Comput Sci Res Dev
DOI 10.1007/s00450-011-0182-0

SPECIAL ISSUE PAPER

Composing user-specific web applications from distributed plug-ins

**Markus Jahn · Reinhard Wolfinger ·
Markus Löberbauer · Hanspeter Mössenböck**

Abstract Plug-in frameworks support the development of component-based software that is extensible and can be customized to the needs of specific users. However, most plug-in frameworks target desktop applications and do not support web applications that can be extended by end users. In contrast to that, our plug-in framework Plux supports customizable and extensible web applications. Plux tailors a web application to the needs of every user, by assembling it from a user-specific component set. Furthermore, Plux supports end-user extensions, by integrating components provided by the end user into the web application. And finally, Plux supports distributed web applications, by integrating components on the client machines into the web application.

Keywords Component-based software · Plug-in architecture · Web programming · Run-time adaption

1 Introduction

Although modern software systems tend to become more and more powerful and feature-rich they are still often felt to be incomplete. It will hardly ever be possible to hit all user requirements out of the box, regardless of how big and complex an application is. One solution to this problem are plug-in frameworks that allow developers to build a thin layer of basic functionality that can be extended by plug-in components and thus tailored to the needs of specific users.

Most plug-in frameworks target desktop applications, but are typically unsuitable for building extensible web applications. For us, a web application is a program that is concurrently used by multiple persons, over a network using a web browser. In domains where customer requirements vary greatly (e.g., in business software) a web application should be extensible by end users to meet their specific needs.

Making a web application extensible must go beyond componentization. Every user should be able to extend the application in his own way, i.e. he should be able to add custom extensions without changing the application for other users. We are aware that extensions provided by end users raise security concerns. Therefore Plux implements a security mechanism based on signed assemblies and .NET code access security. Depending on the identity of the manufacturer, Plux decides if a component can be loaded and in which parts of the application it can be used. Components can also be partially trusted, in that case they can be executed in a sandbox with limited rights. However, security of plug-in-based applications is beyond the scope of this paper.

Distribution of components across several computers is another issue, because installing components only on the server does not cover all extensibility scenarios. For example, if a component needs to access client-side hardware, such as a barcode scanner, the component must run on the client and not on the server. Such client-side components should be capable of being integrated into the web application as well.

Over the past few years we have developed the plug-in framework Plux. Originally, Plux focused on dynamically

M. Jahn · R. Wolfinger (✉) · M. Löberbauer · H. Mössenböck
Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University, 4040 Linz, Austria
e-mail: wolfinger@ase.jku.at

M. Jahn
e-mail: jahn@ase.jku.at

M. Löberbauer
e-mail: loeberbauer@ase.jku.at

H. Mössenböck
e-mail: moessenboeck@ase.jku.at

Published online: 24 June 2011

reconfigurable desktop applications. In this paper, however, we present an extended version of Plux for web applications, which addresses the problems of extensibility and distribution. A Plux web application can be extended by custom plug-ins both on the server-side and on the client-side. Plug-ins which are distributed across multiple computers can still be integrated into a single seamless application.

Our research was conducted in cooperation with BMD Systemhaus GmbH. BMD is a medium-sized company offering a comprehensive suite of enterprise applications, such as customer relationship management, accounting, production planning and control. Because BMD's target market is fairly diversified, ranging from small tax counsellors to large corporations, customization and extensibility are essential parts of BMD's business strategy. As BMD offers both a desktop and a web version of their software, they want to use Plux for both versions and reuse components where possible.

This paper is organized as follows: Sect. 2 describes the basic concepts of the Plux framework. It focuses on the concepts that are common to the desktop and to the web version of Plux. Section 3 describes the architecture of the Plux composition infrastructure and the automatic composition process. Section 4 presents Plux for web applications and shows usage scenarios for client and server plug-ins using a motivating example. Section 5 is a follow-up to Sect. 3 and describes the additional Plux concepts required for web applications. Section 6 shows how to host a Plux web application on an ASP.NET server. Section 7 discusses related work. It describes to what extent current desktop plug-in frameworks can be used to build extensible web applications and how non-plug-in-based web development platforms address extensibility. Section 8 finishes with a summary and an outlook to future work.

2 Concepts of Plux

The Plux framework supports the dynamic composition of applications using a plug-and-play approach [22]. It facilitates extensible and customizable applications that can be reconfigured without restarting them. Reconfiguring applications can be done in a plug-and-play manner and does not require any programming. If a user wants to add a feature, he just drops a plug-in (i.e., a DLL file) into a directory. Plux discovers the plug-in on-the-fly and integrates it into the application without requiring a restart. Similarly, if the user wants to remove a feature, he removes the corresponding plug-in from the directory.

Together with our industrial partner BMD, we applied Plux to their customer relationship management (CRM) product [15]. By allowing dynamic addition and removal of CRM features, we support a set of new usage scenarios, such as on-the-fly product customization during sales conversations or incremental feature addition for step-by-step user trainings [23].

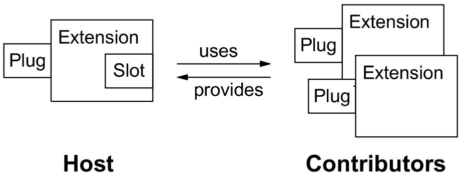


Fig. 1 Extensions with slots and plugs

The main characteristics of Plux are: the *composer*, the *composition events*, the *composition state*, and the *replaceable component discovery mechanism*. These characteristics distinguish Plux from other plug-in systems [2], such as OSGi [16], Eclipse [7], and NetBeans [3], and allow Plux to replace programmatic composition by automatic composition. *Programmatic composition* means that components query a service registry and integrate other components programmatically. *Automatic composition* means that the components declare their requirements and provisions using metadata; the *composer* in Plux uses these metadata to match requirements and provisions and to connect matching components automatically. During composition, Plux sends *composition events* to which the affected components can react. Plux also maintains the *current composition state*, i.e. it stores which components use which other components. As components can retrieve the global composition state, they do not need to store references to the components they use. *Discovery* is the process of detecting new components and extracting their metadata. Unlike in other plug-in systems, the discovery mechanism is not an integral part of Plux, but is a plug-in itself. This makes the mechanism replaceable. The following subsections cover those characteristics in more detail.

2.1 Metadata

Plux uses the metaphor of extensions that have slots and plugs (Fig. 1). An *extension* is a component that provides services to other extensions and uses services provided by other extensions. If an extension wants to use a service of some other extension it declares a *slot*. Such an extension is called a *host*. If an extension wants to provide its service to other extensions it declares a *plug*. Such an extension is called a *contributor*.

Slots and plugs are identified by names. A plug matches a slot if their names match. If so, Plux will try to connect the plug to the slot. A slot represents an interface, which has to be implemented by a matching plug. The interface is specified in a so-called *slot definition*. A slot definition has a unique name as well as optional parameters that are provided by the contributors and retrieved by the hosts. The

```
public enum LoggerKind { Warning, Error }

[SlotDefinition(“Logger”)]
[ParamDefinition(“Kind”, typeof(LoggerKind))]
public interface ILogger {
  void Print(string msg);
}
```

Fig. 2 Definition of the *Logger* slot

```
[Extension]
[Plug(“Logger”)]
[Param(“Kind”, LoggerKind.Error)]
public class ErrorLogger : ILogger{
  public void Print(string msg){
    Console.WriteLine(msg);
  }
}
```

Fig. 3 *ErrorLogger* as a contributor for the *Logger* slot

names of slots and plugs refer to the respective slot definitions.

The means to provide metadata is customizable in Plux. The default mechanism extracts metadata from .NET attributes in assembly files. Attributes are pieces of information that can be attached to .NET constructs, such as classes, interfaces, methods, or fields. At run time, the attributes can be retrieved using reflection [6].

Plux has the following custom attributes: The *SlotDefinition* attribute to tag an interface as a slot definition, the *Extension* attribute to tag classes that implement components, the *Slot* attribute to specify requirements for optional contributors in hosts, the *Plug* attribute to specify provisions in contributors, the *ParamDefinition* attribute to declare required parameters in slot definitions, and the *Param* attribute to specify provided parameter values in contributors. Although the default mechanism for providing metadata (namely by .NET attributes) limits parameter values to compile-time constants, Plux in general can use arbitrary objects as parameter values.

Let us look at an example now. Assume that a host wants to print log messages as errors or warnings. The loggers should be implemented as contributors that plug into the host. Every logger should use a parameter to specify whether it prints errors or warnings. First, we have to define the slot into which the logger can plug (Fig. 2).

Next, we write logger contributors. Figure 3 shows the logger for errors. The logger for warnings is implemented similarly (not shown). Since *ErrorLogger* has a *Logger* plug it has to implement the interface *ILogger* specified in the slot definition of *Logger*. It also has to provide a value for the parameter *Kind* specified in the slot definition.

Finally, we implement the application that uses the loggers (Fig. 4). In order to be able to use loggers it has a *Logger* slot. It also has an *Application* plug that fits into the *Application* slot of the Plux core. At startup, Plux creates an instance of *HostApp* and connects it to the core. The full implementation of *HostApp* is shown in Sect. 2.4.

```
[Extension]
[Plug(“Application”)]
[Slot(“Logger”)]
public class HostApp : IApplication{
  public HostApp(Extension e){...}
  void Work(){...}
}
```

Fig. 4 Application host with a *Logger* slot

2.2 Discovery

In order to match requirements and provisions, Plux needs the metadata of the extensions. Extensions are deployed as plug-ins, i.e. DLL assembly files. A plug-in can contain several functionally related extensions that should be jointly installed. The discoverer is the part of Plux which discovers plug-ins and provides the metadata for the extensions in the plug-in. Plux supports dynamic discovery, i.e. plug-ins can be added and removed without restarting the application. The default discoverer reads the metadata from attributes stored in the plug-in assemblies. As the discoverer is an extension itself, one can write custom discoverers, e.g., to retrieve metadata from a database or from a configuration file.

2.3 Composition

Composition is the mediating process which matches the requirements of hosts with the provisions of contributors. In Plux, this is done by the composer. The composer assembles programs from the extensions provided by the discoverer. Thereby it connects the slots of hosts with the plugs of contributors.

When the discoverer provides a new extension, the composer integrates it into the program on-the-fly. Similarly, if an extension is removed from the plug-in repository, the composer removes it from the program.

Integrating an extension means, that the composer instantiates it and connects its plugs with the matching slots of extensions in the program. If a plug is connected to a slot, we call this relationship *plugged*. Removing an extension means that the composer unplugs the instances of this extension from the slots where they are plugged, i.e. it removes the plugged relationship for the corresponding slots and plugs.

Slots can declare whether they want an instance of their own or a shared instance of a contributor. The composer connects a new instance to slots that want their own instance and the same (shared) instance to slots that want the shared instance.

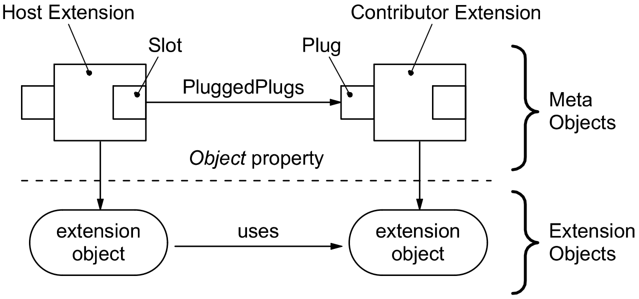


Fig. 5 Meta-objects for instantiated extensions in the composition state

2.4 Composition state

In Plux, all connections between components are established by the composer. Therefore the composer has full knowledge about the instantiated extensions, their slots and plugs as well as about their connections. This is called the *composition state*. If a host wants to use its plugged contributors, it can simply retrieve them from the composition state. For every instantiated extension, the composition state holds the *meta-object* of the extension, the meta-objects of its slots and plugs as well as a reference to the corresponding *extension object* (Fig. 5). For every slot, the composition state also indicates which plugs are connected to this slot.

Figure 6 describes the host of Fig. 4 in more detail showing how meta-objects can be used by a program. When the composer creates an extension it passes the extension's meta-object to the constructor. In Fig. 6, the constructor retrieves the meta-object of the slot *Logger* and starts a new thread. In the *Run* method, the host does its work and uses the connected loggers to print a message. It retrieves the loggers using the *PluggedPlugs* property of the logger slot. For each logger, it checks the logger kind using the parameter *Kind*. Finally, it retrieves the extension objects for loggers of the desired kind and prints the message.

2.5 Composition events

In addition to accessing the composition state, a host can listen to composition events. This is appropriate for hosts that want to react to added or removed contributors immediately, e.g., in order to show them in the user interface. Figure 7 shows a modified version of our host from Fig. 6. It uses the *Slot* attribute to register event handler methods for the *Plugged* and *Unplugged* events. In this example, the event handlers just print out which logger was plugged or unplugged.

```
[Extension]
[Plug(“Application”)]
[Slot(“Logger”)]
public class HostApp : IApplication{
  Slot s; // logger slot
  public HostApp(Extension e){
    s = e.Slots[“Logger”];
    new Thread(Run).Start();
  }
  void Run(){
    while(true){
      string msg; LoggerKind kind;
      Work(out msg, out kind);
      foreach(Plug p in s.PluggedPlugs){
        if((LoggerKind) p.Params[“Kind”]
            == kind){
          Extension e = p.Extension;
          ILogger logger = (ILogger)e.Object;
          logger.Print(msg);
        }
      }
      Thread.Sleep(2000);
    }
  }
  void Work(out string msg,
      out LoggerKind kind){
    /* not shown */
  }
}
```

Fig. 6 Application host using logger contributors

```
[Extension]
[Plug(“Application”)]
[Slot(“Logger”,
  OnPlugged=“Plugged”,
  OnUnplugged=“Unplugged”)]
public class HostApp : IApplication{
  ...
  void Plugged(CompositionEventArgs args){
    Extension e = args.Plug.Extension;
    ILogger logger = (ILogger) e.Object;
    logger.Print(“plugged: ” + e.Name);
  }
  void Unplugged(CompositionEventArgs args){
    ...
    logger.Print(“unplugged: ” + e.Name);
  }
  void Run(){...}
  void Work(...){...}
}
```

Fig. 7 Modified application reporting connected contributors

This completes the example. We compile the slot definition interface *ILogger* to a DLL file, the so-called *contract* assembly. Contracts and plug-ins should be separate DLL files, because bundling slot definitions with extensions would constrain customization. We could not use a slot definition in a program without also including the extensions that come with it. If we compile the classes *ErrorLogger* and *HostApp* to plug-in DLL files and drop them into the plug-in repository of Plux everything will fall into place. The Plux infrastructure will discover the extension *HostApp* and plug it into the *Application* slot of Plux. It will also discover the extension *ErrorLogger* and plug it into the *Logger* slot of *HostApp* (Fig. 8).

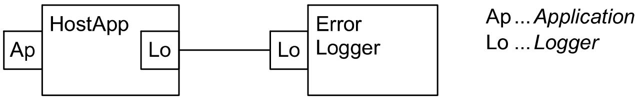


Fig. 8 Composed application with host and logger contributor

2.6 Lazy activation

In order to minimize startup time and memory usage, Plux supports lazy activation of extensions. This means that contributors are only instantiated on demand, i.e. when the host accesses the extension's *Object* property. For the example in Fig. 6 this has the effect that loggers which do not match the desired kind are not instantiated; only their meta-objects exist.

If a contributor is no longer needed and the host wants to release the resources used by it, the host can deactivate the contributor. To deactivate means to free the extension object, so that only its meta-object remains. On the next access to the *Object* property, the contributor is automatically reactivated.

Hosts can listen to the composition events *Activated* and *Deactivated* if they want to distinguish between activated and deactivated contributors. Typically such a host cooperates with another host. The first host handles only activated contributors while the other one controls which contributors get activated. For example, a window host might show a child window for every activated contributor, while a menu host might allow the user to activate and deactivate contributors causing child windows to be opened or closed.

2.7 Selection

The composition infrastructure allows selecting plugged contributors in a slot. This is useful if hosts cooperate in such a way that one host uses only selected contributors and another hosts controls which contributors are selected. Applied to the logger host from Fig. 6 this could mean that the host only accesses selected loggers, while another host allows the user to select or deselect contributors, e.g., by checking or unchecking them in a list. Figure 9 shows the modified *Run* method of the logger host, which works with the selected loggers only.

2.8 Programmatic composition

The mechanism described in the previous sections, where the composer makes connections and extensions retrieve connections, is called automatic composition. In addition to that, hosts can assemble contributors using programmatic composition, i.e. the host can control how the composer assembles the program. For example, the host can use API calls to integrate specific contributors, a script interpreter can assemble a program from a script, or a serializer can restore a previously saved program.

```
void Run() {
  while(true) {
    string msg; LoggerKind kind;
    Work(out msg, out kind);
    foreach(Plug p in s.SelectedPlugs) {
      if((LoggerKind) p.Params[“Kind”]
          == kind) {
        Extension e = p.Extension;
        ILogger logger = (ILogger) e.Object;
        logger.Print(msg);
      }
    }
    Thread.Sleep(2000);
  }
}
```

Fig. 9 Run method of application host with contributor selection

2.9 More features

Other features of Plux that cannot be discussed at length here are the management of *composition rights* (e.g., which extensions are allowed to open a certain slot, and which extensions are allowed to fill it), *slot behaviors* that allow developers to specify how slots behave during the composition (e.g., one can limit the capacity of a slot to *n* contributors, or automatically remove a contributor from a slot when a new contributor is plugged), *component templates* to define generic extensions, that can be reused with different metadata in different parts of the program, as well as a *scripting API* that allows experienced users to override the operations of the composer. For a more extensive description of these features see [11, 21, 22].

2.10 Summary

In Sect. 2 we described how Plux assembles programs from extensions. The extensions use metadata to declare their requirements and provisions. The discoverer discovers plug-ins and provides metadata for the extensions in the plug-ins. The composer matches requirements and provisions; it assembles a program by plugging plugs of contributors into slots of hosts. Hosts retrieve the composition state and react to events sent by the composer. Plux instantiates contributors lazily, i.e., the associated extension object is only instantiated when the first host accesses it. Contributors which are no longer needed can be deactivated in order to release their resources. As contributors of a host can be selected, hosts can consider this selection, thus allowing the user to switch between contributors. For scenarios where automatic composition does not yield the desired result, programmers can use the API of the composer to influence how the program is assembled.

Figure 10 explains the graphical notation for the composition state which we use in the rest of this paper. The verbose notation shows both the extension objects and their

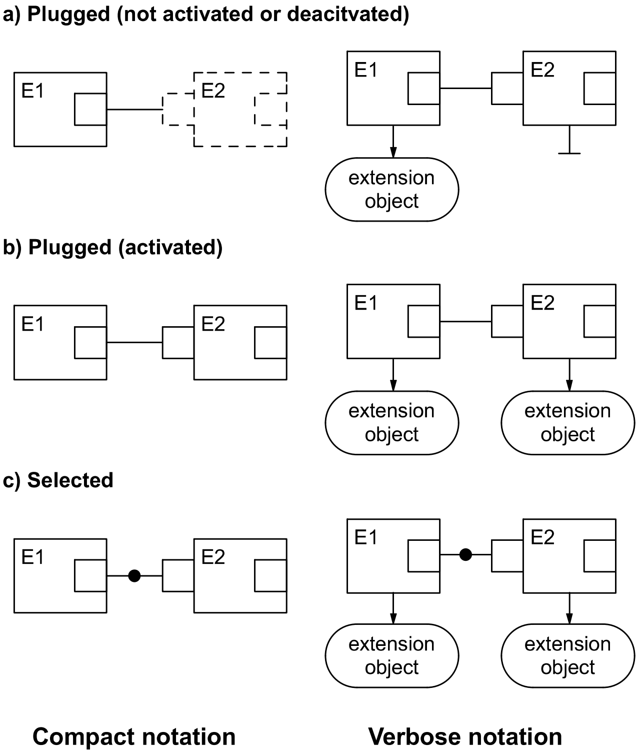


Fig. 10 Compact and verbose notation for composition relationships

meta-objects (as introduced in Fig. 5). For clarity, however, we use the compact notation wherever possible.

Figure 10a shows a contributor plugged into a host, where the contributor has not yet been activated or has been deactivated. Figure 10b shows the same host and contributor, but this time the contributor is activated. Figure 10c shows the notation for a selected contributor. Note that contributors can also be selected when they are not activated (not shown).

3 Composition infrastructure of Plux

The composition infrastructure builds programs from contracts and plug-ins. It discovers extensions from a plug-in repository and composes the program from them by connecting matching slots and plugs. The plug-in repository is typically a directory in the file system containing contract DLL files (with slot definitions) and plug-in DLL files (with extensions).

3.1 Architecture

Figure 11 explains the subsystems of the composition infrastructure and how they interact. The *discoverer* ensures that at any time the type store contains the metadata of extensions and slot definitions from the plug-in repository. When the discoverer detects an addition to the repository, it extracts the metadata from the DLL file and adds them to the type store. Vice-versa, when it detects a removal from the repository, it removes the corresponding metadata from the type store. The discoverer is implemented as an extension itself. Thus, it can be replaced with or extended by other discoverers.

The *type store* maintains the metadata of slot definitions and extensions which are available for composition and notifies the composer about changes. When new metadata become available or when metadata are removed, the composer updates the program. In addition to that, the type store can be queried for contributors, e.g., by the composer when it tries to fill slots.

The composer assembles a program by matching requirements and provisions. It listens to changes in the type store and updates the program accordingly. If extensions become available or unavailable, it integrates or removes them and updates the composition state held in the instance store.

The *instance store* maintains the composition state of a program, i.e. the meta-objects of extensions, slots and plugs as well as the relationships between them. The instance store is also used by other tools which, for example, visualize the composition state and its changes during run time. Plux includes a visualizer tool which uses a notation similar to the one in the figures of this paper.

3.2 Composition process

The composition process is directed by the composer. On changes in the type store, the composer updates the program by matching slots and plugs. If a contributor becomes available in the type store, the composer queries the instance store for matching slots. A plug matches a slot if their names match. The composer plugs a matching plug into a slot if the slot definition is available, the plug implements the interface of the slot definition, and the plug provides values for the parameters declared by the slot definition. To plug a contributor means to instantiate it, add it to the instance store, and add a *plugged* relationship between the host and the contributor to the instance store. As the composer now treats the contributor itself as a host, it opens its slots and fills them with other contributors. In that way the composer continues the composition until all qualifying extensions are assembled.

Vice-versa, if a contributor is removed from the type store, the composer queries the instance store for relationships containing the contributor's plugs. If it finds such relationships, it unplugs the contributor's instance. To unplug a contributor instance means to close its slots, to remove the *plugged* relationship from the instance store, to remove the contributor instance from the instance store, and to release it. Closing the contributor's slots causes the decomposition to be propagated, i.e. all contributors are unplugged from those slots as well.

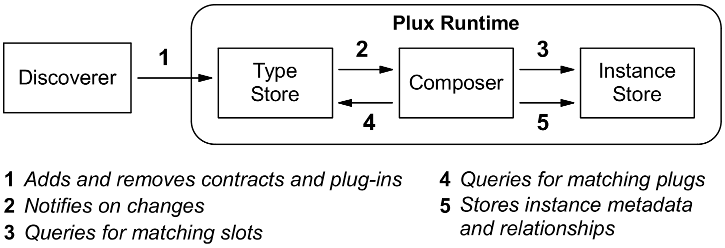

Plux Runtime
Discoverer
1
Type Store
2
4
Composer
3
5
Instance Store
1 Adds and removes contracts and plug-ins
2 Notifies on changes
3 Queries for matching slots
4 Queries for matching plugs
5 Stores instance metadata and relationships

Fig. 11 Architecture of the composition infrastructure

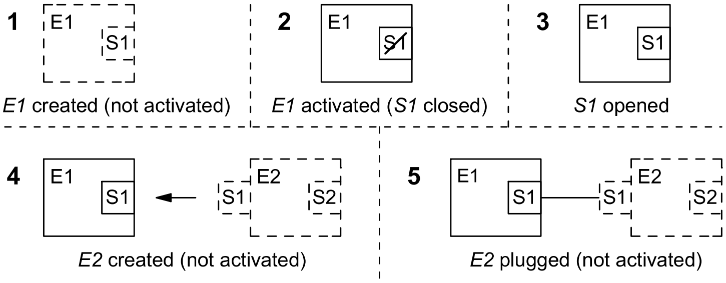

1
E1
S1
E1 created (not activated)
2
E1
S1
E1 activated (S1 closed)
3
E1
S1
S1 opened
4
E1
S1
E2
S1
S2
E2 created (not activated)
5
E1
S1
E2
S1
S2
E2 plugged (not activated)

Fig. 12 Steps of the Plux composition process

Figure 12 shows the steps performed by the composer when it activates a host and fills its slots. In Fig. 12.1, the extension *E1* is plugged into some host (not shown) but is not activated. In Fig. 12.2, the host of *E1* accesses the *Object* property of *E1*. Thus, the composer activates *E1*, i.e. it instantiates the associated extension object. As part of the activation, the composer opens the slot *S1* of *E1* (Fig. 12.3). Opening *S1* causes the type store to be queried for plugs matching *S1*. In our example, the composer finds *E2* with the plug *S1*. It creates an instance of *E2* and plugs it into *S1* of *E1* (Fig. 12.4). In Fig. 12.5 the composition is completed and *E2* is in same state as *E1* in Fig. 12.1.

3.3 Runtime

The Plux runtime implements the composition infrastructure. To start a Plux program, one has to launch the runtime and provide a discoverer as well as the plug-in repository with the components that make up the program. At startup, the runtime activates its built-in *Startup* extension, which is the root for the Plux program. It has two slots: one for discoverers and one for applications.

To start the logger application from Sect. 2, we put the *Logger* contract, the *HostApp* plug-in, and the *ErrorLogger* plug-in into the plug-in repository, which is a file system folder. When we launch the runtime, we pass the folder and a file system discoverer as command-line arguments. The composer plugs the discoverer into the *Discoverer* slot of the *Startup* extension (Fig. 13). The *Startup* extension activates the discoverer, which discovers the contracts and the plug-ins from the provided folder. After the *HostApp* extension has become available in the type store, the composer plugs it into the *Application* slot of the *Startup* extension. The *Startup* extension activates *HostApp*, whereon the composer fills *HostApp*'s *Logger* slot, i.e., it plugs the *ErrorLogger* extension.

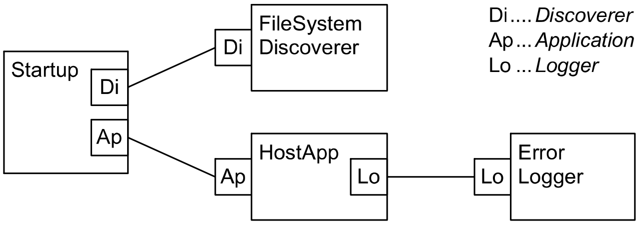


Fig. 13 Plux runtime with startup extension and composed logger application

4 Enabling Plux for building web applications

Web applications face similar problems as desktop applications: If they get big and feature-rich, they become hard to understand and difficult to maintain. They are hardly customizable and usually not extensible by end users. Furthermore, web applications cannot access the local hardware of client computers. In order to solve these problems we applied the plug-in approach also to web-based software. While the original version of Plux targeted single-user desktop applications, we enhanced it so that it can be used to build multi-user web applications.

Plux makes web applications extensible. Extensions can either be installed by the administrator or by the end user. Authorized users can install them directly on the server, while non-authorized users can install them on the client. Regardless of where an extension is installed, it is always seamlessly integrated into the web application.

Plux allows setting different user scopes. Extensions can be made available for *all users* of a web application, for a

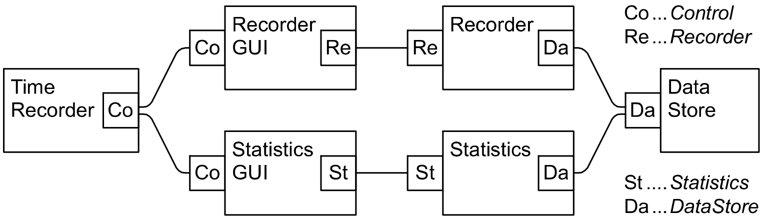


Fig. 14 Base composition of the time recorder web application

group of users, or for a *single user*. Thus every user can have an individual set of components, i.e., an individual composition state.

Depending on their type of integration, extensions are classified into three categories: (a) *Server-side* extensions are installed and executed on the server. (b) *Client-side* extensions are installed and executed on the client. (c) *Sandbox* extensions are installed on the server, but executed in a sandbox on the client. Regardless of where the extensions are executed, Plux composes them into a coherent web application giving the user a seamless experience.

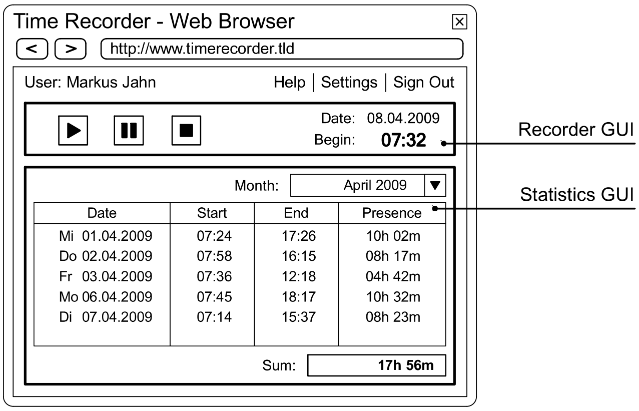
In this section, we describe several usage scenarios demonstrating the need for extensible web applications. As a running example we use a time recorder as a case study. The usage scenarios cover the different user scopes as well as the different types of extension integration.

Our time recorder can be used to record and evaluate working hours. Figure 14 shows its architecture. Every feature of the time recorder is implemented as an extension, and the business logic is separated from the user interface. The *Data Store* extension stores arbitrary data for its hosts, e.g., working time data for the *Recorder* extension. The *Recorder* extension allows a user to log the start time and the end time of his work. The *Statistics* extension computes working time metrics. *Recorder* and *Statistics* are plugged into the user interface, i.e., into the *Recorder GUI* and the *Statistics GUI*, which are plugged into the *Time Recorder* host. In order to keep the figures simple, extensions that are irrelevant for the usage scenarios are hidden in Figs. 14–20. For example, the startup extension and the discoverer extension of Plux are not shown.

Figure 15 shows the user interface of the time recorder. The *Recorder GUI* displays the current date and time, and allows the user to start, stop, and pause the recording. The *Statistics GUI* shows the recorded working hours for the selected month.

As the time recorder is extensible, the user interface must be extensible as well. Control contributors, such as *Recorder GUI* and *Statistics GUI*, declare their desired positions in their metadata. The layout manager of the time recorder retrieves these positions and arranges the contributors accordingly.

The composition in Fig. 14 is the base configuration available to all users. As all these extensions are server-side extensions, they are installed and executed on the server. In the following subsections, we show how users can extend this web application with user-specific extensions. We will describe the different types of extension integration and explain in which scenarios they are suitable.


Time Recorder - Web Browser
☒
<
>
http://www.timerecorder.tld
Help | Settings | Sign Out
▶
❚❚
■
Recorder GUI
Statistics GUI
▼

User: Markus Jahn

Date: 08.04.2009
Begin: **07:32**

Month: April 2009

Date	Start	End	Presence
Mi 01.04.2009	07:24	17:26	10h 02m
Do 02.04.2009	07:58	16:15	08h 17m
Fr 03.04.2009	07:36	12:18	04h 42m
Mo 06.04.2009	07:45	18:17	10h 32m
Di 07.04.2009	07:14	15:37	08h 23m

Sum: **17h 56m**

Fig. 15 User interface of the time recorder web application

4.1 Server-side extensions

Let us assume that the time recorder is used in a company with three groups. Group A wants an extension that allows users to add notes to recorded time stamps. Therefore they develop a note plug-in, consisting of a *Notes* extension, which stores notes for the time stamps in the data store, and a *Notes GUI* extension, which can be used to edit a note from the *Notes* extension. As the users of group A want to access the note plug-in from any computer, it is installed on the server. And because only members of group A should see the note plug-in, its user scope is set to this group (user scopes are covered in Sect. 5.1). The composition for group A is shown in Fig. 16. It comprises the base composition (Fig. 14) extended by the note plug-in.

The server-side extensions are installed and executed on the server. Because they are executed on the same server as the time recorder, there is no performance penalty caused by remote communication and they are available regardless from which computer the user connects. However, server-side extensions increase the work load on the server and may execute malicious code. For that reason, users typically need to be authorized to install extensions on the server.

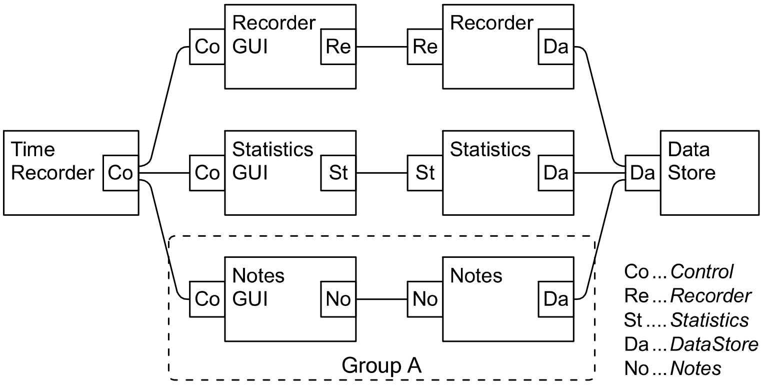


Fig. 16 Composition for Group A comprising the base composition extended by the server-side extensions for notes

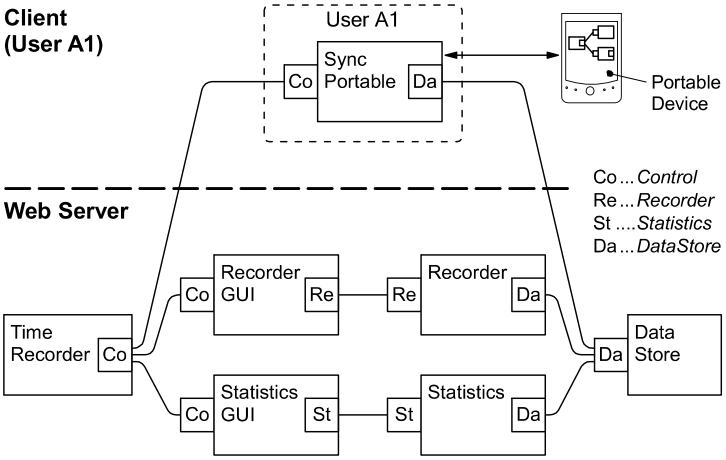


Fig. 17 Extending the time recorder web application with a client-side extension in the scope of User A

4.2 Client-side extensions for single users

Assume that user A1 is an engineer in the field. He wants to track his working hours using a portable device. Because his device cannot connect to the Internet, he periodically has to synchronize it with the time recorder.

To synchronize the device, user A1 connects it to his office computer where he has installed the client-side extension *Sync Portable*. Because this extension is executed on the client computer, it can access the portable device there. Figure 17 shows the composition for user A1. To synchronize the data between the device and the time recorder, the *Data Store* extension is also plugged to the *Sync Portable* extension.

Client-side extensions are installed on the client and are remotely plugged into the web application. To plug remotely means that the host and the contributor are executed on different computers. Plux creates proxies on both sides on-the-fly. These proxies handle the communication between the host and the contributor transparently, i.e., Plux allows every extension to run remotely without any special coding effort.

Client-side extensions allow users to build components that integrate local hardware or software into the web application. Furthermore, since client-side extensions are installed on client computers, they open the web application also for extensions of users who are not authorized to install extensions on the server. The disadvantage of client-side extensions is that the remote execution of extensions causes some communication overhead.

4.3 Client-side extensions for a group of users

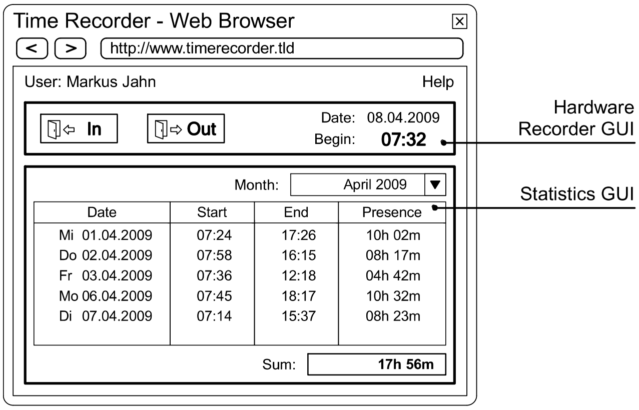
Now we look at a scenario where a group B of users wants to use the same set of client-side extensions, even if these extensions are running on different computers.

Assume that the users in a group B use a hardware time clock to track their working hours. The time clock is connected to a dedicated computer on which the *Hardware Recorder GUI* is installed as a client-side extension, which integrates the time clock into the time recorder.

The user scope of the *Hardware Recorder GUI* is set to group B. Thus, if a user of group B uses the time recorder from any computer (e.g., to check his statistics) he will see the user interface of the *Hardware Recorder GUI* (Fig. 18). If a user inserts his id card into the time clock, the user interface allows him to press the *In* and *Out* controls. As long as no id card is inserted, the controls are disabled. Note that the users of group B get this user interface on any computer, but since only the dedicated time clock computer has the time clock connected, the *In* and *Out* controls stay disabled on other computers.

Since users of group B must use the time clock instead of the *Recorder GUI*, a configuration file on the server (covered in Sect. 5.1) configures Plux such that the *Recorder GUI* is excluded from the composition for users in group B (Fig. 19). In other words, the server-side *Recorder GUI* is replaced by the client-side *Hardware Recorder GUI*, which uses the remotely plugged *Recorder*.

The hardware recorder extension in this scenario is made available to multiple users; this might suggest a server-side extension. However, we need to use a client-side extension here, because the extension accesses local hardware. Since the hardware recorder is set up for group scope, it is integrated into the time recorder application for every user of group B. Thus, a user from group B can access the hardware recorder from a different computer that the one on which it is executed. Client-side extensions are always executed on the computer on which they are installed. One might wonder, how the client-side extension *Hardware Recorder GUI* can contribute to the user interface. As the user interface is described in HTML, the client-side extension contributes its HTML parts to its host extension *Time Recorder*. The *Time Recorder* arranges the HTML parts from all its contributors into the final user interface.



Time Recorder - Web Browser ☒

< > http://www.timerecorder.tld

User: Markus Jahn Help

In Out Date: 08.04.2009 Begin: 07:32

Month: April 2009 ▼

Date	Start	End	Presence
Mi 01.04.2009	07:24	17:26	10h 02m
Do 02.04.2009	07:58	16:15	08h 17m
Fr 03.04.2009	07:36	12:18	04h 42m
Mo 06.04.2009	07:45	18:17	10h 32m
Di 07.04.2009	07:14	15:37	08h 23m

Sum: 17h 56m

Fig. 18 Modified time recorder user interface of the *Hardware Recorder GUI* extension

Another possible reason for using client-side extensions with group scope instead of a server-side extension is authorization. Even if one is not authorized to install an extension on the server, one can make it available to multiple users as a client-side extension with group scope.

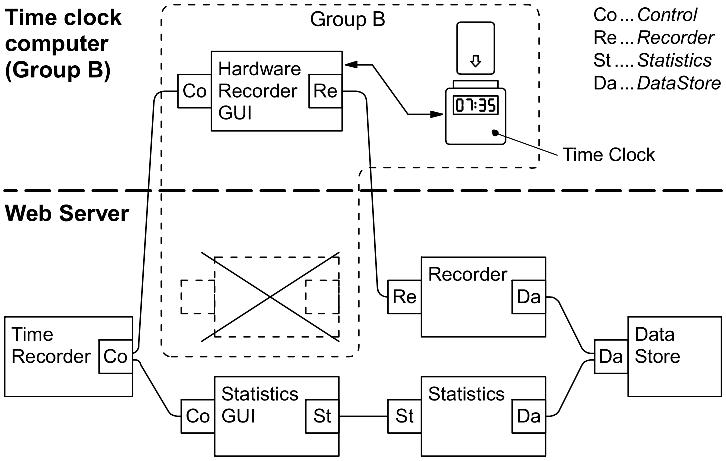
4.4 Sandbox extensions

For the next scenario, assume that users of a group C want a richer user interface, e.g. one that was built with Silverlight [19] instead of HTML. Silverlight code runs in a sandbox within the web browser of the client, so the best way to integrate such code into a Plux application is to implement Silverlight components as extensions that reside on the server but are downloaded to the web browser on demand and are executed there.

We implement the Silverlight extensions *Recorder SL GUI* and *Statistics SL GUI* as replacements for *Recorder GUI* and *Statistics GUI* and install them on the server. They are discovered there as sandbox extensions and are configured with the scope for group C. We exclude the original *Recorder GUI* and *Statistics GUI* for group C using the configuration file on the server (see Sect. 5.1). When a member of group C starts the time recorder, the sandbox extensions are downloaded from the server to the client computer and are executed in the Silverlight environment. The business logic extensions on the server are remotely plugged into the Silverlight user interface extensions on the client. The composition for users of group C is shown in Fig. 20.

Like client-side extensions, the sandbox extensions in this scenario are executed on the client, but unlike client-side extensions, they are installed on the server and downloaded to the client on demand. On the client, they are executed in a sandbox, e.g., in the Silverlight environment. This integration type is useful for building rich user interfaces for web

Fig. 19 Replacing the *Recorder GUI* extension by the *Hardware Recorder GUI* extension executed on another computer



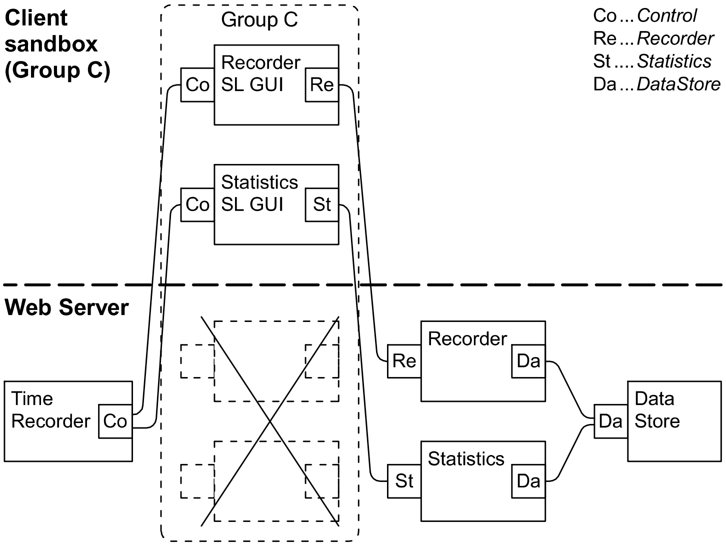


Fig. 20 Sandbox extensions are installed on the server, but transferred to the client on demand and executed in a client-side sandbox

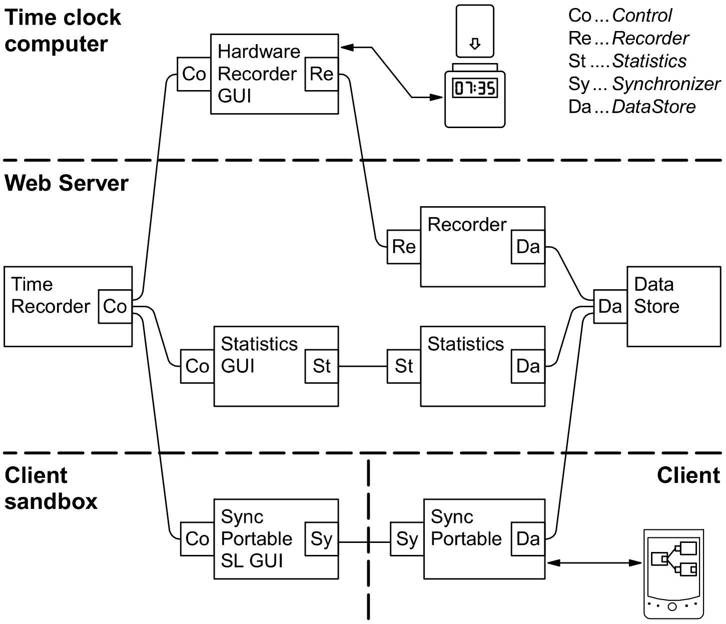


Fig. 21 The component architecture for a specific user composed by server-side, client-side and sandbox extensions

applications. Because sandbox extensions are executed on the client, they reduce the work load on the server.

4.5 Concluding example

Finally, assume that user B1 wants to use the portable device when he is in the field, as well as the time clock when he is on site. The customized time recorder for user B1 uses all integration types presented in this section. As user B1 is a member of group B, the base composition built from the server-side extensions is extended by the client-side extension *Hardware Recorder GUI* (with group scope). Furthermore, because he uses the portable device, he installed the client-side extension *Sync Portable* (with user scope) on his computer, as well as the *Sync Portable GUI* as a sandbox extension on the server (Fig. 21).

The composition is distributed over three different computers: the web server, the time clock computer, and B1's client computer. The *Sync Portable GUI* is installed on the web server, downloaded to every client computer, and executed in the Silverlight sandbox there. As *Sync Portable* and *Sync Portable GUI* run both on the same client computer, the communication between them does not affect the server.

Server-side, client-side and sandbox extensions are implemented in exactly the same way. The different modes in which they are composed and executed are managed by Plux. Therefore, a server-side extension of one web application can be reused as a client-side extension in some other web application and vice versa. The only exception are client-side extensions that use local devices: these extensions have to run on the client on which this device is installed.

This section showed scenarios for the integration of server-side and client-side extensions into a coherent web application. The next section explains the architecture of the Plux web runtime and how it implements the support for the described scenarios.

5 Plux web runtime

The web runtime of Plux provides the infrastructure for running plug-in-based web applications like the ones that were shown in Sect. 4. It extends the Plux desktop runtime with the following features:

Multi-user support. For every user the web runtime maintains an individual set of plug-ins and an individual composition state.

Distribution support. The web runtime can compose a web application from extensions running on different computers by plugging them remotely. It provides a distributed composition infrastructure that maintains the composition state of a distributed web application.

5.1 Multi-user support

This section describes the infrastructure necessary to support multiple users with different sets of server-side extensions on a single web server. The handling of client-side extensions and distribution is shown in Sec. 5.2.

The web runtime supports multiple users by maintaining one *runtime node* per user on the web server. A runtime node comprises the infrastructure necessary to compose the web application for the corresponding user, i.e. a type store, an instance store, and a composer (Fig. 22). The type store maintains the type metadata for the user's extensions. The instance store maintains the user's composition state. The instances in a user's composition state are isolated from the composition states of other users, i.e., instances are not shared among users.

The composer is bound to the type store as well as to the instance store of a user. It assembles the extensions from the type store and stores the composition state in the instance store. For reasons of responsiveness, every runtime node is executed in a separate thread so that the composition for multiple users occurs concurrently.

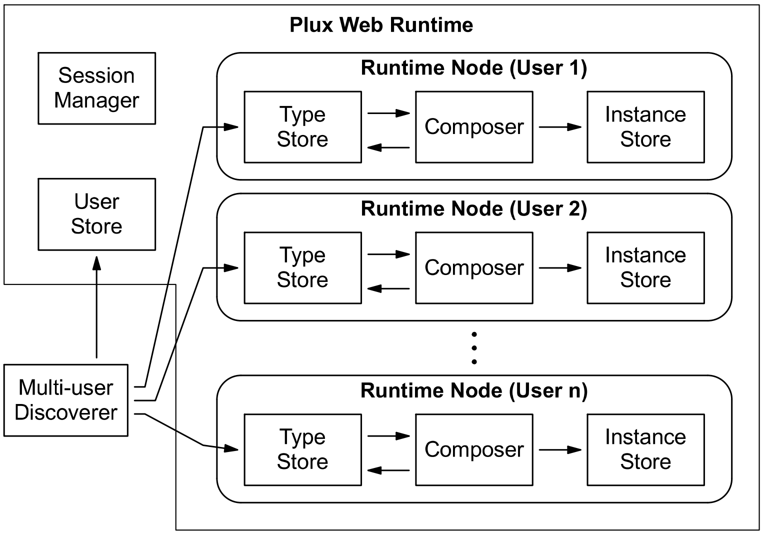
The server runs a *session manager* which creates a new runtime node when a user session begins, and releases the node when the session ends. During a session it reuses the runtime node for every request of this user.

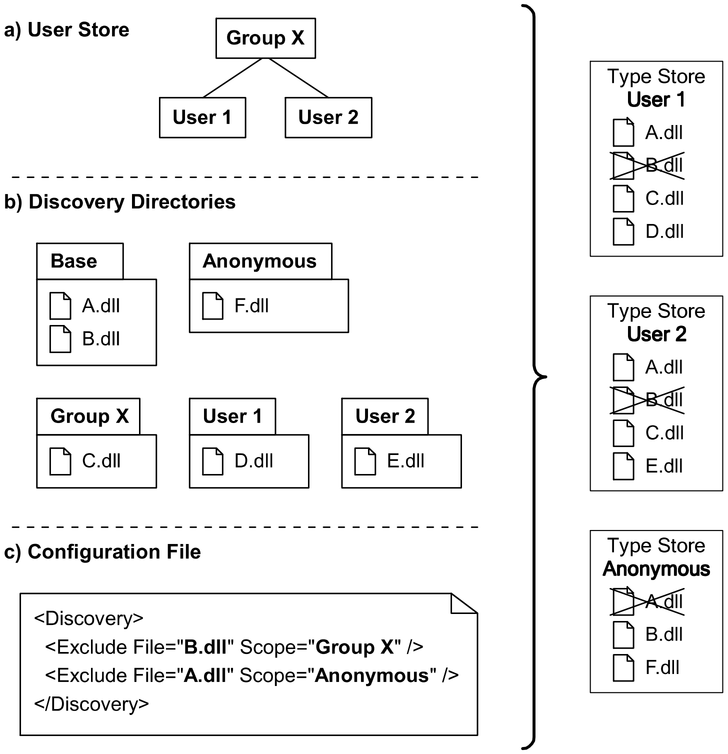
Users can be hierarchically organized into groups. A user can be a member of multiple groups and a group can contain multiple users and groups. These membership relations are maintained in a *user store* (Fig. 23a). The default user store uses the .NET Membership API. However, it can be replaced by a different mechanism.

Server-side extensions are kept in different directories on the server. For every user and for every group there is a directory with the user or group name, which contains the extensions that are specific to this user or group. In addition to that, there is a *Base* directory containing the extensions for all users as well as an *Anonymous* directory containing extensions for unauthenticated users (Fig. 23b).

The user's identity and his group memberships determine which plug-ins are available for him. For example, user 1 gets the plug-ins *A.dll* and *B.dll* from the *Base* directory, *C.dll* from the directory of group X, and *D.dll* from his own user directory. In other words, he inherits the plug-ins from *Base* and *Group X* and adds his own plug-ins.

Fig. 22 Architecture of the multi-user composition infrastructure





```
<Discovery>
  <Exclude File="B.dll" Scope="Group X" />
  <Exclude File="A.dll" Scope="Anonymous" />
</Discovery>
```

Fig. 23 The multi-user discoverer uses the user store, the discovery directories, and an optional configuration file to populate the users' type stores

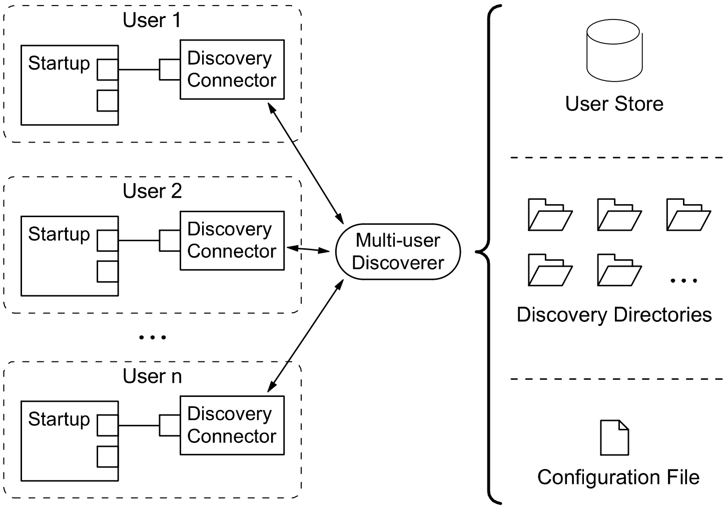


Fig. 24 Integration of the shared multi-user discoverer with the individual runtime nodes using discovery connectors

Sometimes it is also necessary to exclude a plug-in from the inherited set of plug-ins. This is specified with a global *configuration file* like the one in Fig. 23c, which excludes *B.dll* for members of group X as well as *A.dll* for unauthenticated users.

In order to find out which extensions are available for a specific user, the discovery mechanism of Plux had to be extended. The *multi-user discoverer* retrieves group memberships from the user store, discovers plug-ins from the respective directories, excludes the plug-ins specified in the configuration file, and inserts the resulting set of plug-ins into the user's type store. This is the way how the user scopes from Sect. 4 are implemented.

The multi-user discoverer is integrated into every runtime node using *discovery connectors*, which are implemented by a plug-in that is specified as an initial plug-in when the web runtime is started. As a result, a discovery connector is inserted into the type store of every runtime node. When a runtime node instantiates its startup extension the composer plugs a discovery connector into the discovery slot of this startup extension. The discovery connectors communicate with the multi-user discoverer, which is shared for all users (Fig. 24). The multi-user discoverer monitors the user store,

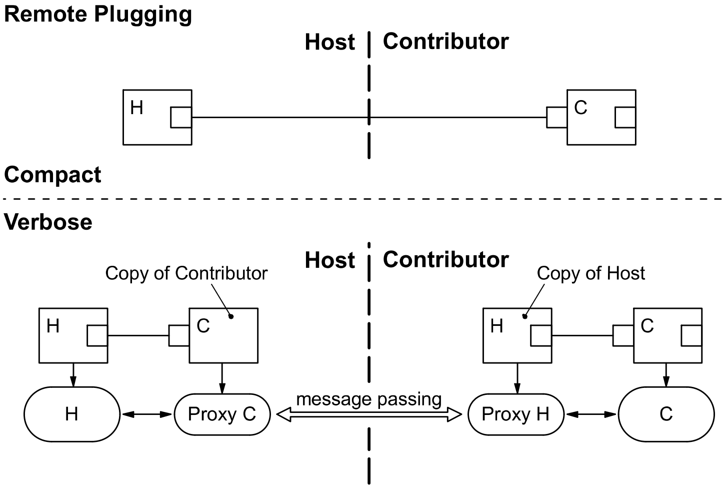


Fig. 25 Compact and verbose notation for remotely plugged extensions

the plug-in directories, and the configuration file and notifies the discovery connectors on changes so that they can update their type stores.

5.2 Distribution support

Distribution support allows the web runtime to execute the extensions on different computers and still to compose a coherent application from them. The web runtime supports distribution by remotely plugging extensions and by synchronizing the type stores and instance stores across multiple runtime nodes. For every user there is a runtime node in the web runtime and a runtime node in each client runtime (a user has multiple client runtimes if he uses client-side extensions from multiple computers). We implemented client runtimes in different flavours, namely as plug-ins for Mozilla Firefox and Microsoft Internet Explorer, as a Silverlight application for rich client applications, and as a standalone application, e.g., for the headless time clock computer in Sect. 4.

Remotely plugged extensions communicate via *extension proxies* on both sides of the line. An extension proxy consists of a proxy object, which communicates with the remote proxy, and a copy of the communication partner's meta-object, which represents the remote extension in the local instance store. The web runtime dynamically creates a contributor proxy on the host's runtime node, and a host proxy on the contributor's runtime node. These two proxies handle the remote communication, such that the host and the contributor need not care about distribution.

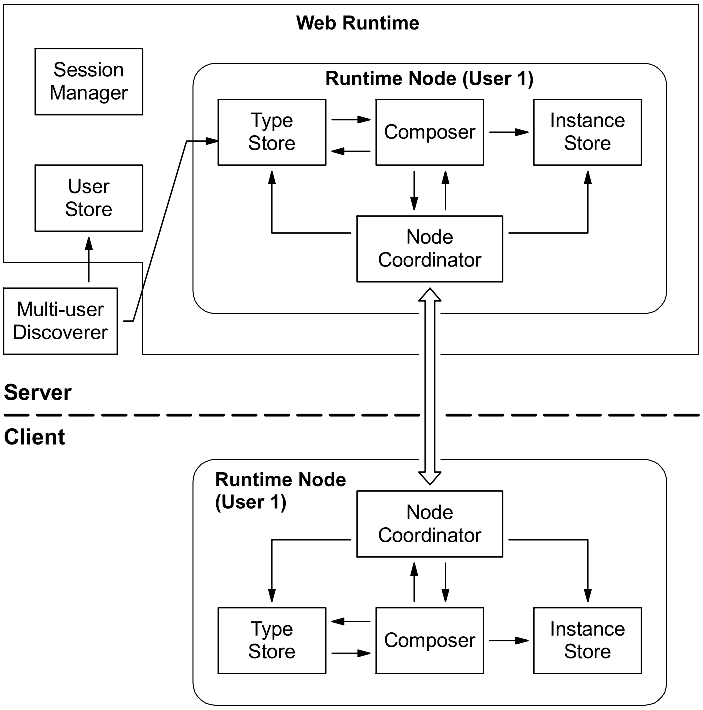
Figure 25 shows a verbose and a compact notation for remotely plugged extensions. For clarity, we use the compact notation wherever possible in this paper. When the host H wants to call a method of the contributor C, it calls the corresponding method of the host-side *Proxy* C, which sends the call to contributor-side *Proxy* H. On the contributor-side, *Proxy* H calls the method of the contributor C. The results are sent back in the same way.

Every client as well as every Silverlight environment has its own runtime node with a replicated copy of the user's server-side composition state, i.e., the user's type store and instance store. In order to synchronize the composition states in the runtime nodes of the same user, the web runtime combines them into a common *user runtime*. Figure 26 shows the user runtime of user 1 with two runtime nodes that are connected via *node coordinators*. The node coordinators synchronize the type stores and the instance stores, coordinate the composers, and provide a communication API for the proxies.

For consistency reasons, only one node per user runtime can be active at a time. In order to ensure this, the connected runtime nodes pass a token between each other. Only the node with the token can be active. Composition operations on this node do not lead to an immediate update of the other nodes. Only when the token is passed to some other node of the same user runtime, this node gets updated, i.e., it synchronizes its type store and its instance store with the data from the node that released the token. The node coordinator passes the token along when control is transferred to an extension that is executed on some other runtime node. Furthermore, a node coordinator can request the token with a broadcast to the connected runtime nodes. This happens, for example, when a user interaction initiates a composition operation on a runtime node that does not have the token.

Figure 27 shows the steps when a client-side contributor C is remotely plugged into a server-side host H. In Fig. 27.1 the server runtime node is active and the slot of H is closed. In Fig. 27.2 the server-side composer opened the slot of H and plugged C into H. This can be done locally on the server, because the server type store is synchronized with the client type store and therefore holds a local copy of C's metadata. When H tries to access the extension object of C in Fig. 27.3, the server composer needs to activate C. As C

Fig. 26 Distributed user runtime with interconnected runtime nodes



is to be executed on the client, the server coordinator passes the token to the client. Please note that before Fig. 27.3 the composition state was not replicated to the client, because the server runtime node involves a client runtime node only, when a contributor of the client is accessed. When the client receives the token, it updates its type store and its instance store, i.e., the composition state of the server is replicated on the client. The client composer also creates a proxy object for H. In Fig. 27.4 the client composer activates C. Then the token is passed back to the server (Fig. 27.5) and the server composer finally creates a proxy object for C. This concludes the remote plugging process. The server-side host H can now use the client-side contributor C.

As the composers on the different runtime nodes need to know on which node an extension should be executed, the discoverer must annotate the extension types with this information. The discoverer has to find out whether a plug-in is a server-side plug-in, a client-side plug-in, or a sandbox plug-in. To distinguish server-side plug-ins from sandbox plug-ins, they are installed in separate directories on the server. The multi-user discoverer annotates the types from these plug-ins accordingly. Client-side plug-ins are discovered by discoverers on the client.

Figure 28 shows an example of how the type stores are populated. For user 1, who is a member of group X, the server-side plug-ins *A.dll* and *B.dll* are retrieved from the *Base* directory, *C.dll* from the *Group X* directory, and *D.dll* from the directory *User 1*. Furthermore, the sandbox plug-ins *G.dll* and *H.dll* are retrieved from the sandbox directories *Group X* and *User 1*. Finally, the client-side plug-in *I.dll* is retrieved from the *Plugin* directory on the client computer of user 1. The plug-in *B.dll* is excluded as specified in the configuration file. When the type store for user 1 is built, the plug-ins are classified as server plug-ins, client plug-ins, and sandbox plug-ins. The type stores for user 2 and for unauthenticated users are built accordingly.

Figure 29 shows how server-side and client-side discoverers are integrated into a web application. On the server side, a discovery connector communicating with the multi-user discoverer is plugged into the discovery slot of the *Startup* extension in every user runtime. The multi-user discoverer retrieves server-side plug-ins as well as sandbox plug-ins and delivers them to the *Startup* extension, which populates the type stores of the user runtimes. Client-side discoverers are remotely plugged into the same discovery slot of the *Startup* extensions. They discover client-side plug-ins and deliver them to the *Startup* extensions.

Installing plug-ins for a web application means to drop files into a plug-in directory where they will be discovered. Users can easily put plug-ins into the plug-in directory on their client. However, if they want to install server-side plug-ins or sandbox plug-ins, they need to put them on the server. Administrators typically want to prevent users from directly accessing directories on the server. Thus, Plux provides an API for server-side plug-in installation as an alternative. Web applications can use this API to provide a user interface which allows users to install server-side plug-ins. Administrators can control which user is allowed to install server-side plug-ins by setting permissions in the user store.

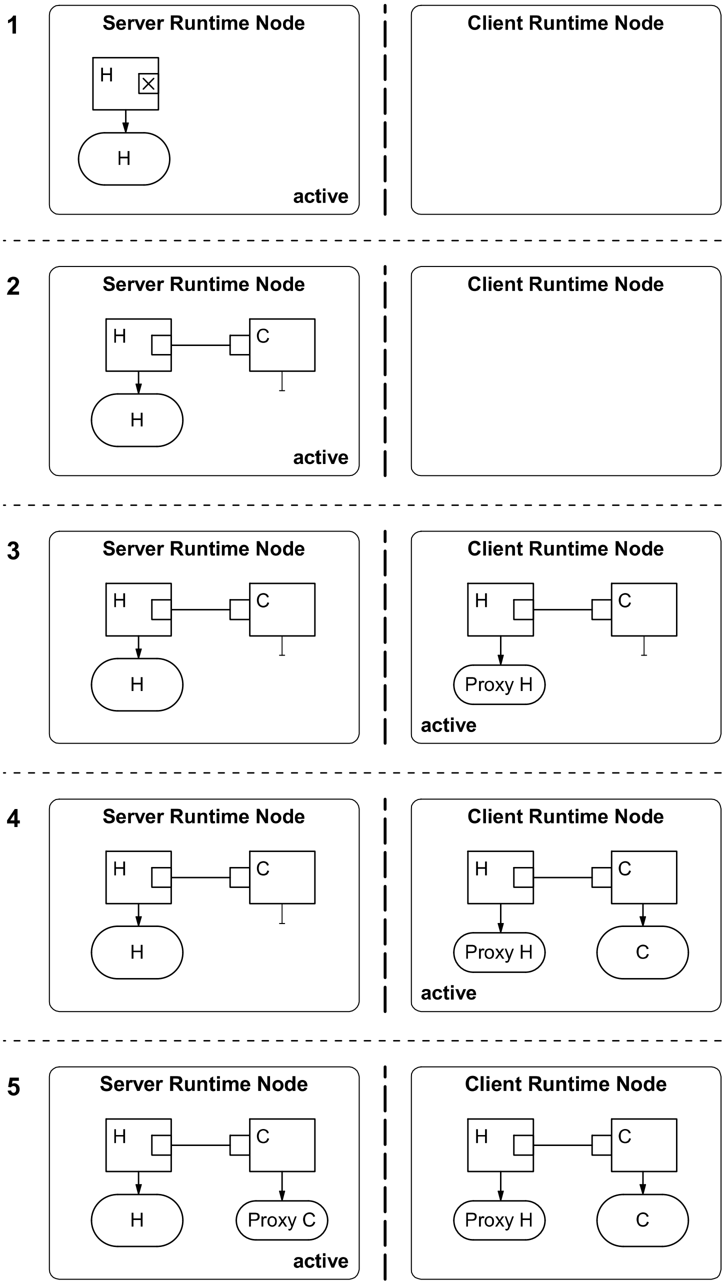


Fig. 27 Steps of the remote plugging process

Plux includes an upload tool that demonstrates the usage of this API.

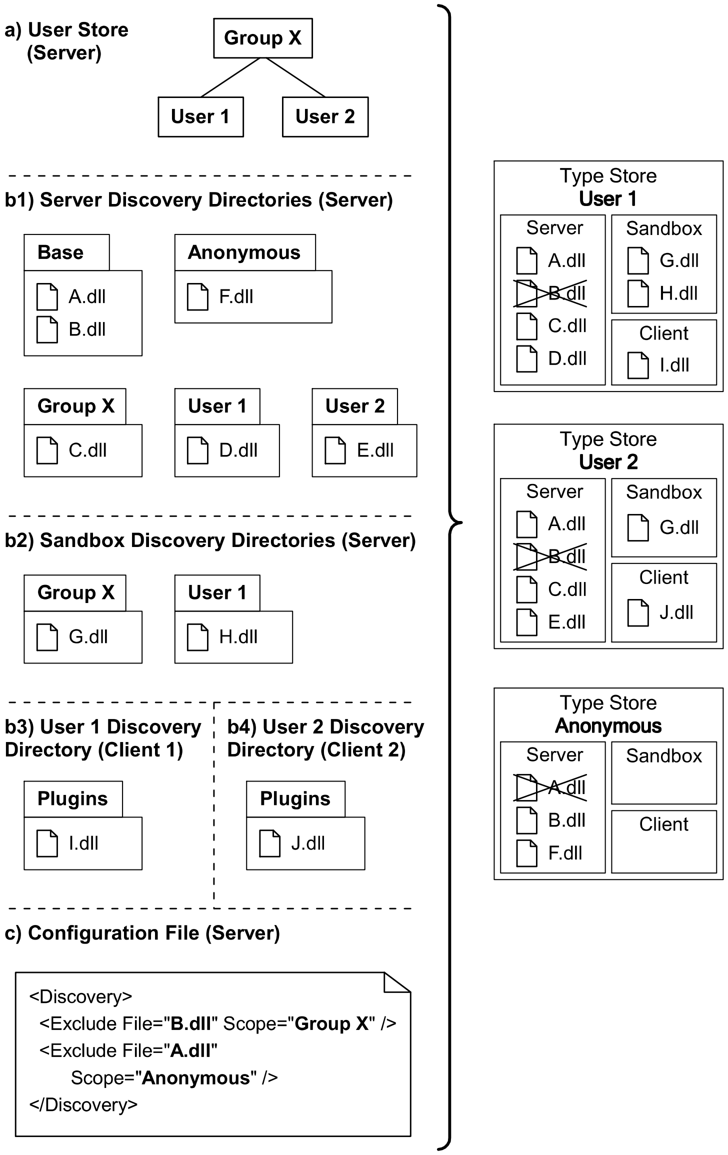
6 Hosting Plux web applications

Web applications built with Plux are hosted on the Microsoft Internet Information Server and ASP.NET [1]. An ASP.NET web application is stored in a virtual directory that contains web pages, library assemblies and further optional resources (e.g., images and configuration files). Thus, a Plux web application can be hosted in a virtual directory that contains the web page by which the application is accessed as well as the plug-ins and the Plux web runtime that composes the application.

6.1 Structure of the virtual directory

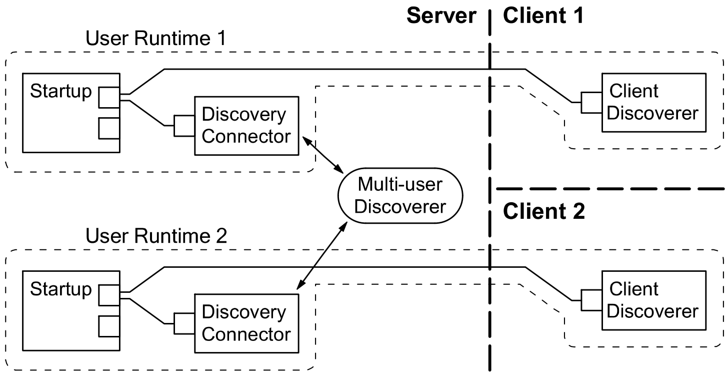
The virtual directory for a Plux web application (Fig. 30) contains a *Bin* directory for assemblies that are executed on the server and a *ClientBin* directory for Silverlight assemblies that are downloaded to the client to be executed in the client's sandbox environment. The *Bin* directory contains a subdirectory *Plugins* for server-side plug-ins that are discovered by the multi-user discoverer. It also contains

Fig. 28 Populating the types stores of different users with server plug-ins, client plug-ins and sandbox plug-ins



```
<Discovery>
  <Exclude File="B.dll" Scope="Group X" />
  <Exclude File="A.dll"
       Scope="Anonymous" />
</Discovery>
```

Fig. 29 Integrating client-side discoverers into the user runtimes



the framework assemblies of Plux, i.e., *Plux.Web.Server.dll*, *Plux.Web.Startup.dll* and *Plux.Web.Discoverer.dll*.

The assembly *Plux.Web.Server.dll* implements the Plux web runtime and provides an API to start, stop and access Plux web applications. Moreover, it implements an ASP.NET custom web control that can be used in ASP.NET web pages. The custom control starts the Plux web runtime on the first request and reuses it on further requests. It also

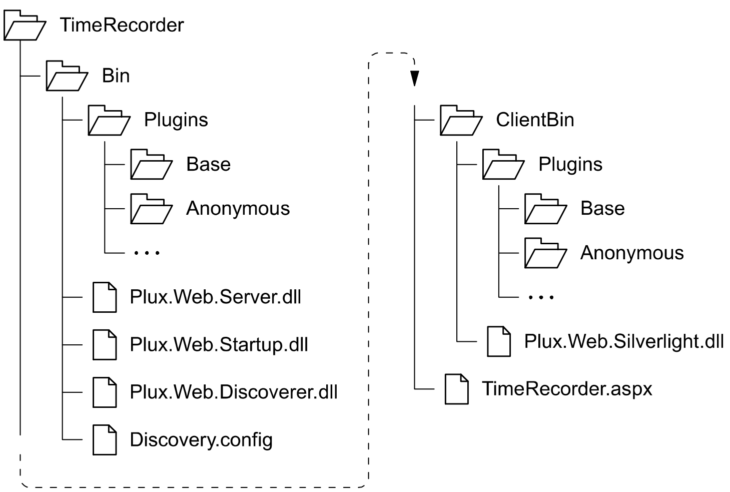


Fig. 30 Structure of the virtual directory for Plux web applications hosted with ASP.NET

determines the current user and renders the user interface of the web application that is plugged into the user's *Startup* extension to the web page.

The assembly *Plux.Web.Startup.dll* is a plug-in that contains the *Startup* extension, which is instantiated for every user at startup.

The assembly *Plux.Web.Discoverer.dll* implements the multi-user discoverer. It uses an optional configuration file *Discovery.config*, which is stored in the same directory as the discoverer plug-in. Since all plug-ins at the outermost level of the *Bin* directory are discovered by a special bootstrap discoverer, the multi-user discoverer is discovered at startup and is plugged into the discoverer slot of the *Startup* extension.

Similar to the *Bin* directory, the *ClientBin* directory contains a subdirectory *Plugins* for sandbox plug-ins that are executed in the client's Silverlight environment. It also contains the Silverlight assembly *Plux.Web.Silverlight.dll*, which implements the runtime node for the Silverlight environment.

Finally, the virtual directory contains the ASP.NET web pages (.aspx pages) that can be requested by the users. For example, the virtual directory in Fig. 30 contains the web page *TimeRecorder.aspx*.

6.2 Including the Plux custom control into an ASP.NET web page

Plux.Web.Server.dll implements an ASP.NET custom control (named *Application*) that can be used to deploy Plux web applications on .aspx pages. This is shown with an example in Fig. 31.

Every .aspx page starts with the <%@ Page> directive, which specifies, among other things, the language in which the page is written or where to find an associated code-behind file.

```
<%@ Page ... %>
<%@ Register Assembly=“Plux.Web.Server”
   Namespace=“Plux.Web” TagPrefix=“plux” %>

<!DOCTYPE ... >
<html>
  <head>
    <title>Time Recorder</title>
  </head>
  <body>
    <form id=“form1” runat=“server” >
      <plux:Application runat=“server” />
    </form>
  </body>
</html>
```

Fig. 31 Including the Plux custom control *Application* into the ASP.NET web page *TimeRecorder.aspx*

In order to make the custom control known to the .aspx page, the assembly *Plux.Web.Server.dll* has to be registered using the <%@ Register> directive. This directive specifies the custom control's assembly, the namespace in which the control is located, and a tag prefix that is used to qualify the control in the .aspx page. Finally, the custom control *Application* can be included under the name <plux:Application> into the <form> tags.

On every request, the .aspx page gets rendered, i.e., the <plux:Application> tag is replaced with the HTML content, which is rendered by the Plux web application that is plugged into the *Startup* extension of the current user.

6.3 Implementation of a Plux web application

Finally, we have to provide an application extension that can be plugged into the *Startup* extension. Therefore, we implement an extension *TimeRecorder* that provides a plug *Plux.Web.Application* (Fig. 32). As required by the plug, the extension implements the interface *IApplication*, which specifies a property *Control*. This property returns another

```
using Plux;
using Plux.Web;
using System.Web.UI.WebControls;

[Extension]
[Plug(“Plux.Web.Application”)]
[Slot(“Control”)]
public class TimeRecorder : IApplication{
  public WebControl Control{
    get{
      Slot cs = Slots[“Control”];
      return LayoutManager.Arrange(cs);
    }
  }
  ...
}
```

Fig. 32 Implementation of the web application *TimeRecorder*

ASP.NET custom control, which is added to the custom control *Application* and is responsible for rendering the user interface. *Application* is a container for all extensions that are plugged into the application slot of the user's *Startup* extension.

The property *Control* of the class *TimeRecorder* in Fig. 32 returns a custom control that renders the user interface of the time recorder. A *LayoutManager* is used to arrange the layout of all extensions that are plugged into the *Control* slot of the *TimeRecorder*.

Now we compile *TimeRecorder* to a .NET assembly and put it into the *Plugins* directory for server-side plug-ins. There it will be discovered and plugged into the *Startup* extension. Finally, the custom control *Application* instructs the custom control of *TimeRecorder* to render the time recorder's user interface to HTML, which is sent back to the client's browser.

7 Related work

Plug-in frameworks have become quite popular recently. However, most of them were designed for building rich client applications and not for building web applications. Even frameworks that do allow building web applications lack support for per-user extensibility and customization.

Web programming platforms, on the other hand, support building web applications for multiple users, but without plug-in extensibility. Of course, there are web applications that support customization and per-user extensibility. However, such features are usually not provided by the platform, but have to be programmed by hand.

7.1 Browser Plug-ins

The most frequent form of plug-ins on the web are browser plug-ins. For example, the Firefox browser can be extended with plug-ins that run Flash animations or display PDF documents. However, such plug-ins are leaf components, i.e., they do not have slots for being extended with other browser plug-ins, and cannot communicate with each other. We are aware, that there are extensible browser plug-ins such as the Greasemonkey extension for Mozilla Firefox. However, such plug-ins provide their own extensibility mechanism, which is different from the extensibility mechanism of the browser. For example, Greasemonkey uses scripts for that [9]. In contrast to that, Plux plug-ins are first-class building blocks of web applications. Every plug-in can be extended with other plug-ins, and all of them can communicate with each other via interfaces that are specified by slots.

7.2 Plug-in frameworks

The *Eclipse* Rich Client Platform [7] is a Java-based development platform for extensible desktop applications. Eclipse is based on the OSGi framework Equinox [16], which allows dynamic loading of components. Like Plux, Eclipse consists of a core that can be extended with plug-in components. The differences between Eclipse und Plux are: (1) The discovery mechanism for retrieving the metadata of components is an integral part of Eclipse, whereas in Plux it is an extension that can be replaced. (2) In Eclipse, the Java implementation of components is separated from their metadata, which are kept in XML files. In Plux, the metadata of components are specified with .NET attributes, which are placed directly in the source code. (3) In Eclipse, the hosts integrate their contributors themselves, whereas in Plux, programs are assembled automatically by the composer. (4) Eclipse maintains a registry of installed components, but only the hosts know which components they actually use. In contrast to that, Plux maintains a global composition state that keeps track of which components use which other components. (5) To support dynamic reconfiguration, an Eclipse host must provide both an implementation for integrating contributors at startup time and another one for integrating them dynamically at run time. In Plux, a single mechanism is used for integrating contributors both at startup time and at run time.

The Eclipse Rich AJAX platform RAP [18] allows building web applications using Eclipse plug-ins. A server-side Equinox environment [8] loads the Eclipse plug-ins on the web server and RAP delivers the user interface to the web browser using web technologies such as HTML and JavaScript. However, RAP does not maintain a global composition state, neither for a single user and even less for multiple users. It also does not support per-user customization.

SOFA 2 [4, 5, 10] is a system for building distributed component-based applications. The hierarchical component model distinguishes primitive and composite components. Primitive components are programmed, whereas composite components are declaratively composed from other primitive or composite components. The runtime environment allows users to transparently distribute the components across multiple nodes. The distributed components communicate with each other using automatically generated connectors. For communication, the connectors can use method invocation, message passing, streaming, or distributed shared memory. The runtime environment allows reconfiguring an application by adding, removing, and updating components at run time. The differences between Sofa and Plux are: (1) Sofa uses an architecture description language (ADL) to specify how a program should be composed from components. In contrast to that, Plux automatically establishes the desired composition by matching the requirements and provisions of components specified in their metadata. (2) Like Plux, Sofa can dynamically reconfigure an application. However, Sofa requires configurations to be statically specified in ADL, whereas Plux composes programs in a plug-and-play manner. (3) Sofa lacks support for extensible multi-user web applications, as it does not maintain the composition state for multiple concurrent users.

7.3 Web programming platforms

The Java Enterprise Edition (Java EE) [14] allows developing multi-tiered web applications using server and client components. Components on the server can be web components such as Java Servlets and JavaServer Faces as well as business components such as Enterprise JavaBeans. Components on the client can be application clients (rich-client applications) and applets. The differences between Java EE and Plux are: (1) Java EE applications are programmatically composed, whereas Plux applications are composed automatically in a plug-and-play manner. (2) Java EE lacks built-in support for per-user customization as well as for transparent remote plugging and unplugging. It also does not maintain a global composition state.

Web services [20] are a means to provide a public API for components over a network. Thus they can be used to build distributed component-based programs. One could compare a web service with a plug of a Plux component. In order to make a web service discoverable, it is registered in a public registry. A web service registry can be compared to a simplified Plux type store, containing only plugs. Consumers of web services typically integrate them using programmatic composition. Outside these components, the information about who uses which web service is unknown, i.e., the composition state is unavailable.

7.4 Portlets

Portlets [12] and [13] are user interface components that allow building extensible web applications. They can be integrated into a web site using a portal server, e.g. Apache Pluto [17]. The Portlet API is a thin layer on top of Java Servlets that provides means for integrating user interface components into web applications. Portlets can communicate with each other, which makes them a means to build extensible user interfaces. They can communicate via a session state if they run on the same server or via events that can be dispatched to portlets on remote servers. However, the assembly of programs from portlets must still be written manually.

8 Summary and future work

In this paper we presented the dynamic plug-in framework Plux that targets both rich client and web applications. The focus of the paper was on multi-user web applications that are extensible by end users.

With Plux, every user can add his own components to a web application and has an individual composition state. The distributed composition infrastructure of Plux automatically composes a seamless application from components that can reside on different computers. Server-side components are installed and executed on the server. Client-side components are installed and executed on the client and can therefore use local resources there. Sandbox components are installed on the server, downloaded to the client on demand, and executed in a sandbox on the client. Components of any kind can be provided for a single user or for a group of users. The infrastructure transparently connects components that are executed either on the same computer or on different computers. The communication between remote components is handled by the infrastructure using dynamically generated proxies.

Plux has been implemented under Microsoft .NET. However, its concepts are easily transferrable to any other platform (e.g. Java) that supports dynamic loading of components, interfaces, metadata annotations, as well as reflection. We have used Plux on various case studies, one of which is the time recorder web application that was used as an example throughout this paper.

Currently we are working on a security model that will allow us to specify plugging rights for individual slots, which are based on the identity of the contributors. We are also working on a layout manager for extensible user interfaces. Furthermore, in order to make web applications better scalable it must be possible to persist the composition of a user on the server and to restore it at the next round trip. Therefore, persistence is another aspect on our agenda.

Further information on Plux as well as a tutorial and a downloadable version of the framework can be found at http://ase.jku.at/plux/.

References

1. ASP.NET (2010) Microsoft ASP.NET. http://www.asp.net. Accessed 30 July 2010
2. Birsan D (2005) On plug-ins and extensible architectures. ACM Queue 3(2):40–46. doi:10.1145/1053331.1053345
3. Boudreau T, Tulach J, Wielenga G (2007) Rich client programming, plugging into the NetBeans platform. Prentice Hall International, Englewood Cliffs
4. Bures T, Hnetynka P, Plasil F (2006) SOFA 2.0: balancing advanced features in a hierarchical component model. In: 4th int conf on softw eng res manag and appl (SERA 2006), pp 40–48. doi:10.1109/SERA.2006.62
5. Bures T, Hnetynka P, Plasil F, Klesnil J, Kmoch O, Kohan T, Kotrc P (2007) Runtime support for advanced component concepts. In: 5th int conf on softw eng res manag and appl (SERA 2007), pp 337–345. doi:10.1109/SERA.2007.115
6. ECMA (2006) ECMA international standard ECMA-335, Common Language Infrastructure (CLI), 4th edn
7. Eclipse (2003) Eclipse platform technical overview. Object Technology International, Inc. http://www.eclipse.org. Accessed 28 July 2010
8. Equinox (2010) Server-side Equinox. http://www.eclipse.org/equinox/server. Accessed 28 July 2010
9. Greasemonkey (2010) Greasespot: the weblog about greasemonkey. http://www.greasespot.net. Accessed 15 December 2010
10. Hnetynka P, Plasil F (2006) Dynamic reconfiguration and access to services in hierarchical component models. In: The 9th int symp on compon-based softw eng (CBSE 2006), pp 352–359. doi:10.1007/11783565_27
11. Jahn M, Löberbauer M, Wolfinger R, Mössenböck H (2010) Rule-based composition behaviors in dynamic plug-in systems. In: The 17th Asia-Pac softw eng conf (APSEC 2010), pp 80–89. doi:10.1109/APSEC.2010.19
12. JSR168 (2003) JSR 168: Portlet specification. http://jcp.org/en/jsr. Accessed 28 July 2010
13. JSR286 (2008) JSR 286: Portlet specification 2.0. http://jcp.org/en/jsr. Accessed 28 July 2010
14. JSR313 (2007) JSR 313: Java Platform Enterprise Edition 6 specification. http://jcp.org/en/jsr. Accessed 28 July 2010
15. Mittermair C (2010) Zerlegung eines monolithischen Softwaresystems in ein Plug-In-basiertes Komponentensystem. Master thesis, Johannes Kepler University, Linz
16. OSGi (2006) OSGi Service Platform, Release 4. The Open Services Gateway Initiative. http://www.osgi.org. Accessed 28 July 2010
17. Pluto (2010) Apache Pluto. http://portals.apache.org/pluto. Accessed 28 July 2010
18. RAP (2010) The Rich Ajax Platform. http://www.eclipse.org/rap. Accessed 28 July 2010
19. Silverlight (2010) Microsoft Silverlight. http://www.microsoft.com/silverlight. Accessed 30 July 2010
20. W3C (2002) W3C Web services. http://www.w3.org/TR/ws-arch. Accessed 28 July 2010
21. Wolfinger R (2010) Dynamic application composition with Plux.NET: composition model, composition infrastructure. Dissertation, Johannes Kepler University Linz
22. Wolfinger R, Löberbauer M, Jahn M, Mössenböck H (2010) Adding genericity to a plug-in framework. In: 9th int conf on gener program and compon eng (GPCE), pp 93–102. doi:10.1145/1868294.1868308
23. Wolfinger R, Reiter S, Dhungana D, Grünbacher P, Prähofer H (2008) Supporting runtime system adaptation through product line engineering and plug-in techniques. In: 7th IEEE int conf on compos-based softw syst, pp 21–30. doi:10.1109/ICCBSS.2008.30

Markus Jahn is a researcher in the Christian Doppler Laboratory for Automated Software Engineering at Johannes Kepler University Linz. He received his Master's degree (Dipl.-Ing.) in Computer Science from the University of Linz. His ongoing Ph.D. research focuses on distributed plug-in architectures for user-specific web applications on the basis of the plug-in platform Plux (http://ase.jku.at/plux). Contact him at Johannes Kepler University, 4040 Linz, Austria; jahn@ase.jku.at.

Reinhard Wolfinger is an assistant at the System Software Institute of the Johannes Kepler University Linz. He received his Ph.D. in Business Informatics from the University of Linz in 2010. His research focuses on plug-in component architectures and dynamic application composition. His plug-in platform Plux is available at http://ase.jku.at/plux. Contact him at Johannes Kepler University, 4040 Linz, Austria; reinhard.wolfinger@jku.at.

Markus Löberbauer is a researcher in the Christian Doppler Laboratory for Automated Software Engineering at Johannes Kepler University Linz. He received his Master's degree (Dipl.-Ing.) in Computer Science from the University of Linz. His ongoing Ph.D. research focuses on testing and debugging of component-based software on the basis of the plug-in platform Plux. Contact him at Johannes Kepler University, 4040 Linz, Austria; loeberbauer@ase.jku.at.

Hanspeter Mössenböck is a professor of Computer Science at the Johannes Kepler University Linz and the head of the Christian Doppler Laboratory for Automated Software Engineering. He received his Ph.D. from the University of Linz in 1987 and was an assistant professor at ETH Zurich from 1988 to 1994. His current research interests include programming languages and compilers as well as object-oriented and component-based software engineering. He is the author of several books on programming education and the designer of the compiler generator Coco/R. Contact him at Johannes Kepler University, 4040 Linz, Austria; hanspeter.moessenboeck@jku.at.