message security offers more authentication modes as security does not depend on the underlying transport protocol transmitting the messages [26, p. 124-131].

**Communication with the Administration Center** The server application uses a *WSHTTPBinding* with transport security, but without *reliable session* support, to secure communications with the administration center. This is the only secure binding configuration possible for communicating with the administration center that operates under partial trust (see Section 5.4 on page 54).

The binding (Listing 6.8) uses *HTTP Basic Authentication* (see [24]) to authenticate the administration center. To specify this client authentication mode, it sets *clientCredentialType* to *Basic* [21, 23].

For HTTP transports, the server certificate for SSL is assigned externally. To do so, the server machine administrator uses the *httpcfg* tool to bind the certificate to the administration service endpoint port [22].

```
<wsHttpBinding>
    <binding name="AdministrationBinding" allowCookies="false">
        <reliableSession enabled="false" />
        <security mode="Transport">
            <transport clientCredentialType="Basic" />
        </security>
    </binding>
</wsHttpBinding>
```

Listing 6.8: Binding Configuration for the Administration Service Endpoint

**Communication with the Gateway Agents** To secure the communications with the gateway agents, the server application uses a *netTCPBinding* with transport security. However, it transmits credentials at the message level.

Again, the binding configuration (Listing 6.9) states that it uses username and password as authentication credentials at the *message layer*. Note, that it sets *clientCredentialType* to *None* for the *transport layer*. This configuration allows to use transport security for integrity, confidentiality and server authentication, and to use message security for client authentication. This means that WCF encrypts communication over the Internet with SSL over TCP and at the same time transmits username and password credentials within this communication channel's messages. Authentication via username and password is not possible at the transport layer with *netTCP* if both hosts do not share the same Windows domain [23] [26, p. 99f].

For *netTCP* transports, the server certificate is assigned through WCF configuration. The certificate is specified in the service behavior (Listing 6.10), which uses the certificate's thumbprint to look it up in the machine's certificate store [25].

```
<netTcpBinding>
    <binding name="ControlBinding" portSharingEnabled="true">
        <security mode="TransportWithMessageCredential">
           <transport clientCredentialType="None" protectionLevel="EncryptAndSign"
               />
           <message clientCredentialType="UserName" />
        </security>
    </binding>
</netTcpBinding>
```

Listing 6.9: Binding Configuration for the Control Service Endpoint

```
<behavior name="ServerBehavior">
    <serviceMetadata />
    <serviceCredentials>
        <serviceCertificate findValue="c340ddf98d51d3224e9d212157fe81f94a5ddda4"
            x509FindType="FindByThumbprint" />
        <userNameAuthentication userNamePasswordValidationMode="Windows" />
    </serviceCredentials>
    <serviceAuthorization principalPermissionMode="UseWindowsGroups" />
</behavior>
```

Listing 6.10: *Behavior* of the Server Application WCF services

### 6.4.2 Role-based Authorization

The server application uses role-based authorization to both authenticate and authorize the gateway agents and the administration center instances which connect to its services. Both the gateway agents and the administration center instances prove their identity through username and password. This fulfills the concept of authentication as the vendor only discloses these credentials to the respective gateway agent or operator. Also, the server application uses these credentials to determine the service methods a gateway agent or an administration center instance may call.

**User Account Mapping** azDeploy maps these credentials to Windows user accounts on the server machine. These user accounts do not need to exist on the machines where the gateway agents or the administration center instances run. Thus, the administrator of the server machine can manage them via the Microsoft Management Console. Gateway agent users must be members of the Windows group *az_gateways* whereas operators must be members of the Windows group *az_operators*. Each gateway agent installation stores its username and password in its application configuration file on the gateway machine. Each operator enters his/her username and password when s/he logs onto the administration center.

**Implementation** To implement role-based authorization with Windows user accounts, each WCF service exposing an endpoint to a network must be configured accordingly: The service behavior (Listing 6.10) specifies how to conduct authentication and authorization. Firstly, to map supplied credentials to Windows user accounts, *usernamePasswordValidationMode* is set to *Windows*. Therefore, only clients providing valid Windows user account credentials (authentication) can call service methods. Secondly, the WCF services are configured to employ authorization based on Windows groups, by setting *principalPermissionMode* to *UseWindowsGroups*. In turn, each service method can state to which Windows groups it is available through using the *PrincipalPermission* attribute [21]. Listing 6.11 demonstrates how the *GetHosts* service method makes itself only available to operators.

```
[PrincipalPermission(SecurityAction.Demand, Role = "az_operators")]
public Host[] GetHosts(string siteName) {
    return clientManager.GetCallback(siteName).GetSiteHosts();
}
```

Listing 6.11: Securing a Service Method with Role-based Authorization

# 7 Setup

This chapter is a guide for setting up AZDEPLOY. In the described scenario, the server application and both the gateway and client agents run as applications and not as background services. It is also possible to run these three applications on the same machine. The server application, the gateway agent, the client agent, and all setup operations must be run under a user account with administrator privileges. The following sections will discuss setting up the server machine, gateway machines, appliances and operators' workstations.

## Server Machine

Setting up the server machine consists of six steps:

**1. Creating and Installing Test Certificates** Create a temporary root certificate with the *makecert* tool (Listing 7.1, line 1). Enter a password used to protect the certificate's private key. The *makecert* tool is part of *Windows SDK 6.0A*. Then add the temporary root certificate into the *Trusted Root Certification Authorities* store. Next, create the service certificate (Listing 7.1, line 2). Replace *DEV* by the server machine's hostname. This command also adds the service certificate to the machine's certificate store. Then, use the *Microsoft Management Console* to find out the service certificate's thumbprint [26, p. 485-489] [67].

Finally, use the *httpcfg* tool to assign the certificate to the port of the administration service endpoint (Listing 7.1, line 3). Set the last parameter to the service certificate's thumbprint. The httpcfg tool is part of *Windows Support Tools [22].*

```
1 makecert -n "CN=TempCA" -r -sv TempCA.pvk TempCA.cer
2 makecert -sk azdeploy -iv TempCA.pvk -n "CN=DEV" -ic TempCA.cer -sr localmachine -
      ss my -sky exchange -pe
3 httpcfg set ssl -i 0.0.0.0:8002 -h c340ddf98d51d3224e9d212157fe81f94a5ddda4
```

Listing 7.1: SSL Setup

**2. Setting Up the Repository** To create the repository, firstly create a new directory in the file system. The path of this directory is called *repository path.* Then create the sub-directories *agent*, *database* and *software* within the repository. Next, put the setup packages for the agents called *GatewaySetup.msi* and *ClientSetup.msi* in the *agent* directory. For each supported database schema, create a directory with the schema's name in the *database* directory. Put the upgrade script for the respective schema into the newly created directory. Then, put the application installation packages directly into the software directory. Listing 7.2 shows the directory structure of a minimal repository with the repository path *C:\repository*.

```
C:\repository\agent\GatewaySetup.msi
C:\repository\agent\ClientSetup.msi
C:\repository\database\JornadaClient\JornadaClient-1.0.sql
C:\repository\software\JornadaClient1.0.msi
```

Listing 7.2: A Minimal Repository

**3. Creating the Server Database** To establish the server database, create an empty SQL Server 2008 database on the server machine and execute the *serverdb.sql* script.

**4. Configuring the Server Application** The server application is configured through the application configuration file *Server.exe.config* (Listing 7.3). Within this file, replace all occurrences of *DEV* with the hostname of the server machine. Within the server behavior configuration, change the value of the *serviceCertificate* element's *findValue* attribute to the thumbprint of the created certificate. Finally, adapt the properties *RepositoryPath* and *ServerDatabaseConnectionString* within the application settings section.

```
<configuration>
  [...]
    <serviceCertificate findValue="c340ddf98d51d3224e9d212157fe81f94a5ddda4"
        x509FindType="FindByThumbprint" />
  [...]
  <applicationSettings>
    <Server.Properties.Settings>
      <setting name="ServerDatabaseConnectionString" serializeAs="String">
        <value>Data Source=localhost\sqlexpress;Integrated Security=SSPI;Initial
            Catalog=ServerDB</value>
      </setting>
      <setting name="RepositoryPath" serializeAs="String">
        <value>C:\repository\</value>
      </setting>
    </Server.Properties.Settings>
  </applicationSettings>
</configuration>
```

Listing 7.3: Server Application Configuration File

**5. Hosting the Administration Center** Replace *DEV* by the server machine's hostname in the application configuration file of the administration center. Then create an *Internet Information Services* website and publish the administration center to it with Microsoft Visual Studio 2008. Note that the administration center must be signed with the service certificate too.

**6. Creating Groups and Users** Create two Windows Groups called *az_gateways* and *az_operators*. Then create a Windows user account for each gateway and add it to the *az_gateways* group. Then, add the operators' Windows user accounts to the *az_operators* group. Finally, start the server application.

## Other Machines

**Gateway Machines** Add the temporary root certificate to the machine's certificate store. Within the gateway agent application configuration file, set the *ServiceHostname* property to the hostname of the server machine. Set the *UserName* and *Password* properties according to the gateway user account on the server machine. Finally, start the gateway agent.

**Appliances** Create an empty directory in the file system for the database backups. Within the client agent application configuration file, set the *DatabaseBackupPath* property to this directory's path. Set the *GatewayHostname* property to the gateway machine's hostname. Create the file *connection_strings.xml* in the directories *settings/XmlConnectionStringProvider* within the client agent directory. Insert the configuration shown in Listing 7.4 and adapt the configuration string accordingly. Finally, start the client agent and the application starter.

**Operators' Workstations** On operators' workstations, it suffices to add the temporary root certificate to the machine's certificate store. To launch the administration center, navigate to the URL hosting it.

```xml
<?xml version="1.0"?>
<ConnectionStringStore xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:
    xsd="http://www.w3.org/2001/XMLSchema">
  <ConnectionStrings>
    <ConnectionStringEntry>
      <Identifier>JornadaLocal</Identifier>
      <ConnectionString>Data Source=DEV\sqlexpress;Integrated Security=SSPI;</
          ConnectionString>
    </ConnectionStringEntry>
  </ConnectionStrings>
</ConnectionStringStore>
```

Listing 7.4: Connection String Configuration

# 8 Usage

Section 8.1 shows how operators control azDeploy via the administration center web application. Section 8.2 shows how developers prepare installation packages.

## 8.1 Deploying Applications and Database Schemas

This section uses three consecutive scenarios to show how to deploy applications and database schemas. It assumes a customer site with two appliances. The controls in the screenshots were resized to take up less space.

**1. Initial Deployment** The Jornada Client and the databases are set up in the following way:

1. Log onto the administration center with your username and password.

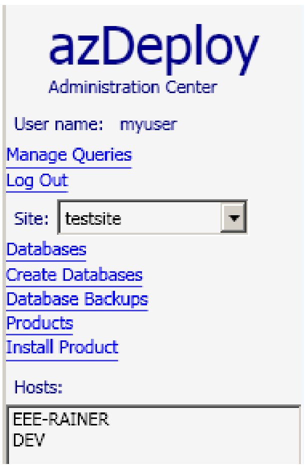
2. Select the target site in the navigation pane (Figure 8.1).

3. Click *Install Product* to navigate to the *Install Product* page (Figure 8.2). Select the Jornada Client 1.0 package, mark all appliances, and enter a descriptive comment. Click the *Install Package* button. azDeploy installs the Jornada Client on both appliances. azDeploy starts the Jornada Client on the appliances and the administration center displays the text *Completed.* Since the appliances' local databases do not exist yet, the Jornada Client cannot operate and thus locks its user interface (Figure 8.3).

4. Click *Create Databases* in the navigation pane (Figure 8.1). On the *Create Databases* page (Figure 8.4), mark all appliances, enter the name *JornadaLocalDB* for the databases, and click the *Create Databases* button. azDeploy creates the empty databases.

5. Click *Databases* on the navigation pane to navigate to the *Databases page* (Figure 8.5). Mark all databases and click *Initialize Databases* in the context menu. On the *Initialize Databases* page (Figure 8.6), select the database schema *Jornada-Client*, enter a descriptive comment, and click the *Initialize Databases* button. azDeploy only assigns the schema to the databases, but does not install a specific version. Thus, operators can also put legacy databases with an existing schema under version control.

6. To actually install a specific version of the schema, navigate again to the *Databases* page and mark all databases. Click *Upgrade Databases* in the context menu to navigate to the *Upgrade Databases* page (Figure 8.7). Select the schema version 1.0, enter a descriptive comment and click the *Upgrade Databases* button. AZ-DEPLOY installs the schema on the databases. The JORNADA Client instances unlock their user interface and are ready to use (The deployment without further configuration is possible since the JORNADA Client assumes a specific database name and uses *Integrated Security* to connect to the local database).

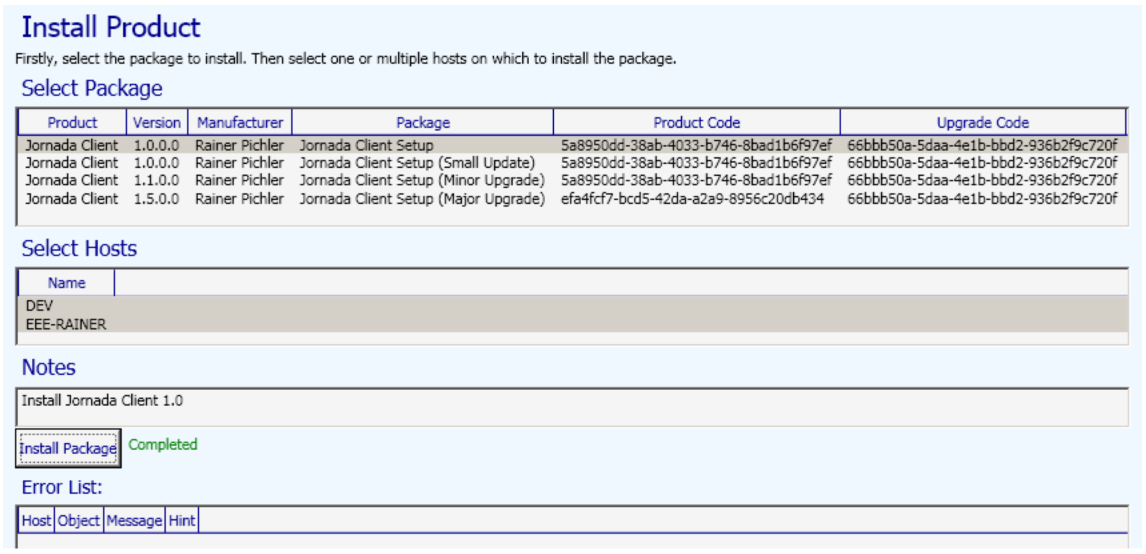


User name: myuser

Site: testsite

EEE-RAINER
DEV

Figure 8.1: Navigation Pane


Install Product

Firstly, select the package to install. Then select one or multiple hosts on which to install the package.

Select Package

| Product | Version | Manufacturer | Package | Product Code | Upgrade Code |
|---|---|---|---|---|---|
| Jornada Client | 1.0.0.0 | Rainer Pichler | Jornada Client Setup | 5a8950dd-38ab-4033-b746-8bad1b6f97ef | 66bbb50a-5daa-4e1b-bbd2-936b2f9c720f |
| Jornada Client | 1.0.0.0 | Rainer Pichler | Jornada Client Setup (Small Update) | 5a8950dd-38ab-4033-b746-8bad1b6f97ef | 66bbb50a-5daa-4e1b-bbd2-936b2f9c720f |
| Jornada Client | 1.1.0.0 | Rainer Pichler | Jornada Client Setup (Minor Upgrade) | 5a8950dd-38ab-4033-b746-8bad1b6f97ef | 66bbb50a-5daa-4e1b-bbd2-936b2f9c720f |
| Jornada Client | 1.5.0.0 | Rainer Pichler | Jornada Client Setup (Major Upgrade) | efa4fcf7-bcd5-42da-a2a9-8956c20db434 | 66bbb50a-5daa-4e1b-bbd2-936b2f9c720f |

Select Hosts

| Name |
|---|
| DEV |
| EEE-RAINER |

Notes

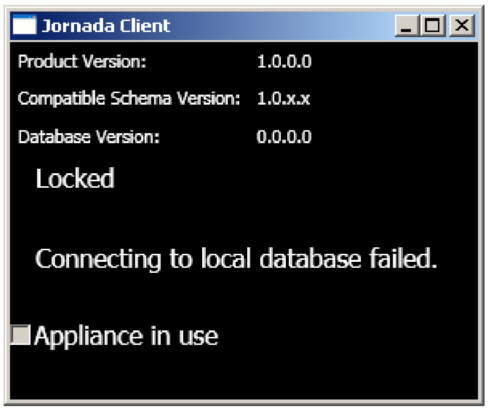
Install Jornada Client 1.0

Completed

Error List:

| Host | Object | Message | Hint |
|---|---|---|---|

Figure 8.2: Installing JORNADA Client 1.0



Jornada Client

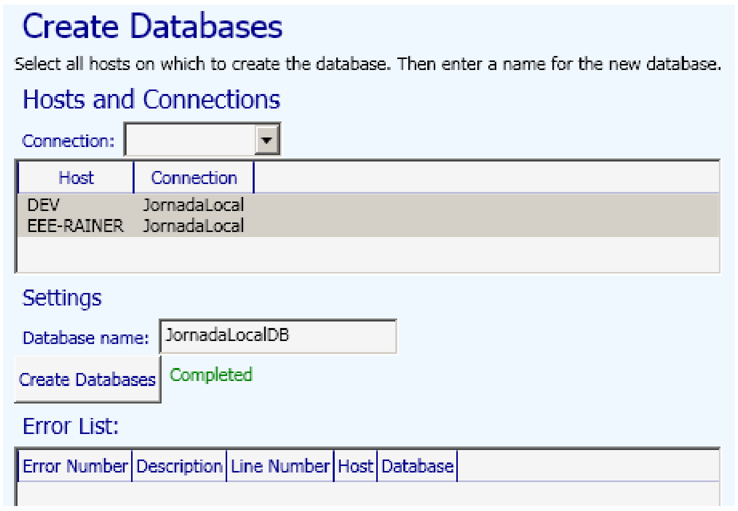
| | |
|---|---|
| Product Version: | 1.0.0.0 |
| Compatible Schema Version: | 1.0.x.x |
| Database Version: | 0.0.0.0 |

Locked

Connecting to local database failed.

Appliance in use

Figure 8.3: JORNADA Client Waiting For Database



Create Databases

Select all hosts on which to create the database. Then enter a name for the new database.

Hosts and Connections

Connection:

| Host | Connection |
|---|---|
| DEV | JornadaLocal |
| EEE-RAINER | JornadaLocal |

Settings

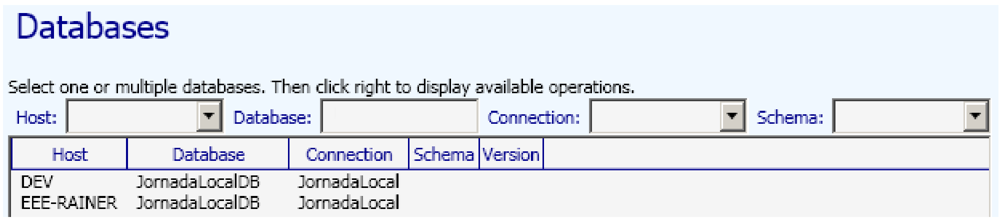
Database name: JornadaLocalDB

Create Databases Completed

Error List:

| Error Number | Description | Line Number | Host | Database |
|---|---|---|---|---|

Figure 8.4: Creating Databases



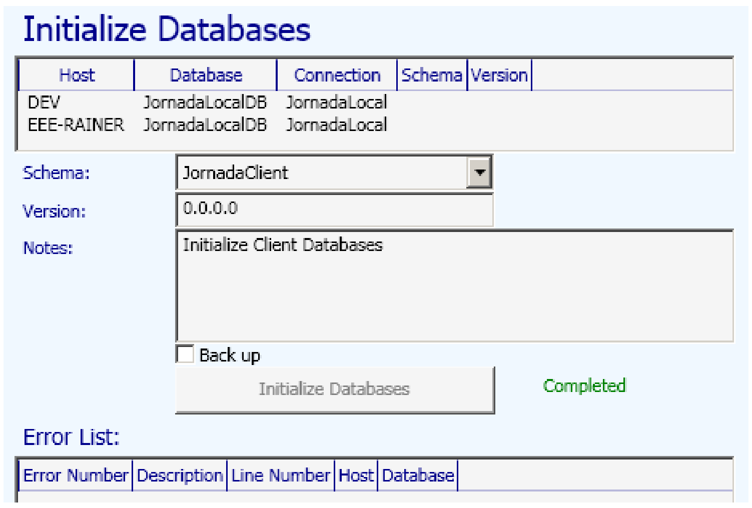
Databases

Select one or multiple databases. Then click right to display available operations.

Host: Database: Connection: Schema:

| Host | Database | Connection | Schema | Version |
|---|---|---|---|---|
| DEV | JornadaLocalDB | JornadaLocal | | |
| EEE-RAINER | JornadaLocalDB | JornadaLocal | | |

Figure 8.5: Databases Page



Initialize Databases

| Host | Database | Connection | Schema | Version |
|---|---|---|---|---|
| DEV | JornadaLocalDB | JornadaLocal | | |
| EEE-RAINER | JornadaLocalDB | JornadaLocal | | |

Schema: JornadaClient

Version: 0.0.0.0

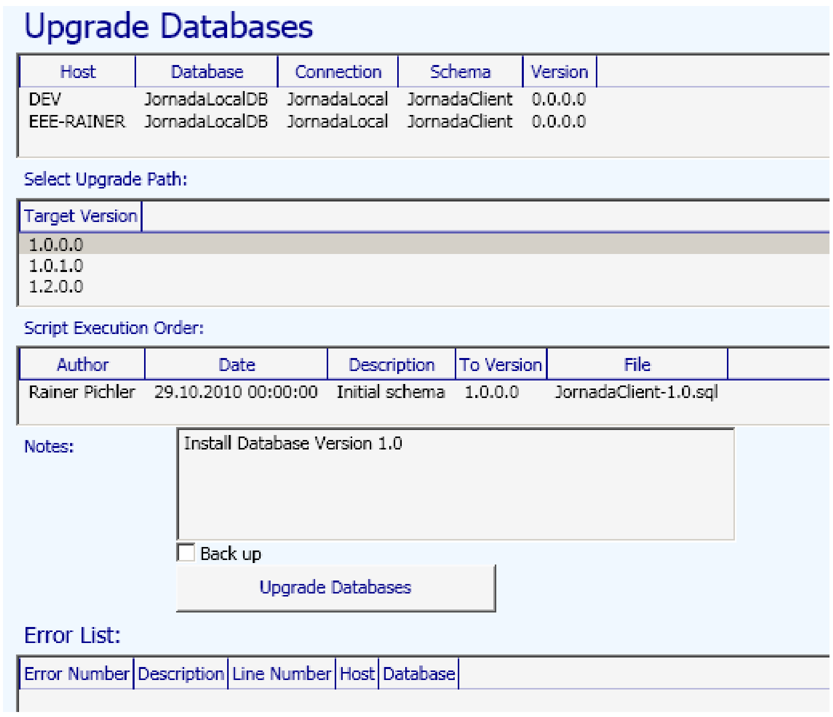
Notes: Initialize Client Databases

☐ Back up

Error List:

| Error Number | Description | Line Number | Host | Database |
|---|---|---|---|---|

Figure 8.6: Initializing Databases



Upgrade Databases

| Host | Database | Connection | Schema | Version |
|---|---|---|---|---|
| DEV | JornadaLocalDB | JornadaLocal | JornadaClient | 0.0.0.0 |
| EEE-RAINER | JornadaLocalDB | JornadaLocal | JornadaClient | 0.0.0.0 |

Select Upgrade Path:

| Target Version |
|---|
| 1.0.0.0 |
| 1.0.1.0 |
| 1.2.0.0 |

Script Execution Order:

| Author | Date | Description | To Version | File |
|---|---|---|---|---|
| Rainer Pichler | 29.10.2010 00:00:00 | Initial schema | 1.0.0.0 | JornadaClient-1.0.sql |

Notes: Install Database Version 1.0

☐ Back up

Error List:

| Error Number | Description | Line Number | Host | Database |
|---|---|---|---|---|

Figure 8.7: Installing Database Schema 1.0

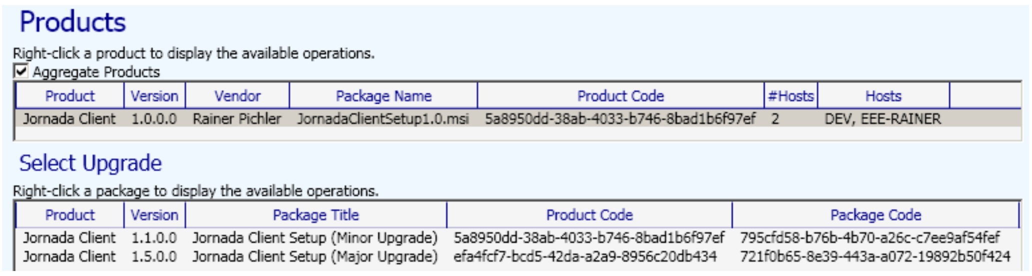
**2. Application Upgrade with Errors** This scenario shows how the Jornada Client can be upgraded and how one appliance denies the upgrade:

1. To simulate a busy appliance, the Jornada Client's *Appliance in Use* checkbox is ticked.

2. Click *Products* in the navigation pane (Figure 8.1) to navigate to the *Products* page (Figure 8.8). Mark the Jornada Client entry and select the upgrade to version 1.1. Click *Install Product* in the upgrade entry's context menu to navigate to the *Install Product* page (Figure 8.9). Enter a descriptive comment and click the *Install Product* button. Since one appliance is in use, azDeploy can only upgrade and restart the Jornada Client on the idle appliance. Thus, the administration center shows an error for the appliance in use in the error list. This is also the general error handling behavior: If the actual operation fails, azDeploy leaves the affected appliance in a consistent state by rolling back the changes it made. In turn, it shows an error in the administration center's error list so that the operator can resolve the problem. Nevertheless, it does not roll back the operations that already were applied successfully to other appliances. Thus, azDeploy does not implement site-wide consistency (see Chapter 9).

3. The Jornada Client's *Appliance in Use* checkbox is unticked again.

4. Repeat step 2 for the now idle appliance to also upgrade it's Jornada Client.



Products

Right-click a product to display the available operations.

☑ Aggregate Products

| Product | Version | Vendor | Package Name | Product Code | #Hosts | Hosts |
|---|---|---|---|---|---|---|
| Jornada Client | 1.0.0.0 | Rainer Pichler | JornadaClientSetup1.0.msi | 5a8950dd-38ab-4033-b746-8bad1b6f97ef | 2 | DEV, EEE-RAINER |

Select Upgrade

Right-click a package to display the available operations.

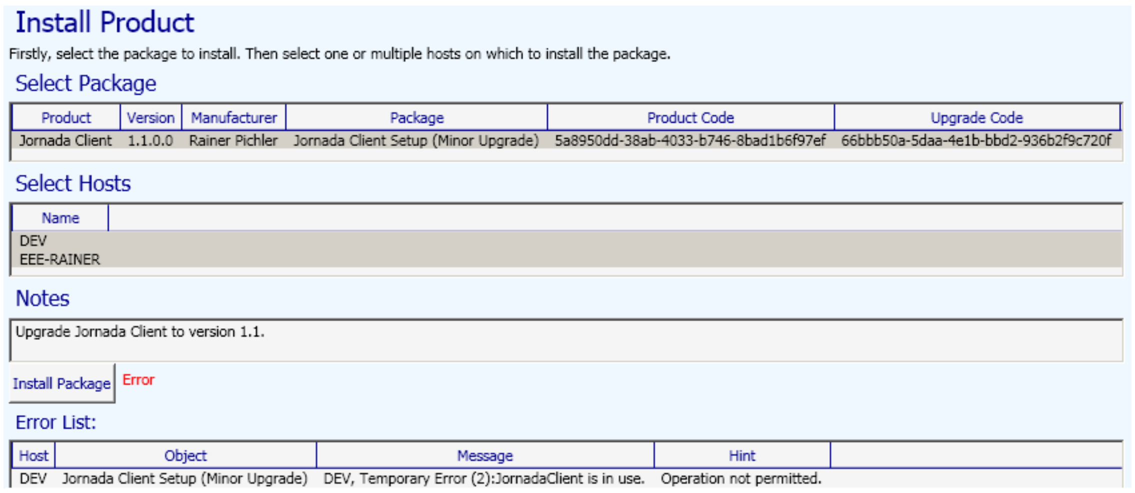
| Product | Version | Package Title | Product Code | Package Code |
|---|---|---|---|---|
| Jornada Client | 1.1.0.0 | Jornada Client Setup (Minor Upgrade) | 5a8950dd-38ab-4033-b746-8bad1b6f97ef | 795cfd58-b76b-4b70-a26c-c7ee9af54fef |
| Jornada Client | 1.5.0.0 | Jornada Client Setup (Major Upgrade) | efa4fcf7-bcd5-42da-a2a9-8956c20db434 | 721f0b65-8e39-443a-a072-19892b50f424 |

Figure 8.8: Products Page

**3. Application and Database Upgrade** This scenario shows how both the Jornada Client and the databases can be upgraded:

1. Let azDeploy upgrade all appliances' databases to the schema version 1.2 (analogous to Figure 8.7). Since the new schema version is incompatible to the Jornada Client 1.1 (Table 8.1), the Jornada Client instances lock their user interface.

2. Let azDeploy upgrade the Jornada Client to version 1.5 on both appliances. Since the newly installed Jornada Client is compatible with the schema version 1.2 (Table 8.1), the appliance can be used again.



Install Product

Firstly, select the package to install. Then select one or multiple hosts on which to install the package.

Select Package

| Product | Version | Manufacturer | Package | Product Code | Upgrade Code |
|---|---|---|---|---|---|
| Jornada Client | 1.1.0.0 | Rainer Pichler | Jornada Client Setup (Minor Upgrade) | 5a8950dd-38ab-4033-b746-8bad1b6f97ef | 66bbb50a-5daa-4e1b-bbd2-936b2f9c720f |

Select Hosts

| Name |
|---|
| DEV |
| EEE-RAINER |

Notes

Upgrade Jornada Client to version 1.1.

Install Package Error

Error List:

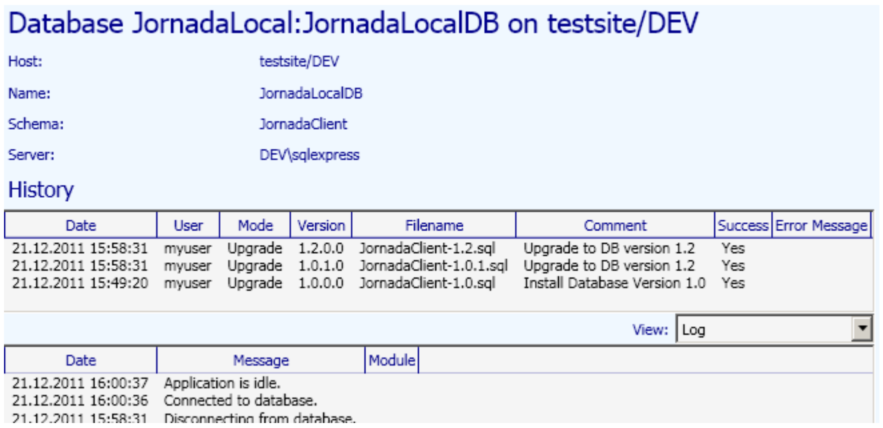
| Host | Object | Message | Hint |
|---|---|---|---|
| DEV | Jornada Client Setup (Minor Upgrade) | DEV, Temporary Error (2):JornadaClient is in use. | Operation not permitted. |

Figure 8.9: Upgrading to JORNADA Client 1.1

| JORNADA Client | Database Schema |
|---|---|
| 1.0, 1.1 | 1.0.x |
| 1.5 | 1.2.x |

Table 8.1: Application and Database Compatibility

3. Mark one database on the *Databases* page (analogous to Figure 8.5) and click *Show Database Details* in the context menu. On the *Database Details* page (Figure 8.10), the database's upgrade history can be inspected. It shows that the last upgrade consisted of two upgrade scripts. Also, the JORNADA Client's log entries can be inspected since such a view has been defined on the server. Custom queries can be run and saved by clicking *Execute Query* in the database's context menu on the *Databases* page (Figure 8.5).



Database JornadaLocal:JornadaLocalDB on testsite/DEV

| | |
|---|---|
| Host: | testsite/DEV |
| Name: | JornadaLocalDB |
| Schema: | JornadaClient |
| Server: | DEV\sqlexpress |

History

| Date | User | Mode | Version | Filename | Comment | Success | Error Message |
|---|---|---|---|---|---|---|---|
| 21.12.2011 15:58:31 | myuser | Upgrade | 1.2.0.0 | JornadaClient-1.2.sql | Upgrade to DB version 1.2 | Yes | |
| 21.12.2011 15:58:31 | myuser | Upgrade | 1.0.1.0 | JornadaClient-1.0.1.sql | Upgrade to DB version 1.2 | Yes | |
| 21.12.2011 15:49:20 | myuser | Upgrade | 1.0.0.0 | JornadaClient-1.0.sql | Install Database Version 1.0 | Yes | |

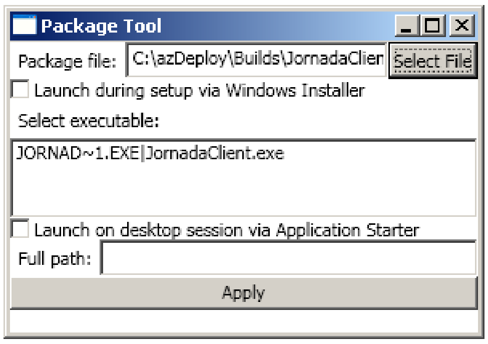
View: Log

| Date | Message | Module |
|---|---|---|
| 21.12.2011 16:00:37 | Application is idle. | |
| 21.12.2011 16:00:36 | Connected to database. | |
| 21.12.2011 15:58:31 | Disconnecting from database. | |

Figure 8.10: Database Details

## 8.2 Preparing Application Packages

Application setup packages are prepared for deployment via the package tool (Figure 8.11) in the following way: Click *Select File* to select the package file. To run an executable of the package under administrative privileges as a part of the installation process, for instance to migrate configuration files, tick the checkbox *Launch during setup via Windows Installer* and select the executable in the listbox control. To run the application within the desktop session after the setup, tick the checkbox *Launch on desktop session via Application Starter* and enter the executable's absolute installation path. Finally, click the button *Apply*.



Package file: C:\azDeploy\Builds\JornadaClien
Launch during setup via Windows Installer
Select executable:
JORNAD~1.EXE|JornadaClient.exe
Launch on desktop session via Application Starter
Full path:

Figure 8.11: The Package Tool

# 9 Discussion

This thesis introduced azDeploy which can deploy application upgrades and database schema upgrades to multiple appliances in a remote network. Although the implementation covers the presented requirements and features, it is considered a prototype.

**More Testing** To mature, azDeploy needs more testing. Firstly, azDeploy must be tested with large networks to test how well it scales. This was not possible throughout the development due to a lack of that many machines. Testing with two machines only revealed that the parallelism approach works and the server application and gateway agents are not blocked while client agents execute long-running operations. Running multiple instances of the client agent to simulate a larger network is a cumbersome task and promises little insight due to its nature as a system tool: For instance, it would not be possible to install the same application to multiple such "appliances" at once. Also, the vendor applications would not know with which instance of the notification service they should interact.

Secondly, azDeploy needs to prove that it fulfills the requirements of a real-world scenario and yields the advantages over other approaches as discussed in Chapter 1. Such testing also provides insight whether the notification services suffice to integrate vendor applications seamlessly into the deployment process.

**Bottom Line** Designing and developing azDeploy posed several challenges: Windows Communication Foundation and Windows Installer were new topics for the author. Also, the decision to run the administration center under partial trust and the need for the gateway agent imposed restrictions on the whole architecture. Finally, keeping azDeploy generic required to design flexible yet useful interfaces.

All in all, this work yielded two outcomes: Firstly, the author gained deeper knowledge in Windows Communication Foundation and Windows Installer. Secondly, it resulted in a generic system concept and in a prototype. Because this thesis' topic is a complex one, there are still worthwhile features to implement. Thus, the final section of the thesis gives suggestions for improving azDeploy through leveraging existing functionality.

## Further Work

Throughout writing the thesis and implementing the prototype, several ideas for further work emerged. The first four suggestions aim to improve the performance whereas the last two suggestions concern the architecture of azDeploy:

**File Transfers** Switching to streamed transfers (see [68]) instead of transmitting chunks of data may speed up file transfers. Additionally, caching files on the gateway machine can be implemented with a small effort (see Chapter 4.7).

**Message Pass-through** For most operations, the gateway agent merely acts as a proxy between the server application and the client agent. Thus, it may pay off for it to not transform the underlying messages into operations which in turn create new messages, but pass them on unmodified to the client agents.

**Parallel Notifications** The notification services contact all appliances and vendor applications sequentially. For productive use, they should use asynchronous operations to notify all appliances in parallel. Then, for instance, each vendor application can try to lock itself for users due to a configuration change within a longer time span without blocking the notification services which wait for an approval of the configuration change request.

**Service Instancing** The server application may turn out to be the bottleneck of AZ-DEPLOY as it only processes one message from either the administration or the control endpoint at any time. Because an operator administers exactly one site at a time and sites do not interact with each other, it is possible to create a separate per-session *MainService* instance for each connected site. This way, *MainService* could communicate with several gateway agents in parallel without concurrency problems as each instance does yet process messages sequentially. However, still a single *MainService* instance would serve the administration endpoint unless this endpoint supports sessions too. This central instance must then be able to contact and receive remote events from all other instances.

**Tasks and Notifications** The interaction between the task system and the notification services should be refined. The task system is designed to be simple but yet flexible: The server application transforms composite tasks into a set of simple sub-tasks through *control flows* (see Section 4.5.2 on page 31). In turn, the client agent executes these sub-tasks. Therefore, some sub-task implementations can be reused in various complex tasks and at the same time, the client agent does not (need to) know about the composite task at all.

However, the latter constrains the notification services to work on the level of sub-tasks. Therefore, when upgrading a database schema with making a backup first, the vendor applications cannot realize that the preceding backup is part of conducting the database schema upgrade. In turn, it may happen that the client agent backs up the database and receives the upgrade scripts successfully, but upgrading the database fails because the application is in use and refuses the upgrade.

This problem can be solved through notifying the vendor applications about the composite tasks: A solution is to trigger notifications via separate sub-tasks instead of handling them within the sub-tasks running the actual operation steps. Therefore, a composite task's sequence of sub-tasks would be enclosed by sub-tasks triggering the notifications for the operation as a whole. In the database schema upgrade example, the sub-task order would be: *NotifyRequestTask*, *NotifyActivityTask*, *TransferFilesTask*, *BackupDatabaseTask*, *UpgradeDatabaseTask* and *NotifyCompletionTask*. Thus, upgrading the database schema would appear as a single operation to the vendor applications.

**Extended Consistency** Finally, consistency may be extended: azDeploy applies application and database schema upgrades within transactions. Thus, when applying an application upgrade to several appliances, it may be that it only works for some appliances and the rest of the appliances rolls back to the previous configuration. Especially if also the database schema was upgraded to a version breaking compatibility before, the appliances still using the old application release would switch to an out of service state. To make these appliances work again, the operators need to try to upgrade them again via azDeploy, as long as the previous upgrade attempt failed only temporarily. If the upgrade fails permanently, they must manually upgrade the application. All in all, azDeploy guarantees only that the appliances stay in a consistent configuration, but not the customer site as a whole.

However, ensuring site-wide consistency can be achieved through splitting the upgrade operations into two parts: The first part begins a transaction and runs the whole operation, whereas the second part either commits or rolls back the transaction. This way, when upgrading an application on multiple appliances, azDeploy starts the transaction and the upgrade on all appliances. If the upgrade was successful on all appliances, azDeploy commits the transaction on all appliances. In contrast, if it failed on a single appliance, azDeploy rolls back the transaction on all appliances. The same applies to database schema upgrades. This way, even application and database schema upgrades dependent on each other could be applied within one site-wide transaction.

This principle can be implemented via the task system: Existing tasks like *UpgradeDatabaseTask* would implement the first part of the operation. Additionally, a new *EndTransactionTask* either commits or rolls back the transaction. Finally, a server-side *control flow* (see Section 4.5.2 on page 31) coordinates the upgrade across multiple appliances or databases as described in the previous paragraph.

While working with database transactions is a straightforward task, splitting application upgrades into the two parts discussed earlier may require some effort: It should be investigated whether azDeploy can control Windows Installer transactions via the automation interface. If not, *custom actions* (see Section 6.2.1 on page 58) could integrate .NET code into the installation sequence. In turn, this code could communicate with the client agent and influence the installation process.

# Bibliography

[1] Version Class, MSDN Library, http://msdn.microsoft.com/en-us/library/system.version%28v=VS.90%29.aspx

[2] Windows Communication Foundation Endpoints: Addresses, Bindings, and Contracts, MSDN Library, http://msdn.microsoft.com/en-us/library/ms733107%28v=VS.90%29.aspx

[3] Designing Service Contracts, MSDN Library, http://msdn.microsoft.com/en-us/library/ms733070%28v=VS.90%29.aspx

[4] System-Provided Bindings, MSDN Library, http://msdn.microsoft.com/en-us/library/ms730879%28v=VS.90%29.aspx

[5] Publishing Metadata, MSDN Library, http://msdn.microsoft.com/en-us/library/aa751951%28v=VS.90%29.aspx

[6] Skonnard, A.: WCF Bindings In Depth, MSDN Magazine (Jul 2007), http://msdn.microsoft.com/en-us/magazine/cc163394.aspx

[7] Using Data Contracts, MSDN Library, http://msdn.microsoft.com/en-us/library/ms733127%28v=VS.90%29.aspx

[8] Specifying and Handling Faults in Contracts and Services, MSDN Library, http://msdn.microsoft.com/en-us/library/ms733721%28v=VS.90%29.aspx

[9] Sending and Receiving Faults, MSDN Library, http://msdn.microsoft.com/en-us/library/ms732013%28v=VS.90%29.aspx

[10] Lowy, J.: What You Need To Know About One-Way Calls, Callbacks, And Events, MSDN Magazine (Oct 2006), http://msdn.microsoft.com/en-us/magazine/cc163537.aspx

[11] Using Sessions, MSDN Library, http://msdn.microsoft.com/en-us/library/ms733040%28v=VS.90%29.aspx

[12] Sessions, Instancing, and Concurrency in Windows Communication Foundation, MSDN Library, http://msdn.microsoft.com/en-us/library/ms731193%28v=VS.90%29.aspx

[13] Configuring HTTP and HTTPS, MSDN Library, 2009, http://msdn.microsoft.com/en-us/library/ms733768%28v=VS.90%29.aspx

[14] Windows Presentation Foundation Partial Trust Security, MSDN Library, 2009, http://msdn.microsoft.com/en-us/library/aa970910%28v=VS.90%29.aspx

[15] Securing ClickOnce Applications, MSDN Library, http://msdn.microsoft.com/en-us/library/76e4d2xw.aspx

[16] Lowy, J.: Code Access Security in WCF, Part 1, MSDN Magazine (Apr 2008), http://msdn.microsoft.com/en-us/magazine/cc500644.aspx

[17] How to: Host a WCF Service in IIS, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/ms733766.aspx

[18] Vasters, C.: Introduction to Reliable Messaging with the Windows Communication Foundation, MSDN Library, 2006, http://msdn.microsoft.com/en-us/library/aa480191.aspx

[19] DispatcherTimer Class, MSDN Library, http://msdn.microsoft.com/en-us/library/system.windows.threading.dispatchertimer%28v=VS.90%29.aspx

[20] Dispatcher Class, MSDN Library, 2008, http://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcher%28v=VS.90%29.aspx

[21] Bustamante, M.: Fundamentals of WCF Security, CODE Magazine (Nov/Dec 2006), http://www.code-magazine.com/article.aspx?quickid=0611051

[22] How to: Configure a Port with an SSL Certificate, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/ms733791%28v=VS.90%29.aspx

[23] Selecting a Credential Type, MSDN Library, http://msdn.microsoft.com/en-us/library/ms733836%28v=VS.90%29.aspx

[24] Franks, J. et al.: HTTP Authentication: Basic and Digest Access Authentication, Network Working Group, 1999, ftp://ftp.rfc-editor.org/in-notes/rfc2617.txt

[25] How to: Use Transport Security and Message Credentials, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/ms789011%28v=VS.90%29.aspx

[26] Meier, J.D. et al.: Improving Web Services Security, Microsoft patterns & practices, 2008, http://wcfsecurityguide.codeplex.com/releases/view/15892

[27] Skonnard, A.: Service Station: Extending WCF with Custom Behaviors, MSDN Magazine (Dec 2007), http://msdn.microsoft.com/en-us/magazine/cc163302.aspx

[28] Windows Installer, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/cc185688.aspx

[29] Patching and Upgrades, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa370579%28v=VS.85%29.aspx

[30] Windows Installer Features, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa372840%28v=vs.85%29.aspx

[31] Installation Context, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/dd765197%28v=VS.85%29.aspx

[32] About the Installer Database, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa367441%28v=VS.85%29.aspx

[33] Database Tables, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa368259%28v=VS.85%29.aspx

[34] CustomAction Table, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa368062%28v=VS.85%29.aspx

[35] File Table, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa368596%28v=VS.85%29.aspx

[36] InstallExecuteSequence Table, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa369500%28v=VS.85%29.aspx

[37] Upgrade Table, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa372379%28v=VS.85%29.aspx

[38] Summary Property Descriptions, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa372049%28v=VS.85%29.aspx

[39] Package Codes, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa370568%28v=VS.85%29.aspx

[40] Property Reference, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa370905%28v=VS.85%29.aspx

[41] Using a Sequence Table, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa372404%28v=VS.85%29.aspx

[42] InstallUISequence Table, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa369543%28v=VS.85%29.aspx

[43] INSTALL Action, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa369547%28v=VS.85%29.aspx

[44] FindRelatedProducts Action, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa368600%28v=VS.85%29.aspx

[45] Using Properties in Conditional Statements, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa372435%28v=VS.85%29.aspx

[46] Installed Property, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa369297%28v=vs.85%29.aspx

[47] Custom Action Types, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa372048%28v=VS.85%29.aspx

[48] Standard Action Reference, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa372023%28v=VS.85%29.aspx

[49] ProcessComponents Action, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa370853%28v=VS.85%29.aspx

[50] RemoveFiles Action, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa371199%28v=VS.85%29.aspx

[51] InstallFiles Action, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa369503%28v=VS.85%29.aspx

[52] RegisterProduct Action, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa371162%28v=VS.85%29.aspx

[53] RemoveExistingProducts Action, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa371197%28v=VS.85%29.aspx

[54] InstallFinalize Action, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa369505%28v=VS.85%29.aspx

[55] Custom Action Type 18, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa368077%28v=VS.85%29.aspx

[56] Custom Action Return Processing Options, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa368071%28v=VS.85%29.aspx

[57] About the Automation Interface, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa367439%28v=VS.85%29.aspx

[58] Database Object, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa368254%28v=VS.85%29.aspx

[59] Installer Object, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa369432%28v=VS.85%29.aspx

[60] Summary Information Stream Property Set, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa372045%28v=VS.85%29.aspx

[61] Working with Queries, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa372879%28v=VS.85%29.aspx

[62] MsiCloseHandle Function, MSDN Library, http://msdn.microsoft.com/en-us/library/aa370067%28VS.85%29.aspx

[63] Marshal.ReleaseComObject Method, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.marshal.releasecomobject%28v=VS.90%29.aspx

[64] MsiInstallProduct Function, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa370315%28v=VS.85%29.aspx

[65] MsiConfigureProduct Function, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/aa370070%28v=VS.85%29.aspx

[66] How to: Create Temporary Certificates for Use During Development, MSDN Library, 2010, http://msdn.microsoft.com/en-us/library/ms733813%28v=VS.90%29.aspx

[67] Working with Certificates, MSDN Library, http://msdn.microsoft.com/en-us/library/ms731899%28v=VS.90%29.aspx

[68] Streaming Message Transfer, MSDN Library, http://msdn.microsoft.com/en-us/library/ms731913.aspx

The MSDN references without publication year date to 04.09.2011.

# List of Abbreviations

| | |
|---|---|
| CAS | Code Access Security |
| COM | Component Object Model |
| GUID | Globally Unique Identifier |
| HTTP | Hypertext Transfer Protocol |
| IIS | Internet Information Services |
| MSI | Microsoft Windows Installer |
| SSL | Secure Sockets Layer |
| URL | Uniform Resource Locator |
| WCF | Windows Communication Foundation |
| WPF | Windows Presentation Foundation |
| XAML | Extensible Application Markup Language |
| XBAP | XAML Browser Application |

# Lebenslauf

| | |
|---|---|
| Name: | Rainer Pichler |
| Geburtsjahr: | 1986 |
| Geburtsort: | Linz, Österreich |
| Nationalität: | Österreich |

## Bildung

| | |
|---|---|
| seit 2006 | Magisterstudium Wirtschaftswissenschaften mit Spezialisierung Organisation an der JKU Linz |
| 2009-2012 | Masterstudium Software Engineering mit Nebenfach Netzwerke und Sicherheit an der JKU Linz |
| 2005-2009 | Bakkalaureatsstudium Informatik an der JKU Linz |
| 2000-2005 | HTBLA Leonding Expositur Perg für EDV und Organisation mit Schwerpunkt “Kommerzielle Datenverarbeitung”, Matura mit ausgezeichnetem Erfolg bestanden |
| 1996-2000 | BG/BRG Freistadt |

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 30.12.2011

Rainer Pichler