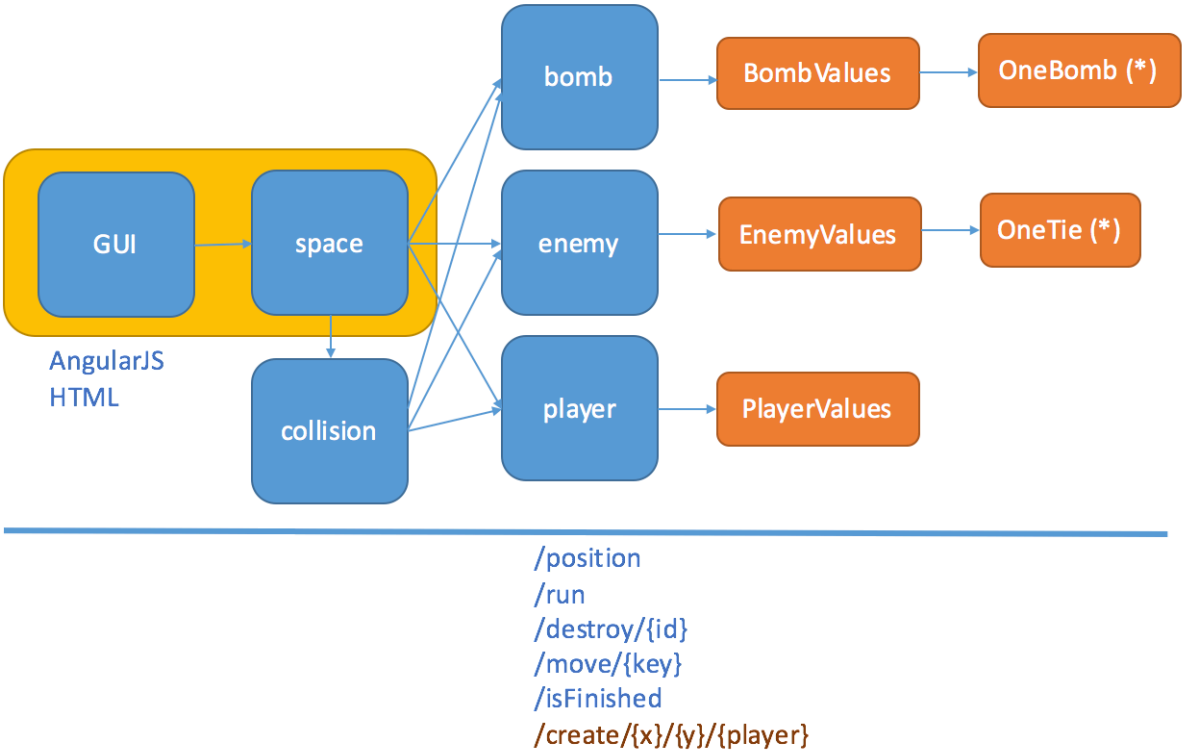
Microinvader

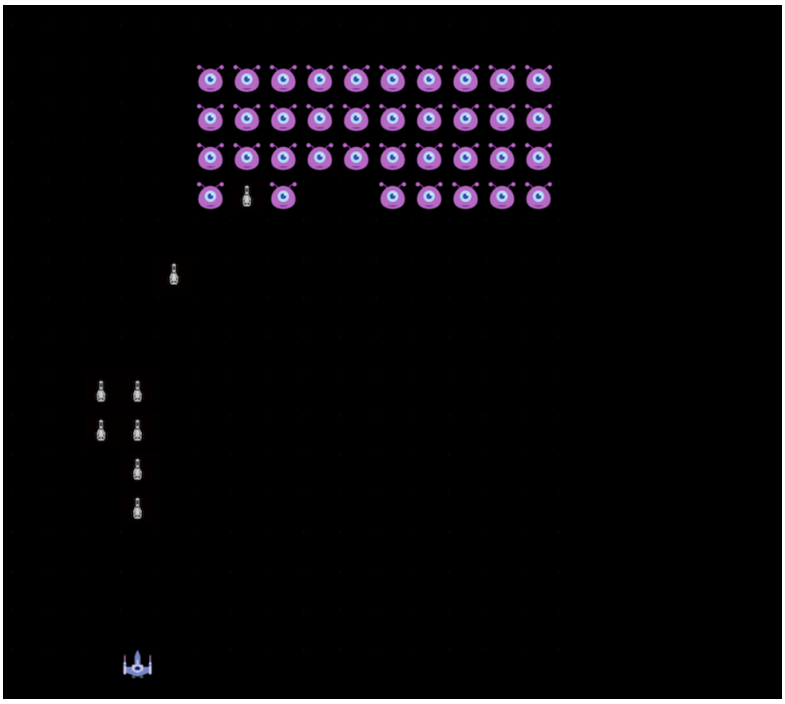
By Glauco Reis (gsreis@br.ibm.com.br) (glauco@portalbpm.com.br)

Microinvader is an simplistic implementation of space invaders game (very simple for education purposes) created to explore principles of Microservices. The game is fully functional and visual also, and since there are moving pieces around screen, some topics (like Circuit breaker) can visually be shown.

The diagram of Microinvader (version 1.0) is:



and the screenshot of game is:



Bombs, Player and Enemies are independent MicroServices. It runs controlled by Space, that is a service orchestrator (Gateway or Front Controller) for entire game. All communications are handled by Space. We did and effort to make Bomb, Enemy and Player independent each other, to make an easy replacement of pieces.
Since there is no communication between them, we need a special MicroService called Collision, responsible for detect when two pieces into to the same position should be destroyed.
The game is a grid of 20x20 positions (today is hard coded). The user interface is a rich client made with HTML and AngularJS. No JSP, Servlet, JSF, Struts or other Server side User's interface were used.
Today one of main principles of Microservices (should be stateless) is not respected. Since the program is used to explain concepts, the option was to start simple, and them evolve to make it more compliant with principles. So it will be changed further to aggregate more functionalities.
The user interface is a simple table with a grid (20x20).

```
<!DOCTYPE html>
<html lang="en-US">
<script src="./scripts/angular.min.js"></script>
<script src="./scripts/controller.js"></script>

<body ng-app="space" ng-controller="control" style="background: black;" ng-
keypress="keyPressed($event)">
        <table style="border: 0; margin: 0; padding: 0">
                <tbody>
                        <tr ng-repeat="y in [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]"
                        ng-init="parentIndex = $index">
                                <td ng-repeat="x in
```

```
                                    [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]"
                                    width="32" height="32"><img
                                    src="{{grid[$parent.$index][$index]}}"></td>
                            </tr>
                      </tbody>
                </table>
</body>
</html>
```

The code can be found on space.war->game.html.
There is an AngularJS controller, with a grid of elements.

```
        $scope.grid = [[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
                       [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
                      ];
```

The dynamics of game is handled by one html page using a timer.
The process consists to call a set of services in a sequence, and populate information on grid. Since grid is connected to table using AngulaJS, all elements will be updated with no flicker. As we can see in the presentations video, there is no smooth and the game seems to have cohesion.
The main routine, managed by Angular is:

```
$scope.callAtInterval = function() {
        // some code removed
        if (!$scope.finished) {
            $scope.callRestRun();      // update pieces position
            $scope.callRestPosition(); // get position after move
            $scope.populateGrid();     // redraw screen
        }
        $scope.callRestFinished(); // check if game ends (Update finished flag)
  }
```

While the game is not finished:

- callRestFinished test if player or enemies are destroyed and set finished flag
- callRestRun move all pieces one position ahead
- callRestPosition get new position for pieces

- populateGrid get URL of image grid's position

Just Space's MicroService is known by Angular. It is a BFF (Back end for front end) pattern, API Gateway, or in the old ages of GOF94 a so called Facade.
All images are provided by Servlets, and all MicroServices has an URL (/image/{id}). Each Microservice is responsible to provide its own image. Space (the coordinator) is responsible to provide image for background.
There is a timer on browser that schedule this execution over and over, 400ms spaced.

```
$interval( function(){ $scope.callAtInterval(); }, 400);
```

callRestRun makes the magic of game happens. The routine is on space.war->Space.java.

```
    @GET
        @Path("/run")
    public void run() throws Exception {

        // (1) redirect commands to other Microservices
        for (int i = 0; i < urls.length; i++)
                callRest(urls[i] + "run");

        // create bombs from enemies
                String json = callRest(getURLFromName("enemy") + "position");
                JsonArray array = Json.createReader(new StringReader(json)).readArray();
                int biggerX = 0;
                int lowerX = 0;
                int biggerY = 0;
                for (int j = 0; j < array.size(); j++) {
                        int objectX = ((JsonObject)array.get(j)).getInt("x");
                        int objectY = ((JsonObject)array.get(j)).getInt("y");
                        if (objectX > biggerX) biggerX = objectX;
                        if (objectX < lowerX) lowerX = objectX;
                        if (objectY > biggerY) biggerY = objectY;
                }
                // 30% of chance to generate a bomb from enemy
                Random r = new Random();
                if (r.nextInt(100) < 30) {
                    int posx = lowerX + r.nextInt(biggerX - lowerX);
                    int posy = biggerY+1;
                    callRest(getURLFromName("bomb") + "create/" + posx + "/" + posy + "/false");
                }
    }
```

The main part is to repeat a process to orchestrate calls to other Microservices (not yellow piece). All other MicroServices has the same verb "run". Some of then make something useful (Bomb, Enemy, Collision), but the choice was to create a common set of verbs (position, run, destroy, move and finished). In the old ages of object orientation, it was known as Polymorphism.
There are some reasons for this approach (a common interface):

- Since most of Microservices looks like a family of objects (all are Sprites), make sense to have a common set of verbs.

- Make It easy to change or understand one Microservice, since you have visited others before
- Change in functionalities can be implemented with minimal changes on orchestrator
- New Microservices can be implemented using a common interface (no need to understand big picture)

The yellow piece in the Run method takes care of create bombs from enemies. It finds the elements more closed to bottom and select a random strategy to create a bomb. There is 30% of chance to create a bomb from enemy at each execution.

Let's see specific implementation of run method. The enemy it is:

```
@GET
@Path("/run")
public void run() throws Exception {
   if (getValues() != null)
   {
        if (getValues().isFinished()) return;
        int x0 = getValues().getTies()[0].getX();
        int y0 = getValues().getTies()[0].getY();

        if (x0 <= 9) incrementAllX();
        else {
               returnAllXtoBegin();
               if (y0 <= 15)  incrementAllY();
               else { // reach the end of game
                      destroyAllEnemies();
               }
        }

    }
  }
```

It just moves all elements to right side, until the end of line is reach (x0 < 9 – we have ten enemies, and when the last one reach the end x=19, the first should be at position 9). When this situation is found, move one line down, and move all elements to initial position x). It's a line feed and a carriage return, or a typewriter machine (sorry by younger that don't know what those things mean ☺).

BTW: We have 10 enemies by row, with 20 positions by line. So when all elements are right most, the first element on line will fit position 9.

The player move is:

```
@GET
@Path("/run")
   public void run() throws Exception {
}
```

This means “no action to be made on user”. Today, all movements of player are handled by keyboard. The reason to leave a common set of verbs is to open opportunity in future to make it more competitive (transport player to another position on screen). We just need to implement this verb.
Finally, the Bomb run is:

```
@GET
@Path("/run")
public void run() throws Exception {
	if (getValues() != null) {
		if (getValues().isHasFinished()) return;
		for (int i = 0; i < getValues().getBombs().size(); i++) {
			OneBomb b = getValues().getBombs().get(i);
			if (b.isDestroyed()) continue; // nothing to do with a finished bomb
			if (b.isFromPlayer()) { // bomb from luke
				if (b.getY() <= 0) {
					b.setDestroyed(true);
					continue;
				}
				b.setY(b.getY() - 1);
			}
			else { // bomb from dart
				if (b.getY() >= 20) {
					b.setDestroyed(true);
					continue;
				}
				b.setY(b.getY() + 1);
			}
		}
	}
}
```

getValues() grabs a collection of objects placed in a Servlet context. Today the game is single instance and single user. In fact, today the game is not prepared to escalates on a Cloud. More discussions on this topic should be made.
The Bomb’s run just test if it is from player and make it on position down (`b.setY(b.getY() - 1)`) or one position up instead.

All executions are independent and no object knows each other. In really, if we ignore or even stop the Collision Microservice, all pieces will run forever, with no collisions happen in the game at all. So, the run of collision make the magic of integration happen:

```
@GET
@Path("/run")
public void run() throws Exception {
	JsonArrayBuilder allelements = Json.createArrayBuilder();
	// join ll elements toguether
	for (int i = 0; i < urls.length; i++) {
		String json = callRest(urls[i] + "position");
		JsonArray array = Json.createReader(new StringReader(json)).readArray();
		for (int j = 0; j < array.size(); j++) {
			allelements.add(array.get(j));
		}
	}
```

```
            // if two elements match same position
            JsonArray array = allelements.build();
            boolean sprites[] = new boolean[array.size()];
            for (int i = 0; i < sprites.length; i++) {
                        sprites[i] = ((JsonObject)array.get(i)).getBoolean("destroyed");
               }
            for (int i = 0; i < array.size(); i++) {
                        for (int j = i+1; j < array.size(); j++) {
                               JsonObject obj1 = (JsonObject)array.get(i);
                               JsonObject obj2 = (JsonObject)array.get(j);
                               if (obj1.getInt("x") == obj2.getInt("x") &&
                                    obj1.getInt("y") == obj2.getInt("y") &&
                                     obj1.getInt("id") != obj2.getInt("id") &&
                                     !obj1.getBoolean("destroyed") &&
                                    !obj2.getBoolean("destroyed") &&
                                     !sprites[i] &&
                                     !sprites[j] )
                               {
                                      sprites[i] = sprites[j] = true;
                                      String resturl1 = findRest(obj1.getString("type"));
                                      if (resturl1 != null) {
                                             callRest(resturl1 + "destroy/" + obj1.getInt("id"));
                                      }
                                      String resturl2 = findRest(obj2.getString("type"));
                                      if (resturl2 != null) {
                                             callRest(resturl2 + "destroy/" + obj2.getInt("id"));
                                      }
                               }
                        }
        }
}
```

It just calls position on all other Microservices (including itself, but response is an empty array) and place all of them into a single collection. After that, test if position x,y of each other are the same, and destroy both if so. All elements remain in collection, but has the flag “destroyed”. This open the possibility to make some improvements in the future (count bombs from player and enemies or make a destroyed enemy comes to live again!).

Destruction is made using an {id}. All objects have a unique identifier that can be used to destroy a specific element. In really, as we told before, destruction is just about set attribute “destroyed” to true.

Just collision and space know other Microservices. There is an array of elements used as parameter:

```
        public static String[] urls = {
        "http://" + System.getProperty("enemyip", "127.0.0.1:9081") + "/enemy-1.0/",
        "http://" + System.getProperty("playerip" , "127.0.0.1:9081") + "/player-1.0/",
        "http://" + System.getProperty("bombip" , "127.0.0.1:9081") + "/bomb-1.0/",
        "http://" + System.getProperty("collisionip" , "127.0.0.1:9081") + "/collision-1.0/"};
```

All ports and IPs should be configured using a property file (jvm.options on liberty)

```
-Dbombip="127.0.0.1:9081"
```

```
-Dcollisionip="127.0.0.1:9081"
-Denemyip="127.0.0.1:9081"
-Dplayerip="127.0.0.1:9081"
-Dspaceip="127.0.0.1:9081"
```

It is important to provide some kind of flexibility on Microservices implementation, and there is a principle from 12Factor to rule that (III- Store config in the environment). Some other pattern increases this flexibility, like registry. We can make all other Microservices to register itself on a common backend service, and them space can searches for Microservices to interact with. Today there is no implementation for that, but this static array should be replaced with a more flexible implementation.

Finally, lets game!

There are some keys mapped to run:

0 (number 0) – reset game to beginning
o (letter o) – goes to left side
p (letter p) – goes to right side
space – launch bomb from player

To run, just download WARS from github, and open it into eclipse. All wars have the source code inside it. Map all wars into server (liberty profile), and put the jvm.options with ip and port. Run the server, and make sure the port is 9081 and the WAR is being expanded (just put `<applicationManager autoExpand="true"/>` on server.xml). Open a browser and call http://127.0.0.1:9081/space-1.0 and the game must go on!

Another choice to run is to download Microservices.zip, open on a directory and run steps:
mvn install (on directory)
cd target/wlp/bin
./server start microinvader
open browser and type:
http://127.0.0.1:9081/space-1.0/game.html

Maven will take care of download liberty profile, install WARs inside it. The start and stop are handled by simple command lines, but the maven file is comprehensive and very small. There is a hierarchy with a parent and POMs for each one of the WAR files.