Patterns-based design and development for architects, Part 2: Using architecture patterns

Discover how patterns improve performance, availability, and scalability

Arun Chhatpar (arunchhatpar@gmail.com)
Software Architect
Freelance Software Architect and Consultant

05 June 2007

Design patterns are one of the best ways to share design ideas. They give software architects and designers a tool, or a language, to capture their experiences by solving common recurring problems in a methodical way. In this two-part series you explore ways to use design patterns to solve your everyday design problems. This tutorial, Part 2 in the series, uses the same railway reservation system discussed in Part 1 to show you advantages and pitfalls of applying different architecture patterns.

View more content in this series

Before you start

This two-part series is for all programmers, architects, developers, and technical enthusiasts who are interested in improving their software application designs. After finishing this series, you will be able to use best practices while choosing the right design patterns for solving specific problems.

About this series

This series demonstrates how to apply design patterns to architecture design problems using a railway reservation case study.

Part 1 introduces a railway reservation system and walks you through several design considerations that help determine where to use design patterns to improve the design and thus, the overall performance of the system.

This tutorial discusses nonfunctional requirements of the application, explaining why you, as a software architect, must take care of requirements that affect the performance, availability, scalability, and enhanceability of an application. It also touches on design considerations for disaster recovery and fail-back capabilities. This tutorial concludes with a discussion about using frameworks in your designs.

About this tutorial

An *architecture pattern* helps define a fundamental structural organization or schema for software systems. It also provides a predefined set of subsystems, specifying responsibility of each component used in the system, and includes rules and guidelines for organizing the relationships between them. A *design pattern* provides a schema for refining the subsystems or components used to define the whole software application. It also provides a vocabulary to define relationships between subcomponents. Design patterns are used to describe solutions to a general design problem within a particular context.

This tutorial extends the railway reservation system case study discussed in Part 1 by applying some architecture patterns. There are various categories of architecture patterns. The *deployment* architecture patterns are most important because the way an application is deployed is critical to all the nonfunctional requirements of an application, such as performance and availability.

In this tutorial you'll learn about the nonfunctional requirements of the railway reservation system from an architect's point of view. You'll learn how to improve on the basic design of the system using different architecture patterns. This tutorial also discusses using frameworks such as MVC, Struts, and Spring to deal with other design issues, including code reuse and overall application development time using one, or a combination of, the frameworks.

Some of the architectural considerations discussed in this tutorial are:

- High performance
- High scalability
- Failover and failback capabilities
- Disaster recovery

Prerequisites

This tutorial assumes you are familiar with design patterns and with basic object-oriented concepts. Some understanding of Unified Modeling Language (UML) is helpful, but not required. The sample code is written in Java™, but it's simple enough to be translated to the language of your choice.

You can download Java 5.0 if needed.

An architect's view of system requirements

On a new project, some of the first considerations are the business requirements of the target application. Business rules are not in a vacuum, however. Other requirements such as performance, scalability, and availability should also be near the top of the list when considering a design. An architect plays a vital role in analyzing these "nonfunctional" requirements of the system.

This section explores some of the nonfunctional requirements, and puts them in the context of the railway reservation system from Part 1.

The railway reservation case study

There are certain requirements that an architect must not overlook while designing the application. The objective way to ascertain the requirements is as simple as someone explaining the problem in plain English. Listing 1 is an example of such a conversation between the client and the architect of the reservation system.

Listing 1. Understanding the nonfunctional requirements

```
Architect: What kind of performance do you want from the finished application?

Client: We want the response from the system to be within the acceptable
tolerance limits of average Web users.

Architect: That's a good way to put it, but it would be very helpful if we had some
concrete numbers for the expected performance.

Client: OK, a user should get a response within 10 seconds for any
transaction made on the application.

Architect: Generally, the response time is highly dependent on the type of connection.

Client: I agree. The response time should not exceed 10 seconds on a broadband
connection and 15 seconds on dial-up.

Client: I also want the system to be available most of  the time.  I can only
accept a down time of 2 days a year. I understand you need to occasionally
restart the servers for various reasons, but I want you to build the system
so that restarts don't affect end users.  If we have to provide a backup
server that runs while the main server is restarted,
we can provide one for this deployment.

Architect: Got it! How many users are you expecting to use the system?
And, how many of them will be using the system at the same time?

Client: To start, I'm expecting a user base of 500. But, I want the system to support
as many as 2,000 users without any major modifications. At any point in time
I expect 5% of the users to use the system concurrently.

Architect: One last question -- Where do you host your servers?

Client: We have data centers in two cities, which are the only places where we
can host our servers. That brings up a good point. What happens to the application
if the server hosting it goes down?

Architect: We can have servers with failover capabilities, so if one of the
servers goes down, the other takes over. Since you have two data centers,
we can also have disaster-recovery capabilities so a replica of your
environment is available in the second data center. Even if one data center
goes down completely, the other takes over.

Client: I like that! Can you give me a proposal for all of this?
```

As an architect, you now have very good data that would help you make decisions to build a robust and fail-safe system. The next section looks at the requirements in a more technical context.

Subjective description of the same nonfunctional requirements

From Listing 1 you can derive the nonfunctional requirements for the system, as shown in Table 1.

Table 1. Nonfunctional requirements

Requirement	Explanation
Performance	Response within 15 seconds on dial-up connection, and within 10 seconds on broadband.
Availability	System should be available at least 363 days/year; acceptable down time is only 48 hours.
Scalability	System should be able to support 2,000 users with 100 of them concurrent.
Failover and disaster recovery	If the main application server dies, the system should be able to failover to a different server. The system should also be able to recover from a disaster (for example, the data center goes down or becomes inaccessible).

Now that the requirements are nailed down, let's look at designing the railway reservation system to handle all of the requirements.

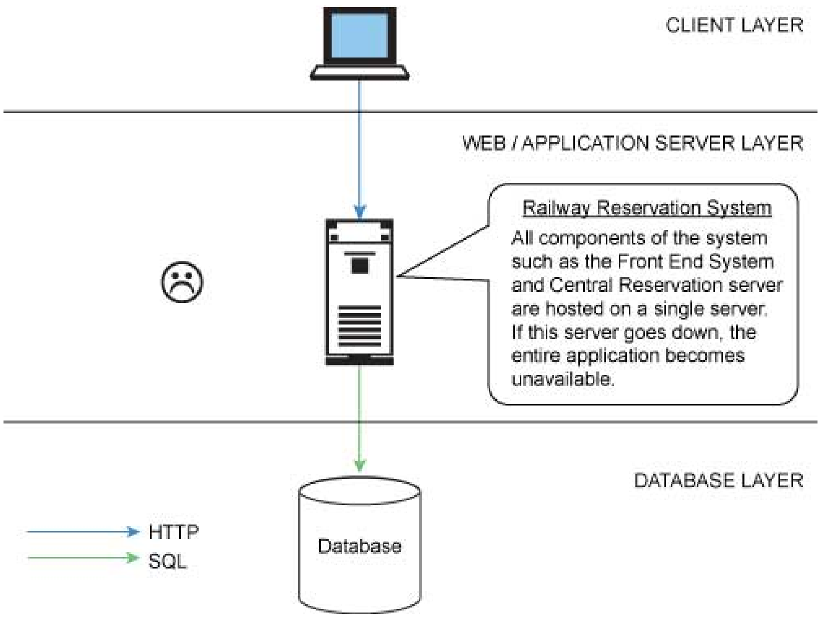
Architecture patterns

This section discusses the four nonfunctional requirements, defined in the previous section, in the context of the railway reservation system. The basic architecture pattern is the starting point to show you how to improve on and take care of each of the requirements progressively with each pattern.

Single server deployment

Single server deployment is the most basic, rudimentary architecture pattern and is not advisable for production level deployments. It is fine to use this architecture for development and unit testing purposes. In this design, all server components are hosted on a single sever. It's easy to guess what could happen if this server goes down. Figure 1 shows the deployment architecture.

Figure 1. Deployment architecture


CLIENT LAYER
WEB / APPLICATION SERVER LAYER
Railway Reservation System
All components of the system such as the Front End System and Central Reservation server are hosted on a single server. If this server goes down, the entire application becomes unavailable.
DATABASE LAYER
Database
HTTP
SQL

Because the entire application is hosted on a single Web server, there are more drawbacks to this deployment than advantages.

Advantages

- Easy deployment. It's easy to deploy all components on a single server.
- Easy maintenance. It also becomes very easy to maintain the application because everything is on a single box.

Drawbacks

- Performance of the system will be inversely proportional to the number of concurrent users. It will start deteriorating as the number of concurrent users increases because all requests are being served from a single server.
- Server availability has a direct impact on application availability. If the server goes down, the application will be down and unavailable.
- This type of architecture does not scale easily because it is not distributed. All application layers (presentation, business logic, and data) are hosted on a single server.

You would usually use this kind of deployment while developing and unit testing. It is very primitive for a critical application deployment. There are ways to change this by applying other patterns; the following section discusses one such design strategy.

Deployment using load balancers

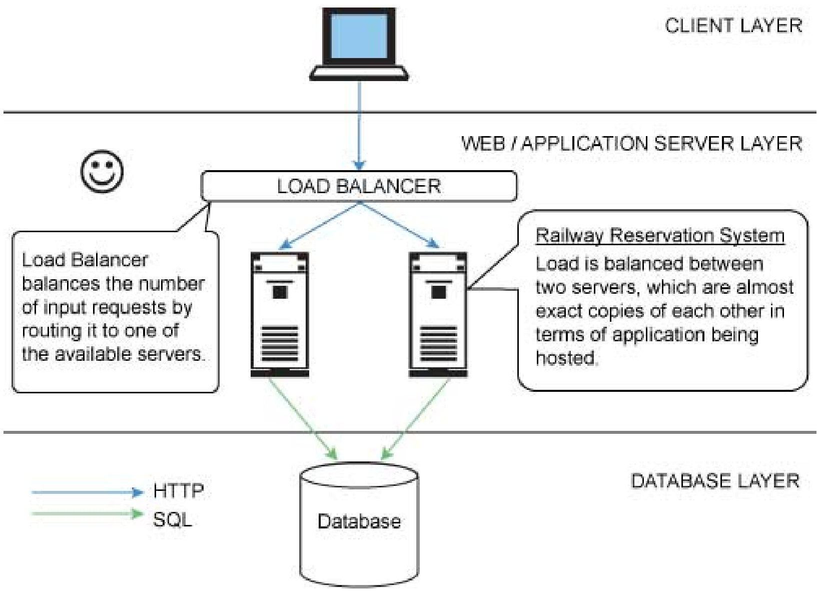
The easiest way to overcome single-server deployment and improve availability is to add multiple servers to serve requests. But that introduces the problem of requiring the end user to know the address of both servers. Users would have to manually connect to the other site in case one fails. This is not optimal for the user experience.

The deployment technique called *load balancing* can fix the problem. Clients send requests to a load balancer, and it, in turn, directs requests to the actual servers. It's all transparent to the client or user. There are various algorithms available by which the load balancer directs requests to servers. With "round robin," one of the commonly used algorithms, the load balancer sends subsequent requests to different servers in a queue. A nice feature of load balancers is that *stickiness* can be programmed with them. When a client is directed to a server by the load balancer, "stickiness" enables all subsequent requests from that client to be sent to the same server.

Railway reservation system layout

Figure 2 shows the railway reservation system layout with a load balancer pattern applied .

Figure 2. Load balanced servers


CLIENT LAYER
WEB / APPLICATION SERVER LAYER
LOAD BALANCER
Load Balancer balances the number of input requests by routing it to one of the available servers.
Railway Reservation System
Load is balanced between two servers, which are almost exact copies of each other in terms of application being hosted.
DATABASE LAYER
HTTP
SQL
Database

In this scenario, the load is distributed equally among the two servers.

Advantages

- The most immediate benefit is the improvement in availability. Even if one server goes down, the other is available to serve requests.
- Performance and scalability are slightly improved, because the overall load is distributed among different servers.

Drawbacks

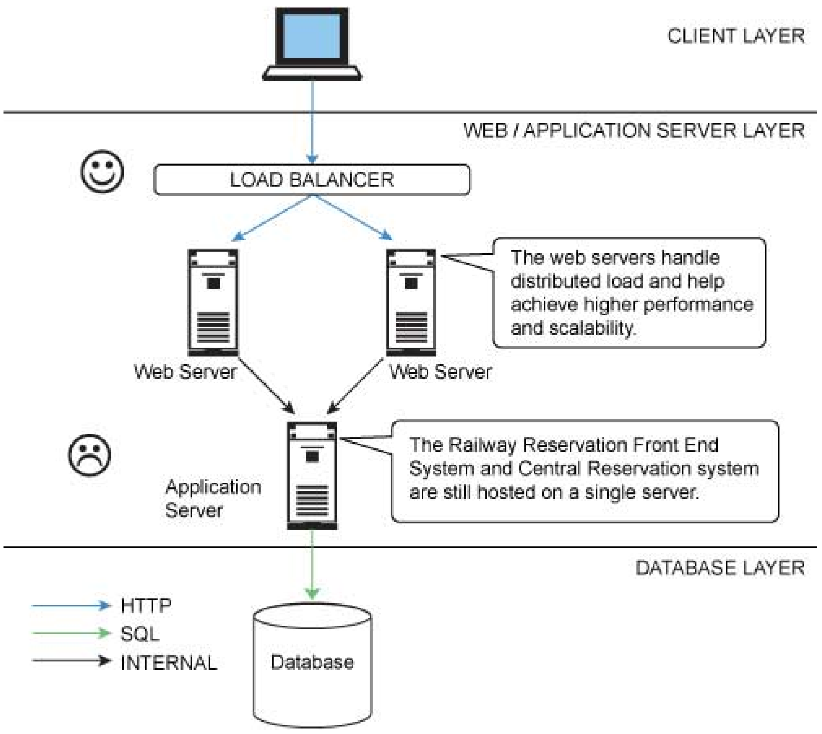
- The application architecture is still not distributed. All application layer components are hosted on a single server.

This pattern takes care of some, but not all, of the nonfunctional requirements. The next pattern shows how you can take advantage of distributed components to make more improvements.

Distributed server deployment

As shown in the load balancer scenario, the load on servers is distributed but the application is still undistributed. The presentation, business logic, and data are all hosted on a single server. Although sufficient for many applications, if you want to separate the Web layer from other layers to put it in a demilitarized zone, you need a separate Web server and application servers, as shown in Figure 3.

Figure 3. Distributed servers


CLIENT LAYER
WEB / APPLICATION SERVER LAYER
LOAD BALANCER
Web Server
Web Server
The web servers handle distributed load and help achieve higher performance and scalability.
Application Server
The Railway Reservation Front End System and Central Reservation system are still hosted on a single server.
DATABASE LAYER
HTTP
SQL
INTERNAL
Database

In this scenario, the load is well distributed -- not just by requests, but also by the kind of tasks that servers perform. This architecture is definitely better suited for good performance and higher scalability. Once you implement a distributed deployment architecture pattern, it's just a matter of adding more servers and gathering metrics until you achieve the desired numbers to meet the nonfunctional requirements.

Advantages

- Higher availability. Because more than one server is available to serve requests, if one goes down it can be easily handled and the availability of your application increases.
- Improved performance. The individual Web servers would be more capable of handling an increase in requests more efficiently, thus improving the overall performance of the application.
- Better scalability. Using this pattern lets you add new servers dynamically without affecting the availability of the application, making it highly scalable for higher traffic.

Drawbacks

- The application server still acts as a single point of failure.

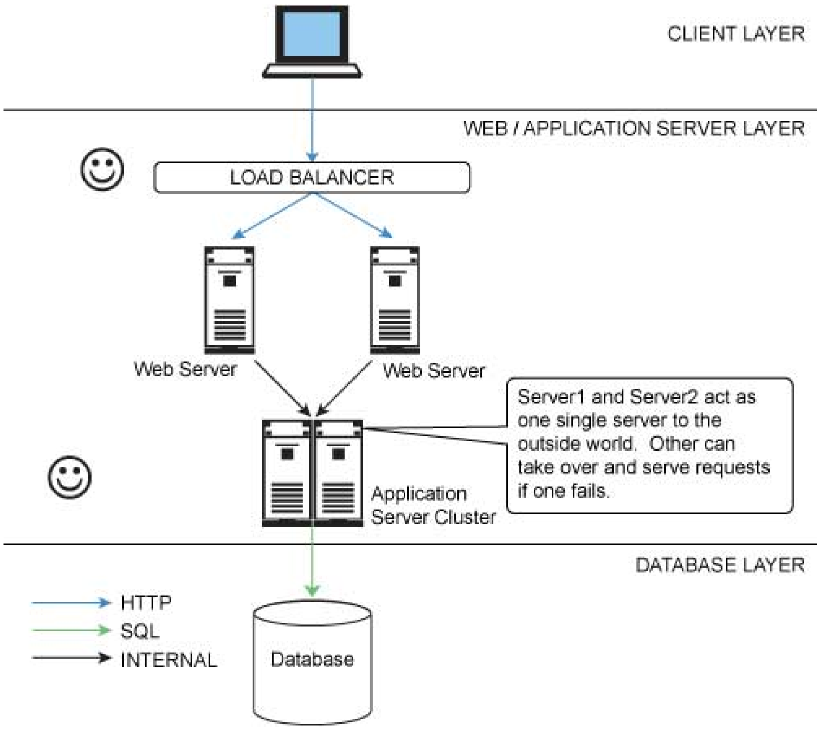
You might have noticed by now that with the introduction of each architecture pattern we are getting closer to fulfilling all nonfunctional requirements of the system. So far we have seen improvements in availability, performance, and scalability. The next few sections introduce some more patterns that provide even more improvements.

Layered deployment

In the distributed server deployment pattern, the application server is still a single point of failure. But what if the application server goes down? The Web server may still be able to serve static requests, but the dynamic requests will fail. For critical applications, such as the solution we are designing for the railway reservation system, you need a design with failover capability.

Failover capabilities can be introduced in servers by deploying them in clusters. A *cluster* is a set of servers that perform similar tasks and appear to be a single server to the external world. Figure 4 illustrates the concept.

Figure 4. Application server cluster providing failover capabilities


CLIENT LAYER
WEB / APPLICATION SERVER LAYER
LOAD BALANCER
Web Server
Web Server
Server1 and Server2 act as one single server to the outside world. Other can take over and serve requests if one fails.
Application Server Cluster
DATABASE LAYER
HTTP
SQL
INTERNAL
Database

In this scenario, the application server cluster fails over from server 1 to server 2 if there is any problem with server 1. The cluster can be configured to operate either in a hot-hot pair or a hot-cold pair. In a hot-hot pair, both the servers are continuously processing requests. In a hot-cold pair, the cold server executes requests only when the cluster fails over from the hot server to the cold one. When the hot server comes up the cluster automatically fails back from the cold server to the hot one. This configuration is helpful when the back-up server in a cluster is of a lower configuration than the main server.

Advantages

- High availability.
- Better performance.
- Better scalability.
- There is no single point of failure -- an added benefit of using a cluster of servers. The applications can failover to another server that can serve all the requests, and everything is transparent to the client or user.

Drawbacks

- The application will go down if the data center goes down. Because the entire application is hosted in a single data center, if that data center goes down in a disaster then your application or service can become unavailable.

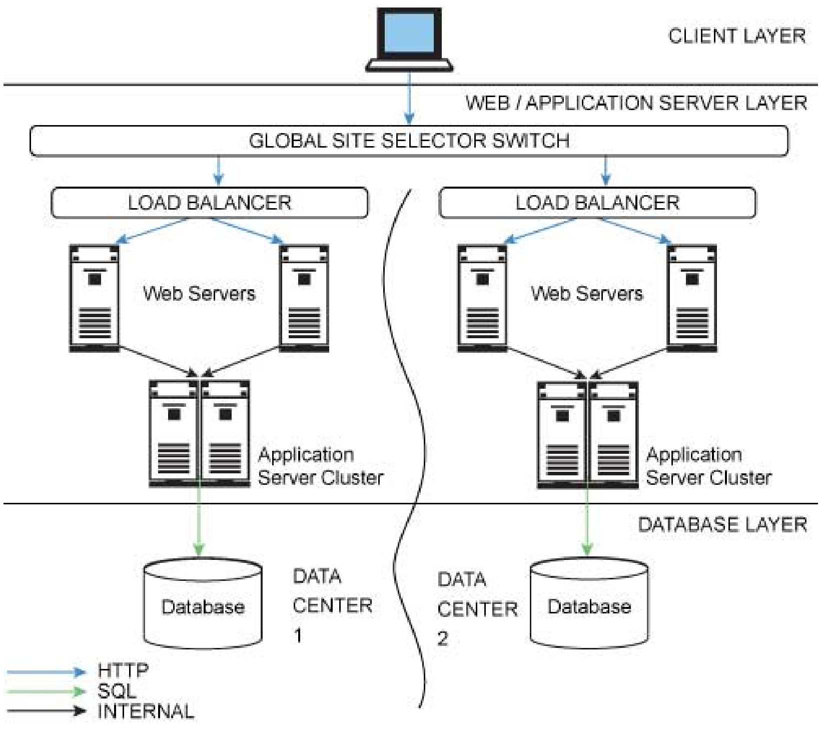
In this case, a data center represents the physical location where your application servers are hosted.

Almost all of the nonfunctional requirements have been covered so far with architectural patterns. One last thing to do is to enable your application to quickly recover from disasters. The last pattern will show how to make that happen.

Fail-safe deployment

If all servers are hosted in the same data center, the application will go down if the entire data center goes down. This kind of event is a disaster. How could our application quickly recover from a disaster? What is the business continuity plan in case of a disaster? While many applications may not require sophisticated disaster recovery, high-end critical applications do require disaster recoverability. You can achieve business continuity in case of disaster by replicating a complete deployed environment in a different data center, as shown in Figure 5.

Figure 5. Disaster recovery architecture



The switch from the main data center to the alternate data center can either be done manually, or it can be automated. For a manual switch, the DNS entry for the Web site has to be changed to point to the virtual IP address of the load balancer in the alternate data center. This task can be automated by devices such as global site selector switches. Setting up this kind of architecture is quite complicated and requires a lot of daily backups and that a restore process is in place.

A significant item shown in Figure 5 is that all data from the database in data center 1 has to be backed up to a database in data center 2.

Advantages

- Makes your application or service highly available to clients and users.
- High reliability.
- Best user experience. Because your application is available almost 24/7, 365 days of the year, it makes for a very pleasant user experience.

Drawbacks

- Implementation costs are high. You need hardware to support two server deployments, which can be a big-ticket item.
- Maintenance costs are higher, because you have to replicate every bit of information on two separate servers.
- Needs a lot of effort to achieve total recoverability.

So far, the list of issues that needed addressing have all been covered by using architecture patterns. Some are easy to implement, and others need more work. There is one more pattern that you shouldn't miss in this discussion: the MVC. You've likely heard of, or used, this pattern in your designs.

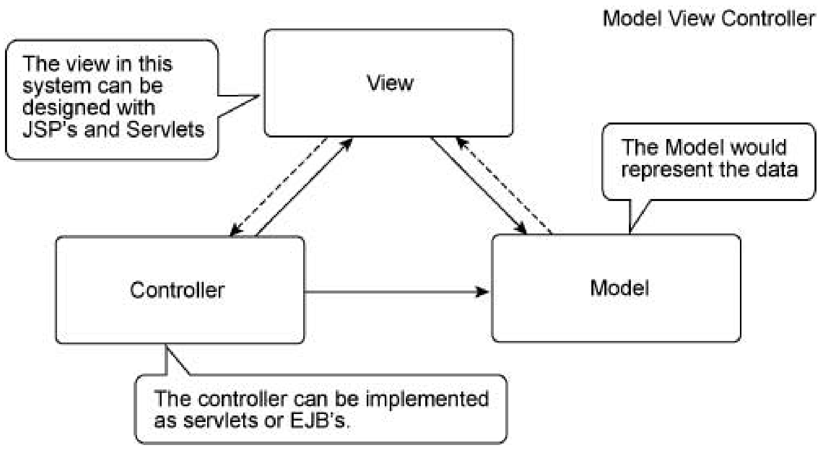
The Model-View-Controller (MVC) pattern

The MVC is one of the most commonly used architecture patterns in software systems. Its simplicity makes it easy to understand and implement in your designs. The MVC pattern is a way of breaking an application, or even a subsystem of an application, into the following parts:

- Model: Represents data being passed around
- View: Actual user interface (UI) used by the client or users
- Controller: Controls the flow of data and defines and controls the flow of events in the UI

This pattern allows the UI concerns (view) to be separated from data (model) so that changes to the UI don't affect the data handling, and so data can be reorganized without changing the UI. Figure 6 shows the basic design of an MVC system.

Figure 6. Designing railway reservation front end using MVC pattern


Model View Controller
The view in this system can be designed with JSP's and Servlets
View
The Model would represent the data
Controller
Model
The controller can be implemented as servlets or EJB's.

The MVC pattern solves the data dependency problem by decoupling data access and business logic from data presentation and the UI by using an intermediate component -- the controller.

Advantages

- Clarity in design. The separation of the system into function-specific components helps clarify the system design.
- Ability to use multiple views. The decoupling of the view from the other two components makes it possible to change the views without affecting the overall design. It also allows for multiple views for the same system. In the railway reservation system, this can become very useful in case our system needs to support legacy green screens for existing agents.
- Easy to accommodate changes. MVC makes it easy to make changes to different parts of the system without affecting other components in a big way.

Drawbacks

- It introduces new levels of complexity. The downside of separating components and achieving new levels of indirection is that it increases complexity of the solution slightly.

With very few drawbacks, MVC is one of the most commonly used architecture patterns in software design. There are a lot of frameworks that use MVC as their base model of design, such as Apache Struts, Spring, Stripes, and Tapestry. (See Resources for more information.) The next section discusses using frameworks in your design.

Using frameworks in your design

A very useful and effective way to make your applications more scalable is by using existing software frameworks . A *framework* is a reusable design for a software system. An application framework is usually a set of predesigned libraries or classes that are used to implement the standard structure of an application for solving a specific design artifact.

A big advantage of using frameworks, such as Struts and Spring, in the railway reservation system is that they help reduce the overall development time. Both of the frameworks come with many predesigned object-oriented application program interfaces (APIs), almost ready to use in your applications. Although developers have to learn how to use the APIs, it's still faster than starting from scratch. Frameworks also promote object-oriented concepts, making your code more reusable in other projects. The next sections explain how the Apache and Spring frameworks would improve the railway reservation system design.

Apache Struts

Struts is one of the more mature MVC frameworks for building Web applications based on servlets and JavaServer Pages (JSP components). Since the front end for the railway reservation system will be Web pages, Struts could be an ideal fit for our design. The flow of events in a typical enterprise Web application is:

1. Users submit information to the server using a Web form.

2. The information is given to a service component that processes it, usually interacts with the database, and produces an HTML formatted response.
3. Or, if the service uses server technologies such as JSPs, then those JSPs and Java code would be responsible for generating the same HTML response.
4. Users see the resulting page in their view.

These two techniques that generate a response are inadequate for large projects because it intermingles presentation logic with business logic, making maintenance difficult. The goal of Struts is to follow the MVC pattern to make a cleaner separation of model from the view and controller. Struts has a well-defined framework to make this possible.

Advantages

- A very powerful framework to build upon. You don't have to reinvent the MVC wheel, and can use Struts in your Web applications right away.
- Faster development cycles (one of the biggest benefits of using frameworks).
- Improved scalability. The architects who designed Struts made a sincere effort to allow your application to be as scalable as possible.

Drawbacks

- Unknown issues can creep up. If there is ever a bug in the Struts API, you have to wait for the Struts team to fix it.

The advantages of Struts make it a very strong candidate for developing the front end for our railway reservation system. The inherent design of Struts would also help achieve a higher level of decoupling between components. One objective of our design is code reuse, and using Struts would only help with that effort.

There is lot more to Struts than just the few points mentioned here, but it is beyond the scope of this tutorial to cover all of it. (See Resources for more about Struts.)

Spring framework

The Spring framework, which is much more powerful than Struts, is another lightweight framework based on MVC. It solves one of the big problems faced by application architects and developers: achieving a high level of decoupling between the components. Spring helps remove dependencies between components by using a unique technique called *Inversion of Control*. This also simplifies the testing of each component, because they are not tightly coupled with each other.

There are various ways Spring can be used in the railway reservation system design:

- Spring MVC is the Web application component similar to Struts and can be used to design the front end Web interface of our application.
- Spring has a neat Java Database Connectivity (JDBC) and Object Relational Mapping API that can help enhance the database connectivity design of our system.
- Spring also comes with support for aspect-oriented programming, which can help achieve higher scalability in the design.

The Spring framework comes with many APIs that could be very helpful in achieving the scalability needed for the railway reservation system. Read more about it in the six-part tutorial series, "Apache Geronimo and the Spring Framework" (see Resources).

Summary

In this tutorial you learned about the many ways you can improve the performance, availability, and scalability of an application. It's possible to achieve everything requested in the nonfunctional requirements of a system, but it requires a bit of thinking and expertise to devise a mix of the right deployment solutions that can address all the issues. Similar to design patterns, architecture patterns provide you, the architect, a way to learn from and improve on your designs.

Hopefully, this two-part series has provided some insight into design and architecture patterns to help you solve your own application dilemmas. If a given application is underperforming, there is a reason. The reason can usually be found and fixed either by using one of the patterns described here, or in some of the other articles and tutorials in Resources.

Resources

Learn

- Architecture Patterns from the Open Group Architecture Framework (TOGAF) gives a nice introduction to architecture patterns. It also has clear definitions showing the difference between a design pattern and architecture pattern.
- Definition Of Architectural Pattern provides in depth information about software architecture patterns.
- The Patterns Library, hosted by the Hillside Group, provides information about patterns, links to online patterns, papers, books dealing with patterns, and patterns-related mailing lists.
- Patterns-Discussion FAQ, maintained by Doug Lea, provides a very thorough and highly readable FAQ about patterns.
- Software frameworks on Wikipedia, with links to other resources.
- "Apache Geronimo and the Spring Framework" tutorial series (developerWorks, Sep 2006) covers the complete Spring Framework, front to back, including how to implement its functionality with Apache Geronimo.
- "Web services architecture using MVC style" (developerWorks, Feb 2002) takes a look at how the MVC pattern can be applied to the call static or dynamic Web services.
- Design Patterns: a personal perspective provides a simple explanation of all 23 design patterns.
- Read "A Survey of common design patterns" from Developer.com.
- Read Designing Enterprise Applications with J2EE platform, Second Edition to learn about standard approaches to designing multitier enterprise applications with the Java™ 2 Platform, Enterprise Edition.
- *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995) is considered an ultimate authority on design patterns. If you are new to design patterns, then this is where you should start.
- Core J2EE Patterns: Best Practices and Design Strategies by Deepak Alur, John Crupi, and Dan Malks (Prentice Hall, 2001) is a catalog of patterns for the design and architecture of multi-tier enterprise J2EE applications.
- *UML Distilled: Applying the Standard Object Modeling Language*, by Martin Fowler with Kendall Scott (Addison-Wesley, 2000), is a very good book for learning the essentials of UML.
- *The Unified Modeling Language User Guide*, by Grady Booch, Ivar Jacobson, and James Rumbaugh (Addison-Wesley, 1998) can be used as a reference guide to learn more about UML and design patterns.
- Visit the developerWorks Architecture zone for architecture related tutorials and articles.
- Open source: Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies, and use them with IBM products.
- developerWorks Technical events and webcasts: Stay current with developerWorks technical events and webcasts.
- Podcasts: Tune in and catch up with IBM technical experts.

- Visit IBM Pattern solutions resource center to take advantage of freely available patterns.
- View IBM on demand demos to find out more about IBM middleware and technologies.

Get products and technologies

- Apache Struts framework
- Spring Framework is full of resources on Spring.
- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Participate in the discussion forum for this content.
- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Arun Chhatpar

Arun Chhatpar has more than nine years of significant experience in Java programming and client/server architectures, and is a Sun Certified enterprise software architect. He has been a lead designer and developer for NBCi, and is currently working as an independent software architect/consultant.