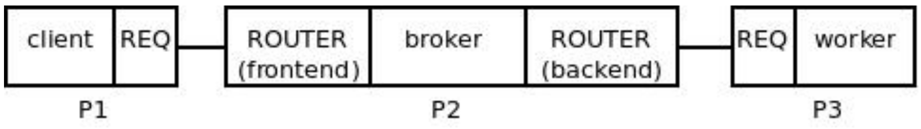
EXAM LAB2 & LAB3 TSR

This exam consists of 20 multiple-choice questions. In every case only one of the answers is the correct one. Answers should be given in a SEPARATE answer sheet you should have been provided with.

All questions have the same value. If correctly answered, they contribute 0,5 points to the final grade. If incorrectly answered, the contribution is negative, equivalent to 1/5th the correct value, which is -0,1 points. So, think carefully your answers. In case of doubt, leave them blank.

The exam can be completed within 1 hour.

Assume an application comprising 3 components: "client.js", "broker.js" and "worker.js". The components communicate via ZMQ sockets, using an architecture like the one described in Lab2:



This application is deployed launching one instance of each "`client.js`" (process P1), "`broker.js`" (process P2) and "`worker.js`" (process P3) in three different machines. Assume the client process' identity is string "`ADFG`", and the identity of the worker process string "`ERTH`". Further, assume that the broker saves no state for ongoing requests to workers. Answer the following questions concerning the application's behavior:

General operation of this setup as seen in Lab2: The general way in which this set up works is described in pages 16-18 of the lab bulletin. Let us present it again focusing in our case.

1. When a client sends a message through its REQ socket, for instance `["NEW REQUEST"]`, that socket prepends an envelope consisting of an empty frame, transmitting the message `["","NEW REQUEST"]` through the wire to the broker's frontend socket.
2. The broker's frontend being a router socket, prepends the ID of the sender socket to the message it delivers to the broker, resulting in the broker receiving the message `["ADFG","","NEW REQUEST"]`.
3. The broker has to send the client's request to one of the workers. The broker selects one from a list containing those that are available. Given that the broker does not keep any state about on-going requests, and given that the broker needs to route back a worker's answer to the client that made that request, it follows that the client's identity must be shipped together with its request to the selected worker, and that worker must include the client's ID in its own answer. Thus, the broker can

reuse the message it got from its frontend as the payload to send to the selected worker. In addition, to comply with the worker's REQ socket protocol, the broker must prepend an envelope consisting of an empty frame to this message, to simulate the operation of a REP socket. Finally, to actually send a message through the backend socket, the broker must prepend the ID of the selected worker to the message. In this case there is only one worker, with ID ERTH. In sum, the broker sends message `[“ERTH”,””,”ADFG”,””,”NEW REQUEST”]` through its backend router socket.

4. In order to route the message to the proper worker, the backend socket consumes the first frame treating it as the ID of the destination socket, sending through the wire the rest of the message, `[””,”ADFG”,””,”NEW REQUEST”]`, to the worker's REQ socket.
5. The worker's REQ socket receives this message and, applying its protocol, strips the envelope consisting of the empty frame in front of the received message. The worker receives `[”ADFG”,””,”NEW REQUEST”]` from its REQ socket.
6. After processing the request, the worker needs to send back its answer. To ensure the broker has enough data to return the answer to the client, it keeps the client's data in the message, sending back `[”ADFG”,””,”REQUEST PROCESSED”]` through its REQ socket.
7. The worker's REQ socket, applying its protocol, prepends an envelope, consisting of an empty frame, to the message it actually sends through the wire towards the broker's backend socket: `[””,”ADFG”,””,”REQUEST PROCESSED”]`.
8. The backend socket, receiving this message prepends it with the ID of the worker, delivering `[“ERTH”,””,”ADFG”,””,”REQUEST PROCESSED”]` to the broker.
9. The broker uses the first frame as the ID of the worker, and adds it to the list of available workers. The broker has no use for the empty frame, so it consumes it, obtaining message `[”ADFG”,””,”REQUEST PROCESSED”]`. This message already has the ID of the client, and the empty frame expected by the client's REQ socket, thus the broker sends it through its frontend router socket.
10. The frontend socket consumes the first frame to determine to which client to route the message, sending `[””,”REQUEST PROCESSED”]` through the wire to the client's REQ socket.
11. The client's REQ socket, upon receipt of this message, strips its envelope (the first, empty frame), and delivers `[”REQUEST PROCESSED”]` to the client's code.

1. **Assume P1 sends string "`NEW REQUEST`" via its REQ socket, and will receive answer "`REQUEST PROCESSED`" through that same socket. Select the true assertion**

A	P3's program gets the following message from its REQ socket: `["ERTH","","ADFG","","NEW REQUEST"]` False. The first two frames are not received
B	When P3 answers P1's request, P2 gets from its backend socket this message: `["ERTH","","ADFG","","REQUEST PROCESSED"]` True. See General explanation above
C	P2 sends P1 the answer with message: `["ERTH","","ADFG","","NEW REQUEST"]` False. The first two frames should not be there. The last frame is wrong.
D	When P3 answers P1's request, P2 gets from its backend socket this message: `["ERTH","","REQUEST PROCESSED"]` False. The ID of the worker is missing, as well as the empty frame inserted by the worker's REQ socket.
E	All the above.
F	None of the above.

2. Assume now we deploy the application with an extra worker instance (process P4). Assume further that P1 sends 10 consecutive requests through its REQ socket. Select the right assertion

A	Having two instances of `worker.js` allows us to cut in half the average processing time of the 10 requests issued by P1 False. Clients may only send requests sequentially through a REQ socket (that is the way REQ operates). This means that only one request may be in transit or being processed at a time. So, at any point in time at most one worker is processing a client's request, which is no different from the situation where we have just one worker.
B	The frontend ROUTER socket of P2 allows concurrent sending of the 10 requests from P1, distributing them to the two worker processes (P3, P4), thus reducing the processing time of P1's requests False. The frontend ROUTER socket of P2 never sees more than a single request at a time. See above.
C	Socket REQ of P1 allows sending the various requests concurrently. False. This has been already explained in the first part of this question.
[✓ D]	Having those two instances of `worker.js` will not help reducing the average processing time of the 10 requests issued by P1 from the case when we only have one instance of `worker.js` True. This has been already explained in the first part of this question.
E	All the above.
F	None of the above.

3. Assume we have now a different version of the broker, `broker2.js`, with a DEALER socket for the backend. Assume furthermore, that we have another version for the worker, `worker2.js`, using a REP socket to communicate with the backend from the broker.
Let us launch the application with one process for "`client.js`" (process P1), one for "`broker2.js`" (process P2) and another one for `worker2.js` (process P3). If P1 sends string "`NEW REQUEST`" via its REQ socket, select the right answer

A	When the request reaches P3 through its REP socket from P2, the message P3 gets is `["NEW REQUEST"]` True. In this configuration we are using a broker based on a ROUTER-DEALER combination of sockets. This is an option for implementing brokers. In that case, brokers use a round-robin strategy (embedded in the DEALER management) in order to forward client requests and it shouldn't take care of identities. This means that such broker doesn't need to prepend nor drop any segment to/from the messages it manages. Let us describe what happens in this scenario: 1. P1 sends a message containing ["NEW REQUEST"] using its REQ socket. 2. The REQ socket puts that message in the network, prepending an empty delimiter. This is its default behaviour. As a result, the message being transmitted is ["", "NEW REQUEST"]. It is eventually received by the frontend socket of the broker. 3. The frontend socket is a ROUTER socket. When a ROUTER socket receives a message, it prepends a segment with the identity of the sender. In this case, this means that the message being delivered to P2 is ["ADFG","","NEW REQUEST"]. P2 doesn't alter the message and sends it via its backend DEALER socket. 4. DEALER socket doesn't apply any message transformation. So, ["ADFG","","NEW REQUEST"] is the message being transmitted to P3. 5. P3 uses a REP socket in order to receive its messages. REP sockets remove and maintain message "envelopes" in order to recover and prepend them to the forthcoming reply. This means that such socket has received ["ADFG","","NEW REQUEST"] but it only delivers ["NEW REQUEST"] to P3. 6. P3 processes that request and sends ["REQUEST PROCESSED"] as its reply. 7. Such reply is sent using its REP socket. Such socket scans its envelope buffer and sees that ["ADFG",""] is there. So, it prepends such envelope to the reply. This means that the message put in the

	network is [“ADFG”,””,”REQUEST PROCESSED”]. 8. This message eventually reaches the backend DEALER socket of the broker. Since dealers don’t transform messages, P2 receives those three segments. 9. This broker is very simple and only needs to forward all received segments to its other socket. In this case, the other socket is the frontend. Such socket is of type ROUTER. ROUTER sockets send messages interpreting the first segment (that is also dropped from the message) as the identity of the connection/client to be used. That first segment contains the string “ADFG” in this example. That is the identity of P1. So, the message [“”,”REQUEST PROCESSED”] is what is finally sent to P1. 10. P1 receives the [“”,”REQUEST PROCESSED”] message in its REQ socket. REQ sockets drop the first empty delimiter for delivering its received messages. This means that REQ finally delivers [“REQUEST PROCESSED”] as the reply for its original request.
B	P3’s REP socket does not need an identity to communicate with P2 True. This has been explained in the steps 7 and 8 of the sequence explained in the previous option. In any case, please read carefully all the sequence. P3 doesn’t need any identity since it directly interacts with a DEALER socket and that type of socket doesn’t manage connection identities.
C	P1’s REQ socket still needs an identity to communicate with P2 True. The REQ socket of P1 is used for interacting with the ROUTER frontend socket of P2. ROUTER sockets manage identities. As a result, P1 should set its identity before connecting its REQ socket to the ROUTER socket of P2.
D	P2’s DEALER socket (backend) needs to send the following message: `[”ADFG”,””,”NEW REQUEST”]` True. Please review step 4 in the sequence described in option A of this question.
[✓ E]	All the above.
F	None of the above.

Consider the following code fragment:

```
var zmq = require('zmq')
  , A = zmq.socket(X)
  , B = zmq.socket(Y);

A.bindSync('tcp://*:1111');
B.bindSync('tcp://*:2222');

```

```
A.on('message', function() {
  var args = Array.apply(null, arguments);
  B.send(args);
});

B.on('message', function() {
  var args = Array.apply(null, arguments);
  A.send(args);
});
```

4. The above code cannot be that of a broker if …

A	Clients connect with the broker via a REQ socket False. This is essentially how the scenario in question 3 works when X is router and Y is dealer. See explanation given there.
B	`X` is `'router'` and clients connect to port `1111` of the broker's node. False. Note that socket A has been bound to port 1111. Thus the scenario in question 3 applies.
[✓ C]	`X` is `'dealer'` and workers connect to port `2222` of the broker's node. True. Note that socket B has been bound to port 2222. X should be 'router' in order to implement a simple broker because the broker MUST know to whom to send the answers back. The dealer socket should be B; i.e., the socket where the workers have been connected to. This means that Y should be in that case the "dealer" string.
D	Workers connect with a REP socket False. The scenario of question 3 applies, as in option A.
E	All the above.
F	None of the above.

In the part **`zmqavanzado`** of lab 2, there are messages composed of 3, 4 and 5 frames:

5. Messages received/sent by P2 through the `backend` socket are not 5 frames long when…

A	The worker reports its availability. True. The architecture of "zmqavanzado" is the same as that presented for question 1, and described in the general operation explained there. In that description, the messages delivered by the backend to the broker are 5 frames long. This is so because they carry two envelopes: the first carries the identity of the worker, the second that of the client, and finally, a frame with the answer's payload. When a worker registers with the broker, the message delivered by the backend to its broker carries just the envelope with the ID of the worker, and the worker's actual message to the broker: 3 frames, not 5
B	The worker reports an error. False. Errors should be forwarded to clients. They need 5 frames.
C	The worker answers a request. False. Answers to requests should be forwarded to clients. They need 5 frames.
D	The message hasn't been sent by the worker. False. The messages that are received by the backend socket of the broker have been sent in all cases by workers.
E	All the above.
F	None of the above.

6. Messages received/sent by P2 through its `frontend` socket <u>are not</u> 3 frames long when...

A	The client makes a request False. As we mentioned in solution to question 5, the architecture is that described in question 1. The descriptions shows that ALL messages coming in through the frontend, or going out through it are 3 frames long.
B	The broker returns a result False. Same explanation as for option A
C	The broker returns an error False. To return an error is another example of action that needs to send a message from the broker to the client. So, it follows the sequence explained in the previous option. Those messages have 3 frames when P2 calls the send() method of its ROUTER frontend socket.
D	The client connects for the first time False. There is no message going through the frontend. So, it does not apply
E	All the above.
F	None of the above.

The **zmqexperto** section of Lab2 suggests using promises...

7. Promises are transmitted from the broker to the client...

A	Directly, as an additional message frame
B	Encoded with JSON, as an additional frame
C	Modifying one of the existing segments
D	In no way, as promises are global variables, and do not need to be transmitted
E	All the above.
F	None of the above. Promises are <u>objects</u> implementing specific mechanisms to handle asynchronous callbacks. They are merely a runtime mechanism, and in no case are they transmitted in any way among processes. From the previous explanation, it is clear they ARE NOT variables, much less GLOBAL variables.

8. In 0MQ …

A	Messages are strings of characters False. See option D.
B	Messages are arrays of values False. See option D.
C	Messages are dictionaries composed of key/value pairs False. See option D.
D	Messages are formed of various segments, delivered atomically True. Messages may consist of several segments or frames. To this end, all those segments can be passed as a single argument to the send() method using an array, placing each segment in a different slot. Another alternative consists in using the second argument of send() in order to pass the zmq.ZMQ_SNDMORE option. In that case, we may use multiple calls to send(), passing a different segment in each of them. In both cases, all segments are finally put in the same logical message. That messsage is delivered atomically (i.e., either all the segments arrive or none does arrive) to its intended receiver.
E	All the above.
F	None of the above.

9. In Lab2, …

A	The set of workers is fixed, and its size is given to the broker as a parameter False. Its size doesn't need to be given to the broker as a parameter. Workers connect to the backend socket of the broker and send to it an availability message to start being considered by that broker. As a result of this, the broker doesn't need any argument or explicit communication in order to know how many workers should be managed.
B	New workers can be added, sending their URL to the client processes False. Clients only need to know the URL of the broker. They never need to use the addresses of the workers, since a client always uses the broker in order to reach a given worker.
C	New workers can be added, sending their URL to the broker False. A worker doesn't need to propagate its address to anybody else. It simply needs to connect to the backend socket of the broker.
D	The set of workers is fixed, and its size is passed to the clients as a parameter. False. As it has been already explained, clients do not need to know anything about workers. This means that the size of the worker set isn't needed by the clients.
E	All the above. False.
F	None of the above. True.

10. In zmqexperto, the dynamic broker configuration is carried out through…

A	… the frontend socket of the broker False.
B	… the backend socket of the broker False.
C	… an additional REQ socket of the broker False.
D	… an additional PUB socket in the broker False.
E	All the above. False.
F	None of the above. True. As explained in the lab's document, we need an additional socket to avoid major changes in the broker operation wrt its frontend and backend (thus cannot use options A and B). In addition, that socket must support interactions started by any agent wanting to reconfigure the broker. REQ sockets must initiate interactions, thus cannot use option C either. Finally, PUB sockets must also initiate interactions (actually only one-way), thus option D is not usable either.

11. In Lab3, the command “`docker run -i -t tsir/baselab3`”

A	Creates a container running the shell on the `tsir/baselab3` image False. See below.
B	Starts a virtual machine running, based on the `tsir/baselab3` image False. See below.
C	Starts a sh process within a preexisting virtual machine running the `tsir/baselab3` image False. See below.
D	Starts the sh program within the `tsir/baselab3` image running in the current environment. False. See below.
E	All the above. False.
F	None of the above. True. Creates and runs a container based on the `tsir/baselab3` image, which has been built using he `tsrImage/Dockerfile`, as explained in the lab document. In that `Dockerfile` the entry point is “`npm start`”, not a shell. Note that containers have nothing to do with virtual machines. Finally, the action will fail, as baselab3 has an empty `/app` folder, which means that “`npm start`” will fail, stopping the container.

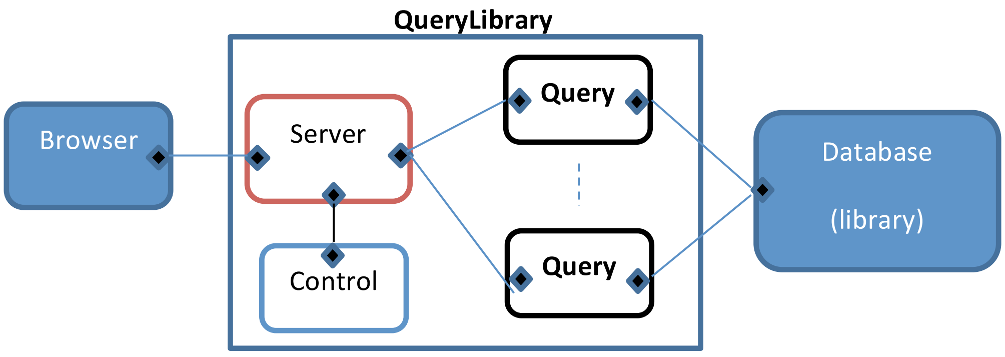
12. In Lab3 ...

A	A component is defined by a directory containing a particular `Dockerfile` True. Bulletin of Lab3 commented this in page 9. Components in lab3 are built with all their code in a directory containing a particular `Dockerfile`
B	The default file executed when a component's image is run cannot be changed by the developer of the component False. As explained in the Lab's document, this can be changed by modifying the `package.json` file which must exist in a component's directory.
C	The default file executed when a component's image is run is `index.js`, within the root of the component's directory. False. As explained in lab3's document, unless `package.json` is changed, the default action of the entrypoint "`npm start`" is to run "`node server.js`"
D	The component does not need to specify a `package.json` file False. See comments above, and lab document.
E	All the above. False.
F	None of the above. False.

13. Assume we have deployed a service using docker containers with identifiers "client", "server", "server_1", "server_2" We can stop the whole service with the command...

A	docker rm –f client server server_1 server_2 True. A similar case was illustrated at page 14 from bulletin of Lab3
B	docker rm –f client server False. It does not affect server_1 and server_2, which may still be running
C	docker rmi –f client server False. "`docker rmi`" deletes images, not containers, and "client" "server" are container names, not image names.
D	docker rmc client server server_1 server_2 False. "`docker rmc`" does not exist.
E	All the above. False.
F	None of the above. False.

Consider a web service with name QueryLibrary, which obtains information about a Library's database. The service consists of 3 components: ***Server***, managing the interaction with the browsers. ***Query***, managing accesses to the data base. And ***Control***, which, through a control port can dynamically modify the web's interface look and feel.



We want to deploy this application using the technologies of dependency resolution used in Lab3.

We have three directories: Components/Server, Components/Query and Components/Control, containing each the definition of the components. Assume the following file contents:

"Components/Server/config/default.js"

```
module.exports = {
  provides : {
     controlPort : 8001,
     queryPort : 8002
     },
  external : {
   webPort : 8000 }
}
```

"Components/Query/config/default.js"

```
module.exports = {
  requires : {
     serverUrl: tcp://localhost:8002
     },
  parameter : {
   BDServer : 192.168.1.1:8003 }
}
```

"Components/Control/config/default.js"

```
module.exports = {
  requires : {
     serverUrl:
tcp://localhost:8001
     }
}
```

"Components/Control/server.js"

```
var utils = require('../utils.js')
......
```

"Components/Query/server.js"

```
var utils = require('../utils.js')
......
```

"Components/utils.js"

```
…
…
…
```

14. Select the right value for the "`links`" attribute within the service descriptor for QueryLibrary …

A	`links: {`
	`      Query : {  serverUrl : [ "Server", "queryPort"] },`
	`      Control : { serverUrl : [ "Server", "controlPort"] }`
	`}`
	True. Both Query and Control components declare a "`requires`" "`serverUrl`" url, which, according to what is explained in Lab3's bulletin (page 12) need to be linked to a "`provides`" element of some component. Server exposes two "`provides`" elements, "`queryPort`" and "`controlPort`". So the above linking specification fits the rules exposed.

B	`links: {`
	`      Query : {  QueryUrl : [ "Server", "queryPort"] },`
	`      Control : { ControlUrl : [ "Server", "controlPort"]`
	`}`
	`}`
	False. Attributes `QueryUrl` and `ControlUrl` are not declared for the Query and Control components, respectively

C	`links: {`
	`      Query : [ "Server", "queryPort"] ,`
	`      Control : [ "Server", "controlPort"]`
	`}`
	False. This is not following the specification for service definitions. Concretely, we are missing the actual "`requires`" elements for both the `Query` and the `Control` components.

D	`links: {`
	`      Query : {  queryPort : [ "Server", "serverUrl"] },`
	`      Control : { controlPort : [ "Server", "serverUrl"] }`
	`}`
	False. `queryPort` is not a `requires` element of Query, etc…

E	All the above. False.
F	None of the above. False.

15. In the QueryLibrary service …

A	The values for the external references (in Server) can be modified in the service descriptor False. Bulletin of Lab3, page 11, service definition exports an object with `components` and `links` attributes. External references can be referred to only in the deployment descriptor (see page 13 from Lab3 bulletin)
B	The values for the ports specified within the “provides” attributes will remain unchanged for any deployment. False. Bulletin of Lab3, page 8, about “`requires`”, “`provides`”, “`export`”, and “`parameter`” attributes belonging to a component description, states that *such default values can (and most likely, will) be changed during a deployment*
C	The values of the external references (in Server) can be changed in the deployment descriptor. True. Please review explanation for option A of this question.
D	The value of the url specified within the “requires” attribute of “Components/Control/config/default.js” cannot change at deployment. False. On the contrary. Please review explanation for option B of this question.
E	All the above. False.
F	None of the above. False.

16. After finishing the deployment of the QueryLibrary service …

A	The only running instances belong to the Server component True. When the images for the Control and Query components are built, only the files within their corresponding directories are copied to the `/app` directory of the image built (see page 6 of the bulletin, when `ADD . /app` is discussed). Unfortunately, `utils.js` is outside that directory, and is not copied to either the `/app` directory or to the `/` directory of the resulting image. The consequence is that when the `require("../utils.js")` is run by the `server.js` file of either `Query` or `Control`, an unhandled exception is raised, and the node process started to run those components stops, and so does the container within which it runs. Thus the containers started for Query and Control components will stop immediately. The `Server` component, not executing that `require` call, is the only one that can survive.
B	There are at most two running instances of the Query component True. There are in fact NONE. See above explanation.
C	There are at most five running instances of the Control component True. There are in fact NONE. See above explanation
D	There is the same number of running instances of the Query and Control components True. 0 == 0
[✓ E]	All the above. True
F	None of the above. False.

General discussion for questions 17 and 18. In section 5 of the bulletin, you are asked to consider and think how to get a couple of different deployers: the `noLocationDeployer` and the `noImageDeployer`. To do so, one must read the code supplied with the Lab, in particular the code for the `basicdeployer` module that can be found within the `Deployers/basicdeployer/index.js` file.

A superficial reading of the code informs us that method `runComponent` is the one actually starting the containers containing the component instances. `runComponent` expects to be given the image to be run, as well as the configuration of the instance of the component. Thus `runComponent` will not need to be modified to get our new deployers.

`runComponent` is called from the deploy method of `Deployer`. This method is the one that actually makes sure that the images are built before calling `runComponent`, passing the image name to `runComponent`. Clearly, if the images are not prepared beforehand, this method has to do something different. In fact, reading lines 192 to 201 of the code, we find the logic to actually build the images. Within those lines we find a commented-out loop calling the `buildImage` method that actually ensures the needed images are built. The code in the loop actually depends on having the location information available. Thus the files supplied for the lab strongly suggest we need to modify the `deploy` method to build the images out of the location information.

If we ask ourselves where the information of the configuration information is used, a superficial reading of the code leads us to the function `configureDeployment`. This function is the one obtaining the configuration information out of the location attribute of the service description. In particular line 113 is the one obtaining such information for a component. If we do not provide explicitly such location within a service description, we need to change how we get to the configuration of the component, and change that line 113 for something else.

17. The `noLocationDeployer` mentioned in lab3 is built modifying …

A	The main code of `deployer.js` False. The main code does not access the service definition, only the deployment description.
B	The `deploy` method of `Deployer` False. Does not access the service definition. See above.
C	The `buildImage` method of `Deployer` False. This is an instrumental method. See above.
[✓ D]	The `configureComponent` function in `basicdeployer.js` True. See the general discussion above.
E	All the above. False.
F	None of the above. False.

18. The `noImageDeployer` mentioned in lab3 is built modifying:

A	The main code of `deployer.js`. False. Does not access the service definition.
[✓ B]	The `deploy` method of `Deployer` True. See general discussion above
C	The `buildImage` method of `Deployer`. False. This method needs no modification.
D	The `configureComponent` function in `basicdeployer.js`. False. Does not deal with images. See general discussion above
E	All the above. False.
F	None of the above. False.

Q. 17 and 18 are not used for base grading, as it was considered that referring to basicdeployer.js could have thrown students out, as no actual basicdeployer.js exists.

19. The command ‘`docker run -t -entrypoint=/bin/cat tsir/balancer /app/config/default.js`’:

NOTE: The –entrypoint option is equivalent to the Dockerfile ENTRYPOINT keyword.

A	Outputs to the console the contents of the default configuration file for component `balancer` True. According to the explanation of the `Dockerfile` used in Lab3 (page 6 of bulletin), the `ENTRYPOINT` directive used to build our images makes the containers start by running the command specified in that directive. In our case it is “`npm start`”. The command of this question is based on the basic functionality of “`docker run`”, in this case creating a container based on the `tsir/baselab3` image. The `--entrypoint` flag overrides the `ENTRYPOINT` directive of the `Dockerfile`, by providing a different command as an entry point, in our case `/bin/cat`. The last argument to our command is “`/app/config/default.js`”, which `docker` passes as an argument to the entrypoint command, `/bin/cat`. The end result is that the image `tsir/balancer` is used to launch a container in tty mode (flag `-t`) running the `/bin/cat` command on the `/app/config/default.js` file. Bulletin of Lab3, pages 5 and 6, states that all the application code for each instance will be placed in the /app directory, including component configuration file `/app/config/default.js`. Thus, we are printing to the console the configuration file of the `balancer` component.
B	Runs component `balancer` normally False. It provides a modification of its usual entrypoint
C	Ends with an error False.
D	Launches an instance of component `balancer` False. Balancer behavior is not achieved with this ENTRYPOINT modification.
E	All the above. False.
F	None of the above. False.

20. **The command "`docker build -t tsir/balancer Components/balancer`"…**

A	Builds image tsir/balancer for component `balancer` from the material found within directory `Components/balancer` True. A similar case was illustrated at page 6 from bulletin of Lab3, using current directory (.) instead of its relative path
B	Crashes because it is not launched within the `Components/balancer` directory. False. Directory name/path is provided as an argument to the command
C	Launches the balancer component False. This command doesn´t launch a container
D	Kills an instance of the balancer component False. This command doesn´t terminates a container instance
E	All the above. False.
F	None of the above. False.