€mporioλambda

by Red Babel

1 E-commerce platforms

E-commerce, also known as electronic commerce or internet commerce, refers to the buying and selling of goods or services using the internet, and the transfer of money and data to execute these transactions. E-commerce is often used to refer to the sale of physical products online, but it also designates any kind of commercial transaction that is facilitated through the internet[1].

Those platforms have a unique consumer journey from clicking the website/app and entering the home page to search for a product to product listing pages (PLP) to product detail pages (PDP) and then the shopping cart. From the shopping cart, the customer can proceed to the payment of the selected products, this part of the user flow is called checkout.[2]

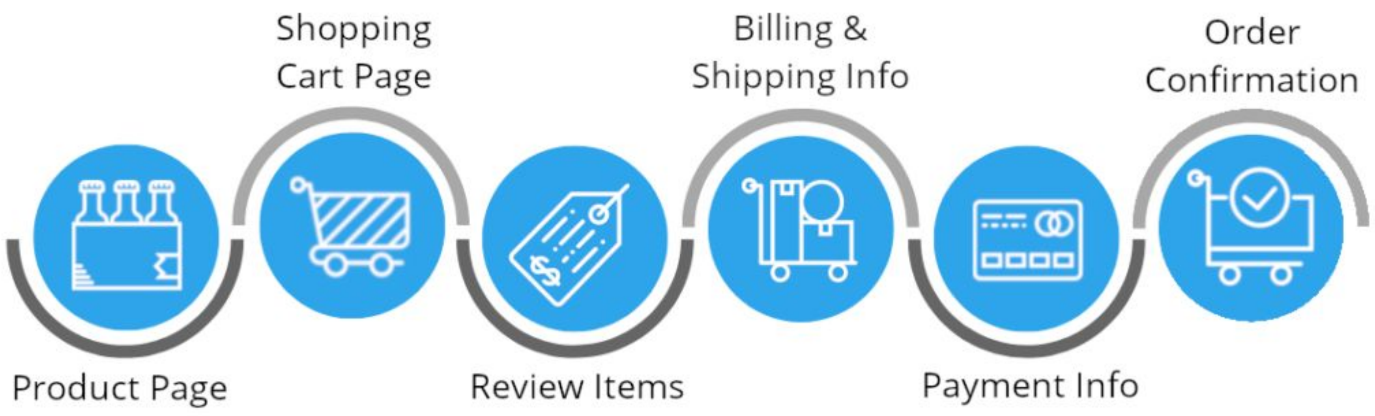

Figure 1: Checkout process. Source: thegood.com, “25 Ecommerce Checkout Process Best Practices that Convert”[3].

1.1 Background information

E-commerce platforms can range from very simple/trivial to highly complex for a variety of reasons. They need to fit the business requirements of the company that holds the product lines. Achieving this fitness-for-purpose is, for lack of better words, not trivial.
Business requirements can vary from:

- Tax calculation: sales taxes and excise taxes that can differ between regions/markets/products,
- Loyalty systems: discounts, promotions, and points systems that can be online and offline;
- Inventory availability: keeping track of stock levels and refill or declare out of stock product, while back syncing with the warehouses responsible for fulfillment/shipment;
- Geographical/Legal requirements: products can be banned in some areas and can not get shipped in other areas;
- Brand/marketing: the user facing website needs to be easily changeable, independently by other systems, while keeping sane and healthy performance.

Most of the examples above sparked entire industries by themself. Companies integrate with third party services in a complex web of inter dependencies.

[1] https://www.shopify.com/encyclopedia/what-is-ecommerce
[2] https://brandalyzer.blog/2018/06/05/the-consumer-journey-on-an-ecommerce-app-home-clp-plp-pdp-cart
[3] https://thegood.com/insights/optimize-checkout-process

More so, business needs are not fixed in time: they change with the market or the company strategy and the platform must change accordingly and quite literally continuously. Since the application must evolve continuously in time and features, the software architecture becomes increasingly more complex in order to accommodate all the previous changes and foresee the future ones.

The technology is rooted in the usual tools and paradigms known in Software Engineering while the way of working and the processes are instead of a very different nature. For all these reasons, building an e-commerce business is a highly complex, intense and dynamic endeavour.

In this project we are trying to explain the complexity of e-commerce applications, by presenting the main components and simply sketching others, but at the same time delivering a fully functional website that fulfills the business requirement. In the end it is the, strange, never ending[4] battle between simplicity, business and technical challenges that make e-commerce applications thoroughly complex.

1.2 Primary components

A typical e-commerce website usually consists of one set of *customer-facing functionalities* and a second set of non-customer-facing functionalities. While the former implementations generically conform one to another, the latter can be wildly different. Here we define the latter as generic *back office functionalities*.

Back office includes all the processes and functionalities used by employees needed for the business to operate. As mentioned, such functionalities can be quite different. Accounting, finance, inventory, order fulfillment, distribution, and shipping are all examples of back office systems.

Back office systems can be manual or automated. Effective integration of back office systems with the e-commerce application improves coordination with the customers facing components resulting in better customer service and reduced duplication of effort by staff[5]. Examples of back office processes include: customer support, payments refund, warehouse integration, operations tracking.

The technological needs of the two sets are quite different in nature. In this tender we will focus mainly on the former while the latter will be marginally glossed over.

Customer facing functionalities are usually comprised of the following:
- Home page;
- Product Listing pages;
- Product Description pages;
- Shopping Cart;
- Checkout;
- Account.

1.2.1 Home Page

An e-commerce home page might show various key promotions and a clear structure of what the website can offer. The consumer must be able to access immediately to the search bar and the menu structure. It is very crucial for the customers to be easy to use and a strong accent is put on performance.

[4] https://www.youtube.com/watch?v=ehtsmxu1w10
[5] https://www.ontario.ca/page/integrating-back-office-systems-e-commerce#section-1

In some cases it can be built on a very different technological stack than the rest of the platform. It can be hosted on a third party service, with capability of caching (via content delivery network, CDN), A/B testing[6], analytics/tracking systems and even different cloud providers per geographical region. It is usually ad-hoc optimized for different devices (e.g. desktop, mobile, tablet). Search engines optimizations (SEO) techniques are heavily used in this page.

1.2.2 PLP

A Product Listing Page is where the customers filter the list of products by category. This step would typically come after the homepage which helps customers find products based on their description or title. The PLP is important to promote certain products and increase brand awareness since you can see many options at the same time. SEO techniques are as important as in the Home Page.

On the technological side an approach similar to the home page can be used, although the PLP pages are more dynamic in nature. Different search and filter capabilities are necessary in this context. Paramount here is the structure of the products and all the metadata associated with. This structure needs to be flexible to allow for an efficient implementation on the backend system serving the data. Therefore, the design of the PLP pages, the product metadata structure, and the matching backend pages needs to be done contextually. Decisions made here usually will last, and will impact, the project for a long time after being made. Stock availability, prices, promotions and discounts active, are some, non-trivial, pieces of information that need to be retrieved to show this page.

1.2.3 PDP

Product Detail Pages are the companion of PLPs. They are directly linked to each other. URLs for PDPs pages are considered essential for SEO and called slugs[7]. A PDP contains all the details and specs about the product of interest. It is the page that shows the product's features, giving them a more in-depth look at the product with all the benefits. A call to action (CTA) button is traditionally present with the obvious action "add-to-cart" or "buy-now": add the specific item in the basket/cart of the user.

Such apparently simple detail is usually bolted with many constraints. As an example, that particular item could not be available for that specific user for geographical or legal restriction. In general there might be a dynamic set of business constraints that need to be considered/applied. That logic can be highly complex and it is normally defined in custom software. This responsibility is deferred to the next two components: shopping cart and checkout.

1.2.4 Shopping Cart

A shopping cart is the software component that holds the information of the current items selected for later purchase by the customer. The shopping carts allow customers to:

- review the products they have selected;
- make changes, like adding or deleting items, change product quantity, selecting different variants for the products);
- apply discounts/promotions/vouchers;
- fill the information needed to complete the order such as shipping/billing address, email etc etc;

A desirable feature is persistence of carts between sessions and across devices/browsers. These requirements lead to peculiar design choices in regard to the architecture and the actual implementation. Classic problems of concurrency, versioning and performance are very much relevant in this context.

[6] https://en.wikipedia.org/wiki/A/B_testing
[7] https://en.wikipedia.org/wiki/Clean_URL#Slug

1.2.5 Checkout

Checkout is the digital process that a customer must go through when checking out the items in the cart. The process encompasses the specific steps a consumer must take when completing a purchase. It starts when the customer clicks the Checkout button in the Shopping Cart.

The resulting page usually shows the order summary, the list of the selected items with prices, taxes, discounts and a breakdown of total price. The next step is the payment step, that is usually handled by third party systems for security and convenience reasons. A payment provider offers an abstraction over the complexity of a payment system (e.g. they need to be PCI[8] compliant). Financial information (i.e. sensitive data such as Credit card numbers) never touches the e-commerce system and is only handled by the payment providers. Payment providers then communicate with the backend systems of e-commerce platforms to receive confirmation of the payments. Such integration is asynchronous in nature and Checkout processes need to take this in the architectural design.

The pages holding the checkout process are dynamic and are not required to have the same requirements as the Home page, PLP/PDP. SEO is not relevant here and performance is achieved not via caching strategies but via slick APIs and asynchronous architecture. The technological focus is heavily shifted on the backend.

1.2.6 Account

Account contains all the information available about a user, that can be accessed and managed by the user itself. Order history, profile information, delivery address or payment methods are all examples belonging to this category. A striking difference to the other components is the necessity to have an authentication/authorization layer (Checkout can be an hybrid as guest checkout could be allowed based on the business requirements).

An authentication provider should be implemented but a different route could be to integrate with a third party service such as Amazon Cognito[9] or Auth0[10]. The authorization layer is necessarily custom implemented with different permissions (e.g. customers can access only their own data, while admin can access different partitions/views of the same data).

[8] https://en.wikipedia.org/wiki/Payment_Card_Industry_Data_Security_Standard
[9] https://aws.amazon.com/cognito
[10] https://auth0.com

2 Serverless architectures

Serverless architectures are application designs that incorporate third-party “Backend[11] as a Service” (BaaS) utilities, or that include custom code run in managed, ephemeral containers (those that last for a single invocation only) on a “Functions as a Service” (FaaS) execution platform. By using these ideas, and related ones like single-page applications (SPA), such architectures remove much of the need for traditional always-on server installations. Serverless architectures may benefit from significantly reduced operational cost, complexity, and engineering lead time, at a cost of increased reliance on vendor dependencies and comparatively immature (as yet) supporting services.

Serverless applications are event-driven cloud-based systems where application development relies solely on a combination of third-party services, client-side logic and cloud-hosted remote procedure calls (known as FaaS)[12]. Going serverless lets developers shift their focus from the server level to the task level. Serverless solutions let developers focus on what their application or system needs to do by taking away the complexity of also handling the backend infrastructure.[13]

Serverless computing is a cloud-computing execution model in which the cloud provider runs the server, and dynamically manages the allocation of the required computational resources. Server-side logic is still written by the application developer, but, unlike traditional architectures, it is run in stateless compute containers that are event-triggered, ephemeral, and fully managed by a third party. At present, Amazon Web Services (AWS) Lambda is one of the most popular implementations of a FaaS platform.[14]

2.1 AWS Lambda

AWS Lambda is an event-driven, serverless computing platform provided by Amazon as a part of the Amazon Web Services. It is a computing service that runs code in response to events and automatically manages the computing resources required for the execution of that code[15]. A Serverless app can simply be a couple of lambda functions to accomplish some tasks, or an entire back-end composed of hundreds of lambda functions[16]. In addition to lambda functions, an application might also use other components such as AWS API Gateway (for HTTP events), AWS DynamoDB (scalable and distributed key-value and document database), or AWS S3 (object storage).

Those resources could be managed using CloudFormation, an AWS tool for deploying infrastructure. You describe your desired infrastructure in YAML or JSON, then submit your CloudFormation template for deployment. CloudFormation realizes infrastructure as code (IaC), which is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools[17].

2.2 Serverless framework

The Serverless Framework[18] is a free and open-source web framework written using Node.js. It provides a configuration DSL which is designed for serverless applications. It also enables IaC while removing a lot of

[11] Backend in the sense of the software infrastructure hosting the execution of the application.
[12] https://hackernoon.com/what-is-serverless-architecture-what-are-its-pros-and-cons-cc4b804022e9
[13] https://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless
[14] https://martinfowler.com/articles/serverless.html
[15] https://en.wikipedia.org/wiki/AWS_Lambda
[16] https://en.wikipedia.org/wiki/Serverless_Framework
[17] https://en.wikipedia.org/wiki/Infrastructure_as_code#cite_note-AWS_in_Action,_IaC-1
[18] https://serverless.com

the boilerplate required for deploying serverless applications, including permissions, event subscriptions, logging, etc. When deploying to AWS, the Serverless Framework is using CloudFormation under the hood. This means you can use the Serverless Framework's easy syntax to describe most of your Serverless Application while still having the ability to supplement with standard CloudFormation if needed.

Most importantly, the Framework assists with additional aspects of the serverless application lifecycle, including building your function package, invoking your functions for testing, and reviewing your application logs.

3 EmporioLambda

EmporioLambda (EML) is an incarnation of an E-commerce platform built entirely using Serverless technologies.

In this tender we are asking the supplier to build a generic E-commerce platform that can be shown as a concept software to sell to merchants. EML needs to be deployable using the AWS Merchant account with a minimal amount of manual configuration. We require the supplier to deploy a demonstrative platform to show the effectiveness of the solution created. The topic of the demo can be freely chosen by the supplier upon agreement with the proponent.

To set the perimeter of the project we will dictate a main draft of architecture with some guidelines and some mandatory technological choices. Within such canvas the supplier is free, and *highly* encouraged, to roam around and explore different choices. The proponent will act also as sparring partner to discuss options, choices and technological challenges.

3.1 High level architecture

The overall architecture of EML is based on a microservices architecture. This architectural pattern
is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies[19].

The decision on how to design the services is left to the supplier. The usage of the Serverless framework as centralized management is instead mandatory. A service must be implemented with an Application Programming Interface (API) HTTP based.

Services are essentially a vast number of parts functioning in concert with one another toward a single goal. Nevertheless such systems, over time, reveal cracks or weaknesses that need to have a small piece of code to align everything the way it should be. The Backend for Frontend (BFF) refers to the concept of developing niche backends for each user experience.[20]

A BFF is, in simple terms, a layer between the user experience and the resources it calls on. The BFF is tightly coupled to a specific user experience, and will typically be maintained by the same team as the user interface, thereby making it easier to define and adapt the API as the UI requires, while also simplifying the process of lining up release of both the client and server components.[21].

In this tender we will require the usage of Next.js[22] for the BFF component.

The main parts of EML are built using Serverless and NextJs along with integrations with third party services such as: payment provider, content management service and an identity management service.
Serverless will be responsible for the execution and the deployment of the application. NextJS will be the framework responsible for rendering and serving the UI on the browser and will act also as BFF.

[19] https://martinfowler.com/articles/microservices.html
[20] https://nordicapis.com/building-a-backend-for-frontend-shim-for-your-microservices
[21] https://samnewman.io/patterns/architectural/bff/#bff
[22] https://nextjs.org

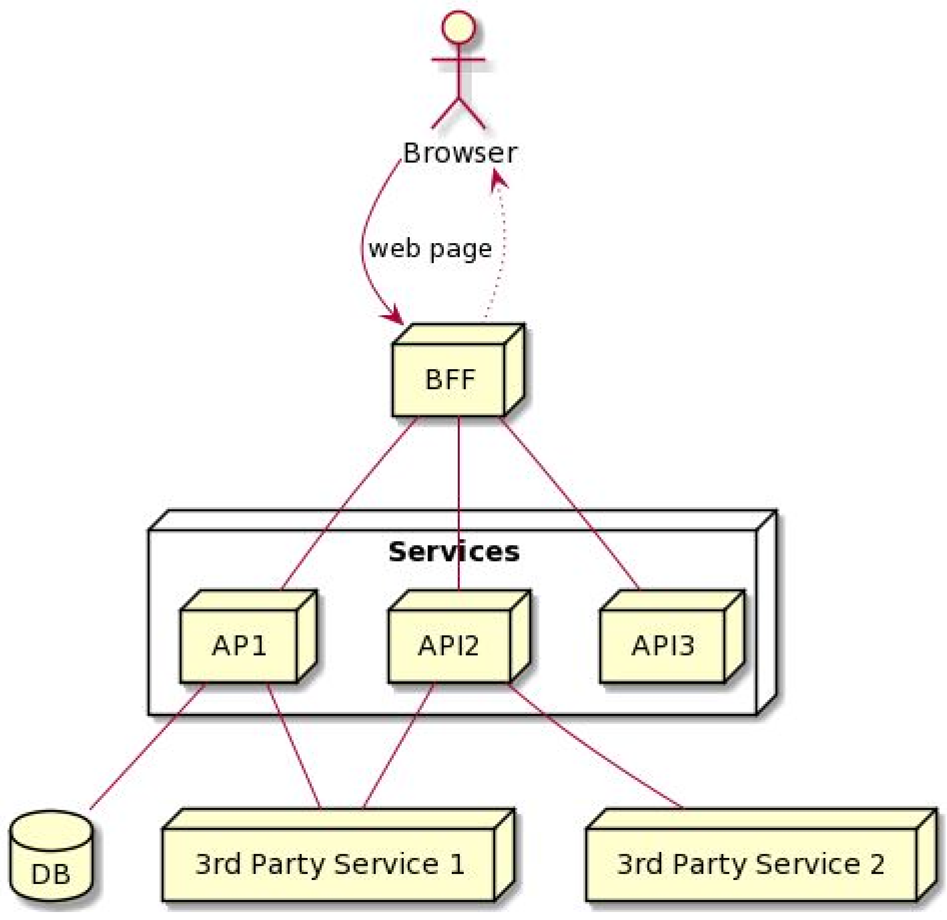

Figure 2: High level architecture diagram

3.2 High level modules

EML architecture consists of 4 main modules:

1. EmporioLambda-frontend (EML-FE): is the module that serves the web pages requested by the customer. EML-FE job is to pre-render the page into HTML on the server on every request, this functionality is called pre-rendering. Pre-rendering can be implemented in 2 different ways:
   - Server-side Rendering (SSR): pre-render the page into HTML on the server on every request[23];
   - Static Generation (SSG): pre-renders the page into HTML on the server ahead of each request, such as at build time;

2. *EmporioLambda-backend* (EML-BE): is the module that exposes the services of the application. EML-BE is responsible for:
   - Implement the application business logic;
   - Manages the data of the application (i.e.: orders, user information, product information);
   - Managing the cart state;
   - Integrate 3rd party services;

[23] https://vercel.com/blog/nextjs-server-side-rendering-vs-static-generation

3. *EmporioLambda-integration* (EML-I): represents all the third party services integrated with EML-BE.

4. *EmporioLambda-monitoring* (EML-MON): is the set of tools used by the admin for monitoring the status of the application.

3.3 Development environments

A best practice in industry is to divide software development across different environments. Each environment provides a set of resources (e.g. networks, servers, file storage, others) needed to run an instance of the system. Every environment is capable of running the whole system, but it is entirely independent of the others. Therefore, using one resource in one environment (e. g. store a file or write in a database) does not conflict with the usage of resources in another. Environments are used to test the system before deployment to production. They also allow developers to run the system locally.

In a simple scenario, we have at least 4 environments encompassing the following flow across them. The developer builds some features and runs them on her local computer. When satisfied, she will build a suite of (verification) tests. Such tests could be run locally or as part of a continuous integration[24] process. The features will be then deployed on a staging environment so that other people can use/test such features. Finally, the release cycle will determine when the final deployment to production will occur.

This project **must** use the following minimal number of environments: Local, Test and Staging. Production is not required for the scope of this project.

1. Local: the local development machine of each developer.

2. Test: deployed in AWS to test integrations between subsystems, accessible by all developers. The environment can be a simplified version from an architectural point of view. No guarantees of data integrity or stability are required (e.g. databases can, and usually are, deleted at will). Breaking changes can land in tests.

3. Staging: must be publicly accessible and the environment should be equal as the production one bar the credentials to the productions integration.

4. Production: not required, but the project should be production-ready.

3.3.1 Third party integrations

Depending on the third party service integrated with EML, the number of environment may vary:
- Identity Manager will have Local, Test and Staging as environments
- Payment service will have Test and Staging as environments
- CMS service (optional) will have Local, Test and Staging as environments

[24] https://docs.microsoft.com/en-us/azure/devops/learn/what-is-continuous-integration

4 Requirements

We require the implementation of all of the high level modules:

1. EML-FE will be implemented in Next.js. Using Typescript as the main language. This component will act also as BFF;
2. EML-BE will be implemented in Serverless using Typescript as the main language. The deployment shall be done in AWS using AWS Lambda as only computational unit;
3. EML-I will be implemented in Serverless using Typescript as the main language;
4. EML-MON will be implemented using Amazon CloudWatch[25]. An alternative monitoring system to that can be selected is Datadog[26].

4.1 Minimum

4.1.1 Functional requirements

1. *Homepage*. From the homepage the consumer can:
    a. Search for the product he needs;
    b. Access to the shopping cart page;

2. *PLP*. The customer can:
    a. Select multiple product items;
    b. Add the selected products to the shopping cart;
    c. Navigate to the shopping cart page;

3. *PDP*. A dynamically generated web page where the customer can read all the information related to the selected product. In addition the user can add the product to his cart. In the PDP page the user can:
    a. View the product information (e.g.: description, images, price, taxes, ...);
    b. Add the product to the cart;
    c. Click the link to redirect to the shopping cart page;

4. *Shopping Cart page*. Where the user can:
    a. Review all the products he selected;
    b. Review the the total price and total sales tax;
    c. Proceed to the checkout;

5. *My profile*. Where the registered user can:
    a. Update profile information (ie: picture, description, email address);
    b. Have an overview of all the orders he made on the website;

6. *Checkout*. The page where the user can:
    a. Insert payment details;
    b. Insert email address for the delivery of the notes;
    c. Proceed to the payment;

[25] https://aws.amazon.com/cloudwatch
[26] https://www.datadoghq.com

If the payment is successful the consumer will be redirected to the final step of the checkout process, where he can review the order just created. If the payment fails, the application will notify the consumer, and he can retry to insert again the payment details and retry the payment.
In case of successful payment, the application will send the product(s) via email (previously inserted in the checkout process) to the customer;

7. *Merchant dashboard*. Where the merchant can:
    a. Add and remove products
    b. Update products description
    c. Have an overview of all the orders that has been created on the application

4.1.2 Roles

The website must be able to support the following user roles:

- Admin, the user with this role can:
    - deploy the application to the cloud
    - Manage the configuration of the 3rd party integrations
- Merchant, the user with this role can:
    - overview of all the orders
    - Add, remove and update product information
- Customer, users with this role can:
    - Can search, filter and add to the cart as guest or logged in user
    - If logged in can update, change his profile information
    - Can delete the created account
    - Can proceed with the payment of the selected products only if it is logged in

The roles are modelled and implemented using AWS Cognito Identity[27] as Identity manager.

4.1.3 Integrations

The only mandatory integration required is with the payment provider. The selected provider is Stripe[28]. A different choice could be proposed upon agreement with the proponent.

4.2 Optional

4.2.1 Content Management Systems

In order to improve the user experience, EML can integrate a Content Management System (CMS). Through the CMS, the admin can, for instance, modify the copy of the website dynamically, or change the layout of the PDP or PLP pages **without redeploying** the entire code for every change. The CMS of choice for this option is Contentful[29]. A different choice could be proposed upon agreement with the proponent.

4.2.2 Identity Manager

Another common issue that e-commerce platforms encounter is identity management. In order to be able to migrate the website from one infrastructure to another, and in this way without being locked in a cloud

[27] https://aws.amazon.com/cognito
[28] https://stripe.com
[29] https://www.contentful.com

system, it is necessary to store user credentials in an external third party identity provider. Usually they implement standard security protocols such as OAuth 2.0 in order to facilitate their integration with the e-commerce websites. The Identity Manager of choice is Auth0[30]. A different choice could be proposed upon agreement with the proponent.

4.3 Technology

1. EML will be developed using the latest Typescript version using a promise[31]/async-await[32] centric approach;
2. *typescript-eslint* must be used and enforced using ESLint[33] throughout the development process;

The source code of EML should be published and versioned using either GitHub or GitLab.Along with the source code, the necessary documentation should be provided for the end user to use the system and for the developer to run and deploy the modules.

4.4 Warranty and maintenance

The vendor has to demonstrate at the RA *(Revisione di accettazione)* that the product works correctly and according to requirements. Fixing bugs, flaws and any not-compliance with the requirements are entirely at the expense of the vendor.

4.5 Credits and License

RedBabel holds interests in this project as **proof of concept** of the productivity of the technologies specified above. The system will be distributed under the MIT license, the developer will be mentioned in the copyright credits. Redbabel will be credited to, under the section credits in the README file. Please refer to the Italian law about the management of public bids for what is not specified in this technical specifications document.

4.6 Useful links

- https://vercel.com/blog/nextjs-server-side-rendering-vs-static-generation#e-commerce-next.js-app-example
- https://auth0.com/docs/authorization/which-oauth-2-0-flow-should-i-use
- https://www.serverless.com/blog/serverless-nextjs
- https://github.com/serverless/examples
- https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-scenarios.html

5 The proponent

RedBabel is a webcraft consulting firm based in Amsterdam, NL, and formed by Alessandro Maccagnan and Milo Ertola. The firm operates in Amsterdam and Italy where it holds close relationships with the respective industry ecosystem.
Contacts:

- Alessandro Maccagnan: alessandro@redbabel.com
- Milo Ertola: milo@redbabel.com

[30] https://auth0.com
[31] https://basarat.gitbooks.io/typescript/docs/promise.html
[32] https://basarat.gitbooks.io/typescript/docs/async-await.html
[33] https://github.com/eslint/eslint

To allow for a better and seamless communication between us, the proponent, and you, the vendor, we provide a Slack group available to all group members. To access it: https://communityinviter.com/apps/unipd-math/invitation.