DigiSlice AppComposer™ Getting Started Guide

DigiSliceDigiSlice AppComposer Getting Started Guide

The software described in this book is furnished under a license agreement and may be used only in accordance with the terms of the agreement.

Copyright NoticeDigiSlice

Copyright © 2003 DigiSlice Corporation.

All Rights Reserved.

This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine- readable form without prior consent in writing from DigiSlice Corporation., 2020 SW 8th Ave., West Linn, OR 97068, USA.

ALL EXAMPLES WITH NAMES, COMPANY NAMES, OR COMPANIES THAT APPEAR IN THIS MANUAL ARE IMAGINARY AND DO NOT REFER TO, OR PORTRAY, IN NAME OR SUBSTANCE, ANY ACTUAL NAMES, COMPANIES, ENTITIES, OR INSTITUTIONS. ANY RESEMBLANCE TO ANY REAL PERSON, COMPANY, ENTITY, OR INSTITUTION IS PURELY COINCIDENTAL.

Every effort has been made to ensure the accuracy of this manual. However, DigiSlice makes no warranties with respect to this documentation and disclaims any implied warranties of merchantability and fitness for a particular purpose. DigiSlice shall not be liable for any errors or for incidental or consequential damages in connection with the furnishing, performance, or use of this manual or the examples herein. The information in this document is subject to change without notice.

Trademarks

AppComposer, DigiSlice, and the DigiSlice logo are U.S. trademarks of DigiSlice Corporation.

Other product names mentioned in this manual may be trademarks or registered trademarks of their respective companies and are the sole property of their respective manufacturers.

C O N T E N T S

1 Building applications with components 7
Introducing DigiSlice AppComposer 8
Making efficient use of limited resources 8
What is a component? 9
What makes a good component? 12
What components do you need? 13
AppComposer architecture 14
JavaBeans 17
Enterprise JavaBeans 18
What's next? 19

2 The ABCs of AppComposer 21
Actors and behaviors 23
Activating behaviors 24
Flow of control 24
Capsules 25
Types of capsules 25
A simple servlet capsule 28
Java expressions 40
Types of behaviors 42
Capsule hierarchy – when does it matter? 42
Running the servlet 44
Debugging 44
"There isn't much code!" 45
What's next? 46

3 Web applications 47

The J2EE application model 50

Browser 52

Web server 52

Java servlets and JSPs 52

Enterprise JavaBeans (EJBs) 53

Application server 53

Databases 54

Separating the business logic from the presentation 54

Application authoring environments 54

Basic web server 55

Java web server 56

Java web server with JSPs 57

Java web server with HTML editor 58

Java web server with AppComposer 59

What's next? 60

4 Building a web application with AppComposer 61

The stages of application development 61

Working with JSPs 63

JSPTemplates 64

JSP request flow 65

Using the JSP repeat tag 67

Using the JSP conditional tag 68

Accessing the SQL database 71

Using the SQLSelect bean 72

Using the SQLUpdate bean 74

Performing multiple updates 75

Sessions: maintaining state between pages 76

Working with EJBs 77

Setting up an EJB for use 77

Entity beans and session beans 79

What next? 80

A AppComposer actors and behaviors 83

Actors that ship with AppComposer 83

Html 83

Sql 84

Misc...84

EJBGen 84

AWT Components 84

Display...85

Types of behaviors 85

Action behaviors 85

Action group behaviors 86

ifTest behaviors 86

Return value behaviors...87

Script behaviors 87

Timeline behaviors...87

Counter behaviors 88

Glossary 89

Index 97

CHAPTER 1

Building applications with components

There's an old story about a traveler who returned to his village in 19th-century Hungary with a story of an amazing device he'd seen in Budapest: a telegraph. Try as he might, the man was unable to explain the device successfully to his neighbors. Wire, electricity, impulses – none of it made any sense to the villagers.

Suddenly, he became inspired: "Forget about the wire," he told them. "Imagine instead a huge dog, with its head in Budapest and its tail right here in Visegrad. When you pull the dog's tail in Visegrad, people can hear it bark in Budapest!"

"I see," said one of the villagers, "but how does telegraphy work?"

"The very same way," he replied, "but without the dog!"

The telegraph was an extraordinary advancement in communications, making it possible to transmit and receive messages much more quickly and efficiently than anyone could have imagined. To anyone who hadn't yet seen the telegraph in action, it was difficult to grasp how something so simple could be so efficient. Where was the horse? The rider? The envelope? The enclosure? With the telegraph, all of these details were hidden in the electrical transmission.

Today, we're frequently faced with a similar challenge: trying to grasp new technological solutions that simplify our work by hiding the details that we've come to take for granted.

Introducing DigiSlice AppComposer

DigiSlice AppComposer™ is a standards-based solution for assembling custom web, business-to-business, and other applications from reusable components.

AppComposer allows non-Java professionals to assemble software components into fully-distributed enterprise applications. Using AppComposer, you can build servlets, applets, and fully distributed applications including n-tier and client-server applications.

If a web application can be written in Java, you can create it in AppComposer. The resulting application will be interoperable with other standards-based products. Once you've assembled an application with AppComposer, you can deploy it on any standard platform, including any Java-enabled web or application server, JSP engine, or database environment.

AppComposer's visual component assembly environment lets you create, test, and modify your distributed enterprise application, using industry-standard components and techniques. Using AppComposer's rapid prototyping approach, you can make changes rapidly, try things out, see what works and what doesn't. You can customize or even add new functionality simply by modifying or adding new components.

Making efficient use of limited resources

AppComposer enables you to rapidly assemble enterprise Java applications from components. Using *behaviors* to define how components will interact with each other, AppComposer reduces the complexity frequently associated with building Java applications with EJBs.

Component-based development allows you to make the best use of the development resources you already have. While Java programmers concentrate on developing JavaBeans and EJBs, non-Java professionals can focus on the task of integrating components together into a coherent, efficient application. This enables Java developers to reduce the time associated with mundane coding tasks.

With AppComposer's intuitive approach, non-Java programmers – including C and C++ developers, Visual Basic programmers, Perl programmers, scripters, and domain experts – can build, customize, and modify server- and client-side Java applications, and even build new, reusable components.

By allowing you to visually assemble components without having to write code by hand, AppComposer insulates you from the complexities of Java, such as classes, name resolution, primitive types, or access control. To integrate components into an application, you don't need to know any of the component's internal details.

At the same time, AppComposer provides you with full access to the Java language when required. AppComposer uses Java as the scripting language for extending and customizing existing components.

What is a component?

DigiSlice AppComposer uses *components* to build applications. Components are reusable software building blocks that work as separate functional units. Components are available for various applications, and you can create new components easily.

In component-based software development, all pieces of an application can be built by assembling, adapting, and "wiring together" existing well-defined components into a variety of configurations. For example, an e-commerce application can be assembled from standard components, including product catalogs, shopping carts, and credit card processing.

The potential productivity gains can be stunning. Reusable components not only make software easier and faster to build, they also make software more reliable, secure, scalable, and easier to maintain. More and more, ground up applications are being replaced or modified by component-based application development.

AppComposer is based on the JavaBeans component architecture. JavaBeans have several crucial advantages over other component architectures.

- JavaBeans components have little or no overhead, which means that it is possible to have smaller components and larger numbers of components without penalty. In other component architectures, there is a large amount of overhead for components, which makes their applications slow and large.
- JavaBeans are platform neutral, which means that they do not care what platform they run on. Note that this is not just a Windows vs. Mac vs. Unix issue. Even for a single brand of computer and operating system, there can be significant platform issues.

AppComposer works with many types of components, including JavaBeans and Enterprise JavaBeans™ (EJBs). This document refers to components that are either JavaBeans or EJBs simply as *beans* when addressing tasks or ideas general to both.

You can use AppComposer with any program that works with JavaBeans or EJBs. Any valid bean you create can be used with AppComposer, no matter how you created it, and any bean you create with AppComposer can be used with other programs that work with beans.

Objects and components

In the past 20 years, object-oriented languages have become popular because they tend to reflect "real world" activities better than traditional procedural programming. Objects are software entities that encapsulate attributes and methods, or stated another way, state and behavior.

Component-based development makes use of elements of the object-oriented approach, such as encapsulation, message sends, and behavior. But component assembly with AppComposer is quite a bit simpler than object-oriented programming. Unlike object-oriented languages, AppComposer is concerned only with instances – real, live objects – and doesn't force you to wrestle with defining classes and subclasses.

JavaBeans are designed as discrete black box components. They don't need to know about each other. All you need to be concerned with are the component interfaces.

In the manufacturing world, a part is reusable because it follows a *standard*. For example, standardized nut, bolt, and screw sizes allow wrenches and screwdrivers from any toolbox to work together.

Software is reusable when it has a well-defined interface to the outside world – that is, to its clients and its environment. Software standards promote interoperability – the ability to combine reusable software from different sources.

Beans follow standards that allow any application server to interpret how to implement the services and functions the beans provide. That means that a developer specifies certain bean properties and any environment which runs the bean interprets these generic properties, so that developers do not need to worry about creating code specific to every platform or environment.

- Applications built with JavaBeans can run on any platform that runs Java, which includes all major web servers.
- Applications built with EJBs require a suitable EJB application server (such as BEA WebLogic) that includes an EJB container. There are many such products in the marketplace; some of them are free.

There is sometimes confusion in the Java world because the base class of the visible user interface widgets in the Abstract Windowing Toolkit (AWT) is named "Component." Even though many visual components in AppComposer are derived from the AWT Component class, the concept of a JavaBean or EJB component in AppComposer should not be confused with the AWT Component class.

What makes a good component?

In good component-based design, the components should be independent. The goal is to be able to extract a single arbitrary component from one application and use it in another application. If you find that you also need to pull out other components (or other objects and data structures) in order to use that component, the component is not particularly well-designed.

A component should also be as functionally complete as possible so that it can perform the specified function without depending on other components or objects. For example, a component to perform a credit card transaction should provide all the functionality required to perform that transaction. The information needed to perform the transaction — the credit card number, transaction amount, etc. — should be passed in as primitive or well-known data types. The

component should validate that the credit card number is valid, connect to the bank's server that will perform the transaction, and perform the transaction. In good component-based design, one component (which may contain multiple objects) performs all of these functions.

What components do you need?

AppComposer is shipped with a set of pre-built reusable components that allow you to construct serious enterprise applications. AppComposer's core library includes components for accessing relational databases, generating HTML, reading and writing XML, handling e-mail, and more. For a list of components that are supplied with AppComposer, see Appendix A.

While AppComposer's components address the core technologies that make up a J2EE application, you'll likely need to develop or purchase JavaBeans, EJBs, and other components that address specific needs in your industry or functional domain. These components are available from a rapidly growing list of third-party vendors.

AppComposer architecture

As shown in the following pages, DigiSlice AppComposer includes a set of integrated technologies:

- JavaBeans and EJBs. AppComposer's primary component technologies are Java standards.
- JBoss, an open-source EJB server. Of course AppComposer can work with an EJBs deployed onto any application server, and you can deploy your AppComposer project to any application server.
- Hypersonic SQL (HSQL), a full-featured relational database written entirely in Java.
- Tomcat, an open-source JSP engine that includes the Jasper JSP-to-Java compiler.
- Debug Web Server. Supports execution of servlets and JSPs in addition to serving static HTML pages during development.
- An integrated debugger that allows you to set breakpoints and watch variables, and perform step-by-step execution.

Architecture

JBOSS EJB SERVER

HSQL Database

Application Composer

Debug Server

JSP -> Java Compiler

Authoring Environment (editor)

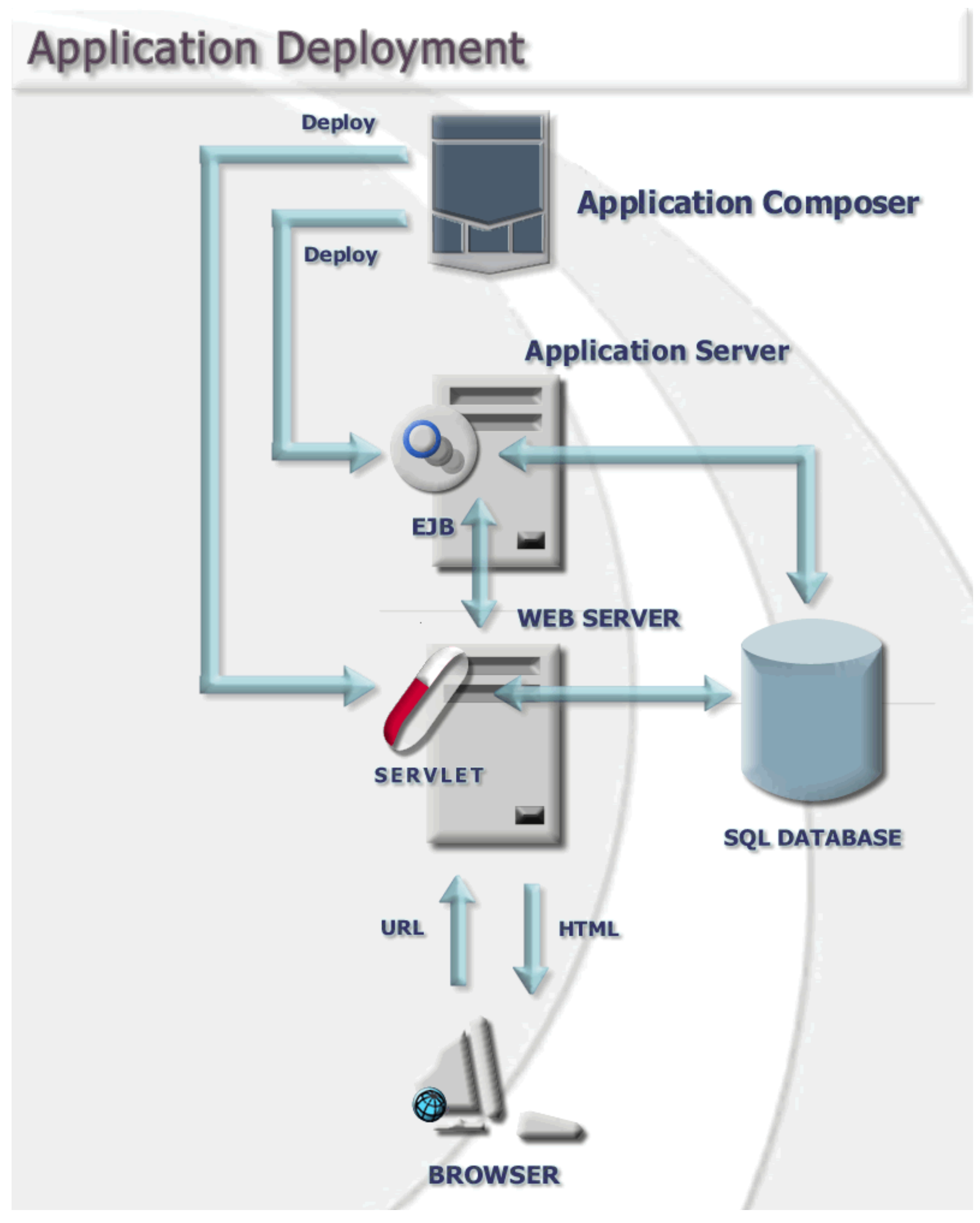
Application Composer Runtime Library
(Internal Components)

EJB Components

Java Compiler

JavaBean Components

DEPLOY



Application Deployment
Deploy
Application Composer
Deploy
Application Server
EJB
WEB SERVER
SERVLET
SQL DATABASE
URL
HTML
BROWSER

JavaBeans

A JavaBean is a packaged Java class with the following qualities:

- It has properties with values that can be set and retrieved.

  Properties are the qualities of a component that you can modify. Properties have values: the color of a background, the size or location of a window, the identification number or balance of a checking account.

- It can generate events.

  Many JavaBeans, particularly those that represent user interface components, generate events. An event is any occurrence of interest to an application, such as a mouse click, a key press, or the opening of a file. A timer can generate an event according to a schedule. A bean can generate any number of events, or none at all.

  Events use the Java two-part naming convention: an event class followed by the name of the specific event, with a dot as a separator. For example, a mouse click is `Mouse.mouseClicked`. The event class is `Mouse` and the specific event is `mouseClicked`.

- It can execute *methods*.

  All JavaBeans have methods. A *method* contains a procedure that executes when the method is invoked. Invoking a method typically causes the component to do something.

  JavaBeans contain two standard methods for each of their properties. Each property must have a `get` and a `set` method. For example, an animated object could use the two methods `setAnimationRate` and `getAnimationRate` to allow you to edit

the current value of the property `animationRate`. AppComposer creates these methods for you and lets you view or edit a component's properties without having to write these methods manually.

Recall that AppComposer uses behaviors to execute methods. Since AppComposer is running your application while you build it, you can easily drop a behavior onto a bean and use it to execute a method on that bean.

- It is serializable, including its state, for storage or for transmission over a network. State might include, for instance, the value of all of its properties.
- It is packaged as a jar file with supporting classes and resources, complete and ready for use in any number of applications.

AppComposer includes many JavaBeans. See the *AppComposer*/`beans` directory for some examples.

For more information on JavaBeans, see:

`http://java.sun.com/products/javabeans`

Enterprise JavaBeans

Enterprise JavaBeans provide cross-platform integration with enterprise services, including:

- Databases
- Distributed transactions
- Security
- Messaging
- Mission-critical robustness and scalability

Enterprise JavaBeans make use of standardized conventions, which any EJB-enabled web application server can interpret, to access these services. These are similar to the properties used by a JavaBean. EJBs benefit the developer because it moves the task of

interpreting these standard interfaces to the application server. The developer no longer needs to worry about writing specific code for each server or environment where the application might be deployed.

For more information about Enterprise JavaBeans, see:

`http://java.sun.com/products/ejb`

What's next?

In the next chapter, we'll look at the ABCs of AppComposer: actors, behaviors, and capsules.

In subsequent chapters, we'll examine the steps involved in building a web application, with special attention to AppComposer's role in this process.

CHAPTER

The ABCs of AppComposer

Now that we've explored the notion of components, let's step away for a minute to consider an industry that couldn't exist without using components: a pizza chain that specializes in home delivery. All shops in the chain use the same basic process.

When a customer calls in an order, the front desk person finds out what items to put on the pizza, how soon the customer wants the pizza, where to deliver the pizza, and any other special instructions.

Once he hangs up the phone, he passes the order to the crew behind the counter, and they go to work. The dough is stretched, the sauce is ladled onto the dough, the cheese is spread onto the sauce, and the various meats and vegetables are evenly placed onto the pizza. The pie goes into the oven, and when it comes out, the crew slices it, boxes it, places it in an insulated bag, and sends it off in the delivery vehicle. All this in less than an hour?

To meet the customer's needs, the members of the pizza crew don't need to know how to cook tomato sauce, dice garlic, prepare the dough, or do any of the other detailed tasks. The dough has been prepared in advance; the tomato sauce is already cooked; the mozzarella cheese has been grated; vegetables and meats are already cut to size and kept fresh in canisters; and the oven is pre-heated. All the crew needs to do is assemble the prepared components into the final product – a hot pizza.

We can summarize the process of building a pizza as follows:

1. Prepare dough
2. Spread tomato sauce on dough
3. Spread cheese on pizza
4. Spread pepperoni and other components on pizza
5. Bake pizza
6. Remove from oven

To continue with our metaphor, the pizza includes a set of components:

- Dough
- Tomato sauce
- Grated cheese
- Sliced pepperoni
- Sliced vegetables

Each of these ingredients is prepared before the pizza is started, and is derived from more basic ingredients: flour and water (to make the dough), tomatoes, olive oil, and spices (to make the tomato sauce), milk, whey, and rennet (to make the cheese) – and we dare not ask what goes into the pepperoni!

The customer doesn't care about how these "components" were built, or how they're structured internally. She doesn't need to know how the olives were pressed to produce the oil, and she's not interested in the irrigation capabilities of some tomato patch. She definitely doesn't want to know how the sausage factory created the pepperoni. All she knows is that her family is hungry, and they want pizza!

A neighborhood pizza shop, thoroughly committed to a homemade approach, might prepare most or all of these ingredients from scratch. Yes, the pizza might taste better – but the shop would need a much larger staff to serve the same number of customers as the chain store, and each employee would need to know a lot more about how to prepare a pizza.

Because the pizza shop uses components, it can handle far more business with a limited number of expert pizza makers.

Actors and behaviors

DigiSlice AppComposer combines component architecture with an authoring technology based on *behaviors*. Behavior-based authoring systems create functionality by allowing you to assign behaviors to *actors*. You can think of behaviors as the "glue" code used by AppComposer to connect components.

A *behavior* causes an actor to do something, enables communication between actors, or responds to user input. Every behavior has a stimulus and a response:

- The stimulus for a behavior is always an event. When the proper event arrives, it activates the behavior.
- How the behavior responds when activated depends on the kind of behavior. The behavior might call a method on an actor, set the value of a property, check whether a condition is true, or generate another event.

Going back to our neighborhood pizza shop, the front desk person who answers the phone is an actor. That actor has a predefined behavior: "Begin preparing pizza." The event, or stimulus, that triggers this behavior is, of course, a phone call from a hungry customer. The response: send the order to the pizza preparation crew.

In AppComposer, actors can include graphic elements that are visible on the display, such as a button or an animation. Actors also can represent nonvisual components, such as a component that reads a file or generates HTML. When an actor is visible on the display, it's called a *visual actor*. AppComposer comes with a set of predefined actors, some visual and some not

AppComposer also comes with several different kinds of predefined behaviors. You can also assemble new behaviors out of existing behaviors and save them for later reuse. This is one of AppComposer's most time-saving features. You can build behaviors for your common tasks and then insert them whenever necessary by using a menu choice. You can then edit any instance of a behavior if you need to customize it.

Activating behaviors

When a behavior receives the event which triggers it and it begins to run, we say it is activated. You can specify any number of events that activate a behavior. When you specify more than one stimulus, the behavior activates if it receives any one of the specified events. For each event, you can also specify a condition that the behavior needs to evaluate to see if it should trigger on that event. When a behavior receives such an event, it evaluates the condition and activates only if the result is true.

Certain kinds of behaviors, once activated, continue to execute for some period of time. These behaviors also allow you to specify an event, along with an optional condition to evaluate, that deactivates the behavior. When a behavior is *deactivated*, it stops executing.

Flow of control

In a traditional programming environment, you might think about describing your application in terms of controls such as program counters. With AppComposer, it makes more sense to describe your application in terms of event flows: which components do you need, and how do they interact with each other?

With AppComposer, events are asynchronous – they don't necessary happen in any particular order. For example, you can define how your application responds when a user clicks a button in the web browser. The user can sit there staring at the browser for any length of time. But as soon as the user clicks a button, that event (the button click) triggers any behaviors that have been listening for it. The behaviors don't know in advance when they'll receive an event. Their job is to "stand on guard."

Capsules

Some visual programming systems make it very easy to build small, toy applications, but do not scale well if you need to build real, complex applications. To solve this problem, AppComposer organizes applications into *capsules*. As indicated by the name, capsules are a visual metaphor for the encapsulation techniques employed by object-oriented languages.

A capsule is a container that keeps together all of the application's business logic – actors, behaviors, data variables, and even other capsules. When you create a capsule, you are creating a new component (a JavaBean™) that you can use, like any component, as an actor in other capsules.

Using capsules has several advantages:

- A capsule, like any component, can be reused in other applications.
- Capsules organize your work, allowing you to focus on one piece at a time or distribute work among developers.
- A capsule has a well-defined interface that provides modularity. As long as the external interface and behavior of a capsule does not change, you can change the internal implementation of a capsule without affecting other parts of your application that use that capsule.

The capsule is deployable in the sense that you can run it on your application server. To manage complexity as your application evolves, you can insert further components into your capsule.

Types of capsules

You can use AppComposer to create different types of capsules:

- **Application** capsules create stand-alone applications or the client portion of client-server applications. Application capsules usually include their own user interface, which you build using standard Java user interface widgets.

- **Servlet** capsules run on a web server and are the primary building blocks of web applications. Servlets typically implement a thin-client user interface in HTML, which uses a browser in place of a custom-built front-end. Servlet capsules can use AppComposer's built-in local web server for testing and debugging.
- You can use **applet** capsules to create applets that run in a client machine through a browser. Note that Java compatibility and security is still an issue in major web browsers, so applets are really usable only in intranets or other controlled environments.
- **JSP bean** capsules create Java Server Pages (JSPs) that include AppComposer-built logic. JSPs are web pages that contain Java calls embedded within HTML files to provide dynamic output.
- A **visual actor** capsule denotes a component derived from the Java class `Component` that you can use as an actor in other capsules. Visual actors are not necessarily fully deployable on their own.

- A **non-visual actor** capsule denotes a component with no visible runtime representation that can be used as an actor in other capsules. Non-visual actors are not necessarily fully deployable on their own.
- **Stateless Session EJB** capsules provide the framework necessary for this type of bean. Stateless session beans can be used for single session interactions. By definition you cannot use this type of bean to save information to use across multiple sessions.
- **Stateful Session EJB** capsules do allow you to save session properties (state) for reuse across multiple sessions or transactions.
- **Stateful Session Synchronization EJB** This is a stateful session bean that gets notification of transaction boundaries. The possible boundaries are:
  - `afterBegin()`: right after a transaction starts
  - `beforeCompletion()`: after everything is complete that was part of the transaction, and before it is committed.
  - `afterCompletion(boolean committed)`: called after the transaction is completed. The boolean that is passed is `true` if the transaction was successfully committed, and `false` if it did not.
- **Message-driven bean** capsules support the EJB 2.0 JMS components. This is a limited-support beta feature for this release.

A simple servlet capsule

To get a better look at how capsules behave, let's look at how AppComposer performs the most basic programming example of all: displaying the message "Hello, world!" in the user's browser. We'll make this example a bit more interesting by including the current time as part of the message.

First, we need to create a servlet capsule with three actors: an HTMLDocument, an HTMLText, and a Clock. All three of these actors are included with AppComposer as part of its basic component library.

Java servlets

A servlet is a Server-side Java class that can be invoked and executed. A Java servlet encapsulates server-side presentation logic in a web application. Java servlets execute using an application server or web server extensions, accessing server resources and applications as needed.

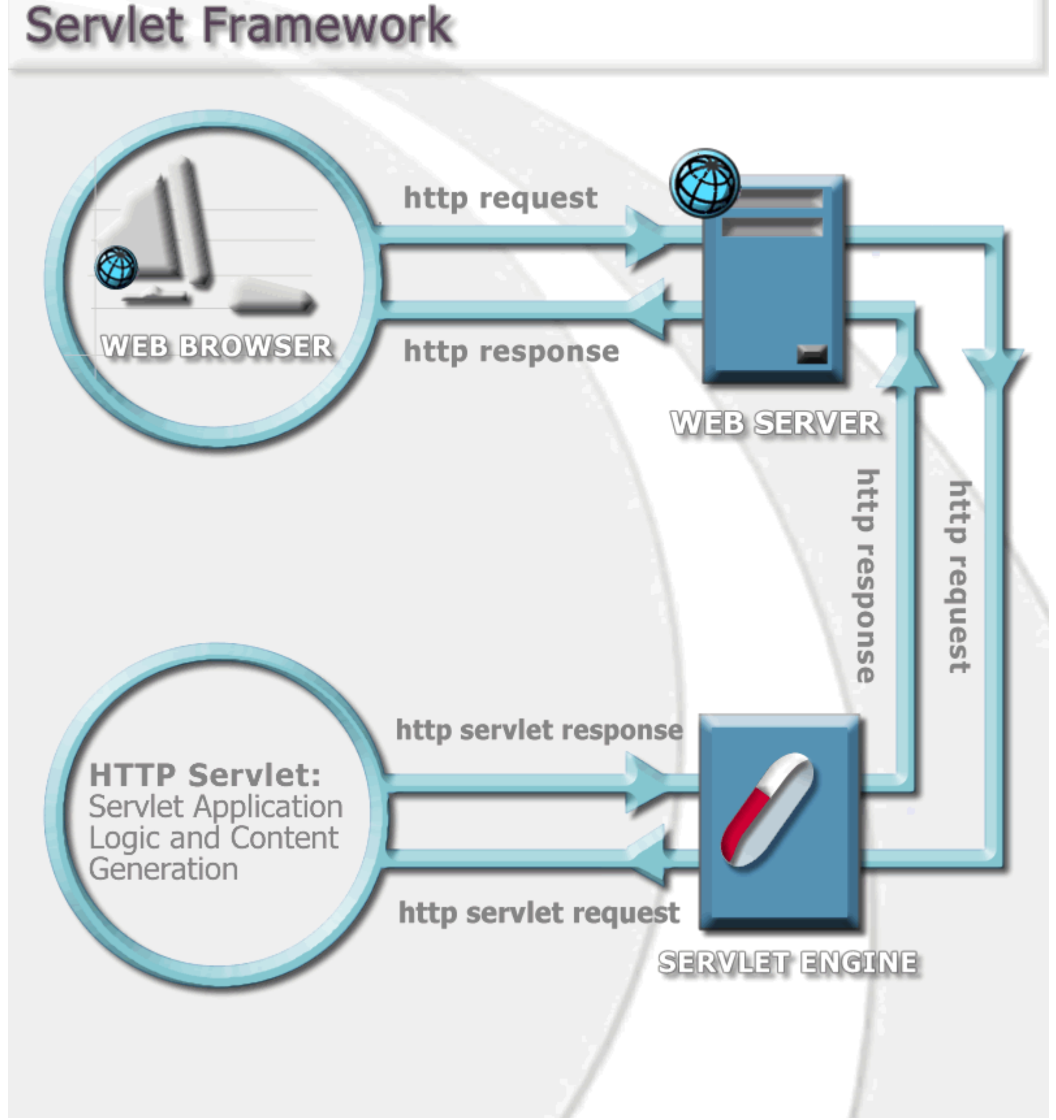
An HTTP Servlet handles HTTP requests and delivers HTTP responses. The servlet contains input and output objects that represent the request and response, respectively. The servlet parses its input, including any parameters.

The servlet uses the output object to respond to the request, and can output HTML. By using servlets, an application can produce very dynamic HTML.

We'll also need to insert two action behaviors into the capsule: one to print the HTMLDocument, and a second to obtain the current time from the Clock.

As we've seen, actors generate events, which can trigger actions and other kinds of behaviors. The action behavior transforms the event into a method execution on the component bean. In our simple example, we'll look at the kinds of events that each actor can generate, and the actions that respond to those stimuli.

Let's start with the simplest case: one component (the capsule itself), one event, one behavior, one method. In the browser, the user clicks a link to a new web page. The URL is sent to the web server, which sends a GET request to the servlet engine. The doGet request is passed along to the servlet capsule, which writes the HTML text into its output stream, which is sent to the browser.



The action behavior calls a simple method:

```
Action (From:capsule Receive:servletRequest.doGet)
    capsule.print ("Hello, world!")
```

Although this syntax might be unfamiliar, the action behavior is really quite simple. When the servlet capsule receives the doGet request, that event triggers the corresponding action: a message to the capsule, telling it to print the text “Hello, world!”. Because the method “print” is already defined for AppComposer capsules, the capsule knows exactly what to do.

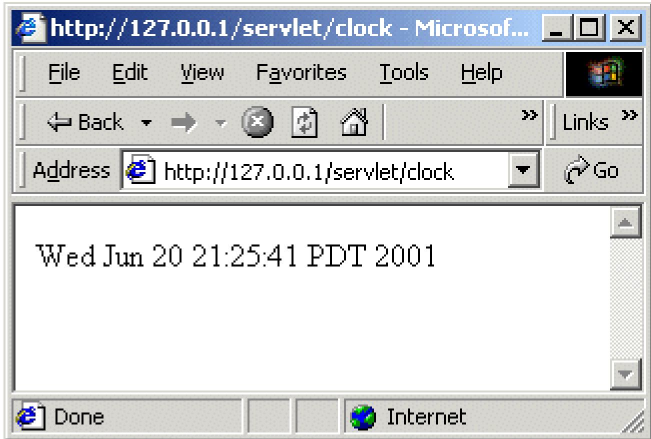
Action behaviors

An *action behavior* is the most common kind of behavior in AppComposer. An action behavior invokes a method on an actor. You can set up an action behavior to execute on any actor you designate. The editor for an action behavior has various fields that specify which method the action uses, which actor in the capsule executes the method, and any parameters the method needs to run.

Although AppComposer makes it easy to use methods to build behaviors, it is important to be familiar with what data a method needs, and how it specifies the value of that data. A single method name may offer several ways of specifying additional data. For example, a method that requires a color as an argument might allow you to specify that color as an object of type `Color` using three integers that represent the red, green, and blue components of the color. Alternatively, you might be able to specify a string, such as “orange” or use a pulldown menu to select the color.

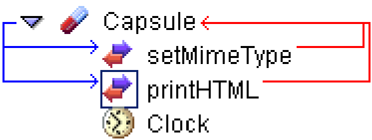
AppComposer allows you to define each required argument using either a constant or an evaluated expression. One of AppComposerDigiSlice’s powerful features is that it allows you to specify an evaluated argument using any Java expression. This expression can use values supplied by other actors, call Java functions, and effectively do anything you can program in Java.

Now let's look at a slightly more complex example, with two components: the capsule (as before) and the clock. As before, there's one event, one behavior, and one method, but this time, the action has an argument: evaluate the current time (from the Clock).



With AppComposer, you can use components to generate the HTML for you. HTML components make your application more portable – and less cryptic. When you build a document from components, you don't need to worry about the HTML syntax. AppComposer's built-in HTML components take care of those details for you.

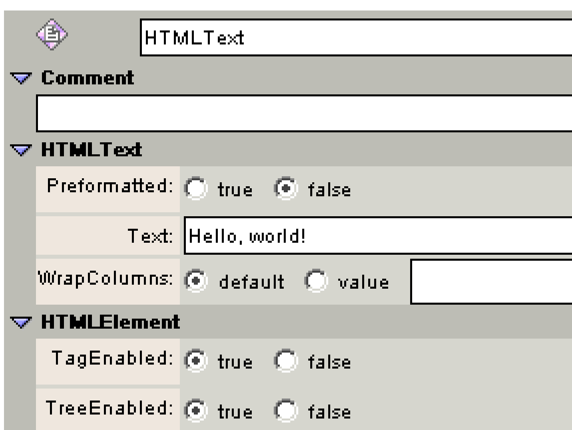
1 First, insert a SetMimeType action behavior so that the browser knows the type of response to expect – in this example, HTML



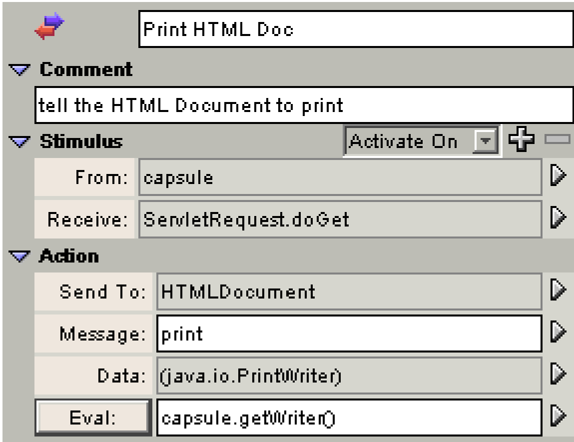
You'll find it helpful to use the Show Activation Events and Show Action Targets buttons to visually trace the connections between the behaviors and actors in your capsule. The blue arrows represent the events that trigger action behaviors. The red arrows represent message sends from one component to another.

2 Insert an HTMLDocument and change its title to "Time."

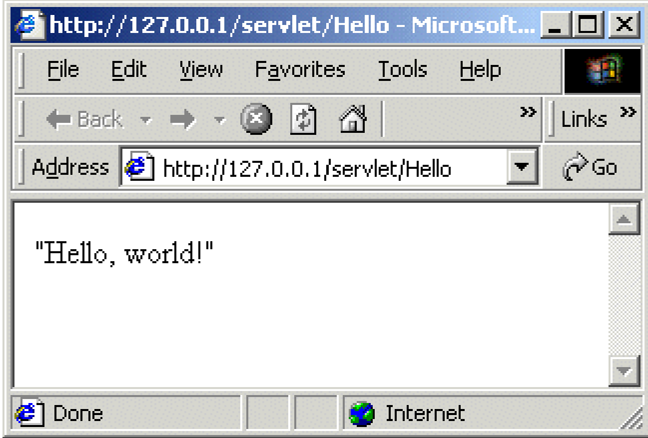
3 Insert an HTMLText component as a "child" of the document, and change its text to "Hello, world!"



4 In the simple example, the action sent a message telling the capsule to print the text “Hello, world!”. Now that we’ve defined an HTML document, we need to change the action to tell the HTML document, rather than the capsule, to print.



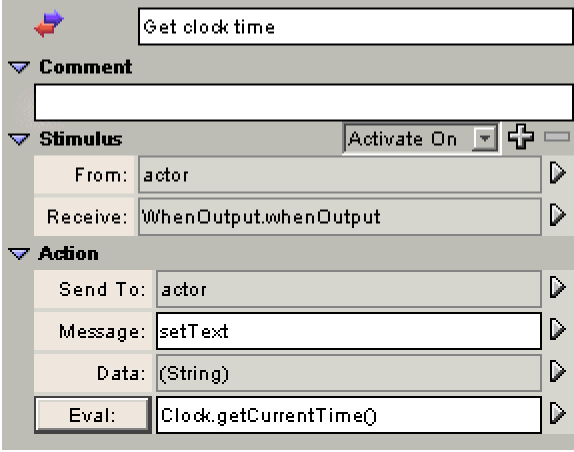
5 The browser output is what you’d expect:



Thus far, we've only been concerned with printing raw text. Now let's define a second action behavior to set the Text property with the clock's current time. Before the HTMLText component is output, it triggers the event "whenOutput." The new action behavior is listening for this event, and when it takes place, the behavior evaluates the expression "clock.getCurrentTime" and places the result into the Text component's "text" property.

Because the HTMLDocument is a "parent" to the HTMLText component, the HTMLText component can additionally be associated with actions that affect the HTMLDocument. When the HTMLDocument receives a print message, it prints its child, HTMLText, along with any other child HTML components that may be defined.

Both HTML components generate a whenOutput event, which can be used to trigger actions. The HTMLDocument event is ignored here, but your second action is set up to listen for the HTMLText whenOutput event. When that event takes place, the action sends the setText message to update the HTMLText component's "text" property, using the current time from the Clock component.



You'll notice that this action uses the alias "actor." In AppComposer, you can refer to components by name or by one of three aliases: actor, parent, capsule. When you use an alias, you localize control of the behavior – which allows you to reuse it in any HTMLDocument that includes a Text component.

Aliases

Normally, code references objects, functions, and data variables by name or path. This limits reusability, since you have to change the references manually to apply, for example, a behavior to a different actor. To increase reusability, AppComposer provides three *aliases* that you can use when defining behaviors. These aliases reference objects relatively, by their parent-child relationship to the behavior. That way, you can copy or save and reuse a common behavior, and apply it to a different actor or capsule, without having to edit it internally. These aliases are available from various menus as you define behaviors. They include:

parent	This alias refers to the immediate parent of the current behavior, which might be the capsule, an actor, or another behavior.
actor	This alias refers to the nearest direct or indirect ancestor that is either an actor or, if none is found, the capsule itself. This alias can refer to the same entity as `parent`.
capsule	This alias refers to the capsule that immediately contains the behavior.

The action behaviors that we defined for setting the mime type and printing are used frequently, so we can make them reusable within an action group behavior that is activated whenever the servlet receives a doGet request.

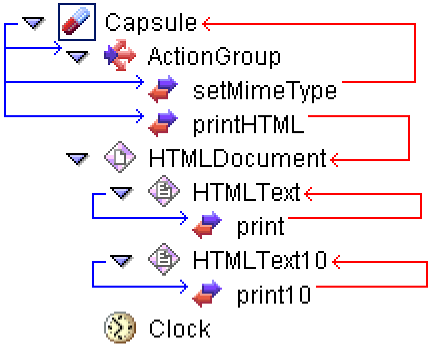
Action group behaviors

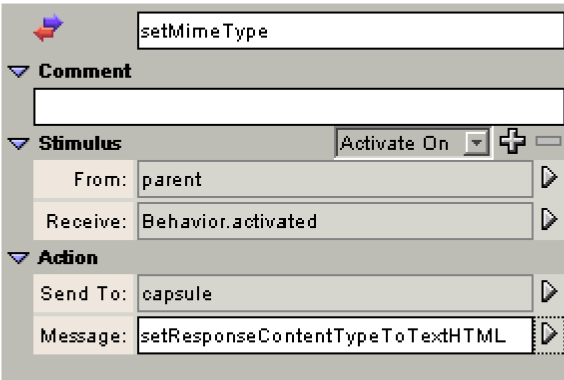
An *action group behavior* lets you group several behaviors so that you can move or copy them as a unit and make them respond as a unit. When activated, the action group behavior generates an event called `Behavior.activated`. The behaviors grouped within the action group can all receive this single event as their stimulus. However, this is up to you — you can set the stimulus for each behavior in the group independently.

Action groups are a very useful means of collecting several behaviors that all use the same activating event and perform related functions. Action group behaviors do not, however, actually constrain you in terms of functionality. They provide an easy way to copy, move, or reuse sets of actions; but if you wish you can use them to group wholly unrelated actions, which all have different activating events, and act on different actors. Action groups are also very useful to help logically organize a capsule's elements.

Whenever any two behaviors have the same stimulus, the capsule guarantees to execute them in the order they're listed. This is especially applicable inside an action group.

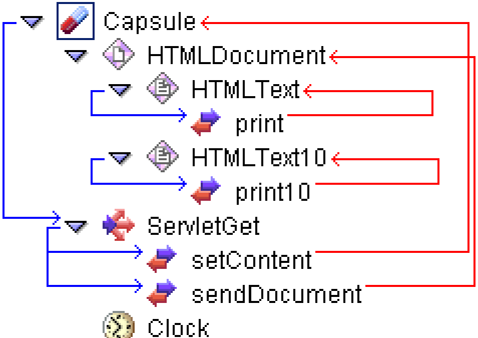
- First, we move the behaviors under the action group. Each behavior is activated when its parent (the action group) is activated.



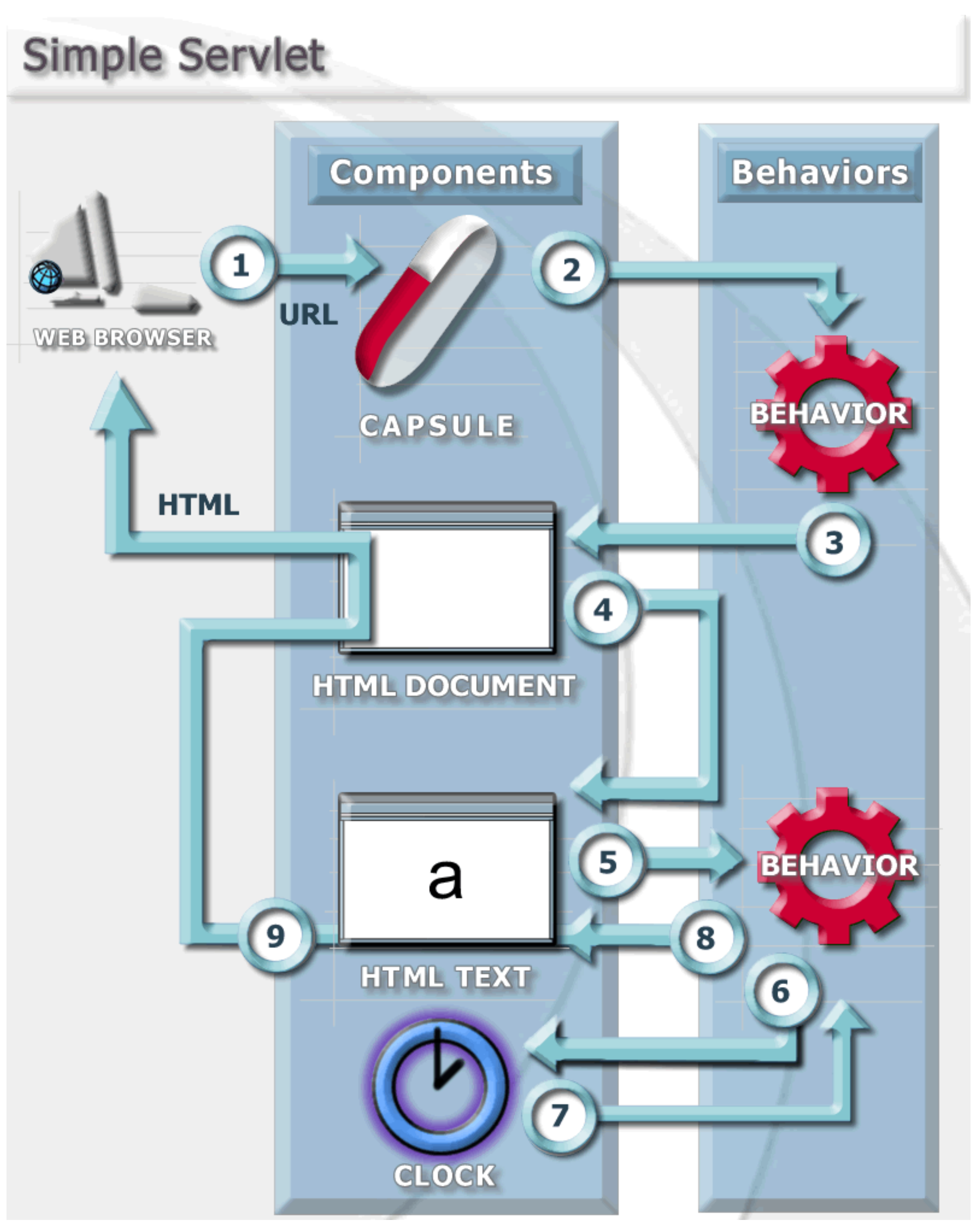


- To ensure that the behaviors that will work for any HTML document, regardless of where it sits in a capsule, just change the recipient of its action to actor (rather than capsule). The action group, and any copy that you might make of it, will then work for whatever HTMLDocument you choose as its parent.

We've defined a behavior that works for any HTML document, regardless where it sits in a capsule. In fact, this behavior is used so frequently that it is provided within AppComposer as a *saved group* called ServletGet. A saved group allows you to reuse any behavior or collection of behaviors. The ServletGet saved group combines two actions: setting the content type to HTML, and printing the document.



AppComposer's ability to save, reuse and edit common behaviors and actors is one of its most time-saving features.


Simple Servlet
Components
Behaviors
WEB BROWSER
1
URL
CAPSULE
2
BEHAVIOR
HTML
3
4
HTML DOCUMENT
5
BEHAVIOR
a
9
8
HTML TEXT
6
7
CLOCK

Summary

Here is a summary of the simple servlet that we've been examining in this chapter. The step numbers correspond to the steps in the "Simple Servlet" diagram on the preceding page.

1. The URL is sent to the web server, which sends a GET request to the servlet engine. The doGet request is passed along to the servlet capsule.
2. When the servlet capsule receives the request, that event triggers the action behavior "printHTML," which ....
3. sends the "print" message to the HTMLDocument.
4. The HTMLDocument prints its child, HTMLText. Before the HTMLText component is output, it triggers the event "whenOutput."
5. The "getCurrentTime" action behavior is listening for this event, and when it takes place ....
6. the behavior sends the message "getCurrentTime" to the Clock,
7. evaluates the result,
8. and places the result into the Text component's "text" property.
9. The capsule writes the HTML text (including the current time) into its output stream, which is sent back to the browser.

Java expressions

AppComposer provides the ability to embed scripts written in the Java programming language. You can use this capability in three ways:

- Java expressions can provide arguments to method calls in action behaviors.
- Boolean (true/false) expressions can control the activation and deactivation of any kind of behavior, or can define the test condition in an ifTest behavior.
- Script behaviors can execute sequences of Java statements.

Programmers normally do not use compiled languages like Java for runtime scripting because of the difficulties involved at runtime in compiling and linking scripts dynamically. Instead, programmers have used dynamic interpreted languages like Perl or Unix shell scripts. These types of languages can lead to performance problems in large applications, since the host machine has the extra step of interpreting the script every time the program does something. In order to use a compiled language like Java for scripting, AppComposer takes advantage of several advances in dynamic compilation and linking techniques, so that scripts can be recompiled and relinked even while your application is running.

Most of the powerful visual authoring systems that do provide scripting capabilities use a proprietary language, typically one that is unique to that system. This forces the advanced user who wants to use scripts to learn yet another new programming language, without the benefit of the many books and training materials available for a standard and popular language like Java. In addition, most tool-specific languages lack features and library functions found in a standard programming language. When you write a script in AppComposer using Java, you can take advantage of all the features and vast numbers of library functions available in Java.

To make Java more suitable for writing scripts, DigiSlice adds several upwardly compatible features to Java:

- AppComposer dynamically resolves names so that scripts can reference other objects (actors, behaviors, and data variables) and can set their values.
- AppComposer performs conversions between Java primitive types and their object counterparts. For example, AppComposer integer data variables can be passed, without error, to methods that expect either type `int` or type `Integer`. This vastly simplifies using values in AppComposer without sacrificing compatibility with the Java language.

- AppComposer bundles scripts (both expressions and script behaviors) as automatically generated method calls on automatically generated objects, to remain compliant with the Java specification. This relieves your programming overhead: you write the expression as a script, and AppComposer turns it into compilable Java.

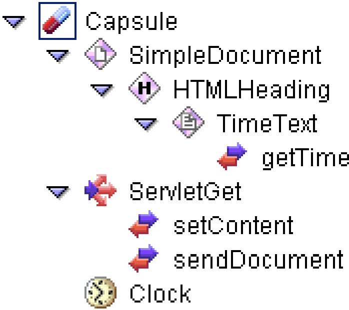
The result is a powerful scripting language that is remarkably easy to use.

Types of behaviors

In addition to the three types of behavior that we've seen thus far (action, action group, and saved groups), AppComposer defines five other types of behaviors: ifTest, return value, script, timeline, and counter. For details about behavior types, see Appendix A.

Capsule hierarchy – when does it matter?

Items in a capsule are organized in a hierarchy, which you manipulate as an outline. The figure below shows the outline view of a capsule.



The capsule is at the top of the hierarchy. The remaining levels of the hierarchy contain the capsule's *children* — actors, behaviors, and data variables that do the real work in the capsule. Note that the order in which entries appear in the hierarchy does not always reflect their execution order.

You determine activating events when you create behaviors, and the activating events may be placed anywhere within the capsule hierarchy. However, when two behaviors have the same activating event, their relative positions in the hierarchy do come into play. If several behaviors do have the same activating event, then they execute from top to bottom, according to their order in the outline view.

Order is also important when some components (beans) care about nesting. Some components, such as the HTML components, make inherent use of the parenting relationship. When an HTMLDocument prints, it instructs its children (HTMLText and other HTML components) to also print, so that all children are activated through that hierarchy to form the entire document.

Similarly, visual elements in an application or an applet make use of the parenting relationship. A visual window with a button lower in the hierarchy "parents" the button to that window.

In general, AppComposer does not constrain the types of parent-child relationships you can create in a capsule. The determination as to whether certain parent-child or actor-behavior combinations constitute good programming practice is left up to you.

There are some components that must have certain types of parents. For instance, visual components that are subclasses of `java.awt.Component` must have parents that are subclasses of `java.awt.Container.`

The capsule does not enforce any specific hierarchical relationships among its constituents. You can select any item in the outline and move it up or down in the hierarchy. When you do so, its parent may change.

Arranging the capsule outline with behaviors close to the actors they receive events from, and close to the actors they affect, makes the capsule easier to understand and maintain.

Running the servlet

AppComposer runs your application continuously as you build it. There are no separate edit, compile, and run steps to see the results of your work when you use the web server or AppComposer's toybox. When you have an idea, nothing prevents you from seeing immediately whether it works — and if it does not work, you can experiment with it until it does.

The ability to work with a running application is one of AppComposer's most powerful features. You use the toybox to run an application capsule while you create or modify it. For servlet capsules, you use the debug web server to display the output in a web browser while you create or modify it.

Debugging

AppComposer's advanced test and debug capabilities let you evaluate design alternatives and catch errors in real time. Applications can be manipulated even while they run for instant feedback. A distributed debugger, meanwhile, enables you to single-step through server- and client-side programs, set watched variables and breakpoints, examine program data, and identify runtime and compile time errors.

Because AppComposer runs your application as you modify it, you are encouraged to experiment. If you encounter an error, or the outcome of your change is not what you expect, you can begin debugging your capsule immediately. DigiSlice AppComposer provides the following debugging tools:

- When AppComposer encounters a bug in your application, one or more error messages appear in the Errors tab of the console. Use these to help locate the source of the problem. If you double-click the error message, AppComposer automatically selects the problem element from the outline.
- AppComposer has a console pane that displays certain kinds of output messages. You can also use this console for displaying debugging statements. For example, if you want to know when a certain script executes, you can add a line to the script such as the following:

  ```
  System.out.println("now executing Script A");
  ```

  Leave the console pane visible while you run your capsule to visually determine whether your debugging statement is being executed.
- The AppComposer debugger allows you to set breakpoints, single-step through your application one behavior at a time, and see the values of data items as they change. An arrow in the outline view shows you where execution has stopped, and a debug window allows you to inspect the values of any data variables you have asked to watch. You can see when the data variables change, and what they hold at any time.

"There isn't much code!"

At first glance, you might wonder, what kind of a programming paradigm is this? Where's all the code?

The truth is that, with AppComposer, you don't need to do nearly as much programming as you would otherwise need. The pre-existing components already contain the code needed to handle the details. This leaves you to concentrate on the logic pertinent to your problem, which mostly consists of how to plug these components together.

Your job is to define the events and triggers that drive your application. And because components hide the details of programming, building an application in AppComposer is commonly much faster than coding the equivalent application in Java.

In a sense it's not a programming problem at all, but a matter of using events and behaviors – pipes, really – to hook the various components together. And in that sense, the component model has quite a bit in common with plumbing, or home construction.

For example, you can build a house from raw materials – sand particles – or you can use components – windows, walls, bricks, plumbing, etc.

In any case, the builder has to be highly skilled to connect those components together to make a cottage, a house, a mansion, or a state building. It's not an issue of whether the components are rich enough. (And when components aren't rich enough, you can get new components manufactured – custom-built for the job.) You can do exactly the same thing in AppComposer.

In fact, there is a lot of code – and it's inside the components, already written and tested for you.

What's next?

In the following chapters, we'll examine the steps involved in building a web application, with special attention to AppComposer's role in this process.

CHAPTER 3

Web applications

DigiSlice AppComposer was designed to help you assemble applications out of software components. It is especially useful for building enterprise applications such as web applications.

Until recently, most computer applications, such as word processors or spreadsheets, ran on single computers. That type of application contains both the user interface and *logic layer* parts of the application. The user interface is the windows, buttons, menus, and other means by which you communicate with the program. The logic layer manipulates the underlying data. The entire application — both the user interface and underlying logic — runs on a single computer and accesses data on that same computer.

Client-server business applications use a model that separates the user interface from the underlying logic. The user interface runs on the user's computer and makes requests as a *client* to a *server* computer that manipulates the underlying data. For example, an airline reservation system might have many thousands of client PCs, one for each reservation agent, all connected to a central computer that makes flight reservations and issues tickets. Client-server applications, such as airline reservation systems and bank teller systems, are a kind of distributed application.

The World Wide Web is itself a distributed application that runs over the Internet. It consists of computers that function as *web servers* — computers that serve resources such as text, images, sounds, or other applications. These resources are accessed using a *URL* (Uniform Resource Locator), which functions as the address to the resource. A web browser client uses URLs to access various resources over the Internet. One such resource is an HTML web page. HTML consists of tags that mark the structure of a web page's content.

Technologies such as CGI (Common Gateway Interface) and Java Server Pages (JSPs) allow web pages to present dynamic content — content retrieved from databases or computed on demand. These technologies run on the web server, and are called *server-side* technologies.

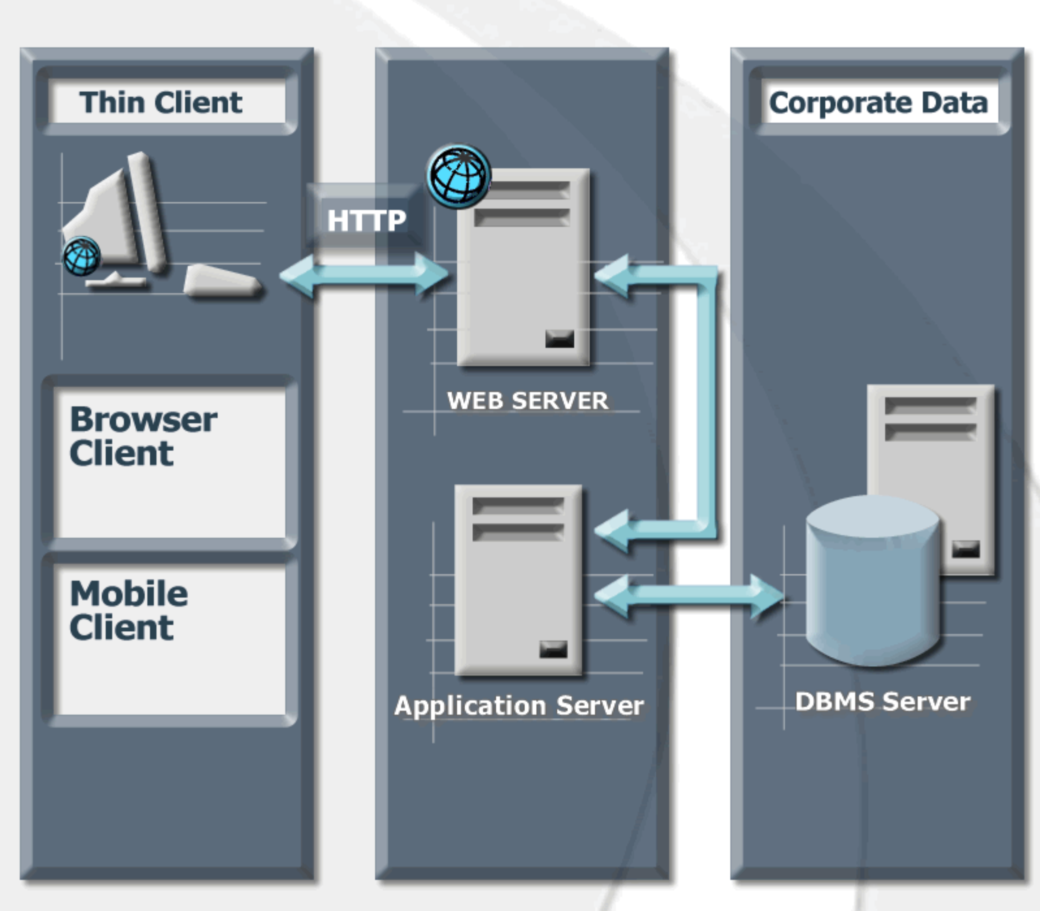
A Web application is a distributed application whose thin client is a web browser. Examples of web applications range from simple hit counters, to web sites that provide stock prices and weather reports, to complex e-commerce and business-to-business applications.

Web applications make it easy to provide distributed front-end access to existing client-server applications. For example, a business with a client-server human resources system might want to build a simple web application that allows employees to see how much vacation time they have accrued. Employees could then use their browsers to interact with the application, rather than all of them needing to get a copy of the human resources software on their machines.

Although creating static web pages using HTML is relatively easy, creating server-side, interactive web applications can be very complex – and expensive. This kind of web application is typically created from the ground up by programmers, without the benefit of prebuilt components or rapid development environments.

Open standards, like HTML (hypertext markup language) and HTTP (hypertext transfer protocol), have allowed the development of powerful applications that interoperate with each other and do not require extra overhead to deploy into different environments.

Because there are few open standards for server-side web applications, most of the available development tools use proprietary technology, which typically locks you into using proprietary servers, languages and other limitations.



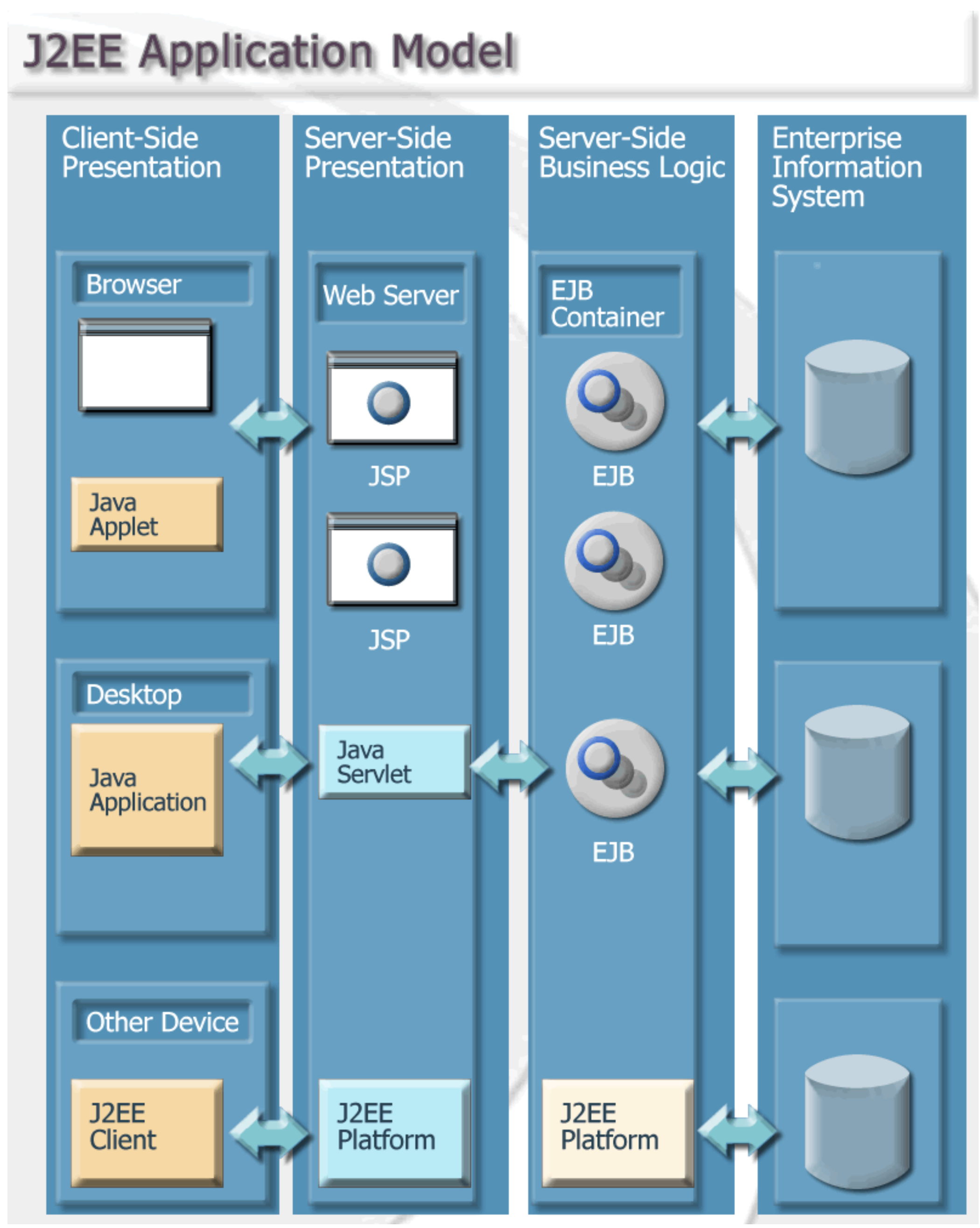
The J2EE application model

Java 2 Platform, Enterprise Edition (J2EE) defines a standard for developing multi-tier enterprise applications. It takes advantage of most features in Java 2 Platform Standard Edition:

- "Write once, run anywhere" portability
- Security model
- JDBC API, and CORBA interoperability models

In addition, J2EE adds full support for Enterprise Java Beans (EJB); Java Servlets API and Java Server Pages; and XML technology.

Because transaction management, life-cycle management, and resource pooling are built directly into the J2EE platform, you don't need to worry about these issues and are free to concentrate on providing business logic and functionality.


J2EE Application Model
Client-Side Presentation
Server-Side Presentation
Server-Side Business Logic
Enterprise Information System
Browser
Web Server
EJB Container
JSP
EJB
Java Applet
JSP
EJB
Desktop
Java Application
Java Servlet
EJB
Other Device
J2EE Client
J2EE Platform
J2EE Platform

Browser

The browser knows how to connect to the server over the Internet, and specifies the pages or resources that the user wants in the form of a URL (uniform resource locator). The URL defines the protocol to be used, the server (and port) to connect to, and any specific resources that the browser is requesting. The browser expects the server to return its response in the form of HTML code.

Web server

The server, meanwhile, knows how to listen for requests from clients (web browsers), and knows how to find the requested resources. The server manages its own file system and responds to the browser in HTML.

Java servlets and JSPs

Java servlets and Java Server Pages (JSPs) are server-side technologies that manage interactions with the end user.

A Java servlet encapsulates server-side logic in a web application. Java servlets execute using an application server or web server extensions, accessing server resources and applications as needed.

Java Server Pages (JSP) is a web scripting technology similar to server-side JavaScript (SSJS) or Active Server Pages (ASP). A Java Server Page is essentially an ordinary HTML page with special tags that prompt the client to ask the server for information.

JSP functionality can be extended with tag libraries. JSP tags provide a standardized means of incorporating dynamic elements into static HTML pages. JSPs give you the ability to separate the page description – the presentation logic – from the business logic that makes a Web page dynamic.

AppComposer comes with its own JSP tag library for enhanced functionality and capsule integration.

For further general definition of JSP technology, see:

`http://java.sun.com/products/jsp/faq.html#1`

Enterprise JavaBeans (EJBs)

Enterprise Java Beans (EJBs) are server-side components that capture the business logic of the application. EJBs allow you to build business logic “components” that are distributed, transactional, secure – and reusable.

Enterprise JavaBeans provide cross-platform integration with enterprise services, including:

- databases
- distributed transactions
- security
- messaging
- mission-critical robustness and scalability

Enterprise JavaBeans make use of standardized conventions, which any EJB-enabled web application server can interpret, to access these services.

For more information about Enterprise JavaBeans, see:

`http://java.sun.com/products/ejb`

Application server

Applications built with EJBs require a suitable EJB application server (such as BEA WebLogic) that includes an EJB container. The EJB container provides services such as transaction management and resource pooling.

AppComposer comes integrated with JBoss, an open-source EJB server implementation. You can configure AppComposer to work with an application server of your choice.

For more information on JBoss, see:

`http://www.JBoss.org`

Databases

Databases provide organized storage and access to information.

Databases are the primary way corporate data is stored. Most databases use the relational model for structuring information, using tables (rows and columns) to store the data.

AppComposer supports integration with databases. AppComposer is bundled with Hypersonic SQL (HSQL), and supports access to a large number of database implementations via JDBC drivers. For more information about PointBase, see:

`http://hsqldb.sourceforge.net`

AppComposer's connection manager uses a pooling mechanism to support multiple data sources with multiple connections. What's more, AppComposer provides a full set of SQL components to facilitate the authoring of data-enabled applications.

Separating the business logic from the presentation

JSP technology supports a clean separation of programming logic – that is, flow of control – from the presentation. The JSP designer can concentrate on the graphical design of the page – the presentation logic – while the Java developer handles the processing and the data access elsewhere, in JavaBeans and EJBs.

AppComposer supports this use of JSPs. Managing the flow of control in AppComposer, rather than in the JSPs, allows you to develop an application that and are available to the components that support its much easier to maintain. You can incorporate changes to the user interface without necessarily affecting the application's flow of control.

Application authoring environments

To create a web application, you can choose from several kinds of authoring environments. Let's look at how you might construct an application in each environment and see where AppComposer fits in.

Basic web server

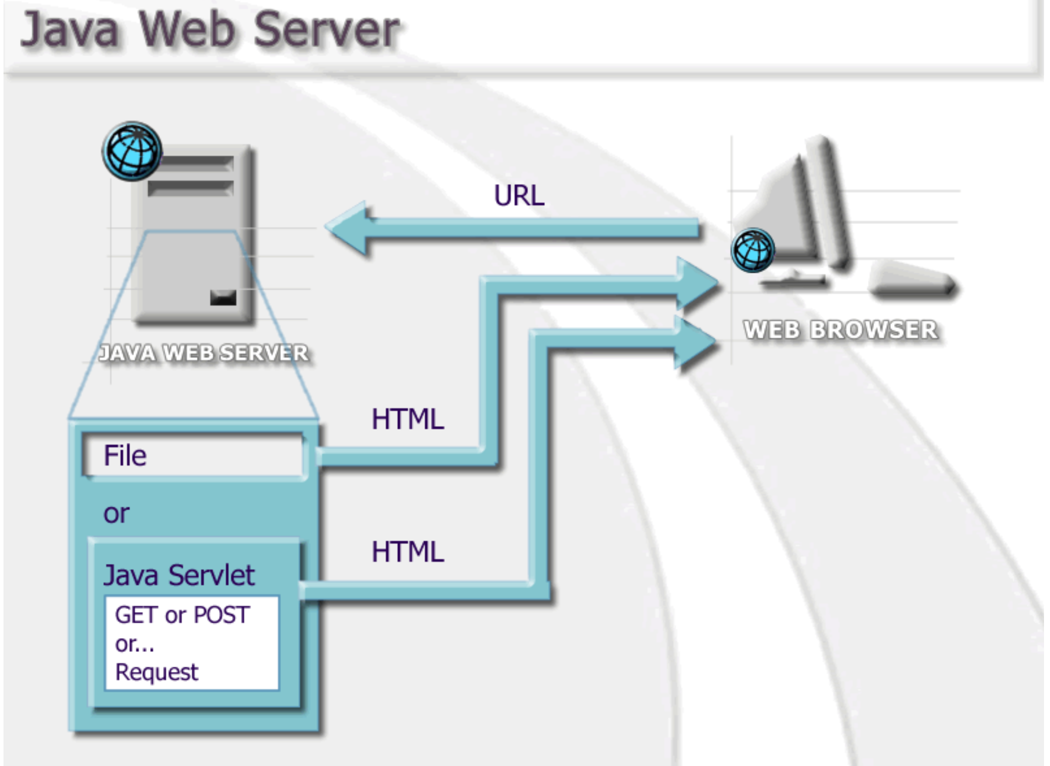
In a basic web server, the user clicks a button or a link in the browser, thereby providing the URL of a web page to view or the filename of a CGI shell script to execute. In either case, the web server returns HTML to the browser.

This environment is indeed basic; interaction in HTML is limited to clicking on links, and even CGI is limited to simple text-based forms.

Java web server

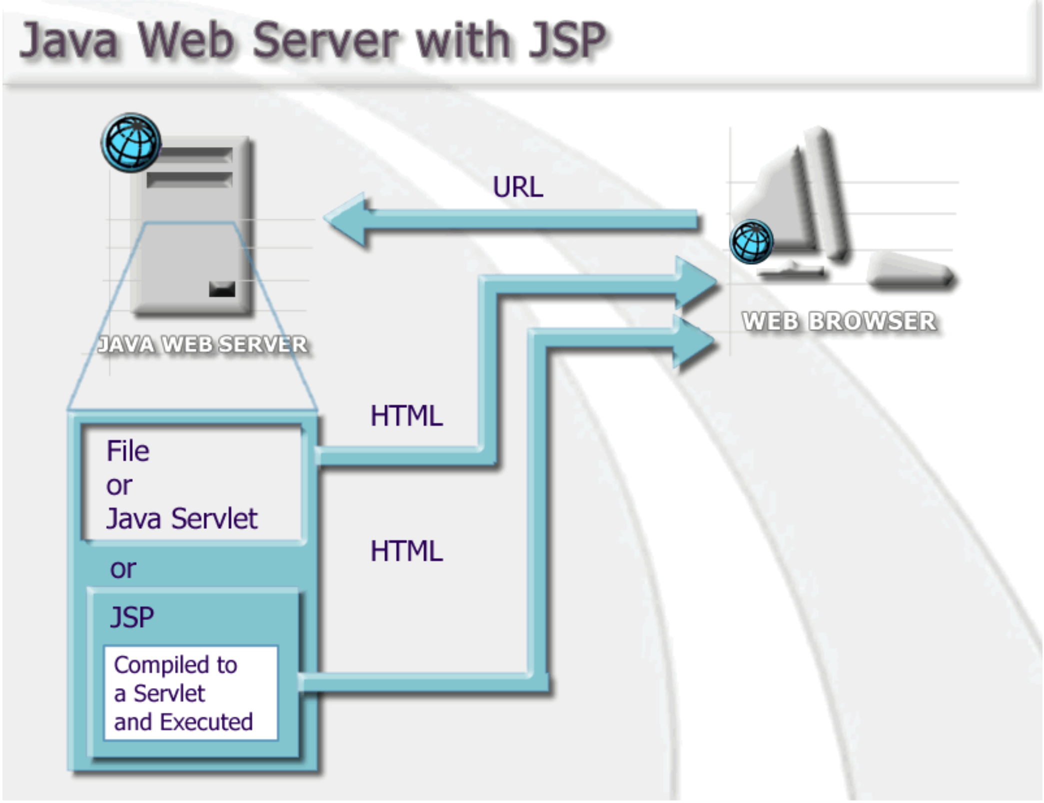
In a Java web server, servlets and Java Server Pages (JSPs) replace the CGI shell scripts.

When the user clicks a button or a link, the web server either returns an HTML file or executes a Java servlet containing a GET or POST or another Java method. The result of executing the Java servlet is output to the browser as HTML.



Java web server with JSPs

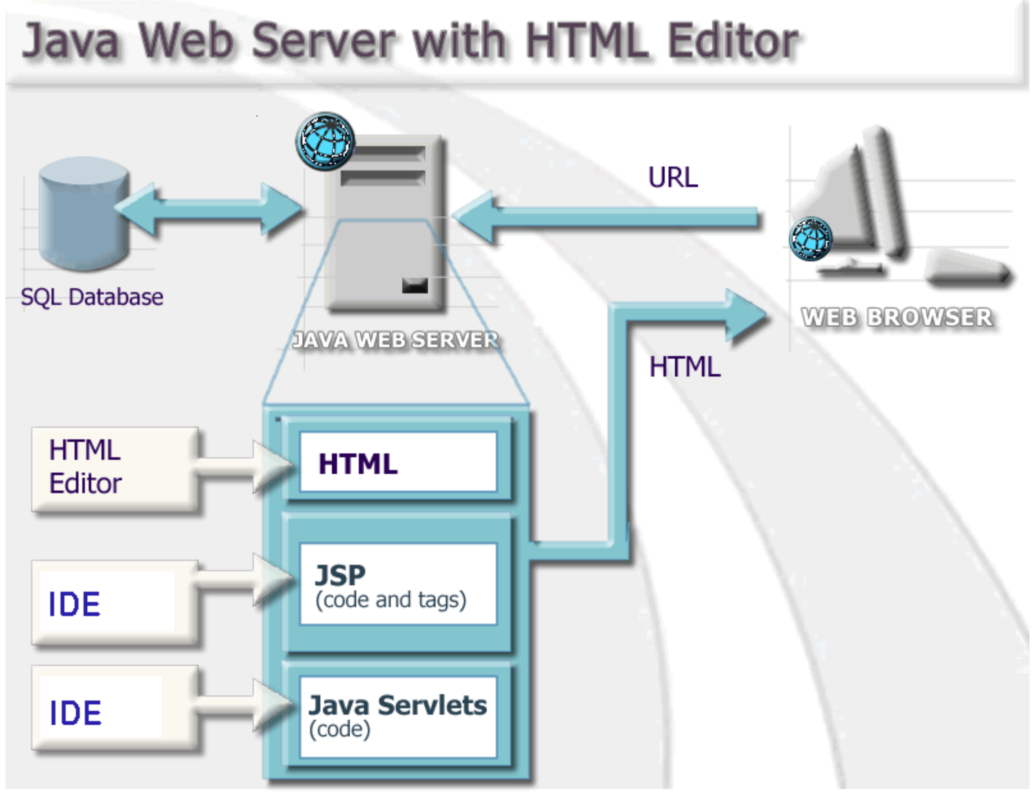
When the user's action calls a JSP, the Java web server executes that JSP. The JSP – containing the text for the JSP page, along with HTML tags and JSP tags – is compiled to a Java servlet and then executed. As above, the result is output to the browser.



Java web server with HTML editor

You can use different tools to create the various parts of a web application. For example, you can use any HTML editor to create the HTML pages and the text for your JSP pages. To create the Java code for your JSP pages, and to create Java servlets, you'd use an integrated development environment (IDE).

In this approach, the Java code serves as the "glue" code that establishes the flow of control for your application.

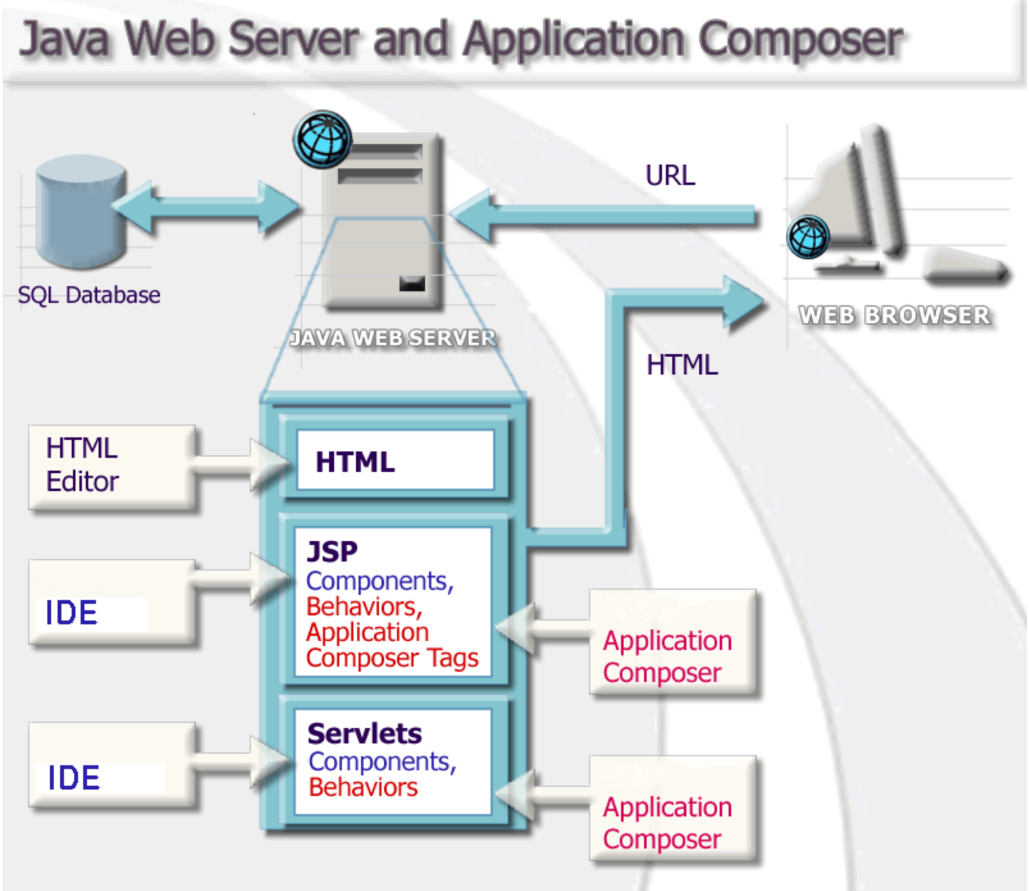


Java web server with AppComposer

With AppComposer, you create components and behaviors that establish the business logic for your application.

Using JSP tags with AppComposer capsules allows you to separate the data model or business logic, specified in the servlet capsule, from the presentation logic, specified in the HTML or JSP file.

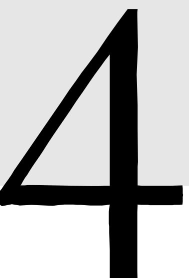
In this model, you can use an IDE to create the EJBs and other Java components in your capsule.



What's next?

In the next chapter, we'll look at the steps involved in building a web application with DigiSlice AppComposer.

C H A P T E R



Building a web application with AppComposer

The stages of application development

The following paragraphs present a series of steps involved in building a web application with DigiSlice AppComposer. In designing your web application, you may want to follow these stages, or variants of them.

1. Before you begin to build your web application, it's important to plan it carefully. Consider the pages needed for the processes that you wish to implement. Consider the navigation that a user is to make between those pages to form your network and solve the use cases that you come up with.

2. Parts of your web application will be static, represented by HTML files. Other parts are dynamic. To represent those parts, you can use JSPs or Java servlets or a combination of both. In designing your application, it is important to consider the flow between static HTML pages and the dynamic parts of your application.

   It is possible to redirect or forward to a different page from a dynamic page. In figuring out your flow from one page to another, you may encounter various error conditions or state changes where redirecting a flow is important. You can make use of the forward and sendRedirect messages in your servlet to capture these cases.

In AppComposer, you are probably best served by using a servlet capsule for each dynamic page of your application. A servlet represents a URL used to query the web server for an HTML page. Each servlet may use one or more JSPs to represent its results. It will also use components to formulate those results. This is a good time to evaluate the components that you have on hand and determine which components you'll need to find or create.

3 Set up your HTML pages and add them to your project.

Projects are application-level containers for capsules and their external resources. The project reflects the way that you organize your web application. Your project consists largely of capsules, JSPs, HTML files, images, database connections, and other resources. By using projects to organize your capsules, you maintain relative access to all needed resources.

You should build all of your capsules within the context of a project, even if the project contains only one file.

4 Visually design your web pages. Set up the JSPs that you will need and make them available in the project.

5 Set up your servlet capsules that use those JSPs. Identify the events from the server that you'll use to decide which JSP to display under which circumstances.

6 If your web application uses a database, set up your tables and add any needed database drivers to your classpath.

7 Incorporate components into the servlets that allow dynamic data to be fetched so that it can be transformed and represented as part of your JSP output.

a Import new EJB components into AppComposer and deploy them on their executing servers.

b Bring new JavaBeans into AppComposer by placing their JAR files into the beans directory.

8. Test, test, test!
9. Once an application is successfully composed it must be deployed into a target environment for testing or production use. Deployment involves three steps:
   a. You must deploy your EJBs to their destination application server.
   b. You must set up your application to use EJBs on the deployment server.
   c. Deploy your actual project.

Later in this chapter, we'll pay particular attention to issues involved in setting up JSPs, database tables, and EJBs.

Working with JSPs

With AppComposer, you're able to divorce your application's business logic from the JSP, so that it solely deals with presentation. Because the servlet handles all of the business logic, it's much easier to maintain your JSPs.

In this approach, the JSP's role is simply to present data or allow the user to submit data. In either case, you use the tag library that's provided with AppComposer. The library includes two types of JSP tags: control (present the data) and user interface (handle input or display output).

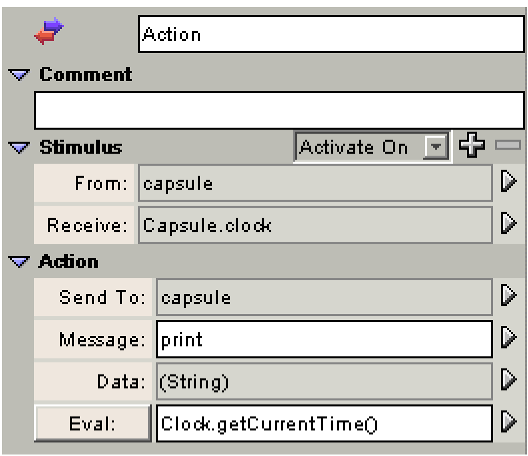
AppComposer provides three JSP tags – call, conditional, and repeat – to control the HTML output sent from the servlet capsule to the browser. Each of these control tags causes a capsule method to be invoked.

The simplest control tag is the call tag. This tag calls a capsule method, allowing arbitrary HTML to be inserted into the document; that is, it allows the capsule to print arbitrary data to the HTML output stream. The call tag is also used to trigger the capsule to perform actions when a certain point is reached in a JSP file.

Let's assume that you've created a JSP file called clock.jsp. That file contains a single composer control tag:

```
<composer:call method= "clock"/>
```

This tag calls the method "clock", which tells the capsule to print the current time.



The JSP file (clock.jsp) contains the HTML code and JSP tags that tell the browser how to display the current time. The JSP file does not contain any of the logic for obtaining the time; that logic is contained in the capsule.

When the JSP call tag sends the method "clock", the capsule sees the method as an incoming event and relays it to any object listening for that event – in this case, the capsule itself.

JSPTemplates

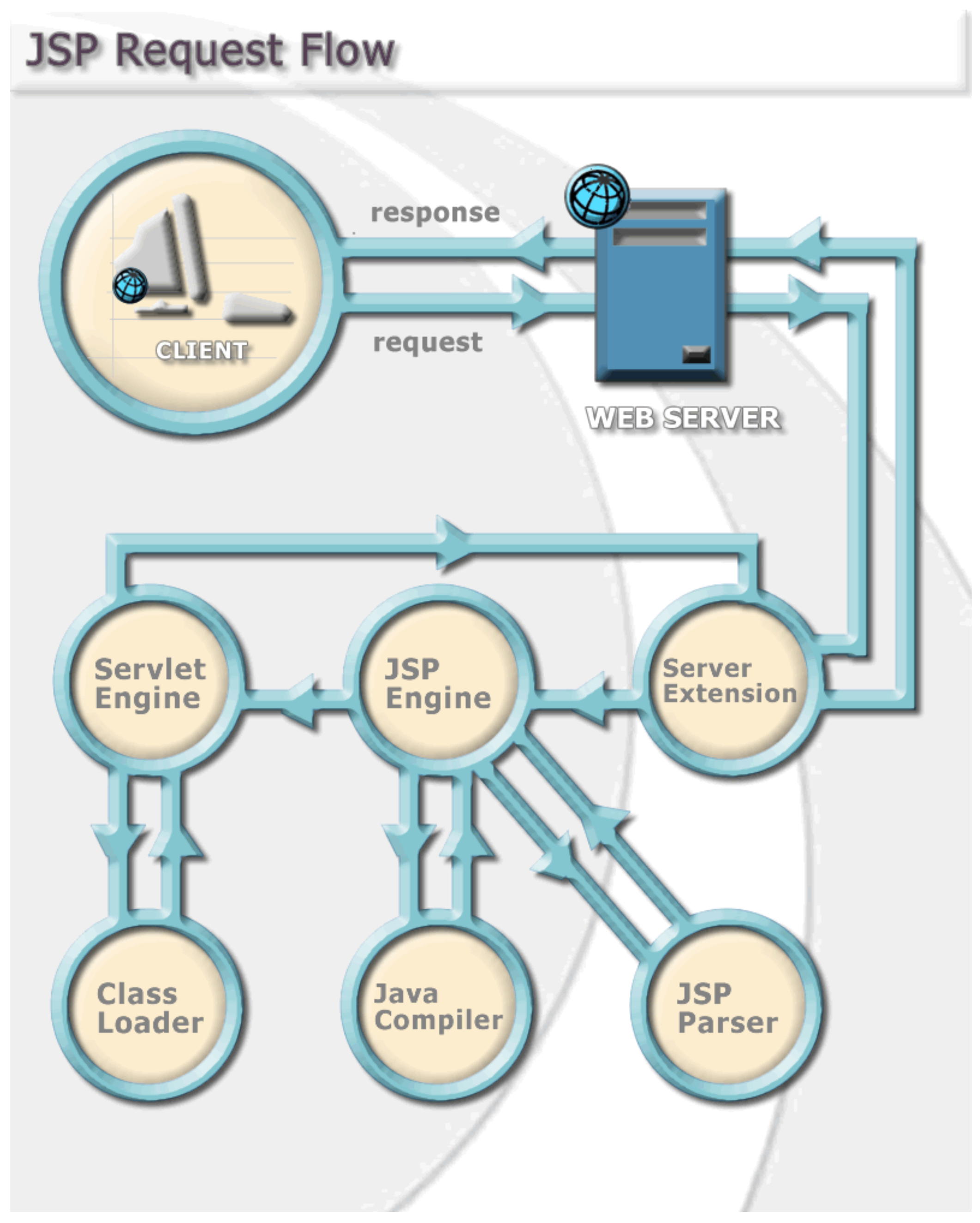
The capsule contains a JSPTemplate, which corresponds to the JSP used by the capsule. By its nature, a template is not complete. A template is something that you start with; the JSPTemplate sets up the JSP and is dynamically modified.

More formally, a JSPTemplate is a JavaBean that processes the commands within the corresponding JSP, resolving callbacks from AppComposer tags inserted into the JSP. In essence, the JSPTemplate serves as a bridge between the JSP and the capsule.

When the print method is sent, the JSPTemplate outputs the page to which it is linked (via the template's path property). If that JSP contains control tags, the associated capsule methods are called from the compiled JSP.

JSP request flow

As shown in the following diagram, the JSP file is compiled into an HTTP servlet by the JSP compiler. The compiled servlet sends its response to the browser, usually in HTML form.



JSP Request Flow
response
request
CLIENT
WEB SERVER
Servlet Engine
JSP Engine
Server Extension
Class Loader
Java Compiler
JSP Parser

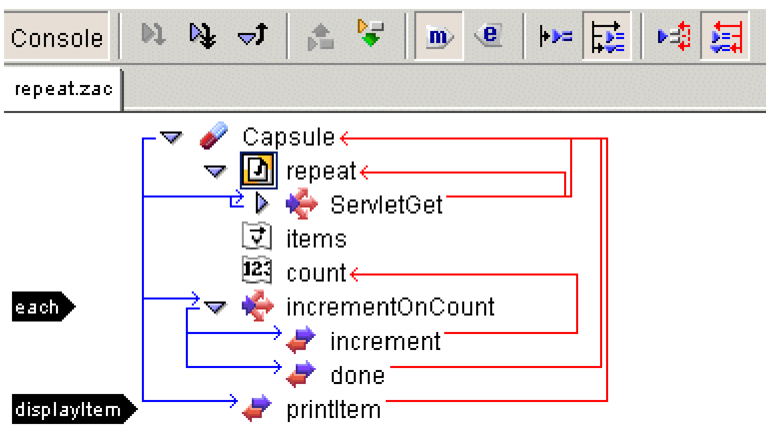
Using the JSP repeat tag

Let's look at an example of how you might use the composer repeat tag. This tag is useful in tables and lists – really, anywhere you might want to visually represent some collective quantity on a page.

The repeat tag repeatedly calls a method until the capsule reports a false status. You can optionally specify a maximum number of times to execute the body of the tag.

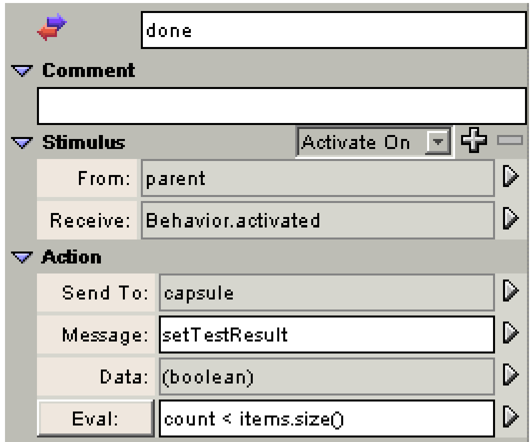
In this example, we do a repeat using the method "each".

```
Using the Composer repeat Tag
<table width="75%" border="1">
<composer:repeat method="each" limit="1000"> <tr>
    <td>Item</td>
    <td><composer:call method="displayItem"/></td>
  </tr>
</composer:repeat>
</table>
</body>
</html>
```



In the capsule, you can click on the "show methods" button to view all methods in the capsule. This is a very handy way to help you understand the flow of control in your application.

For each line in the table being retrieved from the database, we increment the counter. To tell the capsule whether to continue the repetition, you can use the method setTestResult.



Using the Composer repeat Tag

Item	one
Item	two
Item	three
Item	four
Item	five
Item	six

Using the JSP conditional tag

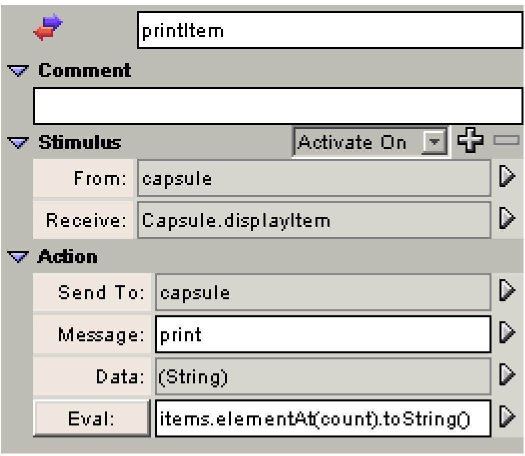
The conditional tag calls a capsule method. If the method specified in the tag evaluates to true, AppComposer processes the body of the tag. If the method returns false (or is not found), the body of the tag is ignored.

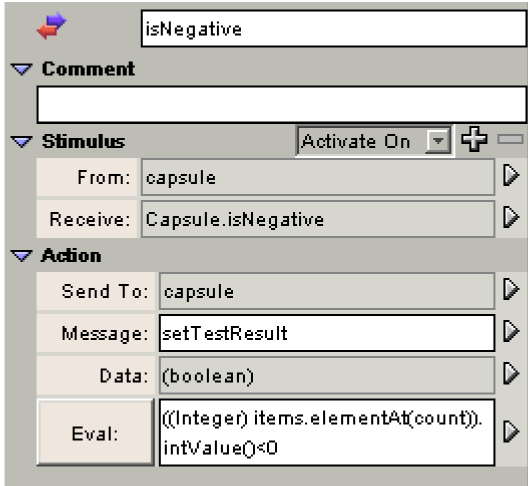
Let's modify our example slightly. We'll edit the JSP to use the <composer:conditional> tag twice: once to manage output for a positive result, and a second time for a negative result.

```
<composer:conditional method="isPositive"/>
<composer:conditional method="isNegative"/>
```

```
Using the Composer conditional and repeat Tags
<table width="75%" border="1">
<composer:repeat method="each" limit="1000"> <tr>
    <td>Item</td>
    <td><composer:conditional method="isPositive">
<composer:call method="displayItem"/>
</composer:conditional>
<composer:conditional method="isNegative">
<font color="#FF0033">
<composer:call method="displayItem"/>
</font>
</composer:conditional>
</td>
  </tr>
</composer:repeat>
</table>
</body>
</html>
```

Again, we take care of all of the application's business logic in AppComposer, in the capsule. We need to add three behaviors: printItem (fetch the count to a string, since there is no string); isPositive and isNegative (set the test result on the capsule, with coercion to an integer).





In the result, negative values are displayed in red.

Using the Composer conditional and repeat Tags

Item	0
Item	1
Item	2
Item	3
Item	4
Item	-10
Item	-5
Item	5

Accessing the SQL database

Thus far, we've seen how to use AppComposer tags in JSPs as a way of linking the business logic – the manipulation of data in AppComposer – with the presentation logic – how that information is displayed in the JSP.

AppComposer provides components to help you connect the JSP with your SQL database.

AppComposer includes a full set of components that encapsulate SQL data manipulation commands:

- SqlDelete – executes a SQL DELETE command
- SqlInsert – executes a SQL INSERT command
- SqlSelect – executes a SQL SELECT command
- SqlUpdate – executes a SQL UPDATE command
- SqlProcedureCall – executes a stored procedure
- SqlRawStatement – executes any SQL command
- SqlConnection – encapsulates a database connection

JDBC

JDBC, the Java Database Connectivity API, is a Java standard for accessing databases. Connecting to a database requires a JDBC driver, usually provided by the manufacture of the database product. JDBC supports the execution of SQL commands, parameter bindings and result set processing.

With the correct driver, you can link from AppComposer to any existing database. ODBC and PointBase drivers are provided with AppComposer. For other databases, use the driver supplied by the vendor or by a third party.

To configure your system to connect to the database, you can either add drivers, or add descriptions of the database connection. (For details, see the *AppComposer User's Guide*.)

Using the SQLSelect bean

You can use the JSP Repeat tag in conjunction with the SQLSelect component to get information from the SQL database and display it in the JSP.

Let's consider an SQL database that contains customer information for a sales organization. To obtain the internal customer number and customer name from the SQL database, you might include these two <composer:call> tags:

```
<p>Using SQL Select </p>
<table width="75%" border="1"><tr>
    <td><b>Customer Number</b></td>
    <td><b>Customer Name</b></td>
  </tr>
<composer:repeat method="each" limit="1000"> <tr>
    <td><composer:call method="displayNum"/></td>
    <td><composer:call method="displayName"/></td>
  </tr>
</composer:repeat>
</table>
</body>
</html>
```

To use AppComposer's SQLSelect statement, here are some of the values you might need to change:

- Table – choose the table from which to get the data.
- Columns – get the customer name and associated customer number.
- Sorts – display the names in ascending alphabetical order.
- Formats – represent and manipulate the customer number as a string.
- Options – remove AutoNextOnExecute, when using the Repeat tag to manage iteration through the results.

In the capsule, you would also need to insert action behaviors to initialize the table, iterate through the database, and display the information in the JSP.

Using the SQLUpdate bean

If you need to give the user a way to change customer names in the browser, you could use the <composer:form> JSP tag to create a form for each row:

```
<p>Using SQL Update </p>
<table width="75%" border="1">
  <tr>
    <td><b>Customer Number</b></td>
    <td><b>Customer Name</b></td>
  </tr>
  <composer:repeat method="each" limit="1000">
  <tr><composer:form action="sql2" method="POST">
    <td><composer:call method="displayNum"/>
        <composer:hidden name="displayNum" value=""/></td>
    <td><composer:text name="displayName"/>
    <composer:submit value="Submit"/></td>
    </composer:form>
  </tr>
  </composer:repeat>
</table>
</body>
</html>
```

User interface tags

AppComposer user interface tags mirror HTML tags of the same or similar name, but with additional functionality and control.

When translated by the JSP engine, user interface tags such as <composer:form> or <composer:text> are converted to standard HTML form. Using the user interface tags, AppComposer can get or set all values in a form in a single action – which is much more efficient than having to perform the same steps with straight HTML.

AppComposer user interface tags must be embedded in a AppComposer form tag to utilize their added functionality.

For a complete list of user interface tags, see the *AppComposer User's Guide*.

In the capsule, you would also need to make the necessary changes so that information from the JSP is passed properly to the database. For example, you might insert action behaviors to fill out form values as each row is generated. You would also need to modify the ServletGet saved group to also allow POST requests.

Finally, you would use AppComposer's SQLUpdate statement to tell the SQL database how to process the user input.

Performing multiple updates

You can simplify the process of updating the database by bunching all parameters at once. Let's look at how you would do this for multiple parameters in a form. That is, if your form had many (~20) parameters from the database and you needed to name each one, you should be able to do this in batch.

One of the main issues when updating a web site is that database tables frequently might change. In particular, columns might be added. By designing your application carefully, you can change the JSP and the SQL statements, and nothing else – and the application will work, even when new columns are added. This approach

makes it easy to maintain, and ultimately makes your capsule simpler. (The alternative would be to specify every field in the capsule with two separate actions, one by one – one action to display, and one to create the SQL update statement. More complicated, and much more work to maintain.)

The key to this approach is to set up the JSP to use the same parameter names (matching case, etc.) as in the SQL database. AppComposer looks for an exact mapping between JSP parameter names and SQL fields. In essence, you need to construct a seamless passthrough between the JSP and the SQL database.

Sessions: maintaining state between pages

With web applications, state has to be maintained between pages if you want to correlate information from one page to another and relate it to a particular user. You can do this by one of three methods:

- Carry the state in hidden fields between pages.
- Use cookies to store application state on the client browser.
- Use sessions to store connected session state on the application server.

Sessions have several advantages, chiefly that it is easier to maintain state on the server side, as it is closer to your application.

Typically, you'll use a session to remember whether a user has logged in, who they are, what's in their shopping cart, or any of the other data pieces that relate to the transient state of your web application.

Sessions are separate from your web pages and servlets. A servlet that makes use of a session must explicitly store and retrieve data to maintain this state. A session can be considered to be like a hashtable – it stores key-value pairs, where the value can be any object. You might use the presence of a key in a session to determine whether or not a user has logged in, and if not, redirect the user to a login page.

Typically, you might make this test in every servlet that you write for a user-validated web application. You would do this because the user can bookmark any URL and effectively enter the site that you're creating at any place, at any time. It's important to be aware that the site alone does not account for the flow from page to page.

Working with EJBs

Enterprise Java Beans provide a means of building "business logic" components that can be installed in a distributed environment. This allows the components to be used by a wide range of different kinds of client applications.

By their very nature, enterprise applications are inherently difficult to build and maintain; they are large-scale, multi-user, distributed applications that are often transactional and access a variety of data storage devices (relational database, mainframe, etc.). Enterprise Java Beans can make it easier to build enterprise applications.

Setting up an EJB for use

AppComposer makes it easy to use EJB components to handle the data modeling work of the application.

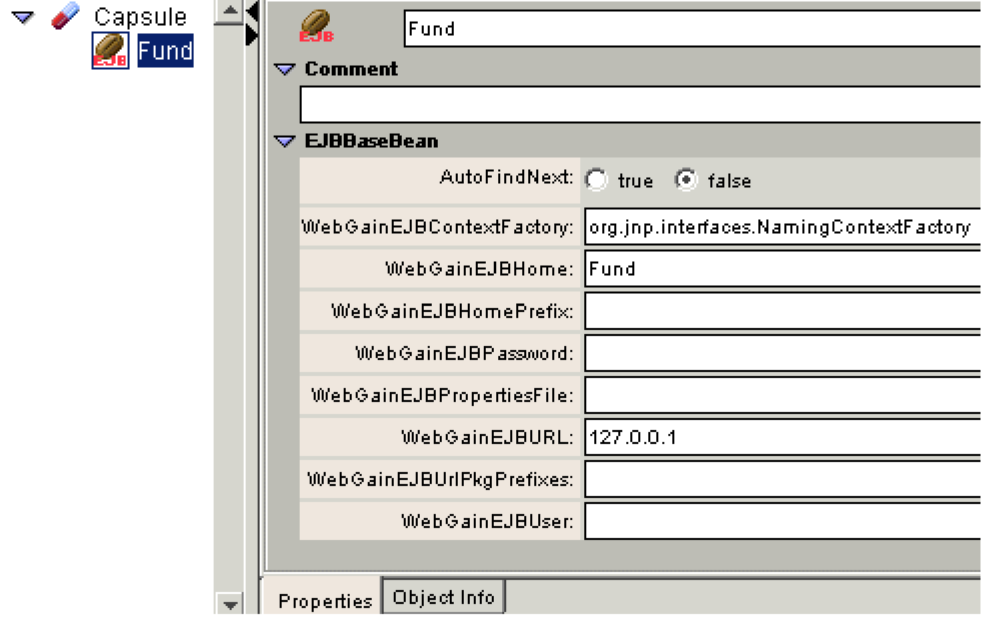
When you import an EJB, a JavaBean wrapper (EJBBaseBean) is created automatically. The JavaBean wrapper encapsulates the home and remote interface of the imported EJB.

AppComposer takes care of the details so that working with an EJB is no different than working with any other component. Once you've imported the EJB, the generated EJB wrapper is available from the Insert menu under EJBGen.

AppComposer is shipped with several tutorials to help you further your learning. Let's use the Mutual Fund tutorial to illustrate how to work with EJBs.

In a new capsule, insert the EJB that implements the Fund functionality: You can edit this bean to see its server connection. Whenever you do any operation on the bean that requires communication to the EJB server, the server to which the bean connects is described by a set of parameters, as defined in the bean's properties. To connect to a different server, just change the bean's properties.

Let's say you had a server running on a machine with a different IP address. You would simply change the DigiSliceDigiSliceEJBURL for that bean.



Entity beans and session beans

There are two distinct uses of EJBs – and two distinct kinds of EJBs exist to fulfill these roles.

An entity bean represents a persistent data object – an object with durable state that exists from one access to the next. Entity beans are secure and transactional. As such, the data encapsulated by an entity bean can be stored and retrieved from a relational database or some other persistent storage. Persistence can be managed by the container or by the entity bean itself.

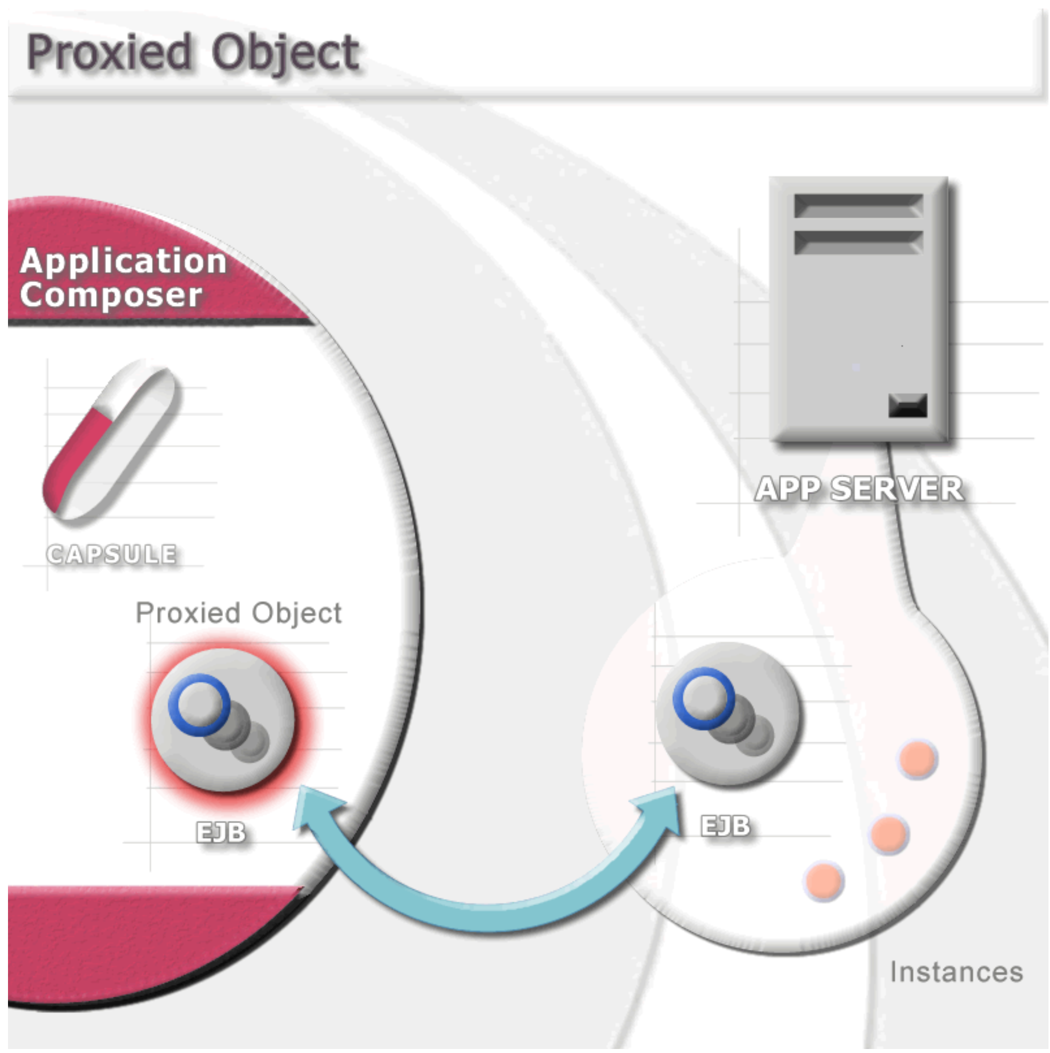
Session beans, meanwhile, are used to implement a business task, service, procedure, operation, or transaction. As such, session beans are inherently transient.

Proxied objects

EJBs differ from JavaBeans in that you cannot call methods on an EJB right away. Before accessing an EJB, you must first fetch one from the EJB application server.

You can either create an EJB by calling a create method or find one by calling a find method. Once you've created or found the EJB, you can use any of the other methods to communicate with that EJB on its application server.

When you import an EJB, AppComposer creates a JavaBean representation of the EJB in the capsule – a "proxied object." The actual EJB remains on the EJB application server.



What next?

DigiSlice AppComposer includes a set of tutorials that show how to build simple applications with AppComposer, and then progress in complexity all the way up to building a full-fledged web-based e-commerce application.

AppComposer also ships with a variety of example projects. Most of the example projects are servlets. They are designed to run on a web server and generate HTML output that can be displayed in a standard web browser.

To continue your learning with the tutorials and examples, choose DigiSlice AppComposer Documentation from AppComposer's Help menu.

A P P E N D I X

AppComposer actors and behaviors

Actors that ship with AppComposer

AppComposer is shipped with a set of pre-built reusable components that allow you to construct serious enterprise applications. AppComposer's core library includes components for accessing relational databases, generating HTML, reading and writing XML, handling e-mail, and more.

Html

HTML actors map to the various HTML elements: heading, title, text, etc. You can use HTML actors to dynamically generate HTML in your applications.

HTML actors are nested, using a parent-child hierarchy within the capsule. When the application outputs HTML, it follows a cascaded process. The print message is sent to the HTMLDocument component. The HTML Document in turn sends a print message to each of its immediate component children, then outputs an ending tag.

Sql

AppComposer includes a full set of actors that encapsulate SQL data manipulation commands:

- SqlDelete – executes a SQL DELETE command
- SqlInsert – executes a SQL INSERT command
- SqlSelect – executes a SQL SELECT command
- SqlUpdate – executes a SQL UPDATE command
- SqlProcedureCall – executes a stored procedure
- SqlRawStatement – executes any SQL command
- SqlConnection – encapsulates a database connection

Misc

These actors provide useful functionality such as date/time, fetching data from a URL, sending e-mail, generating events, output to the console, formatting, and more.

EJBGen

These EJB components are used by examples shipped with AppComposer. This menu includes any EJBs that you import into AppComposer.

AWT Components

These actors provide Java Abstract Windowing Toolkit (AWT) functionality. You can use the toybox to run an application capsule while you create or modify it. Using AWT actors (button, checkbox, list, scrollbar, etc.) in the toybox, you can see the effects of your changes immediately.

Display

AppComposer provides additional display actors (line, oval, rect, etc.) that you can use in the toybox in conjunction with AWT components such as button, checkbox, list, scrollbar, etc.

Types of behaviors

AppComposer defines the following types of behaviors for your use in capsules.

Action behaviors

An *action behavior* is the most common kind of behavior in AppComposer. An action behavior invokes a method on an actor. You can set it up to execute the method on any actor you designate, including its parent actor. The editor for an action behavior has various fields, which specify which method the action uses, what the method acts upon, and any parameters the method needs to run.

Although AppComposer makes it easy to use methods to build behaviors, it is important to be familiar with what data a method needs, and how it specifies the value of that data. A single method name may offer several ways of specifying additional data. For example, a method that requires a color as an argument might allow you to specify that color as an object of type `Color` using three integers that represent the red, green, and blue components of the color. Alternatively, you might be able to specify a string, such as “orange” or use a pulldown menu to select the color.

AppComposer allows you to define each required argument using either a constant or an evaluated expression. One of AppComposer’s powerful features is that it allows you to specify an evaluated argument using any Java expression. This expression can use values supplied by other actors, call Java functions, and effectively do anything you can program in Java.

Action group behaviors

An *action group behavior* lets you group several behaviors so that you can move or copy them as a unit. When activated, the action group behavior generates an event called `Behavior.activated`. The behaviors grouped within the action group can all receive this single event as their stimulus. However, this is up to you — you can set the stimulus for each behavior in the group independently.

Action groups are a very useful means of collecting several behaviors that all use the same activating event and perform related functions. Action group behaviors do not, however, actually constrain you in terms of functionality. They provide an easy way to copy, move, or reuse sets of actions; but if you wish you can use them to group wholly unrelated actions, which all have different activating events, and act on different actors. Action groups are also very useful to help logically organize a capsule's elements.

ifTest behaviors

AppComposer allows you to specify conditional activation for any behavior. That is, you can set your behavior to evaluate whether a specified condition is true when it receives its activating event. If the criteria are true, the behavior executes, otherwise it does not, even if it receives its activating event. Although conditional activation is useful in many cases, AppComposer also provides *ifTest behaviors,* a more general-purpose mechanism.

An ifTest behavior allows you to embed an explicit *if* test as a behavior. An ifTest behavior consists of a boolean Java expression, which can evaluate to either true or false. When an ifTest behavior receives an activating event, it evaluates its associated expression. If the result is true, it generates an `ifTrue` event; if the result is false, it generates an `ifFalse` event. Remember that all behaviors use events as triggers. You can use the `ifTrue` and `ifFalse` events that an ifTest behavior generates to activate new actions which depend on the outcome of the if test. For instance, you could build an ifTest

expression to evaluate whether a form's field input is empty. The `ifTrue` event could trigger an action that sends the user an error page. The `ifFalse` event could trigger an action that sends the user a welcome page.

Return value behaviors

Return value behaviors allow you to return an object to the caller of the behavior. For instance, if a repeat tag in a JSP page activates a return value behavior, you can have the behavior return a boolean expression to tell the tag whether to repeat or not. Currently you can only return objects, not primitive types.

Script behaviors

A *script behavior* allows you to execute any sequence of Java statements in response to an event.

Building applications out of components presents a huge integration problem. Components from different sources are not often designed to work well together. Script behaviors allow you to use Java code to perform this integration. While this is a powerful feature, note that a heavy use of script behaviors is a sign that you should be obtaining components better suited to your needs.

You can also use a script behavior when you cannot find or do not want to write a component to provide a desired functionality. Again, while this is a powerful feature that allows you to create any application in AppComposer, it is usually better to create a component with the desired functionality. You can create new components directly in AppComposer, or program them using any Java programming environment, such as DigiSlice VisualCafé.

Timeline behaviors

A *timeline behavior* modifies the properties of one or more actors over time, similar to an animation storyboard or a musical score.

A timeline can modify any property of any actor. You specify the values of a given property at certain points along the timeline, and these values change accordingly when the behavior runs. For many types of properties, like numeric values or colors, AppComposer can interpolate between values so that the value changes smoothly as the timeline progresses.

You can set events to deactivate a timeline behavior before it finishes executing.

Counter behaviors

A *counter behavior* also manipulates occurrences over time. A counter behavior counts, starting when it activates, and generates events at intervals until it reaches the specified stopping point.

You can set events to deactivate a counter behavior before it finishes executing.

GLOSSARY

A

action behavior
An AppComposer behavior that receives an event as a stimulus and responds by invoking a method on an actor. See *behavior*.

action group behavior
A convenience that allows you to group several AppComposer behaviors so they can be moved or copied as a unit. See *behavior*.

action target
An actor to which a method is sent. See *actor*. Compare *activation event*.

activate
To send the stimulus to a behavior that causes it to execute. See *behavior, stimulus*.

activation event
An event that activates a behavior. See *behavior, event*. Compare *action target*.

actor
An AppComposer element to which you can assign behaviors; you can use any JavaBean as an actor. Also, an AppComposer alias referring to the first actor above a behavior in the AppComposer capsule hierarchy. See *alias, behavior*.

alias
One of the three AppComposer pseudovariables capsule, parent, or actor, that refer to the specific entity fulfilling that role above the reference entity in the capsule hierarchy. These aliases allow you to define behaviors that you can easily reuse by linking the behavior to a relative, rather than absolute element in the capsule. See *actor, capsule, parent*.

applet
A (typically) small application that is downloaded in a web browser and executes on the web client.

AWT
Abstract Windowing Toolkit, the components used to build a user interface in Java applications.

B

bean

A Java component. See *component*.

behavior

An AppComposer entity having a stimulus and a response, used to cause an actor to do something, to communicate with another actor, or to respond to user input. Because the stimulus is always an event, a behavior is a way to ensure that a given event is followed by the specified method execution, property modification, or other event. See *actor, event, response, stimulus*.

C

capsule

In AppComposer, a means of organizing an application into modules, each having a hierarchy of components. A capsule itself is a component that can be contained in other capsules. Also, an AppComposer alias referring to the first capsule above an entity in the AppComposer capsule hierarchy. See *alias*.

child

An AppComposer entity that is directly beneath another in the capsule hierarchy. See *capsule*.

client

A computer or program that requests resources from a server. For example, a web client (browser) requests web pages from a web server. A program can act as both client and server; for example, a web server acts as a client when it requests data from a database server.

component

A persistent reusable building block of code that works as one functional unit and is can be used in various applications.

connection pool

A group of database connections that remain open for any transactions that need them, so that a transaction can use them without the overhead of opening and closing them explicitly.

cookie — A text file sent by a web server, which the web browser stores on the client and sends back in response to requests from the same web server; used by AppComposer to store a session identifier. See *session, web server*.

counter behavior — An AppComposer behavior that counts time, starting when it receives its stimulus, and generates specified events at specified intervals until it reaches the specified stopping point. See *behavior*.

D

deactivate — To cause a behavior to stop executing.

deploy — To move a file into a production environment where it can be used. Typically this relates to making WAR or JAR files available to a server.

distributed application — An application whose parts run on more than one computer across a network. Compare *applet, servlet*.

E

EJB — Enterprise JavaBean, a robust bean that adds portable access to distributed services such as database access, transactions, and messaging. See *bean*.

event — An occurrence of possible interest to an application, such as a mouse click.

F

form parameter — A name and value pair derived from an input field of a web form.

H

HTML

HyperText Markup Language consists of sets of tags that mark the structure of a web page's contents, thus enabling a web browser to determine how to display the page.

HTTP

HyperText Transfer Protocol, the protocol used by web servers and browsers to request and return web pages and other data. See *web server*.

I

ifTest behavior

An AppComposer behavior with an embedded *if* statement. When an ifTest behavior receives its stimulus, it evaluates the associated *if* test, generating an `ifTrue` event if the result is true, or an `ifFalse` event if the result is false. See *behavior*.

J

JDBC

Java Database Connectivity, an interface that allows Java applications to access the data in a database.

JSP

JavaServer Page, a kind of servlet stored in a web page. See *servlet*.

L

link

A hypertext pointer from one web page to another.

logic layer

The programming code that manipulates the underlying data in a software program.

M

method

A sequence of Java statements attached to a component that execute when invoked, usually by sending the name of the method as a command to an actor. See *actor, component*.

MIME

Multipurpose Internet Mail Extension, a way of identifying the kind of content sent over a network, used by web servers to identify what is being sent to the web browser.

N

non-visual actor

An actor with no visible representation on the computer display. See *actor*. Compare *visual actor*.

O

ODBC

Open DataBase Connectivity, an interface that allows Java applications to access the data in a database.

P

parent

An AppComposer entity that is directly above another in the capsule hierarchy. Also, an AppComposer alias referring to the first entity above this behavior or actor in the AppComposer capsule hierarchy. See *actor*, *alias*, *capsule*.

R

response

The method that executes, or the property modification, or the event that is generated, when the stimulus of a behavior is received. See *behavior*. Compare *stimulus*.

return value behavior

Return value behaviors return an object (from an expression or script) to the behavior's caller.

S

saved group

A named, saved, reusable AppComposer behavior. AppComposer allows you to create, save, and reuse behaviors, and makes them available from the Saved Groups menu. See *behavior*.

script An sequence of Java statements.

script behavior An AppComposer behavior that allows the execution of any chunk of Java code in response to an event. See *behavior*.

serialize To turn an executable entity such as a component into data that can be sent across a network or stored. See *component*.

server A computer or program satisfying the requests of a client; for example, a web server offering resources over the World Wide Web. See *web server*. Compare *client*.

servlet An application that runs on a web server and constructs appropriate responses to requests as they stream in from web browsers. See *web server*. Compare *client*.

ServletGet A predefined saved group that ships with AppComposer, which enables servlets to respond to GET requests from a browser.

session A means by which a web server can identify multiple requests as coming from the same web browser. See *client, web server*.

stimulus The event that triggers a behavior, causing its response to execute. See *behavior*. Compare *response*.

T

timeline behavior An AppComposer behavior that modifies the properties of one or more actors over time. See *behavior*.

toybox The AppComposer view that lets you see your application run as you create and modify it.

U

URL A Uniform Resource Locator is used to specify resources on the World Wide Web, such as web pages and servlets. See *HTTP, web server*.

user interface — Means by which users interact with an application—commonly, with windows, buttons, and menus.

V

view — A view allows you to see, and often to edit, a specific object. For example, AppComposer allows you to view and edit a capsule in several ways.

visual actor — An actor with a visible representation on the computer display. See *actor*. Compare *non-visual actor*.

W

watch variable — A variable whose value is visible in the debugger, so that you can watch it change.

web server — A computer more or less permanently connected to the Internet that offers to web users a set of resources—text, images, music, or other downloadable files.

INDEX

A

Abstract Windowing Toolkit
  defined 89
accessing
  database 71
action behaviors
  about 30
  defined 89
action group behaviors
  about 36
  defined 89
action targets
  defined 89
activating
  behaviors 24
activation
  order of behaviors 42
activation event
  defined 89
actor alias 35
actors
  about 23
  defined 89
  HTML 83
  SQL 84
  visual 26
aliases
  about 35
  defined 89
applets
  capsule type 26
  defined 89
applications
  capsule type 25
architecture
  Application Composer's 14
AWT
  defined 89

B

beans
  defined 90
Behavior.activated events 36
behaviors
  about 23
  action 30
  action group 36
  activating 24
  counter 88
  defined 90
  execution order 42
  return value 87
  saved 38
  script 87
  timeline 87

C

capsule alias 35

capsules
- about 25
- applet type 26
- application type 25
- defined 90
- hierarchy 42
- JSP bean type 26
- non-visual actor type 27
- servlet type 26
- visual actor type 26

child
- defined 90

client
- defined 90

components
- about 9
- AWT 12
- defined 90
- vs. objects 11

conditional
- JSP tag 68

connection pools 90

cookies
- defined 91

counters
- about 88
- defined 91

creating
- pizza 21

D

databases
- accessing 71

deactivating behaviors 24

debugging
- capsules 44

deployment
- diagram 15

E

EJBs
- about 18
- transaction boundaries 27
- using 77

events
- Behavior.activated 36
- ifFalse 86
- ifTrue 86
- in Java 17

execution
- order of behaviors 42

expressions
- using 40

F

flow of control 24

form parameter
- defined 91

H

HTML
- actors 83

HTTP
- defined 92

I

ifFalse events 86

ifTrue events 86

J

J2EE 50
Java
  using expressions 40
JavaBean
  described 17
JavaBeans 10
  about 17
JDBC 72
JSP bean
  capsule type 26
JSPs
  conditional tag 68
  repeat tag 67
  using 63

M

message-driven bean capsules 27
methods
  about Java 17
  defined 92
  executing 23
MIME
  defined 93
Multipurpose Internet Mail Extension
  defined 93

N

non-visual actors
  capsule type 27
  defined 93

O

objects
  proxied 79
  vs. components 11
ODBC
  defined 93

P

parent
  defined 93
parent alias 35
proxied objects 79

R

repeat
  JSP tag 67
response
  defined 93
return value behaviors 87
running
  capsules 44

S

saved behaviors
  about 38
  defined 93
script behaviors
  defined 94
scripts
  about behavior 87
  defined 94
  using Java for 40
serialize
  defined 94

ServletGet
  defined 94
servlets
  about 28
  capsule type 26
  defined 94
sessions
  defined 94
  using HTTP 76
source code 45
SQL
  accessing database 71
  actors 84
SQLSelect 72
SQLUpdate 74
stateful session EJB
  capsule type 27
stateful session synchronization EJB
  capsule type 27
stateless session EJB
  capsule type 27
stimulus
  defined 94

T

timeline behaviors
  defined 94
timelines
  about behavior 87
transaction boundaries 27

U

URL
  defined 94

V

visual actors
  capsule type 26
  defined 95

W

watch variables
  defined 95
web applications
  about 47
web servers
  defined 95