Borland®

Migrating Projects to the Borland® JBuilder™ Development Environment

By Axel Kratel, JBuilder Product Manager

Borland Software Corporation

September 2002

Contents

Introduction

The Borland® JBuilder™ development environment is equipped with very powerful project import capabilities that make it easy to import most standard Java™ code and render it in the form of a JBuilder project, complete with EJB™ modules and JBuilder Web applications. In addition to these fundamental capabilities, JBuilder now also provides the ability to automatically migrate WebGain VisualCafé™ projects into the JBuilder environment. This paper shows you how to take advantage of this new functionality as well as the existing import capabilities already available in JBuilder.

If you are reading this paper, then you most likely have some code you would like to bring into JBuilder, and you are eager to get productive. Thankfully, JBuilder makes that process relatively simple by providing a VisualCafé import project wizard that can chew through just about any code and spit out a nice JBuilder project. This paper shows you where to download and how to install and use the JBuilder open tool that extends JBuilder to provide the VisualCafé import capabilities.

JBuilder™

white paper

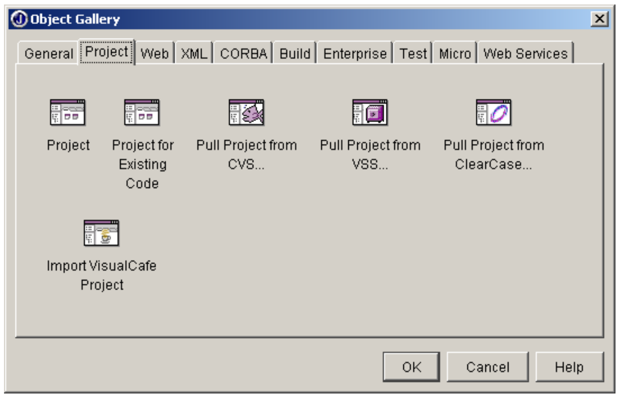
Of course, under the hood, this open tool actually takes advantage of an already existing infrastructure for importing just about any code into JBuilder even if there is no VisualCafé project file to parse. So we'll also show you how to import your favorite open-source project or sample application right into JBuilder.

Of course, it helps to have a few hints and tips on how to take best advantage of these powerful features, so this paper walks you through some basic concepts you need to know before you get started. We'll then tackle a few specific examples, including importing a simple SWING-based application, the Stylepad from the Java JDK,® importing JPetStore, an open source, servlet and JSP™-based version of the Sun® Microsystems Java Pet Store sample application. Finally, we'll tackle importing an EJB application sample from the WebLogic application server.

Basic preparations

To take advantage of the VisualCafé import capabilities, you need to download and install the VisualCafé import open tool update. The update includes both the latest JBuilder update and the actual VisualCafé import open tool. The update can be downloaded from the JBuilder downloads page at http://www.borland.com/products/downloads/download_jbuilder.html. Follow the instructions in each respective download in order to update your JBuilder 7 installation.

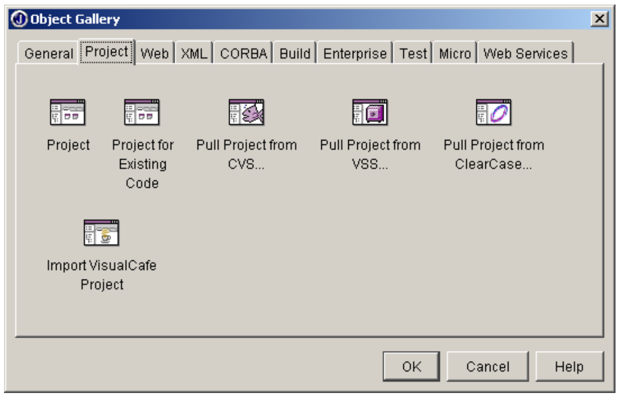
Make sure you close JBuilder before doing any installations. After the installations are complete, restart JBuilder. You can see whether the VisualCafé import open tool is installed by opening the object gallery (**File|New**) and clicking the **Project** tab. You should see "Import VisualCafé Projects" as one of the objects in the gallery.



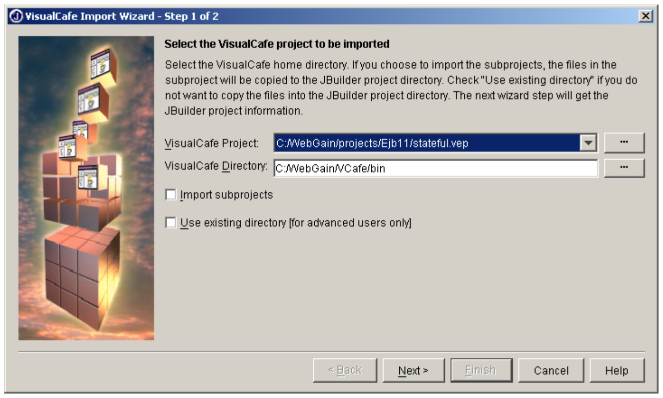
You should now be ready to import VisualCafé projects.

Importing VisualCafé™ projects into JBuilder™

The whole process of importing a VisualCafé project is relatively seamless. Select the VisualCafé import wizard from the object gallery: **File|New** menu path, select the **Project** tab, and select the **Import VisualCafé Project** icon.



This will bring up the VisualCafé import wizard.


VisualCafe Import Wizard - Step 1 of 2

Select the VisualCafe project to be imported

Select the VisualCafe home directory. If you choose to import the subprojects, the files in the subproject will be copied to the JBuilder project directory. Check "Use existing directory" if you do not want to copy the files into the JBuilder project directory. The next wizard step will get the JBuilder project information.

VisualCafe Project: C:/WebGain/projects/Ejb11/stateful.vep

VisualCafe Directory: C:/WebGain/VCafe/bin

☐ Import subprojects

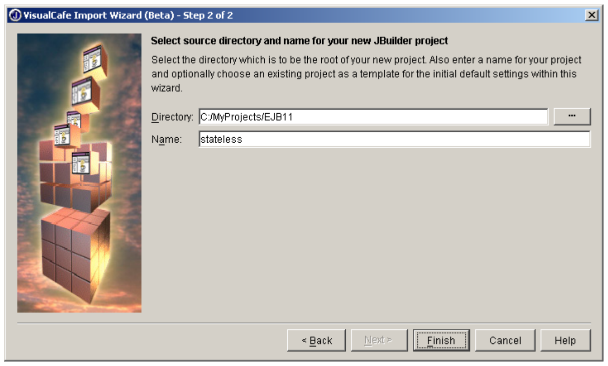
☐ Use existing directory [for advanced users only]

Note that the wizard will recall the last entries you have entered. You now need to enter the VisualCafé project file you want to import. Look for any *.vep files. It is recommended that you also enter the directory where VisualCafé is installed, because VisualCafé stores some global path settings in the vc.ini file found in the VisualCafé bin directory. But this step is not mandatory. Thus if you have a VisualCafé project without having a VisualCafé installation on your workstation, you can still import the project, assuming, of course, that you copied all the necessary source code associated with the project to your workstation.

JBuilder optionally will also import any subprojects associated with your project. You can select the import subproject button to do this. Often, subprojects might contain a client part of a J2EE™ application. In JBuilder, you can put both the client and the server code into one single directory.

You also have the option to have JBuilder create the JBuilder project right in the existing directory, but this option is only recommended for advanced users. The default action is to copy all the necessary source files from the VisualCafé project directory to your new JBuilder project directory. This is the recommended approach because it will copy all the source and related files into a directory structure that is compatible with JBuilder. However, it is possible to leave the source code where it is; JBuilder is flexible enough to work with source code no matter where it is stored on your hard drive. However, if you choose the latter option, be prepared to make some manual modifications.

Once you have selected the appropriate VisualCafé project to import, click **Next** to bring up the second and final step in the wizard.


VisualCafe Import Wizard (Beta) - Step 2 of 2

Select source directory and name for your new JBuilder project

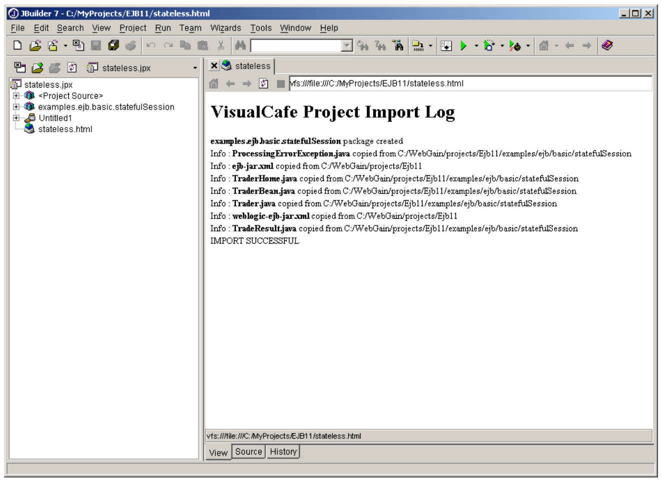
Select the directory which is to be the root of your new project. Also enter a name for your project and optionally choose an existing project as a template for the initial default settings within this wizard.

Directory: C:/MyProjects/EJB11

Name: stateless

Simply enter the location of the new JBuilder project you want to create; give the project a **name** and click **Finish**.

JBuilder will now create the entire project for you. The project at this point is completely imported into JBuilder. In the process, JBuilder will have created an import log in the form of an HTML file and included that file in the project. JBuilder also automatically creates an EJB module and a Web application node in the project if either one of these were part of the original source code. A runtime is also added so that you can easily run the project. Note that both the Web application and the EJB module are named untitled1. You will need to rename them to whatever you like. The name has no bearing on the source code.


JBuilder 7 - C:/MyProjects/EJB11/stateless.html
File Edit Search View Project Run Team Wizards Tools Window Help
stateless.jpx
stateless.jpx
<Project Source>
examples.ejb.basic.statefulSession
Untitled1
stateless.html
stateless
vfs:///file:///C:/MyProjects/EJB11/stateless.html
View Source History

VisualCafe Project Import Log

```
examples.ejb.basic.statefulSession package created
Info : ProcessingErrorException.java copied from C:/WebGain/projects/Ejb11/examples/ejb/basic/statefulSession
Info : ejb-jar.xml copied from C:/WebGain/projects/Ejb11
Info : TraderHome.java copied from C:/WebGain/projects/Ejb11/examples/ejb/basic/statefulSession
Info : TraderBean.java copied from C:/WebGain/projects/Ejb11/examples/ejb/basic/statefulSession
Info : Trader.java copied from C:/WebGain/projects/Ejb11/examples/ejb/basic/statefulSession
Info : weblogic-ejb-jar.xml copied from C:/WebGain/projects/Ejb11
Info : TradeResult.java copied from C:/WebGain/projects/Ejb11/examples/ejb/basic/statefulSession
IMPORT SUCCESSFUL
```

Your project is ready to use. It's a good idea to verify that the project compiles and runs. If it compiled and ran in VisualCafé, then it should compile and run in JBuilder.

Some things you should know

Generally, the biggest challenge in importing any project into JBuilder is to make sure the classpaths are configured properly. In VisualCafé, developers define classpaths that include the necessary jars that contain libraries to build and to deploy. In JBuilder, the proper JARs are included in libraries defined in the IDE. In most cases, if a project won't compile or run, it's because a library is not set up correctly.

The import wizard creates the necessary libraries based on the information it obtains from parsing the VisualCafé project file. In addition, the vc.ini file in the VisualCafé bin directory also contains global classpath settings that might be needed for your project. If you defined the VisualCafé directory during the import process, JBuilder will then also extract any necessary jar from the global classpath settings. After importing your project, you will need to verify that the libraries are indeed set up properly and that there are no redundant libraries. Redundant libraries can get defined if your VisualCafé project had specific classpath settings that also included common build and runtime libraries already set up by default in JBuilder.

Also, if you copied the VisualCafé project from another system, some of the original jars needed for the project may no longer be in the locations described in the original VisualCafé project file. JBuilder will still create the libraries that correspond to those jars, but you will need to manually copy them into the right location or edit the libraries to point to the appropriate jars.

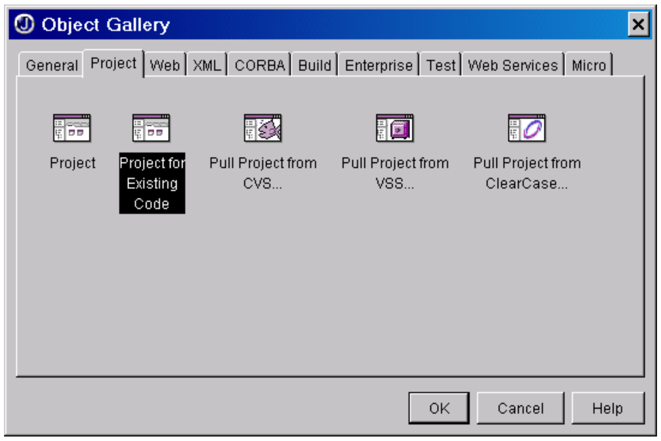
JBuilder also assumes that you are working with BEA WebLogic Server,™ as that is the only application server supported by VisualCafé. After importing your project, you will also need to verify that the application server configuration is set up properly for your new JBuilder project. You will also need to bring up the WebLogic console and properly configure the database connection pool and other runtime parameters that your project will use. Please consult the JBuilder documentation for more details on how to work with the WebLogic Server.

Importing general projects into JBuilder™

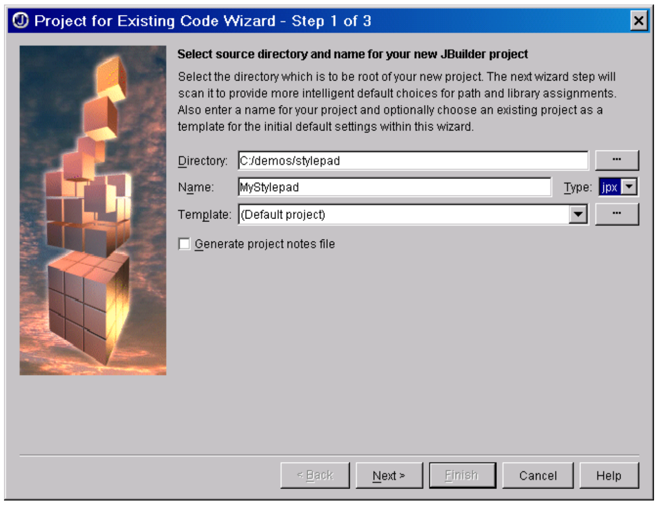
The VisualCafé import utility actually makes use of a fundamental layer of import functionality already available in JBuilder. This functionality can actually be employed to import just about any source code into JBuilder regardless of what IDE was originally used to create the source code. This functionality is available via the project from existing code wizard, also known as the project import wizard.

The project import wizard was designed to make the whole process of importing a project as easy as possible with the least amount of work. Unlike the project import features found in other IDEs, the project import wizard in JBuilder will actually auto-discover any source code, libraries, Web applications, and EJB applications buried in a subdirectory. All you have to do is to point the wizard to the actual directory of your choice that contains the relevant files and subdirectories, and let JBuilder do the rest.

To get to the project import wizard, just open the object gallery, (**File | New** …) select the **Project** tab and double-click the **Project for Existing Code** icon.



The first step of the wizard asks you to enter a directory. This is the directory that contains all the files and subdirectories of your project. You don't want to select just the directory that contains the source. Rather, select the directory that contains every subdirectory and file relevant to the project you want to import. Keep in mind that in some cases, you might have something slightly too big for JBuilder so swallow, and you will need to break up your project into more manageable morsels. It's not the amount of code that's at issue. Rather, if your directory contains shared code, an EJB JAR, and a mobile application, then you will need to create two separate projects—one for your EJB tier and one for the mobile application (this topic is discussed in greater detail in the following section). Let's assume that you have a manageable project in the directory you select. On the same dialog, don't forget to **name your project** as well. It can be any name of your choice. This will be the name of the .jpx file that JBuilder will create. When you're done, click **Next**.

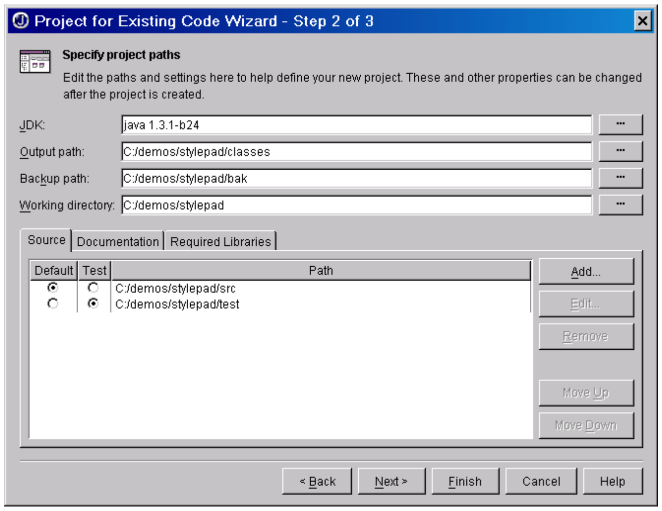


Select source directory and name for your new JBuilder project

Select the directory which is to be root of your new project. The next wizard step will scan it to provide more intelligent default choices for path and library assignments. Also enter a name for your project and optionally choose an existing project as a template for the initial default settings within this wizard.

Directory: C:/demos/stylepad
Name: MyStylepad Type: jpx
Template: (Default project)
☐ Generate project notes file

Next, JBuilder will ask you to verify the project data it auto-discovered. Make sure that the information is correct. Note that this dialog doesn't show the actual EJB and Web applications it discovered. These get set up automatically in the project. Also note that JBuilder automatically adds a library entry for your project. This entry will contain any jars found in lib directories. Once you've verified the information, click **Next**.



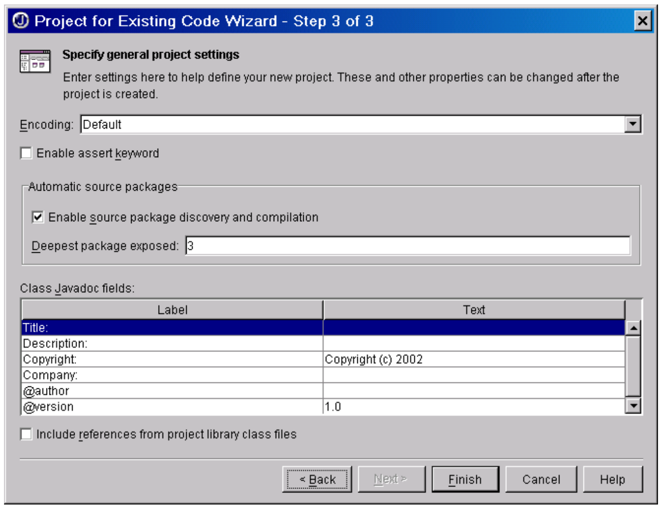
Specify project paths

Edit the paths and settings here to help define your new project. These and other properties can be changed after the project is created.

JDK: java 1.3.1-b24
Output path: C:/demos/stylepad/classes
Backup path: C:/demos/stylepad/bak
Working directory: C:/demos/stylepad

Default	Test	Path
●	○	C:/demos/stylepad/src
○	●	C:/demos/stylepad/test

Finally, set up any additional project settings you wish on the last dialog of the wizard, and click **Finish**.



Project for Existing Code Wizard - Step 3 of 3

Specify general project settings

Enter settings here to help define your new project. These and other properties can be changed after the project is created.

Encoding: Default

☐ Enable assert keyword

Automatic source packages

☑ Enable source package discovery and compilation

Deepest package exposed: 3

Class Javadoc fields:

Label	Text
Title:	
Description:	
Copyright:	Copyright (c) 2002
Company:	
@author	
@version	1.0

☐ Include references from project library class files

Once JBuilder is done setting up the project, you need to look through the project and make sure everything imported the way it was supposed to. You may need to do some additional configuration such as selecting the appropriate application server and adding specific runtime configurations to run different parts of the project (such as a runner to launch the server) or to run a SWING client application. If your project does include a SWING application, then you may also need to make some basic changes to any SWING code so that you can pull the application into the JBuilder GUI designer (this topic is not covered in this paper; for more information, visit the Borland Developer Network Web site at http://bdn.borland.com). The next sections walk through the technical details on what to do to prepare the project for import, and they give some detailed examples, including applications that use servlets, JSP, and EJB components.

The basics

Before you get started and launch the project import wizard, there are a few things you need to know so you can properly prepare your project for import. The basic two areas you need to pay attention to are the directory structure of the project and the libraries it uses. Once you understand what JBuilder expects to see, it will become relatively straightforward to import many projects by making some simple directory structure changes before running the project import wizard.

Let's consider the most complex scenario and assume that we have a project with an EJB tier, a Web-based client, and a SWING-based client to access the EJB tier. So let's take a look at all the files and directories that should floating around to make up the project.

Source files: The first order of business is to properly import the actual source files. JBuilder essentially looks for any subdirectory named "src" under the main project directory and assumes that each of these "src" directories are the top node of individual packages. If you have multiple "src" directories, and these do not all correspond to packages—or, in the unusual instance that one of your packages is actually called "src"—you will be given a chance to eliminate individual package nodes during the import process. If no "src" subdirectory is present, then JBuilder will simply import all the code that is in the main project directory and put it in an unnamed package. In general, though, the cleanest approach is to place all the source code in a directory called "src." Be careful to make sure the package structure matches what is in any deployment descriptors or Web XML configuration files.

EJB deployment descriptors: In order for JBuilder to be able to incorporate your EJB code as part of an EJB module, it will need to be able to find the actual deployment descriptors. JBuilder doesn't care where those files are stored, as long as they are somewhere underneath the actual project root directory. Of course, we recommend a clean and elegant approach to place all the EJB deployment files into a folder called META-INF under the main project root directory. Once the project is done importing, you can actually modify the properties of the newly created EJB module and make sure the EJB designer synchs up with the deployment descriptors in the META-INF directory. More on how to do this is outlined in a later section of this paper.

Web application files and directories: Each Web application is typically associated with a Web application context root directory. This directory maps to a URL context defined for the

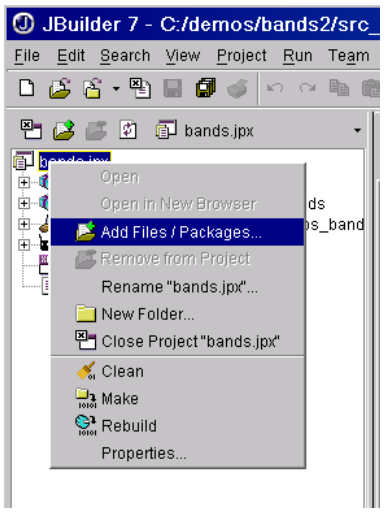
Web application. Thus, if the URL context is localhost/myapp and the directory is c:\myprojects, then an index.jsp file placed in the locahost/myapp directory will become accessible under the URL http://localhost/myapp/index.jsp. In the context root directory, there needs to be a WEB-INF directory that contains all the files that describe how the Web application behaves. The servlet container will make this directory invisible so that it cannot be accessed via a URL. The main configuration file in that directory is Web.xml, but there may be other files depending on the servlet container and frameworks you are using. This directory is often also where source, tag libraries, and general libraries can optionally be placed. All other files and directories under the context root directory should be HTML or JSP files that make up the Web application.

Ant build files: If your project includes an external build file, then JBuilder will pull it in automatically. It does not matter where it is located in the directory structure. However, note that if you are copying any files, you may need to edit the build.xml file to update any absolute paths. Otherwise, your build.xml file might be broken.

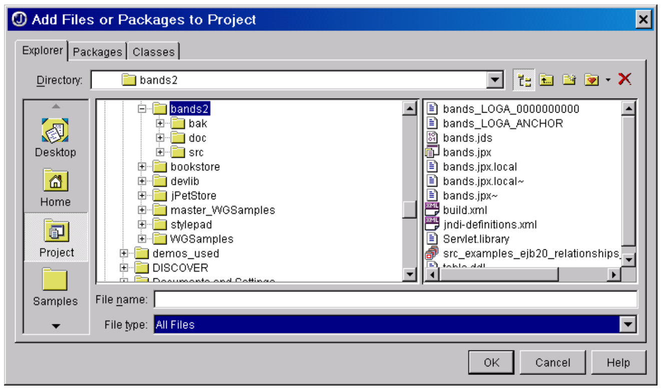
Javadoc: Your project might include Javadoc. JBuilder can import the Javadoc files, but do note that JBuilder expects to see the Javadoc in a separate directory that has the same package structure as the source.

Other files: JBuilder will ignore any other files, such as database SQL scripts, outside notes, and non-Javadoc documentation—basically any files not directly associated with the Java platform. But note that any file can simply be added to the JBuilder project manually on an individual basis. The advantage of adding non-Java source files to the project is that they can then readily be edited from within JBuilder, provided they are ASCII format files.

To add any non-Java source ASCII file, right-click the main project node, and select **Add Files/Packages**.



In the resulting file dialog, make sure to set the file type to **All files**; otherwise, the non-Java files will be hidden from the dialog.



Directory: bands2

bands2, bak, doc, src, bookstore, devlib, jPetStore, master_WGSamples, stylepad, WGSamples, demos_used, DISCOVER

bands_LOGA_0000000000, bands_LOGA_ANCHOR, bands.jds, bands.jpx, bands.jpx.local, bands.jpx.local~, bands.jpx~, build.xml, jndi-definitions.xml, Servlet.library, src_examples_ejb20_relationships

File name:

File type: All Files

Select the file you want to add, and click **OK**. The file will then be accessible from within JBuilder so that you can make changes to it.

What you should know about directories

JBuilder is actually quite flexible, and there are very few restrictions on what you can do. One restriction is that if your project includes a Web application, then the WEB-INF/lib and WEB-INF/classes directories should not contain any files that cannot be regenerated during the build process. Another restriction is that all files and directories related to the project need to be under the project root directory. Of course, any jar files that need to be included need not be stored under the project root directory as long as they get set up as a library in JBuilder and added to the project.

However, if you take the time to make some simple preparations in order to provide JBuilder with a clean project directory hierarchy, the import process will work a lot smoother. Assume our project directory is called myproject, and let's walk through some best practices to provide optimal results. All the source code should be best placed in myproject/src. You will need to make sure that you preserve the full package hierarchy under src. Next, if you have any Javadoc, place it in a directory called myproject/doc. Any jars that are needed as libraries for the project should be placed in a directory called myproject/lib. If you have any EJB deployment descriptors, place them in a directory called myproject/META-INF. Finally, if you have a Web application, make sure you have a WEB-INF directory under the root context directory with all the appropriate descriptors and tag libraries. The root context directory can be any name as long as it is somewhere under the myproject directory.

You should now be able to seamlessly import the project into JBuilder and be up and running in a matter of minutes. Of course, you don't have to follow the directory structure outlined above. During the import process, JBuilder will give you the opportunity to custom-define the source and documentation directory. So for example, if you have a Web application, and you distribute the source in an src directory under the WEB-INF directory, you can simply point JBuilder to that src directory location during the import process.

When to divide into subprojects

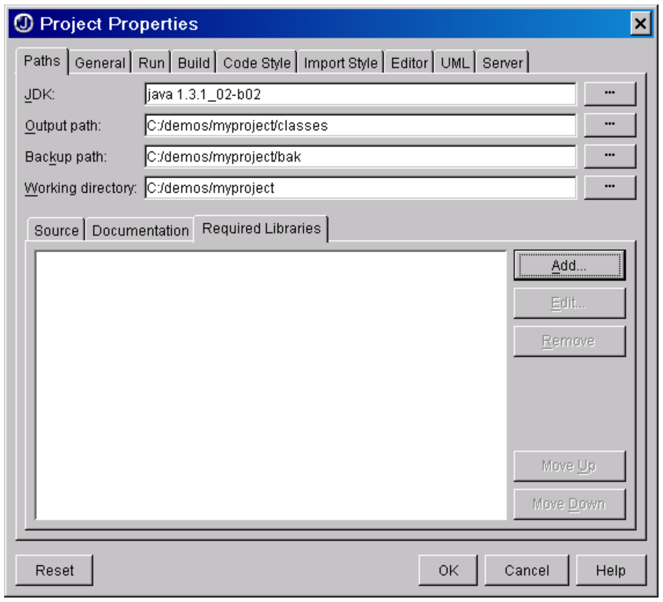
If you have a very large application with multiple tiers, the next obvious question you may ask is when to subdivide the project into distinct JBuilder projects. The constraint to keep in mind is that JBuilder expects you to use a single application server configuration per project and a single JDK per project. Thus, if you have a mobile tier and a Web tier in your project, you will need to create two distinct projects in JBuilder (since the mobile tier must be based on the Sun J2ME™ SDK, and the Web tier will be based on a standard JDK).

However, you can still use only a single project if you have multiple tiers that include Web or EJB components, as long as the components use the same application server configuration. The application server configuration for a single project can be set so that you use one server type for EJB tiers and another server type for Web tiers. But each EJB tier in that project will then need to use the same EJB container, and each Web tier will need to use the same servlet container. Therefore, if your application for example includes a Web tier that runs on Tomcat, and another Web tier that runs on WebLogic, you will need to set up two distinct projects for each Web tier.

Additional configurations

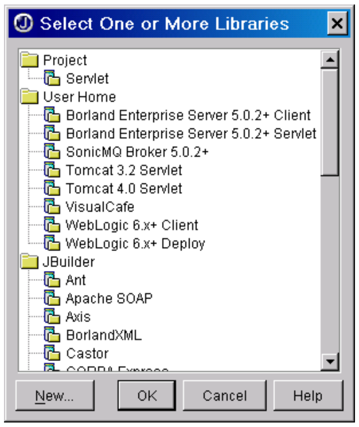
Once you have imported your project, you will need to take three more steps to complete the configuration for the project: set up any libraries that are not under the project root directory, set up the application server configuration, and set up any runtime configurations.

Since JBuilder has no way of knowing about any dependencies on external libraries that are not included under the project root directory, you will need to tell JBuilder about these libraries. You can do that by selecting the **project properties** under the project menu. In the resulting dialog box, select the **required libraries** tab.


Project Properties
Paths | General | Run | Build | Code Style | Import Style | Editor | UML | Server
Source | Documentation | Required Libraries
Add... Edit... Remove Move Up Move Down
Reset OK Cancel Help

JDK: java 1.3.1_02-b02
Output path: C:/demos/myproject/classes
Backup path: C:/demos/myproject/bak
Working directory: C:/demos/myproject

Then, click **Add…** and select the right library.

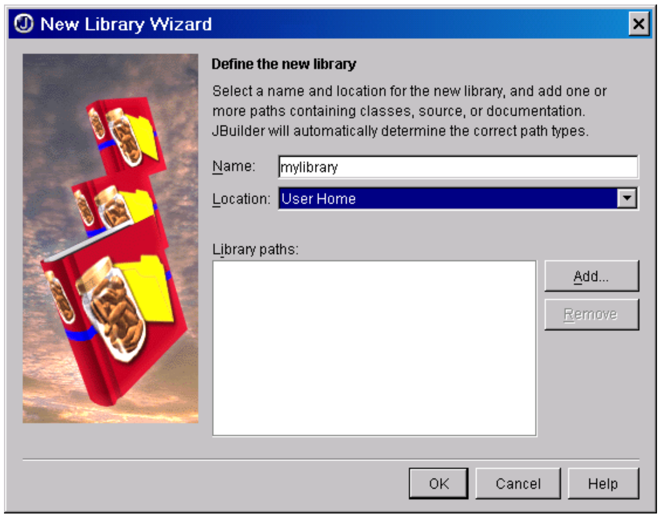


Project
- Servlet

User Home
- Borland Enterprise Server 5.0.2+ Client
- Borland Enterprise Server 5.0.2+ Servlet
- SonicMQ Broker 5.0.2+
- Tomcat 3.2 Servlet
- Tomcat 4.0 Servlet
- VisualCafe
- WebLogic 6.x+ Client
- WebLogic 6.x+ Deploy

JBuilder
- Ant
- Apache SOAP
- Axis
- BorlandXML
- Castor

Most likely, your project will require some special jar files not distributed with JBuilder. In this case, you will need to create a new library for your project. Just click **New…** and create a new library for your project. This will trigger the New Library Wizard.



Define the new library

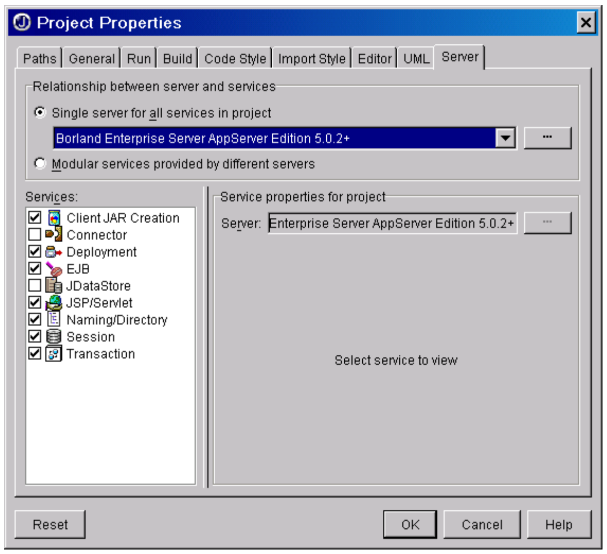
Select a name and location for the new library, and add one or more paths containing classes, source, or documentation. JBuilder will automatically determine the correct path types.

Name: mylibrary
Location: User Home
Library paths:

In the wizard, select the **name** you wish to give to your library. It can be anything that will help you remember what is in that library. Your code won't care about the library name; only the library paths matter. Also select the **location** in JBuilder. If you select Project, the library will be available only for that specific project. User Home will make sure it is available for you to use with other projects. JBuilder means it will be available for all users of JBuilder on that particular machine.

Click **Add…** to add each directory, jar, or class file, and select **OK**. In the following dialog, simply select the library you just created and click **OK** to add it to the product.

Next, you will need to configure your project to use your chosen application server. The assumption here is that you have already set up JBuilder (Tools|Configure Servers…) to work with your application server. The only thing left to do is to configure the application server setup for your project by selecting the menu path **Project|Project Properties** and selecting the **Server** tab.


Project Properties
Paths | General | Run | Build | Code Style | Import Style | Editor | UML | Server
Reset
OK
Cancel
Help

Relationship between server and services

[✓ Single server for all services in project]: Borland Enterprise Server AppServer Edition 5.0.2+

Modular services provided by different servers

Services:

- [✓] Client JAR Creation
- [ ] Connector
- [✓] Deployment
- [✓] EJB
- [ ] JDataStore
- [✓] JSP/Servlet
- [✓] Naming/Directory
- [✓] Session
- [✓] Transaction

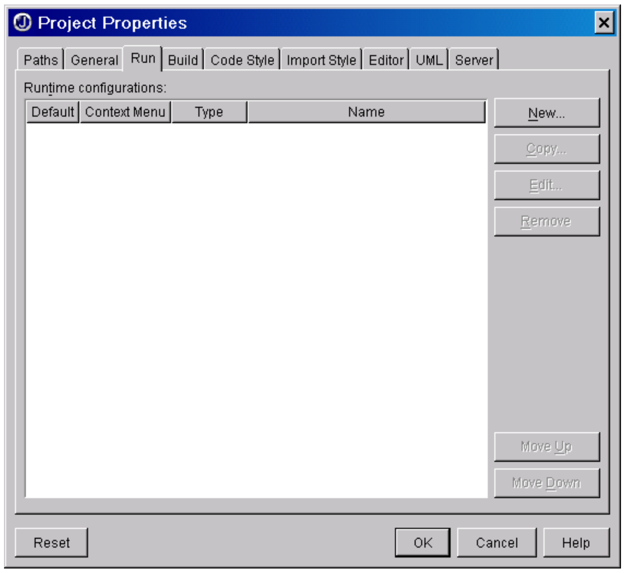
Service properties for project

Server: Enterprise Server AppServer Edition 5.0.2+

Select service to view

You can use a single server for all services in the project, or you can choose modular services. The latter is useful, for example, if you use one application server as an EJB container and a separate application server as a servlet container.

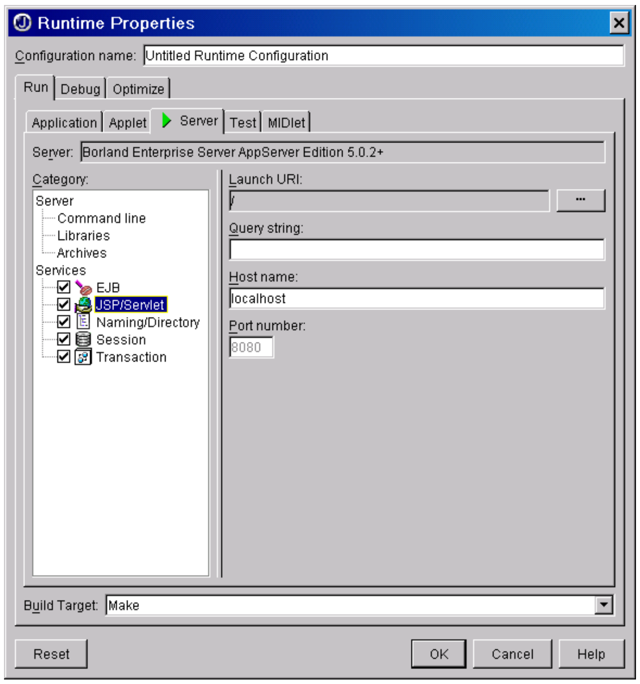
Finally, we need to create the appropriate run configurations to run the various tiers in our application. Select the **Run** tab,


Project Properties
Paths | General | Run | Build | Code Style | Import Style | Editor | UML | Server
New...
Copy...
Edit...
Remove
Move Up
Move Down
Reset
OK
Cancel
Help

Runtime configurations:

Default	Context Menu	Type	Name

and click **New…** to create a runtime configuration.



Configuration name: Untitled Runtime Configuration

Server: Borland Enterprise Server AppServer Edition 5.0.2+

Category:

- Server
  - Command line
  - Libraries
  - Archives
- Services
  - [✓] EJB
  - [✓] JSP/Servlet
  - [✓] Naming/Directory
  - [✓] Session
  - [✓] Transaction

Launch URI: /

Query string:

Host name: localhost

Port number: 8080

Build Target: Make

For tiers that use the server, select the **Server** tab. Click on the **JSP/Servlet** service in the category window, then, click **Command line** under Server in the Category window. On the right-hand side, you will then be able to select the **URI** to launch at runtime. Then, click **Archives** in the Category window and make sure the proper JARs and WARs are selected to be deployed to the server. Give the runtime configuration a **name** and click **OK**.

For SWING-based applications, select the **Application** tab and enter the **class** to run. If you have JUnit-based test suites included in the project, use the Test tab and configure the appropriate fields. All of the runtimes you have configured are now available in the project to run, debug, and optimize your code base.

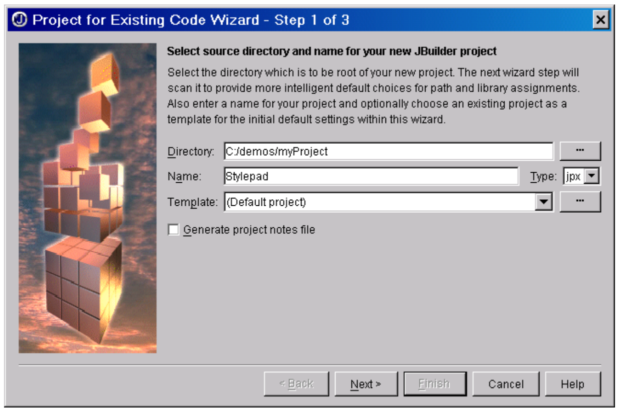
If you want to take full advantage of the Borland platform, use JDataStore™ as the database to power your development process. Why install a full instance of a large database such as Oracle® or Microsoft® SQL Server™ on your development workstation if you can use a high-performance, database—

written entirely in Java—with a tiny footprint? To migrate your application to the JDataStore database, consult the appendix.

Of course, an excellent way to learn how to truly leverage the flexibility of JBuilder is to do some hands-on examples. The next sections describe in greater detail how to import projects by looking at specific examples of various flavors, starting from a simple SWING application all the way to a multi-tier J2EE application.

Example: importing Stylepad

In this example, we take the Stylepad application from the Sun JDK and readily import it into JBuilder. We start by copying all the content of `<jdk1.3.1>\demo\jfc\Stylepad` to a directory called myproject. We then start the import wizard from the object gallery (**File | New**, select the **Project** tab on the object gallery, and double-click the **Import From Existing Code** icon).


Project for Existing Code Wizard - Step 1 of 3

Select source directory and name for your new JBuilder project

Select the directory which is to be root of your new project. The next wizard step will scan it to provide more intelligent default choices for path and library assignments. Also enter a name for your project and optionally choose an existing project as a template for the initial default settings within this wizard.

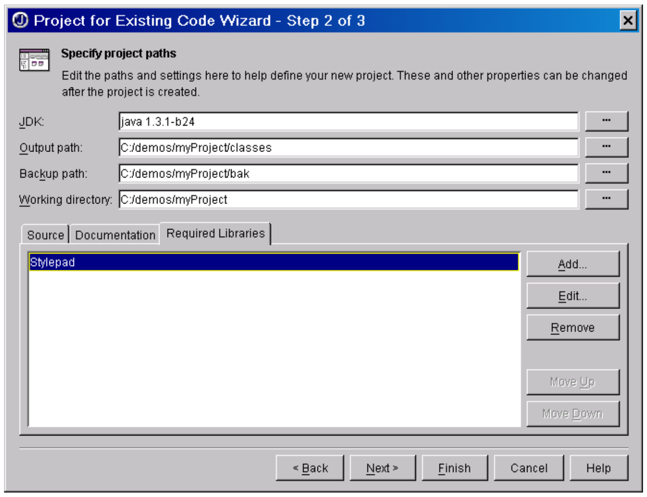
Directory: C:/demos/myProject

Name: Stylepad Type: jpx

Template: (Default project)

☐ Generate project notes file

Point the wizard to the **myProject** directory in which you put the stylepad files, and name the project **Stylepad**. Click **Next**. Verify that the project paths are correct: the source should point to the directory **myProject/src>** Then, click the required **Libraries** tab to edit the library that JBuilder added for just this project.


Project for Existing Code Wizard - Step 2 of 3

Specify project paths

Edit the paths and settings here to help define your new project. These and other properties can be changed after the project is created.

JDK: java 1.3.1-b24

Output path: C:/demos/myProject/classes

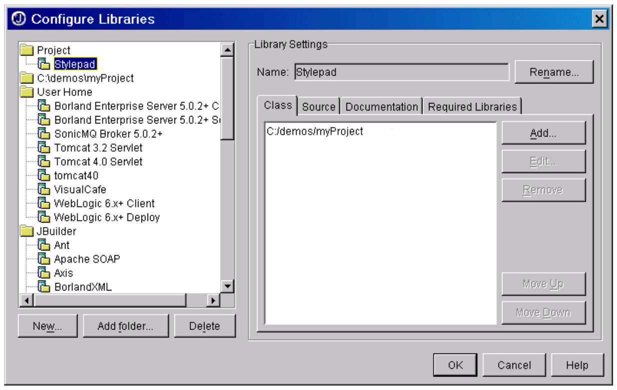
Backup path: C:/demos/myProject/bak

Working directory: C:/demos/myProject

Source | Documentation | Required Libraries

Stylepad

Highlight the **Stylepad** library and click the **Edit** button. The library JBuilder created is initially blank, but we know that the resources directory that came with the Stylepad contains files that the Stylepad application needs to be able to access, so you will need to add the project root directory to the library. Add the **myProject** directory and click **OK** to save.


Configure Libraries

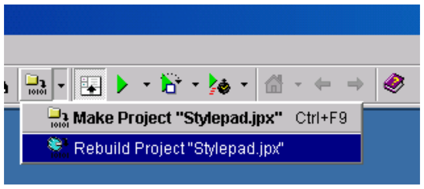
Library Settings

Name: Stylepad

Class | Source | Documentation | Required Libraries

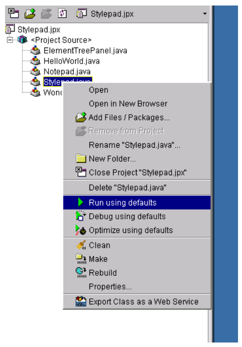
C:/demos/myProject

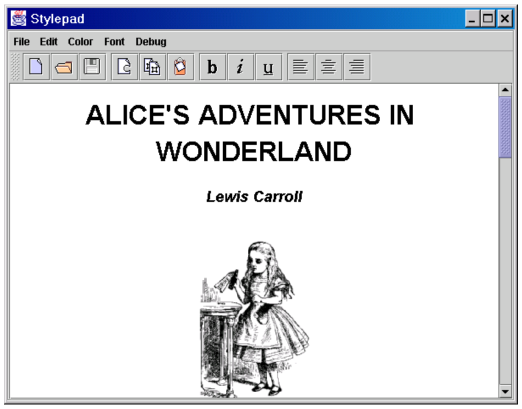
On the project import, click **Finish** to finish the import. You now should be able to build and run the application. Select the **Build** dropdown above the content pane and click **Rebuild Project.**



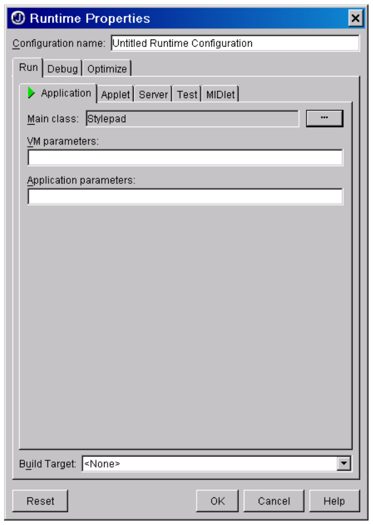
Make Project "Stylepad.jpx" Ctrl+F9
Rebuild Project "Stylepad.jpx"

You are now ready to run the Stylepad application. Open the source code tree, right-click the **Stylepad** class and select **Run using defaults**. The Stylepad should come up.





Note that you can also run the application by setting up a runtime configuration. Select the menu item **Run | Configurations**, and click **New**.



Configuration name: Untitled Runtime Configuration

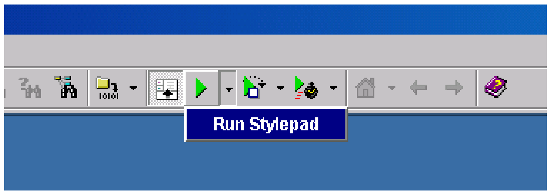
Main class: Stylepad

VM parameters:

Application parameters:

Build Target: <None>

Select the **Application** tab and set the main class to **Stylepad**. Set the **Build target** to **none** if you do not want JBuilder to rebuild the project each time you run the application. Give the runtime configuration a **name**, run Stylepad, for example, and click **OK**. Click **OK** on the subsequent dialog. You should now be able to access the runtime configuration from the dropdown above the content pane.



Note that if you want to be able to edit the code visually in the SWING designer of JBuilder, you will need to make a few changes to the code itself. The JBuilder designer assumes that all object initializations are placed in a method called JBinit. How to do this is described in more details in the appendix.

Example: importing JPetStore

Let's consider another example that uses a Web application and requires additional libraries to support the Struts framework it uses. We will show how easy it is to import JPetStore into JBuilder.

First, download the actual source code from http://www.ibatis.com/. At the time of this writing, the download was called jpetstore-1.0.zip. Unzip the contents of the zip file into a directory called myProject.

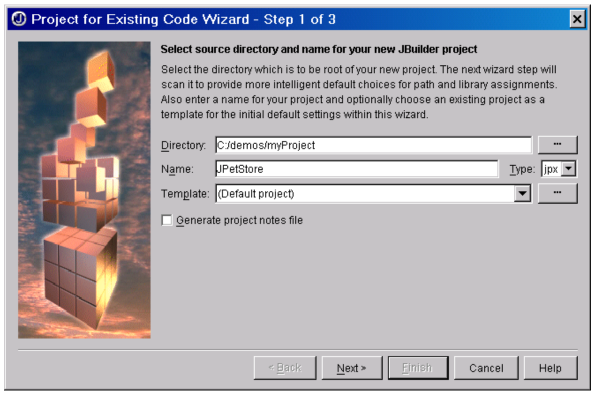
The beauty of this example is that the directory structure of this project is slightly confusing because the author did a couple of things in a way that would puzzle JBuilder. This is a perfect example of an instance where it would be best to intervene and rearrange a few directories before beginning the import process. There are three basic issues with the directory structure of this project:

1) The directory "src" is actually the Web application root context directory.
2) The Java source files are stored in a directory under WEB-INF called classes: this is a mistake, as the classes directory would usually contain compiled classes.
3) The WEB-INF directory contains a directory called lib. JBuilder requires jars to be stored outside of the WEB-INF directory. Since JBuilder dynamically re-builds both the classes directory and the lib directory as a function of the Web application dependencies, the jars in the lib directory would then be deleted.

The first problem is readily solved by renaming the src directory something else, such as Web_root, public_html, or JPetStore. We choose JPetStore. The second issue is easily resolved by renaming the classes directory to src. Finally, the third challenge is solved by moving the lib directory from the WEB-INF directory to the project root directory, myProject.

One more item to note is that we do not need the servlet.jar library in devlib, since JBuilder provides all the necessary servlet jars along with Tomcat. Thus, delete the myProject/devlib directory. Now, we are ready to run the project import wizard.

Select the project import wizard; point it to the myProject directory (**File | New…**, select the **Project** tab, and double click the **Project for Existing Code** icon).


Project for Existing Code Wizard - Step 1 of 3

Select source directory and name for your new JBuilder project

Select the directory which is to be root of your new project. The next wizard step will scan it to provide more intelligent default choices for path and library assignments. Also enter a name for your project and optionally choose an existing project as a template for the initial default settings within this wizard.

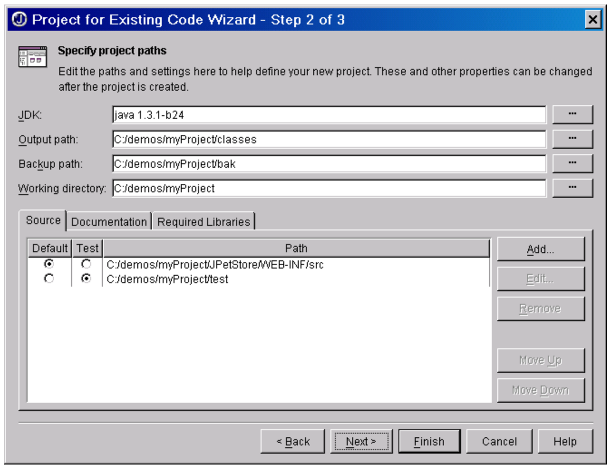
Directory: C:/demos/myProject

Name: JPetStore   Type: jpx

Template: (Default project)

☐ Generate project notes file

Name the project **JPetStore** and click **Next**.


Project for Existing Code Wizard - Step 2 of 3

Specify project paths

Edit the paths and settings here to help define your new project. These and other properties can be changed after the project is created.

JDK: java 1.3.1-b24

Output path: C:/demos/myProject/classes

Backup path: C:/demos/myProject/bak

Working directory: C:/demos/myProject

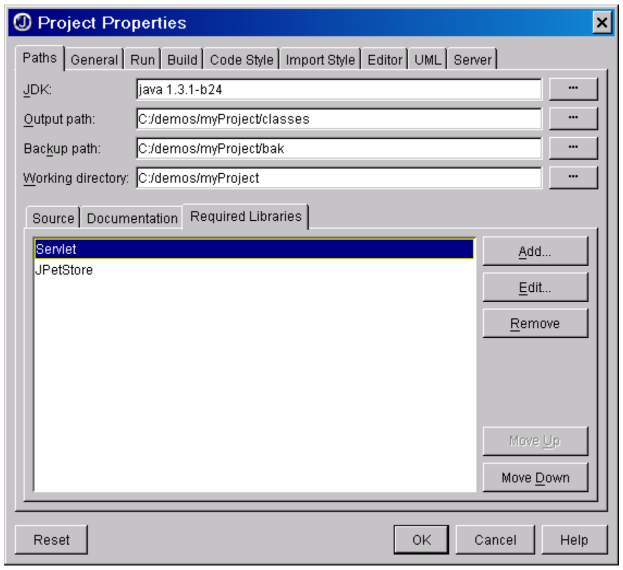
Source | Documentation | Required Libraries

Default	Test	Path
●	○	C:/demos/myProject/JPetStore/WEB-INF/src
○	●	C:/demos/myProject/test

On the next screen, notice that JBuilder automatically recognized that the source is located in the myProject/JPetStore/WEB-INF/src directory. Click **Finish**. The project should readily be imported.

Note in the project that JBuilder automatically set up a Web application called JPetStore and added the build.xml file. If you want to use the build.xml file, you need to update the build.xml file to reflect the changes you made to the directory structure.

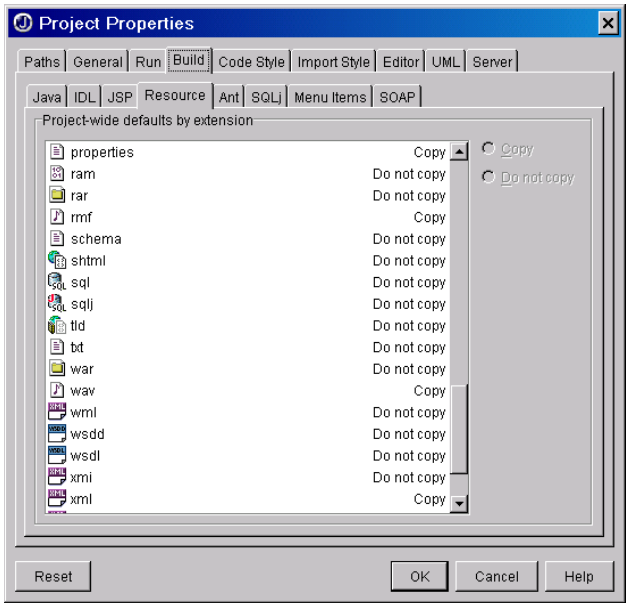
Before trying to build the project, make sure that the Tomcat servlet library is set above the JPetStore library that JBuilder added to the project. Select the menu item **Project | Project Properties**, click the **Required Libraries** tab, highlight the **Servlet** library, and click **Move Up** until the Servlet library is at the top. The Servlet library contains the Tomcat 4.0 servlet libraries which need to supersede anything that might be in the project. Click **OK**.


Project Properties
Paths | General | Run | Build | Code Style | Import Style | Editor | UML | Server
Source | Documentation | Required Libraries
Add...
Edit...
Remove
Move Up
Move Down
Reset
OK
Cancel
Help

JDK: java 1.3.1-b24
Output path: C:/demos/myProject/classes
Backup path: C:/demos/myProject/bak
Working directory: C:/demos/myProject

Servlet
JPetStore

Next, because the project source includes .properties and .xml files, we need to make sure these files get included with the compiled classes during the build process. To make that happen, click the **Build** tab and select the **Resources** tab. Make sure that xml and properties files are set to copy.


Project Properties
Paths | General | Run | Build | Code Style | Import Style | Editor | UML | Server
Java | IDL | JSP | Resource | Ant | SQLj | Menu Items | SOAP
Copy
Do not copy
Reset
OK
Cancel
Help

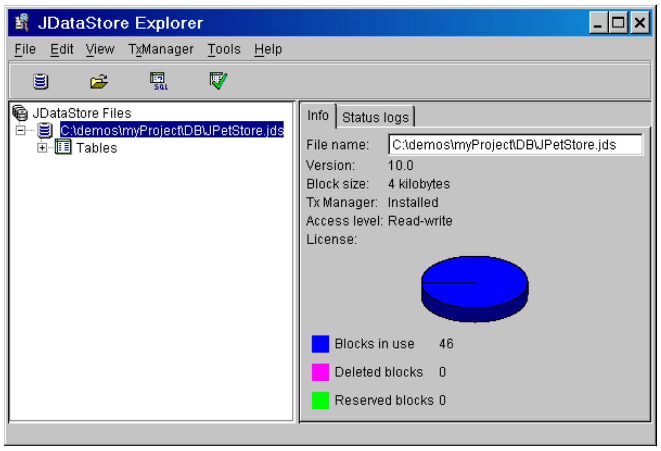
Project-wide defaults by extension

properties	Copy
ram	Do not copy
rar	Do not copy
rmf	Copy
schema	Do not copy
shtml	Do not copy
sql	Do not copy
sqlj	Do not copy
tld	Do not copy
txt	Do not copy
war	Do not copy
wav	Copy
wml	Do not copy
wsdd	Do not copy
wsdl	Do not copy
xmi	Do not copy
xml	Copy

Finally, you need to create the database that will provide the persistence for the application. You can readily use JDataStore the high performance, compact Java database that comes with JBuilder. Before you do this, we recommend you download the latest available version of JDataStore available from the Borland Web site at: http://www.borland.com/products/downloads/download_jdatastore.html. At the time of this writing, JDataStore 6 was the most recent release, so the following steps will be based on JDataStore 6. The download should include instructions on how to install JDataStore and update the version of JDataStore available in JBuilder.

Once you have the latest version of JDataStore installed, first add the JDataStore library to the project so that the JDBC® driver for JDataStore is available to the classes of the application. You can do this by editing the project properties (**Project|Project Properties** menu path) and selecting the **Required Libraries** tab. Click **Add** and add the **JDataStore library** entry available under the JBuilder folder. Click **OK**.

Next, we need to create the database and load the schema and data. To create the database, all we have to do is bring up the JDataStore explorer (**Tools|JDataStore Explorer** menu path), select **File|New** and create a new database. Create the database in a subdirectory of the project called DB (myProject/DB).



C:\demos\myProject\DB\JPetStore.jds

File name: C:\demos\myProject\DB\JPetStore.jds
Version: 10.0
Block size: 4 kilobytes
Tx Manager: Installed
Access level: Read-write
License:

🟦 Blocks in use 46
🟪 Deleted blocks 0
🟩 Reserved blocks 0

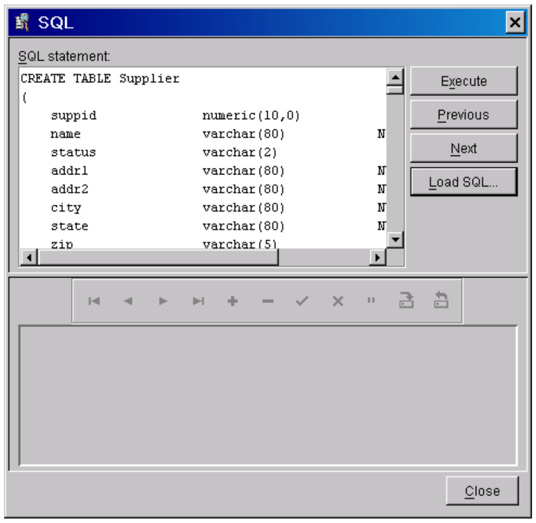
To load the schema, select the menu path **Tools|SQL**, and then click the **Load SQL** button. In the resulting file dialog, you can load the Oracle SQL script `myProject/other-`

`ddl/oracle/ JPetShop_Schema.sql` that came with the JPetStore project. Note that you will need to make some minor modifications to the script to make it work with JDataStore. You will need to replace all occurrences of the keyword number with the keyword numeric, and all occurrences of varchar2 with varchar. Finally, you will need to replace the column name time of the table OrderStatus with the word TIME in double quotes. So for that table, you should have

```
CREATE TABLE OrderStatus
(
    orderid                 numeric(10,0)
NOT NULL,
    linenum                 numeric(10,0)
NOT NULL,
    "TIME"                    date
NOT NULL,
    status                  varchar(2)
NOT NULL
);
```

You will also need to comment out or delete all the drop table statements.

(You might want to use JBuilder to make the edits to the file first using the replace functions of JBuilder.)



```
CREATE TABLE Supplier
(
    suppid              numeric(10,0)
    name                varchar(80)          N
    status              varchar(2)
    addr1               varchar(80)          N
    addr2               varchar(80)          N
    city                varchar(80)          N
    state               varchar(80)          N
    zip                 varchar(5)
```

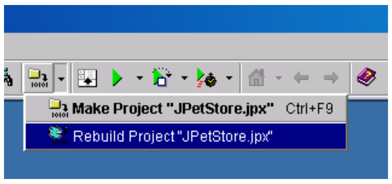
When your SQL script is ready, just click the **Execute** button. This should create the table. Repeat the above steps with the script myProject/other-ddl/oracle/ JPetShop_DataLoad.sql. You will not need to make edits to this script.

You should now have a database that is ready to be used. The only remaining step is to edit the application properties file to point the application to the new JDataStore database. Just edit the file **`myProject/JPetStore/WEB-INF/src/properties/SimpleDataSource.properties`** and add the following:

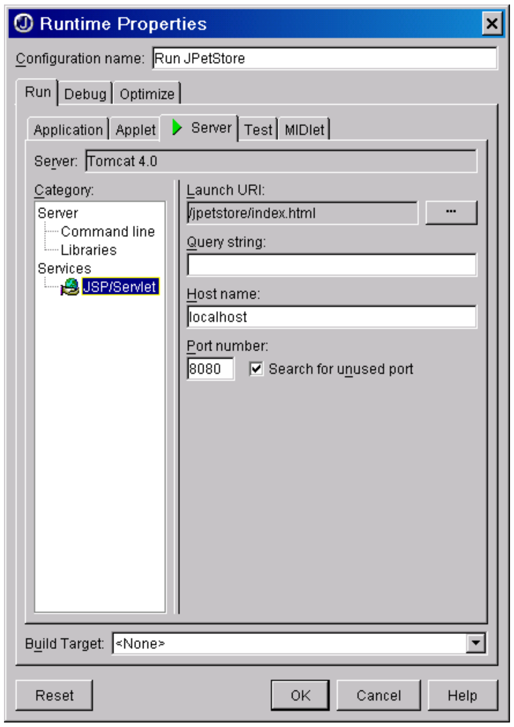
```
#JDataStore
JDBC.Driver=com.borland.datastore.jdbc.DataStoreDriver
JDBC.ConnectionURL=jdbc:borland:dslocal:C:/demos/myProject/DB/JPetStore.jds
JDBC.Username=user
JDBC.Password=
```

Make sure the other database entries are all commented out.

Now we are ready to build the project. Select the **Rebuild** option from the Build dropdown.


Make Project "JPetStore.jpx" Ctrl+F9
Rebuild Project "JPetStore.jpx"

This will build the project. You should see no errors during the build process. Next, we run the project. To run the project, we need to set up a runtime configuration. This can be done by selecting the menu path **run|configurations** and clicking the **New** tab. Select the **Server** tab, and click the **JSP/Servlet** entry. Then, set the launch URI to **jpetstore/index.html** and set the configuration name to **Run JPetStore**. Set the **Build target** to none. This will ensure that the project doesn't rebuild each time you try to run the project. Click **OK**.



Runtime Properties

Configuration name: Run JPetStore

Server: Tomcat 4.0

Category:
- Server
  - Command line
  - Libraries
- Services
  - JSP/Servlet

Launch URI: /jpetstore/index.html

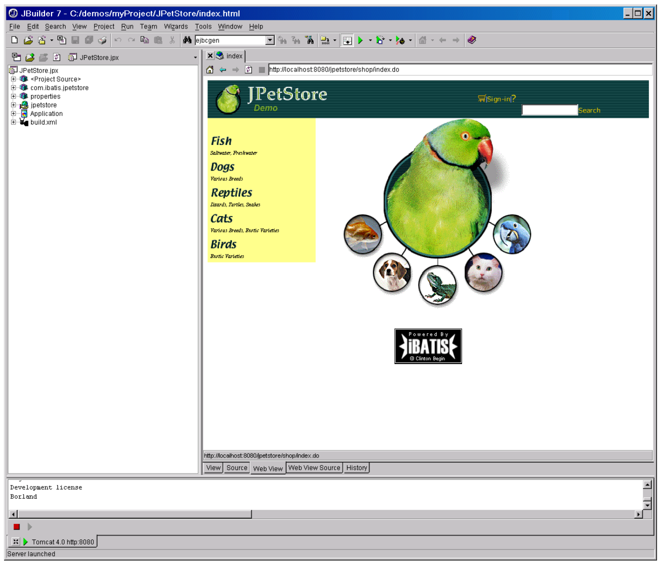
Query string:

Host name: localhost

Port number: 8080 ☑ Search for unused port

Build Target: <None>

Now we can run the project. Click the **green arrow** above the content pane to run. You should now see the main index.html Web page come up. Click the **Enter the Store** link. You should now see the Petstore home page.



So far, we have managed to pull in an entire Web application into JBuilder. The import process is smooth and easy once some simple adjustments to the directory structures are made. Most of the work actually turns out to involve configuring the database to run with the application. The value of this example is in illustrating how some minor directory rearranging can ensure a smooth import of even the most complex application.

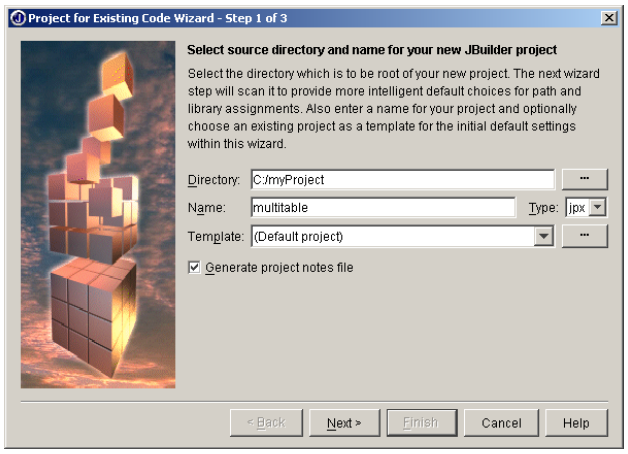
Example: importing an EJB™ 2.0 project from WebLogic™

Next, we tackle a simple EJB 2.0 sample from WebLogic Server 7.0. We choose the EJB 2.0 cascadeDelete one2many application that is available in the WebLogic 7.0 samples. So we first copy the files from `WebLogic700\samples\server\src\examples\ejb20\multitable` to a new project directory myProject. This code also requires the examples.utils package, so make sure to copy the code in the examples/utils directory as well.

Upon closer inspection of the deployment descriptors, we can see that the descriptors actually expect the source to be in a package examples.ejb20.multitable. When we copied over the files, we lost that package structures. So we move all the Java source files into a directory `myProject/src/examples/ejb20/multitable`, and we move all the HTML Javadoc files into a directory `myProject/doc/examples/ejb20/multitable`. Don't forget the utils package as well, which should be copied to the **examples/utils** directory. Finally, to keep things clean, we move all **deployment descriptors** into the directory **myProject/META-INF**. This last step is not required; the program will load the descriptors from a subdirectory.

Now, we are ready to import the project into JBuilder. We start the import wizard and walk through the same steps as in prior examples to import the project.

Start the Project from the Existing Code wizard, and point the wizard to your new myProject directory that contains the WebLogic project files (**File|New** menu path, select the **Project** tab, and click the **Project From Existing Source** icon).



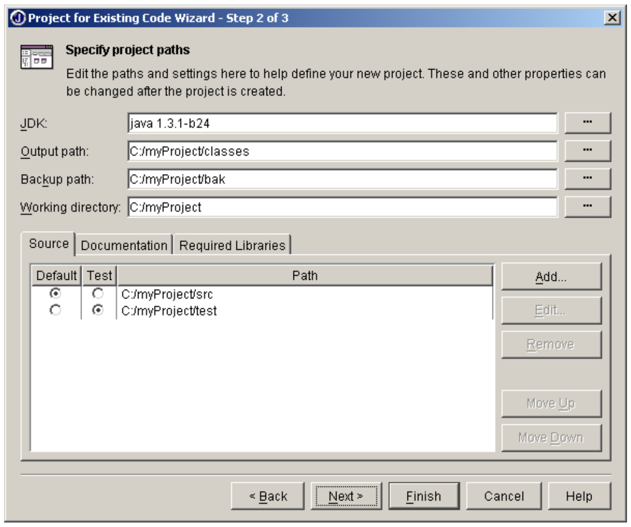
Project for Existing Code Wizard - Step 1 of 3

Select source directory and name for your new JBuilder project

Select the directory which is to be root of your new project. The next wizard step will scan it to provide more intelligent default choices for path and library assignments. Also enter a name for your project and optionally choose an existing project as a template for the initial default settings within this wizard.

Directory: C:/myProject
Name: multitable Type: jpx
Template: (Default project)
☑ Generate project notes file

Point the wizard to the **myProject** directory and give the project a **name**. Click **Next**.



Project for Existing Code Wizard - Step 2 of 3

Specify project paths

Edit the paths and settings here to help define your new project. These and other properties can be changed after the project is created.

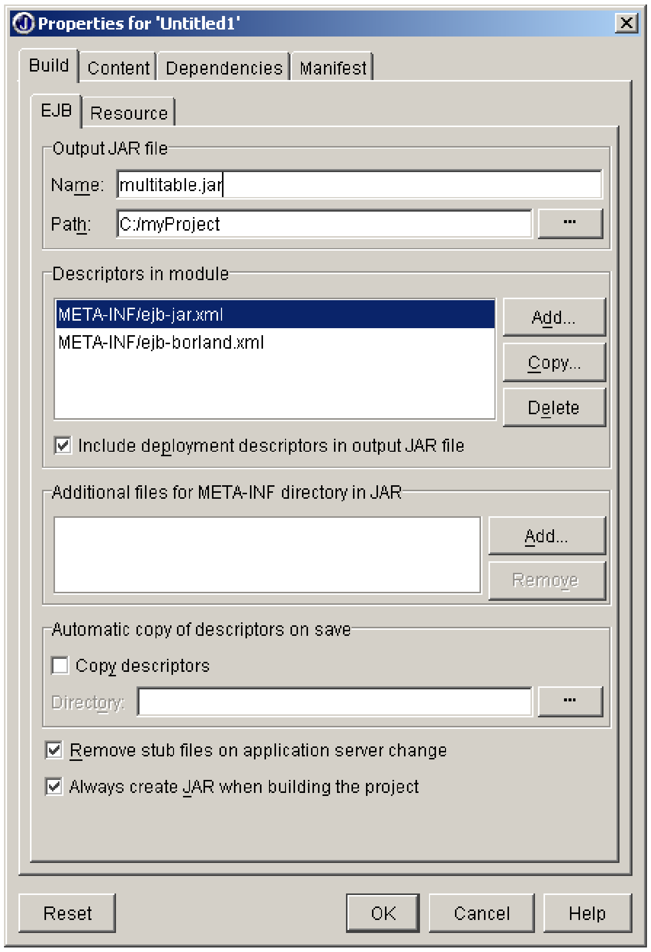
JDK: java 1.3.1-b24
Output path: C:/myProject/classes
Backup path: C:/myProject/bak
Working directory: C:/myProject

Source | Documentation | Required Libraries

Default	Test	Path
●	○	C:/myProject/src
○	●	C:/myProject/test

Make sure the project parameters are as expected, (the source directory should point to myProject/src) and click **Finish**. At this point, JBuilder will create a project, complete with an EJB 2.0 module.

Note that the EJB module has the name Untitled1. You can change the name to anything you like by right-clicking the module and selecting the rename option. Also, don't forget to rename the JAR name by right-clicking the EJB module and selecting the properties option.



Properties for 'Untitled1'

Build | Content | Dependencies | Manifest

EJB | Resource

Output JAR file
Name: multitable.jar
Path: C:/myProject

Descriptors in module
META-INF/ejb-jar.xml
META-INF/ejb-borland.xml

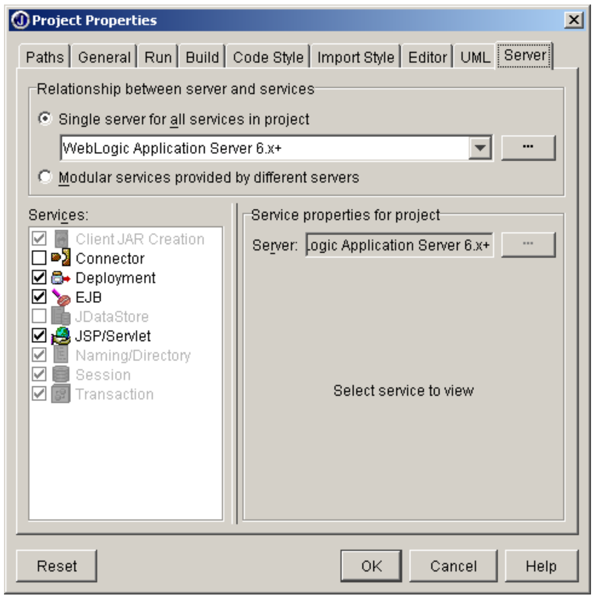
☑ Include deployment descriptors in output JAR file

Additional files for META-INF directory in JAR

Automatic copy of descriptors on save
☐ Copy descriptors
Directory:

☑ Remove stub files on application server change
☑ Always create JAR when building the project

You can readily open the EJB 2.0 module any time and see the EJB represented in the EJB designer. Before you open the module, however, make sure that the application server selected for this application is WebLogic 6.x. Do this by selecting the menu path **Project|Project Properties**, select the **Server** tab, and make sure **WebLogic 6.x** is selected as a single server for all services in the project.

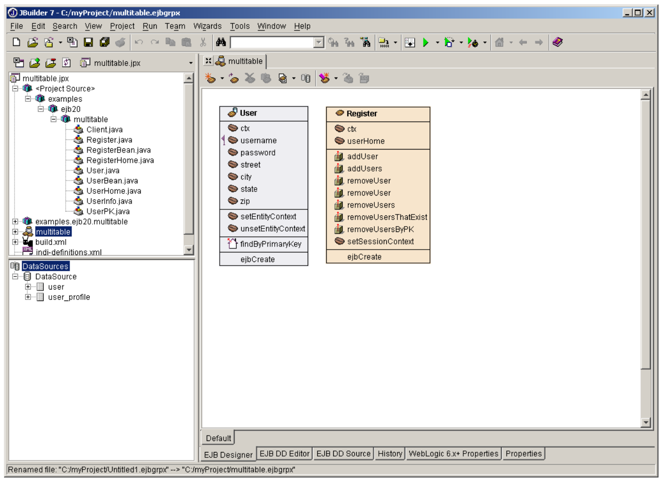

Project Properties
Paths | General | Run | Build | Code Style | Import Style | Editor | UML | Server
Reset | OK | Cancel | Help

Relationship between server and services
- (●) Single server for all services in project: WebLogic Application Server 6.x+
- ( ) Modular services provided by different servers

Services:
- [✓] Client JAR Creation
- [ ] Connector
- [✓] Deployment
- [✓] EJB
- [ ] JDataStore
- [✓] JSP/Servlet
- [✓] Naming/Directory
- [✓] Session
- [✓] Transaction

Service properties for project
Server: WebLogic Application Server 6.x+
Select service to view

Click **OK**. You are now ready to open the EJB 2.0 module. You can do that by double-clicking the **EJB module** node in the project pane.


JBuilder 7 - C:/myProject/multitable.ejbgrpx
File Edit Search View Project Run Team Wizards Tools Window Help
multitable.jpx
multitable
Default
EJB Designer | EJB DD Editor | EJB DD Source | History | WebLogic 6.x+ Properties | Properties

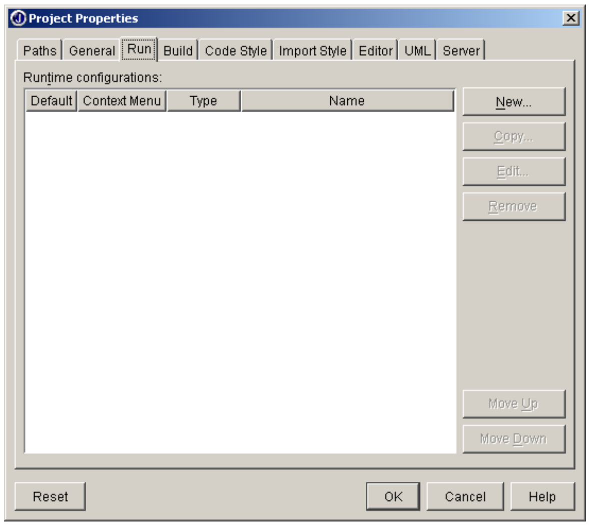
```
multitable.jpx
  <Project Source>
    examples
      ejb20
        multitable
          Client.java
          Register.java
          RegisterBean.java
          RegisterHome.java
          User.java
          UserBean.java
          UserHome.java
          UserInfo.java
          UserPK.java
  examples.ejb20.multitable
  multitable
  build.xml
  jndi-definitions.xml

DataSources
  DataSource
    user
    user_profile
```

User	Register
ctx	ctx
username	userHome
password	addUser
street	addUsers
city	removeUser
state	removeUser
zip	removeUsers
setEntityContext	removeUsersThatExist
unsetEntityContext	removeUsersByPK
findByPrimaryKey	setSessionContext
ejbCreate	ejbCreate

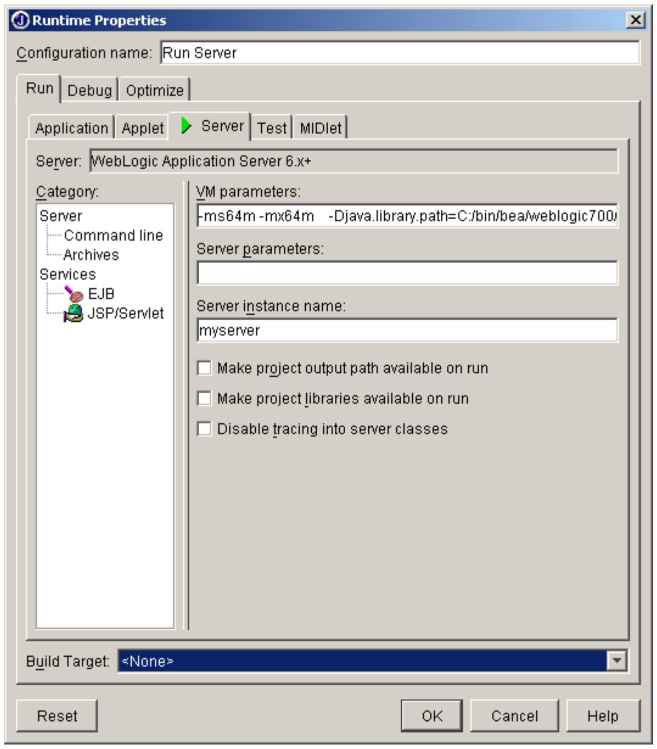
Renamed file: "C:/myProject/Untitled1.ejbgrpx" --> "C:/myProject/multitable.ejbgrpx"

Note that when you first open the module, no data source is indicated. This is a known issue and can be fixed by right-clicking the **Datasource** node and selecting "**import from database.**" In the subsequent dialog, select **cancel**. The datasource that was defined in the deployment descriptors should now be visible. In order to make sure this datasource doesn't conflict with any other datasource defined on the server side, we rename the datasource to a unique name associated with our project. For this example, right-click the **Datasource**, rename the datasource **multi-tableDB**, and hit **return**.

Next, we just add two run configurations, one for the server and one for the client. Select the menu path **Run | Configurations**, and in the resulting dialog box, select **New**.


Project Properties
Paths | General | Run | Build | Code Style | Import Style | Editor | UML | Server
Runtime configurations:
Default | Context Menu | Type | Name
New...
Copy...
Edit...
Remove
Move Up
Move Down
Reset | OK | Cancel | Help

In the next screen, set run configuration to **Run Server** and select the **Server** tab. Then click **OK**.



Configuration name: Run Server

Server: WebLogic Application Server 6.x+

VM parameters: -ms64m -mx64m -Djava.library.path=C:/bin/bea/weblogic700/

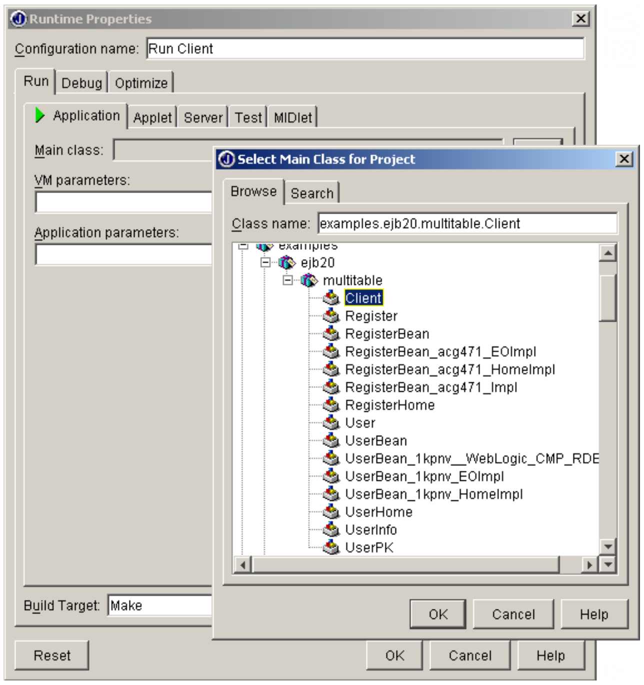
Server parameters:

Server instance name: myserver

- [ ] Make project output path available on run
- [ ] Make project libraries available on run
- [ ] Disable tracing into server classes

Build Target: <None>

Next, we add a run configuration for the client. Select **New** again. This time, name the run configuration **Run Client**, select the **Application** tab (this is the default), and click the **… button** to the right of the Main Class field. Select the class **examples.ejb20.multitable.client**.



Configuration name: Run Client

Main class:

VM parameters:

Application parameters:

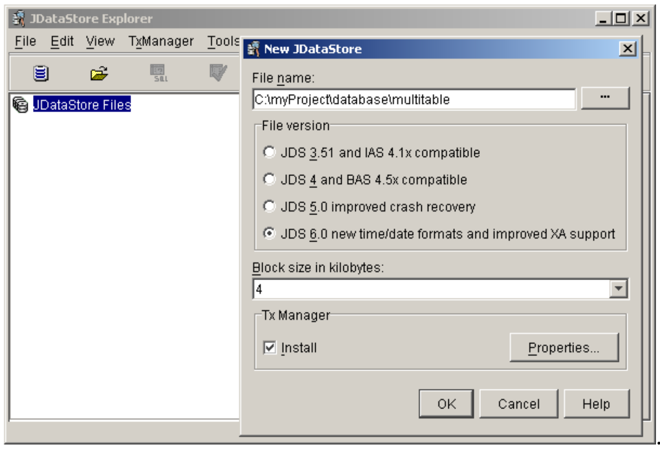
Build Target: Make

Class name: examples.ejb20.multitable.Client

```
examples
  ejb20
    multitable
      Client
      Register
      RegisterBean
      RegisterBean_acg471_EOImpl
      RegisterBean_acg471_HomeImpl
      RegisterBean_acg471_Impl
      RegisterHome
      User
      UserBean
      UserBean_1kpnv__WebLogic_CMP_RDE
      UserBean_1kpnv_EOImpl
      UserBean_1kpnv_HomeImpl
      UserHome
      UserInfo
      UserPK
```

Click **OK**, then click **OK** again to save the new Run Client runner. Finally, click **OK** to close the run configurations dialog.

Before we run the project, we still need to create a database for the project and to configure WebLogic to run the application. To create the database, we use the JDataStore™ Explorer. Select the menu path **Tools|JDataStore Explorer**, and then select **New**. This will prompt you to enter a **file name** for the new database. For this project, we call the database multitable.jds, and we create this file in the directory myProject/database.



File name: C:\myProject\database\multitable

File version

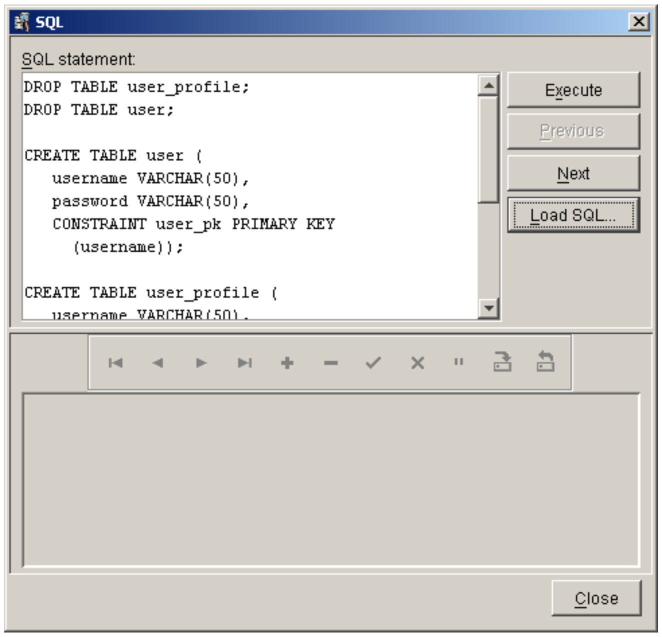
( ) JDS 3.51 and IAS 4.1x compatible
( ) JDS 4 and BAS 4.5x compatible
( ) JDS 5.0 improved crash recovery
[✓ JDS 6.0 new time/date formats and improved XA support]

Block size in kilobytes: 4

Tx Manager

[✓ Install]

Leave the other fields with their defaults and click **OK**. This will create the database. At this point, just load the database script that came with the WebLogic example. Select the **Tools|SQL** menu path, and in the resulting dialog, click the **Load SQL** button to load the database creation script table.ddl.



SQL statement:

```
DROP TABLE user_profile;
DROP TABLE user;

CREATE TABLE user (
   username VARCHAR(50),
   password VARCHAR(50),
   CONSTRAINT user_pk PRIMARY KEY
     (username));

CREATE TABLE user_profile (
   username VARCHAR(50),
```

Delete the DROP TABLE statements and replace the word user in the line CREATE TABLE user with "**USER**" (this is necessary because the word user is a reserved SQL word). 0Then click the **Execute** button to create the tables in the database. Verify that the tables have been created and exit JDataStore Explorer.

The last step is to configure the database pools in WebLogic. First, build the entire project and run the application server. When running a WebLogic runtime for the first time, the jar is actually not deployed. That way, you can run the WebLogic console and configure the proper database pools.

In the WebLogic console, first create a connection pool called multitableDBPool with the following parameters:

```
url:
jdbc:borland:dslocal:C:\myProject\database
\multitable.jds
driver:
com.borland.datastore.jdbc.DataStoreDriver
Properties:
  user=user
  password=pass
Connection initial capacity: 10
Connection maximum capacity: 100
```

Then, add a Tx Datasource called multitableDB with the following parameters:
`JNDI Name: multitableDB` (this must map the deployment descriptor datasource name)
`Datasource Name: multitableDBPool` (Same name as above)

Add neither the pool nor the actual datasource to the server targets until you have restarted the server.

At this point, you should be ready to run both the application server and the client.

Borland®

100 Enterprise Way
Scotts Valley, CA 95066-3249
www.borland.com | 831-431-1000