The **SAFE** Activation Framework for Extensibility

Official SAFE User Manual

THE
SAFE
ACTIVATION
FRAMEWORK FOR
EXTENSIBILITY

January 8, 2014

(SAFE v1.123, *Rev* : 3008)

— Compact Version —

Raphael M. Reischuk

Team and contributors

Core

- **Raphael M. Reischuk**
  Saarland University, Germany.
  `http://www.infsec.cs.uni-saarland.de/~reischuk/`
- **Michael Backes**
  Saarland University, Germany.
  Max-Planck Institute for Software-Systems, Germany.
  `http://www.infsec.cs.uni-saarland.de/~backes/`
- **Johannes Gehrke**
  Cornell University, NY, USA.
  `http://www.cs.cornell.edu/johannes/`

Extensions

- **Florian Schröder**
  Master student, Saarland University, 2012.
- **Parth Tripathi**
  Internship, Saarland University, 2012.
- **Shefali Garg**
  Internship, Saarland University, 2013.

Bug reports, helpful feedback, suggestions of new features

- Santiago Aragón, 2012.
- Istvan Burbank, 2012, 2013.

Contents

Chapter 0

Getting started

This manual (*Rev* : 3008) refers to the SFWC compiler in version 1.123. Please make sure you have the correct version installed. Note that we are changing the system very frequently these days. So make sure you always consider the latest version of the manual and also the latest version of the SAFE development framework.

Moreover, please note that this manual is still very much in a draft mode. In case you miss any information that you think should be contained in here, please send an email to the author.

Further note that this is the compact version of the manual. An extended version is available upon request. In order to estimate which additional information is contained in the extended version, all positions with more profound information in the extended version are explicitly marked.

How to navigate through this document?

If you want to see actual **code** and some **demo applications** (no explanations and theory thereof), please dive down to Chapter 6 "Sample Applications" on page 53.

In case you want to read a motivating **introduction to modern trends in web application development**, please start at the very beginning with Chapter 1 on page 7.

An introduction to the **SAFE** activation framework is provided in Chapter 2. Starting on page 13, SAFE and its main building blocks, the **f-units**, are introduced.

Chapter 3 presents the **SFW modeling language** which is necessary for the declarative development of f-units.

The concept of **customization** is introduced in Chapter 4.

A couple of **technical questions** (file handling, server setup, compilation, etc) are answered in Chapter 5.

Chapter 6 contains **demos and sample applications**.

Chapter 7 provides the **syntax** for various SAFE components and specifications.

Finally, Chapter 8 mentions common error messages and helps finding solutions.

Notation

Whenever special caution is required from the developer, the margin shows a **red exclamation point** as warning symbol.

Chapter 1

Introduction

1.1 Motivation

More and more software is delivered through the web, following today's cloud idea of delivering Software as a Service (SaaS). The code of such rich internet applications (RIAs) is split into client and server code, where the server code is run at the service provider and the client accesses the application through a web browser. In data-driven web applications, the state of the application resides in a database system (or in a key-value store), and users interact with this persistent state through web clients. In this document, we describe **a framework for personalizing** such **data-driven web applications**. By personalization we mean that a user has the capability of customizing the functionality of an RIA to fit her unique application needs.

Example 1. As a first example, consider Facebook user Mark, who no longer likes a single news feed for all of his contacts; Mark wants to split the news feed into two columns, one for his friends and one for his business contacts. Today, Mark would have to wait (and hope) for Facebook to create this functionality as part of an upgrade of its interface. We envision a world where Mark could take the initiative himself; he could directly "program" this extension and integrate it for himself into the running Facebook application. Mark could also provide this extension as an "App" to other users who desire the same functionality. Note that this is not a "Facebook Application" as enabled by the Facebook API, but it is a customization of the core user-facing Facebook functionality through a user-defined extension. *

Example 2. As a second example, consider a conference management system such as Microsoft's Conference Management Tool (CMT) [5]. From time to time, the team behind CMT introduces a new feature that has been long requested by the community (see, for example, the features currently marked "(new!)" on the CMT website). None of these extensions is difficult to build, but today any changes are only within the realm of the CMT developers. In addition, due to limited resources, the team only incorporates extensions requested by the majority of users and thus forgoes the opportunity to serve the long tail. For example, consider Surajit who wants to run his conference with shepherding of borderline papers. Currently, Surajit has to wait and hope that the CMT team considers his functionality important enough to release it as part of its next upgrade. However, **we believe that innovation and integration of such new functionality can be significantly increased** if Surajit could directly take initiative, program the extension himself, and then share it with others in the research community who desire similar functionality. Thus **we want custom extensions to be built by any member of the community** instead of being left only to the CMT team. *

In both of these examples, **personalization of an existing data-driven web application by a third party** who was not the developer of the original application is the key to success. Note that personalization not only benefits the user who programmed it; an extension could later on be shared with other users, making the application automatically an "extension app store" where users can (1) run the RIA directly as provided, (2) personalize it with any set of extensions developed and provided by the community, (3) personalize it themselves through easy and well-defined user interfaces, and then (4) share or sell their extensions to the community.

The tremendous benefits of personalization also come with huge challenges. First, the often organic growth of today's RIAs makes it hard to keep track of the diversity of locations to which code has to be integrated, thereby obeying various security and safety constraints regarding, for instance, namespaces and assertions. This dispersion of code "all over the place", which is exacerbated by the integration of different programming models and languages for the client and server, makes it hard to bundle functionality for replacement through personalization. But since developers cannot anticipate all possible ways of extending an application, **how do we design a web application such that future extensions are easy to integrate?** Second, the code of the extensions will have to be activated, it may have to pass data back and forth with other application components, and it requires access to the state of the application in the database. **How do we address the security concerns of integrating such untrusted code into a running web application?**

1.2 SAFE

The **Safe Activation Framework for Extensibility** [3, 4] is a framework for the design of data-driven web applications tailored to user-provided customization. Let us give a brief overview of SAFE and its features.

Design for Personalization

SAFE structures data-driven web applications into a *hierarchical programming model* inspired by Hilda [8, 7]. Functionality is clustered into so-called **f-units** that contain all the relevant code to implement a component of the application. The control flow of the application has a clean hierarchical semantics: An f-unit is activated by its parent f-unit and becomes its child resulting in a tree of activated f-units. This so-called *activation tree* naturally corresponds to the hierarchical DOM structure of an HTML page. There are two well-defined points of information flow for an f-unit: Its activation call, through which the an f-unit was activated by its parent an f-unit, and queries to the database where the state of the application is stored. Thus a user who would like to personalize an application simply has to replace an existing f-unit with a new f-unit of her choice or design. Such customizations are dynamic in that f-units are registered and activated without stopping the running system. These dynamic software updates (DSU) avoid costly unavailabilities of the running system [6, 1].

SAFE has a security model that is tailored towards the integration of untrusted code by splitting the code of an f-unit automatically between client and server. Database queries specified by a programmer will never appear in the client code, sanitization of query values to prevent SQL injection attacks automatically occurs on the server, and event handlers for asynchronous update request end up in the client. SAFE also contains a reference monitor which takes care of all low-level details such as secure registration of f-units, access control, and verification of user actions and requests received from the client.

Note that even only achieving modularity when designing data-driven web applications is nearly impossible. The f-units in SAFE can be thought of as classes in object-oriented programming. For web applications, however, there are several different languages (for example, HTML, PHP, Java, JavaScript, SQL, CSS) providing different data models for the different application layers (e.g., the relational model for databases, Java objects for the application logic, hyperlinks for website structure, and form variables for web pages). This variety makes it hard to achieve modularity since fragments of different languages are in different parts of the source tree. Usually, a single JavaScript command like include('moduleA') is not sufficient. Assume, as an example, moduleA is responsible for displaying some <div> elements which are supposed to appear only two seconds *after* the main HTML page has been loaded. In this case, an event handler for onload events of the document has to be modified accordingly. Typically, such an event handler is a named JavaScript function, referenced in the <body> tag of the main HTML page: <body onload='pageLoaded()'>. The JavaScript function pageLoaded() is uniquely declared at some other location, most likely in the <head> area of the HTML page. This declaration has to be updated if moduleA needs some actions to be performed when the page has been loaded; some lines of JavaScript code have to be added to the body of the function. For a different language, for example for PHP, the integration of new functionality again is different. Another difficulty in the integration of new functionality is to ensure that namespaces of different pieces of code do not interfere. Assume that we have two code fragments A and B which each have an HTML element with id studentList and corresponding CSS specifications. A namespace concept would separate the CSS for A from the CSS of B, and we have to add this manually in order to resolve this conflict. As part of its hierarchical programming model, SAFE provides solutions to address all these problems.

Client-Server Consistency

Modern interactive web applications give the user a feeling of locally executing a fully-fledged software binary by communicating with the server asynchronously. The typical way of implementing this is through client-side event-driven programming. One challenge when writing this client-side code is that the state of the application at the client can be different from the state at the server, since other clients simultaneously connect to the same application and may modify the state of the system at the server, for example when one user updates a data item that another user is currently displaying. To avoid such inconsistent updates, the programmer would have to manually include all kinds of consistency checks, which is error-prone and cumbersome. SAFE alleviates the developer from this burden by making consistency checks a first-class citizen in the model, providing an easy to use *SQL-based declarative state monitoring interface* that automatically derives the necessary checks. SAFE automatically compiles the developer code to safe state transitions which cleanly abstracts out concurrent updates into standard serialization semantics known from interacting with a database.

Ease of Development

SAFE also includes many different mechanisms to minimizing the amount of low-level code that a developer has to write. (1) Programs in SAFE are written in SFW, a high-level programming language that abstracts away many low-level code fragments through appropriate high-level statements. For example, it is often cumbersome to specify explicit loops and to iterate over the objects of a particular data structure thereby struggling with implementation details like counters, pointers or break conditions of that particular loop. SFW contains high-level constructs for many of these commonly re-occurring patterns. (2) One of the design principles of SFW is that we did not invent a new language, but rather create a *framework that encompasses existing languages.* Our framework provides the full expressiveness of languages like HTML, PHP, SQL, and Javascript, but allows for shorter, yet semantically precise *shortcuts that significantly reduce the amount of code* a developer has to write. (3) Note that application developers may have to know about other elements in the DOM tree in order to ensure that all elements have pairwise unique IDs, and other elements are correctly addressed, e.g., whether an element has the innerHTML property or the value property. SAFE's modularization fosters *local* *understanding* because it automatically ensures that IDs are unique and that the developer only needs to locally care about the elements of the corresponding f-unit. (4) Today a lot of similar event-driven code for asynchronous server requests has to be written. However, the code for the update of an exam grade in a course management system is not much different from the code of updating the matriculation number of a student. In the spirit of DRY (Don't Repeat Yourself), as in *Ruby on Rails* [2], SAFE requires the developer to specify information and code at most once. For example, the code for the initial rendering of an f-unit is also used later to provide partial updates of modified data. No complicated event handlers have to be specified to rebuild certain elements in the browser's DOM tree. Another feature to reduce the amount of hand-written code is the paradigm of *convention over configuration*: SAFE decreases the number of decisions a developer has to make by establishing useful conventions on parameters and names of variables.

Chapter 2

SAFE — the SAFE Activation Framework for Extensibility

SAFE provides a hierarchical programming model which naturally builds upon the hierarchical DOM structure of web pages. The most constitutive components in SAFE are its so called **f-units**, see Figure 2.1 for an illustration. An f-unit clusters all code fragments for a specific functionality within a web page, including the business logic, the visual appearance, and the interaction with users or other f-units. This clustering provides a clear level of abstraction through well-defined *interfaces* for each f-unit. The modularity of an f-unit relieves the programmer from struggling with variable scopes and their interference.

As a result, this abstraction provides an elegant way of composing web pages out of several different f-units. A web page, also referred to as **SFW page**, inhabits exactly one so-called **activation tree** (inspired by Hilda [8, 7]), in which f-units are hierarchically organized. Figure 2.2 shows an example of an activation tree with its corresponding

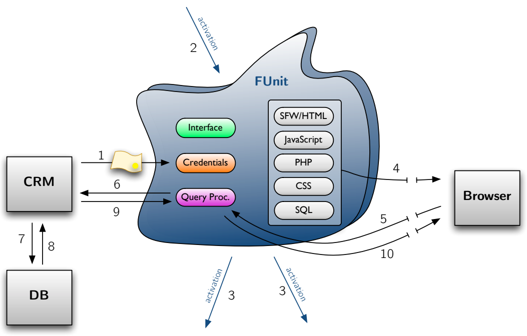
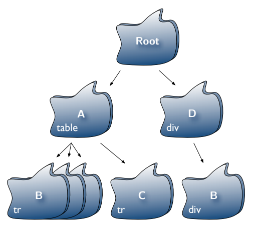


Figure 2.1: Integration of an f-unit.



```
<HTML>
  ...
  <table id='Root.A'>
    <tr id='Root.A.B_1'> ... </tr>
    <tr id='Root.A.B_2'> ... </tr>
    ...
    <tr id='Root.A.C'> ... </tr>
  </table>
  <div id='Root.D'>
    <div id='Root.D.B'> ... </div>
  </div>
  ...
</HTML>
```

Figure 2.2: Activation tree and its corresponding web page.

HTML code. A node in the activation tree (i.e., an f-unit) corresponds to one or more nodes in the HTML DOM tree.

The insertion of an f-unit FooBar in the activation tree is referred to as **activation** of FooBar (Figure 2.1, step 2). More precisely, an f-unit is activated by its parent f-unit and thereby receives activation data through well-defined interfaces. The f-unit can use the activation data or data obtained directly from the database through queries to display parts of the web page. An f-unit can also activate other f-units, its child f-units (Figure 2.1, step 3). More details on the activation are provided in Section 3.4.

Throughout this document, we will often use the name FooBar to refer to an arbitrary f-unit. For various reasons: FooBar has the nice initial letter F and contains two capital letters inside. We use this fact to demonstrate how filenames (in particular lowercase ones) corresponding to FooBar are derived, and when and how they are used.

2.1 F-units — the main building blocks of SAFE applications

F-units constitute both functionality and data of a web application. Hierarchically organized in the activation tree, f-units bundle functionality of different tiers, e.g., client code (HTML, JavaScript, CSS), server code (PHP, SQL), and reactive code for asynchronous message transfers (JavaScript, AJAX). An f-unit is integrated using the SAFE Integrator from the SAFE tool suite. In the integration process, the various files an f-unit consists of are installed to the system. These files of an f-unit are introduced in this section. Please consider Section 2.2 for more details on the integration process.

We assume an f-unit with given name FooBar. The first character of an f-unit name must be uppercase (please check Section 7.1 for more details about the syntax). All files of FooBar are organized in the bundle folder FUnitFooBar.bundle: The HTML/SFW skeleton of FooBar together with the activation of child f-units is specified in the file funitfoobar.sfw, the f-unit's local data is stored in tables in the database whose definitions are specified in the file funitfoobar.db, style (CSS) and client side functionality (JavaScript) is specified in the files funitfoobar.css and funitfoobar.js. We will discuss each of these files in detail below.

[.sfw] The file FUnitFooBar.bundle/funitfoobar.sfw contains the HTML skeleton of an f-unit, includes the activation calls for other f-units, and specifies the interface for activations.

The first line considered by the SFW compiler while parsing an f-unit SFW file has the following syntax.

```
<funit FooBar(static1,static2,...)#a2e92b94c834f3e...>
```

The name of the f-unit is specified right after the funit keyword and must match the name of the bundle folder and its contained filenames. In this case, the filename would have to be FUnitFooBar.bundle/funitfoobar.sfw. The round brackets contain the names of the **static activation arguments** (☛ Section 3.4.2, page 26). These activation arguments can be named by arbitrary strings and can be accessed only within the scope of the f-unit FooBar. After the hash symbol #, the **authentication credentials** are specified. The credentials are hand out by the owner of an application in order to authenticate third-party f-units against the application. The credentials are obtained during the integration process (☛ Section 2.2, page 18).

Any code above the tag <funit ...> is ignored and can hence be used as a location for documentation of the f-unit.

Please consider Chapter 3 to learn more about the inner scope of the funitfoobar.sfw file and the SFW tags in general.

[.db] The file FUnitFooBar.bundle/funitfoobar.db contains various declarations of tables and views for FooBar. The syntax for database files is explained in detail in **??**.

✓ **Local tables** are f-unit-associated data stores that can only be accessed by the single f-unit that owns the tables. Local tables are created in the application-wide database within the namespace of the owning f-unit, e.g., FooBar's local table tab is registered as foobar_tab in the database.

Each local table exists only as a single instance in the database, independent of the number of activated instances of FooBar. In other words, all instances of FooBar share the same database tables.

Below are two examples for the declaration of local tables. The first declaration of students consists of 5 fields. The first field uid refers to a user in the system (USER) and serves as primary key (PRIMARY). The specification of type USER automatically derives the type of the uid field from the generic users table and automatically establishes a corresponding foreign key constraint.

The type OWNER is used to define a column that holds the owner of the data item stored in the corresponding row. OWNER is a subtype of type USER. Please note that every table needs to define exactly one owner column. ⓘ

```
LOCAL TABLE students (
  uid USER PRIMARY
  matriculation VARCHAR(20)
  enrollment DATE
  department VARCHAR(100)
  professor OWNER
)
```

The second example defines a local table messages which has the special property SINK, meaning that no entries can ever be removed from that table. The field id specifies a unique message identifier and is modeled as primary key (PRIMARY) with an automatically incremented value (AUTO). The special type OWNER in the field from imposes constraints to ensure that a tuple in the table messages can only be inserted/updated under the condition that the from field carries the user id of the user who is authenticated at runtime.

```
LOCAL TABLE SINK messages (
  id PRIMARY AUTO
  from OWNER
  to USER
  msg TEXT
  INVARIANT friends(from,to) OR colleagues(from,to) OR admin(from)
)
```

Moreover, a couple of invariants is specified: the *predicate* friends must hold for the values from and to, i.e., sender and recipient of messages must be friends. A **predicate** can either be an *input table*, a *local table*, a *local view*, or an *output table*. In this case, also colleagues and the special user admin can send messages to any other user in the system.

Local tables can hold **initial data**. These data sets are specified as regular insertion queries.

```
INSERT INTO messages SET from='#public', msg='Welcome!'
```

Along the design guidelines of SAFE, every data set needs a dedicated owner. As users are first-class citizens in SAFE, there is no such user available at integration time. Therefore, initial data must be annotated with the static owner #public.

✓ **Local views** are views that can be used in any acyclic context inside FooBar. Unary local views can be thought of as groups, like the group of admins.

```
LOCAL VIEW admin   = SELECT 'root'
LOCAL VIEW friends = SELECT a.uid, b.uid
                     FROM sfw_users a, sfw_users b
```

✓ **Input tables** constitute database schemata that specify the format which FooBar expects when FooBar is wired to other f-units. A *wiring* combines fields of input tables with specified fields of another f-unit's output tables. The specified types of an input table have to match the types of the corresponding output table. Types of input tables are specified as follows:

```
INPUT TABLE colleagues (
  u1 USER
  u2 USER
)
```

✓ **Output tables** are defined in a similar way as local views are defined. Output tables, however, expose data to all other f-units and are hence publicly visible. The declaration of output tables is derived from standard SELECT syntax:

```
OUTPUT TABLE msg_titles (
  SELECT ...
)
```

The public names of output tables are automatically prefixed in order to avoid name clashes: the output table msg_titles of f-unit FooBar is available in the database under the name foobar_msg_titles.

We stress that every output table needs two special fields named key and owner. The owner field must hold the owner of a dataset, the key column needs to have unique key values, so it is recommended to base the key values on unique key values of existing tables. Every output table automatically obtains a special column ukey, which constitutes a prefixed key that is unique among all keys of all output tables. Such globally unique values are necessary for the wiring of f-units (☛ Section 4.1.1). The ukey column does not have to be declared by the developer. ❗

Output tables can have the special attribute STEADY. **Steady output tables** must not depend on local data that could potentially change upon activation. For example, assume FooBar to have a local table counter which counts the number of activations of FooBar. Every time an instance of FooBar is activated, a counter is increased, hence the table counter is updated. If an output table exposes data contained in this local table counter, the output table may not be declared STEADY.

```
OUTPUT TABLE msg_titles (
  SELECT ... FROM ... WHERE ...
  STEADY
)
```

[.int] The file FUnitFooBar.bundle/funitfoobar.int contains the **public interface** for FooBar. be generated automatically using the interface generator. Alternatively, interfaces can be specified manually.
// todo: more details here.

[.css] The file FUnitFooBar.bundle/funitfoobar.css contains various CSS3 declarations for FooBar. Here is an example with a class-based rule and an identifier-based rule.

```
div.neutral {
  background-color: #DA6F00;
  float: left;
}

#logo {
  float: left;
  color: #FFFFFF;
  font-size: 3.25em;
  text-shadow: 0 1px 1px #3E3E3E;
}
```

Section 3.7 provides more information about cascading style sheets in SAFE.

[.js] The file FUnitFooBar.bundle/funitfoobar.js contains the JavaScript code for FooBar. A JavaScript function f must be defined using the syntax this.f = function(args) { . . . }. Here is an example for a function show.

```
this.show = function(gid,...) {
  gid = this.storage.wrap('selectedPeer', gid);
  if (gid !== null) {
    document.getElementById('group' + gid).style.display = 'block';
    if (this.last_gid === undefined || this.last_gid !== gid) {
      ...
    }
  }
  this.last_gid = gid;
}
```

The function is called in the JavaScript context of FooBar with a manually prefixed function call. We will – hopefully soon – turn the manual prefixing in an automated prefixing.

```
<js>
  funitfoobar.show(gid,...);
</js>
```

(!) When used as *check function* (☛ Section 3.5) or as *finalize function* (☛ Section 3.5), the occurrence of the function name must not be prefixed, hence funitfoobar must be omitted.

The compilation and deployment of JavaScript code is similar to the deployment of CSS (as described above).

More information about JavaScript is contained in Section 3.6.

2.1.1 The Master f-unit

There is one generic **Master f-unit** from which all other f-units inherit certain properties. This f-unit called FUnitMaster is the same for all web applications and is usually not altered during the modeling of an application. Its folder FUnitMaster.bundle is delivered with SAFE and requires no further modification. The integration of an f-unit (☛ Section 2.2) automatically creates the dependencies to the Master f-unit.

2.2 Integration

The process of adding an f-unit to the system is referred to as **integration** of an f-unit. Besides f-units, also SFW pages must be integrated before they can be used. The same holds for customizations. The following steps have to be executed:

1. Upload the f-unit bundle (for instance FUnitFooBar.bundle) or the page (for instance page.sfw) or the customization (for instance cust1.cst) to the root directory of your web

server. The f-unit bundle must contain at least the SFW specification funitfoobar.sfw, the interface file funitfoobar.int, and the database specification funitfoobar.db.

2. Start the SAFE Integrator from SAFE's tool suite. It should be located on your web server at /tools/integrator/.

3. Search the list of available objects for the f-unit, the page, or the customization you want to integrate. In case the requested object does not appear in the list, it has not been uploaded completely. In particular concerning f-units, make sure that all files mentioned in the upload step (step 1 above) are contained in the bundle.

   Click *integrate* to integrate the object or click *remove* to remove the object.

Technical aspects of the integration process are discussed in Section 5.5.

Chapter 3

The Modeling Language SFW

SFW has been designed for a secure and unified handling of the interleaving combinations of HTML, PHP, SQL, CSS, and JavaScript. In addition, SFW provides shortcuts for common programming patterns by extending the syntax of the HTML 5 standard. Developers hence do not have to learn a completely new language. Instead, they can rely on what they have been using for years — but they can create rich web application much faster and more reliably than before.

This chapter introduces the main features of SFW, starting with general concepts such as the SFW *specification tags*, the *activation* concept, SFW *forms*, etc.

3.1 Declarative specifications

The rough idea behind a specification at a higher abstraction level and the production of computer-generated source code thereof is motivated by the following facts. *First*, Web applications often use common patterns which are better taken from well-tested code libraries instead of being manually programmed every time from scratch. Typical replication methods like copy & paste often introduce logical or structural flaws that are hard to detect at a later stage. *Second*, human errors in the programming process should generally be detected and avoided at the earliest possible point in time. Such errors introduce not only bugs in the functionality, but might also impose severe security threats. *Third*, a restricted set of allowed operations simplifies the detection of malicious code or even prevents security holes which might be introduced on purpose. Assume for instance a system-provided (and hence trusted) encryption function for which all randomness is correctly chosen. A malicious programmer using the predefined function cannot bring his own "pseudo-randomness" into play, and hence cannot introduce a backdoor to decrypt confidential user data offline. Although SAFE allows the specification of arbitrary program code with potentially malicious functionality, it will certainly look suspicious if existing functionality is rewritten.

The two latter aspects are particularly relevant for data-driven Web applications with sensitive and privacy-critical user data. Not only is the programming of such security checks vulnerable to bugs and attacks, but also is it quite tedious to program all necessary checks, to add error handling for the reactive multi-tier code, to chose encryption/signature keys correctly, to establish database connections, etc. The automated compilation of higher-level languages hence not only produces better program code, but also significantly reduces the workload of a programmer.

3.2 Database queries

One of the major purposes of data-driven web applications is the handling of diverse information, typically stored in databases. The exploding nature of information aggregation makes fast and reliable database access inevitable. The developer of a web application shall hence not be bothered with technical details about database connection, but instead she shall concentrate on the functionality to be modeled. It is thus straightforward to design a web application using a framework which tremendously simplifies the interface to the database. In SAFE, all details about the database connection are specified in a global specification file (☛ Section 5.4.1). The execution of queries against the database becomes easier than ever before.

In the following, we focus on SAFE's database abstractions; all technical details on database queries are discussed in Section 5.7 on page 45.

SELECT queries. Within SFW, there are various ways to execute SELECT queries against the database. There is a difference in retrieving a single tuple from retrieving multiple tuples. The **first tuple** returned from the execution of a query is obtained via the statement

```
<queryFirst query="SELECT name AS group FROM groups WHERE gid='5'">
<p>You are registered for group $$group.</p>
```

This statement binds the variable $$group to the corresponding value in the returned tuple (or to the empty string if there is no such tuple).

In case **all tuples** of a query shall be obtained from the database, a for loop is the right choice. The following example prints an unordered list <ul> in which each result tuple is printed as a list item <li>.

```
<ul>
  <for query="SELECT gid AS id, name AS title FROM groups">
    <li> Group $$title ($$id) </li>
  </for>
```

```
</ul>
```

The SQL row variables $$id and $$title are bound to the corresponding values inside the for block.

INSERT / UPDATE / DELETE queries. When issuing queries in order to update the database, the developer has to ensure that all owner constraints be addressed appropriately. More precisely, every **INSERT** query must explicitly set the owner column to $%me representing the authenticated user. An **UPDATE** query must affect only those database tuples that are owned by the currently authenticated user. In particular, no update query is allowed to modify the owner column. Likewise, every **DELETE** query is restricted to touching tuples that are owned by the authenticated user. More technical information about update queries is contained in Section 5.8.

3.3 General control statements

Branching. In order to check whether variables are empty or not, SFW supports the following two if statements. The execution of the corresponding blocks only depends on whether the specified variable is empty:

```
<if:Empty $street>                      <if:NotEmpty $street>
  Street: n/a                             Street: $street
</if>                                   </if>
```

Another two if statements treat equality checks.

```
<if:Equal($pw,"523!qs3ma")>             <if:NotEqual($category,$$cat)>
  Login successful!                       Invalid category!
</if>                                   </if>
```

There exist the following extensions:

```
<if:True $exp>  ... </if>         <if:Positive $exp> ... </if>
<if:False $exp> ... </if>         <if:Negative $exp> ... </if>
```

The semantics of True and False is the same as in PHP: the expression $exp is evaluated to either Boolean *true* or *false* using the PHP interpreter. The content inside the positive (negative) tag is evaluated whenever $exp>0 ($exp<0).

Alternatives can be expressed using the else statement:

```
<if:Empty $street>
  Street: n/a
<else>
  Stree: $street
</if>
```

All <if> and <else> statements can be nested, for instance to express else if branches. The **Boolean connectives** OR and AND with parentheses can be used to create hierarchical expressions.

```
<if:Empty($street) AND (NotEmpty($city) OR Equal($status,1))>
  Missing: street
</if>
```

As above, parentheses can be skipped around unary predicates. And all predicates can be written using lowercase letters.

```
<if:empty $street AND (notempty $city OR equal($status,1))>
  Missing: street
</if>
```

Iteration. In order to print icons multiple times, you can repeat a block a certain number of times. If, for instance, you want to display stars to express how much customers like a product, you might find the following helpful.

```
<repeat $rating>
  <img src='star.png'>
</repeat>
```

The list of SFW control statements is not yet complete and will be expanded in the future.

3.4 Activation

The term **activation** refers to the action of creating fresh instances of f-units, the main building blocks of which web applications in SAFE are composed. The activation of an f-unit in SFW resembles the instantiation of a class in object-oriented programming. There are similarities in several aspects:

- *Multiple instances* f1,f2 ... of an f-unit FooBar can coexist in the activation tree.
- *Activation arguments* can be passed to an f-unit upon activation. This is one of two ways for an f-unit to receive *input* from the calling page or from other f-units.[1] More precisely, an activation call can carry *static* activation arguments (☛ Section 3.4.2), as well as *dynamic* activation arguments (☛ Section 3.4.3).

In contrast to instantiations in object-oriented languages, activations come in two variants: there are *static* and *dynamic* activation calls.

1. **Static activation calls** activate exactly *one* instance of an f-unit, independent of any data the activating f-unit has. As an example consider a table listing all registered students for a course. The table needs a headline, independent from how many

[1]The second way to receive input is by wiring (☛ Section 4.1).

students have registered. The headline is displayed via a static activation call of the f-unit StudRegTableHead.

```
<activate:StudRegTableHead("Summer 2013","CS 512")/>
```

The activated instance obtains the static activation arguments (☛ Section 3.4.2) Summer 2013 and CS 512.

Please note that static activation calls must not occur in the body of SFW loops (e.g., in for loops ☛ Section 3.2). For multiple similar activations, it is recommended to use dynamic activation calls (see below). ⓘ

2. **Dynamic activation calls** activate as many instances as there are elements in a collection provided by the dynamic activation arguments. Continuing the above example, we might wish to activate instances of an f-unit StudRegTableRow that displays a table row holding student data. For each of the n students we wish to activate exactly one instance of StudRegTableRow, so that each row in the table corresponds to exactly one student. Clearly, we could program some for-loop to create n static activation calls of StudRegTableRow. Instead, we can simply specify a dynamic activation call:

```
<activate:StudRegTableRow("Summer 2013", "CS 512")
  query="SELECT matriculation, firstname AS name FROM students" />
```

The query returns n tuples, hence n instances of StudRegTableRow are activated. Please note that n can be zero, in which case no activation occurs. Each of the n activated instances obtains the static arguments Summer 2013 and CS 512 (☛ Section 3.4.2), and the dynamic arguments $@matriculation and $@name (☛ Section 3.4.3). The latter hold the corresponding values from the respective tuples and declare the arguments $@matriculation and $@name to be used inside the f-unit StudRegTableRow.

Please note that pages (in contrast to f-units) cannot use queries inside dynamic activation calls. For two reasons: pages own no database tables themselves, and they cannot access the database tables of any f-unit. ⓘ

Given an activation call A for f-unit F, both static and dynamic activation arguments are received only by the instances of F that are activated by A. Any other instances of F in the activation tree are not affected, hence any other instances do not receive activation arguments as specified by A. Note that a wiring (☛ Section 4.1) for F provides the same input data to every instance of F, which is completely independent from the specific activation calls.

3.4.1 Unique activation ID

Every activated instance is assigned a unique **activation ID**. This holds for both static and dynamic activations. The unique identifier is a hashed value that is influenced by several values. For dynamic activation calls with an activation query, (among others) the field id in the query influences the hash value. If the query has no such id field, a generic counter value is used instead. For dynamic activation calls with PHP array, the keys of the activation array are considered. Please have a look at the example in Section 3.4.3 for an illustration of the keys of the activation array.

3.4.2 Static activation arguments

Values $v_1, v_2, \ldots$ passed upon the activation of an f-unit Course are called **static activation arguments**. Every activated instance for the corresponding activation call of Course obtains the values $v_1, v_2, \ldots$.

```
<activate:Course("Summer 2013","CS 512") />
```

These values can be of arbitrary type: constants and variables are allowed. The values are accessible from inside the f-unit Course using the following syntax.

```
<funit Course(term,title)#af35b91c7...>

  <h1> Term $!term </h1>                          //yields: Term Summer 2013
  <p>  Course $!title contains ... </p>           //yields: Course CS 512 ...

</funit>
```

All static activation arguments must be added to the <funit> tag in the beginning of each SFW file in order to declare the names. The order of the static activation arguments is important. Moreover, every activation call of an f-unit requires the exact number of arguments as specified in the <funit> tag. If f-unit Course requires 2 static arguments, SAFE will complain if Course is activated with either more or less than 2 static arguments.

The expansion of static arguments during compilation by the SFWC compiler is explained in Section 5.12.1.

3.4.3 Dynamic activation arguments

In the case of dynamic activations, each activated f-unit can obtain different values from the activation call. These values are referred to as **dynamic activation arguments**. The source for the data can either be a SQL query or a PHP array.

```
<activate:Course("Summer 2013","CS 512")
  query="SELECT s.name, s.matriculation AS matr FROM students AS s" />
```

The PHP array can be generated automatically, e.g., by some POST/GET input, can come from a database source or can be provided manually (as shown below).

```
<php>
    $students = array(
                    1 => array("name" => "Alice", matr => "1234567"),
                    2 => array("name" => "Bob",   matr => "1234568")
                 );
```

```
</php>

<activate:Course("Summer 2013","CS 512") array=$students />
```

The keys of the activation array can be chosen arbitrarily. Instead of 1,2,..., arbitrary strings can be used. These keys influence the unique activation ID (☛ Section 3.4.1).

In contrast to static activation arguments, the specification of dynamic activation arguments is more flexible in that an f-unit does not have to specify which dynamic arguments the f-unit requires. This freedom implies that if an entry in the $students array is not specified (e.g., the matriculation is not set), there will be appear an empty value instead of a warning. Dually, the activation query or the activation array can contain more fields than the activated f-unit consumes.

The expansion of dynamic arguments during compilation by the SFWC compiler is explained in Section 5.12.1.

3.5 Forms and database updates

In traditional web development, one would specify an HTML form as follows:

```
<form id="myForm123">
  ...
  <input type="button" onclick="javascript:submitForm('myForm123')">
  ...
</form>
```

However, as this code might be activated several times, the form ID would appear several times in the DOM tree, which would result in invalid HTML code (☛ Section 3.8.2). In this case, the form ID can be left blank, then the compiler generates the ID automatically. The automatically generated ID is unique throughout the overall HTML document and can be referenced later via a dedicated variable (☛ Section 3.9).

Forms with such special SFW features are enclosed in SFW form tags

```
<sfw:form>
  ...
</sfw:form>
```

The form contains input elements each with a name that is referenced using the following syntax.

```
<sfw:form>
 ...
 <input type='text' name='firstname' value='Alice'>
 ...
 <input type='button' value='update'
   onclick="query:UPDATE students SET name='$#firstname' WHERE ...; check">
 ...
</sfw:form>
```

There are two input elements, a text input and a button. A click on the button executes the specified query against the database. References to values of input elements in the query, in this case for the first name, are done via $#inputelem, in this case $#firstname. Please do not omit the usual quotes in the query, e.g. for name='...' and for any other string values. An overview of syntactic placeholders such as $#firstname can be found on page 31 in Section 3.9. Instead of onclick, any other JavaScript event can be used.

It is important to mention at this point that all user input will be sanitized by the database server before the specified queries are executed. It is the separation of code (the specified query) and data (the user inputs) that allows for a perfect input sanitization that prevents all types of SQL injection attacks.

References to PHP variables $id, to static activation arguments $!prm, or to dynamic activation arguments $@prm can be used with the usual syntax in any context. The compiler does not place the update query in the client code, but instead only a reference to the query. The query ends up in the corresponding AJAX code on the server. All dynamic values however, i.e. form values, PHP variables, and activation arguments cannot appear in the static query in the AJAX code on the server. A value like $id above hence shows up in the client code and is sent to the AJAX code on the server. All these **dynamic values** are automatically **encrypted** and then **MAC'ed** to ensure confidentiality and integrity of the data. The developer of an f-unit will not notice anything (no key generation, no validation, etc.), but instead, security is automatically enforced in the background.

The JavaScript function check is called before the query is executed. The specified query is only executed if this so-called **check function** check returns the string "OK". Any other string will be displayed as error message. If no value is returned, the client will display "undefined". So please make sure the check function returns an appropriate string value. The check function is declared like any other JavaScript function (Section 3.6). It expects a single argument: the ID of the surrounding form, if existent. If no checks need to be performed before the execution of the specified query, the semicolon and the function name can be omitted. Note however, that you need to specify the check function in case you wish to specify a finalize function (see below).

After the query has been executed, a second JavaScript function, the **finalize function**, can be called. In this case, a second function name has to be specified after a second semicolon: [#41b9e58d]

```
<input type='button' value='update'
 onclick="query:UPDATE students SET name='$#firstname' WHERE ...; check; finalize">
```

The finalize function can be used to hide a form after the form data has been processed or to receive the exit status of the query execution. The function has to be declared in the local JavaScript namespace of the corresponding f-unit, similar to the check function. The finalize function expects a single argument, which is the exit status with one of the following values. (1) Successful execution with partial page update. Parts of the page have been refreshed automatically via AJAX in the background. (2) Successful execution with local updates. (3) Successful execution with global updates. The page will be reloaded, hence you should ignore this case. Any other exit status indicates an error.

3.6 JavaScript

The declaration of a JavaScript function myFunc for the f-unit FooBar in the file FUnitFooBar.bundle/funitfoobar.js looks as follows:

```
this.myFunc = function(arg1,arg2,...) {
  // function body here
};
```

The function is called in the JavaScript context of FooBar with a manually prefixed function call:

```
<js>
  funitfoobar.myFunc(arg1,arg2,...);
</js>
```

Instead of using <js> to start a JavaScript block, you can use <js:lazy> instead. This defers the execution of the specified code to a point in time when the DOM tree has fully been loaded. The following example hides the own instance:

```
<js:lazy>
  document.getElementById('$@@').style.display = 'none';
</js>
```

Clearly, you need to wait until the instance of your f-unit has been constructed, before you can hide it.

For technical information on how JS files are included, please have a look at Section 5.4.2.

3.6.1 Using SFW variables in JavaScript

The example above shows how to interfere SFW variables with JavaScript. The SFW value $@@ is expanded to the ID of the HTML element which represents the current f-unit instance (☛ Section 5.12.1). The value is "pasted" in clear, so you need to put quotes around the value in a JavaScript environment.

3.6.2 Return messages for asynchronous data updates

The status report of an asynchronous database update (busy/success/error) is output to a special element in the DOM with id sfwMessage. In order to make the message appear visually more appealing or to position the message at the right position, the element sfwMessage is assigned one of the following CSS classes:

- sfwMessageBusy is for all status messages denoting that an update is in progress.
- sfwMessageSuccess is used for successful transactions.
- sfwMessageFailure is used in cases with any database update error.

3.7 CSS — Cascading Style Sheets

Whenever a page page.sfw is requested by a browser, the CSS specifications of all f-units occurring in the activation tree of page.sfw are dynamically compiled to a **global CSS file** page.gen.css, which is automatically included by the corresponding page. The compilation of required CSS specifications happens at activation time of the corresponding f-units. The advantage of this late compilation guarantees that also the CSS specification of customized f-units is taken into account.

The CSS for f-unit FooBar is specified in its FUnitFooBar.bundle/funitfoobar.css file (☛ page 17) using standard CSS syntax. Style rules for the wrapper surrounding FooBar are specified without any CSS selector, just the style rules in curly braces:

```
{
  padding: 5px;
  background-color: #CC88AA;
  ...
}
```

If JavaScript inside FooBar needs to access this wrapper, e.g. using jQuery, it should use the selector $('. sfwfunitwrapper.sfwFUnitFooBar').

Sandboxing of CSS for nested f-units is explained in Section 5.13.

3.8 General advice

The composition of web applications out of several f-units requires a clear separation of f-unit content throughout the overall application. Any interference of program variables, of CSS selectors, or of element IDs across f-units must be prevent by **sandboxing** techniques. Usually, these techniques magically work in the background and are thus hidden from the developer.

3.8.1 SFW Comments

Any code in an .sfw file specified above the tags <funit> and <html> is ignored by the compiler. Developers can use this space to document the corresponding f-unit or HTML page.

Comments inside the tags <funit> and </funit, or <html> and </html> depend on the current mode. In PHP, the comments can be initiated with # or with //. In plain SFW, we recommend standard HTML comments:

```
<!-- comments here -->
```

Such comments are not contained in the final output of the SFWC compiler.

3.8.2 Unique element IDs in the DOM tree

We currently ask developers to avoid static element IDs wherever possible. Think of an f-unit FooBar that is dynamically activated more than once. A line such as

```
<button id='myButton'> Click me! </button>
```

is not modified by the compiler and hence the ID myButton would occur multiple times in the DOM tree. This results in invalid HTML code. We therefore ask developers to either let IDs be generated automatically (for instance form IDs ☛ Section 3.5) or to specify IDs that contain the unique activation ID of the activated f-unit instance. Developers can access the activation ID via $@@ (☛ Section 3.9). The button ID can hence be specified as (!)

```
<button id='myButton$@@'> Click me! </button>
```

which guarantees unique element IDs for the button across the entire DOM tree. The ID can be reused at any point inside the f-unit FooBar.

3.9 Variables, arguments, and their expansion

This section provides an overview of all (special) variables and identifiers, and the activation arguments. Note that the expansion (or compilation) of special variables does not have to be considered by a usual developer.

The following arguments are *variable*, i.e., all characters in the range from 'a' to 'z' can be exchanged by other characters from that set. In other words, all strings such as 'var', 'arg', and 'elem' are just placeholders that represent an (almost) infinite number of possible variables, arguments, and elements.

- **$var** represents a standard **PHP variable**.

  PHP variables can be accessed for read access in any SFW context, for instance by

  ```
  <h1> $headline </h1>
  ```

  Assignments of PHP variables must be specified in PHP blocks:

  ```
  <php>
        $headline = "Welcome " . strtoupper($firstname) . "!";
  </php>
  ```

- **$$var** represents an **SQL row variable**.
- **$!arg** represents a **static activation argument** (☛ Section 3.4.2).
- **$@arg** represents a **dynamic activation argument** (☛ Section 3.4.3).
- **$#elem** holds the value of the **input element** with name `elem` in the surrounding form.

The following arguments are *constant* in that all characters from 'a' to 'z' are fixed and cannot be exchanged.

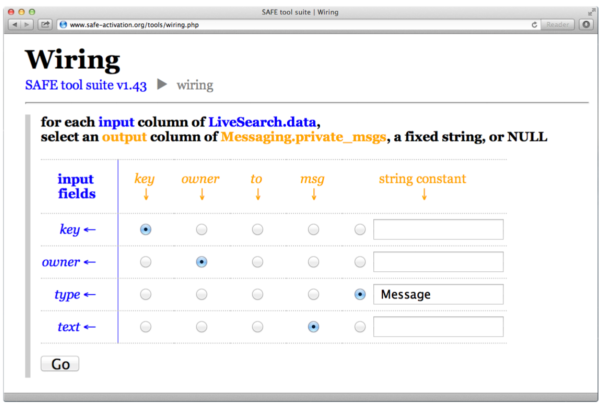
- **$%#** represents the **dynamic activation number**. If you want to enumerate the positions in a dynamic activation call, `$%#` provides the number n for the n-th activated instance.
- **$%data** represents the **local data identifier** and is expanded to the local data directory of the f-unit.
- **$%form** represents the **last form identifier** which holds the ID of the form with the last automatically generated ID.
- **$%me** represents the **authenticated user identifier**, hence the unique system-wide identifier of the authenticated user. This special value can only be used in database queries, not in arbitrary SFW contexts.
- **$%res** represents the **local resource identifier** and is expanded to the local resource directory of the f-unit.
- **$%rows** represents the **SQL row counter**, which contains the number of rows for the result of the latest query that has been executed.
- **$%uhash** represents a **hash of the authenticated user ID**. If no user is authenticated, the value is empty.
- **$@$** represents the **clock variable**. It contains the current clock value of the CRM.
- **$@@** represents the **activation ID** of the activated f-unit instance.

Chapter 4

Wiring and Customization

4.1 Wiring

SAFE's wiring mechanism lets a **customizer** establish fixed data flows between f-units. An output table A.out of some f-unit A is said to be *wired* to an input table B.in of some f-unit B. We also refer to this wiring as A→B, hence A is wired to B. This wiring is a mapping of columns of the output table to columns of the input table. Figure 4.1 shows the graphical web interface for the wiring of f-units. The input column type shows a special case: type does not obtain its data from a column of the output table, but instead has a constant string value.



Wiring

SAFE tool suite v1.43 ▶ wiring

for each input column of LiveSearch.data,
select an output column of Messaging.private_msgs, a fixed string, or NULL

input fields	key ↓	owner ↓	to ↓	msg ↓	string constant ↓	
key ←	●	○	○	○	○	
owner ←	○	●	○	○	○	
type ←	○	○	○	○	●	Message
text ←	○	○	○	●	○	

Go

Figure 4.1: Screenshot of the wiring process.

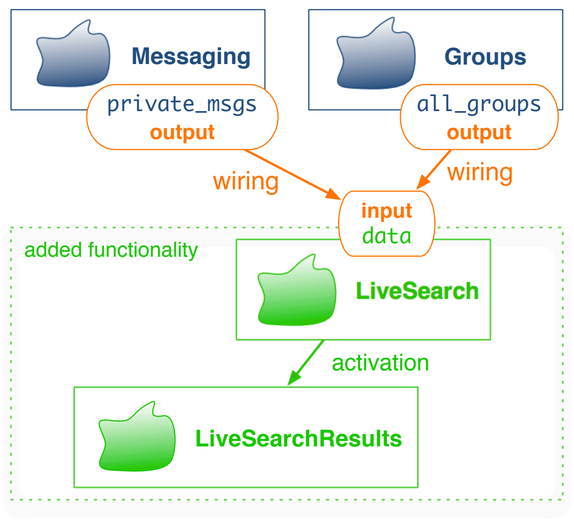


Figure 4.2: Wiring: union of two output tables.

4.1.1 The ukey column

Every output table must declare two special fields: the owner column to refer to the owner of a row, and the key column to hold a unique key value (☛ page 16). Automatically upon integration, every output table obtains a column ukey which constitutes a prefixed key that is unique among all keys of all output tables. Such globally unique values are necessary for the wiring of f-units. Consider the scenario depicted in Figure 4.2. The output tables Messaging.private_msgs and Groups.all_groups are both wired to the input table LiveSearch.data. The f-unit LiveSearch models a functionality to incrementally search arbitrary content in a web application. The haystack that shall be searched by LiveSearch consists of the datasets that are wired to the input table data.

Whenever LiveSearch wants to refer to a certain entry in its displayed search results (for instance, to display more information, to provide a direct link, etc), LiveSearch needs a unique ID of the corresponding object. If both Messaging and Groups use internal keys in the range $\{0, 1, 2, \ldots\}$, the union would contain several keys more than once. This multiplicity makes it impossible for LiveSearch to uniquely refer to its items.

To overcome this uniqueness problem, every output table is augmented by the ukey column. The values in this column are prefixed in a way that the contained keys are guaranteed to be unique throughout the overall application. It is hence generally required to wire the ukey column of an output table to the key column of an input table.

The ukey column does not have to be declared by the developer.

The ukey as dynamic activation argument

Assume a wiring A.out→B.in. A dynamic activation of instances of f-unit B by f-unit A can be based on output table A.out. One dynamic activation parameter can then be the ukey value of A.out, which allows the activated instances of B to refer to a particular row in its wired input table.

```
<activate:B() query="SELECT ukey,title ,type FROM out"/>
```

4.2 Customization

Customizations allow for easy adaptation of SAFE web applications in terms of functionality and style. More formally, a **customization** $\mathcal{C} = (\mathcal{O}, \mathcal{W})$ consists of two distinct components:

$\mathcal{O}$ is a set of **operations on the activation tree** that insert nodes into the tree and remove nodes from the tree.

$\mathcal{W}$ is a set of **wiring links** that link output tables of one f-unit to the input tables of other f-unit.

Chapter 5

Technical Advice

5.1 How to model an application in SAFE

This section provides a checklist for developers who want to model a SAFE application.

1. Create a new directory to serve as the main application folder or copy it from the delivered skeleton.
2. The application folder needs to contain a copy of FUnitMaster (☛ Section 2.1.1). Please make sure that FUnitMaster.bundle is in your application folder. If you have set up your copy of SAFE using the SAFE setup wizard, the master f-unit should be available right away.

5.2 Web server setup

This section explains which software and tools should run on your web server and how to configure your web server so that you can host a SAFE application. We highly recommend to use the **SAFE setup wizard** that guides you through the process of setting up your personal copy of SAFE. Please copy the zipped archive of the wizard to your web server and extract the archive. You should see a folder wizard. Please open the URL corresponding to this directory using your favorite web browser. Proceed step-by-step and set up your system.

1. You need a standard (open source) web server like the Apache HTTP server (☛ `http://httpd.apache.org/`). Note that the configuration should accept local .htaccess files.

2. Data maintenance is done (open source) using a MySQL server (☛ `http://www.mysql.com/`). After installing a database server, please execute the SAFE wizard or execute the SQL definitions as specified in the file SAFE–***–USER.SQL. The specification contains one or two users whose names and passwords need to be adjusted in both the database specification (SAFE–***–USER.SQL) and in the CRM configuration (☛ Section 5.4.1). We recommend to set up the credential seed in the beginning and to not alter the value later. In case you want to use just a single user, please do not execute all commands to create users, all GRANT commands, and also the definers from VIEW commands. ⓘ

3. If you have not used the SAFE wizard, copy/create at least the following directories and files to the root directory of your web server (for instance to /var/www):

sfwc/	should contain the executable compiler sfwc
src/	contains various necessary components such as the CRM, etc.
tools/	contains the SAFE tool suite
config.crm.php	main CRM config file (☛ Section 5.4.1)

   All your f-units and pages will also end up in the root directory of your web server.

4. An interpreter for Perl (☛ `http://www.perl.org/`) is necessary in order to compile the source code of f-units as a part of the integration process (☛ Section 2.2, page 18).

5. In order to generate interfaces from f-units and SFW pages, a PHP tool available in tools/createInterface.php can be used to automatically extract interface information from f-units and pages.

5.3 User management

The CRM offers a number of methods to manage the users in a SAFE application. By these users we mean user accounts that are created during the runtime of a SAFE application, hence first-class objects, no database users.[1]

Any f-unit with name ending in Auth is allowed to execute the following public methods:

- login(uid,pw)
  authenticates an existing user at the CRM and creates a new session.
- logout(uid)
  destroys a session for the specified user.
- registerUser(uid,pw,lastname,firstname)
  registers a new user. The last two arguments lastname and firstname can be omitted.
- unregisterUser(uid,pw)
  removes the user from the system.

[1]For database users, please consider Section 5.4.1.

- changeUserPassword(uid,oldpw,newpw)
  sets a new password for the specified user.
- changeUserName(uid,pw,lastname,firstname)
  changes the names for the specified user.

5.3.1 Delegation

Every data item in SAFE belongs to a particular user, the *owner*. Only the owner of a data item can modify or delete the data item. There is an exception, however. Users shall be able to delegate permissions among other users. For example, consider a user Alice who wants to give her rights to another user, say, Bob. Alice may delegate authority to Bob, and Bob can then act on behalf of Alice, using the identity Bob speaks for Alice. When a user Alice speaks for a user Bob, then Alice can modify (or delete) any data item owned by Bob. Delegations may be cascaded, e.g., Clara speaks for (Bob speaks for Alice).

Delegation is **transitive**, i.e., if A speaks for B and B speaks for C, then A speaks for C as well. Delegation can be **cyclic**, hence if A speaks for B and B speaks for C, then C can still speak for A. Delegated updates do not change the **ownership** of a data item.

How to set up delegation? Any f-unit with name ending in Auth is allowed to execute the following public CRM methods:

- delegatePrivileges(a,b)
  delegates privileges from a to b.
- revokePrivileges(a,b)
  revoke delegation from a to b.
- getDelegations(a)
  returns a PHP array of delegations for a
  containing the two keys from and to.

5.3.2 Groups and roles

It is often inconvenient to explicitly list all principals that are trusted in some respect (e.g., all principals that may alter goals in a student administration system) both because the list would be long and because it may change too often. Groups provide an indirection mechanism: All the users having some common privileges are placed in a group and the group is given a specific set of privileges. If G is a group, there may be some privileges associated with it, so that a member of a group G can have all the privileges of G. A principal is allowed to have the privileges of G only if he is a member of G. A member A of several groups F, G, H can choose to use the privileges associated with any subset of his groups, say G and H. So, a proper mechanism is required for joining groups, proving membership, delegation of authority, certifying such delegations, and deciding whether a request should be granted.

A group in SAFE is internally represented as a user. The only difference is a special name, i.e., a group name is prefixed with gr: or with something similar. A user A is augmented to a group as soon as at least one other user B is specified to be a member of A.

How to manage groups? Any f-unit with name ending in Auth is allowed to execute the following public CRM methods:

- joinGroup(g,u)
  add user u to group g.
- leaveGroup(g,u)
  remove user u from group g.
- getGroupMembers(g)
  returns a PHP array of the members of group g.
- getGroups(u)
  returns a PHP array of the groups of user u.

5.4 Source files and configuration files

All source files must be encoded using UTF-8. Character encodings should not be annotated or included in any of the source files.

5.4.1 Configuration files

If you have not set up your SAFE system using the SAFE wizard, a configuration file config.crm.php in the root directory of your application is necessary to specify the database access for the CRM. It should contain the following 8 lines of specification. We highly recommend to set all the following values in the beginning of the development process and to keep them unaltered during the lifetime of an application. If you have used the SAFE wizard, please ignore anything below.

```
<?php

  $this->DB_HOST          = "localhost";              # <- adjust this line.
  $this->DB_DATABASE      = "safe";

  $this->DB_USER_SFW      = "safe_sfw";
  $this->DB_USER_SFW_PW = "password1";                 # <- modify this line.

  $this->DB_USER_CRM      = "safe_crm";
  $this->DB_USER_CRM_PW = "password2";                 # <- modify this line.
```

```
   $this->DB_SECRET_HASH_SEED = "SuPeRsEcReT";          # <- modify this line.

   $this->FUNIT_CREDENTIAL_SEED = "credentialseed";     # <- modify this line.

?>
```

Details about the individual values are explained in the next section.

Modes of operation

The CRM can be run in two modes: in the multi-user mode and in the single-user mode. The multi-user mode requires two separate database user accounts, and provides higher security guarantees. The single-user mode is easier to maintain and to set up, but it provides weaker security guarantees.

In the **multi-user configuration**, there are two database users, the SFW user and the CRM user. The SFW user has only restricted access and may only modify *data* of the database. The CRM user should also have permission to modify the *structure* of the database, hence creating tables, views, triggers, constraints, etc. More precisely, the CRM user needs GRANT and also SUPER privileges.

In case your database provider offers only a single database user (instead of two) with only limited privileges (instead of full privileges with GRANT and SUPER), it is possible to set up a **single-user configuration**. All database connections from the CRM are mapped to this single account, no separation of privileges occurs. However, the CRM still enforces access control for f-units and users. In order to run the CRM in the single-user configuration, the two lines for the SFW user must be omitted from the configuration file. We stress that the user configuration cannot be changed during the lifetime of the CRM. In other words: once set up, the CRM configuration must not be changed. !

CRM seeds

Every CRM configuration needs a seed for verifying authenticated application users and enforcing user sandboxes. To this end, the configuration file requests a secret value DB_SECRET_HASH_SEED, which should be kept secret.

The authentication credentials for f-units (☛ Section 2.1) are derived using a deterministic hash function with various input values. One of the input values is the application-wide FUNIT_CREDENTIAL_SEED as specified above. Every web application should keep this seed confidential in order to prevent developers from forging authentication credentials for their f-units.

5.4.2 JavaScript files

Any piece of JS code has to obey the obligations of JavaScript's strict mode (i.e., no global variables, attention with `eval()`, etc.).

- **Inline JavaScript snippets in the `.sfw` file.** These snippets are contained in the SFW source file, enclosed in `<js>` and `</js>` tags. The JavaScript code is inlined right at the position where it has been specified. The compilation by the SFWC compiler turns on JavaScript's *strict mode* by adding the line `"use strict";` at the beginning of each such block.
- **F-unit specific JavaScript code in the `.js` file.** All JavaScript code specified in f-unit FooBar's JavaScript file FUnitFooBar.bundle/funitfoobar.js is collected by the PHP autoloader for f-units in sfw.utils.php. Whenever an f-unit is requested for the first time by a page page.sfw, the corresponding JavaScript code is added to the global JavaScript file page.gen.js. This global file is included at the beginning of every page and is executed in JavaScript's *strict mode*. Again, the line `"use strict";` is prepended.
- **External includes.** JavaScript code can be included from external sources using the SFW command `<js :...>` in the header of any page. We do not recommend this options since it can cause severe security risks.

All JavaScript files are checked with JSHint. The SAFE integrator stores information about the JavaScript verification status, i.e., whether the .js file of an f-unit has successfully been sanitized with JSHint.

5.5 Integration

The integration of f-units, pages, and customizations is described in Section 2.2. This section explains the technical steps below the hood.

5.5.1 Integration of f-units

For the integration of an f-unit FooBar, the following steps are executed.

1. **Generation of the PHP class file.** The PHP class file FUnitFooBar.bundle/funitfoobar.php is generated from the SFW source file FUnitFooBar.bundle/funitfoobar.sfw.

2. **Compilation of the SFW source file**. The SFW specification is compiled using the SFWC compiler to generate the PHP file FUnitFooBar.bundle/funitfoobar.sfw.php. If necessary, also the file FUnitFooBar.bundle/funitfoobar.ajax.php is generated.

3. **Interface.** The interface file FUnitFooBar.bundle/funitfoobar.int is parsed and the corresponding activations and arguments thereof are registered by the CRM:

    - The table sfw_activation_input obtains a new entry for FooBar with its static and dynamic activation arguments.
    - The table sfw_activation obtains new entries for every activation that is specified for FooBar.

4. **Database tables.** The database file FUnitFooBar.bundle/funitfoobar.db is parsed and the corresponding activations and arguments thereof are registered by the CRM. For a comprehensive description of the database tables, please have a look at Section 5.11.

    - Specified tables (local tables, input tables, output tables) are created in the database (e.g., funitfoobar_data, funitfoobar_input, funitfoobar_output, ...).
    - Permissions for the new tables and views are granted. All invariants are expressed with MySQL triggers and the like, e.g., for each local table to verify the given user information and enforce owner invariants. Helper output tables and procedures are created.
    - Input tables and output tables are registered in table sfw_iotables to make them available for the wiring process.
    - Dependencies from FooBar's local tables to its output tables and to its local views are registered in table sfw_tabledeps.
    - Dependencies from relations used by FooBar's table declarations (e.g., in invariants) are registered in table sfw_tabledeps with link to the table in which the relation is used.
    - Cleanup procedures are created in order to deintegrate the f-unit later.
    - The CRM table sfw_integration_funits stores the derived table declarations together with rewind operations for the de-integration. Moreover, the hashes of FooBar's database file, of its interface file, and of its SFW file are stored for the identification of changes upon re-integration.
    - Initial data (☛ page 16) for local tables is stored (if specified).

5. **Directory permissions.** Verify (and correct if necessary) the privileges of the data and the resource directory.

5.5.2 Integration of pages

For the integration of a page page.sfw, the following steps are executed.

- The SFW specification is compiled using the compiler SFWC to generate the PHP file page.sfw.php.
- The interface file page.int is parsed and the activations are registered by the CRM. The CRM stores the activation data in its table sfw_activation.
- The integration process is terminated by an entry in the CRM table sfw_integration_pages, in which a hash of the .sfw file and of the .int file are stored. In an error has occurred before, the integration process is aborted.

5.6 Activation

After a page P and all its f-units have been integrated in the system, i.e., page and f-units are registered at the CRM, P can be called from a client browser. We assume that a client requests P together with a customization c (☛ Chapter 4).

The customization $c = (o, w)$ defines operations o on the activation tree for page P and specifies a wiring w to link different f-units occurring in the activation tree. The wiring is displayed as dashed edges in Figure 5.1. A dashed edge (u, v) means that at least one output table of f-unit u is wired to at least one input table of f-unit v. The customized activation tree (solid edges) together with the wiring edges form the so-called **combined graph** $\mathcal{G}_c$. An edge (u, v) in the combined graph represents a dependency from u to v, i.e., v depends on u as (1) f-unit u activates instances of f-unit v or (2) f-unit u provides wired input to f-unit v.

Note: The feature of specifying operations o on the activation tree is currently disabled. In the following, we hence assume the operations to be the identity function $o = id$. The wiring w, however, can be specified using the wiring tool from the SAFE tool suite.

The combined graph must be cycle-free in order to guarantee a terminating fixed point during activation. Each customization c has at least one topological order on f-units in the resulting cycle-free combined graph. This topological order is referred to as **activation order**, in which order the f-units of the combined graph $\mathcal{G}_c$ are activated to ensure that all dependencies are resolved correctly. One possible activation order for the combined graph in Figure 5.1 is $(\mathsf{A}, \mathsf{E}, \mathsf{G}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{F}, \mathsf{H})$.

Getting back to the browser request for page P under customization c: SAFE automatically ensures that all f-unit instances are activated in the activation order for c. Moreover, the resulting DOM structure of P is exactly as specified in the activation tree for P.

Please consider the extended manual for detailed insights on the steps that are executed at the activation of a page.

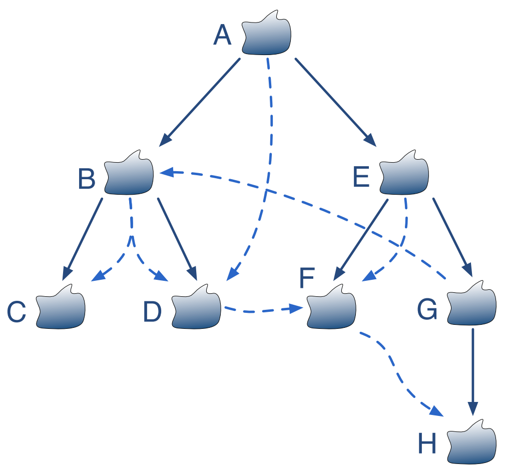


Figure 5.1: Customized activation tree (solid edges) with wiring (dashed edges).

5.7 Database queries via PHP

A database query can be issued by one of the several SFW commands (☛ Chapter 3) or by explicit PHP commands. We recommend not to implement functionality directly using PHP. However, here is how it works.

```
<php>
  $myCRM = new CRM();
  $ret = $myCRM->query("SELECT ... ");
  unset($myCRM);
</php>
```

First, a new CRM connector is established. The SFWC compiler automatically inserts the credentials of the corresponding f-unit as first argument to new CRM();. Second, the query() command issues the specified query against the database. The query is modified by the CRM, e.g., the names of tables are prefixed with the name of the f-unit that issues the query. In other words, database access is restricted to tables that are owned by the corresponding f-unit.

The query() command returns an array with the following keys:

- SFW_DB_RESOURCE, the database resource from which data can be extracted using the PHP functions mysql_fetch_assoc() or mysql_fetch_array().
- SFW_NUM_ROWS, the number of rows returned in the result set in case of a SELECT query. Otherwise, the value is 0.
- SFW_AFFECTED_ROWS, the number of affected rows in case of data updates induced by the specified query. In case of SELECT queries, the value is 0.
- SFW_INSERT_ID, the automatically generated ID of the last insert operation (whenever auto increment is specified on the corresponding table).

- SFW_CLOCK, the value of CRM's logical clock, which has been set by the latest data update query.
- SFW_NEW_CONTENT, a Boolean flag to indicate whether new content is available (1) or not (0).
- SFW_ERROR, a Boolean flag to indicate whether an error has occurred (≥ 1) or not (0).
- SFW_MESSAGE, the error message in the case of a database error.

5.8 Execution of update queries

A query for FooBar is typically specified inside an <sfw:form> tag. The corresponding SFW file funitfoobar.sfw is translated to funitfoobar.sfw.php and to funitfoobar.ajax.php. The first file contains a JavaScript call of sfwSendForm() which sends an AJAX request to the second file.

Figure 5.2 on page 47 illustrates the following 8 steps of a database update.

1. The compiled source code in funitfoobar.sfw.php contains a call of the JavaScript function sfwSendForm(). If the specified check function does not return the string "OK", the update is stopped at this point. Amongst other arguments, the unique identifier of the query to be executed, and the values of the form with the triggered input field are sent to the function body as specified in sfw.utils.js.

2. The JavaScript function sfwSendForm() issues an AJAX request to the AJAX part of the f-unit, funitfoobar.ajax.php. The query identifier and the form values are forwarded to the AJAX file. There, the corresponding query is selected from the set of stored queries. Also, the credentials are stored.

3. The actual query with the form values and the credentials are sent to the CRM. The CRM verifies the credentials, prefixes the query, sanitizes the form values, and ...

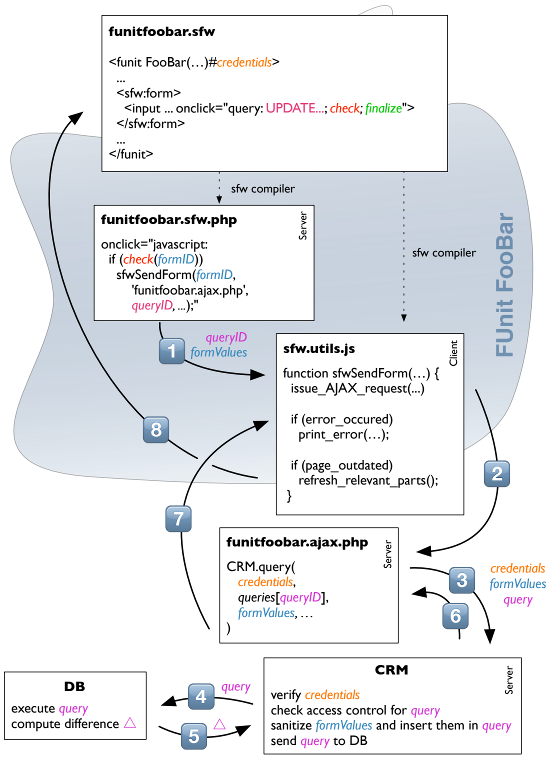
4. ... sends the instantiated query to the database.

5. The database executes the query and returns a status message with a difference δ of modified data items.

6. The CRM reactivates all out-dated f-unit instances (☛ Section 5.8.1). It hands the fresh instances with the query status message back to the AJAX handler.

7. The AJAX handler forwards the status message and the reactivated f-units to the JavaScript function.

8. Outdated f-unit instances in the DOM tree are refreshed by the sfwSendForm() function. In case an database error has occurred, the error message is displayed instead. Finally, the specified finalize function is executed.



funitfoobar.sfw

```
<funit FooBar(…)#credentials>
  …
  <sfw:form>
    <input … onclick="query: UPDATE…; check; finalize">
  </sfw:form>
  …
</funit>
```

funitfoobar.sfw.php

```
onclick="javascript:
 if (check(formID))
   sfwSendForm(formID,
     'funitfoobar.ajax.php',
     queryID, …);"
```

sfw.utils.js

```
function sfwSendForm(…) {
  issue_AJAX_request(…)

  if (error_occured)
    print_error(…);

  if (page_outdated)
    refresh_relevant_parts();
}
```

funitfoobar.ajax.php

```
CRM.query(
  credentials,
  queries[queryID],
  formValues, …
)
```

CRM

```
verify credentials
check access control for query
sanitize formValues and insert them in query
send query to DB
```

DB

```
execute query
compute difference △
```

Figure 5.2: Illustration of a client-triggered, event-driven database update.

5.8.1 Reactivation after database updates

Whenever the client clock c differs from the system clock s, i.e., the difference $d := s - c$ is greater than 0, the CRM method reActivateOutdatedFunits() is executed after the execution of a query q. At this point, it is not relevant whether $d > 0$ because of the query q or because of a query another user has issued. The function generates a data structure for the activation tree and successively fills all nodes of the activation tree.

```
reActivateOutdatedFunits($clientClock, $systemClock, $pageID)
 1. # create data structure for activation tree: $sfwPartialActivationTree
 2. # get outdated f-unit instances from stored activation tree
    getOutdatedFUnits($clientClock,$transactionClock,$pageID)
 3. # find all affected instances from the stored activation tree for $pageID
 4. # add instances for which no ancestor is activated to $sfwPartialActivationTree
 5. # precompute activations and generate missing code (as above)
    precomputeAndTrackActivationsForPage($pageID, $sfwPartialActivationTree, ...)
```

5.9 Execution of refresh queries

The execution of refresh queries is similar to update queries in that a query is executed and, thereafter, outdated f-unit instances are refreshed. However, refresh queries do not alter the database.

Assume the following specification for an explicit refresh in SFW

```
<activate:FooBar() query="SELECT id, RAND() AS data FROM table" refresh='elem.click'/>
```

where elem is the unique ID of an HTML element in the DOM tree. A click on elem triggers the execution of the following steps.

1. The compiled source code in funitfoobar.sfw.php contains a call of the JavaScript function sfwRefresh(). Amongst other arguments, the unique identifier of the query to be executed, and the values of the form with the triggered input field are sent to the function body as specified in sfw.utils.js.

   ```
   sfwRefresh(funit, queryID, refreshWrapper, ajaxUrl, formID, pageID,
                   callerID, staticChildID, totalActPositionsForPage)
   ```

2. The JavaScript function sfwRefresh() issues an AJAX request to the AJAX part of the f-unit, funitfoobar.ajax.php. The query identifier and the form values are forwarded to the AJAX file. There, the corresponding query is selected from the set of stored queries. Also, the credentials are stored.

3. The actual query with the form values and the credentials are sent to the CRM. The CRM verifies the credentials, prefixes the query, sanitizes the form values, and ...

4. ... sends the instantiated query to the database.

5. The database executes the SELECT query and returns a fresh activation dataset.

6. The CRM hands the result set back to the AJAX part.

7. The AJAX part activates one instance per row in the result set. The static activation arguments are taken from the values cached in the database before. The rendered HTML code for the activated instances is sent back to the JavaScript function.

8. Corresponding f-unit instances in the DOM tree are refreshed by the sfwRefresh() function.

5.10 Update dependencies

When investigating which f-units in the activation tree need to be updated after a data update of a certain f-unit instance, we need to consider possible dependencies between

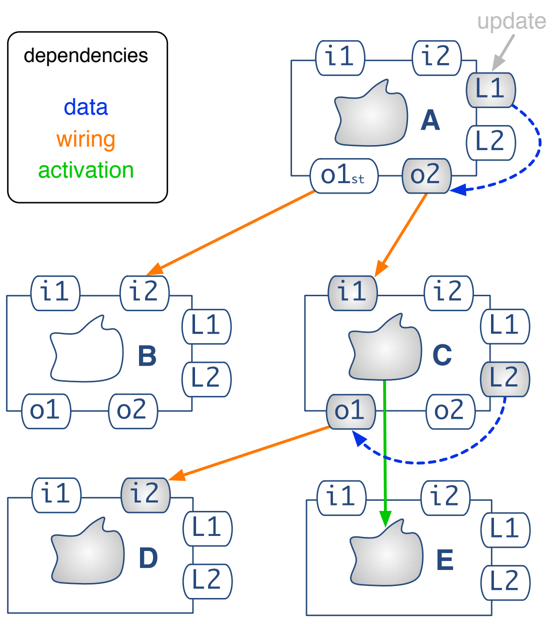


Figure 5.3: Update dependencies: All items colored in gray need to be updated.

f-units, or more precisely between the tables of the f-units. There are three types of direct dependencies: **Data dependencies** come through a direct connection in the database within an f-unit when an output table exposes data stored in a local table. **Wiring dependencies** occur between two f-units when the output table of an f-unit is wired to the input table of another f-unit. **Activation dependencies** exist between two f-units whenever an f-unit activates another f-unit.

Figure 5.3 illustrates the dependencies that need to be considered when an update is executed. More precisely, assume the data in local table A.L1 receives an update. Then clearly, the f-unit A needs to be refreshed since it is likely that A uses its local data.

Data dependency. The output table A.o2 depends on the local table A.L1 and hence might also be affected by the update of A.L1.

Wiring dependency. F-unit C is wired to f-unit A via input table C.i1. Clearly, the f-unit C needs to be refreshed.

While refreshing, C might change its local table C.L2 (for instance to increase a counter). The data dependency to C.o1 causes f-unit D to be refreshed.

Activation dependency. As f-unit C activates f-unit E, the activation arguments sent to E might be taken from local table C.L2 or from input table C.i1. Hence, f-unit E needs to be refreshed.

The only exception is the case for **steady tables**. These tables do not contain data that changes upon activation. However, these cases have to be treated carefully. Whenever a local table, on which the steady output table is based, changes, the output table might also change.

5.11 CRM Tables

Please consider the extended manual for a list of all tables maintained by the CRM.

5.12 Variables and their scopes

Some SFW commands bind values to user-defined variables, e.g. the following (stupid) commands introduce two variables, $$id, $$title .

```
<for query="SELECT gid AS id , name AS title FROM groups">
  //                       ^^             ^^^^^
  // $$id takes different values with each iteration
  // $$title also takes different values

  <queryFirst query="SELECT gid AS id FROM groups WHERE gid='5'">
  //                                 ^^
  // $$id == 5 (with every iteration , i.e., previous values are overwritten)
  // $$title , instead , remains unchanged from above
</for>
```

The command <queryFirst...> overwrites the variable $$id.

Double binding. What happens if there is no such entry with groups.gid==5? Which value is assigned to $$id? In this case, the previous value is used. This setting might cause strange results and might be changed in future versions. Developers can check the number of returned rows in the result set of the last query using the *SQL row counter* $%rows (☛ Section 3.9).

5.12.1 Expansion of special variables and identifiers

This section shows a list of concrete substitution/expansion steps. The context can be one of the following:

Context	Description	Example
reg-sfw	regular SFW context	<h1> $var </h1>
dq-attr	double-quoted SFW tag attribute	<queryFirst query="$var">
dq-attr-q	double-quoted attribute only inside SQL queries	<queryFirst query="$%me">

Name	Context	Expr	Embedding	Expansion
variables:				
PHP var	reg-sfw	$var	<? pr($var); ?>	<? pr($var); ?>
SQL var	reg-sfw	$$var	<? pr($$var); ?>	<? pr($sfw_SQL_ROW['var']); ?>
Stat. arg	reg-sfw	$!arg	<? pr($!arg); ?>	<? pr($this->arg); ?>
Dyn. arg	reg-sfw	$@arg	<? pr($@arg); ?>	<? pr($this->sfwDynActArgs['arg']); ?>
Form elem.	reg-sfw	$#elem	undefined	undefined
constants:				
Clock	reg-sfw	$@$	<? pr($@$); ?>	<? pr($this->sfwDynActArgs['SFW_CLOCK']); ?>
Act. ID	reg-sfw	$@@	<? pr($@@); ?>	<? pr($this->sfwDynActArgs['SFW_ACTIVATION_ID']); ?>
Act. nr	reg-sfw	$%#	<? pr($%#); ?>	<? pr($this->sfwDynActArgs['SFW_ACTIVATION_NR']); ?>
Data dir	reg-sfw	$%data	$%data	FUnitFooBar.bundle/data
Form ID	reg-sfw	$%form	<? pr($%form); ?>	<? pr($sfwLastFormID); ?>
User ID	reg-sfw	$%me	undefined	undefined
Res. dir	reg-sfw	$%res	$%res	FUnitFooBar.bundle/resources
Row count	reg-sfw	$%rows	<? pr($%rows); ?>	<? pr($sfwNumRows); ?>
User hash	reg-sfw	$%uhash	<? pr($%uhash); ?>	<? pr($sfwUserHash); ?>

Name	Context	Expr	Embedding	Expansion
variables:				
PHP var	dq-attr	$var	{$var}	{$var}
SQL var	dq-attr	$$var	{$$var}	{$sfw_SQL_ROW['var']}
Stat. arg	dq-attr	$!arg	{$!arg}	{$this->arg}
Dyn. arg	dq-attr	$@arg	{$@arg}	{$this->sfwDynActArgs['arg']}
Form elem.	dq-attr-q	$#elem	$#elem	dynamically exp. by CRM only inside queries
constants:				
Clock	dq-attr	$@$	{$@$}	{$this->sfwDynActArgs['SFW_CLOCK']}
Act. ID	dq-attr	$@@	{$@@}	{$this->sfwDynActArgs['SFW_ACTIVATION_ID']}
Act. nr	dq-attr	$%#	{$%#}	{$this->sfwDynActArgs['SFW_ACTIVATION_NR'])}
Data dir	dq-attr	$%data	$%data	FUnitFooBar.bundle/data
Form ID	dq-attr	$%form	$%form	$sfwLastFormID
User ID	dq-attr-q	$%me	$%me	dynamically exp. by CRM only inside queries
Res. dir	dq-attr	$%res	$%res	FUnitFooBar.bundle/resources
Row count	dq-attr	$%rows	$%rows	$sfwNumRows
User hash	dq-attr	$%uhash	$%uhash	$sfwUserHash

The notation `<? pr (...); ?>` symbolically represents PHP's `echo` command together with HTML entity conversion and correct encoding, etc.

5.13 Sandboxing CSS

This part is contained in the extended version of the SAFE manual.

Chapter 6

Sample Applications

6.1 Hello World

The hello world application consists of a single f-unit that is activated from the main SFW file hello−world.sfw.

```
<html>

  <head>
    <title> Hello World in SAFE </title>
  </head>

  <body>

    <activate:HelloWorld('Hello')/>

  </body>

</html>
```

The static activation call of f-unit HelloWorld in the body of the main SFW file passes the constant string 'Hello' as static activation argument to the f-unit. The f-unit simply outputs this argument, as specified in the file FUnitHelloWorld.bundle/funithelloworld.sfw:

```
<funit HelloWorld(salutation)#af323ab7a...>

  <h1> $!salutation World! </h1>

</funit>
```

The static activation argument salutation is declared in the headline in the <funit> tag. It can be referenced from inside the f-unit via $!salutation.

Of course, we could have hardcoded the string "Hello World!" in the f-unit itself. But wouldn't that be just too boring?!

6.2 Students list

The second example comprises a dynamic activation (☛ Section 3.4).

```
<html>

  <head>
    <title > Student  List  in  SAFE  </title >
   </head>

  <body>

    <h1> Registered  Students  </h1>

    <table >
      <tr >
        <th>  matriculation  </th>
        <th> name  </th>
        ...
       </tr >

      <activate :StudListTr ()
           query="SELECT  matriculation ,  name,  ...
                        FROM  students
                       WHERE  registered >0"   />
     </table >

   </body>

 </html>
```

After executing the activation query, the *n* returned tuples are used to activate *n* instances of StudListTr. In each instance, the arguments $@matriculation and $@name are bound to the corresponding values. The file FUnitStudListTr.bundle/funitstudlisttr.sfw hence reads:

```
<funit  StudListTr ()# df429ac8c ... >

  <td>  $@matriculation  </td>
  <td> $@name  </td>
  ...

 </funit >
```

The attentive reader might miss <tr> and </tr> tags for the individual rows. The answer lies in the name of the f-unit: instances of f-units ending with Tr, in this case StudListTr, are embedded in wrappers of type <tr>.

6.3 Database updates

We consider the simple specification of HTML buttons that are linked with the execution of database queries. The code snippet in Figure 6.1 shows the corresponding specification in the declarative modeling language SFW. More precisely, the code shown in Figure 6.1 is taken from an f-unit to administrate groups in a social network application. The parts

```
<h1> Groups </h1>
The following groups exist:
<ul>
  <for query="SELECT gid, name, owner='$%me' AS isowner, ismember FROM groups"/>
    <li>
      $$name
      <form>
        <if $$ismember>
          <input type="button" value="Leave"
            onclick="query:DELETE FROM groupmembers WHERE gid='$$gid' AND uid='$%me';"/>
        <else>
          <input type="button" value="Join"
            onclick="query:INSERT INTO groupmembers SET gid='$$gid', uid='$%me';"/>
        </if>

        <if $$isowner>
          <input type="button" value="Remove"
            onclick="query:DELETE FROM groups WHERE gid='$$gid';"/>
        </if>
      </form>
    </li>
  </for>
  <li>
    <form>
      <input type="text" name="gtitle"/>
      <input type="button" value="Create Group"
        onclick="query:INSERT INTO groups VALUES ('$#gtitle',$userID);"/>
    </form>
  </li>
</ul>
```

Figure 6.1: HTML buttons linked to concrete database queries.

colored in blue are SFW-specific deviations from standard HTML. Line 3 introduces a bullet list of groups whose entries are extracted from the SELECT query as specified in the <for> tag in line 4. The query selects four data fields each of which is available in the subsequent scope via the SFW syntax $$field, e.g., $$name in line 6, or $$ismember in the <if> tag in line 8. The content inside the <for> block is "executed" (or more precisely: displayed) in the HTML document once for every result tuple of the query execution. The SFW placeholder $%me in the Boolean comparison of the owner column inside the query specifies the authenticated user of the application. Depending on the values of $$ismember and $$isowner, the buttons to join (line 12), to leave (line 9), and to remove a group (line 17) are displayed. In the following, we will focus on the button to create a new group (line 26).

Instead of specifying a JavaScript function for the onclick event of a button, the developer simply specifies the actual database query to be executed. The query for the button in line 26 inserts two values in the specified table. The first value is the user input as specified in the form. SFW offers the syntactic placeholder $# gtitle to represent the value entered in the HTML input element named gtitle . The second value $userID is a dynamic value which is no direct user input, but a standard PHP variable. All such dynamic values occurring in the query need special attention: dynamic values must also be contained in the corresponding form in the HTML document (in order to ensure the dynamically evaluated values end indeed up in the query). However, such dynamic values cannot simply appear in the DOM tree in plaintext since a malicious client could easily modify these values, for instance by replacing the user ID Alice by the user ID Eve. The integrity and confidentiality of those values have to be ensured by suitable security mechanisms for which technical implementation details make a manual implementation of the overall query execution even more complex and error-prone.

The treatment of dynamic values is just one point on the list of tasks a traditional Web developer would have to take care of. The blow-up factor of roughly 10 after compilation can be justified by looking at the following (incomplete) list of necessary steps for a traditional hand-written update procedure:

1. Create a regular HTML form inside the current HTML document. The protocol (GET, POST, etc.) and the recipient have to be suitably specified.
2. Create a PHP file to answer the submitted form and mention the URI of the file in the form.
3. In the PHP file, authenticate yourself at the database and establish a secure connection.
4. For each variable transmitted via the form, escape special characters to prevent SQL injection attacks.
5. Insert escaped values in the query.
6. Verify that the authenticated user of the Web application has sufficient permission to execute the query with the current values.
7. Verify that the query can be executed in terms of data consistency, i.e., has the query

been issued from a state which is sufficiently fresh?

8. Send query to the database for execution.
9. Process the result and output a status message or refresh parts of the Web application. Here, it will be necessary to determine the parts of the Web application which must be updated. Hence, all relevant dependencies must be derived.
10. Specify event-driven AJAX code to send the form values to the PHP handler, and to receive the data updates for all corresponding elements in the DOM tree.
11. Implement a comprehensive error handling which takes into account all possible kinds of errors (database connection errors, query execution errors, invalid values, etc.). This error handling has to be specified at all tiers; hence in PHP, JavaScript, and possibly also at the database tier.

The database update methodology of SAFE implements a superset of the above steps and thereby significantly reduces the burden of the developer down to a concise declarative specification as simple as the one shown in Figure 6.1. Due to space constraints we omit the compiled code for the example in Figure 6.1. The compiled code consists of a total of 248 lines.

Consistency guarantees. SAFE provides three different consistency guarantees: (1) consistency at the client, (2) at the database level, and (3) for concurrent interactions of multiple users. For the *client consistency*, assume an instance of an f-unit `F` receives an update from a connected client, e.g., an UPDATE query for a table `F.t` is executed. SAFE then automatically refreshes all elements in the DOM tree which are (possibly) outdated. The outdated elements are all those f-unit instances that depend on f-unit `F`, more precisely on the table `F.t`. An f-unit `G` depends on `F` whenever there is a data wiring from `F` to `G`, or if there is a data flow from `F` to `G` while `F` is activating `G`. The SAFE compiler provides all necessary JavaScript code to refresh the DOM elements corresponding to the outdated f-unit instances. The developer does not have to address this tedious and usually error-prone task. Two screenshots showing the refresh without reloading of the page is shown in Figure 6.2.

Besides consistency for dependencies introduced through the wiring and the activation, the SAFE framework also copes with *consistency at the database level*. The specification of tables for an f-unit may mention special dependency types that declare a table to be a subtype of another table. For example, an f-unit to manage comments might be attached to another f-unit managing groups. Such a subtype imposes classical foreign-key constraints with all its implications: a comment can only exit inside a group, i.e., whenever a group is deleted, all corresponding comments must be removed. SAFE guarantees consistency also at this level.

Moreover, SAFE also offers *concurrent consistency* for multiple clients: assume two clients are accessing an application at the same time. Client C_1 updates some pieces of data at time t_1. If client C_2 tries to modify a data item at time $t_2 > t_1$ which has been removed at t_1 before, SAFE detects the invalid request, does not allow the transaction, but instead updates C_2 accordingly.

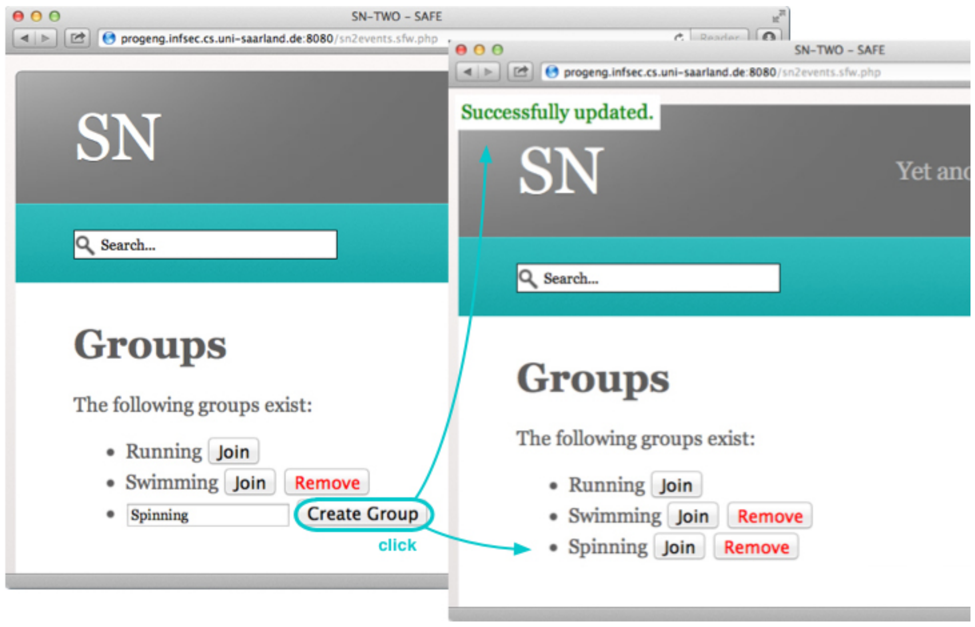


Figure 6.2: Two screenshots: before and after clicking the “Create Group” button.

Chapter 7

Syntax

This chapter provides the syntax for various SAFE components.

7.1 F-units

The **short name** of an f-unit starts with a capital letter and continues with characters from the classes A–Z, a–z, and 0–9. The length is up to 30 characters. An example is FooBar. The **long name** is prefixed with FUnit, for example FUnitFooBar. Usually, the short name is used, e.g., in activations, in the interfaces, for customizations, in the <funit> tag at the beginning of an .sfw file, etc. The long name only occurs in the **f-unit bundle** in the form FUnitFooBar.bundle, i.e., in the directory of the file system in which all files of FooBar are placed. The files have lowercase file names of the form funitfoobar.<EXT>.

The file extensions are the following (all files are discussed in Section 2.1):

Ext.	Type	Reference
.sfw	SFW source file	☛ page 15, and ☛ Chapter 3
.db	database specification	☛ page 15, and ☛ **??**
.int	interface file	☛ page 17
.css	CSS source file	☛ page 17, and ☛ Section 3.7
.js	JavaScript source file	☛ Section 3.6
.php	PHP source file	

7.2 Pages

The file name of a page can contain letters A–Z, a–z, and 0–9. It may contain the dot character '.'. It must not end with a number in front of the .sfw extension.

The following names are thus allowed for a page.

- mypage.sfw
- yourPage.sfw
- my2ndPage.sfw
- my.3rd.Page.sfw

The following names are not allowed:

- invalid extension: page.txt
- invalid number: page2.sfw
- invalid space character: my␣page.sfw
- invalid hyphen and underscore: my-own_page.sfw

Chapter 8

Troubleshooting

8.1 In interaction with SAFE

```
#1146 – Table 'safe.sandbox_assertion_failed' doesn't exist
```

Reason: wrong owner or no owner specified in database update queries.

The error is reported most often by an attempt to update the database. The cause for this problem are insufficient privileges. The query is either lacking an explicit owner, or an existing data set belongs to a different user than the user issuing the update query.

8.2 SFWC

Error SFWC-001 A page tries to issue a query against the database. This is not allowed since pages cannot declare their own database tables. Database queries can hence only be issued by f-units.

Error SFWC-002 Tags or text has been encountered outside the `<body> ... </body>` block or even outside the `<html> ... </html>` block.

Error SFWC-003 A page contains a static activation argument. This is not allowed since pages are never activated.

Error SFWC-004 An f-unit contains an undeclared static activation argument. This is invalid since all static activation arguments must be declared at the beginning of an f-unit (☛ Section 3.4.2).

Error SFWC-005 Invalid filename for page. Every page must obey the syntax for page filenames (☛ Section 7.2).

Error SFWC-006 Invalid filename for f-unit. Every f-unit must obey the syntax for f-unit filenames (☛ Section 7.1). In particular, the name must match with what is specified in the <funit> tag in the beginning of the f-unit. The name of the f-unit is specified right after the funit keyword and must match the name of the bundle folder and its contained filenames (☛ Section 2.1).

Error SFWC-007 An f-unit contains <html> tags. This is invalid since f-units are activated inside of pages. Only pages can contain <html> tags.

Bibliography

[1] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[2] David Heinemeier Hansson. Ruby on Rails, `http://rubyonrails.org`, 2011.

[3] Raphael M. Reischuk, Michael Backes, and Johannes Gehrke. SAFE extensibility for data-driven web applications. In *WWW '12: Proceedings of the 21st International World Wide Web Conference*, Lyon, France, 2012.

[4] Raphael M. Reischuk, Florian Schröder, and Johannes Gehrke. Secure and customizable web development in the safe activation framework (demo paper). In *CCS '13: Proceedings of the 20th ACM Conference on Computer and Communications Security*, Berlin, Germany, 2013.

[5] Microsoft Research. Microsoft's academic conference management service (CMT), `http://cmt.research.microsoft.com/cmt`, 2011.

[6] Suriya Subramanian, Michael W. Hicks, and Kathryn S. McKinley. Dynamic software updates: a vm-centric approach. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2009.

[7] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan J. Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *WWW '07: Proceedings of the 16th International World Wide Web Conference*, pages 341–350, 2007.

[8] Fan Yang, Jayavel Shanmugasundaram, Mirek Riedewald, Johannes Gehrke, and Alan Demers. Hilda: A high-level language for data-driven web applications. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, 2006.

Index