



FirstSpirit™

Unlock Your Content

FirstSpirit™ AppCenter FirstSpirit™ Version 5.0

Version	0.46
Status	in process
Date	2013-02-18
Department	FS-Core
Copyright	2013 e-Spirit AG
File name	APPC50DE_FirstSpirit_Modules_AppCenter

e-Spirit AG
Barcelonaweg 14
44269 Dortmund | Germany

T +49 231 . 477 77-0
F +49 231 . 477 77-499

info@e-spirit.com
www.e-spirit.com

e-Spirit

Table of contents

1	Introduction.....	4
1.1	FirstSpirit AppCenter – FirstSpirit as an integration platform.....	4
1.2	Application integration in WebClient.....	5
1.3	Classification.....	6
1.4	General information	8
1.5	Restrictions.....	8
1.6	License model	9
1.7	Preview.....	10
1.8	Topics covered in this document	11
2	Concept: Integrating a web application into FirstSpirit.....	12
2.1	Communication: FirstSpirit JavaClient – web application	12
2.1.1	Controlling the integrated browser	12
2.1.2	Communication between the browser instance and JavaClient ...	13
2.1.3	Converting data types	16
2.1.4	Return by means of callback function	17
2.2	DOM access: access to the data of the integrated browser	18
3	Standard enhancements.....	20
3.1	Interface: ApplicationService	21
3.2	Interface: ApplicationTab	23
3.3	Interface: ApplicationTabAppearance.....	25
3.4	Interface: ApplicationTabConfiguration	28



3.5	Interface: TabListener	30
3.6	Abstract Class: ApplicationType	31
3.7	Interface: BrowserApplication.....	31
3.8	Interface: BrowserListener.....	34
3.9	Interface: BrowserApplicationConfiguration.....	35
3.10	Interface: ClientServiceRegistryAgent	37
4	Example: Integrating Google Maps in FirstSpirit	38
4.1	First steps	39
4.1.1	Note on the FirstSpirit license model.....	39
4.1.2	Note regarding legal implications	39
4.1.3	Generate Google Maps API key	39
4.1.4	Note on configuring the FirstSpirit server.....	40
4.1.5	Installing the Google Earth plug-in	41
4.1.6	Example project	41
4.2	Application areas of the Google Maps integration	42
4.2.1	Address search with geolocalization	42
4.2.2	Changing the coordinate using the Google Maps integration.....	43
4.2.3	Showing additional information (Google Balloons)	44
4.2.4	3D display using Google Earth.....	45
4.2.5	Route directions ("How to find us")	46
4.3	Implementation: Application integration for Google Maps	48
4.3.1	(SwingGadget) input component <code>CUSTOM_GEOLOCATION</code>	50
4.3.2	MapsPlugin – Generating a new instance of the type MapsPlugin51	
4.3.3	MapsPlugin – Opening the application within a tab	53
4.3.4	MapsPlugin – run JavaScript (Java » JavaScript).....	57
4.3.5	MapsPlugin - GeolocationUpdater (Injection Java » JavaScript) .	61



4.3.6	Show markers and assign an input component	64
4.3.7	Listener – responding to changes	74
4.3.8	Updating the geodata of the input component (JavaScript » Java).....	79
4.3.9	Responding to tree navigation events (Java » JavaScript)	87
4.3.10	Updating the browser instance (Java » JavaScript)	89
4.3.11	MapsPlugin – Address search (Google-Geolocation)	93
4.3.12	maps.html – Introduction	99
4.3.13	Excursus: HTML and JavaScript	100
4.3.14	maps.html - Loading the Google Maps API.....	101
4.3.15	maps.html - Initializing the container for the map display	102
4.3.16	maps.html – Creating a new map object.....	103
4.3.17	maps.html - Center map (center point or display area)	104
4.3.18	maps.html – define map type.....	106
4.3.19	maps.html - load map object via events	108
4.3.20	maps.html – converting address data (Geocoding)	109
4.3.21	maps.html – GeolocationUpdater (Injection Java / JavaScript) ..	110
5	Listings	114



1 Introduction



This document is provided for information purposes only. e-Spirit may change the contents hereof without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. e-Spirit specifically disclaims any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. The technologies, functionality, services, and processes described herein are subject to change without notice.

1.1 FirstSpirit AppCenter – FirstSpirit as an integration platform

From the outset, FirstSpirit has been conceived and implemented as an integration platform. This includes consistent focusing of all in-house implementations on the core components of the "FirstSpirit" product and deliberate outsourcing of specific functions to third party products of the respective market leaders. The success of such a best-of-breed strategy stands and falls with the capability of the system integration. The decisive requirement, to enable successful implementation of this popular outsourcing idea in a software product, is "seamless integration": there must be no break between the products used for the end user. The user prompting must be fully integrated, seamless and visually appear to the user as a unified whole.

This idea of seamless integration of third party applications in the FirstSpirit editing environment is called "AppCenter".

The FirstSpirit AppCenter provides an area within the editing system, within which independent applications can run, which are not part of FirstSpirit (so-called "AppCenter applications"). Examples of AppCenter applications are the integration of Microsoft Office or the functions for integrated image editing. The integrated web browsers Mozilla Firefox and Microsoft Internet Explorer are also AppCenter applications; they are called "integrated preview". All these AppCenter applications were implemented by e-Spirit as parts of its product. There is also a range of AppCenter applications, which are implemented as FirstSpirit modules. These AppCenter modules can be developed by e-Spirit itself, as well as by a partner.

The following examples, which have already been successfully integrated by e-Spirit within the scope of the AppCenter, give an impression of the options the AppCenter provides in addition to



the applications currently implemented as product components:

- by integrating Google Maps or Google Earth, geocoordinates can be easily and intuitively used in the FirstSpirit JavaClient (see Chapter 4 page 38).
- by integrating the online video offering of MovingImage24, videos can be selected and integrated into FirstSpirit with the click of the mouse.

From a technical point of view, the AppCenter consists of a set of interfaces, which have been released by e-Spirit for use by partners, so that they can implement and integrate individual applications within the scope of the AppCenter, in order to adapt the Client to their special needs. The implementation of AppCenter applications and their integration into FirstSpirit is generally called "application integration". This term is used to describe the most seamless integration possible of third party software, even if it based on completely different technology, in the editorial interface of JavaClient.

To sum up:

- The editor finds a fully-integrated desktop, in which they can immediately work with the tools they are familiar with (for example, office applications).
- Existing, customized applications can be integrated without a lot of time or effort, even if they are non-Java applications.
- New mini-apps can be quickly and effectively developed for the customer, tailored to their use case and based on existing technology. These can be small .NET-based web applications or Flash applications.

1.2 Application integration in WebClient

Application integrations are available in form of reports in WebClient from version FirstSpirit version 5 on too. Enhancements will be made in FirstSpirit version 5.1 with the objective of making application integrations available cross-client in JavaClient and WebClient.



1.3 Classification

A differentiation is made between the following variants of web application integration:

a) Blackbox integration:

The web application to be integrated provides an interface, of whatever kind, which is used for the integration. This interface can either be defined in the form of an API, via HTTP parameters or JavaScript or can also exist in the form of defined HTML or URL constructs. In all cases, however: the inner structure of the application to be integrated does not have to be known, instead, the interaction takes place solely via the defined API interfaces.

b) Graybox integration:

The Graybox integration assumes that the interface between the web application and the FirstSpirit Client use the HTML code in the web browser. This HTML code must be analyzed and if necessary manipulated, in order to reach the information required (see Chapter 2.2 page 18). This means, knowledge about the internal structure of the web application to be integrated is required, which is not necessary in the case of blackbox integration.

c) Whitebox integration:

Whitebox integration exists if the web application has been especially developed or modified for use within the scope of FirstSpirit application integration. That is to say, in a whitebox integrated web application, specific FirstSpirit interfaces are addressed or entry points suitable for FirstSpirit are provided, in order to implement the application integration. This type of integration naturally not only requires access to the source code of the web application, but also the possibility of changing the application.

A blackbox integration is therefore especially suitable for the integration of web applications, which already exist and cannot or should not be changed (enterprise-specific web application) – however, the absolute requirement is for the necessary interfaces to be available.

If the web application to be integrated does not provide the necessary interfaces, or the source code of the web application to be integrated is not to be or cannot be changed, a graybox integration is the right solution – although here it must be noted that if a change is made to the web application (e.g. relaunch) adjustments will probably have to be made to the integration. Graybox integration is therefore only suitable for integrating web applications, which are only subject to a few (e.g. Wikipedia) or even no changes (legacy enterprise web applications).



The form of integration with the most options is whitebox integration, which enables very deep integration between the web application and FirstSpirit – on the one hand the web application can be controlled from JavaClient and vice versa, the JavaClient can be controlled from the web application. Whitebox integration offers enormous potential, especially in conjunction with FirstSpirit modules, as parts of the module's user interface can be implemented in the form of a web application, which provides advantages especially if specific interfaces only are available for web applications.

To sum up: In principle, within the scope of application integration, all types of web application can be seamlessly integrated into the user interface of the FirstSpirit JavaClient – especially within the scope of graybox integration, even for web applications which were not actually intended for integration. However, (due to the underlying principle) an integration without explicit API involves greater dependency on the specific implementation of the integrated application, i.e. changes in the application to be integrated potentially result in the need to adjust the integration code. The techniques and methods of the FirstSpirit graybox integration reduce these dependencies as far as possible, but principally cannot completely remove the change dependencies.



1.4 General information

When integrating and using individually customized AppCenter applications it must be noted that FirstSpirit provides the interfaces necessary for the application integration, but in general does not have any influence on the integrated applications themselves.

Integrated external applications are not part of the FirstSpirit product. Among other things, this means that responsibility for the function of the integrated applications lies with the manufacturer of the application or with the customer or partner who implements the application.

Problems can be reported within the scope of the FirstSpirit product support and (where possible) are corrected, if they lie on the level of the integration interface. However, e-Spirit is not obliged to provide debugging within the integrated third party applications.

Use of the user's own applications in the AppCenter requires a license. For further information see Chapter 1.6 page 9.

1.5 Restrictions

The application integration is based on the existing web browser integration, of the Microsoft Internet Explorer and Mozilla Firefox browsers, in the FirstSpirit JavaClient (see Chapter 2.1 page 12). When using the web browser integrations in the JavaClient, in principle, restrictions can occur, e.g. because several of the integrated applications do not fully work with all platforms or bit versions (32 or 64 bit).

Use of Internet Explorer Version 8 or higher is recommended. Internet Explorer up to Version 8 does not support Base64 decoding. This can cause problems when image elements are injected within the scope of the application integration (e.g. with the display of the FS_BUTTON component in the integrated preview or the integration of an image database).

For details of the requirements and restrictions of application integration, see FirstSpirit™ Release Notes for Version 4.2R4.



1.6 License model

The license parameter `license.APPTAB_SLOTS` specifies how many different application integrations can be used. These include applications which are available in the AppCenter of the JavaClient as well as applications in WebClient, for example self-implemented reports. With `license.APPTAB_SLOTS=5`, for example, five different applications can be used. Which applications these are is immaterial. Because unlike the licensing of FirstSpirit (module) add-ons, here it is not the function that is licensed, but the number of integrated applications.

Each of these application integrations licensed through this parameter can be opened in any number of JavaClients or WebClients, moreover in any number of tabs in JavaClient. The first Client, in which an application integration is opened fills a license ("AppTab Slot") for this application (the application is "registered") and increases the counter of the license parameter by 1.



The following applies: An AppTab slot is assigned for one calling instance, e.g. a script, and the opening of one URL. For example, if several URLs are opened within a script, this is an infringement of the FirstSpirit AppCenter license conditions.

The registration continues to exist, even after the respective Client has been exited. If the value of the `license.APPTAB_SLOTS` parameter has been reached, another application integration can be started in the respective client for test and demo purposes. A corresponding warning will be shown in the Clients or – for self-implemented WebClient reports – in the project properties (application for the "Server and Project configuration", "WebEdit settings" / "Report plug ins") and a warning will be logged in the file `fs-server.log`. In addition, no other applications can be started.

Several applications, which are displayed in the AppCenter of the JavaClient, but are delivered with the FirstSpirit core product as a standard, or are licensed via a separate parameter (e.g. the office integration for FirstSpirit), do not fall under the licensing parameter `license.APPTAB_SLOTS` and are not counted as an application integration; at present, these are:

- Integrated WYSIWYG preview (via Mozilla Firefox or Microsoft Internet Explorer)
- Integrated preview of media
- Integrated display of the FirstSpirit online Help
- Integrated, enhanced image editing in JavaClient



The input component `FS_BUTTON` (see online documentation for FirstSpirit) enables the integration of the user's own applications in the AppCenter of the JavaClient. If `FS_BUTTON` is used, each script and each class referenced from `FS_BUTTON` is counted as an application, which requires licensing via the `license.APPTAB_SLOTS` parameter.

The type and number of applications currently licensed through the `license.APPTAB_SLOTS` parameter can be checked in Server Monitoring, sub-menu "AppCenter Licenses" below the "FirstSpirit" / "Control" menu.

The "Reset Uses" button can be used if necessary to reset the number of registered applications to 0. Registered applications, which are currently open in Clients, can continue to be used until the application or the corresponding application tab is closed. The server does not have to be rebooted.

1.7 Preview

Development with regard to web application integration is currently not yet completed in FirstSpirit: Enhancements in the implementation should enable deeper integration.

Following the application integration for the editing workstation, the intention is to shift the focus to the developer workstation (e.g. integration of development environments) in a further development stage. The integration complexity will probably be significantly higher.



1.8 Topics covered in this document

Having explained the term "application integration" in this introductory chapter and compared the terms "Blackbox" vs. "Graybox" vs. "Whitebox" integration, in the following the technical principles for customized integration of web applications are described. Equally, the underlying interfaces, packages and classes are listed and explained. All concepts as well as the necessary FirstSpirit API interfaces are introduced by way of example implementations.

This document focuses on client-side application integration (see Chapter 4 page 38, section 3) with FirstSpirit. Concepts as well as the necessary FirstSpirit API interfaces are introduced by way of example implementations.



The documentation is currently being edited. Several aspects and interfaces are not yet documented or are not yet completely documented.

Chapter 2: In this chapter, the concepts behind the application integration are explained first. In particular, communication between the integrated web application and the FirstSpirit JavaClient (from page 12) is discussed.

Chapter 3: This chapter introduces the standard enhancements of the FirstSpirit Access API for application integration. Among other things, the interfaces for the control of the application tabs and the integration of web applications are presented (from page 19).

Chapter 4: This chapter describes the example implementation for integrating Google Maps into the FirstSpirit JavaClient. The implementation introduced creates a simple and intuitive option for working with geographic coordinates within the FirstSpirit editing environment. To do this, a SwingGadget input component is developed, which is closely linked to the Google Maps web application (from page 38).



2 Concept: Integrating a web application into FirstSpirit

2.1 Communication: FirstSpirit JavaClient – web application

The FirstSpirit JavaClient has had a seamlessly integrated web browser, which not only displays a direct preview of the editorial content in JavaClient, but also visualizes the relationship between the content entered in the Client and its effect or display on the website. The Mozilla Firefox and Microsoft Internet Explorer web browsers are optionally available for this.

This browser integration is also used for the integration of web applications. For a web application to be integrated in the FirstSpirit JavaClient, the following aspects must be covered first:

- Controlling the integrated browser (see Chapter 2.1.1 page 12)
- Communication between the browser instance and JavaClient (see Chapter 2.1.2 page 13)
- Converting data types (see Chapter 2.1.3 page 16)
- Return by means of callback function (see Chapter 2.1.4 page 17)

2.1.1 Controlling the integrated browser

In order to integrate a web application into the FirstSpirit JavaClient, it is necessary to control the browser integrated in FirstSpirit. The FirstSpirit AppCenter API provides the necessary interfaces in order, for example, to open a new tab (or a new browser instance) within the application area in which the required web application can be opened (for a description of the interfaces, see Chapter 3, page 20ff.).

The entry point for the control of a tab is the `ApplicationService` (see Chapter 3.1 page 20). This service can be used to open a new application of a specific type within the application area. The required `ApplicationType` is passed on opening the application (Abstract Class: `ApplicationType` see Chapter 3.6 page 31). Integration of a web application requires the `ApplicationType BrowserApplication`, which provides an interface for opening and controlling a new browser instance in the application area (Interface: `BrowserApplication` see Chapter 3.7 page 31).

The `openApplication(...)` method of the `ApplicationService` interface returns an instance of the type `ApplicationTab`. The `ApplicationTab` interface provides general methods for controlling the tab, for example, the tab can be brought to the front or closed using



the corresponding method invocations (Interface: ApplicationTab see Chapter 3.2 page 23). In addition, the instance of the type `ApplicationTab` can be used to get the (browser) application, which was opened within the application tab. This instance then provides access to other specific options for controlling the integrated application (Interface: BrowserApplication see Chapter 3.7 page 31). To track changes, suitable listeners can also be registered, an instance of the type `BrowserListener`, which responds to changes within the web application (Interface: BrowserListener see Chapter 3.8 page 34) and an instance of the type `TabListener`, which responds to changes within the tab (e.g. selection or deselection by the user (Interface: TabListener see Chapter 3.5 page 30)).

2.1.2 Communication between the browser instance and JavaClient

The integrated browser engines are of course not implemented in Java, but native for the respective Client operating system. The most important point for the integration of a web application in FirstSpirit is therefore the communication between the Java level of the FirstSpirit JavaClient and the native browser level of the web application. Two communication channels have to be considered:

- 1) **JavaClient » Web application:** Changes or events, which are triggered via the FirstSpirit JavaClient must be made known to the web application. For example, if a certain address is searched for within the geolocation input component (entry of an address string and click the Search button), a request for geocoding of this address string must be sent to the web application (Google Maps) and the map section adjusted within the integrated browser (see example Address search with geolocalization in Chapter 4.2.1 page 42).
- 2) **Web application » JavaClient:** The reverse path, i.e. the adoption of a change or event within the web application in the JavaClient must also be possible. For example, the coordinate determined by Google Maps and the complete address information should also be updated in the geolocation input component (see example Address search with geolocalization in Chapter 4.2.1 page 42).

The FirstSpirit Client API (Java) communicates with the integrated browser engine via JavaScript. The requirement is therefore to enable bidirectional communication between the Java and the JavaScript level. Specifically, three options have been created for setting up bidirectional communication:

- 1) **Run JavaScript:** targeted, unidirectional communication in the direction Java » JavaScript.
- 2) **Run JavaScript and evaluated returned value:** see above, however, with the ability to



evaluate the returned value.

- 3) **Provide Java object in the JavaScript environment:** is mainly used for unidirectional communication, however, in the direction JavaScript » Java.

Re. 1) For the first **communication direction Java » JavaScript** an interface is provided for calling a JavaScript method from Java. Basically, this only involves running JavaScript code in the form of a string. The `BrowserApplication` interface has been extended to include the `void executeScript(String script)` method, which runs the passed JavaScript code in the currently open browser document (Interface: `BrowserApplication` see Chapter 3.7 page 31).

Re. 2) The second option is somewhat more complicated. This also involves running JavaScript code, but also tries to evaluate the returned value and to convert it into suitable Java objects. The `BrowserApplication` interface has been extended to include the `Object evaluateScript(String script)` method, which runs the passed JavaScript code in the currently open browser document and returns a return value. (Interface: `BrowserApplication` see Chapter 3.7 page 31). A range of conversion rules are applied to the returned value as, unlike Java, JavaScript only supports a limited set of simple data types (see Chapter 2.1.3 page 16).

Re. 3) For the second **communication direction JavaScript » Java** another interface is provided, in order to make methods of a Java object accessible in a JavaScript environment. Specifically, a Java object is injected into the JavaScript environment (web browser) to generate a substitute object (proxy) in the form of a JavaScript object, whose (JavaScript) methods correspond to those of the Java object. Following the injection the corresponding methods can be invoked from the JavaScript. The internal FirstSpirit implementation generates a corresponding JavaScript method for each method of the Java object instances. When called from the JavaScript environment, this method sends an event, which is evaluated on the Java side. The suitable Java method is determined from the method signature and the passed parameters and is called accordingly.

Basically, any Java object can be injected into the JavaScript environment, however, restrictions must be noted and observed (see Chapter 2.1.3, page 16). The relevant method `void inject(Object object, String name)` is provided via the `BrowserApplication` interface of the FirstSpirit AppCenter API (Interface: `BrowserApplication` see Chapter 3.7 page 31).

A further, minor restriction concerns the parameter passing. As synchronicity cannot be guaranteed for event generation/evaluation, returned values of the Java methods are returned by means of callback (see Chapter 2.1.4 page 17)



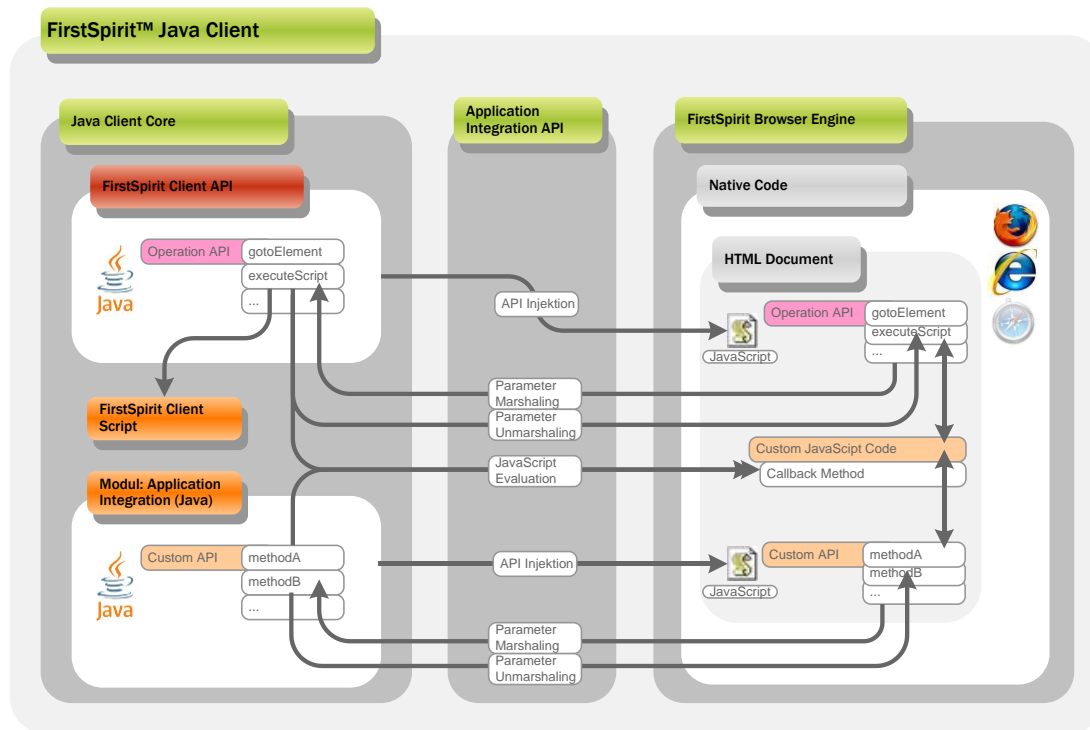


Figure 1: Java - JavaScript communication (Java-JavaScript bridge)

A prototype for this integration is introduced in the already mentioned example of the geolocation input component. Here a call interface is implemented between the FirstSpirit Client API and a web application, which runs in an instance of the integrated browser, (Implementation: Application integration for Google Maps see Chapter 4.3 page 48).



2.1.3 Converting data types

In this case too, two communication directions must be considered:

- 1) Injecting a Java object into the JavaScript environment: The API injection interface can be used to inject any Java object into a JavaScript environment. All methods of the Java object are adopted, but no attributes (e.g. name attribute). As, unlike Java, JavaScript only supports a limited set of data types, certain restrictions also apply to the methods of this object. Only a range of simple, atomic data types, and lists and maps can be converted. Complex object types not known in JavaScript on the other hand are not supported. In addition, the method adoption and mapping of the Java data types on JavaScript data types is only possible on the highest level, this means:

```
MyJavaObject {  
    void helloWorld(String message);  
}
```

can be called on the JavaScript page as follows, after calling `inject(MyJavaObject, "myObject")` :

```
window.myObject.helloWorld("Hello!")
```

But a nested structure is not possible, this means:

```
MyJavaObject {  
    MyComplexObject getMyComplexObject();  
}  
MyComplexObject {  
    void helloWorld(String message);  
}
```

cannot be called on a JavaScript page as follows after calling `inject(MyJavaObject, "myObject")` :

```
window.myObject.getMyComplexObject().helloWorld("Hello!");
```



- 2) Parameter passing from the JavaScript into the Java environment: Apart from the injection of a Java object in the JavaScript environment, it is also possible for parameters to be passed from the JavaScript environment into the Java environment, for example, for the evaluation of a returned value. All parameters passed from the JavaScript environment, are elevated in the Java environment, analogous to the object conversion to date (see Figure 1).

In addition: The converted Java objects are merely copies of the JavaScript objects. Therefore, a change to the Java object has no effect on the JavaScript environment.

An overview of the possible transformation of the data types (or parameters) on changing between the Java and the JavaScript environment, is provided by the `convertToScript(...)` method of the `BrowserApplication` interface (see Chapter 3.7 page 31), for example:

- `js:number` «» `Double`
- `js:boolean` «» `Boolean`
- `js:string` «» `String`
- `js:object` «» `Map<String, Object>`

2.1.4 Return by means of callback function

As synchronicity cannot be guaranteed for event generation or evaluation, the returned values of the Java methods must be returned by means of the callback function. Therefore, a Java method `String getName()`, becomes the `void getName(function:callback)` method in the JavaScript environment. The last parameter passed always names the relevant callback function, which is to be called when the calling function has been dealt with (see Figure 1).



2.2 DOM access: access to the data of the integrated browser

As already explained in Chapter 2.1, the integrated browser engines are a native implementation, which cannot be readily reached from the Java environment. This means, access to the data of the integrated browser or the data of the web application (the HTML or the browser document) is initially not possible in Java. Especially within the scope of a seamless graybox integration, it is however necessary to access the inner structures of the web application from the Java implementation, as here either an API is not provided at all or only limited API access (JavaScript) is provided.

The FirstSpirit AppCenter API has therefore been extended to include an interface, which enables access to the DOM tree of the integrated browser (Mozilla Firefox or Internet Explorer). The task of this interface is to enable a Java program (the FirstSpirit JavaClient or a module) read and write access to precisely the data currently displayed in the integrated web browser. The `Document getCurrentDocument()` method of the `BrowserApplication` interface returns the current browser document as w3c-DOM¹ (see Chapter 3.7 page 31). Therefore, the complete content of the web browser is made available as a document model within the Java environment. The HTML structures of the integrated web application can then be run through and analyzed on this document. The document model made available to the Java application is however not limited to read access, but can also be manipulated, whereby all changes to the document immediately become visible in the integrated browser.

The technical sequences for use of the DOM facade are clearly illustrated in the following figure:

¹ http://en.wikipedia.org/wiki/Document_Object_Model



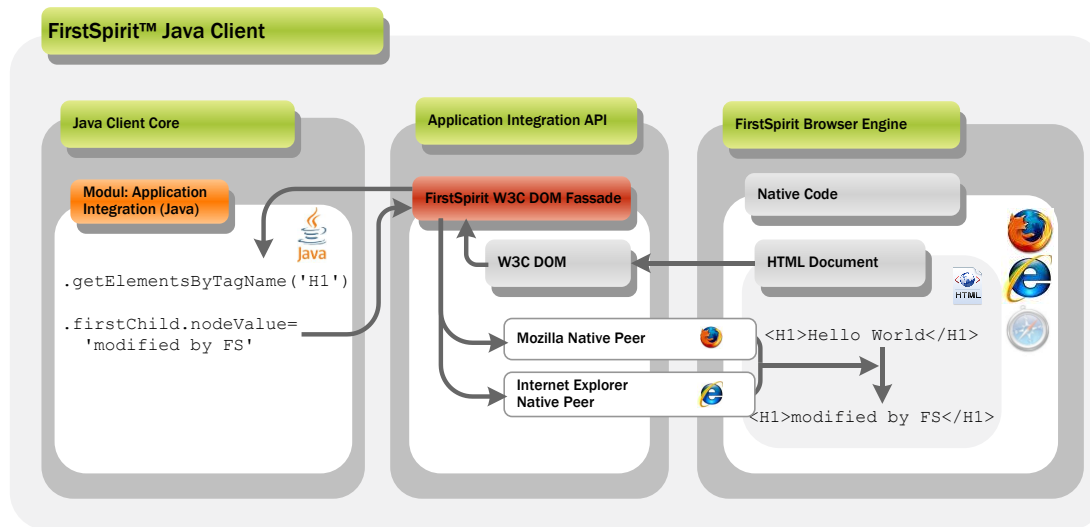


Figure 2: DOM access

The access to the data shown in the web browser and manipulation of this data is relatively easy to do via the interface, but in this case there are also principally restrictions.

The document model changes itself dynamically. A change within the web application is therefore, for example, immediately traced in the DOM. In practice, use of the DOM object can result in too many "non-reproducible" errors of the class NPE or AIOB; for example, if child iteration takes place within the implementation, but the child object "soon thereafter" no longer exists.



Therefore, in the implementation of an (graybox) application integration for the FirstSpirit JavaClient, suitable error handling is of central importance for the stability of the implementation.

For the implementation of this DOM interface, particular attention was paid to the aspect of fully transparent synchronization of all concurrencies (parallel processing), as otherwise deadlock situations would have been unavoidable in the connection of native running processes and changing operations.

Note: DOM access to Flash or Silverlight applications is NOT possible, as the (accessible) HTML code does not contain all the relevant parameters and an insight into the Flash application itself is not possible!



3 Standard enhancements

From a technical point of view, the AppCenter consists of a set of interfaces ("FirstSpirit AppCenter API"), which have been released by e-Spirit for use by partners, so that they can implement and integrate specific applications within the scope of the AppCenter.

At present, the FirstSpirit AppCenter API is limited to the infrastructure needed for the integration of web applications. Appropriate interfaces for the integration of native applications (see integration of Microsoft Office) as well as Java applications (see integration for Java image editing, Java Image Editor) have already been realized, but are not (yet) publically available.

Interfaces for configuring and controlling the application tab:

- Interface: ApplicationService (see Chapter 3.1 page 21)
- Interface: ApplicationTab (see Chapter 3.2 page 23)
- Interface: ApplicationTabAppearance (see Chapter 3.3 page 25)
- Interface: ApplicationTabConfiguration (see Chapter 3.4 page 28)
- Interface: TabListener (see Chapter 3.5 page 30)

Interfaces for integrating a web application or a browser:

- Abstract Class: ApplicationType (see Chapter 3.6 page 31)
- Interface: BrowserApplication (see Chapter 3.7 page 31)
- Interface: BrowserListener (see Chapter 3.8 page 34)
- Interface: BrowserApplicationConfiguration (see Chapter 3.9 page 35)



All the interfaces introduced in the following are part of the FirstSpirit Developer API. In contrast to the Access API, the requirements concerning the stability for the Developer API are lower: The Developer API remains stable within a minor version line, i.e. methods, classes and functions may be changed in case of a minor version change.

The documentation is currently being edited. Several aspects and interfaces are not yet documented or are not yet completely documented.



3.1 Interface: ApplicationService

Package: `de.espirit.firstspirit.client.gui.applications`

The entry point for controlling a tab is always the `ApplicationService`. This service can be requested from the FirstSpirit Framework via different brokers. To do this, the typified `SwingGadgetContext` (see Developer Manual for Components) must be used to request an instance of the type `SpecialistsBroker` with the help of the `SpecialistsBroker` `getBroker()` method.

```
SwingGadgetContext<...> _context;
final SpecialistsBroker _specialistsBroker = _context.getBroker();
```

On the `SpecialistsBroker`, with the help of the `<S> S` `requireSpecialist(SpecialistType<S> type)` method, can then be used to request a specialist of the type `ServicesBroker`. By calling the method `<T> T` `getService(Class<T> serviceClass)`, this broker returns an instance of the type `ApplicationService` (see Chapter 4.3.2).

```
final ServicesBroker servicesBroker =
    _specialistsBroker.requireSpecialist(ServicesBroker.TYPE);

final ApplicationService service = servicesBroker.getService(ApplicationService.class);

final BrowserApplication app = service.openApplication(BrowserApplication.TYPE,
    configuration).getApplication();
```

`ApplicationService` can be used to open new applications of a certain type within the application area (see Interface: ApplicationTab) or to get the applications from existing browser instances (see Interface: BrowserApplication). The `ApplicationService` can only be used within the FirstSpirit JavaClient.

The `ApplicationService` provides access to the following methods:

- `ApplicationTab<T> openApplication(final ApplicationType type, final C configuration)`: The method opens an application of a certain type (`ApplicationType`) in a new tab in the application area of the FirstSpirit JavaClient. The type (`Abstract Class: ApplicationType`) of the required application and the configuration for the integrated browser are passed to the method (see Interface: ApplicationTabConfiguration and `Abstract Class: ApplicationType`). The method returns a typified instance of the type



`ApplicationTab` (for example implementation see Chapter 4.3.3 page 53).

Note: For the passing parameter `ApplicationType`: FirstSpirit currently only provides an interface of the type `BrowserApplication`, via which the new browser instances can be opened in the application area of the `JavaClient` (see Abstract Class: `ApplicationType`).

Note: For the passing parameter `ApplicationTabConfiguration`: If an `ApplicationTab` is to be reused within the implementation, the `ApplicationTabConfiguration` should be used to define an identifier (see Interface: `ApplicationTabConfiguration`). The tab can then be got later using the method `ApplicationTab<T> getApplication(final ApplicationType<T, C> type, final Object tabIdentifier)` (see below).

- `ApplicationTab<T> getApplication(final ApplicationType<T, C> type, final Object tabIdentifier)`: This method returns an instance of an `ApplicationTab`, which was opened in the application area of the `JavaClient`, or `null` if no `ApplicationTab` is found, which corresponds to the passed parameters. The type (Abstract Class: `ApplicationType`) of application and the identifier for the `ApplicationTab` are passed (see Interface: `ApplicationTabConfiguration`). The method returns a typed instance of the type `ApplicationTab` (see Chapter 3.6 page 31).
- `boolean isVisible()`: The method determines whether the application area is visible in the `JavaClient` (`true`) or not (`false`).
- `void setVisible(final boolean visible)`: The method opens (`true`) or closes (`false`) the application area in `JavaClient`.

The following example beanshell script demonstrates access to the `ApplicationService` and the opening of an `ApplicationTab`.

```
import de.espirit.firstspirit.client.gui.applications.*;
import de.espirit.firstspirit.client.gui.applications.browser.*;

apps = context.connection.getService(ApplicationService.class);
tab = apps.openApplication(BrowserApplication.TYPE, "Browser");
browser = tab.getApplication();
browser.openUrl("www.e-spirit.com");
tab.setTitle("e-Spirit AG");
tab.close();
```

For an example of use of the `ApplicationService` see Chapter 4.3.3 page 53.



3.2 Interface: ApplicationTab

Package: de.espirit.firstspirit.client.gui.applications

Each page or media preview is displayed in the application area of the FirstSpirit JavaClient in a preview tab. On requesting a new preview (for example, a new page view), the view of the preview tab is automatically updated by the FirstSpirit Framework.

To integrate a web application, first, a new tab must be opened in the application area of the JavaClient. This Application tab is independent of the actual Preview tab. This means that if the web application is integrated, for example, via an input component in FirstSpirit (see geolocation input component), alongside the actual page preview, another tab with the integrated web application can be opened within the application area.

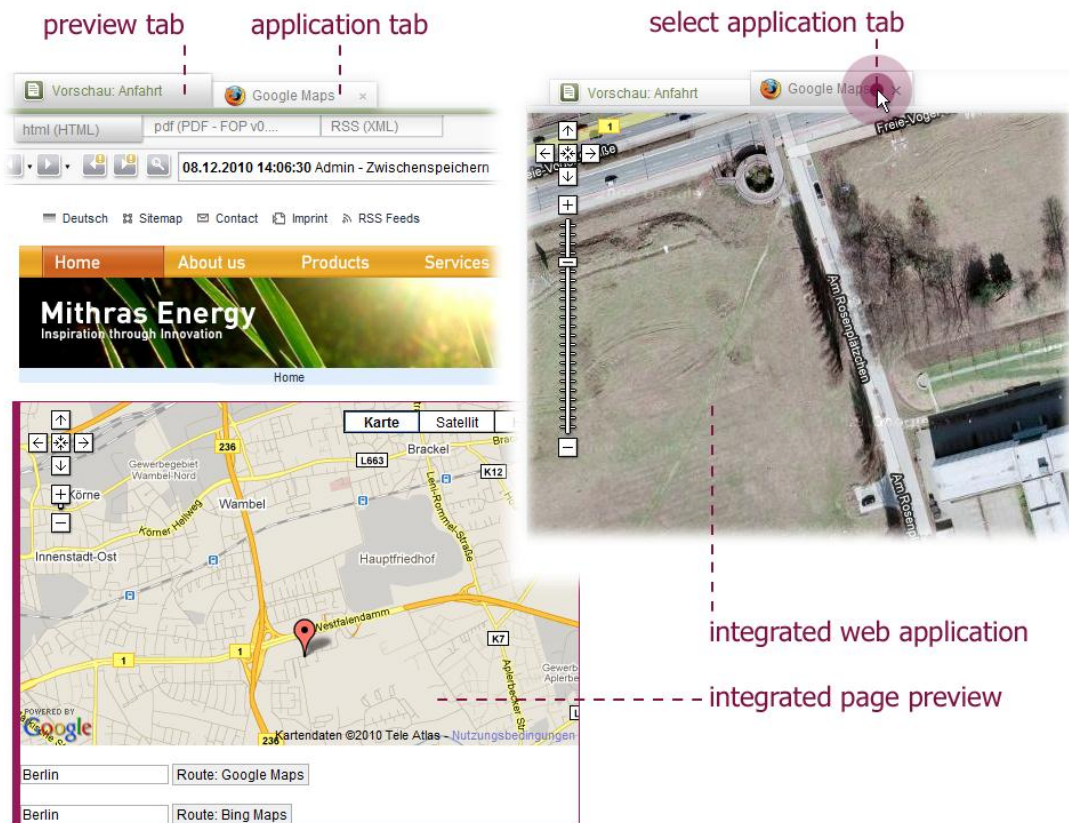


Figure 3: Preview tab and Application tab

Access to the application tab (and the applications contained in it) is controlled by the `ApplicationTab` interface. Unlike the preview tab, which is controlled by the FirstSpirit Framework, the application tab must be controlled by the developer. When the user requests the web application, the developer must first open a new browser instance. To do this, the `openApplication(...)` method is called on the `ApplicationService` (see Interface:



ApplicationService). The method returns an instance of the type `ApplicationTab`, via which the new instance of the integrated browser can be controlled (e.g. closing the tab).

To track changes within the tab (e.g. selection or deselection by the user), an instance of the type `TabListener` must be added (see Interface: `TabListener`).

The `ApplicationTab` interface provides the following methods:

- `void addTabListener(@NotNull final TabListener listener)`: The method adds a `TabListener` to an instance of the type `ApplicationTab`. A `TabListener` responds to events within the application tab, for example, the closing or deselection of a tab by the user (see Interface: `TabListener` in Chapter 3.5 page 30) (for an example, see Chapter 4.3.3 page 53).
- `void removeTabListener(@NotNull final TabListener listener)`: This method removes an existing `TabListener`.
- `void close()`: This method closes the instance of the `ApplicationTab` on which it was called.
- `boolean isClosed()`: this method checks whether or not the instance of the `ApplicationTab` has been closed. The method is closely related to the `void tabClosed()` method from the `TabListener` interface (see Chapter 3.5 page 30). On closing an application tab, these methods are called to enable correct reply to the "Tab has been closed" status. For example, this information is required if an existing application tab is to be reused. In this case the developed must be able to decide whether an `ApplicationTab` opened once is available for a new request (tab has already been opened and can be used for the new request) or not (tab has been closed – a new tab must be opened for the request).
- `void setAppearance(ApplicationTabAppearance appearance)`: This method affects the display of the `ApplicationTab` in the application area. An instance of the type `ApplicationTabAppearance` is passed to the method, which enables configuration of the outer appearance, for example, the addition of an icon to the tab (see Chapter 3.3 page 25).
- `T getApplication()`: This method returns the instance of the application, which was integrated in the application area. The returned value is standardized (see Chapter 3.6 page 31).
- `boolean isSelected()`: The method returns whether the `ApplicationTab` actively appears in the foreground (`true`) or is only opened in the background (`false`). This information is relevant, for example, if a change is made to the corresponding `SwingGadget` input component, which affects the integrated web application. As long as the relevant tab is in the background, the change should not have an effect within the application.
- `void setSelected()`: The method marks the instance of the application tab on which it was called as being active. This means that the corresponding tab is displayed in the



foreground in the application area of the JavaClient.

For an example of use of the ApplicationService see Chapter 4.3.3 page 53.

3.3 Interface: ApplicationTabAppearance

Package: `de.espirit.firstspirit.client.gui.applications`

The display of a tab in the application area of the FirstSpirit JavaClient is influenced using the `ApplicationTabConfiguration` (see Chapter 3.4 page 28) and `ApplicationTabAppearance` interfaces. These interfaces, for example, can be used to define a text or a specific icon, which are to be displayed within the tab. In addition, the `ApplicationTabAppearance` interface provides further configuration options, for example, the possibility of changing the font weight (plain/bold) for the title lettering of the tab (see Figure 4).

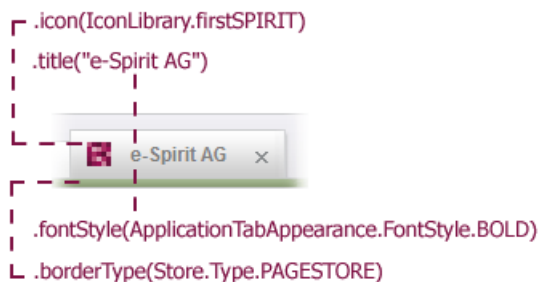


Figure 4: Example of the ApplicationTabAppearance

The methods of the two interfaces partly overlap. For example, an icon can be added to an `ApplicationTab` not only using the `ApplicationTabConfiguration.icon(...)` method but also using the `ApplicationTabAppearance.Builder icon(...)` method.

To simplify the configuration, the `ApplicationTabAppearance` interface provides a builder implementation (`ApplicationTabAppearance.Builder` interface) with the following methods:

- `ApplicationTabAppearance.Builder borderType(final Store.Type borderType)`: The bar, which separates a higher-level `ApplicationTab` from its lower-level tabs, can be displayed in color (see Figure 4). The display depends on the Store type. The default behavior in the JavaClient displays the strip for a page preview (in the Page Store), for example, in green, while in a preview of a page reference (in the Site Store) it is shown in blue. This method can be used to define a Store type (`Store.Type`), to adjust the color of the bar. If a Store type is not defined, a colorless bar is displayed.
- `ApplicationTabAppearance.Builder fontStyle(final FontStyle`



`fontStyle`): This method can be used to influence the font weight, which is applied to the lettering within an `ApplicationTab`. To do this, the method of the required `ApplicationTabAppearance.FontStyle` can be passed. At present, the types `FontStyle.PLAIN` (default value) and `FontStyle.BOLD` are supported (see Figure 4).

- `ApplicationTabAppearance.Builder icon(Icon icon)`: This method can be used to pass an icon (preferred size: 20x20 pixels), which is shown within the `ApplicationTab` (see Figure 4).
- `ApplicationTabAppearance.Builder title(String title)`: This method can be used to pass a text, which is displayed as lettering within the `ApplicationTab` (see Figure 4).
- `ApplicationTabAppearance get()`: This method returns the instance of the type `ApplicationTabAppearance`, which is built on this building.

A new instance of the type `ApplicationTabAppearance` can be created by calling `ApplicationTabAppearance.GENERATOR.invoke()`. The configuration then takes place using the simplified builder pattern. In order for the changed parameters to subsequently have an effect on the display of the `ApplicationTab`, the `ApplicationTabAppearance` must be passed to the `ApplicationTab` using the `ApplicationTab.setAppearance(ApplicationTabAppearance appearance)` method (see Chapter 3.2).

Example:

```
...
private ApplicationTab<BrowserApplication> _tab;

final ApplicationTabAppearance appearance =
ApplicationTabAppearance.GENERATOR.invoke()
    .title("e-Spirit AG")
    .borderType(Store.Type.PAGESTORE)
    .fontStyle(ApplicationTabAppearance.FontStyle.BOLD)
    .icon(IconLibrary.firstSPIRIT)
    .get();

_tab = service.openApplication(BrowserApplication.TYPE,
(BrowserApplicationConfiguration) null);
_tab.setAppearance(appearance);
...
```

An instance of the type `ApplicationTabAppearance` can also be got directly from an instance of the type `ApplicationTabConfiguration` by calling the `ApplicationTabConfiguration.appearance()` or `ApplicationTab`



`Configuration.getAppearance()` methods (see Chapter 3.4). In this case the `ApplicationTabAppearance` is part of the `ApplicationTabConfiguration` and can, for example, easily be passed when a new `ApplicationTab` is opened. Example:

```
...
private ApplicationTab<BrowserApplication> _tab;

final BrowserApplicationConfiguration configuration =
    BrowserApplicationConfiguration.GENERATOR.invoke();

final ApplicationTabAppearance appearance = configuration.appearance()
    .title("e-Spirit AG")
    .borderType(Store.Type.PAGESTORE)
    .fontStyle(ApplicationTabAppearance.FontStyle.BOLD)
    .icon(IconLibrary.firstSPIRIT)
    .get();

_tab = service.openApplication(BrowserApplication.TYPE, configuration);
...
```

The `ApplicationTabAppearance` interface provides the following methods for querying the values, which have been configured for the display of an `ApplicationTab`:

- `Store.Type getBorderType()`: This method returns the `Store` type (`Store.Type`), which was previously set using the `ApplicationTabAppearance.Builder borderType(Store.Type borderType)` method of the builder implementation. If no `Store.Type` has been designed, the method returns `null`.
- `FontStyle getFontStyle()`: This method returns the font style, defined for the lettering within the `ApplicationTab`. The font style can be influenced using the `ApplicationTabAppearance.Builder fontStyle(ApplicationTabAppearance.FontStyle fontStyle)` method. If a special font style has not been defined for the lettering, a normal font style is used (default value `FontStyle.PLAIN`).
- `Icon getIcon()`: This method returns the icon, which was previously defined using the `ApplicationTabAppearance.Builder icon(Icon icon)` method of the builder implementation or was set using the `ApplicationTabConfiguration.icon(Icon icon)` method for the display within the `ApplicationTab`.
- `String getTitle()`: This method returns the text, which is displayed as lettering within the `ApplicationTab`. The text is defined using the `ApplicationTabAppearance`.



`Builder title(String title)` method of the builder implementation.

3.4 Interface: ApplicationTabConfiguration

Package: `de.espirit.firstspirit.client.gui.applications`

The display of a tab in the application area of the FirstSpirit JavaClient is influenced using the `ApplicationTabConfiguration` and `ApplicationTabAppearance` interfaces (see Chapter 3.3 page 25). These interfaces, for example, can be used to define a text or a specific icon, which are to be displayed within the tab (see Figure 4).

Other configuration options exist, depending on the type of application, which is opened within an `ApplicationTab`. The `BrowserApplicationConfiguration` interface, for example, extends the `ApplicationTabConfiguration`, interface to include methods for the configuration of the web application. This means an address line can be shown within the application area, or a specific browser engine can be specified for opening the web application (see Chapter 3.9 page 35).

The following applies: The `ApplicationTabConfiguration` interface merely forms the base class. This base class is extended by other configuration interfaces, which are precisely tailored to the respective application type. When a new `ApplicationTab` is generated (call the method: `ApplicationTab<T> openApplication(final ApplicationType type, final C configuration)`) the required application type is passed. An application-specific instance is also expected for the passing of the configuration. Therefore, if an application of the type `BrowserApplication`, is opened within the `ApplicationTab`, the configuration passed must be an instance of the type `BrowserApplicationConfiguration`.

FirstSpirit currently only offers one secure interface for the application type web applications (`BrowserApplication`) and the corresponding configuration interface `BrowserApplicationConfiguration`. Other application types are already being developed and are expected to be released with FirstSpirit Version 5 (see Chapter 3.6)

The `ApplicationTabConfiguration` interface provides the following methods:

- `ApplicationTabAppearance.Builder appearance()`: This method returns a builder instance of the type `ApplicationTabAppearance.Builder`, which provides other options for configuring the tab display (see Chapter 3.3 page 25).
- `ApplicationTabAppearance getAppearance()`: This method returns an instance of the type `ApplicationTabAppearance`, which provides further options for the configuration



of the tab display (see Chapter 3.3 page 25). It is recommended that the simplified builder implementation of the `ApplicationTabAppearance` interface be used at this point (see above).

- `Object getIdentifier()`: This method returns the identifier, which was defined using the `ApplicationTabConfiguration` with the help of the `public T identifier(final Object tabIdentifier)` method for an instance of the type `ApplicationTab` (see below). If a specific identifier has not been defined, the method returns a string, which is formed from the prefix "BrowserApplication_" and the system time. If the identifier is known, the corresponding `ApplicationTab` can be fetched later using the `ApplicationService` with the help of the `ApplicationTab<T> getApplication(final ApplicationType<T, C> type, final Object tabIdentifier)` method (Interface `ApplicationService` see Chapter 3.1 page 21).
- `public T identifier(final Object tabIdentifier)`: If an `ApplicationTab` is to be reused within the implementation, this method can be used to assign an identifier. The identifier can be used to get the corresponding `ApplicationTab` later using the `ApplicationService` with the help of the `ApplicationTab<T> getApplication(final ApplicationType<T, C> type, final Object tabIdentifier)` method (Interface `ApplicationService` see Chapter 3.1 page 21).
- `public T icon(final Icon icon)`: This method can be used to pass an icon (preferred size: 20x20 pixels), which is shown within the `ApplicationTab` (see Figure 4) (see also Chapter 3.3 page 25).
- `public T openInBackground(boolean openInBackground)`: This method can be used to influence whether the `ApplicationTab`, which is based on this configuration, is opened in the background (`true`) or not (`false`).
- `boolean openInBackground()`: The method returns whether the `ApplicationTab`, which is based on this configuration, is to be opened active in the foreground (`false`) or only in the background (`true`).
- `public T title(final String title)`: This method can be used to pass a text, which is displayed as lettering within the `ApplicationTab` (see Figure 4) (see also Chapter 3.3 page 25).

For an example, see Chapter 3.9 page 35.



3.5 Interface: TabListener

Package: `de.espirit.firstspirit.client.gui.applications`

A new tab is opened in the application area of the JavaClient for the integration of a web application in FirstSpirit. This application tab (unlike the conventional preview tab) must be controlled by the developer. The methods required for this are described in the `ApplicationTab` interface (see Chapter 3.2 page 23).

The tab can be controlled by events. To respond to internal or external event, an instance of the type `TabListener` must be registered on the application tab. This listener contains methods, which inform the developer of events, which take place on the tabs in the application area. For example, if an application tab is closed by the user, the FirstSpirit Framework responds by calling the `void tabClosed()` method.

The interface (and all methods contained in the interface) can be implemented to generate a new instance of the type `TabListener`. However, it is recommended that the internal, abstract adapter implementation (Abstract Adapter Class), provided by the `TabListener` interface be used instead. This predefined class implements all the interface's methods. The developer only has to implement the methods relevant for them (see Listener – responding to changes, Chapter 4.3.7, page 74) and can otherwise fall back on the existing default implementation. The adapter class is also advantageous for subsequent extension of the interface. If the interface is supplemented, for example, with a new method, existing implementations remain compatible. The new method only has to be implemented by the developer if and when necessary.

An instance of the type `TabListener` must then be registered on the event source (here the application tab) by calling the `addTabListener(...)` method.

The `TabListener` interface (or the corresponding adapter implementation) provides the following methods:

- `void tabSelected()`: This method is called when the corresponding `ApplicationTab` is selected, i.e. is displayed visible in the foreground. The method is closely linked to the `void setSelected()` method from the `ApplicationTab` interface (see Chapter 3.2 page 23).
- `void tabDeselected()`: This method is called when a new tab (application or preview tab) is brought to the foreground. In this case the corresponding `ApplicationTab` moves into the background. For example, if the change within an input component is only to be traced, if the `ApplicationTab` concerned is actively displayed in the foreground, this can be achieved using the `void tabDeselected()` method.



- `void tabClosed()`: This method is called by the FirstSpirit Framework, when the instance of an `ApplicationTab` is closed. Whether a tab has already been closed can be determined by calling the `boolean isClosed()` method of the `ApplicationTab` interface (see Chapter 3.5 page 30).

3.6 Abstract Class: ApplicationType

Package: `de.espirit.firstspirit.client.gui.applications`

On opening a new application via the `ApplicationService` a specific application type is passed (see Chapter 3.1 page 21), for example for web applications:

- `BrowserApplication`: Interface for opening and controlling a new browser instance in the application area of the FirstSpirit JavaClient (see Chapter 3.7 page 31).

For other application types, for example, an interface for integrating swing applications, see FirstSpirit AppCenter API.

The abstract class provides the following methods:

- `String name()`: The method returns the full name of the respective `ApplicationType`.

3.7 Interface: BrowserApplication

Package: `de.espirit.firstspirit.client.gui.applications.browser`

In order to integrate web applications in the application area of the JavaClient, access to the integrated browser of FirstSpirit is required. The `BrowserApplication` interface provides methods, for generating and controlling a new instance of the respective browser. Many of the methods it contains are run asynchronously. In order for ordered access to the content of the web application to be possible, an instance of the type `BrowserListener` should be registered on the `BrowserApplication`.

The interface provides access to the following methods:

- `EngineType getEngineType()`: This method returns the current `EngineType` of the browser. FirstSpirit currently integrates two web browser engines, which can be optionally used - Mozilla Firefox and Microsoft Internet Explorer. The required browser engine can be selected via the configuration (see Chapter 3.9 page 35). Apart from a rigid definition, by specifying



`BrowserApplicationConfiguration.GENERATOR.invoke().engineType(EngineType.DEFAULT)` it is also possible to give the default browser engine, saved by the respective user in the FirstSpirit JavaClient (JavaClient menu bar: Menu item: View – Browser Engine). In this case, the `getEngineType()` method returns the corresponding, specific type.

- `BrowserApplication.getEngineVersion()`: This method returns the current version of the browser as a string. **Note:** In the case of "Mozilla Firefox" this is not the Firefox version, but the Xulrunner version. Assignment to the Firefox version must be carried out independently here (if necessary).
- `void openUrl(final String url)`: The method opens the passed URL within a browser instance in the application area of the JavaClient. This can either be the URL of an external web application or a customized implementation, which has to be globally installed on the FirstSpirit server first, and can then be opened in the application area using the `openUrl(...)` method (see Chapter 4.3.3 page 53). This method is run asynchronously. A `BrowserListener` must be registered to determine at what time the method is run. This informs the user at the time it is run, that the location has changed.
- `void openUrl(final Location location)`:
- `String getUrl()`
- `void addBrowserListener(@NotNull final BrowserListener listener)`: The method registers a `BrowserListener` on an instance of the type `BrowserApplication`. A `BrowserListener` responds to changes or events within the web application (Interface: `BrowserListener` see Chapter 3.8 page 34, for example implementation, see Chapter 4.3.7.2 page 76).
- `String convertToScript(Object object)`: The method converts the passed Java object into JavaScript code and returns this as a string. As, unlike Java, JavaScript only supports a limited set of data types, certain restrictions also apply to the methods of this object. Only a range of simple, atomic data types, and lists and maps can be converted. Complex object types not known in JavaScript on the other hand are not supported. If a passed Java object is `null` or is not supported, the method returns `null`:

Java

- Number, Boolean
- String
- List<Object>
- Map<String, Object>

JavaScript

- *(related to String mechanism)*
- "stringcontent" (escapes newline and ")
- [entry0,entry1,entry2,...]
- {'key0':value0,'key1':value1,...}

For further information on the conversion of data types, see Chapter 2.1.3 page 16).



- `<T> BrowserNodeHandlerBuilder<T> createNodeHandlerBuilder()`
- `void executeScript(String script)`: This method runs the passed JavaScript code in the currently opened browser document and therefore enables targeted, unidirectional communication in the direction: Java » JavaScript. The JavaScript code to be run is passed as a string (for example, see Chapter 4.3.4 page 57).
- `Object evaluateScript(String script)`: This method runs the passed JavaScript code in the currently opened browser document and therefore enables targeted, unidirectional communication in the direction: Java » JavaScript, but also returns a returned value. The return values passed from the JavaScript environment are converted into Java objects according to specific conversion rules, for example, a `js:number` object becomes an object of the type `Double` (Converting data types see Chapter 2.1.3 page 16).
- `void removeBrowserListener(@NotNull final BrowserListener listener)`
- `void focus()`: Calling this method shifts the focus onto the current browser instance. This is useful, for example, if the integrated web application contains a form element, which is to be directly assigned an input cursor or, as in the example of Google Maps integration, to enable direct zooming with the mouse wheel, if the application tab is selected by the editor (see example in Chapter 4.3.7.1).
- `void setHtmlContent(String html)`
- `Document getCurrentDocument()`
- `void inject(Object object, String name)`: This method is required for communication between the Java level of the FirstSpirit JavaClients and the JavaScript level of the web application. The method injects the passed Java object as an attribute of the window object in the browser instance on which it was called (for information on the `window` object, see Chapter 4.3.13). The injection generates a substitute object (proxy) in the form of a JavaScript object and registers it under the passed name (for example, see Chapter 4.3.5 page 61).

Access to the DOM tree of the browser instance is not possible at any time. The registration can only take place if the document has been fully loaded. To ensure this, an instance of the type `BrowserListener` must be used (see Interface: `BrowserListener`)(DOM access concept see Chapter 2.2 page 18).

following registration the JavaScript object can be used in the JavaScript environment using the call `window.{name}`. All Java objects methods can then also be called from the JavaScript environment of the integrated browser.

Background: With the injection the FirstSpirit framework generates for each method of the Java object instances a corresponding JavaScript method with (approximately) identical



method signature. On being called from the JavaScript environment, this JavaScript method sends an event, which is evaluated on the Java side and triggers the running of the corresponding Java method there. The suitable Java method is determined from the method signature and the passed parameters and is called.

Example (generation of an instance of the type `BrowserApplication`):

```
ApplicationService appService = servicesBroker.getService(ApplicationService.class);
BrowserApplication browser = appService.openApplication(BrowserApplication.TYPE,
    null).getApplication();
browser.openUrl("www.e-spirit.de");
```

3.8 Interface: BrowserListener

Package: `de.espirit.firstspirit.client.gui.applications.browser`

Access to the content of the browser instance can be controlled with the help of a `BrowserListener`. To do this, an instance of the type `BrowserListener` must be registered on the browser instance (`BrowserApplication`). An instance of the type `BrowserListener` contains methods, which inform the developer about changes to the browser instance in the application area. If, for example, the URL of the browser instance is changed, the FirstSpirit Framework responds by calling the `void onLocationChange(@NotNull String url)` method.

The interface (and all methods contained in the interface) can be implemented to generate a new instance of the type `BrowserListener`. However, it is recommended that the internal, abstract adapter implementation (Abstract Adapter Class), provided by the `BrowserListener` interface be used instead. This predefined class implements all the interface's methods. The developer then only has to implement the methods relevant for them (see Listener – responding to changes, Chapter 4.3.7, page 74) and can otherwise fall back on the available default implementation. The adapter class is also advantageous for subsequent extension of the interface. If the interface is supplemented, for example, with a new method, existing implementations remain compatible. The new method only has to be implemented by the developer if and when necessary.

An instance of the type `BrowserListener` must then be registered by calling the `addBrowserListener(...)` method on the event source (here the `BrowserApplication`).

The `BrowserListener` interface (or the corresponding adapter implementation) provides the following methods:

- `void onLocationChange(@NotNull String url)`: The method is called if the



BrowserListener reports that the URL of the browser instance (instance of the type `BrowserApplication`) has changed.

- `void onDocumentComplete(String url)`: The `void onDocumentComplete(...)` method is called if the `BrowserListener` of the browser instance (instance of the type `BrowserApplication`) reports that the document (including all images) has been completely loaded (for DOM access concept see Chapter 2.2 page 18).

3.9 Interface: BrowserApplicationConfiguration

Package: `de.espirit.firstspirit.client.gui.applications`

The `BrowserApplicationConfiguration` interface extends the base class `ApplicationTabConfiguration` (see Chapter 3.4 page 28) to include configuration options for the display of `BrowserApplications` within an `ApplicationTab`. For example, a specific browser engine can be defined for opening the web application or an address line can be shown for display of the called URL in the application area.

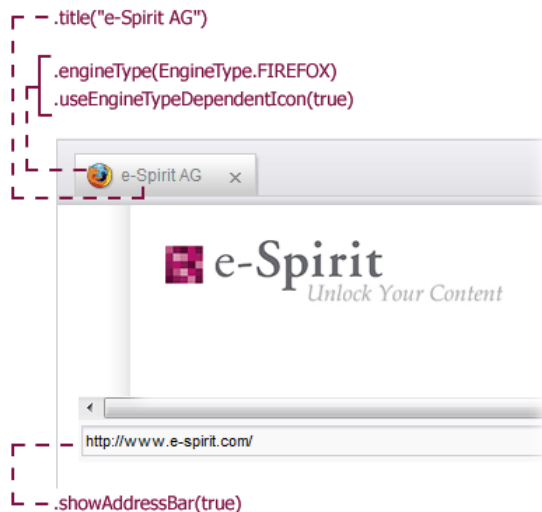


Figure 5: Example of BrowserApplicationConfiguration

The `BrowserApplicationConfiguration` interface is derived from the base class `ApplicationTabConfiguration` and therefore provides all methods described in Chapter 3.4. Furthermore, the `BrowserApplication Configuration` interface contains the following methods:

- `BrowserApplicationConfiguration useEngineTypeDependentIcon(boolean useEngineTypeDependentIcon)`: If a specific icon has not been defined for the tab display, this method can be used to show the icon of the `BrowserEngine`.



- `BrowserApplicationConfiguration showAddressBar(final boolean showAddressBar)`: This method can be used to define whether an address field with the called URL is to be displayed in the bottom part of the application area (default value: `true`) or not (`false`). This method can be used, for example, to show the URL of an external web application or a web application installed on the FirstSpirit Server; the application having been opened in the application area using the `openUrl(...)` method (see Chapter 3.7 page 31). If, on the other hand, an HTML code is initiated (using the `setHtmlContent(...)` method, see Google Maps example) the display of an address field can be suppressed.
- `boolean showAddressBar()`: This method returns whether, for the `ApplicationTab`, which was opened based on this configuration, a address field is shown (`true`) or not (`false`) (see method: `BrowserApplicationConfiguration showAddressBar(final boolean showAddressBar)`).
- `public BrowserApplicationConfiguration engineType(@NotNull final EngineType type)`: This method can be used to define a browser engine, which is to be used to open the web application in the application area. FirstSpirit currently integrates two web browser engines, which can be optionally used - Mozilla Firefox and Microsoft Internet Explorer. Apart from a rigid definition, by specifying `BrowserApplicationConfiguration.GENERATOR.invoke().engineType(EngineType.DEFAULT)` it is also possible to give the default browser engine, saved by the respective user in the FirstSpirit JavaClient (JavaClient menu bar: Menu item: View – Browser Engine).
- `public EngineType getEngineType()`: The method returns the `EngineType`, which was previously defined using the `BrowserApplicationConfiguration.engineType(@NotNull final EngineType type)` method for the opening of the web application in the application area. Note: If, instead of a specific type, a default browser engine was defined (`EngineType.DEFAULT`), this method returns the `EngineType DEFAULT`. The `BrowserApplication.getEngineType()` method can be used to obtain the respective specific type (see Chapter 3.7 page31).

A new instance of the type `BrowserApplicationConfiguration` can be generated with the call `BrowserApplicationConfiguration.GENERATOR.invoke()`. The configuration then takes place using the simplified builder pattern. The application specific configuration is passed on calling the `ApplicationTab<T> openApplication(final ApplicationType type, final C configuration)` method and must match the passed application type (here: `BrowserApplication`).



Example:

```
final ApplicationService service =
    _servicesBroker.getService(ApplicationService.class);

final BrowserApplicationConfiguration configuration =
    BrowserApplicationConfiguration.GENERATOR.invoke()
        .icon(IconLibrary.firstSPIRIT)
        .title("e-Spirit AG")
        .identifier("test")
        .engineType(EngineType.FIREFOX)
        .showAddressBar(true)
        .useEngineTypeDependentIcon(true)
        .openInBackground(false);
```

3.10 Interface: ClientServiceRegistryAgent

Package: de.espirit.firstspirit.agency



4 Example: Integrating Google Maps in FirstSpirit

In the area of presentation layer integration, applications for travel directions, which work with geographic coordinates (geographic latitude and longitude), such as site plans and route planners, have firmly established themselves on the websites of most businesses. In general, these applications have the problem that initially only the address is known, but not the absolutely relevant geographic coordinate.

The example implementation introduced in this chapter is intended to create a simple and intuitive option for working with geographic coordinates within the FirstSpirit editing environment. To this end, a geolocation input component has been developed, which determines all relevant geographic information for an address and saves it for further editing. To do this, the component uses the web application Google Maps, which is seamlessly integrated into the applications area of the FirstSpirit editing environment. Search queries can be directly forwarded from the input component to the Google Maps API, but it is also possible to search using the Google Maps map display in the application area. The geographic information determined is then saved within the input component for further processing.

In the example, close integration of the web application with the FirstSpirit JavaClient is to be achieved which, for example, also enables Google Maps objects to be dragged and dropped into the geolocation input component.

- **Objective:** Introduction of an easy to handle solution for geographic coordinates
- **Technique:** Web application integration, HTML based graybox technique (see Chapter 1.1 page 4).



4.1 First steps

4.1.1 Note on the FirstSpirit license model

Use of the FirstSpirit AppCenter is subject to a new license model. Unlike licensing of a FirstSpirit (module) add-on to date, here it is not the function that is licensed, but the number of integrated applications (see Chapter 1.6 page 9). An example implementation can be used for test and demo purposes, and yet be installed on the FirstSpirit Server without a valid license.

4.1.2 Note regarding legal implications

The example implementations presented here for the integration of Google Maps (or the integration of a picture database) are not standard FirstSpirit functions (see Chapter 1.4 page 8). The implementation is only intended to show by way of example, the possibilities provided by application integration in FirstSpirit and how these can be implemented.



If an application integration (for example, for Google Maps) is to be implemented within a project, the licenses required for use of the integrated application must be requested directly from the application manufacturer. In particular, use of the Google technology is subject to strict restrictions (see Terms of Service for creating a Google account).

4.1.3 Generate Google Maps API key

To use Google Maps on your website you will need the Google Maps API. To use this, you will in turn need a Google Maps API key. This must be requested directly from Google. A Google Maps API key is then valid for a "directory" or a domain. I.e. a key can be used for the following URLs:

- <http://www.myserver.com/maps/index.php>
- <http://www.myserver.com/maps/map.html>

but not, e.g. for

- <http://subdomain.myserver.com/index.html>



In general, it is advisable to register the domain name. The key is then valid for this domain, its sub-domains, all URLs of hosts in these domains and all ports on these hosts².

Note: If Google maps is also to be used in the preview of the editing system, the FirstSpirit Server must be operated in the same domain as the domain registered here. The start page of the FirstSpirit Server must therefore lie within the domain of the registered API key (Note on configuring the FirstSpirit server see Chapter 4.1.4).

First, an account with Google must exist. The API key, which is requested in the next step, is coupled with this Google account. If a Google account is not yet available it can be created under the following URL:

<https://www.google.com/accounts/NewAccount>

Several API keys can be requested for an account.

The API key is requested using the following URL:

<http://www.google.com/apis/maps/signup.html>

To do this the URL, which is to be used for the Google Maps service, is entered in the "URL of my website" field. If the terms of services are accepted and the "Generate API key" button is clicked the key is displayed in the next window.

4.1.4 Note on configuring the FirstSpirit server

The Google Maps API key is only registered for a domain (not for a host name) (see Chapter 4.1.3).

To prevent FirstSpirit from being used via URLs, which do lie within the registered domain, that is, for example, via `http://fs4server` instead of `http://fs4server.myserver.com`, the external Apache httpd, which can be used for FirstSpirit in addition to the integrated Jetty web server, should be configured as follows:

```
RewriteCond %{HTTP_HOST} !^hostname\.domain$ [nocase]
RewriteRule ^/(.*) http://hostname.domain/$1 [redirect=permanent,noescape,last]
```

All calls of the FirstSpirit Start page are then forwarded to a defined URL (with domain).

² For more detailed information on the validity of a Google Maps API key, see also <http://code.google.com/intl/de/apis/maps/faq.html#keysystem>



4.1.5 Installing the Google Earth plug-in

To use the 3D display of Google Earth in the integrated applications area of FirstSpirit (see Chapter 4.2.4) the Google Earth plug-in must be installed. If the plug-in is not installed, a page appears within the application area of the FirstSpirit JavaClient, which prompts you to download the plug-in. Click the "Download Google Earth plug-in" button; the plug-in is saved directly on your workstation. To install the plug-in, open the relevant file `GoogleEarthPluginSetup.exe` with a double-click. Follow the instructions of the installation program³.

Following installation the Google Earth View can be visible within the application area.

4.1.6 Example project

An example project, which uses the geolocation input component described here, can be made available by the FirstSpirit Helpdesk on request. Please contact <https://helpdesk.e-spirit.de>.

The input component can also be integrated in any FirstSpirit projects required. To do this, the page or section templates concerned merely have to be extended.

For further information on template development see online documentation of FirstSpirit (ODFS).


For information on the development of SwingGadget input components, see Developer Manual for Components.

³ For further information on installing and uninstalling the Google Earth plug-in see also <http://maps.google.com/support/bin/answer.py?hl=en&answer=178389>



4.2 Application areas of the Google Maps integration

4.2.1 Address search with geolocation

In this example implementation a geolocation input component is developed for FirstSpirit, which determines the correct geographic position data for an entered address or part of an address (for example, a street name) and saves it for further processing. The input component has an input field for an address string, which can be edited by the editor. By clicking the "Search Geolocation" button the editor can start a text-based search via the Google Maps API. To do this, the Google Maps web application is integrated in the FirstSpirit editing environment. The implementation behind the input component (see Chapter 4.3 page 48) initially opens another tab next to the Preview tab in the application area of the JavaClient, which contains the Google Maps web application. The geographic coordinates determined via the search are assigned a marking (marker) within the map , copied into the SwingGadget input component where it is displayed both as full address information and with the geographic longitude and latitude.

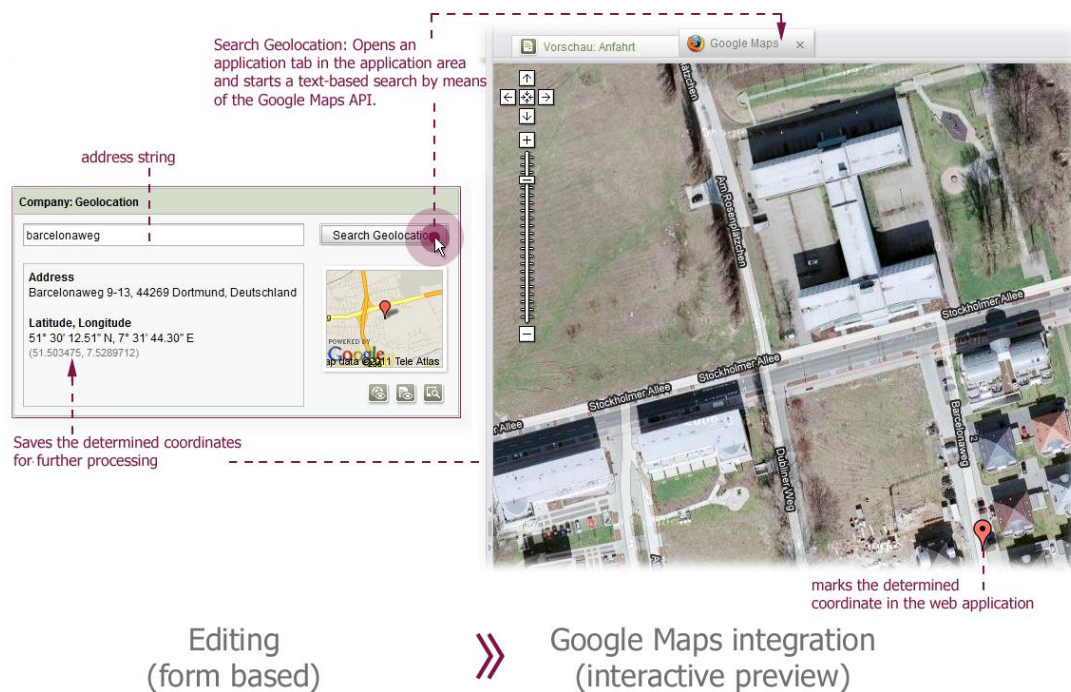


Figure 6: Use case - address search with geolocation



4.2.2 Changing the coordinate using the Google Maps integration

The marking of the coordinate within the integrated Google Maps application can be changed by the editor. The change within the map display then affects the coordinate saved in the SwingGadget input component.

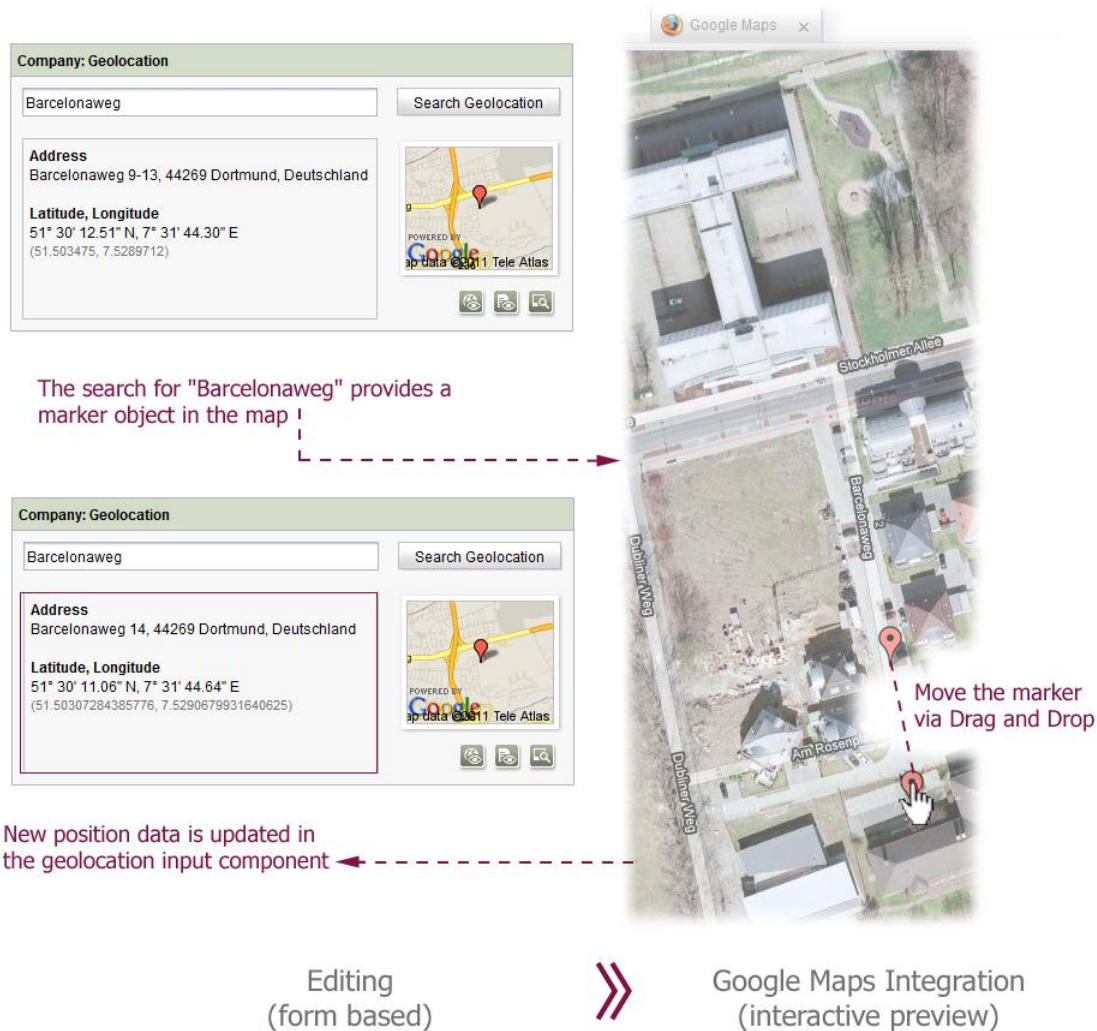



Figure 7: Use case – moving the marker in the map display

The editor must first lock the section concerned to prevent it from being edited. Click the  button to open an (applications) tab with the web application in the application area of the JavaClient. In edit mode the marker set by the example implementation is always displayed within a hybrid map. This applies even if the web application has already been opened in the application area and the editor has selected a display form beforehand.

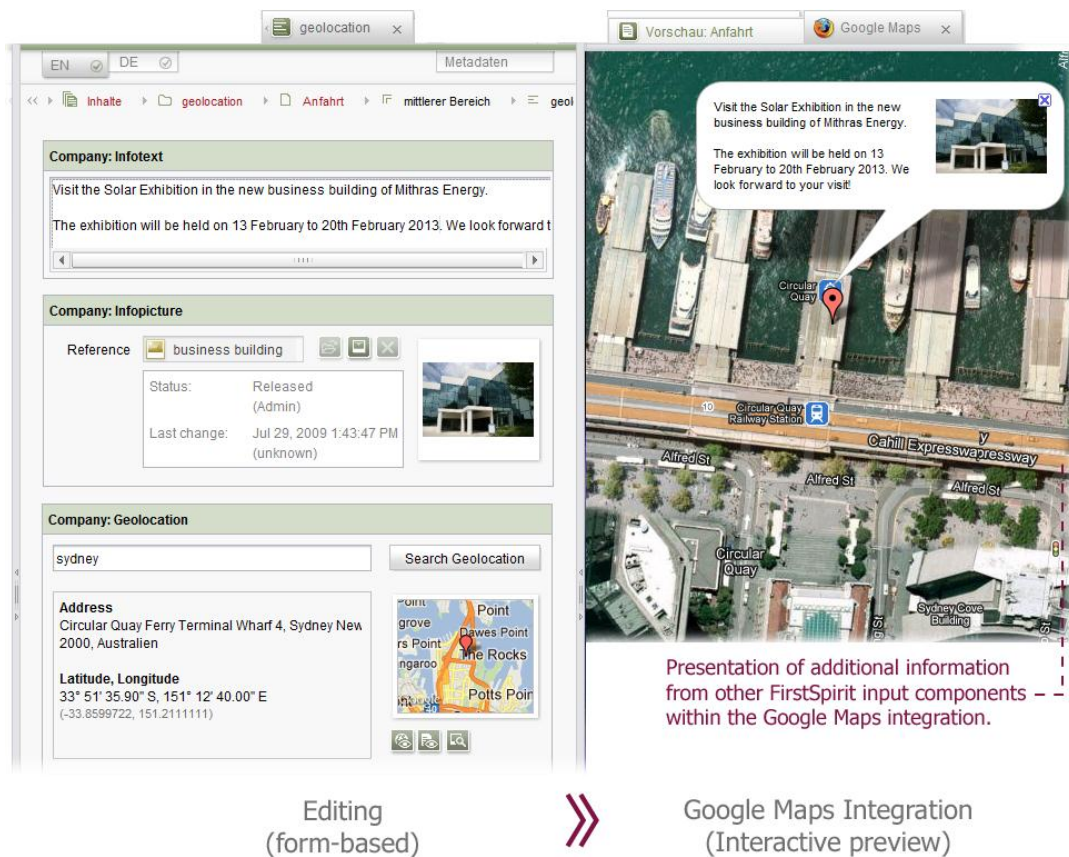




Within the hybrid map the editor can simply move the marker by means of drag-and-drop or via the Google Context menu ("What is here?"). In this way, for example, after searching for a geographic position via the Google Maps API, the marker within the map is corrected and move to the required building. The corrected position data is then copied by the example implementation into the geolocation input component.

4.2.3 Showing additional information (Google Balloons)

Apart from the geographic data, other information on a marker is to be shown within the map. To this end, the form area provides the editor with other input fields (company: infotext and company: infopicture). In this way, each marker saved within a geolocation input component can be assigned short information text and a picture from the FirstSpirit Media Store.



Editing (form-based)



Google Maps Integration (Interactive preview)

Figure 8: Use case – showing additional information

This information entered by the editor is shown as Google "balloons" by clicking the marker, not only within the integrated preview (Preview tab) but also in the integrated web application (Applications tab). These "balloons" are small information windows, which can contain HTML,




CSS or JavaScript code.

4.2.4 3D display using Google Earth

Apart from the conventional display of the coordinate within a hybrid map with low altitude, the input component also offers a 3D view (Google Earth) with high altitude via the Google Maps integration.



This display requires the installation of a Google Earth plug-in (see Chapter 4.1.5 page 41).

If Google Maps is not already open in the integrated application area of FirstSpirit, an Application tab is opened first by clicking the  button. The implementation behind the input component now shows a map of the type G_SATELLITE_3D_MAP instead of the conventional hybrid map. This type of map shows an interactive 3D model of the earth with satellite images. (If the application was already open in the application area, the map display is simply switched to the existing Application tab.)

Within the 3D view the user can conveniently switch between different coordinates (saved in several geolocation input components). On changing between the coordinates (for example, on selecting a new geolocation section via the FirstSpirit navigation tree), 3D fading in and out (aka cross-fading/fading over) from one position to the other takes place in the application tab, with the classic "Zoom-to-Location" effect of Google Earth⁴. All additional information about a marking saved can also be shown within this display (injection of additional information as Google Earth Balloons) (see Chapter 4.2.3). The implementation therefore provides an interactive live preview for the display of geographic information.

⁴ For an example, see <http://earth-api-samples.googlecode.com/svn/trunk/examples/balloon-change-content.html>



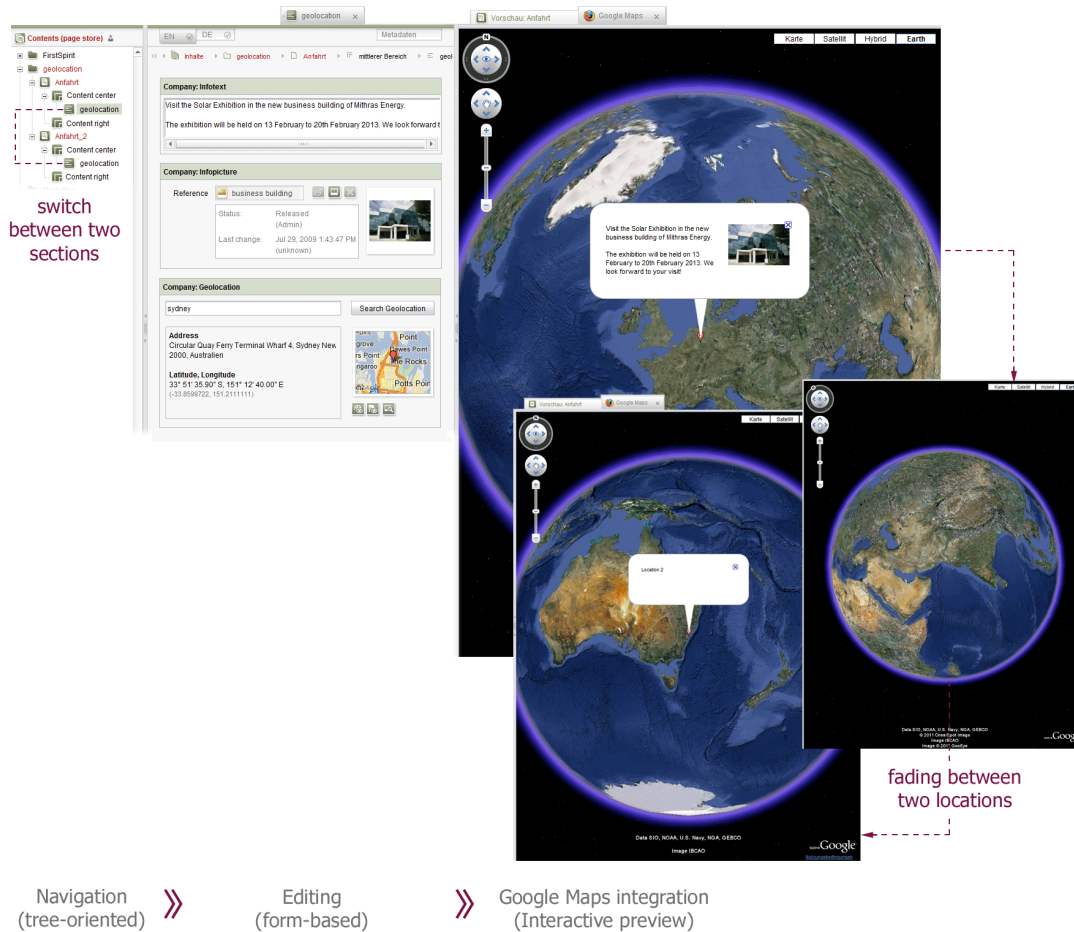


Figure 9: Use case – 3D fading in / out between two positions

4.2.5 Route directions ("How to find us")

The geolocation input component saves the geographic coordinate determined using the Google Maps API for further processing (see Chapter 4.2.1 page 42). This data is used within the website for route planning or directions ("How to find us" instructions in the navigation menu). As soon as a geographic coordinate has been determined using the Google Maps integration (for example, using search) and has been saved in the geolocation input component, a form based on the saved coordinates is generated within the website (or in the Preview tab). The updated website shows a map section (map tile) with the saved coordinates marked on it. In addition, input fields for two integrated route planners (Google Maps and Bing Maps) are displayed with a preselected start location. The coordinates saved within the component are used as the end location/destination. The start location can be changed by the visitor to the website (or by the editor within the application area). When the corresponding button is clicked a Google Maps or Bing Maps window opens with the respective parameters.



The image illustrates the process of generating directions through a form-based interface. It is divided into three main stages:

- Editing (form-based):** A 'geolocation' form is shown with fields for 'Company: Infopicture', 'Company: Geolocation' (containing 'sydney'), 'Address' (Circular Quay Ferry Terminal Wharf 4, Sydney New 2000, Australien), and 'Latitude, Longitude' (33° 51' 35.90" S, 151° 12' 40.00" E). A 'Search Geolocation' button is present.
- Preview (Live Rendering):** A preview of the 'Mithras Energy' website is shown, featuring a map tile with a red pin at the location. A text box explains: 'The website shows a form with a map tile with the saved coordinates marked on it, input fields and buttons for integrated route planners'. Below the map, buttons for 'Route: Google Maps' and 'Route: Bing Maps' are visible.
- Directions:** A Google Maps interface shows a route from 'University of Sydney' to 'Cahill Expressway'. The route is highlighted in blue. A text box explains: 'start route planning: A: the entered start location B: The coordinates saved within the geolocation component are used as the end location/destination'.

Figure 10: Use case - directions



4.3 Implementation: Application integration for Google Maps

In the preceding chapters the concept of the seamless integration of a web application in the FirstSpirit JavaClient (see Chapter 2 page 12) and the necessary extensions of the FirstSpirit Client API were introduced (see Chapter 3 page 20). This chapter, by way of example, now shows a specific implementation of an application integration for the FirstSpirit JavaClient.

The web application Google Maps is to be integrated in the application area of the JavaClient. The integration is controlled by a SwingGadget input component (see Chapter 4.3.1 page 50), which is closely linked to the application. The connection between the Java level of the FirstSpirit JavaClient and the native browser level of the web application is made by means of the new FirstSpirit AppCenter API.

To avoid the implementation of an independent web application, an HTML code is initiated within the example implementation, which initializes a Google Maps container on loading (see maps.html - Initializing the container for the map display Chapter 4.3.15 page 102). Within this HTML page, JavaScript methods are defined, which provide core functions such as "Add entry", "Remove entry" and "Modify viewport". These JavaScript methods are called accordingly on the Java side, in order to display or change a geographic position.

Identification of the individual map entries, IDs are generated, which are used for direct assignment and control (Show markers and assign an input component see Chapter 4.3.6 page 64). A way back (return path), provided by means of an object injection, is required for the modification of the geographic position data and the corresponding change notification (MapsPlugin - GeolocationUpdater (Injection Java » JavaScript) see Chapter 4.3.5 page 61). This object has a method for updating the exact geographic position and a method by means of which the address string can be updated.

The following chapters describe selected code of the example implementation and represent orientation for the user to develop their own integration solutions. In particular, use of the FirstSpirit AppCenter API is explained. The Google Maps API also used is only explained to the extent necessary to understand the example (implementation details are given in the Google Maps API⁵ documentation and the Google Earth API⁶ documentation.) The development of the SwingGadget input component used (see Chapter 4.3.1) is also not described here. All

⁵ <http://code.google.com/intl/de/apis/maps/documentation/javascript/v2/reference.html>

⁶ <http://code.google.com/intl/de/apis/earth/documentation/reference/index.html>



information and interfaces required to develop SwingGadget input components are given in the Developer Manual for Components.

- (SwingGadget) input component `CUSTOM_GEOLOCATION`
(see Chapter 4.3.1 page 50)
- MapsPlugin – Generating a new instance of the type MapsPlugin
(see Chapter 4.3.2 page 51)
- MapsPlugin – Opening the application within a tab
(see Chapter 4.3.3 page 53)
- MapsPlugin – run JavaScript (Java » JavaScript)
(see Chapter 4.3.4 page 57)
- MapsPlugin - GeolocationUpdater (Injection Java » JavaScript)
(see Chapter 4.3.5 page 61)
- Show markers and assign an input component
(see Chapter 4.3.6 page 64)
- Listener – responding to changes
(see Chapter 4.3.7 page 74)
- Updating the geodata of the input component (JavaScript » Java)
(see Chapter 4.3.8 page 79)

The complete source code of the application integration implementation for Google Maps described in this example is located in the Zip archive of the Developer Manual. The archive file can be downloaded from using the online documentation of FirstSpirit (area: Documentation for developers – example implementations).



If using the Google Maps integration the license requirements of the manufacturer must be met (see Chapter 4.1.2 page 39).



4.3.1 (SwingGadget) input component CUSTOM_GEOLOCATION

The application areas are introduced in Chapter 4.2 (page 42 ff.). A new input component `CUSTOM_GEOLOCATION` has been developed for these areas. This input component saves a geographic coordinate, consisting of two decimal values (geographic longitude and latitude) and displays not only the coordinate but also the complete address information belonging to this coordinate (street, town, country). Furthermore, a simple text field for (unformatted) address input is available to the editor, to simplify the search. The address string enter there is used for a text-based search by means of the Google Maps API. As Google Maps includes (provided it is possible/known) the location of the Internet access point, from which the request is made, in this search, for destinations near the access point location it can be sufficient to enter a street name.

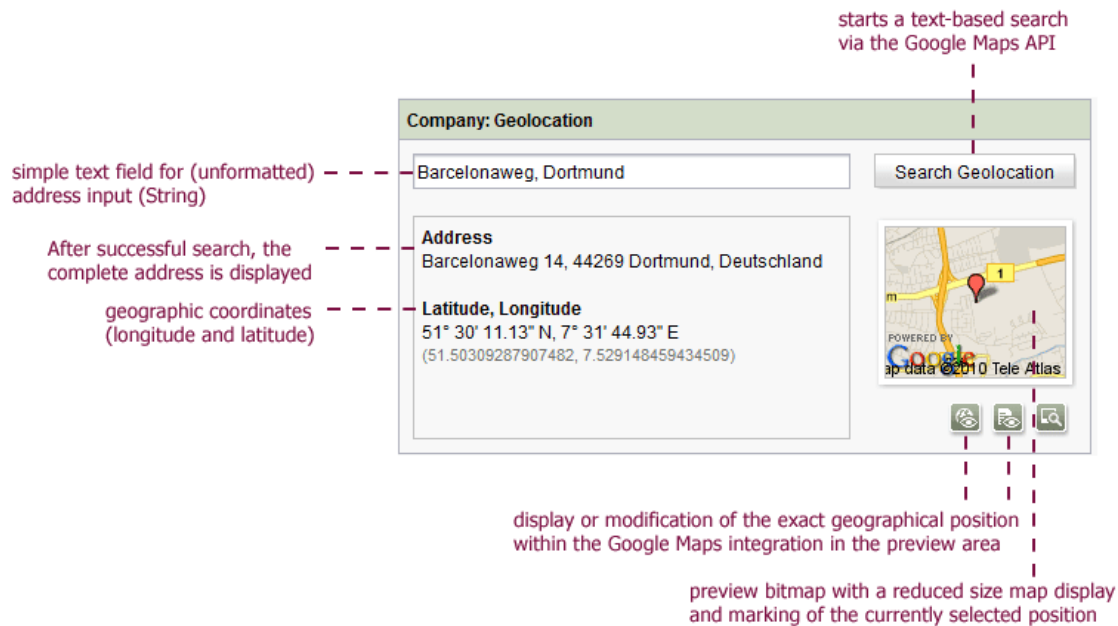


Figure 11: Geolocation input component

The special feature of the geolocation input component lies in its close integration with the web application Google Maps. The component includes a button for display/modification of the precise geographic position and a button which initiates a search on the basis of the address string. With both actions - if it has not already happened or been closed - an application tab is opened in the integrated browser of the JavaClient (MapsPlugin – Opening the application within a tab see Chapter 4.3.3 page 53). The web application for displaying the geocoordinate is opened in this tab. The user can change the displayed coordinate there (for example, using drag-and-drop to move the marking within the map display). The new coordinate (incl. address information) is then dynamically updated in the input component. In addition, the input component shows a preview bitmap with a reduced size map display and marking of the currently selected position from the web application. This preview image does not come from the



FirstSpirit Media Store, but instead is provided via the integrated web application and is also dynamically updated if the position within the integrated application (Google Maps) is changed. The opening of the application tabs and live updating of the map display is carried out by the implementation behind the input component (see Chapter 4.3.2 ff.).

4.3.2 MapsPlugin – Generating a new instance of the type MapsPlugin

Within the SwingGadget implementation (of the input component `CUSTOM_GEOLOCATION`) a new instance of the type `MapsPlugin` must be generated first. The `MapsPlugin` class is responsible for the integration of Google Maps in the application area of the FirstSpirit JavaClient.

This requires a so-called `SpecialistsBroker`. An instance of the type `SpecialistsBroker` provides access to specific services or information via different "specialists". A specialist of the type `ServicesBroker` is required for working with integrated web applications. This can be requested on the `SpecialistsBroker` with the help of the `<S> S requireSpecialist(SpecialistType<S> type)` method. In addition, a service can be requested by calling the `getService()` method. This method is required, for example, later in order to use the `ApplicationService` (see Chapter 4.3.3 page 53).

The typed `SwingGadgetContext` (see Developer Manual for Components) can be used within the SwingGadget implementation to request an instance of the type `ServicesBroker` within the help of the `<S> S requireSpecialist(SpecialistType<S> type)` method. The new instance of the type `ServicesBroker` is finally passed to the `MapsPlugin` class when the `MapsPlugin.getInstance(...)` method is called.

```
1. public class GeolocationSwingGadget...{
2.
3.     private final SwingGadgetContext<GomGeolocation> _context;
4.     private ServicesBroker _servicesBroker;
5.     ...
6.
7.     public GeolocationSwingGadget(final
8.         SwingGadgetContext<GomGeolocation> context) {
9.         super(context);
10.        _context = context;
11.    }
12.    ...
13.    private MapsPlugin getMapsPlugin() {
14.        if (_servicesBroker == null) {
```



```
        _context.getBroker().requireSpecialist(ServicesBroker.TYPE);
15.     }
16.     return MapsPlugin.getInstance(_servicesBroker);
17. }
18. }
```

Listing 1: Geolocation - generating a new instance of MapsPlugin (SwingGadget-Impl.)

The `MapsPlugin.getInstance(...)` method generates a Singleton instance of the `MapsPlugin` class. This Singleton instance initially has a `ServicesBroker` (which is requested within the `SwingGadget` implementation and is passed to the `MapsPlugin` class) and the `MapType` `G_HYBRID_MAP`. This `MapType` is a default map type of the Google Maps API, which represents a mixture of photo tiles and additional information, for example, street or place names.


```
1.  public class MapsPlugin implements TabListener, BrowserListener {
2.
3.      @SuppressWarnings({"UnusedDeclaration"})
4.      public enum MapType {
5.          G_NORMAL_MAP, G_SATELLITE_MAP, G_HYBRID_MAP,
6.          G_SATELLITE_3D_MAP
7.      }
8.      private final ServicesBroker _servicesBroker;
9.      private static MapsPlugin INSTANCE;
10.
11.     ...
12.     private MapsPlugin(final ServicesBroker servicesBroker) {
13.         _servicesBroker = servicesBroker;
14.         _mapType = MapType.G_HYBRID_MAP;
15.     }
16.
17.     public static MapsPlugin getInstance(final ServicesBroker
18.         servicesBroker) {
19.         if (INSTANCE == null) {
20.             INSTANCE = new MapsPlugin(servicesBroker);
21.         }
22.         return INSTANCE;
23.     }
```

Listing 2: Geolocation – MapsPlugin constructor (MapsPlugin-Impl.)



4.3.3 MapsPlugin – Opening the application within a tab

In order to integrate a web application into the FirstSpirit JavaClient, it is necessary to control the browser integrated in FirstSpirit (Controlling the integrated browser see Chapter 2.1.1 page 12). To do this, firstly, a new browser instance (or a new tab) must be generated within the application area, in which the required web application can be opened.

On requesting the integrated web application, for example, by clicking the  button of the geolocation component, the program first checks whether a browser instance of the web application already exists in the application area of the JavaClient. This condition is checked by calling the `ensureApplicationTab()` method within the MapsPlugin implementation:

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     private BrowserApplication _application;
3.     ...
4.     public void add(final GadgetIdentifier gadgetId, ...) {
5.         ensureApplicationTab();
6.         ...
7.     }
8.
9.     ...
10.    //inner method
11.    private void ensureApplicationTab() {
12.        if (_application == null) {
13.            getApplication(); // open application tab
14.        }
15.    }
16. }
```

Listing 3: Geolocation –BrowserApplication instance available? (MapsPlugin-Impl.)

If a browser instance has not yet existed for the web application, the `getApplication()` method is called, which returns a new instance of the web application of the type `BrowserApplication`.

The interfaces of the FirstSpirit AppCenter API required are described in the following:

- 1) `ApplicationService`: The entry point for controlling a tab is always the `ApplicationService`. An instance of the type `ApplicationService` is requested by calling the `<T> T getService(Class<T> serviceClass)` method on the passed `ServicesBroker` (see Chapter 4.3.2). This service can be used to open new applications of a certain type within the application area (see section 3) or to get the



applications from existing browser instances (see section 5) (for a description of the `ApplicationService` interface see Chapter 3 page 20).

- 2) `BrowserApplicationConfiguration`: In addition, a configuration is required for the integrated browser instance. For this, an instance of the type `BrowserApplicationConfiguration` is generated using the call `BrowserApplicationConfiguration.GENERATOR.invoke()`. The `BrowserApplicationConfiguration` interface provides methods, for example, for selecting a specific browser engine for the integration or for showing or hiding an address line within the browser instance. The configuration is passed as a parameter when a new browser instance is opened using the `openApplication(...)` method (see section 3) (for a description of the `BrowserApplication Configuration` interface see Chapter 3.9 page 35).
- 3) `ApplicationTab`: To integrate a web application, first, a new tab (next to the Preview tab) must be opened in the application area of the JavaClient. To do this, the `openApplication(...)` method is called on the `ApplicationService` (see section 1). The type (Abstract Class: `ApplicationType` see Chapter 3.6 page 31) of application required (here: `BrowserApplication`, see section 5) and the configuration for the integrated browser are passed to the method (see section 2). The `openApplication(...)` method returns an instance of the type `ApplicationTab`. The `ApplicationTab` interface provides general methods for controlling the tab, for example, the tab can be brought to the front or closed using the corresponding method invocations (Interface: `ApplicationTab` see Chapter 3.2 page 23).
- 4) `TabListener`: To track changes, for example, the selection of a tab by the user, an instance of the type `TabListener` must be added to the Application tab (Interface: `TabListener` see Chapter 3.5 page 30). A `TabListener` is added to the Applications tab by calling the `addTabListener(...)` method; in this example the `MapsPlugin` class is implemented by the `TabListener` interface itself. However, it is recommended that you use the corresponding adapter implementation. The methods of the interface for selecting, deselecting and closing the application tab can then be implemented as and when necessary (see Listener – responding to changes, Chapter 4.3.7, page 74).
- 5) `BrowserApplication`: By calling the `getApplication()` method on the instance of the type `ApplicationTab`, a new instance of the integrated web application of the type `BrowserApplication` is returned (see section 3 – `openApplication(...)`). The `BrowserApplication` interface provides methods for controlling the new integrated browser instance, for example, opening a URL within the browser instance (Interface:



BrowserApplication see Chapter 3.7 page 31). The required web application (here: Google Maps) must then be opened in the new browser instance generated. Two different ways are available for this integration:

- a) On the one hand, a global web application can be implemented and installed on the FirstSpirit server. In this case the URL of the web application can be called in the application area (using the `openUrl(...)` method). This method is also used to open an external web application.
 - b) If an independent web application is to be bypassed, but the required HTML code can be easily initiated and called (using the `setHtmlContent(...)` method). This second way is also used for the Google Maps integration introduced here. The `maps.html` file is used to initiate the HTML code; this forms a kind of capsule around the Google API calls required (see Chapter 4.3.12 ff.).
- 6) `BrowserListener`: Access to the content shown in the web browser (DOM tree), for example, to manipulate the data, is not possible at any time. To ensure controlled access to the content, an instance of the type `BrowserListener` must be used (Interface: `BrowserListener` see Chapter 3.8 page 34). A `BrowserListener` is added to the `BrowserApplication` by calling the `addBrowserListener(...)` method. In this example the `MapsPlugin` class implements the `BrowserListener` interface itself. However, it is recommended that you use the corresponding adapter implementation. The methods of the interface can then be implemented as and when necessary (see Listener – responding to changes, Chapter 4.3.7, page 74).




```
1. private BrowserApplication _application;
2. private ApplicationTab<BrowserApplication> _tab;
3. private final ServicesBroker _servicesBroker;
4.
5. /**
6.  * Get BrowserApplication instance and may initialize it.
7.  * @return BrowserApplication instance
8.  */
9. private BrowserApplication getApplication() {
10.     if (_application == null) {
11.         final ApplicationService service =
12.             _servicesBroker.getService(ApplicationService.class);
13.         final BrowserApplicationConfiguration configuration =
14.             BrowserApplicationConfiguration.GENERATOR.invoke();
15.         configuration.title("Google Maps");
16.         configuration.showAddressBar(false);
17.         configuration.engineType(EngineType.FIREFOX);
18.         _tab = service.openApplication(BrowserApplication.TYPE,
19.             configuration);
20.         _tab.addTabListener(this);
21.         _application = _tab.getApplication();
22.         _application.addBrowserListener(this);
23.         _application.setHtmlContent(getHtmlContent());
24.     }
25.     return _application;
26. }
```

Listing 4: Geolocation – opening the application within a tab (MapsPlugin-Impl.)



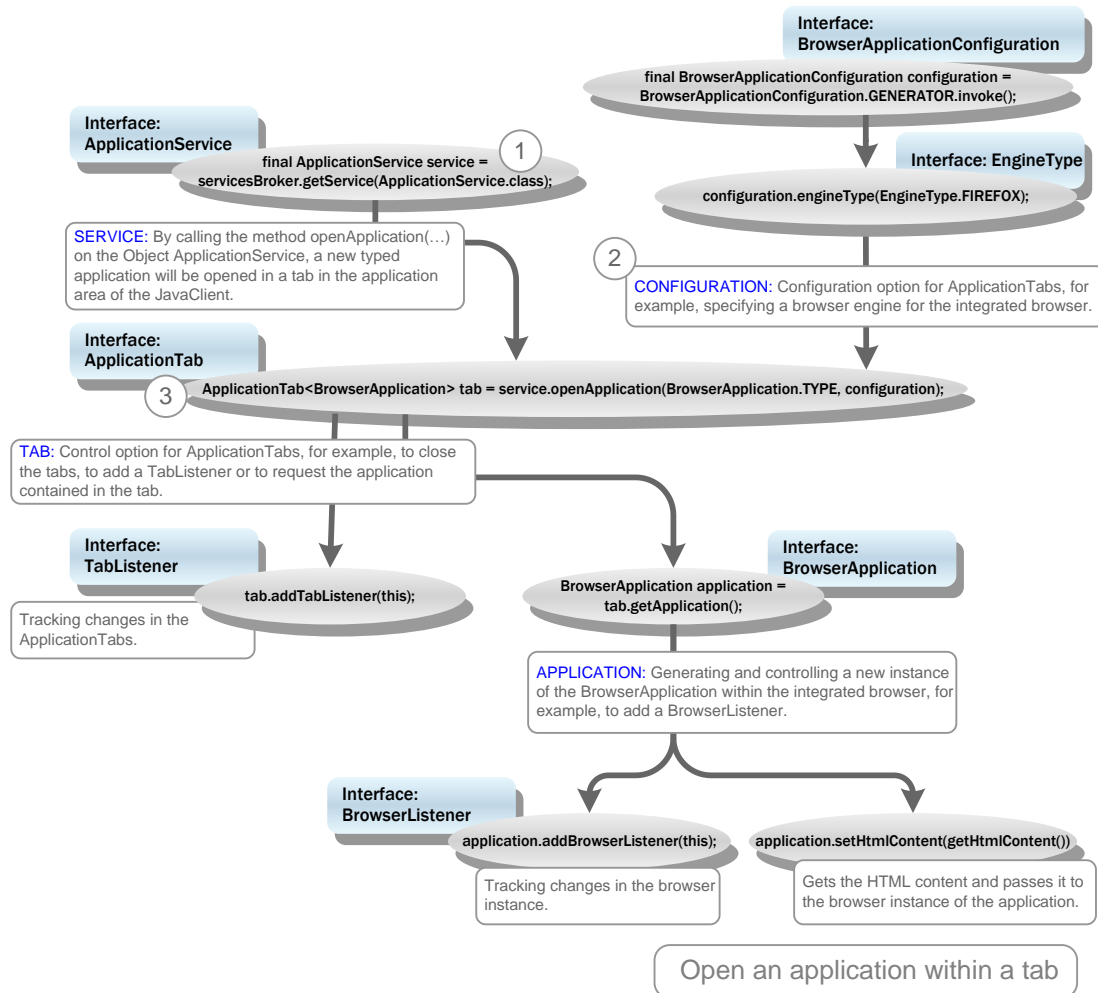


Figure 12: Opening an application within a tab

4.3.4 MapsPlugin – run JavaScript (Java » JavaScript)

As already explained in Chapter 2.1, interfaces must be created for the communication between the Java level of the FirstSpirit JavaClient and the native level of the integrated browser (or the Google Maps API) (see concept in Chapter 2.1.2 (page 13 ff.)).

For example, if a certain address is searched for within the geolocation input component (entry of an address string and click the Search button), a request for geocoding of this address string must be sent to the web application (Google Maps) and the map section adjusted within the integrated browser (see use case Chapter 4.2.1 page 42). This requires an interface, which enables a JavaScript method to be called from the Java environment (communication: Java » JavaScript).



The geolocation implementation uses the `void executeScript(String script)` method for this. This method runs the passed JavaScript code in the currently opened browser document and therefore enables targeted, unidirectional communication in the direction: Java » JavaScript. The JavaScript code to be run is passed as a string. The method is made available to the FirstSpirit AppCenter API via the `BrowserApplication` interface (Interface: `BrowserApplication` see Chapter 3.7 page 31).

However, first the JavaScript code required must be created. To this end, within the `MapsPlugin` implementation, numerous methods are implemented which, depending on the respective use case, compile a script using the `StringBuilder`. The `void updateBrowser()`, method, which is used to update the map display in the browser, for example, calls different methods with the prefix "getScript" in the method name, which return individual JavaScript fragments. For the use case "Address search" for example, the `getScriptFindAddress(...)` method is called there:

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     ...
3.     public void updateBrowser() {
4.         // build script code to update google map
5.         final StringBuilder buf = new StringBuilder();
6.         ...
7.         for (final GeolocationEntry location : ...) {
8.             if (location.isSearchMode()) {
9.                 buf.append(getScriptFindAddress(location));
10.            }
11.            ...
12.        }
13.        final String script = buf.toString();
14.        ...
15.    }
16.    ...
17.
18.    //--- inner methods ---//
19.
20.    /**
21.     * Build script code for an address search entry.
22.     *
23.     * @param location related geolocation entry
24.     * @return script code
25.     */
26.    private String getScriptFindAddress(final GeolocationEntry
```



```

        location) {
27.         final StringBuilder buf = new StringBuilder();
28.         final String pattern = location.getAddress().replaceAll("'",
            "\\\\"');
29.         buf.append("window.findAddress('")
30.             .append(location.getUUID()).append(", ")
31.             .append(pattern).append("');");
32.         return buf.toString();
33.     }
34.
35. }

```

Listing 5: Geolocation – create script for the geo-coding of an address string (MapsPlugin-Impl.)

A so-called `GeolocationEntry` is passed to the method (see Chapter 4.3.6 page 64). This entry contains, among other things, the address string, which was entered in the search field of the geolocation input component. The `getScriptFindAddress(...)` method now uses this information to compile a JavaScript code, which not only contains the address string but also the `uuid` of the corresponding SwingGadget input component. The `uuid` is required, in order to establish clear, unique assignment of the `GeolocationEntries` to an input component (see Chapter 4.3.6 page 64).

The script fragment returned by the `getScriptFindAddress(...)` method, for example, looks like this (for information on the `window` object, see Chapter 4.3.13):

```

window.findAddress('uuid:-1336359343', 'Barcelonaweg');

```

Other methods are also called within the `void updateBrowser()` method. The returned JavaScript fragments are then compiled to form a script, for example:

```

window.clearOverlays();
window.setMapType(G_HYBRID_MAP);
window.findAddress('uuid:-1336359343', 'Barcelonaweg');
window.setEditable('uuid:-1336359343', true);
window.setInfoHtml('uuid:-1336359343', "<span ... />...</span>");

```

Each of these calls has its equivalent in a JavaScript function of the `maps.html` file (`maps.html` – Introduction see Chapter 4.3.12 page 99). The call `window.findAddress('uuid:-1336359343', 'Barcelonaweg')` for example, runs the following JavaScript function:

```

1.  /**
2.   * Use Geocoder to find the exact geolocation of the specified address
3.   * pattern
4.   *

```



```
5.     * @param uuid UUID of geolocation instance
6.     * @param pattern address string
7.     */
8.
9.     window.findAddress = function(uuid, pattern) {
10.         window.GoogleGeocoder.getLatLng(pattern, function(point) {
11.             if (point) {
12.                 var latitude = point.lat();
13.                 var longitude = point.lng();
14.                 window.addOverlay(uuid, latitude, longitude);
15.                 window.setViewPoint(latitude, longitude);
16.                 window.GeolocationUpdater.update(uuid, latitude,
17.                     longitude);
18.                 updateInfo(uuid, latitude, longitude);
19.             }
20.         });
21.     };
```

Listing 6: Geolocation – JavaScript function findAddress (maps.html)

This function starts a geocoding request via the Google Maps API. The precise procedure is described in Chapter 4.3.20 (page 109). Other calls, for example, remove the markings (selections) to date from the map display (`window.clearOverlays()`) or define the MapType for the map display (`window.setMapType (G_HYBRID_MAP)`).

The entire script with all the functional calls it contains, which is compiled within the `void updateBrowser()` method, is then run by calling the `void executeScript(String script)` method on the instance of the type `BrowserApplication` (Interface: `BrowserApplication` see Chapter 3.7 page 31):

```
1.     public class MapsPlugin implements TabListener, BrowserListener {
2.         ...
3.         private BrowserApplication _application;
4.
5.         public void updateBrowser() {
6.             final StringBuilder buf = new StringBuilder();
7.             buf.append(getScriptClearOverlays());
8.             ...
9.             buf.append(getScriptFindAddress(location));
10.            ...
11.            final String script = buf.toString();
12.            ...
```



```

13.         _application.executeScript(script);
14.     }
15. }

```

Listing 7: Geolocation – run the collated JavaScript fragments (MapsPlugin-Impl.)

With this the first communication channel (Java » JavaScript) is adequately described. In the second step, the information from the web application must be returned back into the FirstSpirit input component (JavaScript » Java) (see Chapter 4.3.5 page 61).

4.3.5 MapsPlugin - GeolocationUpdater (Injection Java » JavaScript)

In the previous chapter, the communication was described starting from the FirstSpirit JavaClient in the native level of the integrated browser by running JavaScript code (see Chapter 4.3.4 page 57). This chapter looks at the reverse communication channel (JavaScript » Java). Starting from the integrated web application (or the integrated browser instance), changes or events, for example, due to moving of the marking within the map display, should also be updated in the geolocation input component. In this case, the Java side must be informed of the change in the web applications and must be able to suitably respond to this change (see concept Communication between the browser instance and JavaClient in Chapter 2.1.2, page 13)

To update the Java side, the Java object `GeolocationUpdater` is used within the example implementation. A new instance of the type `GeolocationUpdater` is generated with the `void onDocumentComplete(final String url)` method is run and is injected into the web application under the name "GeolocationUpdater".

Background: Access to the DOM tree of the browser instance is not possible at any time. The injection can only take place if the document has been fully loaded. To ensure this, an instance of the type `BrowserListener` must be used (Interface: `BrowserListener` see Chapter 3.8 page 34). The `void onDocumentComplete(...)` method is always called if the `BrowserListener` of the browser instance (instance of the type `BrowserApplication`) reports that the document (including all images) has been completely loaded (DOM access concept see Chapter 2.2 page 18) (Listener – responding to changes, see Chapter 4.3.7, page 74).

```

1.     public class MapsPlugin implements TabListener, BrowserListener {
2.         ...
3.         private boolean _active;
4.         ...
5.
6.         //--- BrowserListener ---//
7.         ...

```



```
8.     public void onDocumentComplete(final String url) {
9.         final BrowserApplication application = getApplication();
10.
11.         // document is loaded; inject java/javascript bridge
12.         application.inject(new GeolocationUpdater(this),
13.             "GeolocationUpdater");
14.
15.         _active = true;
16.
17.         // initialize map viewport and marker
18.         updateBrowser();
19.         ...
20.     }
```

Listing 8: Geolocation – injection of Java object GeolocationUpdater (MapsPlugin-Impl.)

On an instance of the type `BrowserApplication` the `void inject(Object object, String name)` method is called. The method is made available to the FirstSpirit AppCenter API via the `BrowserApplication` interface (Interface: `BrowserApplication` see Chapter 3.7 page 31).

The method injects the passed Java object `GeolocationUpdater` as an attribute of the window object in the browser instance on which it was called (for information on the `window` object see Chapter 4.3.13). The injection generates a substitute object (proxy) in the form of a JavaScript object and registers it under the passed name (here: "GeolocationUpdater"). Following registration the JavaScript object can be used in the `maps.html` file by calling `window.{name}` (here: `window.GeolocationUpdater`) (see Chapter 4.3.21 page 110). All methods of the Java object `GeolocationUpdater` can then also be called from the JavaScript environment of the integrated browser.

Background: With the injection the FirstSpirit framework generates for each method of the Java object instances a corresponding JavaScript method with (approximately) identical method signature. On being called from the JavaScript environment, this JavaScript method sends an event, which is evaluated on the Java side and triggers the running of the corresponding Java method there. The suitable Java method is determined from the method signature and the passed parameters and is called.

The Java object `GeolocationUpdater`, for example, has the Java method `public void update(final String uuid, final double latitude, final double longitude)`, which updates a `GeolocationEntry` on the passed coordinate and informs the `ModificationListener` belonging to the input component of the update (use of the



ModificationListener see Chapter 4.3.7.3).

```
1.     @SuppressWarnings({"UnusedDeclaration"})
2.     public static class GeolocationUpdater {
3.
4.         private final MapsPlugin _mapsPlugin;
5.
6.
7.         public GeolocationUpdater(final MapsPlugin mapsPlugin) {
8.             _mapsPlugin = mapsPlugin;
9.         }
10.
11.
12.         public void update(final String uuid, final double latitude,
13.             final double longitude) {
14.             _mapsPlugin.update(uuid, latitude, longitude);
15.         }
16.
17.         public void info(final String uuid, final String address) {
18.             _mapsPlugin.info(uuid, address);
19.         }
20.     }
21. }
```

Listing 9: Geolocation – GeolocationUpdater (MapsPlugin-Impl.)

Following the injection of the Java object `GeolocationUpdater` the update method can be called on the JavaScript- proxy object `window.GeolocationUpdater`. Within the example implementation, the updating is triggered by an event in the web application (for example, by moving the marker in the map display or the geocoding of an address string). In this case, the corresponding `EventListener` on the JavaScript side calls the JavaScript function `window.GeolocationUpdater.update (uuid, newlat, newlng)`.

```
1.     /**
2.      * Add an marker to the GMap2 instance.
3.      *
4.      * @param uuid UUID of geolocation instance
5.      * @param latitude
6.      * @param longitude
7.      */
8.     window.addOverlay = function(uuid, latitude, longitude) {
```





```
9.     var marker = new GMarker(new GLatLng(latitude, longitude),
    {draggable: true});
10.    window.GoogleMap.addOverlay(marker);
11.    GEvent.addListener(marker, 'dragend', function() {
12.        var point = marker.getLatLng();
13.        var newlat = point.lat();
14.        var newlng = point.lng();
15.        window.GeolocationUpdater.update(uuid, newlat, newlng);
16.        updateInfo(uuid, newlat, newlng);
17.    });
18.    ...
19.    };
```

Listing 10: Geolocation – calling a Java method from the JavaScript code (maps.html-Impl.)

This call is evaluated on the Java side where it triggers the running of the Java method `public void update(final String uuid, final double latitude, final double longitude)`. The Java method receives the (new) address information from the integrated web application, for example, which is determined during the geocoding of an address string by the Google Maps API (JavaScript » Java). The renewed assignment of the entry for the geolocation input location in the Java environment is made using the `uuid` parameter (see `GeolocationEntry get(final String uuid)` method in Chapter 4.3.6, page 64).

Basically, any Java object can be injected into the JavaScript environment. However, certain restrictions apply to this object for the method adoption and the mapping of the Java data types on JavaScript data types (see Chapter 2.1.3 page 16).

4.3.6 Show markers and assign an input component

The input component `CUSTOM_GEOLOCATION` can be used to display a geographic coordinate, consisting of two decimal values (geographic longitude and latitude) within a Google Maps map (see (SwingGadget) input component `CUSTOM_GEOLOCATION` in Chapter 4.3.1, page 50). For this, the Google Maps API uses an object of the class `GMarker`. This marker object contains the values for the geographic coordinate and displays this within a map . The object is added to the map display with the help of the Google Maps method `addOverlay()`⁷.

⁷ For further information see Google Maps API reference:

<http://code.google.com/intl/de/apis/maps/documentation/javascript/v2/reference.html#GMarker>



```
1.    var marker = new GMarker(new GLatLng(latitude, longitude), {draggable:
      true});
2.    window.GoogleMap.addOverlay(marker);
```

Listing 11: Geolocation – adding a new GMarker object (maps.html)

The Java object `GeolocationEntry` is used on the Java side. An instance of the type `GeolocationEntry` not only contains the values, which describe the coordinate (`double latitude`, `double longitude`) but also an instance of the type `GadgetIdentifier`.

Background: In order for a marker to be uniquely assigned to an input component, a `GadgetIdentifier` is used within the Java implementation. A `GadgetIdentifier` is provided by the FirstSpirit Framework and is used to uniquely identify a named form element within the FirstSpirit component model. The `GadgetIdentifier` of a `SwingGadget` input component can be got using the `GadgetIdentifier getGadgetId()` method of the abstract class `AbstractValueHoldingSwingGadget <T, F extends GomFormElement>`.

Within the example implementation, each `GeolocationEntry` can therefore be uniquely assigned to a specific `SwingGadget` input component using a `GadgetIdentifier`. This means that if the input component is changed, for example, by selecting a new section in the FirstSpirit navigation tree, the marker to date within the Google Maps integration is removed and replaced with the new marker (the currently selected input component).

Furthermore, a `GeolocationEntry` has a `ModificationListener`, which responds to changes to the entry within the input component (see `Listener – responding to changes`, Chapter 4.3.7, page 74).

```
3.    public class MapsPlugin implements TabListener, BrowserListener {
4.        ...
5.        private static class GeolocationEntry {
6.            private final String _uuid;
7.            private final GadgetIdentifier _gadgetId;
8.            private ModificationListener _listener;
9.            private double _latitude;
10.           private double _longitude;
11.           ...
12.
```



```

13.     GeolocationEntry(final GadgetIdentifier gadgetId, final double
        latitude, final double longitude, final ModificationListener `
        listener) {
14.         _uuid = "uuid:" + gadgetId.hashCode();
15.         _gadgetId = gadgetId;
16.         _latitude = latitude;
17.         _longitude = longitude;
18.         _listener = listener;
19.     }
20.     ...
21. }
22. }

```

Listing 12: Geolocation – GeolocationEntry constructor (MapsPlugin-Impl.)

The addition of a marker to the Google Maps map is initiated from the Java environment of the SwingGadget input component. The input component `GeolocationSwingGadget`, triggered by an event, calls the `addToMap(...)` method. This method first calls the `getGadgetId()` method to get the `GadgetIdentifier` of the component. The `MapsPlugin` implementation is then called, which checks whether an entry for the SwingGadget input component with this `GadgetIdentifier` already exists. If no entry exists, the `getMapsPlugin().add(...)` method is called to add a new entry (of the type `GeolocationEntry`). **The method not only passes the geographic longitude and latitude for the entry but also the GadgetIdentifier of the input component to the MapsPlugin implementation:**

```

1.     public class GeolocationSwingGadget extends
        AbstractValueHoldingSwingGadget<...> implements ...{
2.         ...
3.         public JComponent getComponent() {
4.             @Override
5.             public void actionPerformed(final ActionEvent e) {
6.                 if (!isDefault()) {
7.                     addToMap(GeolocationSwingGadget.this.getValue());
8.                     ...
9.                 }
10.            }
11.        }
12.        ...
13.        private void addToMap(final Geolocation value) {
14.            final GadgetIdentifier gadgetId = getGadgetId();
15.            if (!getMapsPlugin().contains(gadgetId)) {
16.                getMapsPlugin().add(gadgetId, value.getLatitude(),

```



```

        value.getLongitude(), this);
17.     }
18.     ...
19.     }
20.     updateMapsPlugin (gadgetId) ;
21. }
22. ...
23. }

```

Listing 13: Geolocation – passing the GadgetIdentifier to MapsPlugin (SwingGadget-Impl.)

Within the MapsPlugin implementation, the inner method `GeolocationEntry get(final GadgetIdentifier gadgetId)` is called first. This returns the `GeolocationEntry`, which is assigned to the input component with the passed `GadgetIdentifier`. If a `GeolocationEntry` still does not exist for this input component (`entry==null`), a new entry is created (`new GeolocationEntry(gadgetId, ...)`). The `GadgetIdentifier` of the input component (among other things) is passed to the constructor.

```

1.  public class MapsPlugin implements TabListener, BrowserListener {
2.      ...
3.      /**
4.       * Checks if registered geolocation entries contains one with the
5.       * specified gadget id.
6.       *
7.       * @param gadgetId gadget id to search for
8.       * @return {@code true} if the given identifier exists in this maps
9.       * plugin, {@code false} otherwise
10.     */
11.     public boolean contains(final GadgetIdentifier gadgetId) {
12.         return _application != null && get(gadgetId) != null;
13.     }
14.
15.     /**
16.     * Register a geolocation instance with the specified
17.     * latitude/longitude position and modification listener.
18.     *
19.     * @param gadgetId gadget id of geolocation
20.     */
21.     public void add(final GadgetIdentifier gadgetId, ...) {
22.         ensureApplicationTab();
23.         GeolocationEntry entry = get(gadgetId);
24.         if (entry == null) {

```



```
25.         entry = new GeolocationEntry(gadgetId, ...);
26.         synchronized (_entries) {
27.             _entries.add(entry);
28.         }
29.     } else {
30.         ...
31.     }
32. }
33.
34. ...
35. private GeolocationEntry get(final GadgetIdentifier gadgetId) {
36.     synchronized (_entries) {
37.         for (final GeolocationEntry entry : new
38.             ArrayList<GeolocationEntry>(_entries)) {
39.             if (entry.getGadgetId().equals(gadgetId)) {
40.                 return entry;
41.             }
42.         }
43.     }
44.     return null;
45. }
46. ...
47. private static class GeolocationEntry {
48.     private final String _uuid;
49.     private final GadgetIdentifier _gadgetId;
50.
51.     GeolocationEntry(final GadgetIdentifier gadgetId, ...) {
52.         _uuid = "uuid:" + gadgetId.hashCode();
53.         _gadgetId = gadgetId;
54.         ...
55.     }
56.
57.     ...
58. }
59. }
```

Listing 14: Geolocation – creating a new GeolocationEntry (MapsPlugin-Impl.)

A new instance of the type `GeolocationEntry`, in addition to the passed `GadgetIdentifier`, for the unique assignment of the entry to a `SwingGadget` input component on the Java side, contains another variable `uuid` (universal unique identifier), for unique



assignment of the entry on the JavaScript side. Within the example implementation, this variable is assigned a unique hash value, which is based on the hash value of the corresponding `GadgetIdentifiers` and is returned via the `int hashCode()` method of the `GadgetIdentifier` class.

Background: Communication between the Java environment of the FirstSpirit JavaClient and the native level of the integrated browser takes place via JavaScript. The JavaScript code to be run is passed as a string. The additional variable is necessary, as an instance of the type `GadgetIdentifier` cannot be converted into an object of the type `String`. (MapsPlugin – run JavaScript (Java » JavaScript) see Chapter 4.3.4 page 57).

Initially therefore, only one Java object exists (of the type `GeolocationEntry`), which contains the required coordinate and information for assignment of this coordinate to an input component. In the next step this information must be assigned to a Google GMarker object within the JavaScript environment. The insertion of a marker (GMarker) in the map display is controlled by a JavaScript function. The corresponding call is first compiled within the MapsPlugin implementation and is then run using the `void executeScript(String script)` method (see Chapter 4.3.4).

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     ...
3.     /**
4.      * Updates browser and google map instance.
5.      * This method must be called to update geolocation
6.      * entries and viewport.
7.      */
8.     public void updateBrowser() {
9.         ...
10.        final GeolocationEntry location;
11.        buf.append(getScriptAddOverlay(location));
12.        ...
13.        final String script = buf.toString();
14.        ...
15.        _application.executeScript(script);
16.    }
17.
18.    /**
19.     * Build script code to adding a geolocation entry to map.
20.     *
21.     * @param location related geolocation entry
```



```
22.     * @return script code
23.     */
24.     private String getScriptAddOverlay(final GeolocationEntry location)
25.     {
26.         final StringBuilder buf = new StringBuilder();
27.         buf.append("window.addOverlay('")
28.             .append(location.getUUID()).append(",")
29.             .append(location.getLatitude()).append(',')
30.             .append(location.getLongitude()).append(")");
31.         return buf.toString();
32.     }
```

Listing 15: Geolocation – passing information from the GeolocationEntry object (MapsPlugin-Impl.)

The script fragment returned by the `getScriptAddOverlay(...)` method looks like this (for information on the `window` object, see Chapter 4.3.13):

```
window.addOverlay('uuid:-336359343',51.50297,7.52914);
```

This call is used not only to pass the latitude and longitude value but also the uuid value of the current `GeolocationEntry`s from the Java implementation to the JavaScript implementation. The corresponding JavaScript function `window.addOverlay = function(uuid, latitude, longitude)` from the `maps.html` file accepts this parameter and, based on this information, inserts a new `GMarker` object in the map display⁸.

```
1.     window.Mapping = {}; // UUID <> GMarker instance
2.     /**
3.     * Add an marker to the GMap2 instance.
4.     *
5.     * @param uuid UUID of geolocation instance
6.     * @param latitude
7.     * @param longitude
8.     */
9.     window.addOverlay = function(uuid, latitude, longitude) {
```

⁸ For further information see Google Maps API reference:

<http://code.google.com/intl/de/apis/maps/documentation/javascript/v2/reference.html#GMarker>



```
10.         var marker = new GMarker(new GLatLng(latitude, longitude),
11.             {draggable: true});
12.         window.GoogleMap.addOverlay(marker);
13.         GEvent.addListener(marker, 'dragend', function() {
14.             var point = marker.getLatLng();
15.             var newlat = point.lat();
16.             var newlng = point.lng();
17.             window.GeolocationUpdater.update(uuid, newlat, newlng);
18.             updateInfo(uuid, newlat, newlng);
19.         });
20.         if (window.HtmlCache[uuid]) {
21.             marker.bindInfoWindowHtml(window.HtmlCache[uuid],
22.                 {maxWidth:300});
23.         }
24.         if (window.EditableCache[uuid] === false) {
25.             marker.disableDragging();
26.         }
27.         window.EditableCache[uuid] = null;
28.         window.Mapping[uuid] = marker;
29.     };
```

Listing 16: Geolocation – adding GMarker object to the map display (maps.html)

The Java environment (or the input component `GeolocationSwingGadget`) must be informed of all changes to the marker (`GMarker` object). For example, if a geocoding request is sent via the Google Maps API, the address information determined should be updated in the `SwingGadget` input component. The same applies to a change to the `GMarker` object within the web application (e.g. by moving the marker in the map display). This means that the information from the `GMarker` object must be reassigned to a `GeolocationEntry` of the Java environment. The `uuid` variable is used for this.

If an update is made using the `GeolocationUpdater` object, the `uuid` value (and the changed latitude and longitude values) is passed to the update method of the `MapsPlugin` implementation (see Chapter 4.3.5 page 61). The `uuid` value can be used for renewed assignment of the entry to an input component in the Java environment. The internal method `GeolocationEntry.get(final String uuid)` of the `MapsPlugin` implementation is used for this. The method returns the `GeolocationEntry` of the input component with the passed `uuid`. The changed latitude and longitude values can then be updated on the matching `GeolocationEntry` within the Java environment. The `ModificationListener` belonging to the input component is informed of the update.




```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     private final List<GeolocationEntry> _entries = new
       ArrayList<GeolocationEntry>();
3.
4.     /**
5.      * Updates the geolocation of the specified geolocation entry and
6.      * may notify the related modification listener.
7.      *
8.      * @param uuid UUID of geolocation instance
9.      * @param latitude decimal latitude value
10.     * @param longitude decimal longitude value
11.     */
12.     public void update(final String uuid, final double latitude, final
       double longitude) {
13.         final GeolocationEntry location = get(uuid);
14.         if (location != null) {
15.             location.setLatitude(latitude);
16.             location.setLongitude(longitude);
17.             location.notifyPointModification();
18.         }
19.     }
20.     ...
21.     //--- inner methods ---//
22.     private GeolocationEntry get(final String uuid) {
23.         synchronized (_entries) {
24.             for (final GeolocationEntry entry : new
               ArrayList<GeolocationEntry>(_entries)) {
25.                 if (entry.getUUID().equals(uuid)) {
26.                     return entry;
27.                 }
28.             }
29.             return null;
30.         }
31.     }
32.     ...
33.     ...
34. }
```

Listing 17: Geolocation – assignment of the GeolocationEntries back into the Java environment (MapsPlugin-Impl.)



4.3.7 Listener – responding to changes

The FirstSpirit AppCenter API has been enhanced to include two listener interfaces, which respond to events within the browser instance and within the application tab:

- `TabListener` interface (for a description of the interface, see Chapter 3.5, page 30, for an example see Chapter 4.3.7.1, page 74).
- `BrowserListener` interface (for a description of the interface see Chapter 3.8 page 34, for an example implementation, see Chapter 4.3.7.2 page 76).

Apart from these default interfaces of the FirstSpirit AppCenter API, integration of the Google Maps web application requires a further listener implementation which responds to events within the integrated application, for example, moving a marking (GMarker) by means of Drag&Drop by the user:

- `ModificationListener` interface (for a description of the interface see Chapter 4.3.7.3 page 78, for an example see Chapter 4.3.8 page 79).

In addition, within the example implementation, a Java AWT `EventListener` is used, which should at least be mentioned briefly here. A `HierarchyListeners` can be used to hang the `GeolocationSwingGadget` input component into the hierarchy of the FirstSpirit input components. This is necessary so that updating of the application area takes place on changing workspaces (or the geolocation input component) (see Chapter 4.3.9 page 87).

4.3.7.1 TabListener

An application tab can be controlled by events. To respond to internal or external event, an instance of the type `TabListener` must be registered on the application tab. A `TabListener` is added to the `ApplicationTab` by calling the `addTabListener(...)` method. In this example the `MapsPlugin` class implements the `TabListener` interface itself, therefore, all methods of the interface must also be implemented (Note: If this is not wanted, the corresponding adapter class should be used (see Chapter 3.5)).

When the application tab is closed by the user, the FirstSpirit Framework responds by calling the `void tabClosed()` method. This method is implemented within the example implementation. Background: The implementation provides for the assignment of a certain marker (a geographic coordinate) to each `CUSTOM_GEOLOCATION` input component within a Google Maps map display (see Chapter 4.3.6). The `GeolocationEntries` are managed in an `ArrayList`. If an editor closes the Applications tab of the Google Maps integration, the list of `GeolocationEntries` should



be discarded.

When the application tab is selected by the user, the FirstSpirit Framework responds by calling the `void tabClosed()` method. This method is implemented within the example implementation, in order to focus on the currently selected browser instance. To do this, the `SwingUtilities.invokeLater(new Runnable() { ...` call is used to pass the `BrowserApplication.focus()` method to the Swing-EventQueue (see Chapter 3.7). As a result the focusing is performed asynchronously, only after the end of all events.

Background: It is only possible to use the mouse wheel to zoom within the integrated web application if the focus is also on the application. This can either be implemented manually by the user (click the application) or by the developer, as in this example.

```
1. public class MapsPlugin implements TabListener, ... {
2.     ...
3.     /**
4.      * Get BrowserApplication instance and may initialize it.
5.      *
6.      * @return BrowserApplication instance
7.      */
8.
9.     private BrowserApplication getApplication() {
10.        if (_application == null) {
11.            ...
12.            _tab = service.openApplication(...);
13.            _tab.addTabListener(this);
14.            ...
15.        }
16.        return _application;
17.    }
18.
19.    //--- TabListener ---//
20.
21.    public void tabSelected() {
22.        if (_focus) {
23.            SwingUtilities.invokeLater(new Runnable() {
24.                public void run() {
25.                    _application.focus();
26.                }
27.            });
28.            _focus = false;
```



```
29.         }
30.     }
31.
32.
33.     public void tabDeselected() {
34.     }
35.
36.
37.     public void tabClosed() {
38.         synchronized (_entries) {
39.             _entries.clear();
40.         }
41.         _active = false;
42.         _application = null;
43.     }
44. }
```

Listing 18: Geolocation – registering and using the TabListener (MapsPlugin-Impl.)

4.3.7.2 BrowserListener

In addition to the `TabListener`, the example implementation uses an instance of the type `BrowserListener`. Access to the content of the browser instance can be controlled with the help of a `BrowserListener`. A `BrowserListener` is registered by calling the `addBrowserListener(...)` method on the `BrowserApplication`. In this example the `MapsPlugin` class implements the `BrowserListener` interface itself, therefore, all methods of the interface must also be implemented (Note: If this is not wanted, the corresponding adapter class should be used (see Chapter 3.8)).

The example implementation uses the `void onDocumentComplete(...)` method to inject the Java object `GeolocationUpdater` into the web browser instance (see Chapter 4.3.5 page 61). The method is called if the `BrowserListener` of the browser instance (or of the instance of the type `BrowserApplication`) reports that the HTML document (including all images) has been completely loaded. Background: This is necessary to ensure orderly access to the DOM tree (w3c-DOM) (DOM access concept - see Chapter 2.2 page 18).

In addition, the `void onLocationChange(@NotNull String url)` method is used. The method is called if the `BrowserListener` reports that the URL of the browser instance (or the instance of the type `BrowserApplication`) has changed. In this case the Boolean `_active` is set to value false. As a result, updating of the browser instance and focusing are prevented.



```
1. public class MapsPlugin implements ..., BrowserListener {
2.     ...
3.     private boolean _active;
4.     /**
5.      * Get BrowserApplication instance and may initialize it.
6.      *
7.      * @return BrowserApplication instance
8.      */
9.
10.    private BrowserApplication getApplication() {
11.        if (_application == null) {
12.            ...
13.            _tab = service.openApplication(...);
14.            ...
15.            _application = _tab.getApplication();
16.            _application.addBrowserListener(this);
17.            ...
18.        }
19.        return _application;
20.    }
21.
22.
23.    //--- BrowserListener ---//
24.
25.
26.    public void onLocationChange(@NotNull final String url) {
27.        _active = false;
28.    }
29.
30.
31.    public void onDocumentComplete(final String url) {
32.        final BrowserApplication application = getApplication();
33.
34.        // document is loaded; inject java/javascript bridge
35.        application.inject(new GeolocationUpdater(this),
36.            "GeolocationUpdater");
37.
38.        _active = true;
39.
40.        // initialize map viewport and marker
41.        updateBrowser();
```



```
41.
42.         if ( _focus ) {
43.             SwingUtilities.invokeLater(new Runnable() {
44.                 public void run() {
45.                     application.focus();
46.                 }
47.             });
48.             _focus = false;
49.         }
50.     }
```

Listing 19: Geolocation – registering and using a BrowserListener (MapsPlugin-Impl.)

4.3.7.3 ModificationListener

Apart from the two default listeners, another interface is also required for the example implementation – a `ModificationListener`, which is responsible for updating the information within the `SwingGadget` input component.



The `ModificationListener` interface is not a part of the FirstSpirit AppCenter API, but instead is part of the example implementation for Google Maps integration.

The input component `GeolocationSwingGadget` saves a geographic coordinate, consisting of two decimal values (geographic longitude and latitude) and the complete address information (street, town, country) corresponding to this coordinate (see description of the input component in Chapter 4.3.1). These values are determined by the web application Google Maps and must be communicated from the browser instance into the FirstSpirit Java environment.

To this end, the `GeolocationSwingGadget` class implements the `ModificationListener` interface. The interface provides the following methods:

```
1.     package de.espirit.firstspirit.opt.geolocation.google;
2.
3.     public interface ModificationListener {
4.
5.         void onModification(double latitude, double longitude);
6.
7.         void onModification(String address);
8.     }
```

Listing 20: Geolocation – `ModificationListener` interface (not part of the AppCenter-API)

The corresponding methods are implemented in the example implementation and are responsible for:

- updating the latitude, longitude values
- updating the address information
- updating the thumbnail display

which are displayed within the input component (see Chapter 4.3.8 page 79). The methods of the interfaces are then called, if the address information or the coordinate (latitude and longitude values) within the browser instance have changed.

4.3.8 Updating the geodata of the input component (JavaScript » Java)

The input component `GeolocationSwingGadget` is closely linked to the web application Google Maps. This means, if the geoinformation (or the position of the `GMarker` object) changes within the integrated Google Maps application, the values of the corresponding `SwingGadget` input component must also be updated (for a description of the use case, see Chapter 4.2.2, page 43).

The passing of the values from the web application into the FirstSpirit Java environment is triggered by the Java object `GeolocationUpdater`, which is injected into the browser instance at a suitable time (see Chapter 4.3.5 page 61). The `GeolocationUpdater` class has the `void update(String uuid, double latitude, double longitude)` method and the `void info(final String uuid, final String address)` method, which communicate the changed values to the Java object `GeolocationEntry` of the `SwingGadget` input component and inform the `ModificationListener` belonging to the input component of the update (for a description of the `ModificationListener` interface see Chapter 4.3.7.3).



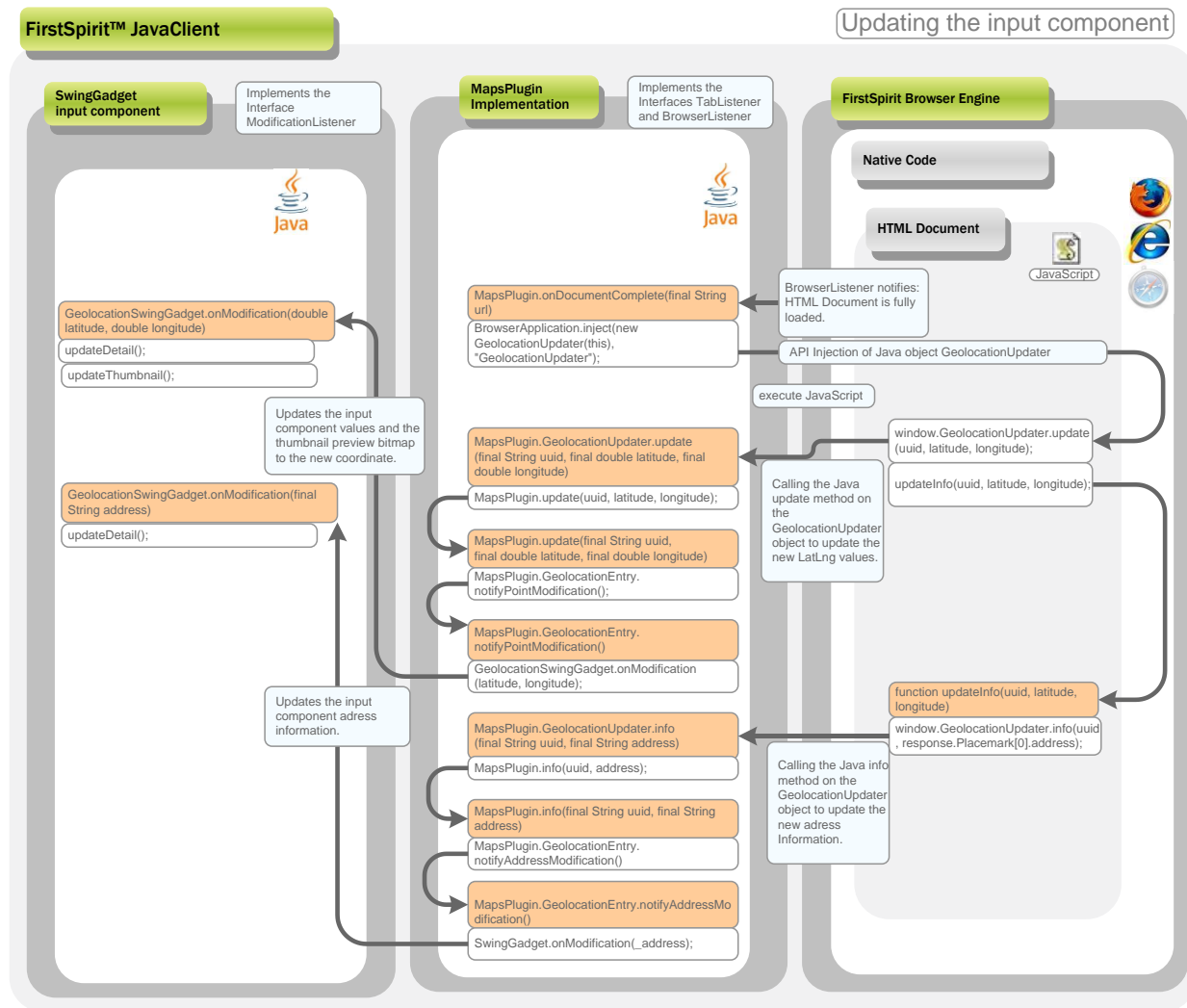


Figure 13: Updating the input component

The Java methods responsible for the update are called from the HTML document via JavaScript calls (see Chapter 4.3.5.). This always takes place if the GMarker object is changed within the Google Maps application, for example, by a drag-and-drop action of the editor within the Google Maps map display or by a search query via the address field of the SwingGadget input component.




```
<html>
<head>
  <title>Google Maps</title>
  ...
  window.GeolocationUpdater = null;
  /**
   * Notify MapsPlugin about latitude/longitude modification.
   *
   * @param uuid UUID of geolocation instance
   * @param latitude
   * @param longitude
   */
  function updateInfo(uuid, latitude, longitude) {
    var request = new GetLocationsRequest(new GLatLng(latitude, longitude));
    request.execute(function(response) {
      window.GeolocationUpdater.info(uuid,
      response.Placemark[0].address);
    });
  }
  /**
   * Add an marker to the GMap2 instance.
   *
   * @param uuid UUID of geolocation instance
   * @param latitude
   * @param longitude
   */
  window.addOverlay = function(uuid, latitude, longitude) {
    var marker = new GMarker(new GLatLng(latitude, longitude), {draggable:
true});

    window.GoogleMap.addOverlay(marker);
    GEvent.addListener(marker, 'dragend', function() {
      var point = marker.getLatLng();
      var newlat = point.lat();
      var newlng = point.lng();
      window.GeolocationUpdater.update(uuid, newlat, newlng);
      updateInfo(uuid, newlat, newlng);
    });
    ...
  };
};
```

Listing 21: Geolocation – Initiate updating via the GeolocationUpdater object (maps.html)



If the GMarker object within the web application is changed, first of all `window.GeolocationUpdater.update(uuid, newlat, newlng)` is called (see Figure 13). This call is evaluated on the Java side where it triggers the running of the Java method `GeolocationUpdater.update(final String uuid, final double latitude, final double longitude)` (see Chapter 4.3.5 page 61). The changed latitude and longitude values and a `uuid` value are passed to the method. The `uuid` value is used to get the `GeolocationEntry` of the `SwingGadget`. This is done using the internal method `GeolocationEntry.get(final String uuid)` of the `MapsPlugin` implementation (see Chapter 4.3.6). The changed latitude and longitude values are then set on the `GeolocationEntry` and the corresponding `ModificationListener` is informed about the update. To do this, the `GeolocationEntry.notifyPointModification()` method is called. This method in turn calls the `MethodModificationListener.onModification(double latitude, double longitude)` method of the inner `GeolocationEntry` class.

```
35. public class MapsPlugin implements TabListener, BrowserListener {
36.     private final List<GeolocationEntry> _entries = new
        ArrayList<GeolocationEntry>();
37.
38.     /**
39.      * Updates the geolocation of the specified geolocation entry and
40.      * may notify the related modification listener.
41.      *
42.      * @param uuid UUID of geolocation instance
43.      * @param latitude decimal latitude value
44.      * @param longitude decimal longitude value
45.      */
46.     public void update(final String uuid, final double latitude, final
        double longitude) {
47.         final GeolocationEntry location = get(uuid);
48.         if (location != null) {
49.             location.setLatitude(latitude);
50.             location.setLongitude(longitude);
51.             location.notifyPointModification();
52.         }
53.     }
54.     ...
55.     //--- inner methods ---//
56.     private GeolocationEntry get(final String uuid) {
57.         synchronized (_entries) {
```



```
58.         for (final GeolocationEntry entry : new
59.             ArrayList<GeolocationEntry>(_entries)) {
60.             if (entry.getUUID().equals(uuid)) {
61.                 return entry;
62.             }
63.         }
64.         return null;
65.     }
66.
67.     ...
68.     private static class GeolocationEntry {
69.         private ModificationListener _listener;
70.
71.         GeolocationEntry(..., final ModificationListener
72.             listener) {
73.             ...
74.             _listener = listener;
75.         }
76.         ...
77.         public void notifyPointModification() {
78.             if (_listener != null) {
79.                 _listener.onModification(_latitude, _longitude);
80.             }
81.         }
82.     }
83.
84.     public static class GeolocationUpdater {
85.
86.         private final MapsPlugin _mapsPlugin;
87.         ...
88.         public void update(final String uuid, final double
89.             latitude, final double longitude) {
90.             _mapsPlugin.update(uuid, latitude, longitude);
91.         }
92.     }
```

Listing 22: Geolocation – updating the latitude, longitude value (MapsPlugin-Impl.)

The `GeolocationSwingGadget` class implements the `ModificationListener` interface



(for a description of the interface see Chapter 4.3.7.3). The call `onModification(double latitude, double longitude)` of the `MapsPlugin` implementation is therefore passed directly to the `GeolocationSwingGadget.onModification(final double latitude, final double longitude)` method. This accepts the changed latitude and longitude values and ensures, by calling the `updateDetail()` method, for updating the values in the input component. The preview bitmap shown in the input component is also updated by calling the `updateThumbnail()` method, is and then displays the changed position.

```
1.  public class GeolocationSwingGadget ... implements
    ModificationListener, ...{
2.      ...
3.      private double _valueLatitude;
4.      private double _valueLongitude;
5.      private String _valueAddress;
6.      ...
7.
8.      public void onModification(final double latitude, final double
        longitude) {
9.          _valueLatitude = latitude;
10.         _valueLongitude = longitude;
11.         updateDetail();
12.         updateThumbnail();
13.     }
14.
15.
16.     public void onModification(final String address) {
17.         _valueAddress = address;
18.         updateDetail();
19.     }
20. }
```

Listing 23: Geolocation – Using the ModificationListener interface (SwingGadget-Impl.)

The address information, which matches the changed coordinates, must also be updated. To do this, within the JavaScript environment, first of all a Geolocation request with the changed latitude, longitude values is sent to Google (`new GetLocationsRequest(new GLatLng(latitude, longitude));` see page 111). This call requests detailed address information on the basis of the latitude and longitude values passed. `window.GeolocationUpdater.info(uuid, response.Placemark[0].address)` is then called. This call is evaluated on the Java side where it triggers the running of the Java method `GeolocationUpdater.info(final String uuid, final String address)`



(see Chapter 4.3.5 page 61). The result of the geocoding query and a `uuid` value are passed to the method. The `uuid` value gets the matching `GeolocationEntry` again. This is done using the internal method `GeolocationEntry get(final String uuid)` of the `MapsPlugin` implementation (see Chapter 4.3.6).

The address information is then set on the `GeolocationEntry` and the responsible `ModificationListener` is informed about the update. To do this, the `GeolocationEntry.notifyAddressModification()` method is called. This method in turn calls the `ModificationListener.onModification(final String address)` method of the inner `GeolocationEntry` class (see Figure 13).

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     private final List<GeolocationEntry> _entries = new
       ArrayList<GeolocationEntry>();
3.
4.     /**
5.      * Updates the address information of the specified geolocation entry
6.      * and may notify the related modification listener.
7.      *
8.      * @param uuid UUID of geolocation instance
9.      * @param address address string of the related geolocation
10.     */
11.    public void info(final String uuid, final String address) {
12.        final GeolocationEntry location = get(uuid);
13.        if (location != null) {
14.            location.setAddress(address);
15.            location.notifyAddressModification();
16.        }
17.    }
18.    ...
19.    //--- inner methods ---//
20.    private GeolocationEntry get(final String uuid) {
21.        synchronized (_entries) {
22.            for (final GeolocationEntry entry : new
23.                ArrayList<GeolocationEntry>(_entries)) {
24.                if (entry.getUUID().equals(uuid)) {
25.                    return entry;
26.                }
27.            }
28.            return null;
29.        }
30.    }
31. }
```



```
28.         }
29.     }
30.
31.     ...
32.     private static class GeolocationEntry {
33.         private ModificationListener _listener;
34.
35.         GeolocationEntry(..., final ModificationListener
36.             listener) {
37.             ...
38.             _listener = listener;
39.         }
40.         ...
41.         public void notifyAddressModification() {
42.             if (_listener != null) {
43.                 _listener.onModification(_address);
44.             }
45.         }
46.     }
47.
48.     public static class GeolocationUpdater {
49.
50.         private final MapsPlugin _mapsPlugin;
51.         ...
52.
53.         public void info(final String uuid, final String
54.             address) {
55.             _mapsPlugin.info(uuid, address);
56.         }
57.     }
```

Listing 24: Geolocation – Updating the address information (MapsPlugin-Impl.)



4.3.9 Responding to tree navigation events (Java » JavaScript)

The integrated web application should respond to events within the JavaClient. This not only concerns the events triggered by the Geolocation input component (for example, the address search via the "Search coordinate" button), but also tree navigation events or the change to the active workspace by the editor. When a new geolocation input component is selected (for example, via the tree navigation), the map display in the application area must be updated (see description of the use case in Chapter 4.2.4, fading in/out the map display from one coordinate to the next on changing between different input components (page 45 f.)). Background: Each input component is assigned a GeolocationEntry. On changing between two input components the current map display must be discarded and replaced by a new display with the new coordinate.

With the help of the Java AWT EventListeners `HierarchyListener`, a mechanism is implemented, which decides when the map display in the application area has to be updated. To do this, the input component `GeolocationSwingGadget` is attached to the hierarchy of the FirstSpirit input component. If the component hierarchy changes, for example, on changing the tab in the middle workspace, the `HierarchyListener` is informed and calls the `public void hierarchyChanged(HierarchyEvent e)` method.

The `GeolocationSwingGadget` class implements this method and calls the `void onShowing(boolean showing)` method in it. This method controls the updating of the Google Maps application in the application area, depending on the viewability of the input component in the workspace of the JavaClient. The updating of the browser instance is controlled by the Boolean `showing`. Updating should only take place if the geolocation input component is also visible in the editor's workspace. On calling the `void onShowing(boolean showing)` method, `e.getChanged().isShowing()` is therefore passed. The `HierarchyEvent` returns the component, which is at the highest point of the component hierarchy at the time of the event, and checks whether this component is visible or not.

If the component is visible (`showing`) the initial map type for the display in the application area is selected first. Depending on whether or not the input component has been locked to prevent editing, either the map type Hybrid map is set here or (if the content cannot be changed), the 3D map display (see Chapter 4.3.18 page 106). The geolocation is then determined by calling the `getValue()` method. The method returns an instance of the class `GeolocationImpl`. The returned value is passed to the `addToMap(...)` method, which registers the passed value for a geolocation input component (see Chapter 4.3.6). In the next step the `getMapsPlugin().updateBrowser()` method is called, which initiates the updating of the



browser instance (see Chapter page).

```
1.  public class GeolocationSwingGadget implements HierarchyListener
    ...
2.  {
3.  ...
4.  public JComponent getComponent() {
5.  ...
6.  _panel = new JPanel(layout);
7.  _panel.addHierarchyListener(this);
8.  ...
9.  }
10.
11. ...
12. public void hierarchyChanged(final HierarchyEvent e) {
13.     if ((e.getChangeFlags() & HierarchyEvent.SHOWING_CHANGED) != 0)
14.         {
15.             onShowing(e.getChanged().isShowing());
16.         }
17.
18.
19. private void onShowing(final boolean showing) {
20.     if (getMapsPlugin().isAccessible() && _valueSet) {
21.         GuiUtil.execute(new Runnable() {
22.             public void run() {
23.                 if (showing) {
24.                     final boolean isEditable = _editable;
25.                     if (isEditable) {
26.                         getMapsPlugin().setMapType(MapsPlugin.MapType.G_HYBRID_MAP);
27.                     } else {
28.                         getMapsPlugin().setMapType(MapsPlugin.MapType.G_SATELLITE_3D_MAP);
29.                     }
30.                     addToMap(getValue());
31.                     getMapsPlugin().updateBrowser();
32.                 } else {
33.                     getMapsPlugin().remove(getGadgetId());
34.                 }
35.             }
36.         });
37.     }
```




```
38.     }
39. }
```

Listing 25: Geolocation – Using the EventListener `HierarchyListener` (SwingGadget-Impl.)

4.3.10 Updating the browser instance (Java » JavaScript)

The browser instance of the integrated web application must be updated depending on certain events.

The `updateBrowser()` method updates the browser instance within the integrated preview of the FirstSpirit JavaClient. The method is only run if a Google Maps application already exists and the Applications tab is displayed in the foreground of the application area.

Search and determining the address details: The search and determination of the address details takes place by means of the Google Maps API. The geolocation and address detail updates are passed on to the JavaClient by means of the injected object and result in updating of the corresponding input component. The route directions - or the call of the relevant Google Maps / Bing page - is implemented by a form within the section template. Google Maps and Bing understandably have different URL parameters, which must be taken into account individually.

```
1.  public class MapsPlugin implements TabListener, BrowserListener {
2.  ...
3.  private BrowserApplication _application;
4.  private boolean _active;
5.  ...
6.
7.  /**
8.   * Updates browser and google map instance.
9.   * This method must be called to update geolocation entries and
10.  * viewport.
11.  */
12.  public void updateBrowser() {
13.      if (_application != null && _active) {
14.          // build script code to update google map
15.          final StringBuilder buf = new StringBuilder();
16.          buf.append(getScriptClearOverlays());
17.          buf.append(getScriptSetMapType(_mapType));
18.          for (final GeolocationEntry location : new
19.              ArrayList<GeolocationEntry>(_entries)) {
20.              if (location.isSearchMode()) {
21.                  buf.append(getScriptFindAddress(location));
22.              }
23.          }
24.      }
25.  }
```



```
21.         } else {
22.             buf.append(getScriptAddOverlay(location));
23.         }
24.         buf.append(getScriptSetEditable(location));
25.         buf.append(getScriptSetInfoHtml(location));
26.     }
27.     buf.append(getScriptSetViewport());
28.     final String script = buf.toString();
29.
30.     // prevent unnecessary rapidly script executions (may
31.     //interfere camera movement), caused by multiple showing
32.     //events
33.     if (_updateScript == null ||
34.         !_updateScript.equals(script) || (_updateTime +
35.             EQUAL_UPDATE_TIMEOUT < System.currentTimeMillis())) {
36.         _application.executeScript(script);
37.         _updateScript = script;
38.         _updateTime = System.currentTimeMillis();
39.     }
40.     /**
41.      * Builds script code to clear all overlays from map.
42.      *
43.      * @return script code
44.      */
45.     private String getScriptClearOverlays() {
46.         return "window.clearOverlays();";
47.     }
48.
49.     /**
50.      * Builds script code for map type switch.
51.      *
52.      * @param type map type to show
53.      * @return script code
54.      */
55.     private String getScriptSetMapType(final MapType type) {
56.         final StringBuilder buf = new StringBuilder();
57.         buf.append("window.setMapType(").append(type).append(");");
58.         return buf.toString();
59.     }
60. }
```



```
48.     }
49.     /**
50.         * Build script code for an address search entry.
51.         *
52.         * @param location related geolocation entry
53.         * @return script code
54.         */
55.     private String getScriptFindAddress(final GeolocationEntry location) {
56.         final StringBuilder buf = new StringBuilder();
57.         final String pattern = location.getAddress().replaceAll("'",
58.         "\\\\'");
59.         buf.append("window.findAddress('")
60.         .append(location.getUUID()).append(", ")
61.         .append(pattern).append("');");
62.         return buf.toString();
63.     }
64.     /**
65.         * Builds script code to modify editable-state of related geolocation
66.         entry.
67.         *
68.         * @param location related geolocation entry.
69.         * @return script code
70.         */
71.     private String getScriptSetEditable(final GeolocationEntry location) {
72.         final StringBuilder buf = new StringBuilder();
73.         buf.append("window.setEditable('")
74.         .append(location.getUUID()).append(", ")
75.         .append(location.isEditable()).append("');");
76.         return buf.toString();
77.     }
78.     /**
79.         * Build script code to update the info html balloon.
80.         *
81.         * @param location entry to build info html for.
82.         * @return script code
83.         */
84.     private String getScriptSetInfoHtml(final GeolocationEntry location) {
85.         final StringBuilder buf = new StringBuilder();
86.         buf.append("window.setInfoHtml('")
87.         .append(location.getUUID()).append(", ")
```



```
83.         String text = location.getInfoText();
84.         if (location.getInfoPicture() != null || text != null) {
85.             buf.append("<span style='font-family: Arial, Sans-Serif;
font-size: 11px;'>");
86.             if (location.getInfoPicture() != null) {
87.                 buf.append("<img align='right'
src='\"'.append(location.getInfoPicture()).append(\"' />");
88.             }
89.             if (text != null) {
90.                 text = text.replaceAll("\n", "<br/>");
91.                 text = text.replaceAll("\"", "&quot;");
92.                 buf.append(text);
93.             }
94.             buf.append("</span>\");
95.         } else {
96.             buf.append("null");
97.         }
98.         buf.append(");");
99.         return buf.toString();
100.     }
101.     /**
102.         * Builds script code to modify viewport and show registered geolocation
entries on map.
103.         *
104.         * @return script code.
105.         */
106.     private String getScriptSetViewport() {
107.         synchronized (_entries) {
108.             final StringBuilder buf = new StringBuilder();
109.             final MapViewport viewBounds = new MapViewport();
110.             for (final GeolocationEntry entry : _entries) {
111.                 viewBounds.include(entry);
112.             }
113.             if (viewBounds.getPointCount() == 1) {
114.                 final double lat = viewBounds.getMinLatitude();
115.                 final double lng = viewBounds.getMinLongitude();
116.                 buf.append("window.setViewPoint("
117.                     .append(lat).append(', ')
118.                     .append(lng).append(");");
119.             } else if (viewBounds.getPointCount() > 1) {
120.                 final double minlat = viewBounds.getMinLatitude();
```



```
121.         final double minlng = viewBounds.getMinLongitude();
122.         final double maxlat = viewBounds.getMaxLatitude();
123.         final double maxlng = viewBounds.getMaxLongitude();
124.         buf.append("window.setViewBounds(")
125.             .append(minlat).append(',')
126.             .append(minlng).append(',')
127.             .append(maxlat).append(',')
128.             .append(maxlng).append(");");
129.     }
130.     return buf.toString();
131. }
132. }
```

Listing 26: Geolocation –UpdateBrowser (MapsPlugin-Impl.)

4.3.11 MapsPlugin – Address search (Google-Geolocation)

Use case (see Chapter 4.2.1 page 42).

The entry point is, for example, a click on the Search button of the geolocation input component (see Chapter 4.3.11 page 93). This action triggers the call of the internal search method. The search method of the SwingGadget implementation passes the , `GadgetIdentifier` of the input component, the address string from the text field and the `ModificationListener` (see Chapter 4.3.7.3 page 78) to the search method of the MapsPlugin implementation.

```
1.     public class GeolocationSwingGadget {
2.         ...
3.         public JComponent getComponent() {
4.             ...
5.             final AbstractAction searchAction = new AbstractAction() {
6.                 public void actionPerformed(final ActionEvent e) {
7.                     search();
8.                     getMapsPlugin().updateBrowser();
9.                     getMapsPlugin().focusBrowser();
10.                }
11.            };
12.            ...
13.        }
14.
15.        private void search() {
16.            final String address = _searchField.getText();
```



```
17.         ...
18.         getMapsPlugin().search(getGadgetId(), address, this);
19.         ...
20.         updateMapsPlugin(getGadgetId());
21.     }
22. }
```

There the search mode is switched on first, but a search via the Google Maps API is not yet started:

```
1.     public class MapsPlugin implements TabListener, BrowserListener {
2.         public void search(final GadgetIdentifier gadgetId,...) {
3.             entry1.setSearchMode(true);
4.             entry1.setAddress(search);
5.         }
6.     }
```

The search for a new geographic position takes place via the Search button of the SwingGadget input component `CUSTOM_GEOLOCATION`. The editor can enter an address or part of an address (for example a street name) in the text field provided and start the search within the integrated web application by clicking the button (see Chapter 4.2.1 page 42).

```
1.     public class GeolocationSwingGadget...{
2.         private JTextField _searchField;
3.         ...
4.         private void search() {
5.             final String address = _searchField.getText();
6.             if (!StringUtil.isEmpty(address)) {
7.                 getMapsPlugin().search(getGadgetId(), address, this);
8.             }
9.             updateMapsPlugin(getGadgetId());
10.        }
11.    }
```

Listing 27: Geolocation – Search by means of an address string (SwingGadget-Impl.)

The `MapsPlugin` `getMapsPlugin()` method first generates the Singleton instance of the `MapsPlugin` class (see Chapter 4.3.2 page 52). The `MapsPlugin` class has its own search method. The search method of the `SwingGadget` implementation passes the `GadgetIdentifier` of the input component (`GadgetIdentifier` see Chapter page), the address string from the text field and the `ModificationListener` to the search method of the `MapsPlugin` implementation:



```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     private final List<GeolocationEntry> _entries = new
       ArrayList<GeolocationEntry>();
3.
4.     /**
5.      * Register a geolocation instance with the specified address string
6.      * and modification listener
7.      *
8.      * @param gadgetId gadget id of geolocation instance
9.      * @param search address string of geolocation
10.     * @param listener modification listener, may be null
11.     */
12.     public void search(final GadgetIdentifier gadgetId, final String
       search, final ModificationListener listener) {
13.         ensureApplicationTab();
14.         final GeolocationEntry entry = get(gadgetId);
15.         if (entry != null) {
16.             _entries.remove(entry);
17.         }
18.         GeolocationEntry entry1 = get(gadgetId);
19.         if (entry1 == null) {
20.             entry1 = new GeolocationEntry(gadgetId, 0, 0, listener);
21.             synchronized (_entries) {
22.                 _entries.add(entry1);
23.             }
24.         }
25.         entry1.setModificationListener(listener);
26.         entry1.setSearchMode(true);
27.         entry1.setAddress(search);
28.     }
29. }
```

Listing 28: Geolocation – Search by means of an address string (MapsPlugin-Impl.)

There the `ensureApplicationTab()` method first checks whether a browser instance of the web application already exists in the application area of the JavaClient. If a browser instance does not yet exist, a new instance is generated (see Chapter 4.3.3 page 53).

To display a new entry within the Google Maps integration, it is necessary to first determine whether old entries already exist. The example implementation uses the inner method `GeolocationEntry get(final GadgetIdentifier gadgetId)` to do this. The method returns a `GeolocationEntry` (provided an entry was saved for the component when the Gadget ID



was passed) or zero (if no entry exists).

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     private final List<GeolocationEntry> _entries = new
       ArrayList<GeolocationEntry>();
3.
4.     private GeolocationEntry get(final GadgetIdentifier gadgetId) {
5.         synchronized (_entries) {
6.             for (final GeolocationEntry entry : new
7.                 ArrayList<GeolocationEntry>(_entries)) {
8.                 if (entry.getGadgetId().equals(gadgetId)) {
9.                     return entry;
10.                }
11.            }
12.            return null;
13.        }
14.    }
15. }
```

Listing 29: Geolocation – Inner method: Get GeolocationEntry (MapsPlugin-Impl.)

If a GeolocationEntry already exists, the returned entry is then removed from the list of GeolocationEntries with the help of the remove method (see example on page 95).

The search method then calls the inner method `GeolocationEntry get(final GadgetIdentifier gadgetId)` once again. This time there is no entry; the method returns zero. The search method then creates a new instance of the type GeolocationEntry with the coordinates 0.0/0.0 and adds this to the list of GeolocationEntries (see example on page 95).

In the next step the `setSearchMode(final boolean searchMode)` method is called, which activates the search mode for the new entry. The `void setAddress(final String address)` method is then used to save the search string from the input field.



The `SwingGadget` implementation now calls the `void updateMapsPlugin(final GadgetIdentifier gadgetId)` method (see example on page 94).

```
1. public class GeolocationSwingGadget...{
2.     private boolean _editable;
3.     ...
4.     private void updateMapsPlugin(final GadgetIdentifier gadgetId) {
5.         getMapsPlugin().setEditable(gadgetId, _editable);
6.         getMapsPlugin().setInfoPicture(gadgetId, getInfoPicture(),
7.             "image/jpeg");
8.         getMapsPlugin().setInfoText(gadgetId, getInfoText());
9.     }
10. }
```

Listing 30: Geolocation – Updating MapsPlugin (SwingGadget-Impl.)

Other methods of the `MapsPlugin` implementation are called here on the instance of the type `MapsPlugin`. First, the edit mode of the component is switched on, then an info image and an info text are added:

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.
3.     /**
4.      * Modify editable-state of an google map marker entry specified by the
5.      * gadget id.
6.      * @param gadgetId gadget id of the related geolocation instance
7.      * @param editable new editable-state
8.      */
9.     public void setEditable(final GadgetIdentifier gadgetId, final
10.        boolean editable) {
11.         final GeolocationEntry entry = get(gadgetId);
12.         if (entry != null) {
13.             entry.setEditable(editable);
14.         }
15.
16.         private static class GeolocationEntry {
17.             ...
18.             private boolean _editable;
19.
20.             public void setEditable(final boolean editable) {
```



```
21.         _editable = editable;
22.     }
23. }
24. }
```

Listing 31: Geolocation –... (MapsPlugin-Impl.)

```
1.  public class GeolocationSwingGadget...{
2.  ...
3.  public JComponent getComponent() {
4.  @SuppressWarnings({"serial"})
5.  final AbstractAction searchAction = new AbstractAction() {
6.  public void actionPerformed(final ActionEvent e) {
7.  search();
8.  getMapsPlugin().
9.  setMapType (MapsPlugin.MapType.G_HYBRID_MAP);
10. getMapsPlugin().updateBrowser();
11. getMapsPlugin().focusBrowser();
12. }
13. };
14. }
```

Listing 32: Geolocation – UpdateBrowser (SwingGadget-Impl.)

This entry is first re-created with the coordinates 0.0/0.0 for the address search (see Chapter 4.3.11 page 93). Then the address string from the input component is set for the GeolocationEntry (see Listing 7).



4.3.12 maps.html – Introduction

In principle, two different ways are available for integrating a web application in FirstSpirit. On the one hand, a global web application can be implemented and installed on the FirstSpirit server. In this case, the URL of the web application can be called in the integrated browser. If an independent web application is to be circumvented, however, the required HTML code can also be easily initiated and called. This second way is also used for the Google Maps integration introduced here.

The maps.html file is required to initiate the HTML code. The file initially consists of a simple HTML structure. This structure must now be extended to include the suitable JavaScript or Google Maps API expressions:

- Load the Google Maps API
- Initialize the container for the map display
- Create new map object
- Centre map on a coordinate
- Define map type
- Load map via events
- Convert address data – geocoding
- GeolocationUpdater (Injection Java / JavaScript)

To do this, the browser must first be informed that it must load an external JavaScript file

The file maps.html has a windows object.

The map is displayed in a Google Maps container:

To prevent an independent web application, an HTML code is directly initiated here, which initializes a Google Maps container on loading.

The connection between web and Java is made by means of the application integration API developed with [TS#87759](#) [RN FS42_4-N3: Vertical prototypes for the application integration | API enhancements]. To prevent an independent web application, an HTML code is directly initiated here, which initializes a Google Maps container on loading. Within this HTML page, JavaScript methods are defined, which provide core functions such as "Add entry", "Remove entry" and "Modify viewport". These JavaScript methods are called on the Java side accordingly in order to display geolocations or to make them modifiable. IDs are generated for identification of the individual map entries, these IDs are then used for direct assignment and control. A way back (return path), provided by means of an object injection, is required here for the modification of the geolocation and the corresponding change notification. This object has a method for



updating the exact geolocation and a method by means of which the address string can be updated.

4.3.13 Excursus: HTML and JavaScript

Knowledge of HTML and JavaScript is advantageous in order to understand the following chapter, but is not absolutely necessary. This chapter provides a brief introductory explanation of the most important HTML tags and JavaScript object types, which are used within the example implementation.

Standard HTML basic structure:

```
<html>
  <head>
  </head>
  <body>
  </body>
</html>
```

JavaScript: The script language JavaScript can be used to subsequently change HTML pages, which are displayed within a browser (i.e. have already been loaded by visitors to the page). JavaScript is an event-controlled language. Certain actions of the user (e.g. a mouse click), can result in a change to the page. JavaScript is closely linked to the browser, which displays the HTML page and provides objects and methods for controlling the browser window (see object window) and for manipulating the DOM tree of an HTML page (see DOM).

Within the HTML file, the browser must be notified that the following code is a JavaScript. To this end the MIME type ("**text/javascript**") is given within the script tag. The opening and closing script tag define where the JavaScript begins and where it ends and therefore form the limits of the script.

```
<html>
  <head>
    <script type="text/javascript">
      ...
    </script>
  </head>
  <body>
  </body>
</html>
```



window object: Each browser window or each frame has a window object. The window object is automatically provided by the browser instance, i.e. does not have to be generated beforehand using the new operator. The object provides methods for modifying the display (e.g. show status bar) and for controlling with the help of events.

Document Object Model (DOM): The DOM is a standard for the access to the content of an HTML document. The individual HTML elements of an HTML document are set in a relation to each other via a tree structure. On the basis of this structure, starting from an element, the script can navigate to another element of the page and if necessary change it (for example, insert a tag). The corresponding methods of the DOM can be used to insert tags in specific places in the HTML page, manipulate attributes or restructure elements. In the following example, for example, the Google Maps map is inserted within a DIV element in the HTML page.

document object: The document object lies below the window object in the object hierarchy and is responsible for the content, which is displayed within a browser instance. The document object is the root node of the tree structure within the DOM. The lower level HTML elements are accessed by calling the methods made available by the DOM, e.g. getElementById(...). In this case an element of the HTML page, which has an id attribute, is loaded. In the following example, the DIV element, which displays the Google Maps map, is referenced. Analogous to the window object, the document object is also automatically provided by the browser instance, i.e. does not have to be generated beforehand using the new operator.

4.3.14 maps.html - Loading the Google Maps API

Before the Google Maps API can be used, the integrated browser must first be informed that it has to load an external JavaScript file. This is achieved within the maps.html file with the following call:

```
1. <html>
2. <head>
3. <title>Google Maps</title>
4. <script type="text/javascript"
   src="http://maps.google.com/maps?file=api&v=2&key="></script>
5. <script type="text/javascript">
6. </head>
7. ...
8. </html>
```

Within the Script tags the browser is notified here that it should load an external JavaScript file from URL <http://maps.google.com/maps?file=api&v=2&key=myKey> (for further information see



chapter Excursus: HTML and JavaScript page 100 ff.). This JavaScript file contains all the necessary functions, classes and libraries, required for use of the Google Maps API. The key passed here (`key=...`), must correspond to the previously generated, individual Google Maps API key (see Chapter 4.1.3 page 39). If the example implementation is to be used, the individual key must be added within the script tag (e.g. `src="http://maps.google.com/maps?file=api&v=2&key=myKey"`).

4.3.15 maps.html - Initializing the container for the map display

A central element of the Google Maps integration is the map display. This display takes place externally via the Google Maps API. However, to display the map in the integrated browser, an HTML element must be provided, which acts as a kind of placeholder or container for the map. To this end, a DIV element with an attribute id ("container") is defined within the `<body>` tags in the maps.html file.

```
1. <body>
2. <div id='container'></div>
3. </body>
```

All HTML nodes of the page are lower level objects of the JavaScript objects document (for further information see Chapter Excursus: HTML and JavaScript page 100 f.). The id attribute (here: "container") can be used to reference the DIV element in the Document Object Model (DOM) of the browser (for further information see Chapter Excursus: HTML and JavaScript page 100 f.). To do this, the `getElementById(...)` method is called on the document object:

```
document.getElementById('container');
```

The Google Maps map display automatically adjusts to the size of the DIV container, in which it is displayed. The display of the DIV container (e.g. width, height) can be changed within the style tags via the corresponding style attributes.

```
1. <style>
2. ...
3. #container {
4.     position: absolute;
5.     visibility: hidden;
6.     width: 100%;
7.     height: 100%;
8. }
9. </style>
```

Initialization of the DIV container (or the map display) is controlled by events, which ensure that all the files required to run the JavaScript functions have already been loaded (see Chapter



4.3.19 page 108). If one of the events occurs, the JavaScript function `init()` is called, which first generates a new map object (see Chapter 4.3.16 page 103).

4.3.16 maps.html – Creating a new map object

With the help of the JavaScript operator `new`, the `init()` function generates a new instance of the type `GMap2` (for a description of the class and the constructor, see Google Maps API Documentation⁹).

```
1.   window.GoogleMap = null; // GMap2 instance
2.   ...
3.
4.   function init() {
5.     window.GoogleMap = new GMap2(document.getElementById('container'), {
6.       mapTypes: [G_NORMAL_MAP, G_SATELLITE_MAP, G_HYBRID_MAP, G_SATELLITE_3D_MAP]
7.     });
8.     ...
9.   }
```

`GMap2` is the central class of the Google Maps API, which represents the map. For the map to be displayed in the integrated browser, it must be connected with a DOM node of the HTML page. To this end, the previously defined DIV element is passed to the constructor. In addition, the required map types are also passed to the constructor. The example implementation uses the following map types:

- `G_NORMAL_MAP`: Display of the map with standard 2D tiles.
- `G_SATELLITE_MAP`: Display of the map with photo tiles (satellite images).
- `G_HYBRID_MAP`: Display of the map with photo tiles and tiles for the display of prominent functions, such as streets and place names.
- `G_SATELLITE_3D_MAP`: Display of the map as an interactive 3D model of the earth with satellite images.

Further properties, for example, for zooming within the map, are defined via the corresponding methods of the Google Maps API¹⁰. The control is added to the map with the `addControl(...)` method. Within the `init` function these are a control for swinging and zooming

⁹ <http://code.google.com/intl/de/apis/maps/documentation/javascript/v2/introduction.html>

¹⁰ see <http://code.google.com/intl/de-DE/apis/maps/documentation/javascript/v2/reference.html#GMap2>



(GLargeMapControl), a control for changing the scale (GScaleControl) and buttons for switching between the defined map types (e.g. map and satellite) (GMapTypeControl).

4.3.17 maps.html - Center map (center point or display area)

In the next step the map display must be initialized. Maps or objects of the type GMap2 must be centered before they can be displayed in an HTML page. This initialization is achieved by calling the setCenter() method on the instance of the type GMap2 (see Chapter 4.3.16 page 103).

To do this, first of all the start coordinate must be defined, which is to be displayed in the map after it has been loaded. The Google Maps API provides the GLatLng¹¹ class for this purpose. An object of the type GLatLng is a geographic point, which is defined by the corresponding coordinates, consisting of the geographic northing and easting, or latitude and longitude (abbreviated form: LatLng). A new object of the type GLatLng is generated by the new operator:

```
new GLatLng(latitude, longitude)
```

The coordinates of a new map display are initially set to 0.0/0.0 (see Chapter page).

The GLatLng coordinate must now be passed to the setCenter() method. Optionally, a zoom level and a map type can also be passed:

```
<html>
  <head>
    <script ...>
    ...
    window.GoogleMap = null; // GMap2 instance
    window.DefaultMapZoom = 18; // Zoom for ~200 meter

    function(latitude, longitude) {
      ...
      var zoomLevel = window.DefaultMapZoom;
      window.GoogleMap.setCenter(new GLatLng(latitude, longitude), zoomLevel);
      ...
    }

    </script>
```

¹¹ see <http://code.google.com/intl/de-DE/apis/maps/documentation/javascript/v2/reference.html#GLatLng>




```
</head>
<body>
  <div id='container'></div>
</body>
</html>
```

The `setCenter()` method now sets the centre of the map at the passed coordinate (using the optionally given zoom level and the map type). This method must always be run initially, before other operations may take place on the map. This also applies to the definition of other map attributes, for example, the map type (see Chapter 4.3.18 page 106).

Within the example implementation the map is used in two JavaScript functions. The `function(latitude, longitude)` determines a centre point of the map view on the basis of a single coordinate, the `function(minLatitude, minLongitude, maxLatitude, maxLongitude)` determines a visible, rectangular selection area of the map view on the basis of two coordinates, which define the limits of the selection area.

```
<html>
  <head>
    <script ...>
    ...
    window.GoogleMap = null; // GMap2 instance

    /**
     * Set viewport to specified latitude/longitude position.
     *
     * @param latitude
     * @param longitude
     */
    window.setViewPoint = function(latitude, longitude) {
      if (window.GoogleMap.getCurrentMapType() == G_SATELLITE_3D_MAP) {
        window.GoogleMap.getEarthInstance(function(ge) {
          if (ge) {
            ge.getOptions().setFlyToSpeed(window.DefaultEarthFlySpeed);
            var lookAt = createEarthLookAt(ge, latitude, longitude);
            ge.getView().setAbstractView(lookAt);
          }
        });
      } else {
        var zoomLevel = window.DefaultMapZoom;
        if (window.LazyMapType == G_SATELLITE_3D_MAP) {
```



```
        zoomLevel = window.DefaultEarthZoom;
    }

    window.GoogleMap.setCenter(new GLatLng(latitude, longitude), zoomLevel);
    if (window.LazyMapType) {
        _setMapType(window.LazyMapType);
        window.LazyMapType = null;
    }
}

// map is initially hidden; show map after viewport modification
document.getElementById('container').style.visibility = "visible";
};

</script>
</head>
<body>
    <div id='container'></div>
</body>
</html>
```

4.3.18 maps.html – define map type

To display the map, the map type must be known beforehand. Within the example implementation, the required map types are passed to the object of the type GMap2 in the constructor (see Chapter 4.3.16 page 103).

The definition of a map type is ensured in the example implementation, following initialization of the map, via the JavaScript function `_setMapType(type)`. Within the function, the Google Maps method `setMapType(type:GMapType)` is called on the instance of the type GMap2. The required map type (GMapType¹²) is passed to the method. The GMapType must be known to the map, i.e. either given directly in the constructor (see example) or previously added via the method `addMapType(type:GMapType)`.

The function is not called until after the map has been initialized (see Chapter 4.3.17) using the `setCenter()` method, within the JavaScript functions `function(latitude, longitude)` und `function(minLatitude, minLongitude, maxLatitude, maxLongitude)`, which center the map view on a specific coordinate or a specific selection area.

¹² <http://code.google.com/intl/de/apis/maps/documentation/javascript/v2/reference.html#GMapType>



```
1. <html>
2.   <head>
3.     <script ...>
4.       ...
5.       window.GoogleMap = null; // GMap2 instance
6.
7.       // earth plugin apply current location on first maptype switch, no
7.       // matter what.
8.       // as workaround we change maptype in that case after viewport
8.       // modification not before.
9.       // Maptype to set after viewport modification
9.       window.LazyMapType = null;
10.
11.
12.     function init() {
13.       window.GoogleMap = new GMap2(document.getElementById('container'),
14.         {
14.           mapTypes: [G_NORMAL_MAP, G_SATELLITE_MAP, G_HYBRID_MAP,
14.             G_SATELLITE_3D_MAP]
15.         });
16.       ...
17.     }
18.
19.     /**
20.      * Set map type to GMap2 instance.
21.      *
22.      * @param type new maptype
23.      */
24.     function _setMapType(type) {
25.       window.GoogleMap.setMapType(type);
26.       ...
```



```
27.     }
28.
29.     /**
30.      * Set viewport to specified latitude/longitude position.
31.      *
32.      * @param latitude
33.      * @param longitude
34.      */
35.     window.setViewPoint = function(latitude, longitude) {
    ...
36.         window.GoogleMap.setCenter(new GLatLng(latitude, longitude),
            zoomLevel);
37.         if (window.LazyMapType) {
38.             _setMapType(window.LazyMapType);
39.             window.LazyMapType = null;
40.         }
41.         // map is initially hidden; show map after viewport modification
42.         document.getElementById('container').style.visibility = "visible";
43.     };
44.
45.     window.setViewBounds = function(minLatitude, minLongitude,
        maxLatitude, maxLongitude) {
46.         ...
47.         if (window.LazyMapType) {
48.             _setMapType(window.LazyMapType);
49.             window.LazyMapType = null;
50.         }
51.         // map is initially hidden; show map after viewport modification
52.         document.getElementById('container').style.visibility = "visible";
53.     };
54.     ...
55.     </script>
56. </head>
57. <body>
58.     <div id='container'></div>
59. </body>
60. </html>
```



4.3.19 maps.html - load map object via events

The map (or the DIV container in which the map is displayed) is initialized event-controlled by onLoad or load events. This ensures that all the necessary files, which are required on running the JavaScript functions, are already loaded. To do this, the window object is extended to include the EventHandler "onLoad" or "load" (event depends on the integrated browser used) (for further information see Chapter Excursus: HTML and JavaScript page 100 f.):

```
1.   if (window.attachEvent) {
2.     window.attachEvent("onload", init);
3.   } else if (window.addEventListener) {
4.     window.addEventListener("load", init, false);
5.   }
```

If one of the events occurs, the JavaScript function `init()` is called, which creates a new map object (see Chapter 4.3.16 page 103). This map display is then initialized and assigned attributes. In order for the map to be displayed within the DIV container, the DIV element (and therefore the map too) must then be shown. This is achieved by calling `document.getElementById('container').style.visibility = "visible"`. This method is not called until after the map has been initialized (see Chapter 4.3.17) using the `setCenter()` method, within the JavaScript functions `function(latitude, longitude)` und `function(minLatitude, minLongitude, maxLatitude, maxLongitude)`, which center the map view on a specific coordinate or a specific selection area.

4.3.20 maps.html – converting address data (Geocoding)

Such geographic coordinates consist of two decimal values, which describe a geographic longitude and latitude. Geocoding is the term used if an address (as an object of the type string) is converted into a geographic coordinate. Google provides a service for geocoding, which can be accessed using the `GClientGeocoder()` object¹³.

In the example implementation a new object of the type `GClientGeocoder()` is generated within the `init` function:

```
1.   <html>
2.   <head>
```

¹³ http://code.google.com/intl/de-DE/apis/maps/documentation/javascript/v2/services.html#Geocoding_Object



```

3.   <script type="text/javascript"
      src="http://maps.google.com/maps?file=api&v=2&key="></script>
4.   <script type="text/javascript">
5.
6.   window.GoogleGeocoder = null; // Geocoder, address <> lat/lng
7.   ...
8.   function init() {
9.       ...
10.  window.GoogleGeocoder = new GClientGeocoder();
11.  }

```

This function first determines a geographic coordinate from the passed address parameter (an object of the type string). Google provides a service (GoogleGeocoder) for this geocoding (see Chapter 4.3.20 page 109). The location of the Internet access point, from which the query is made, is also included. Therefore, if only a street name is given for the address search, a destination near the (starting) location is looked for. Following successful Google geocoding a callback takes place in the JavaScript function. There a new marker is added to the map display (via `window.addOverlay(...)`) and a new center point is set (see Chapter 4.3.17 page 104).

4.3.21 maps.html – GeolocationUpdater (Injection Java / JavaScript)

A Java object (JavaScript proxy –GeolocationUpdater) must be provided in the JavaScript environment for the communication between the native level of the integrated browser (or the Google Maps API) and the Java level of the FirstSpirit JavaClient (see Chapter 4.3.5 page 61). This object is used to provide unidirectional communication from the JavaScript environment of the integrated browser into the Java environment of the FirstSpirit JavaClient. The required Java object must be injected into the HTML document from the outside (from the Java environment) (see Chapter 4.3.5 page 61). The FirstSpirit AppCenter API provides the necessary interfaces and methods (Interface: `BrowserApplication` see Chapter 3.7 page 31).

The example implementation first generates a Java object of the type `GeolocationUpdater` in the `Java MapsPlugin` class (see Chapter 4.3.5 page 61). The `GeolocationUpdater` class has the methods `void update(String uuid, double latitude, double longitude)`, which update a `GeolocationEntry` to the passed coordinate and informs the `ModificationListener` belonging to the input component of the update and the `void info(String uuid, String address)` method, which updates the address information of the `GeolocationEntry`.

```

@SuppressWarnings({"UnusedDeclaration"})
public static class GeolocationUpdater {

```



```
private final MapsPlugin _mapsPlugin;

public GeolocationUpdater(final MapsPlugin mapsPlugin) {
    _mapsPlugin = mapsPlugin;
}

public void update(final String uuid, final double latitude, final
double longitude) {
    _mapsPlugin.update(uuid, latitude, longitude);
}

public void info(final String uuid, final String address) {
    _mapsPlugin.info(uuid, address);
}
}
```

A new instance of the type `GeolocationUpdater` is then injected into the HTML document (`maps.html`) by calling the `void inject(Object object, String name)` method. A name (here: `"GeolocationUpdater"`) is passed, with which this object can be reached within the JavaScript environment.

```
application.inject(new GeolocationUpdater(this), "GeolocationUpdater");
```

Following the injection, not only the object instance itself is available in the JavaScript environment but also the corresponding methods of the `GeolocationUpdater` object. This means that following injection, all methods of the Java `GeolocationUpdater` class can also be called within the JavaScript environment. Therefore, in specific terms, the `update` and `info` methods, which are implemented within `MapsPlugin`, are also available on the proxy within the JavaScript environment:

```
1. <html>
2.   <head>
3.     <title>Google Maps</title>
4.     <script type="text/javascript"
5.       src="http://maps.google.com/maps?file=api&v=2&key="></script>
6.     <script type="text/javascript">
```



```
7.         window.GeolocationUpdater = null;
8.         /**
9.         * Notify MapsPlugin about latitude/longitude modification.
10.        *
11.        * @param uuid UUID of geolocation instance
12.        * @param latitude
13.        * @param longitude
14.        */
15.        function updateInfo(uuid, latitude, longitude) {
16.            var request = new GetLocationsRequest(new
17.                GLatLng(latitude, longitude));
18.            request.execute(function(response) {
19.                window.GeolocationUpdater.info(uuid,
20.                    response.Placemark[0].address);
21.            });
22.        }
23.        ...
24.        /**
25.        * Use Geocoder to find the exact geolocation of the
26.        * specified address pattern
27.        * @param uuid UUID of geolocation instance
28.        * @param pattern address string
29.        */
30.        window.findAddress = function(uuid, pattern) {
31.            window.GoogleGeocoder.getLatLng(pattern,
32.                function(point) {
33.                    if (point) {
34.                        var latitude = point.lat();
35.                        var longitude = point.lng();
36.                        window.addOverlay(uuid, latitude,
37.                            longitude);
38.                        window.setViewPoint(latitude, longitude);
39.                        window.GeolocationUpdater.update(uuid,
40.                            latitude, longitude);
41.                        updateInfo(uuid, latitude, longitude);
42.                    }
43.                });
44.        };
45.    </script>
```

Listing 33: Geolocation – GeolocationUpdater (maps.html)



To do this, the FirstSpirit Framework generates a corresponding JavaScript method for each method of the Java object instances. When called from the JavaScript environment, this method sends an event, which is in turn evaluated on the Java side. Therefore, this means calling:

```
window.GeolocationUpdater.update(uuid, latitude, longitude);
```

in a JavaScript function, leads directly to running of the update() method within the Java MapsPlugin implementation. The suitable Java method is determined from the name and the passed parameters and is called accordingly.

```
1.  /**
2.   * Updates the geolocation of the specified geolocation entry and may
3.   * notify the related modification listener.
4.   *
5.   * @param uuid UUID of geolocation instance
6.   * @param latitude decimal latitude value
7.   * @param longitude decimal longitude value
8.   */
9.   public void update(final String uuid, final double latitude, final double
   longitude) {
10.       final GeolocationEntry location = get(uuid);
11.       if (location != null) {
12.           location.setLatitude(latitude);
13.           location.setLongitude(longitude);
14.           location.notifyPointModification();
15.       }
16.   }
```

All parameters passed within the JavaScript environment, are elevated into the Java environment, analogous to the object conversion to date. As, unlike Java, JavaScript only supports a limited set of simple data types, the conversion is subject to certain restrictions:

However, there is still a small restriction here. As synchronicity cannot be guaranteed for event generation/evaluation, returned values of the Java methods are returned by means of callback. If, therefore, there is a String getName() method, in JavaScript this becomes a void getName(function:callback) method. Therefore, here the last parameter is always the corresponding callback.

The inject method can be used to inject any Java object into a JavaScript environment. However, certain restrictions apply to the methods of this object. For example, only a range of simple data types are supported. Complex object types not known in JavaScript are not supported.





5 Listings

LISTING 1: GEOLOCATION - GENERATING A NEW INSTANCE OF MAPSPUGIN (SWINGGADGET-IMPL.) 52

LISTING 2: GEOLOCATION – MAPSPUGIN CONSTRUCTOR (MAPSPUGIN-IMPL.) 52

LISTING 3: GEOLOCATION –BROWSERAPPLICATION INSTANCE AVAILABLE? (MAPSPUGIN-IMPL.) 53

LISTING 4: GEOLOCATION – OPENING THE APPLICATION WITHIN A TAB (MAPSPUGIN-IMPL.) 56

LISTING 5: GEOLOCATION – CREATE SCRIPT FOR THE GEO-CODING OF AN ADDRESS STRING (MAPSPUGIN-IMPL.) 59

LISTING 6: GEOLOCATION – JAVASCRIPT FUNCTION FINDADDRESS (MAPS.HTML) 60

LISTING 7: GEOLOCATION – RUN THE COLLATED JAVASCRIPT FRAGMENTS (MAPSPUGIN-IMPL.) 61

LISTING 8: GEOLOCATION – INJECTION OF JAVA OBJECT GEOLOCATIONUPDATER (MAPSPUGIN-IMPL.) 62

LISTING 9: GEOLOCATION – GEOLOCATIONUPDATER (MAPSPUGIN-IMPL.) 63

LISTING 10: GEOLOCATION – CALLING A JAVA METHOD FROM THE JAVASCRIPT CODE (MAPS.HTML-IMPL.) 64

LISTING 11: GEOLOCATION – ADDING A NEW GMARKER OBJECT (MAPS.HTML) 65

LISTING 12: GEOLOCATION – GEOLOCATIONENTRY CONSTRUCTOR (MAPSPUGIN-IMPL.) 66

LISTING 13: GEOLOCATION – PASSING THE GADGETIDENTIFIER TO MAPSPUGIN (SWINGGADGET-IMPL.) 67

LISTING 14: GEOLOCATION – CREATING A NEW GEOLOCATIONENTRY (MAPSPUGIN-IMPL.) 68

LISTING 15: GEOLOCATION – PASSING INFORMATION FROM THE GEOLOCATIONENTRY OBJECT (MAPSPUGIN-IMPL.) 70

LISTING 16: GEOLOCATION – ADDING GMARKER OBJECT TO THE MAP DISPLAY (MAPS.HTML) 71

LISTING 17: GEOLOCATION – ASSIGNMENT OF THE GEOLOCATIONENTRIES BACK INTO THE JAVA ENVIRONMENT (MAPSPUGIN-IMPL.) 72

LISTING 18: GEOLOCATION – REGISTERING AND USING THE TABLISTENER (MAPSPUGIN-IMPL.) 75

LISTING 19: GEOLOCATION – REGISTERING AND USING A BROWSERLISTENER (MAPSPUGIN-IMPL.) 77

LISTING 20: GEOLOCATION – MODIFICATIONLISTENER INTERFACE (NOT PART OF THE APPCENTER-API) 77

LISTING 21: GEOLOCATION – INITIATE UPDATING VIA THE GEOLOCATIONUPDATER OBJECT (MAPS.HTML) 80

LISTING 22: GEOLOCATION – UPDATING THE LATITUDE, LONGITUDE VALUE (MAPSPUGIN-IMPL.) 82

LISTING 23: GEOLOCATION – USING THE MODIFICATIONLISTENER INTERFACE (SWINGGADGET-IMPL.) 83

LISTING 24: GEOLOCATION – UPDATING THE ADDRESS INFORMATION (MAPSPUGIN-IMPL.) 85

LISTING 25: GEOLOCATION – USING THE EVENTLISTENER HIERARCHYLISTENER (SWINGGADGET-IMPL.) 88

LISTING 26: GEOLOCATION –UPDATEBROWSER (MAPSPUGIN-IMPL.) 92

LISTING 27: GEOLOCATION – SEARCH BY MEANS OF AN ADDRESS STRING (SWINGGADGET-IMPL.) 93

LISTING 28: GEOLOCATION – SEARCH BY MEANS OF AN ADDRESS STRING (MAPSPUGIN-IMPL.) 94

LISTING 29: GEOLOCATION – INNER METHOD: GET GEOLOCATIONENTRY (MAPSPUGIN-IMPL.) 95

LISTING 30: GEOLOCATION – UPDATING MAPSPUGIN (SWINGGADGET-IMPL.) 96

LISTING 31: GEOLOCATION –... (MAPSPUGIN-IMPL.) 97

LISTING 32: GEOLOCATION – UPDATEBROWSER (SWINGGADGET-IMPL.) 97

LISTING 33: GEOLOCATION – GEOLOCATIONUPDATER (MAPS.HTML) 111

