BFH (Bern University of Applied Sciences), CH-2501 Biel, Switzerland

UniBoard Architecture Specification

Version 0.3

Eric Dubuis

September 12, 2013

Revision History

Revision	Date	Author(s)	Description
0.1	August 13, 2013	Eric Dubuis	Initial draft.
0.2	September 10, 2013	Eric Dubuis	Introduced chain of components.
0.3	September 12, 2013	Eric Dubuis	..., removed typos.

Contents

1 Introduction

This document presents the architectural specification of UniBoard, a (rather) universal public bulletin board. Specific to this architecture specification is that it defines the *standalone*, monolithic as well as the *distributed* variant of the public bulletin board.

To achieve this, the following architectural goals have been in mind:

- The same specification for the architecture of the *standalone* and the *distributed* variant of UniBoard.
- Establishment of a *component composition* in order to achieve the desired functionality.
- Plug-in facility to switch from the standalone variant to the distributed one, and vice versa.

Note that the desired properties of UniBoard have been specified elsewhere (Ref. to be provided).

2 Component Composition

In (Ref. to be provided), several properties of a public bulletin board have been identified. We present in this document the specification of a software architecture allowing to implement these properties by *composition* of respective building blocks, the so-called *components*. A *chain of components* is used as the pattern for the composition.

UniBoard consist of one node (for the standalone variant) or a set of nodes (for the distributed variant). For the distributed variant, a specific component is inserted into the chain of components. That component implements the respective agreement protocol by communicating with peer nodes.

Each component implements the same generic service interface. This allows the desired chain of components. (Of course, not every arbitrary combination of components results in a useful configuration.) The final component in this chain is a persistence service allowing to store information, and to retrieve it later. In fact, our architecture follows the *chain of responsibility pattern*. (TODO Ref. to be provided.)

2.1 Chain of Components

In order to achieve upmost flexibility, all components of UniBoard implement the very same *generic* service interface, simply called *Service* (namespaces in text omitted for brevity). That is, each component (indirectly) implements this interface. Furthermore, applications will use the same service interface, too.

The chain of components is achieved by linking a component to the next one, which in turn is linked to another component. Only the last component in the chain, the persistence service, is not linked anymore to another component. Figure 2.1 shows the static structure.

The *Service* interface consists of two general-purpose methods, *post* and *get*:

```
package org.uniboard.core;                                                    1
                                                                              2
public interface Service {                                                    3
    public Response post(Message message) throws UniBoardException;           4
    public Result get(Query query) throws UniBoardException;                  5
}                                                                             6
```

Not only the interface is rather generic, but also the provided and returned data containers *Message*, *Response*, *Query*, and *Result*. The introduction of generic data containers minimizes the coupling: Components depend on the core classes only, but not on specific classes of other components. See (TODO: Provide Ref. to be provided) for a more detailed elaboration on this topic.

The meaning of the data abstractions can be summarized as follows:

- *Message:* Data that originates from a previous component or, ultimately, from an application for a *post* operation, which is passed to the next component. Each receiving component inspects the data and, if data is according to the expected format and content, processes it and ultimately passes it to the next component (except for the final component, the persistence store), perhaps by adding additional information.

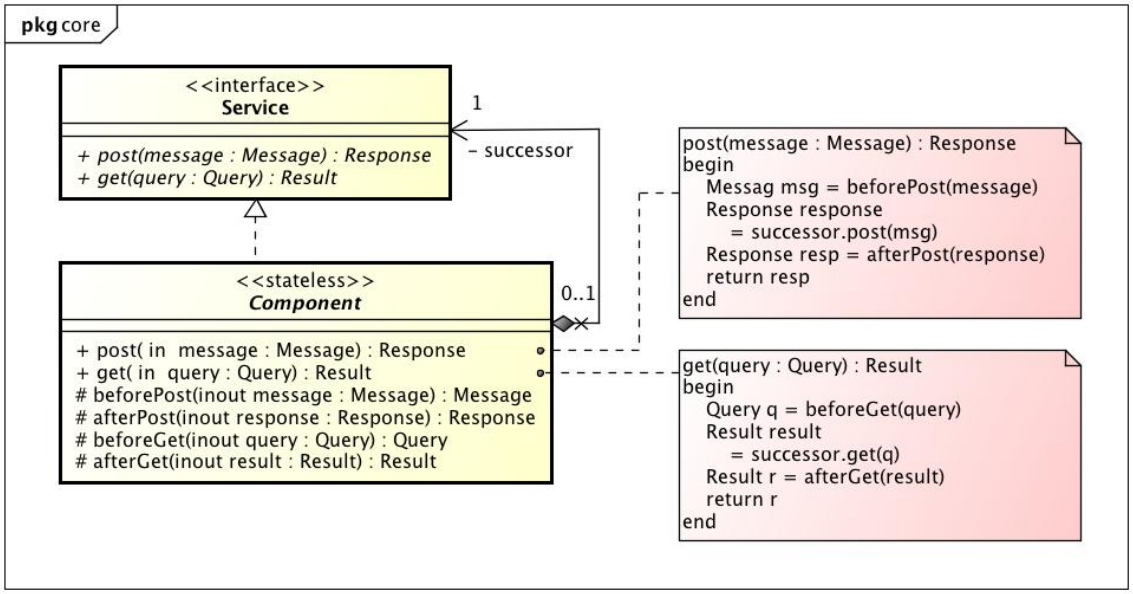

Figure 2.1: UML class diagram defining the interface and linking of the components. Each component implements the same generic interface. The Component class, as shown, is abstract. That implies that concrete components subclass the abstract Component class.

- *Response:* Data that is computed by the component or the downstream component and given back to the calling component or, ultimately, to the application. Each receiving component (or the application) inspects the data and, if data is according to the expected format and content, processes it and ultimately returns it to the calling component or the application, perhaps by adding additional information (except the application).
- *Query:* Data that originates from a previous component or, ultimately, from an application for a *get* operation. Each receiving component inspects the data and, if data is according to the expected format and content, processes it and ultimately passes it to the next component (except for the final component, the persistence store), perhaps by adding additional information.
- *Result:* Data that is computed by the component or the downstream component and given back to the calling component or, ultimately, to the application. Each receiving component (or the application) inspects the data and, if data is according to the expected format and content, processes it and ultimately returns it to the calling component or the application, perhaps by adding additional information (except the application).

2.2 Service Request Delegation

At start-up time, a board is configured by a number of concrete components arranged in a chain of component instances. To service an application, the first component receives a service request and then passes it to the next component. It may also process the service

request before passing it to the next component. Consequently, it may also process the response before returning it to the application.

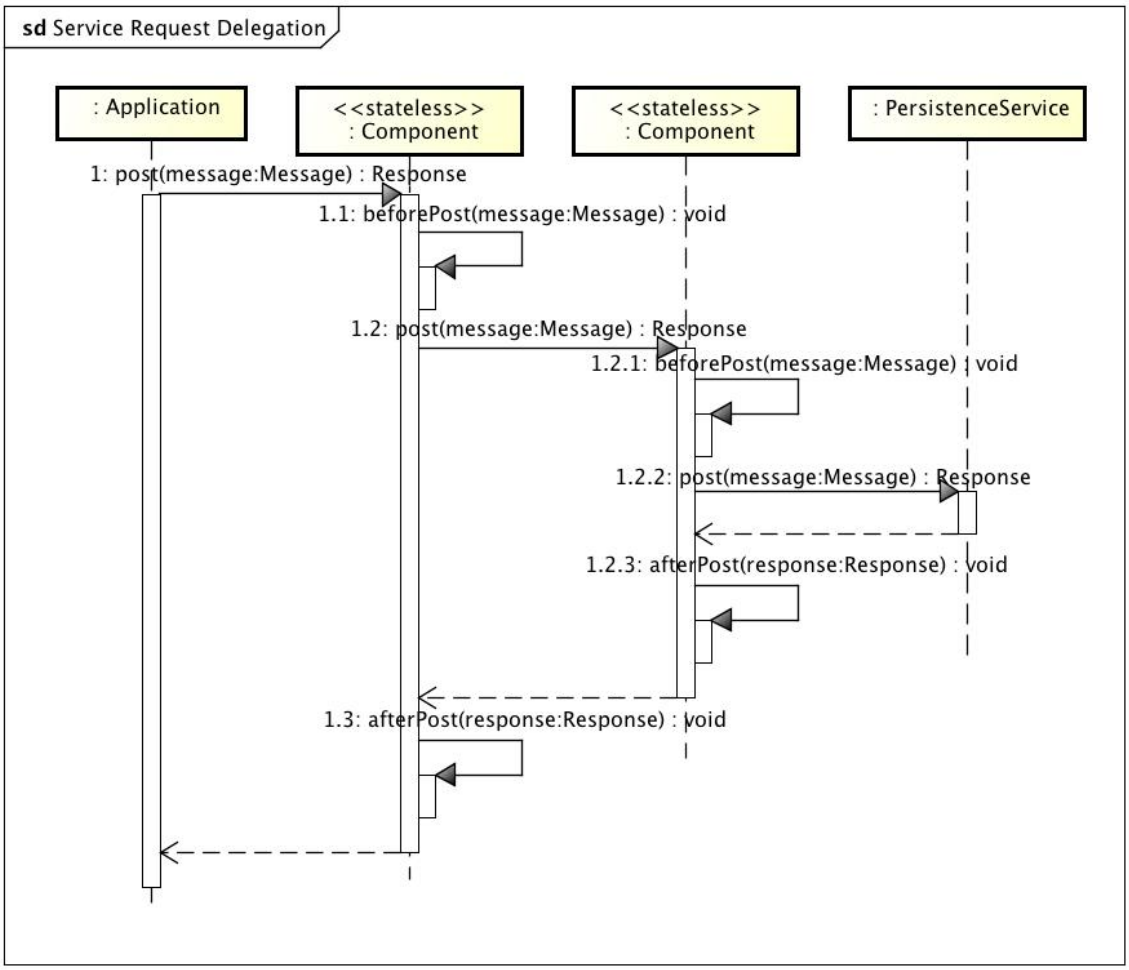

Figure 2.2: UML sequence diagram illustrating the dynamics of the delegation of a service request. Each *post()* request received at a component is processed at first internally by the self-delegated method *beforePost()*, and then passed to the next downstream component. The response of the downstream component is processed at first by the self-delegated method *afterPost()*, and then returned to the calling component (or the application).

The above described behavioral pattern is repeated with every intermediate component. Notice that *synchronous* calls are used between the components. However, asynchronous processing may occur *within* a component, which will likely be the case for the distributed variant of the board.

TODO: Asynchronous calls (and corresponding extensions to the Service interface).

2.3 Handling Service Requests

An application or a component requests a service provided by the downstream component. Basically, the following three processing actions will occur in the downstream component:

1. The received request is processed.
2. The processed request is passed to the downstream component.
3. The result of the downstream component is processed and returned to the calling component (or the application).

The comments of Figure 2.1 sketch the above described processing actions. The template of this behavior (TODO Ref. template pattern, GoF) is predefined in the abstract class Component:

```
package uniboard.core;

public abstract class Component implements Service {
    private Service successor;

    public final Response post(Message message) ... {
        Message beforePost = beforePost(message);
        Response response = successor.post(beforePost);
        Response afterPost = afterPost(response);
        return afterPost;
    }
    public final Result get(Query query) ... {
        // analogous to method post() above
    }
    protected Message beforePost(Message message) ... {
        return message;
    }
    protected Response afterPost(Response response) ... {
        return result;
    }
    // analogous for methods beforeGet() and afterGet()
}
```

Subclasses needing to process a message before passing it to the downstream component merely overwrite method *beforePost()*:

```
package lu.unilu.uniboard.component_x;

import org.uniboard.core.*;

public class Component_X extends Component {
    ...
    protected Message beforePost(Message message) ... {
        // process given message, make new message object
        // (or pass back modified message object, not shown)
        Message processedMessage = ...;
        return processedMessage;
    }
}
```

Similarly, subclasses overwrite methods *afterPost()*, *beforeGet()*, and *afterGet()* in order to implement the desired behavior.

3 Specific Services

In order to implement the desired properties of the (distributed) public bulletin board, a number of concrete services are required:

- *Bulletin Board Service* This is an abstract view of UniBoard. Different type of technologies may be used for implementing this service in order to access UniBoard. However, web service technology (WSDL, SOAP) will be the primary choice.
- *Fault Tolerance Service* This service provides fault tolerance. It may or may not include *Byzantine* fault tolerance.
- *Property Service* This service provides additional properties required by the bulletin board. Its realization is composed of a series of property implementation, all adhering to the very same *Service* interface, exploiting the pluggable layered architecture.
- *Persistence Service* This service provides the view to the persistence storage of the postings sent to UniBoard.

Some details of the *fault tolerance service* will be described in the next chapter. Course views for a *property service* and the *persistence service* are given in the next sections.

TODO: Provide details on accessing the bulletin board.

3.1 A Service Implementing a Property

The public bulletin board requires a number of properties. Our architecture implies that each property is realized within one component. Since a component requires the service of the downstream component, a downstream component is linked with the component automatically; the necessary code is inherited from the subclass *Component*.

Figure 3.1 shows the structure of the authorization service.

3.2 The Persistence Service

The persistence service provides operations to store and retrieve data. It is a component in the sense that it implements the *Service* interface. However, it does not any downstream component and, thus, cannot be derived from the *Component* abstraction.

Figure 3.2 shows the structure of the persistence service component.

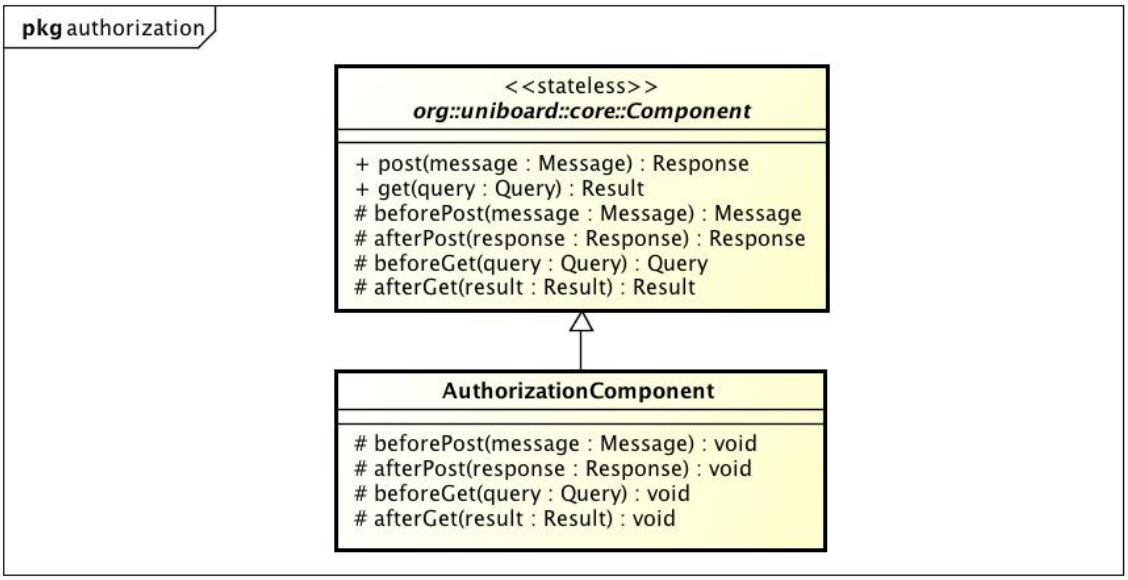

Figure 3.1: UML class diagram for the illustration of the authorization service component. Notice that it is derived from the *Component* class which implies that there is a downstream component link to this component.

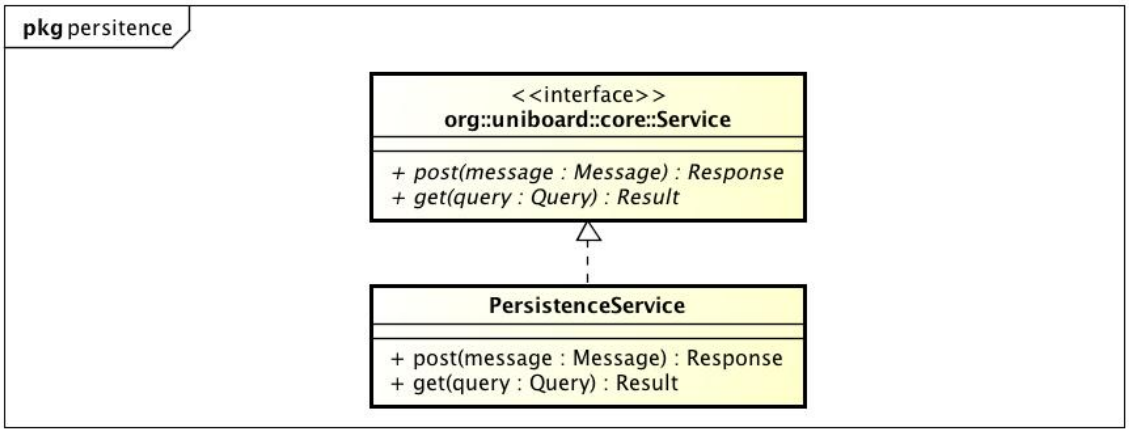

Figure 3.2: UML class diagram for the illustration of the persistence service component. Notice that it merely implements the *Service* interface, and that it is not derived from the *Component* class.

4 Distributed Bulletin Board

This chapters explains how a bulletin board is built as a distributed system. The distributed system consists of a set of nodes. Each node is assumed to be configured accordingly.

The architectural model also assumes that a message to be published is sent to one (trusted) node. (If the application does not trust a given node then it can choose among the other ones.) The message then is processed by the Byzantine fault tolerant client component *BFTClient*. The BFT client component in turn may use downstream components, and it will have to use peers, the so called Byzantine fault tolerant replica components of sort *BFTReplica*.

4.1 BFT Client Component

In its first view, a *BFTClient* acts as a component. That is, it subclasses the generic component class *Component*. As such, it may use the service of the successor (local) component. However, it will also initiate an agreement protocol with peer replica components of sort *BFTReplica*. Figure 4.1 shows the static structure of the Byzantine fault tolerant client component.

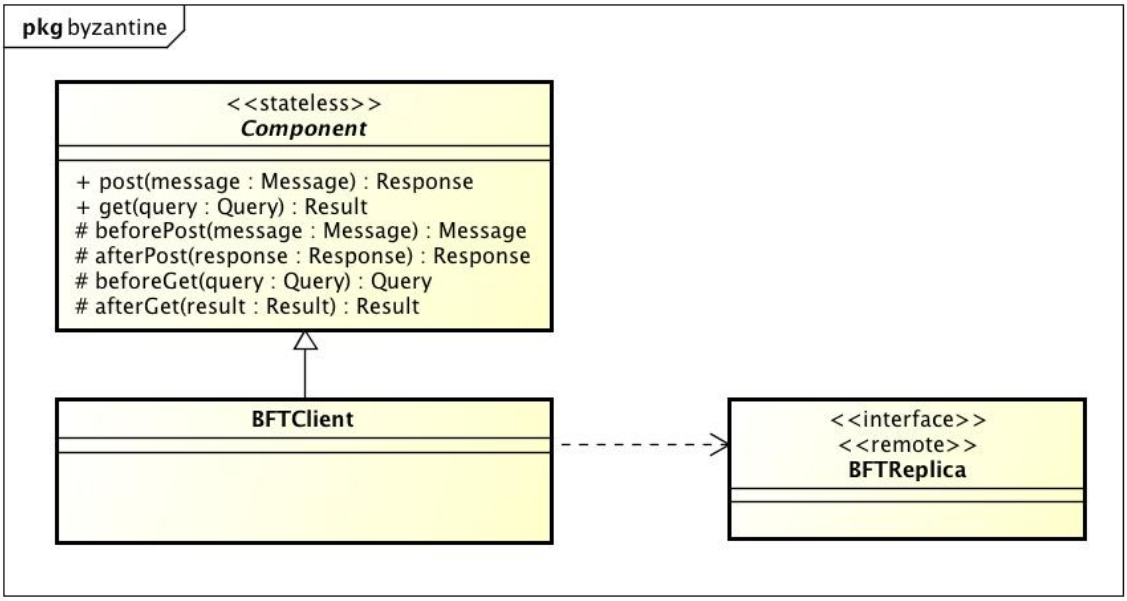

Figure 4.1: UML class diagram illustrating the client side of the replication component. Class *BFTClient* extends the *Component* class. In addition, it uses the remote service accessible via the *BFTReplica* interface.

4.2 BFT Replica Component

Primarily, a Byzantine fault tolerant replica component of sort *BFTReplica* extends the generic *Component* implementation. In addition, it also implements the *BFTReplica* interface. Figure 4.2 shows the static structure of the Byzantine fault tolerant replica component.

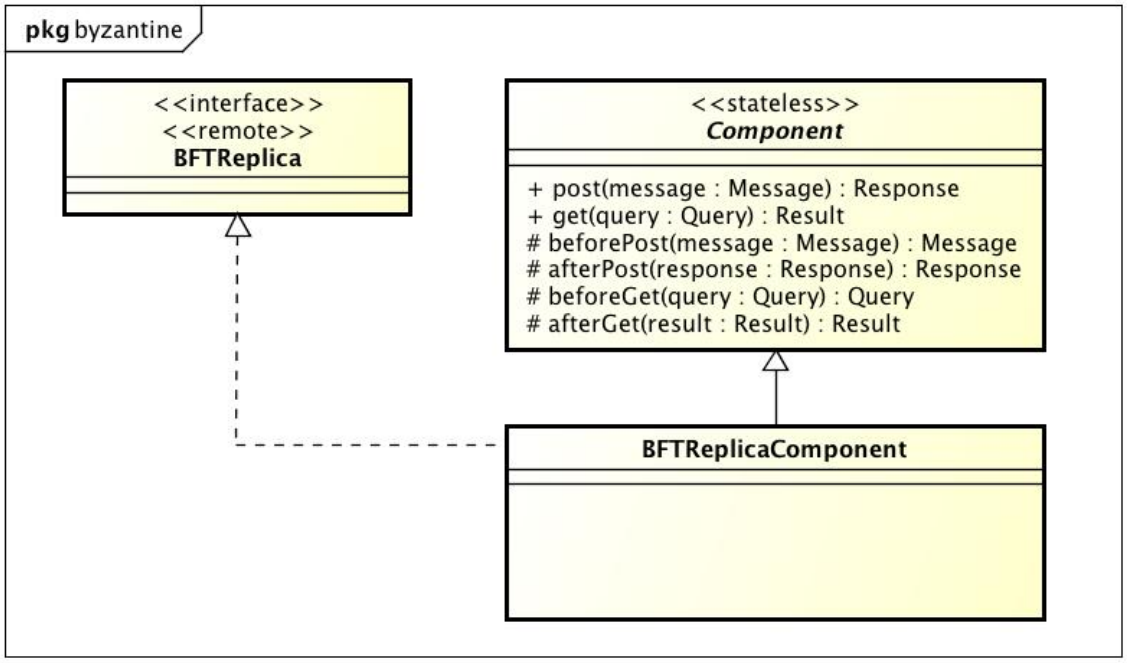

Figure 4.2: UML class diagram illustrating the replication side of the replica component. Class *BFTReplicaComponent* extends the *Component* class (and, thus, is a *Service*). In addition, it implements the *BFTReplica* interface providing specific services to peer replica clients.

The communication between the BFT client and the BFT replicas is an implementation detail and, therefore, not discussed here. (It may involve yet another layered architecture and specialized protocols, and it will use whatever communication technology is appropriate.)

4.3 Deployment View of the Replicated Board

For the replicated configuration, each node is identically configured with a set of required *Component* instances, and with the BFT client and BFT replica component instances.

5 Implementation Hints

Some implementation hints are given here.

5.1 Namespaces, Names

> *Note.* The namespace *org.uniboard* is already used in the domain name system of the Internet. Thus, we will have to discuss its use here.

In order to ease the setup of the code base (project structure, Maven artifacts, names to be defined), certain namespaces must be fixed. The following namespace for the core service abstraction shall be used:

```
package org.uniboard.core;

public interface Service ...
public abstract class Component ...
public abstract class Request ...
public abstract class Response ...
public abstract class Query ...
public abstract class Result ...
```

Implementations will use additional namespaces such as *lu.unilu.uniboard.component1*, *lu.unilu.uniboard.application1*, *ch.bfh.uniboard.application2*, or *ch.bfh.uniboard.component2*.

Class names can also be concretized by following respective technology conventions. When using EJB, the convention for session bean classes is to use the suffix *Bean*.

5.2 Access to the Downstream Component

In order to achieve upper-most flexibility, is is mandatory to access the next downstream component by using the *generic* service interface. For example:

```
package org.uniboard.core;

import javax.ejb.EJB;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;

@Stateless
@LocalBean
public abstract class ComponentBean implements Service {

    @EJB
    private Service successor; // injected by the EJB container

    public Response post(Message message) ... {
        Message beforePost = beforePost(message);
        Response response = successor.post(beforePost);
        Response afterPost = afterPost(response);
```

```
        return afterPost;
    }
    ...
    protected Message beforePost(Message message) ... {
        return message;
    }
    protected Response afterPost(Response response) ... {
        return result;
    }
}
```

Notice: By giving the component class the name *ComponentBean* we follow the propose name convention of EJB (TODO: Ref. to be provided).

The injection performed by the container can be configured at deploy-time. If EAR deployment units are used then the original deployment descriptors or annotations can be overridden upon configuring the EAR deployment unit.

5.3 Skeleton for a Concrete Component

The skeleton of a concrete component looks like:

```
package ch.bfh.uniboard.authorization;

import org.uniboard.core.Service;
import org.uniboard.core.ComponentBean;

@Stateless
@LocalBean
public class AuthorizationServiceBean extends ComponentBean {
    // overwrite beforePost(), afterPost() accordingly
    // overwrite beforeGet(), afterGet() accordingly
}
```

Another example:

```
package lu.unilu.uniboard.byzantine;

import org.uniboard.core.Service;
import org.uniboard.core.ComponentBean;

@Stateless
@LocalBean
public class BFTClientBean extends ComponentBean {
    // overwrite beforePost(), afterPost() accordingly
    // overwrite beforeGet(), afterGet() accordingly
}
```

5.4 A Simple Scheme for Data Containers

Application and components use the generic data containers as described above (TODO Ref. to be provided) for the communication of information to components. This is to keep a high degree of flexibility and to reduce the coupling between components.

The following flexible scheme for the implementation of generic data containers shall be used:

```
package org.uniboard.core;

import java.util.*;
import java.io.Serializable;

public class Message implements Serializable {
    private Map<String, Object> entries = new HashMap<>();
    public void put(String key, Object value) {...}
    public Object get(String key) {...}
}
```

Keys are strings to be agreed among components and application programmers. (TODO: Envision a pre-defined set of keys.) The types of the values are also agreed among components and application programmers. In order to reduce the coupling, basic types and Java system types should be used.

Here is a sketch of code illustrating how components access the information in, for example, a *Message* container:

```
// in a component class
@Override
protected Message beforePost(Message message) ... {
    String id = (String) message.get("IDENTITY");
    byte[] signatureHashValue = (byte[]) message.get("SIGHASH");
    ...
    // Create new message, fill new message m accordingly...
    // Alternatively, the given message object may be updated...
    Message m = new Message();
    // Add information... Then:
    return m;
}
```

The above sketch shows how keys (here *IDENTITY* or *SIGHASH*) can be used, and that applications or components must agree on the types of values (here *String* or *byte[]*).