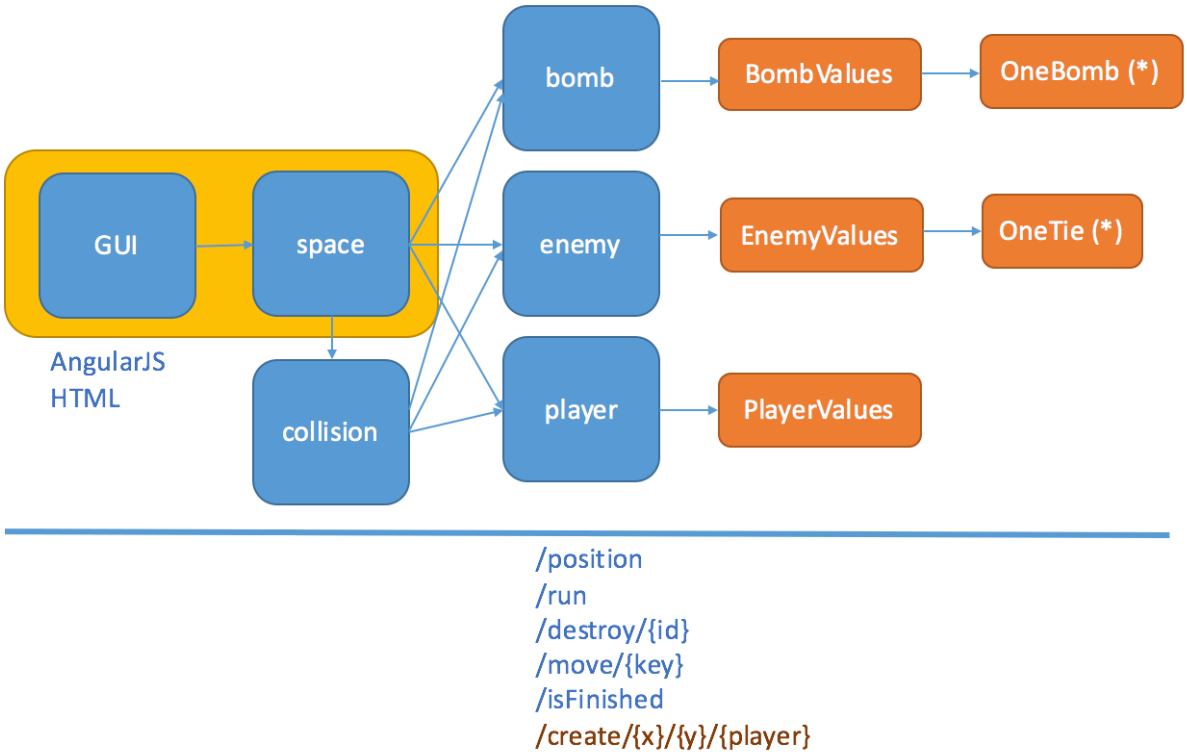
Microinvader

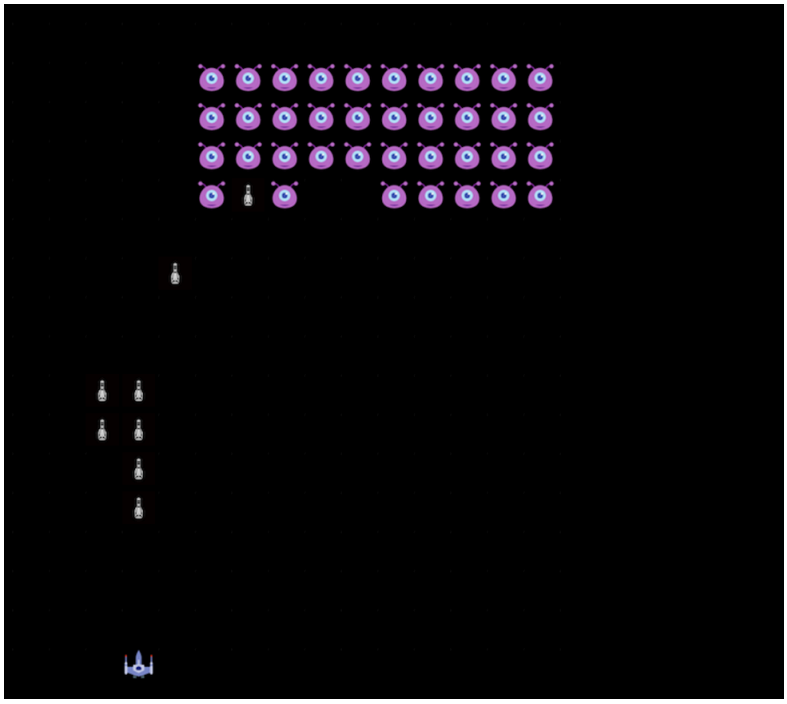
By Glauco Reis (gsreis@br.ibm.com.br) (glauco@portalbpm.com.br)

Microinvader é uma implementação simples e com motivos didáticos do jogo space invaders (muito simples para manter a clareza na explicação). Entretanto, o jogo é totalmente funcional e visual, e como as peças se movem no espaço, e alguns tópicos como circuit breaker podem ser visualmente mostrados.

O diagrama do jogo (versão 1.0) é:



e a captura de tela da execução é:



Bombas, Jogador e Inimigos são microserviços independentes. Eles são controlados por Space, que é um Service Orchestrator (Gateway ou Front Controller) para todo o jogo. Toda comunicação é gerenciada pelo serviço Space. Foi feito um esforço para fazer com que Bombas, Inimigos e o Jogador fossem independentes entre si, objetivando a troca de peças.
Uma vez que nenhum dos três se conhecem (Bombas, Jogador, Inimigos) foi necessária a criação de um MicroServiço chamado Colisão (Collision) para identificar inconsistências no jogo (como a colisão de duas peças que gera uma destruição de ambas).
O jogo é um grid de 20x20 posições (hoje é fixo no jogo). A interface de usuário foi criada com HTML puro e AngularJS, utilizando técnicas de rich client (a chamada aos serviços é feita pelo browser). Sem JSPs, Servlets, JSF, Struts ou outras interfaces de usuário criadas no lado servidor, como têm sido a tendência atual de mercado.
Hoje um dos princípios de MicroServiços (deve ser stateless) não está sendo respeitado. Uma vez que o programa será utilizado para explicar estes conceitos, a opção foi por manter inicialmente simples e evoluir o código ao longo do tempo. Ele será aprimorado para apresentar opções de como se tornar stateless.
A interface do usuário está concentrada em um arquivo simples com um grid de 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>
```

Este código pode ser encontrado em space.war->game.html.
Existe um controller Angular.JS com um grid de elementos.

```
        $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]
                      ];
```

A “dinâmica” do jogo é gerenciada por um timer criado no controlador.
O processo consiste em chamar o “Front controller” que dispara uma sequencia de chamadas que irá mover as peças e repopular as informações do grid.
Como o Angular recebe os eventos e os trata em seu loop interno, os elementos são atualizados sem flicker (a tela não fica piscando a cada atualização).
Observando no vídeo no site, o jogo parece natural, embora não tenha sido uma preocupação principal, e não apresenta lentidões, parecendo ser coeso, embora as peças individuais sejam independentes entre si.
A rotina principal, gerenciada pelo Angular é:

```
$scope.callAtInterval = function() {
        // some code removed
        if (!$scope.finished) {
            $scope.callRestRun();      // atualiza posição das peças
            $scope.callRestPosition(); // pega nova posição após atualizar
            $scope.populateGrid();     // redezenha tela
        }
        $scope.callRestFinished(); // verifica se jogo acabou (atualiza flag finished)
  }
```

Enquanto o jogo não está finalizado:

- callRestFinished testa se jogador ou inimigos foram destruídos e atualiza flag finished
- callRestRun move todas as peças uma posição adiante
- callRestPosition obtém novas posições das peças
- populateGrid pega as URLs para as novas posições do grid

Somente o serviço Space é conhecido pelo Angular. Em termos de MicroServiços, ele é um pattern hoje chamado de BFF (Back End for Front End), API Gateway, ou nos tempos de Erich Gamma (GOF94) simplesmente um Façade (Franceses e Brasileiros podem escrever corretamente, em Inglês não há cedilha e, portanto, foi renomeado para Facade).
Todas as imagens são fornecidas por Servlets (não é apropriado fornecer imagens por serviços REST) e todos os Microserviços respondem com uma URL (/image/{id}). Cada MicroServiço é responsável por fornecer a imagem do "Objeto" que controla na tela. O serviço Space acabou ficando responsável por fornecer a imagem do fundo (espaço mesmo).
Um timer, espaçado de 400ms, executa os passos repetidamente até o final do jogo.

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

callRestRun executa o movimento principal das peças. A rotina está em 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");
                }
    }
```

A execução principal é o código não destacado (a parte que não está em amarelo). Ela repete a chamada "run" para cada um dos outros Microserviços cadastrados (3 linhas de código).

Utilizamos um princípio da Orientação para Objetos (Polimorfismo). Na verdade, existe um conjunto comum de "verbos" REST (position, run, destroy, move, finished) para os Microserviços principais (Bomb, Enemy, Collision, Player). Entre as razões para escolha estão:

- Os Microserviços se parecem com uma família de objetos (Figuras ou Sprites), e faz sentido terem um mesmo conjunto de verbos.
- Torna mais fácil a compreensão de um Microserviço, uma vez que você tenha visitado outros anteriormente.
- Troca de funcionalidades podem ser implementadas com mínimo impacto no orquestrador.
- Novos Microserviços podem ser implementados usando uma interface comum (não é necessário entender a complexidade do sistema).

O pedaço destacado em amarelo no código anterior cuida da criação de bombas a partir dos inimigos. Ele encontra os elementos mais próximos do player, e seleciona uma estratégia de randomicamente gerar ou não uma bomba, com probabilidade de 30% de criação para cada iteração (pode ser modificado).

A implementação do método run no inimigo, por exemplo, é:

```
@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();
               }
        }

    }
  }
```

Ele move todos os elementos para o lado direito, até encontrar o final da linha.
Comentário: Temos 10 inimigos por fileira, e um total de 20 posições na tela. Quando os elementos encostarem na lateral direita, o elemento mais à esquerda estará na posição 9 (a última posição é 19).
Quando esta situação acontece, nós movemos uma linha para baixo, e voltamos todos os elementos para o canto esquerdo da tela. Funciona como um carriage return e um line feed, ou como uma máquina de escrever (desculpem os jovens que não sabem o que estas coisas significam ☺).

O método run do Player é:

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

Exato, ele não faz nada. Para mantermos a uniformidade de chamar os mesmos métodos, optamos por implementar todos e deixar que o Microserviço decida se ele é aplicável ou não.
Isto acaba por gerar uma chamada que não processa nada para um elemento em particular, ou seja, um maior consumo de rede.
Por outro lado, podemos no futuro tornar o jogo mais competitivo, como transportar o jogador para outra parte da tela. Para isto basta implementar o método run.

O método run de Bomb é:

```
@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() obtém um conjunto de objetos que correspondem a cada uma das bombas criadas durante o jogo. Estes objetos ficam armazenados atualmente no ServletContext, o que torna o jogo single instance e single player. Esta também não é uma boa estratégia e estaremos explorando opções para tornar mais “cloud ready” no futuro.
A execução testa se a bomba ainda está “no jogo” (bombas destruídas ficam na mesma coleção, e não são removidas) e move uma posição para baixo se foram criadas por um inimigo ou uma posição acima se foram criadas pelo player.

Como dissemos, todas execuções são independentes e nenhum Microserviço conhece os demais. Na realidade, se o Microserviço Collision for removido ou parado, o jogo fará com que as peças se movimentem de forma independente, sem que nenhum objeto seja alterado pela trajetória do outro.
A execução do Collision faz com que o universo do jogo seja mais realista:

```
@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"));
                                       }
                                    }
                        }
        }
}
```

Ele chama o método position em todos os Microserviços e obtém objetos JSON que contém várias informações comuns, como a posição de cada um, entre outras informações. Depois ele navega por todos eles, e se estiverem na mesma posição (x,y) e não estiverem destruídos, marca os dois como destruídos e continua o processo. Porque não remover elementos da coleção? Bem,

abrimos a possibilidade para uma série de melhorias, como efetuar uma contagem de bombas utilizadas, ou mesmo fazer com que um inimigo destruído volte ao jogo.
Todos os objetos têm um id e um tipo, portanto é fácil chamar um serviço que destrói um elemento em particular.

Apenas os serviços Collision e Space conhecem os outros Microserviços. Atualmente existe um array de endereços fixo no código, mas também existem práticas que tornarão os serviços menos acoplados. Novamente a opção nesta primeira versão foi por algo mais simples.

```
        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/"};
```

As informações de IPs, embora fixas, podem ser passadas por parâmetros na JVM, tornando um pouco menos acoplado.

```
-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"
```

Este tipo de “parametrização” flexibiliza e torna mais fácil movimento entre servidores com IPs dinâmicos, como acontece em ambientes Cloud. Também é um dos princípios do 12Factor, (**III – Store config in the environment**). O uso de Registries também é algo indicado, e será explorado no futuro.

Finalmente, vamos ao Jogo!

Algumas teclas foram mapeadas no jogo:

0 (number 0) – reseta o jogo
o (letter o) – vai para o lado esquerdo
p (letter p) – vai para o lado direito
space – lança uma bomba do jogador

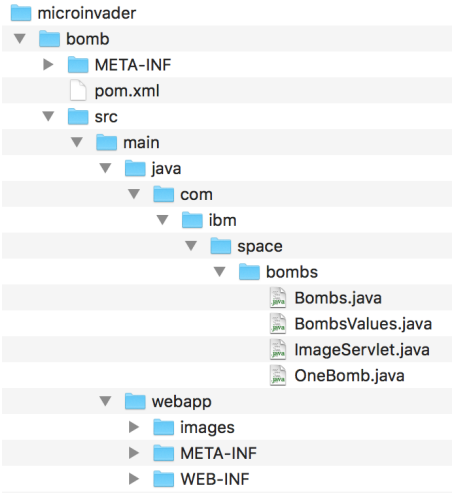
Para executar o jogo, basta que uma JVM esteja instalada na máquina (pode ser 7.0 ou 8.0) e o maven também. Para saber se está instalado, basta abrir uma janela no sistema operacional e digitar:
java –version
mvn –v

Após isto, fazer o download do ZIP do github, e executar os passos:

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

O Maven irá baixar o liberty profile, instalar os WARs e deixar o aplicativo pronto para execução. Os arquivos Maven foram bastante simplificados para tornar mais legível o entendimento. O uso de Maven ou qualquer outra ferramenta deste tipo é altamente recomendado, pois permite que a integração contínua possa ser implementada, e um dos fatores principais de sucesso do uso de Microserviços é a preexistência de um processo automatizado para publicação.

Se desejar estudar mais antes de fazer suas alterações, pode baixar os códigos e importa-los no eclipse ou seu editor preferido. Ele respeita as regras dos projetos Maven, e os fontes estão em uma estrutura como:



Os códigos fonte ficam em src/main/java e a camada web (HTMLs, imagens, Angular, etc) em src/webapp.