**Why would you consider a Scripting Language as JavaScript as your Backend Platform.**

It depends. If my backend would be big, complex and need a lot of structure
I would probably write the backend in Java instead, since that is what Java does.
It's structured, it delivers solid engineering and architecture.
Scripting languages, like JavaScript, are super fast, it doesn't have to be compiled and sometimes measured up to 20x faster than code written in Java or any other alternative. It's simple and flexible.

**Explain Pros & Cons in using Node.js + Express to implement your Backend compared to a strategy using for example Java/JAX-RS/Tomcat**

PROS

______________________________________________________________________

You can write code that works on both server and in the browser.
This way you will not have to first write everything in Java and then later on everything again in JavaScript on the other side of your system. Working with Node.js you need not to divide your client/server relationship, meaning that you can move business logic as you pleases.

With JavaScript, code is much easier to migrate.

When making a build in Node.js you need not to switch programming language all of the sudden like you need to in Java. In Java you have to write the build specification in XML which isn't designed to support programming logic.

Node.js is really fast! The data comes in and the answers come out like lightning.
Node.js doesn't mess around with setting up separate threads with all of the locking headaches. There's no overhead to slow down anything. You write simple code and Node.js takes the right step as quickly as possible.

Javascript is the most popular programming language at the moment.
This means that a lot of people have knowledge about it, and (among other things) companies will not need a specialist to maintain a system.

CONS

______________________________________________________________________

Java has around 20 years of experience as a serverside language.
It has glitches and bugs, but all in all it's safer to use than JavaScript because the Java Virtual Machine is tested through endless of regression tests by the Sun/Oracle company.

Depending on what kind of backend you're creating, JavaScript lacks when it comes to the utility classes – sure a lot more plugins alike are on their way, but the developers at Sun/Oracle have invested a great deal in complex packages for more scientific work with its strong mathematical foundation; there are BigIntegers, elaborate IO routines, and complex Date codes.

You easily get confused with JavaScript where functions which do not have any answers give three different results; undefined, NaN, and null.

JavaScript may provide you with fast code, but this is only the reality if written correct.
Java's Web servers are multi-threaded. Creating multiple threads may take time and memory, but it pays off. If one thread deadlocks, the others continue. If one thread requires longer computation, the other threads aren't starved for attention (usually). If one Node.js request runs too slowly, everything slows down.
There's only one thread in Node.js, and it will get to your event when it's good and ready. It may look superfast, but underneath it uses the same architecture as a one-window post office in the week before Christmas, however it is possible to use the advantage of multicore systems with Node.js, but it's a bit more complicated to implement than with Java.

**Node.js uses a Single Threaded Non-blocking strategy to handle asynchronous task. Explain strategies to implement a Node.js based server architecture that still could take advantage of a multi-core Server.**

To take advantage of more than a single core in a multi-core system, engineers have two main choices.
The first choice is to let resource allocation happen at the system level wherein incoming requests are distributed to multiple single-threaded Node.js processes each running in a virtual machine assigned a single core from the multi-core processor. All the instances live behind a proxy which serves to balance incoming requests to available Node processes.
This choice is a fine one and works nicely as long as the costs of virtualization are considered acceptable.
However, the Node community has come forward with an even better second solution which bakes in multi-core awareness and capability natively into the Node.js platform.
This second choice is the use of a module that enables "clustering" Node.js which runs as many dedicated single threaded processes under a master Node.js process without the need for elaborate virtual machine infrastructure.
This results in significant performance gains without the costs associated with virtualization and is described next.

**Explain, using relevant examples, the Express concept; middleware.**
Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named next. If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.
An Express application can use the following types of middleware:
- Application-level middleware
- Router-level middleware
- Error-handling middleware (NOT EXPLAINED)
- Built-in middleware (NOT EXPLAINED)
- Third-party middleware

ABOUT THE 'NEXT()'
next() call inside a middleware invokes the next middleware or route handler depending on whichever is declared next. But next() call inside a route handler invokes the next route handler only. If there is a middleware next then it's skipped. Therefore middlewares must be declared above all route handlers.
Examples: see middleware.js

**Explain, using relevant examples, how to implement sessions, and the legal implications of doing this.**

When a user first logs in or registers for your site, you know who they are because they just submitted their information to your server. You can use that informaton to create a new record in your database or retrieve an existing one - simple!
But how do you keep them authenticated when they do something crazy like reload the page?
Magic, that's how! Also known as sessions...
That being said, a 'session' is a squishy, abstract term for keeping users logged in.
We care more about the actual mechanism for persisting authentication; namely, cookies.
The most delicious part of user management:
Cookies allow you to store a user's information inside a file on the their browser.
The browser then sends that info back on every request, allowing your application to identify the user and customize their experience. Which is objectively way better than asking for a username and password on every request.

Example: see Sessions.js