**9.1 Different frameworks**

**KOA**

**KOA VS Express**

How is Koa different than Connect/Express?

* Generated-based control flow
* No callback hell.
* Better error handling through try/catch.
* No need for domains.
* Koa is barebones
* Unlike both Connect and Express, Koa does not include any middleware.
* Unlike Express, routing is not provided.
* Unlike Express, many convenience utilities are not provided. For example, sending files.
* Koa is more modular.
* Koa relies less on middleware
  + For example, instead of a “body parsing” middleware, you would instead use a body parsing function.
* Koa abstracts node’s request/response
* Less hackery.
* Better user experience.
* Proper stream handling.

**PROS**

* Generator support from ground up Using generators (a bleeding edge feature, even for Node.js) would clean up your code from the mess caused by all those callbacks; making your code more manageable.
* Development team has a proven track record
* Koa is developed by the team behind a widely used node.js framework (express.js).
* Extremely lightweight
* Koa is very lightweight with just 550 lines of code.
* sync/await keywords are supported and has transcended beyond generator functions
* Generators functions are of course a huge plus, but at the time Koa team has transcended generations functions and shifted towards async/await style programming. It has made the Koa best framework available in the market.

**CONS**

* Community is relatively small.
* Not compatible with express style middleware.
* Koa uses generators which are not compatible with any other type of Node.js framework middleware.

[**Socket.io**](https://socket.io/) **(a library)**

pros

* The real-time analytics gives high-speed support for real time apps as chat room applications, multiplayer games, video conferencing
* The durability of the web and the app development process it gives make it stand out.
* It keeps tracking modifications as well as additional functionality.
* The auto-identification of the errors and correcting them becomes easier.
* There are similar APIs available for client and server-side development.

cons

* WebSocket client cannot be successful in connecting to a Socket.IO server. Besides, the Socket.IO client will not be able to connect to a plain WebSocket.
* Development focus
* Bugtracker- see like 20+ pull requests and 200+ open defects. That's not very good sign, even if 90% are not bugs.
* Stability  
  Protocol and behavior patterns are poorly documented. No unit tests. No protocol tests. No client-side library tests.
* Client has lots of places where race condition can happen, which either kill your server (like issue #438 mentioned above, good it was fixed)

**HAPI**

Pros

* **Robust Plugins:**Offers rapid feature development and quicker identification of bus to fix through customized modules.
* **Ease of Coding:** Helps developers to write manageable, controllable, and distributable code.
* **Secure Framework:**Blocks error messages which may leak data or echo back unsecured loopholes.
* **Caching Advantage:**Improves the web application performance through continuous caching.
* **Microservice Support:** Chairo plugin allows integration of Seneca functionality which is a microservice toolkit for organizing an application’s business logic.
* **Integrated Authorization:** Accessibility to pre-built authentication and authorization APIs.

Cons

* **Code Configuration:** Developers have to figure the code structure without community support
* **Compatibility Issues:** Some of the Hapi-specific modules and plugins such as catbox, joi, boom, etc. are not compatible with Express/Connect.
* **Manual Testing:** Creation of endpoints and testing is manual, due to lack of automation.