



Implementing Finite State Machines with Physhun and Spring

This paper presents the Physhun project, a Spring-based framework for implementing complex processes through Finite State Machine models. Physhun provides finite State Model persistence and transaction management with synchronous or asynchronous behavior.

Justin McCarter and Jim Ladd

18 Jul 2008

Abstract

When software is required to solve complex problems, the proper solutions are often beyond the capabilities of traditional programming techniques. A robust alternative for solving very complex problems is to use the Finite State Machine (FSM) paradigm for modeling the solution space and a FSM-based framework for the implementation. This paper presents the Physhun project, a Spring-based framework for implementing complex processes through Finite State Machine models. Physhun provides finite State Model persistence and transaction management with synchronous or asynchronous behavior. Although development with Physhun can be accomplished much more quickly and efficiently with Physhun Modeler, we will demonstrate development without Modeler to give visibility into the Physhun framework.

Finite State Machines and Process Control

Complex process logic is common in information systems. It can be found in software ranging from GUIs to Operational Support Systems. There are many concise ways of describing complex processes, the most common being flow charts and UML diagrams; however, implementation of complex processes may not always be as straightforward. Complex processes are often implemented as procedural logic which can become lengthy and difficult to manage as process rules change.

A complementary technology to procedural and object-oriented programming is the Finite State Machine paradigm. Finite State Machine (FSM) is a behavioral algorithm that can be used to model and execute complex logic. A finite state machine defines a process as a set of states (nodes) and state Transitions (edges). Implementation of a process with FSM technology involves laying out the valid states for the process, all Transitions between those states, Conditions dictating when Transitions are to be executed and Actions containing logic to be executed on Transition execution.

Finite state machine technology has been used for a number of years in a wide spectrum of industries. Successful projects using finite State Model technology include communication systems, automobiles, avionics systems, and man-machine interfaces. These problem domains share common characteristics; they are usually complex, large in size and reactive in nature. A primary challenge of these domains is the difficulty of describing reactive behavior in ways that are clear, concise, and complete while at the same time formal and rigorous.

Finite State Models provide a way to describe large, complex systems. Finite State Machines view these systems as a set of well defined states, inbound and outbound events, Conditions, and Actions. FSM technology provides a set of rules for evaluating and processing the events, Conditions, and Actions. The partitioning of the problem into the states, events, Conditions, and Actions, the structured processing environment, and the ease of expressing the processing logic are the foremost strengths of FSMs.

The fundamental components of finite State Models include:

State represents the "mode of being" for the system at any given time.

Transition describes a single pathway from a state to another state. The set of all Transitions describe all possible paths among the defined states. A Transition contains an event that it is subscribing to (the event that triggers execution of the Transition), a Condition and an Action. A Transition also contains references to the state from which the Transition is exiting (i.e. the “from” state) and the state to which the Transition is entering (i.e. the “to” state).

Event is the mechanism that the system uses to interact with external systems and with itself.

Condition represents a set of logic that evaluates to a Boolean result. A Condition is used to determine if a Transition is to be executed.

Action is the logic to be executed when a Transition is executed.

A diagram of the components of a typical FSM is shown below:

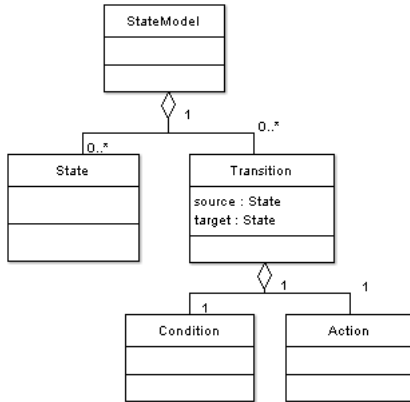


Illustration 1: State Model Core Components

Physhun Overview

Physhun is an open source framework for building and executing Finite State Machines in J2SE and J2EE environments. Although the framework is simple, it is powerful in that it allows processes to be long lived, persistent and transactional. Processes can be purely synchronous or asynchronous. Synchronous processes once started, will run to completion without any further interaction from outside systems or users. Asynchronous processes will have states that require input from external systems or users before the process will continue. Asynchronous behavior is common in workflow systems.

The Physhun framework leverages the Spring framework and exposes all of the value added services provided by Spring including Inversion of Control (IOC), Aspect-Oriented Programming (AOP) and transaction management. Physhun allows the usage of graphical editors such as Physhun Modeler for development of State Models. This allows for accelerated development, ease of maintenance and facilitated communication.

Process control with Physhun is accomplished by defining processes as State Models. A State Model is defined as a JavaBean comprised of the following components:

- States (`com.wazeegroup.physhun.framework.State`)
- Transitions (`com.wazeegroup.physhun.framework.Transition`)
- Actions (`com.wazeegroup.physhun.framework.Action`)
- Conditions (`com.wazeegroup.physhun.framework.Condition`)

A State model can be defined in one of several ways. The first way is to define the beans comprising the State Model in Java code. The second way is to define the State Model as XML that can be consumed by a bean factory (such as one of those provided by the Spring framework). The third way is to define the State Model graphically using an editor like Physhun Modeler. Defining the State Model graphically is appealing because it allows a complex process to be easily understood, communicated and maintained. In all three of these methods the elements comprising the State Model are defined and wired together as Spring beans and can be written to take advantage of Spring services.

The Physhun runtime paradigm is this: A process is defined by a process model. Multiple instances of a given process can be executed. Each process instance has a business object (`ProcessObject`) instance associated with it, which traverses a State Model. The data in the `ProcessObject` can be used to determine which paths in the State Model to take (via execution of Conditions). As the process executes, Actions in the State Model may manipulate the `ProcessObject`.

Physhun processes are instantiated and interacted with at runtime through a `ProcessContainer`. The `ProcessContainer` interface is shown in Table 1. It contains methods for starting processes and sending asynchronous events to existing processes. Executing Physun State Models is as simple as instantiating a `ProcessObject` and passing it to the `Physun ProcessContainer` instance.

```
package com.wazeegroup.physhun.engine;
```

```
import com.wazeegroup.physhun.framework.ProcessObject;
import com.wazeegroup.physhun.framework.StateModel;

public interface ProcessContainer {

    public void startProcess(ProcessObject processObject, StateModel stateModel);

    public ProcessObject sendTriggerEventToProcess(String processID, Object triggerEvent);

    public void resumeProcess(ProcessObject processObject, StateModel stateModel);

}
```

Table 1: ProcessContainer interface

The low-level rules for how State Models are executed are defined in the StateEngine. Generally, the developer does not need to interact with the StateEngine directly, but through the ProcessContainer which will proxy calls to the StateEngine appropriately. Physhun provides standard implementations for ProcessContainer and StateEngine, but the user may define and use different implementations as the need arises.

For long lived and mission critical processes, the Physhun library provides the ability to persist ProcessObjects, suspend and resume long lived processes and tie process execution into distributed transactions.

Example 1: Simple process control

In this example, we will build an overly simplified order process. The order process is this: Order received, check inventory, if the product is not in stock cancel the order. Otherwise, complete the order.

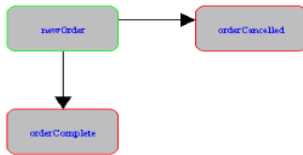


Illustration 2: Simple Example Process Model

The first step in the implementation is to define a ProcessObject. This is the object that will traverse the State Model during process execution. Other FSM technologies may refer to the object as a Business Object, Business Process Object (BPO) or Process Document. The ProcessObject implementation must implement the interface `com.wazeegroup.Physhun.framework.IProcessObject`. The ProcessObject for this example is defined by the class `com.wazeegroup.physhun.examples.simple.Order`, which is shown in Table 2.

```
package com.wazeegroup.physhun.examples.simple;

import com.wazeegroup.physhun.framework.ConcreteProcessObject;

public class Order extends ConcreteProcessObject {
    private String customerId;
    private String itemId;
    public int quantity;

    public Order(String customerId, String itemId, int quantity) {
        super(null);
        this.customerId = customerId;
        this.itemId = itemId;
        this.quantity = quantity;
        setID(toString());
    }
}
```

Table 2: A simple ProcessObject implementation

Note that Order inherits from `com.wazeegroup.Physhun.framework.ConcreteProcessObject`. By inheriting from this concrete class, we are saved from the effort of implementing “plumbing” functionality such as tracking current state and other details allow the ProcessObject to function in the Physhun StateEngine.

The next step is to define the State Model for the order process. A visual representation of the model is shown in Illustration 2. The State Model is a spring bean made up of States and Transitions and is easily defined as an XML document. The XML defining the State Model is shown in Table 3. This XML document can be consumed by a Spring bean factory to inflate the State Model as Java objects. The State Model definition is lengthy, but it is straightforward and easy to understand, especially when viewed with a graphical editor.

Once the ProcessObject and State Model have been defined, any custom Actions and Conditions must be implemented. Conditions define the circumstances that must be met for a Transition from one state to another to occur. At runtime the StateEngine evaluates Conditions to determine what Transition (if any) to execute. Actions define the logic that is executed on a given Transition. The code for a simple Condition is shown in Table 4 and an Action in Table 5.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean abstract="false" autowire="default" class="com.wazeegroup.physhun.framework.ConcreteState"
    dependency-check="default" id="newOrder" lazy-init="default" singleton="true">
    <property name="initialState">
      <value>true</value>
    </property>
  </bean>
  <bean abstract="false" autowire="default" class="com.wazeegroup.physhun.framework.ConcreteState"
    dependency-check="default" id="orderCancelled" lazy-init="default" singleton="true">
    <property name="terminationState">
      <value>true</value>
    </property>
  </bean>
  <bean abstract="false" autowire="default" class="com.wazeegroup.physhun.framework.ConcreteState">
```

Table 3: State Model XML

```
package com.wazeegroup.physhun.examples.simple;

import com.wazeegroup.physhun.framework.AbstractCondition;
import com.wazeegroup.physhun.framework.ProcessObject;

import java.util.Random;

public class CheckOrderAvailability extends AbstractCondition {

    private static final Random _random = new Random();

    public boolean evaluate(ProcessObject iProcessObject) {
        boolean result = _random.nextBoolean();
        System.out.println("Order Availability for Order (" + iProcessObject + "): " + result);
        return result;
    }
}
```

Table 4: A simple Condition implementation

```
package com.wazeegroup.physhun.examples.simple;

import com.wazeegroup.physhun.framework.AbstractAction;
import com.wazeegroup.physhun.framework.ProcessObject;

public class CommitOrder extends AbstractAction {
    public void execute(ProcessObject iProcessObject) {
        System.out.println(iProcessObject + " - completed");
    }
}
```

Table 5: A simple Action implementation

Now that the State Model, ProcessObject, Actions and Conditions have been defined, the final step is to load and execute the process. To execute the process, two additional components are needed: A StateEngine and a ProcessContainer. The StateEngine is the component that contains the rules on how State Models executed, and the ProcessContainer is the component that keeps track of process instances and sends events to those instances. The default StateEngine is `com.wazeegroup.physhun.engine.StandardEngine`, and the default ProcessContainer is `com.wazeegroup.physhun.SimpleProcessContainer`. The ProcessContainer and StateEngine are configured by standard Spring configuration as shown in Table 6. The code used to execute the example is shown in Table 7.

```
<?xml version="1.0"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="engine" class="com.wazeegroup.physhun.engine.StandardEngine" />
  <bean id="container" class="com.wazeegroup.physhun.engine.SimpleProcessContainer">
    <property name="processEngine">
      <ref bean="engine"/>
    </property>
  </bean>
</beans>
```

Table 6: Sample ProcessContainer configuration

```

package com.wazeegroup.physhun.examples.simple;

import com.wazeegroup.physhun.engine.ProcessContainer;
import com.wazeegroup.physhun.framework.StateModel;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class RunSample {
    public static void main(String[] args) {
        if (args.length < 3) {
            System.err.println("usage: RunSample custID itemID numItems");
        } else {
            Order order = new Order(args[0], args[1], Integer.parseInt(args[2]));
            //Get the Spring ApplicationContext
            String[] beanFiles = new String[]{"simple-processConfig.xml", "simple-stateModel.xml"};
            ClassPathXmlApplicationContext ctx =

```

Table 7: Execution of the State Model

Example 2: Long Lived Processes and Asynchronous Events

Our first example showed a very simple order management process. In that example, the process is short lived and synchronous – as soon as the process is started, it runs to completion without dependence on incoming events. The second example is similar to the first, but with added asynchronous behavior – in this example, the process waits for and reacts to asynchronous events indicating changes to inventory. When the order is created, if stock is available to fill the order, the order completes normally. Otherwise, the process waits until it receives notification that the appropriate inventory is available at which point the process is completed. The new State Model is shown in Illustration 3.

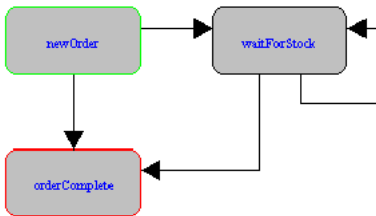


Illustration 3: Asynchronous process State Model

The Transitions from the waitForStock state are triggered by an external event – in this case, notification that inventory changes have occurred. The Transition from waitForStock to orderComplete occurs if the inventory system notifies the process that stock is now available. The waitForStock self-Transition occurs if an inventory change occurs that does not result in appropriate stock being available. In order to define an “asynchronous” Transition, we associate a TriggeredCondition with the Transition. A TriggeredCondition implements the interface `com.wazeegroup.physhun.framework.TriggeredCondition` and is very similar to `Condition`, the difference being that the `evaluate()` method takes an additional `Object` which is the asynchronous event that drives the Transition. The `TriggeredCondition` implementation for this example is shown in [Table 8](#), and the `Condition` and `Transition` definitions from the State Model XML are shown in [Table 9](#).

```

package com.wazeegroup.physhun.examples.async;

import com.wazeegroup.physhun.framework.AbstractTriggeredCondition;
import com.wazeegroup.physhun.framework.ProcessObject;

public class InventoryChanged_orderStockAvailable extends AbstractTriggeredCondition {
    public boolean evaluate(ProcessObject processObject, Object triggerEvent) {
        InventoryChangeEvent evt = (InventoryChangeEvent)triggerEvent;
        System.out.println("Received inventory change event (" + evt.itemID + "/" + evt.newQuantity + " units)");
        Order order = (Order)processObject;
        if (evt.itemID.equals(order.getItemId()) && evt.newQuantity >= order.getQuantity()) {
            //the item we are waiting on is now in stock.
            System.out.println("Item we are waiting for is now in stock. Condition evaluates to true!");
            return true;
        }
        else {

```

Table 8: TriggeredCondition implementation

```

<bean abstract="false" autowire="default" class="com.wazeegroup.physhun.generic.condition.TriggeredDefault"
    dependency-check="default" id="ConditionDefaultTriggered" lazy-init="default" singleton="true"/>

```

```

dependency-check= default id= conditionDefaultTriggered lazy-init= default singleton= true //

<bean abstract="false" autowire="default"
class="com.wazeegroup.physhun.examples.async.InventoryChanged_orderStockAvailable"
dependency-check="default" id="InventoryNowAvailable" lazy-init="default" singleton="true"/>

<bean abstract="false" autowire="default" class="com.wazeegroup.physhun.framework.ConcreteTransition"
dependency-check="default" id="newOrder-orderComplete" lazy-init="default" singleton="true">
  <property name="condition">
    <ref bean="CheckItemInventory"/>
  </property>
  <property name="action">
    <ref bean="CommitOrder"/>
  </property>
  <property name="fromState">

```

Table 9: Asynchronous Transition and Condition State Model definition

In this example, the asynchronous event that drives Transitions from the waitForStock state are notifications of changes to inventory. In an enterprise system, these may be anything from calls to a Web Service to events published on a JMS destination, event queue or other Enterprise Message Bus.

The final step is to bridge the asynchronous events into the running process. This is done via the ProcessContainer by calling the method sendTriggerEventToProcess(). In our example, we mock up inventory change events by spawning a thread that sends mocked up inventory change events to the ProcessContainer. The code used to execute this example and send asynchronous events to the process instances is shown in Table 10.

```

package com.wazeegroup.physhun.examples.async;

import com.wazeegroup.physhun.engine.ProcessContainer;
import com.wazeegroup.physhun.framework.StateModel;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.Random;

public class RunSample {
  public static void main(String[] args) {
    if (args.length < 3) {
      System.err.println("usage: RunSample custID itemID numItems");
    } else {
      Order order = new Order(args[0], args[1], Integer.parseInt(args[2]));
      ClassPathXmlApplicationContext ctx = new
        ClassPathXmlApplicationContext(new String[]{"asynch-processConfig.xml"},

```

Table 10: Execution of the Asynchronous State Model

Example 3: Distributed Transactions and ProcessObject Persistence

In the final example, we add a distributed transaction and ProcessObject persistence to our process. ProcessObject persistence writes the state of the process to a datastore. This is important for two reasons: it allows data to be restored in the event of a system failure, and it allows the construction large scale systems. Systems can run with many concurrent processes without keeping all Process Instances in memory at all times.

The process in this example is identical to the process in Example 2, but we change the commitOrder Action (which is the Action associated with the Transition from waitForStock to orderComplete) to do the following: reserve inventory in the inventory system, enter the order in the billing system and enter the order in the provisioning system. All three of these steps are executed against different systems, but must be done as a single transaction. For this example, our interface to each of the target systems is through the database. To accomplish this distributed transaction, we must do three things in our implementation:

1. Code individual Actions to work with a common transaction manager. To accomplish this, we wire an appropriately configured DataSource object in to each Action, and modify the Actions to acquire DB connectivity through the DataSource.
2. Configure a Spring transaction manager and wire it into the StateEngine and DataSource beans.
3. Define a composite Action that aggregates the individual Actions that make up the distributed transaction.

The XML for the composite Action is shown in Table 11 and a representative Action class implementation is shown in Table 12. Note that each of the Actions use a DataSource. The DataSources are standard Spring JDBC DataSources with a common TransactionManager wired in. That same TransactionManager is also wired in to the Physhun StateEngine. The code in the Actions is simple; it uses the DataSource to acquire a DB connection, through which it manipulates data.

```

<bean class="com.wazeegroup.physhun.generic.action.Composite" id="CommitOrder">
  <property name="actions">
    <list>
      <bean class="com.wazeegroup.physhun.examples.transactional.ReserveInventory"
id="reserveInventory">
        <property name="dataSource">

```

```

        <ref bean="inventoryDataSource"/>
    </property>
</bean>
<bean class="com.wazeegroup.physhun.examples.transactional.BillOrder"
id="billOrder">
    <property name="dataSource">
        <ref bean="billingDataSource"/>
    </property>
</bean>
<bean class="com.wazeegroup.physhun.examples.transactional.SendOrderToWarehouse"

```

Table 11: Composite Action definition for a distributed transaction

```

package com.wazeegroup.physhun.examples.transactional;

import com.wazeegroup.physhun.framework.ProcessObject;
import com.wazeegroup.physhun.framework.PhyshunException;
import com.wazeegroup.physhun.framework.AbstractAction;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

import java.sql.Types;

public class ReserveInventory extends AbstractAction {

    private DataSource dataSource;
    private JdbcTemplate _jdbcTemplate;

```

Table 12: Implementation of Action class that is used in a distributed transaction

Note that ProcessObject persistence does not necessarily have to be implemented from scratch. The PhyshunXML package provides a ProcessObject implementation whose data is all stored internally in an XML Document. This package also provides persistence functionality to Oracle and MySQL databases. If desired, implementation of custom persistence is straightforward. For the sake of illustration this example demonstrates implementation of custom ProcessObject and persistence functionality.

To add ProcessObject persistence, we must do two things:

1. Define a ProcessObjectPersistence class. This is where we define how to store and retrieve the ProcessObject to and from the persistence layer.
2. Wire the ProcessObjectPersistence class defined in step 1 into the StateEngine.

When the StateEngine has a ProcessObjectPersistence object wired in, it will persist the ProcessObject on execution of each state Transition. Furthermore, if the StateEngine has a Transaction Manager wired in, it will persist the ProcessObject as part of the state Transition transaction. This transaction includes persistence of the ProcessObject and execution of any Actions associated with the Transition. The example ProcessObjectPersistence class is shown in Table 13 and the Container and StateEngine XML are shown in Table 14.

Note that the StateEngine and DataSources for Billings, Warehouse, Inventory and Orders (ProcessObject persistence store) databases all use the same TransactionManager. Because the beans are wired in this manner, when the StateEngine executes a Transition, persistence of the ProcessObject and Actions on the transition will be executed on the same transaction. So, if a single Action, or persistence of the ProcessObject fails, the entire transaction will fail as a whole and no data will be written to any of the data sources. The Transition will be rolled back and the process will roll back to the state from which the Transition occurred.

```

package com.wazeegroup.physhun.examples.transactional;

import com.wazeegroup.physhun.framework.ProcessObjectPersistenceSupport;
import com.wazeegroup.physhun.framework.ProcessObject;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowCallbackHandler;
import java.sql.Types;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;

public class OrderPersistenceSupport implements ProcessObjectPersistenceSupport {

    private DataSource dataSource;
    private JdbcTemplate _jdbcTemplate;

```

Table 13: ProcessObjectPersistenceSupport implementation

```
<?xml version="1.0"?>
```

```

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="propertyConfigurator" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="jdbc.properties"/>
  </bean>
  <bean id="jotm" class="org.springframework.transaction.jta.JotmFactoryBean"/>
  <bean id="engineTxManager" class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="userTransaction">
      <ref local="jotm"/>
    </property>
  </bean>
  <bean id="inventoryDataSource" class="org.enhydra.jdbc.pool.StandardXAPoolDataSource"
    destroy-method="shutdown">
    <property name="dataSource">
      <bean id="inv_innerDataSource" class="org.enhydra.jdbc.standard.StandardXADataSource"

```





Table 14: XML definitions for StateEngine and supporting beans

Conclusion

Finite State Machine technology offers a useful, alternative paradigm to implementing complex logic and process flow. The Physhun project is an open source framework that enables rapid development of systems using Finite State Machine technology, while leveraging and exposing the power and flexibility of the Spring Framework. More information about the Physhun project, including examples, source code and binaries can be found on the Physhun project homepage at <http://physhun.sourceforge.net>.

🔍 Dig Deeper on Software development techniques and Agile methodologies

ALL NEWS GET STARTED EVALUATE MANAGE PROBLEM SOLVE

-  [How to become a good software architect in 13 steps](#)
-  [IT projects and software teams need to include Agile people](#)
-  [Why the Waterfall or Agile debate will be around forever](#)
-  [Microsoft supports trend toward containers, serverless computing](#)

[Load More](#)

🔍 Start the conversation

Share your comment

Send me notifications when other members comment.

-ADS BY GOOGLE

+15 million jobs. Find yours.

Search ASIC Engineer Jobs On LinkedIn. Find Your Dream Job Today. [linkedin.com](https://www.linkedin.com)

[CLOUD APPLICATIONS](#) [SOFTWARE QUALITY](#) [HR SOFTWARE](#) [SAP](#) [ERP](#) [DEVOPSAGENDA](#)

SearchCloudApplications

Secure data in the cloud with encryption and access controls

Enterprises can't rely on their IaaS provider to protect sensitive information in the cloud. Implement these data security ...

Microsoft Azure Dev Spaces, Google Jib target Kubernetes woes

Microsoft Azure and Google Cloud both added cloud application development tools that improve and simplify the process of creating...

[About Us](#) [Meet The Editors](#) [Contact Us](#) [Privacy Policy](#) [Advertisers](#) [Business Partners](#) [Media Kit](#) [Corporate Site](#)

[Contributors](#) [Reprints](#) [Archive](#) [Site Map](#) [Answers](#) [Definitions](#) [E-Products](#) [Events](#)

[Features](#) [Guides](#) [Opinions](#) [Photo Stories](#) [Quizzes](#) [Tips](#) [Tutorials](#) [Videos](#)

All Rights Reserved,
Copyright 2000 - 2018, TechTarget