```
class CityController implements
 ApplicationEventPublisherAware {
  ApplicationEventPublisher publisher;

  public int addCity(String cityName) {
    // add the city into the database

    ReloadCacheApplicationEvent event=
   new ReloadCacheApplicationEvent(this,
“cities”);
    publisher.publishEvent(event);

    return cityId;
  }

  void setApplicationEventPublisher(
   ApplicationEventPublisher publisher){
     this.publisher = publisher;
  }
}
```

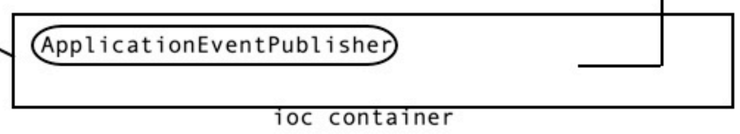
```
class ReloadCacheApplicationEvent
      extends ApplicationEvent {
  String key;

  ReloadCacheApplicationEvent
   (Object source, String key) {
    super(source);
    this.key = key;
  }
  // getter
}
```

```
@Component
class ReloadEventApplicationListener
  implements ApplicationListener
  <ReloadCacheApplicationEvent>{
  @Autowired
  CacheManager cacheManager;

  public void onApplicationEvent(
   ReloadCacheApplicationEvent e) {
     cacheManager.reload(e.getKey());
   }
}
```

```
class CacheManager {
  void reload(String key) {
    sop(“reloading the data of
the key : “ + key);
  }
}
```



(OR)

```
@Component
class ReloadEventApplicationListener {
  @Autowired
  CacheManager cacheManager;

  @EventListener
  public void onReloadCacheEvent(
   ReloadCacheApplicationEvent e) {
     cacheManager.reload(e.getKey());
   }
}
```

```
ApplicationEventPublisher
- void publishEvent(ApplicationEvent)
```

If CityController is directly invoking CacheManager.reload() then we run into 2 problems
1. all the classes in our application whoever is talking to the CacheManager will be tightly coupled with the CacheManager. In future if we change the Cache implementation strategy, then the code across all the classes who are talking to CacheManager will be impacted

2. The enduser's request will be blocked on the CacheManager.reload(..), because upon CityController invoking the CacheManager, the control will be transfered to the CacheManager keeping the request waiting until the reload has been finished, this will eventually leads to connection timeout and an error will displayed to the user

here there is no point in blocking the user request until Cache gets reloaded, the reloading the Cache can be taken asynchronously without block the user.

To overcome these 2 problems we can use event-driven programming.

1. upon the CityController adding a new city into the database, he has to ask the CacheManager to reload the data, this can be done by publishing an event. Event is an class object encapsulated with source, action and data

So representing the action we want to perform we need to create our own Event class like ReloadCacheEvent. All the event classes has commonly hold or represent the source who is publishing the event. So spring framework has provided an base abstract class called ApplicationEvent

```
abstract class ApplicationEvent {
  Object source;

  ApplicationEvent(Object source) {
    this.source = source;
  }
}
```

2. EventListener = The ioc container itself acts as an EventListener in listening for incoming events and identifies an appropriate handler, that has been registered with him and invokes

3. EventHandler = In spring the naming convention of EventHandler is an “Listener” since this class is being invoked by the spring ioc container, we should write this class by implementing from spring provided standard interface

```
interface ApplicationListener<E> {
  void onApplicationEvent(E event);
}
```

4. how to publish the event?
To publish the event we need a class that is capable of taking the event and passing it to ioc container. So spring framework has provided an class called “ApplicationEventPublisher” and it is an implicit or internal object of ioc container that is available by default.
To the ApplicationEventPublisher internal object of ioc container inside our component we need to use Aware injection. Spring has provided an Aware interface called

```
interface ApplicationEventPublisherAware {
  void setApplicationEventPublisher(ApplicationEventPublisher);
}
```