of the EJB container directly. Instead of that it calls the EJB object through the business interface. The EJB object then invokes either an enterprise bean or a service. The invocation of an enterprise bean is also realized by using an interface. This concept of the EJB architecture ensures automatic persistence, transaction and security for the bean.

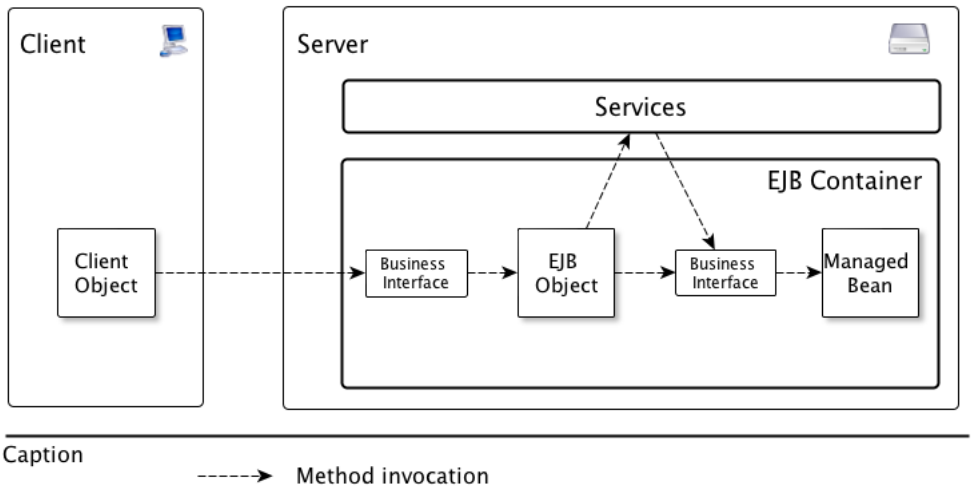


Figure 2.1.: EJB Architecture.

Another task of the EJB container is to provide the required resources for EJB instances. The communication between bean and EJB container can be performed either via callback methods, the EJBContext or the Java Naming and Directory Interface (JNDI). Callback methods are implemented by each bean and used to notify the bean, which took part in the particular lifecycle event. The disadvantage of this mechanism is, that the beans are forced to implement all callback methods even if only one of them is used.

An EJB container provides the service of resource management via the Enterprise Naming Context (ENC). The ENC registers a references of the resources and allocates a unique Name for each registered resource. The access to this names can be gained by using the JNDI-lookup mechanism [18]. Using the EJBContext object the bean can obtain the information of its environment or transaction status [27].

## 2.2. JavaServer Faces

The expansion of the World Wide Web increases the demands for modern web development tools. As a consequence, in the last ten years many web application frameworks have been produced. One of the first attempts to develop Java-based technology for Web applications were the Servlet API and the later emerged

JavaServer Pages (JSP). Both approaches initiated significant improvements in the wold of Web application frameworks and revolutionized web technologies. They provided an object-oriented design and platform independency. Further advancements of JSP are the page-based generation of dynamic HTML code and the support to edit HTML pages using JSP tags [5].

However, the servlet API as well as JSP do not provide easy management of Java code due to the absence of a component-based architecture. Thanks to the component-based architecture, the work with JSF components becomes very convenient.

In order to improve usability, the JSF technology provides a *"component-centric and client-independent development approach"* [5] and enables easy management of application data. The JavaServer Faces framework simplifies the development process of user interfaces and offers effective solutions for the development and maintenance. It also offers different component libraries and interchangeable and extensible framework components like templates and composite-components [26]. JSF extends the Model View Controller (MVC) approach with a component-based user interface and combines the best practices of the technologies described above. The components of JSF are divided into two groups: visual (user interface components) and non-visual. Both JSF component types interact with each other. Visual components serve as a representation for data content. Non-visual components are used for tasks like validation or conversion and therefore provide the possibility of running visual components in the JSF environment.

The term "user interface component" (UIComponent) in JSF denotes a "self-contained" [5] re-usable element for the development of JSF applications. A completed JSF application is a composition of various UIComponents, which includes input text fields, buttons, data grids, etc. UIComponents of a JSF page are combined in form of a component tree or so-called view, whose root is represented by the UIViewRoot component.

### 2.2.1. JavaServer Faces Standard Request-Response Life Cycle

The lifecycle of a JSF application begins with the HTTP request initiated by the client. A HTTP request can be either an **initial** request or a **postback** request. If the HTTP request is performed for the first time, it is an initial request. Postback request means that the client submits the form of a rendered page during the initial request. The lifecycle itself consists of two main phases: *execution* and *rendering*. The execution phase starts with restoring the view as shown in figure 2.2. In the process of the restore view phase the view will be created, the lifecycle events will be handled and the view will be stored in a FacesContext instance. If the request is initial, an empty view will be created in the restore view phase and then the final render response phase will be executed only. The result of the response of an initial request can be, for instance, another

JSF page.

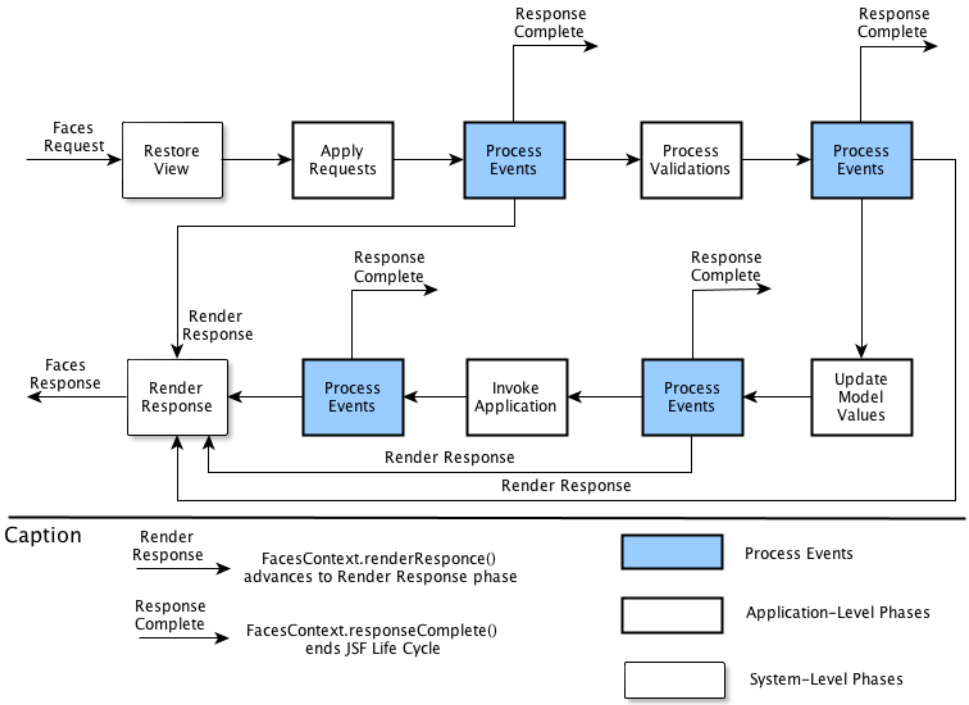


Figure 2.2.: JavaServer Faces Standard Request-Response Life Cycle.

The Response Complete arrows in figure 2.2 redirect the process execution to another resource or web application and therefore cancel the lifecycle process.

If the request is the postback, the view is already created and must be restored during the view restore phase according to the state information of the page. The approach applied to the reconstruction of the page state using Facelets is called partial-state-saving. According to this approach, at first, the new view will be built and then the saved page state will be processed. As a consequence, the size of the page state decreases.

The next sub-phase of the execution phase is the Apply Request Values phase. In this sub-phase all components will be processed and the values, given by the user, will be locally assigned to each component. In the next sub-phase the value data is validated and converted before the corresponding value is written in the data model. It is possible for the JSF developer to interfere in the JSF lifecycle management by setting the *immediate* attribute of a component to true. In this case the validation and conversion of the component value will be performed in the apply request phase directly. As a result of the successful completed apply request phase, the new values for each component will be allocated and the lifecycle events will be processed (see fig. 2.2).

In the process validation sub-phase the local allocated and decoded input val-

ues will be converted to the required form and validated. If the conversion or validation process fails the JSF lifecycle manager adds an error message to the FacesContext instance and starts the render response phase to immediately render the page with an error message. Else the correctly validated local values of components are stored in the server-side object properties now.

If the validation phase was completed without errors, the saved local values are applied to the model in the update model values phase. After the model was updated the application logic is executed in the invoke application phase. This phase deals with the handling of special application events, which are defined by the attributes *action* or *actionListener* of an UIComponent.

As shown in figure 2.2, lifecycle process events are executed after each of the described phases. Here, the transition to the next phase can be aborted, if an error occurs. The lifecycle will either go to the final render response phase or, if the application should be redirected to a certain Web application resources, the response has to be completed (see the Response Complete arrow in fig. 2.2).

The render response phase finishes the lifecycle of a JSF application. During this phase, the component tree will be saved and rendered by using the individual rendering classes of the components.

### 2.2.2. Features and Components of JSF 2.0

JSF in version 2.0 includes a lot of design features, that are implemented according to experience and suggestions from developers of JSF 1.0. Some of them are listed below:

- **System events.** System events enable to react and manage lifecycle events. Event handling is an essential function of the JSF technology. An event object recognizes the component, which initiated the event and saves the event information. Whenever an event is fired by the user, JSF invokes the registered listeners for the event processing. There are three types of events in JSF: *value-change-events* for the handling of input component value changes, *action-events* to handle the control component (buttons and hyperlinks) activations and *phase-events*, which are triggered by the system for the lifecycle processing (the blue boxes in figure 2.2).

- **Standardization of Facelets.** Facelets build a Web template system represented in JSF 2.0 in form of the View Declaration Language (VDL) for the implementation of reusable page components. Basically, Facelets are constructed as a composition structure, which is represented in JSF by UIComponents [20]. Technically this means, that the developer declares one template, which is a simple *xhtml* file. This templates can then be referred to by other pages. Pages that implement a template are called **template clients**. A template client should fill in the non-replaceable

components defined in the template. A page that implements another template, can include its own content, which is defined in the template as replaceable. Facelets also offer the possibility to implement multi-layered hierarchical templates, where the template hierarchy can have an arbitrary depth. The hierarchical templating is used for applications, whose pages have a common layout, while each page has its own page content.

- **Managed beans**. Managed beans are important POJO-based components of JSF for the storage of application data. This architecture of JSF ensures stability and maintainability of the system due to a separation of the view from the domain model [5]. Managed beans implement properties associated with UI components. The object methods and attributes, that are contained in a managed bean, can be accessed via a user-friendly expression language.

  A Java class that represents a managed bean must be registered using the annotation *@ManagedBean* and has to include a public constructor without parameters. JSF instantiates the managed bean object automatically via the Managed Bean Creation Facility after the first access to the managed bean class. The newly created managed bean instance is stored during a certain scope defined by one of the following annotations: *@None-Scoped*, *@Request-Scoped*, *@View-Scoped*, *@Session-Scoped* or *@Application-Scoped*. The Scope defines the duration due to the managed bean object is available. The *@Request-Scoped* has a short duration and provides availability during a single HTTP request, while a managed bean registered by *@Application-Scoped* (the longest scope) exists during the entire lifetime of the application. *@None-Scoped* means that the managed bean object will not be stored and instantiated "on demand" by other managed beans. A managed bean object registered by none scope exists as long as the bean exists, that called it. Note, that managed beans can reference other managed beans with either none scope, an equal scope or with a greater scope.

  JSF 2.0 defines a new additional scope called *@CustomScoped*. If the JSF developer registers a managed bean using the custom scope, he is able to monitor its lifecycle. The custom scope has a longer lifetime than the request scope, but a shorter lifetime than the session scope. The custom scope lifetime lasts as long as the runtime of the faces lifecycle during each request. In this case the managed bean instance will be stored in a map variable of the type *java.util.Map* [5].

  The costs for the accessibility of a managed bean instance are equivalent to the corresponding scope. This means that the request and session scopes are the most expensive regarding to the memory consumption and can easily cause errors that are difficult to debug [26].

- **Ajax Integration.** Ajax is the short form for Asynchronous JavaScript

and XML, which represents a technique that combines JavaScript, the Document Object Model (DOM) and the XMLHttpRequest approach in order to make Web applications more dynamic and responding. Ajax enables to perform asynchronous partial updates of web page components using Ajax engines. If the user initiates an action on a page, he does not communicate with a server through a HTTP request. He interacts with the Ajax engine per JavaScript instead, which processes all operations, which resulted from the interaction of user with the web application. If an operation does not need a connection to the server, the Ajax engine handles it by itself. For operations that require a client-server communication, the Ajax engine directly connects to the server. Therefor, it sends an asynchronous request to the server, so the user interface does not have to be reloaded as a whole [13].

JSF 2.0 provides Ajax functionality in form of the build-in resource library, which can be applied to UI components using the *<f:ajax>* tag to enable Ajax requests. As a result, only the UI components including the *<f:ajax>* tag will be submitted, validated and rendered without performing all lifecycle phases for the whole view [20].

A further possibility to apply the Ajax functionality to a particular UI component is to use the *update* attribute. The value of the *update* attribute contains the ID of the UI component, which has to be updated.

## 2.3. Gargoyle-Codegenerator

In the implementation part of this bachelor thesis the Gargoyle Codegenerator was used to automatically generate basic building blocks of the business logic (EJB source code). The process also included the generation of JSF visualization source code that consists of managed beans and XHTML-pages. The target architecture of the Gargoyle Codegenerator is based on an architecture developed at the LuFGi3 at RWTH Aachen, which is shown in figure 2.3. The architecture is divided into three layers: the client layer, the business layer and the data layer. The client layer represents the user interface, implemented by JavaServer Faces. Methods of the managed bean classes access an application facade in the business layer, which implements the EJB technology. The application facade calls the methods of the CRUD (Create, Retrieve, Update, Delete) classes and includes the corresponding methods for creating, retrieving, updating and deleting of application objects. The application facade and the CRUD Controller implement the business functionality of the whole application [31].

In order to access the persistence layer the methods of the CRUD classes call DAOs. DAO is an abbreviation for data abstraction objects, which represent abstract objects of the data layer. The domain model describes the business objects of the application, which is realized through the DAO methods and the