A Button-Led systems: from objects to actors and services

DISI-Cesena
edited by Antonio Natali

Alma Mater Studiorum – University of Bologna
via Venezia 52, 47023 Cesena, Italy

Table of Contents

A Button-Led systems: from objects to actors and services 1
DISI-Cesena edited by Antonio Natali
1 Requirements 2
1.1 Goals 2
1.2 Software life cycle process 2
1.3 Bottom-up or Top-Down? 3
2 Technology-based design 4
2.1 Function-based software 4
2.2 Event-driven servless applications: AWS Lambda 4
2.3 Lexical Closures 5
2.4 Objects 6
2.5 Observable and Observers 6
2.6 Models 7
2.7 Modelling a Button 7
3 An object-based design 8
3.1 A Led as an object in Javascript 9
3.2 From BLS Mock devices to physical devices 10
3.2.1 Dependency injection and Inversion of Control 10
3.3 From a BLS local system to a BLS distributed 10
4 Evolving the BLS into a distributed, event-driven system 11
5 Evolving the BLS into a message-driven (AKKA) system 11
6 Evolving the BLS into a RESTful system 11
6.1 Spring 11
6.2 BLS using Spring 12
6.3 Running 13
7 Evolving the BLS into a message-based, distributed heterogeneous system 14
8 Devices as microservices 14

1 Requirements

Every IOT system usually performs a basic set of actions:

- Acquire data from sensor devices.
- Perform some control action.
- Send commands to actuator devices.

In this very basic demo, we use a Button as a sensor and a Led as an actuator and the control action represents our business logic. Examples of the business logic implemented by our Button-Led (BLS) system are:

1. ROnOff: the Led is `turned on/off` each time the Button is pressed.
2. RBlink: when the Button is pressed, the Led `starts blinking`. When the Button is pressed again, the Led `stop blinking`. And so on.
3. ...

> Reference project: it.unibo.bls.oo on `https://github.com/anatali/IotUniboDemo`.

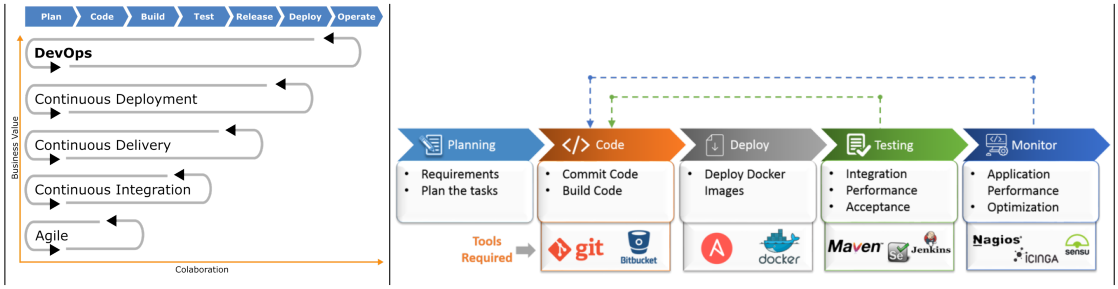
1.1 Goals

Our goals can be summarized as follows:

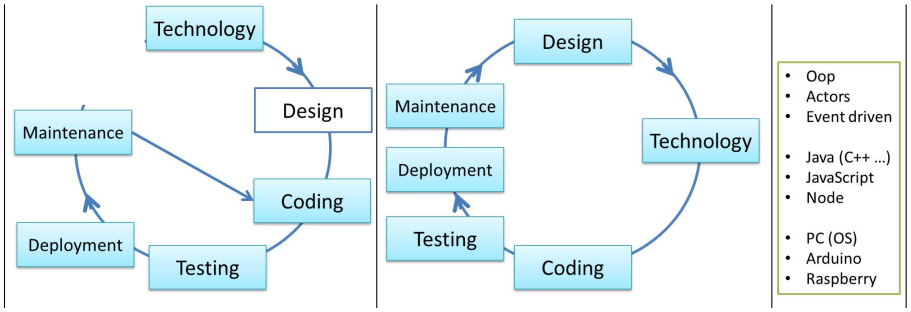
1. Define a 'technology-independent' architecture/prototype of the `BLS` in a local, tightly coupled environment.
2. Specialize the initial prototype according to different technologies. For example:
   (a) The devices are implemented as Mock objects in a virtual environment.
   (b) The devices are concrete things controlled by low-costs devices such as `Arduino/RaspberryPi`.
3. Modify the first working prototype into a loosely-coupled (distributed) system in which each device works within its own computational node: see Subsection 3.3.
4. Modify the distributed prototype by 'transforming' each device into a (RESTful) service.
5. ...

1.2 Software life cycle process

> Discussed in BBS-module5 : Software production.

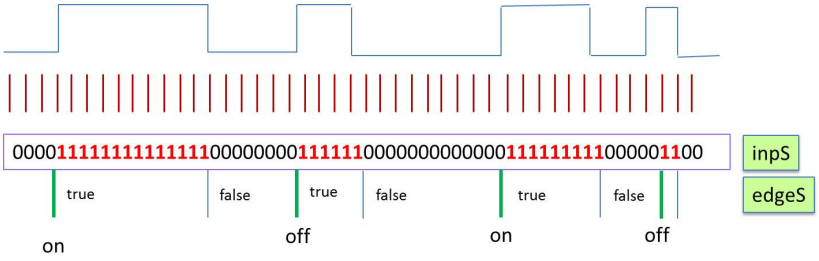


1.3 Bottom-up or Top-Down?



2 Technology-based design

The button is a source that emits a wave that can be sampled.

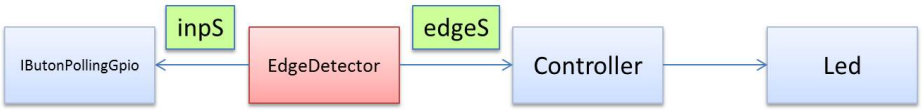


At this level, the problem requires that the following elaborations on the basic input

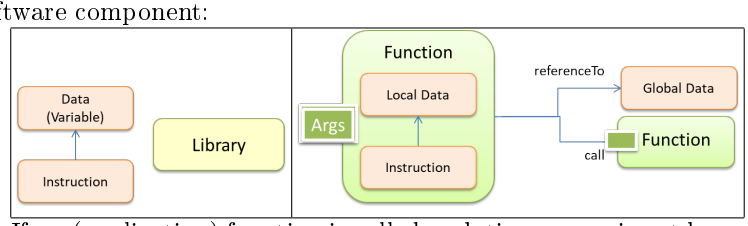
- the detection of the edges in the input sequence
- the detection of edges of type "low to high"

2.1 Function-based software

The responsibility of these functions can be given to two new different entities: an entity *EdgeDetector* and an entity *Controller* that realizes the "business logic" of the system.



The code can be structured in imperative style, by using `functions` as a first kind of software component:



If an (application) function is called each time a new input becomes available, the system is 'reactive' or `event-driven`.

2.2 Event-driven servless applications: AWS Lambda

Usually a function is associated with a definition environment and a run-time environment.

`Serverless computing` is a cloud-computing execution model in which the cloud provider dynamically manages the allocation of machine resources, including those required to execute a function.

`AWS Lambda`, introduced by Amazon in 2014,was the first public cloud vendor with an abstract serverless computing offering. AWS Lambda initially supported only Node.js. It now supports `Python, Java, C#` and `Go`, and code written in other languages can be invoked indirectly via Node.js.

Before running an example of a function as an AWS Lambda , we:

1. Create the function in natAwsServices
2. Install the client interface (`AWS CLI`) on the PC (run `AWSCLI64PY3.msi` for Windows)
3. Work in the installed `AWS CLI` directory (`C:/Program Files/Amazon/AWSCLI/bin`)

The function defined at natAwsServices named blsOnOffEdge has the following code:

```
var state = 'off'; //Initialization (state of a Led Mock)

exports.handler = (event, context, callback) => {
    console.log('Received event:', JSON.stringify(event, null, 2));
    console.log('edge =', event.edge);
     if ('edge' in event) {
        state = event['edge'] ;
    }
    var ledState = 'Led is ' + state ;
    console.log(ledState);   //Logging to Amazon CloudWatch Log
    callback(null, ledState); // End and return
};
```

The function can be called by the AWS Console on the Web in two ways

- without any testing event
- with a testing event expressed in Json:

```
{
  "edge": "off"
}
```

As an alternative, we can call the function from a AWS *Command Line Interface* (AWS-CLI) Installed on our PC. Before doing that, we must configure a (client) user on the PC by using the command aws configure (in C:/Program Files/Amazon/AWSCLI/bin):

```
AWS Access Key ID [None]: ...BCG4A
AWS Secret Access Key [None]:...T8eW
Default region name [None]: eu-west-1
Default output format [None]: json
```

To find the Access Key (ID):

1. go to the security credentials on your account page (Click on your name in the top right corner -> My security credentials;
2. generate access keys over there and use those access keys in your credentials file (aws configure)

Finally we can invoke the function via the AWS-CLI:

```
aws lambda invoke --function-name blsOnOffEdge --payload "{""edge"" : ""on""}" output.txt
aws lambda invoke --function-name blsOnOffEdge --payload "{""edge"" : ""off""}" output.txt
```

in this case the function is executed via the AWS Lambda Invoke API call.

An important point at this stage, is that we can experiment in an easy way some fundamental aspects of functional programming (in JavaScript/Node), including the concept of lexical closure (lambda in Java8) .

2.3 Lexical Closures

> Operationally, a closure is a record storing a function together with an environment: a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.

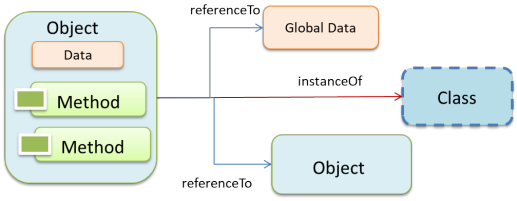
An example is the function named ClosureExample:

```
function ledClosure(state){
   var doSwitch = function( ){
      state = ! state;
      return state
    }
```

```
6     return doSwitch
7   }
8
9   var ledSwitch = ledClosure(false);
10
11  exports.handler = (event, context, callback) => {
12     console.log( ledSwitch() );
13     console.log( ledSwitch() );
14     console.log( ledSwitch() );
15     callback(null, "done"); // End and return
16  };
```

2.4 Objects

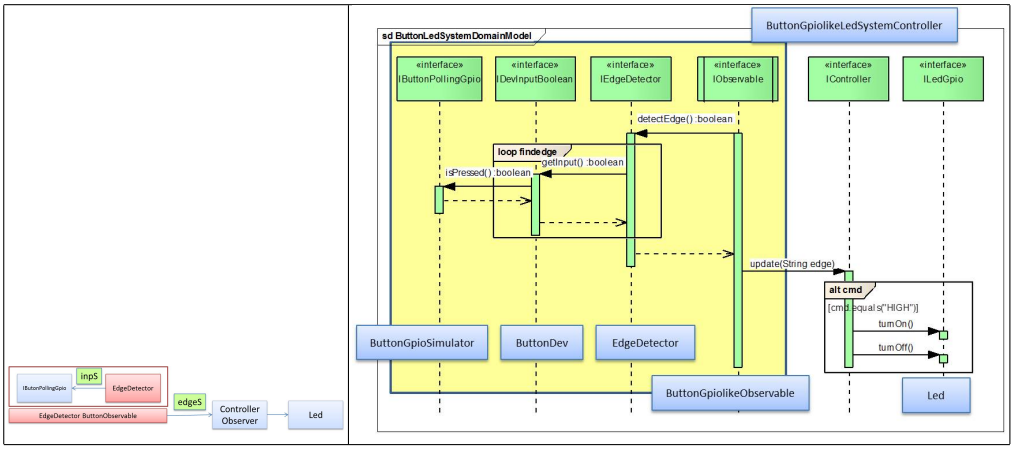
As an alternative of the function-based code of Subsection 2.1, both the *EdgeDetector* and the *Controller* can be modelled as finite state machines (`FSM`) working as *transducers*. They can be viewed as *objects* interacting via procedure-calls.



In any object-oriented model, all the computation usually takes place within a single thread. In our case the main thread could be the thread related to the component that performs the polling of the wave, i.e. the `EdgeDetector`. In this case, the *Controller* is called by the *EdgeDetector* that, must explicitly know the *Controller* in order to call it.

2.5 Observable and Observers

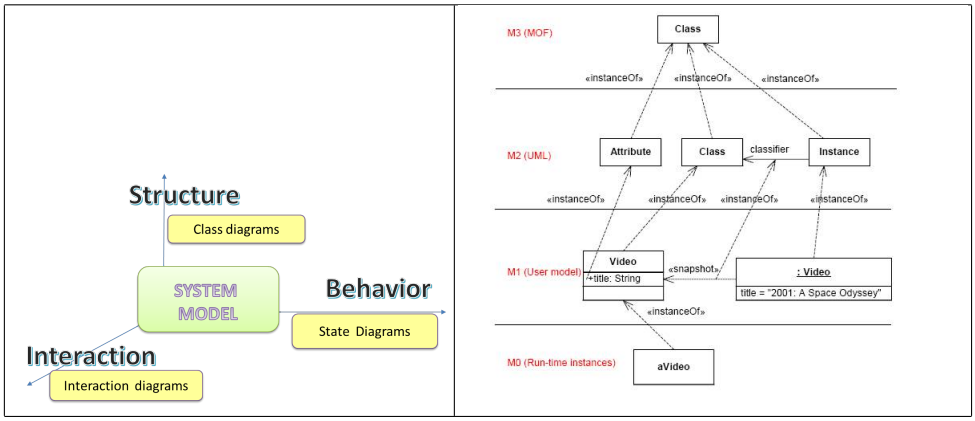
A more flexible architecture an be obtained (without changing the run-time interaction pattern) by conceiving the *Controller* as an *observer* that can be registered to the *EdgeDetector* information source.



However, staring from the idea that a Button is an 'edge detector' device is a too low-level approach for modern software applications. An effort has to be made to introduce a more appropriate `model` of the `Button` entity in terms of structure, interaction and behavior.

2.6 Models

Nel contesto dei processi di costruzione del software, il termine `modello` va primariamente inteso come un insieme di concetti e proprietà volti a catturare `aspetti essenziali` di un sistema, collocandosi in un preciso `spazio concettuale`. Per l'ingegnere del software quindi un modello costituisce una visione semplificata ma **coerente, consitente ed efficace** di un sistema che rende il sistema stesso più accessibile alla **comprensione e alla valutazione** e **facilita il trasferimento di informazione** e la collaborazione tra persone, soprattutto quando è espresso in forma visuale.



2.7 Modelling a Button

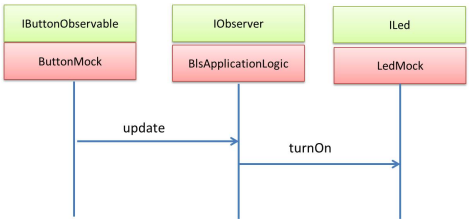
From the `structural` point of view, a button is intended by a customer as an *`atomic`* entity whose `behavior` can be modelled as a *`finite state machine`* (`FMS`) composed of two states ('`pressed`' and '`unpressed`'). The state transition is performed by some agent *external* to the system (an user, a program, a device, etc.). From the `interaction` point of view, the button can expose its internal state in different ways:

- by providing a `property` operation (e.g. `boolean isPressed()`) that returns `true` when the button is in the `pressed` state. In this way the interaction is based on "`polling`";
- by providing a `synchronizing` operation (e.g. `void waitPressed()`) that blocks a caller until the button transits in the **pressed** state. In this way the interaction is based conventional "`procedure-call`";
- by working as an `observable` according to the *Observer* design pattern. In this way the interaction is based on "`inversion of control`" and involves observers (also called "*listeners*") that must be explicitly referenced (via a "*register*" operation) by the button.
- by emitting `events` handled by an event-based support. In this way the interaction is based on "inversion of control" that involves observers (usually known as "*callbacks*") referenced by the support and not by the button itself.
- by sending `messages` handled by a message-based support. In this way the interaction is based on message passing and can follow different "patterns" (in our internal terminology we distinguish between *dispatch*, *invitation*, *request-response*, etc.)

All these "models" could be appropriate in some software application. Thus, a very useful exercise is to define in a formal way each of these models by adopting (at the moment) a test-driven approach.

3 An object-based design

Working in the conceptual space of 'classical' object oriented software development, the logic architecture of the BLS can be summarized by the following UML interaction diagram:



The computation starts from the observable device (*Button*), that calls a method of the object devoted to implement the application logic (that works as the `Observer`). An example is given in the project `it.unibo.bls.oo` that is based on the `Java` lanaguage. The working directory of the project is structured as follows:

- The package `it.unibo.bls.interfaces` includes the definition of the object interfaces.
- The package `it.unibo.bls.devices` includes the implementation of the object interfaces related to the devices (`Button` and `Led`). For each device, two different implementations are given: a `Mock` device and a 'virtual' object implemented with a `GUI`.
- The package `it.unibo.bls.applLogic` includes the definition of the object that implements the application logic.
- The package `it.unibo.bls.appl` includes the `Main` programs.
- The `test` directory includes examples of test units.

A software system working with the `Mock` devices should include a `configuration phase` to create the system components (objects) and properly connect them, according to the system architecture design:

```
1  public class MainBlsMockBase {
2  private IButton btn;
3  private ILed led;
4  //Factory method
5     public static MainBlsMockBase createTheSystem(){
6         return new MainBlsMockBase();
7     }
8     protected MainBlsMockBase( ) {
9         createComponents();
10    }
11    protected void createComponents(){
12        led = LedMock.createLed( );
13        BlsApplicationLogic applLogic = new BlsApplicationLogic(led);
14        btn = ButtonMock.createButton( applLogic );
15        led.turnOff();
16    }
```

Listing 1.1. `MainBlsMockBase.java`: configuration

The system defines also some working activity, to cause the change of the state in the `ButtonMock`:

```
1     public void doSomeJob() {
2         System.out.println("doSomeJob starts" );
3         ((ButtonMock)btn).press();
4         Utils.delay(1000);
5         ((ButtonMock)btn).press();
6         System.out.println("doSomeJob ends" );
7     }
```

Listing 1.2. `MainBlsMockBase.java`: simulated action

Since every system is a **composed** (i.e. it is a non-atomic) entity, it provides also **selector-methods** to get its components:

```
    public IButton getButton(){ //introduced for testing
        return btn;
    }
    public ILed getLed(){ //introduced for testing
        return led;
    }
```

Listing 1.3. `MainBlsMockBase.java`: selectors

The selectors can be useful during the testing, to access the state of the single devices. Finally, there is the main method:

```
public static void main(String[] args) {
   MainBlsMockBase sys = createTheSystem();
   sys.doSomeJob();
 }
}
```

Listing 1.4. `MainBlsMockBase.java`: the main method

3.1 A Led as an object in Javascript

Let us model a Led as an object in JavaScript

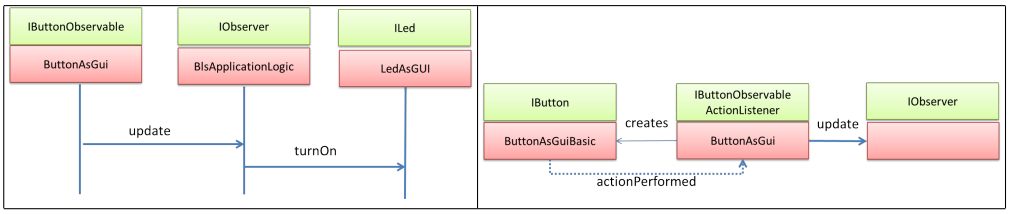
```
//State (specific)
var Led = function(name, guiId){
    //Led Constructor: instance data
    this.name     = name;
    this.guiId    = guiId;
    this.ledState = false;
}

//Methods (shared)
Led.prototype.turnOn = function(){
    this.ledState = true;
}
Led.prototype.turnOff = function(){
    this.ledState = false;
    }
Led.prototype.switchState = function(){
    this.ledState = ! this.ledState;
    }
Led.prototype.getState = function(){
    return this.ledState;
}
Led.prototype.getName = function(){
    return this.name;
}
Led.prototype.getDefaultRep = function(){
    return this.name+"||"+ this.ledState
}
```

The function defined at natAwsServices named `blsROnOff` has the following code:

```
 //Initialization
var ledMod = require("./Led");
var l1    = new ledMod.Led('l1', null);
//Entry
exports.handler = (event, context, callback) => {
    console.log('Received event:', JSON.stringify(event, null, 2));
    console.log('edge =', event.edge);
     if ('edge' in event) { //change the led state at each edge
        l1.switchState();
     }
    console.log( l1.getDefaultRep() ); //Logging to Amazon CloudWatch Log
    callback(null, l1.getDefaultRep()); // End and return
};
```

3.2 From BLS Mock devices to physical devices

The logical architecture previously introduced does not change if we replace the `Mock` devices with concrete devices: For example, in the case of virtual devices implemented with a `GUI`:



The class `ButtonAsGuiBasic` implements a `GUI`-based Button by extending the class `java.awt.Button`. The class `ButtonAsGui` implements the concept of `Button` as `Observable` entity.

Note that `ButtonAsGui` re-uses the class `ButtonAsGuiBasic` but without exploiting inheritance. Rather, it creates an instance of `ButtonAsGuiBasic` and works as its listener. This behaviour is caused by the fact that `Java` does not support multiple inheritance for classes and `ButtonAsGui` already extends the class `java.util.Observable`.

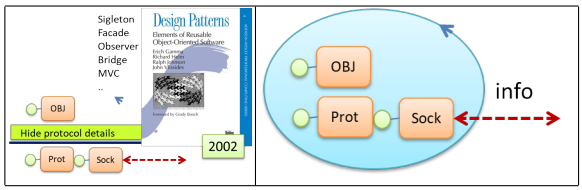
In a more general perspective, we could build this part of our software system by exploiting the `Dependency Injection` pattern.

3.2.1 Dependency injection and Inversion of Control .

In software engineering, an `injection` is the passing of a dependency to a dependent object (a client) that would use it as a service. In the `dependency injection` pattern, passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement.

> Dependency injection is one form of the broader technique of `Inversion of Control`. The client delegates the responsibility of providing its dependencies to external code (the `injector`). The client is not allowed to call the injector code.

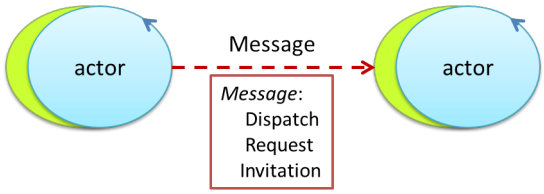
3.3 From a BLS local system to a BLS distributed



4 Evolving the BLS into a distributed, event-driven system

The author of section is Angelo Croatti.
Project it.unibo.bls.oo/it.unibo.bls.vertx on https://github.com/anatali/IotUniboDemo

5 Evolving the BLS into a message-driven (AKKA) system



The author of section is Angelo Croatti.
Project ... on https://github.com/anatali/IotUniboDemo

6 Evolving the BLS into a RESTful system

The author of section is Roberto Casadei.
Projects it.unibo.bls.oo/it.unibo.bls.spring.ledService,
it.unibo.bls.oo/it.unibo.bls.spring.buttonService
on https://github.com/anatali/IotUniboDemo

A straightforward way to enable seamless distribution of the BLS as well as heterogeneity in the implementations of its components is to adopt a Service-Oriented Architecture (SOA), where the system components become services, with a well-defined interface (contract), that interact using interoperable protocols.

ReST (REpresentational State Transfer) is a particular architectural style for a SOA; it defines some key principles and constraints regulating how services have to be designed, most prominently, the stateless-ness of servers and uniform interface (based on HTTP methods, media types, URIs, manipulation of resources through representations, self-contained messages and HATEOAS).

6.1 Spring

Spring Boot is a framework that supports the development of RESTful services. It builds on the Spring Framework which, at its core, is an IoC framework; the idea is to declare "managed components" (called "beans" in Spring) through Java annotations or XML, wire those beans in other components (through field, constructor, or setter injection), and then let the framework deal with the injection of components and management of their lifecycle.

In order to enable Spring Boot in a Gradle project, you need to extend your build definition (build.gradle) with:

```
buildscript { // This block is e.g. for adding dependencies *to the build itself*
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:2.0.2.RELEASE")
    }
}

apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

sourceSets {
    main { resources { srcDir 'resources' } } // Declares directory for resources
}

dependencies {
  // .....
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-web'
}

bootJar {
    mainClassName = 'it.unibo.bls.spring.buttonService.MainButtonService'
    baseName = 'unibo-button'
    version = '1.0.0'
}
```

The task bootJar builds a Fat JAR (i.e., the application code + all the dependencies) under /build/libs; the idea (cf., microservices) is to have a single artefact (also containing the webserver!) that can be easily deployed. (Unfortunately you can only define one "boot JAR"; this is coherent with the microservices philosophy, where each service gets its own repository and build!)

6.2 BLS using Spring

An example of the BLS using Spring is in package it.unibo.bls.spring: two services are defined for the button and the led, in sub-packages 'button' and 'led', resp. Each service consists of:

- A main class, annotated with '@SpringBootApplication', which is the same as having '@Configuration' (this class may declare beans), @EnableAutoConfiguration (attempts to guess and configure beans that you are likely to need by analysing the classpath), @ComponentScan (registers beans by looking for classes annotated with @Component in this package and subpackages), that starts the Spring application for the service (i.e., we give control to Spring for managing the declared components).
- A resource file, under /resources/application-xxx.yml, defining some configuration for the service (e.g., the Spring application name, and the port to which the server should bind).
- A REST controller class (cf., MVC pattern), annotated with '@RestController', which defines the (RESTful) HTTP interface for the service by annotating methods with @RequestMapping(value='uri/path).
- A managed component class, annotated with @Component, that appropriately wraps the component (e.g., the button or the led).

Considers, for instance, the implementation of 'LedController'

```
@RestController @RequestMapping(value = "/led")
public class LedController {
    @Autowired private ILed led;

    @RequestMapping(value = "/state")
    public Boolean state(){
        return led.getState();
    }

    @RequestMapping(value = "/turnOn")
    public void turnOn(){
        led.turnOn();
    }

    @RequestMapping(value = "/turnOff")
    public void turnOff(){
        led.turnOff();
    }
}
```

Notice the use of annotation @Autowired to enable field injection with a bean conforming interface Iled. Such bean is defined as follows:

```
@Component
public class LedComponent extends LedMock { }
```

i.e., it is a candidate for auto-detection when using annotation-based configuration and classpath scanning. Notice that by default the "scope" of a component is "singleton" (i.e., one instance per container); other scopes are e.g. "request" (per HTTP request) and "session" (per HTTP session).

6.3 Running

When the Spring application is started, the framework will start an embedded server (cf., application containers with WAR deployment), binding at the configured port, and will use the declared controllers to map HTTP requests to methods. From the point of view

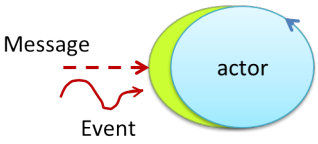
of concurrency, it uses the Servlet model, i.e., one thread per request. Notice that the 'ButtonComponent', which internally reuses 'BlsApplicationLogic', is prone to race conditions (cf., Isolation property of ACID).

In order to run the system, launch the following main classes:

- 'MainButtonService' (NOTE: you also need to pass the JVM option '-Djava.awt.headless=false')
- 'MainLedService'

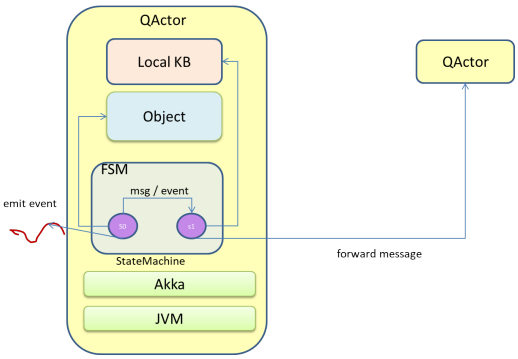
Then, you can press the button by performing a GET request at http://localhost:8080/button/press , and inspect the state of the led by performing a GET request at http://localhost:8081/led/state .

7 Evolving the BLS into a message-based, distributed heterogeneous system



The actor can handle messages/events:
1) as a FSM
2) as a reactor

Events can be emitted by *asynchronous operations*
write on a file
send a request
...



8 Devices as microservices

Deepened in BBS-module7 : Web and Mobile.