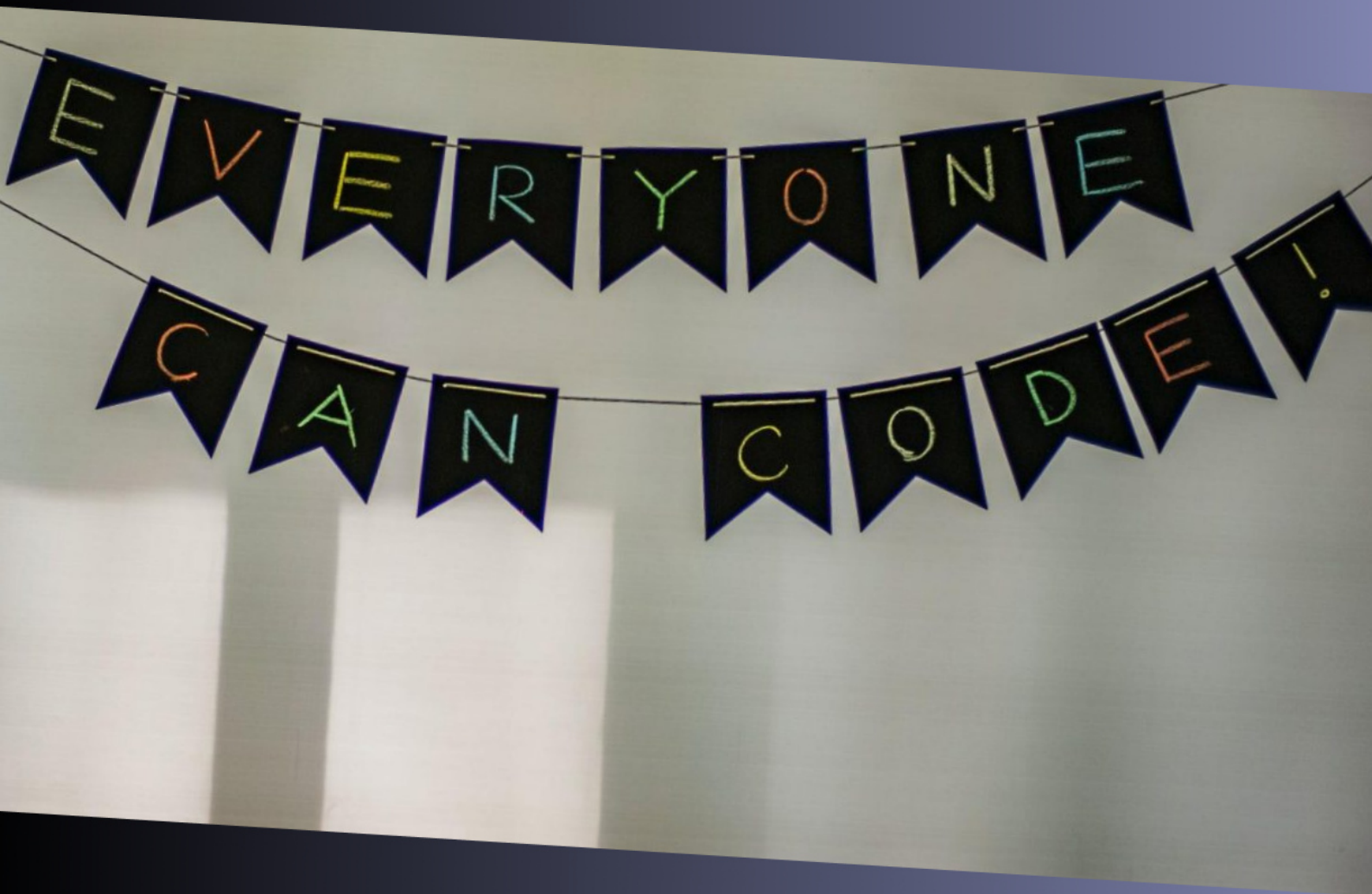




# TECHNOLOGIES BEHIND NO-CODE & LOW-CODE SOLUTIONS AND HOW TO BUILD YOUR OWN



written by

**Yuriy Luchaninov**

JavaScript Group Leader at MobiDev

# SOFTWARE DEVELOPMENT FOR VISIONARIES WHO CREATE PRODUCTS



WEB & CLOUD  
INFRASTRUCTURE



AI AND MACHINE  
LEARNING



IOT & HARDWARE  
INTEGRATION



AR & MOBILE  
APPS

For Startups

For Emerging companies

For Enterprises



**Guaranteed delivery**  
on time and on budget  
**No surprises**



You can **adapt to evolving**  
**business needs and increase ROI**  
with our flexible, proven processes



Top US-level quality  
for **1/3 the price** to bring  
**3x features** to your product


**450+** Products  
launched

**100%** Approval rating  
by **upwork**

**300+** English speaking  
professionals

Find more at **www.mobidev.biz**

 [info@mobidev.biz](mailto:info@mobidev.biz)

 +1 888 380 0276

UNITED STATES OFFICE  
Atlanta, Georgia

UNITED KINGDOM OFFICE  
Sheffield

ENGINEERING OFFICE IN POLAND  
Lodz

ENGINEERING OFFICE IN UKRAINE  
Chernivtsi

# Table of Contents

[Technologies Behind No-code / Low-code Platforms](#)

[Editor -> Code + Meta Information -> Code](#)

[Editor -> JSON -> Code](#)

[Editor -> Intermediate Components -> Code](#)

[What It Takes to Build Your Own No-code / Low Code Platform](#)

No/Low-code platforms have become a trendy topic recently. But there is almost no information on what it takes to create your own solution of this kind.

Therefore, we will focus on the development approaches, architecture of low and no-code platforms, and decomposition. Let's find out how no-code platforms are built.

For starters, we will define low/no-code development. This term designates a platform on which the user can generate any IT product for external use via the UI interface. It can be separate web pages, complete websites, a mobile application, PWA, or chat flow builder. According to [Statista](#), the global low-code platform market revenue is projected to reach 65 billion U.S. dollars in 2027. So, all the aforementioned IT products have development prospects.

The high demand for business applications drives the need for the development of new low and no-code platforms. In most cases, any no-code platform consists of:

- UI part – visual interface with components, drag & drop objects, and forms. It allows users to create interfaces for their products. The UI has to be intuitive and easy to use by a wide range of people with no technical background. [UI/UX designers](#) often put a lot of effort on this stage.
- UI into the code transformer – this part often looks like some kind of magic, since the application code is being generated out of the visual interface and its components.

- Business logic – implementation of components that support interface functionality on the back-end (registration, billing, etc.)
- Automated code builders – an optional part that allows to automate packing the code into an app or a website and deploy it into production.

The above observations suggest that the main difficulty may arise in the second part of the system (UI into the code transformer), yet there are subtleties to be investigated on each step. We will consider the development of all components based on the case of building sites, noting specific moments for chat builders and mobile application constructors as the most common cases.

What is the first thing the end-user sees when embarking on the product development journey? It's a UI interface related to the constructor. So a professional approach and analysis of the interface are crucial here – many do not understand why this stage is time-consuming. That is due simply to the advancement of the editor's interface which influences ease of use. Ease of use means that less effort will be spent on creating a software product, thus streamlining the process for the end-user.

The constructor's design is crucial for the success of the platform and must be carried out by professionals who understand this area. In addition, it is necessary to separate the constructors for creating the UI interface of the developed software product and the constructors for describing its logic. In the second case, difficulties related to the design of this functionality may arise.

## **Technologies Behind No-code / Low-code Platforms**

Approaches to building an internal editor's architecture may differ, influencing platform functionality.

Currently, there are three main approaches to building an internal architecture:

1. Editor -> code + meta information -> code
2. Editor -> JSON -> code
3. Editor -> intermediate components -> code



Let's take a closer look at each of these approaches and find out relevant use cases.

## **Editor -> Code + Meta Information -> Code**

This approach generates the resulting code on the go and allows users to pick and customize certain elements through a visual editor. As the customization of a certain element is done – it automatically changes the codebase to keep it up to date with the changes. In the vast majority of cases, this approach is used precisely as the basis for editing web pages: code becomes an HTML page with its visualization. As all the editing is done on the browser side, it's easy to export the final code as a product by copying the DOM structure through the browser.

The subtleties of the first approach include the following points:

- The need to work not with all HTML markup, but only with components. Anyone who has ever seen the HTML layout of a complex page can understand that it may pose challenges. If you enable the user to edit each tag, then the interface will turn into a mess and become unsuitable. It would be easier to build a page from scratch than apply such a constructor. Therefore, there is a need for an additional markup, the so-called metadata. The information for the editor is added in the form of custom attributes of HTML tags. The data indicates that the inner block can be edited or transferred, while parameters and attributes are easily modified.
- The position of elements or blocks – absolute, relative, static, fixed, and sticky – may be tricky. What's the reason for this? With regard to this point, everything is more complicated and doesn't lie on the surface since the position of each element directly affects the position of other UI blocks on the entire page. This is comparable to a glass into which different balls are poured. If you try to put a wooden block in the middle of glass, all the balls will change their location as they will be squeezed out from their position. We can draw an analogy between the described situation and the positioning of elements on the page. The situation with parameters related to the location of blocks is a little more complicated, though. The alignment of blocks and "squeezing" laws depend on the specified parameter.

- Limited drag and drop functionality may irritate the user because it will be relevant only for some UI blocks. Without getting into too much detail about this problem, we should mention that it is similar to the positioning of blocks.
- Styles in CSS (Cascading Style Sheets) won't work as expected by the user. The reason lies in inheritance: styles set for the parent components will directly affect the child ones, this will confuse the end-user, and will look like a clear UI flaw or bug. And, in turn, the platform's UX and its use will decrease. Thankfully, this can be solved with the help of unique CSS classes or inline styles, though the end-user will still face the challenge of side effects.
- It's impossible to import the code. In addition to the direct HTML markup (as an example in our case), the editor requires adding custom attributes to the markup as a template, which is responsible for the functionality of the editor.

Due to the mentioned points, the approach based on the scheme Editor -> code + meta-information -> code is often used not in no-code but in low-code platforms, the end-users of which understand the technical specifications and side effects of the modifications. The target audience of low-code platforms narrows down to developers who are aware of similar or more effective tools.



**Read also:**

[TOP 10 Web Development Trends: How to Stay Ahead in 2022](#)

## Editor -> JSON -> Code

To address the shortcomings of the first approach and extend the list of supported platforms, without being limited to HTML or code that can be displayed as UI components, we can consider the editor's architecture based on the intermediate data model (Editor -> JSON -> code).

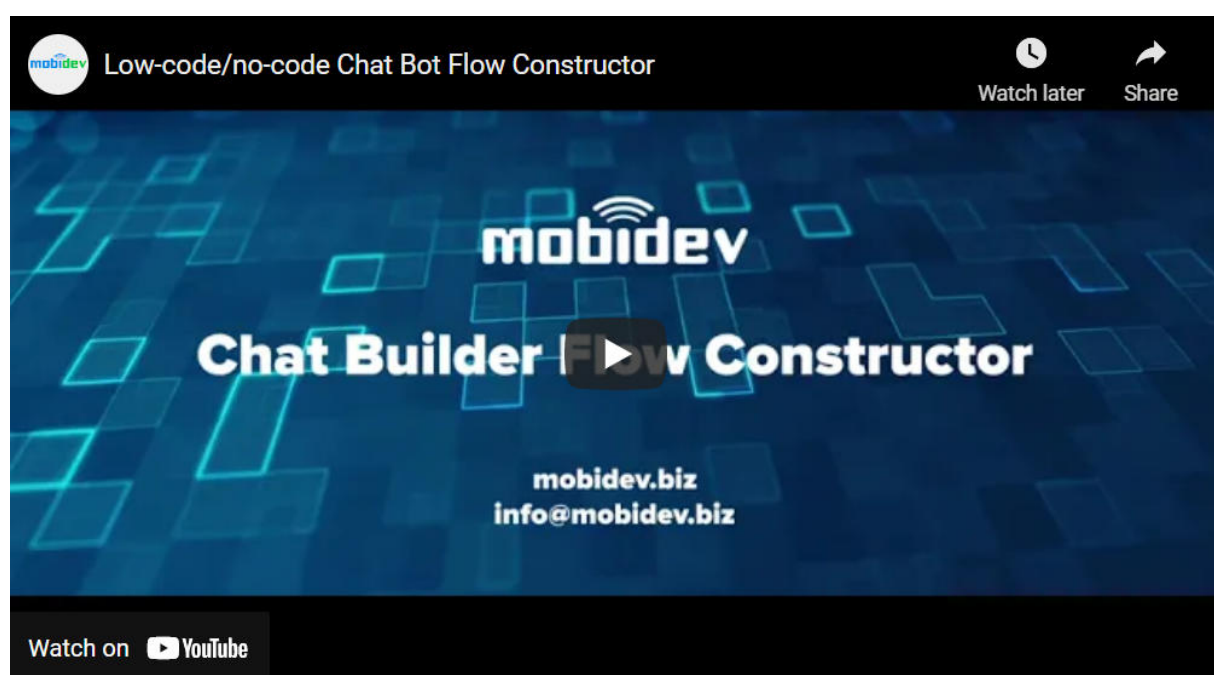
As an intermediate data format, JSON can be replaced by any other format. Although it provides some advantages, which will be discussed in more detail below.

The editor uses an intermediate data model, so it won't have restrictions on the UI part, and a full-fledged drag and drop functionality of various UI components can be implemented. The main task of the editor's UI part is to transform the UI into the intermediate data model as accurately as possible.

The conversion from the data model into the code is implemented with the help of a converter or a compiler. This part causes the main difficulties since it requires not only a high level of expertise in technologies on which the editor is built but also a high level of expertise in those areas which the code is compiled in. Unfortunately, we cannot give tips and recommendations on this part since it all depends on the project and the target technology.

However, here are some items to be noted:

- **This approach applies to the UI interface and use cases where the logic should be described.** In these use cases, the editor flow is being created, and later it forms the basis of the application. In this way, we can compose the logic of not only applications that have a UI, but also chatbot builders or automation platforms.



- **There are three approaches to the compilation:**

1. If it is necessary to build a logical flow. For example, as chat builders, an exchange protocol between logical nodes is developed. And during the execution of the logic according to the flow structure from JSON, logical nodes are being launched in the default order. Logical nodes are isolated and the execution of logic in them does not cause third-party effects in the form of the state of these nodes, therefore this system is quite simply designed and works reliably. [Flow-based programming](#) differs from other paradigms and concepts, including object-oriented programming.
  2. While using this approach to the UI part, templates of specific blocks with the ability to insert child components from similar templates are being applied.
  3. The third approach that's being rarely used is similar to [SAX](#) (Simple API for XML) parsers. It is comparable with the previous one, yet it is easier to take into account the context of code generation. It is complex yet flexible.
- **This approach is closer to waterfall in development** since before starting development it is necessary to analyze and describe in advance all the possible attributes of those UI blocks that will be used. This limitation can be bypassed if you use a similar approach with meta information which stores non-standard data or attributes.

As we've already mentioned, JSON is not required, although JSON and XML are convenient in that they can complete serialization or deserialization of UI code, regardless of the technology for which the final code is generated. Moreover, it is possible to dynamically create interfaces on the fly and, possibly, save traffic since, for example, JSON is much more economical than the same HTML layout with styles.

In this case, it is also possible to import an external code, but it requires the introduction of an additional export format. Such uniqueness of each platform isn't as convenient, though it expands the audience and contributes to the platforms' popularity.

This approach is the most optimal for the implementation of the logic flow for the server-side, as was already described in the compilation approaches. A properly designed server side should always be stateless, and this is what makes it easier for us to implement business logic as a sequence of independent nodes. And this is not something unique because like the Flow development paradigm,



conceptually similar approaches are used in RX and backend frameworks like express/koa, and so on.

## **Editor -> Intermediate Components -> Code**

The third approach to implementing the architecture (Editor -> intermediate components -> code), at first glance, does not differ much from the previous one. There are also UI blocks or components, and there is also a code that is responsible for their location on the screen. However, the essential difference is that the code responsible for these components is not generated, being that it is a part of the library that's connected to the application and is working as an external plug-in resource of the program.

The constructor assembles the product's UI from pre-prepared components of a proprietary library (library of this service). That's why this part of the most complex logic, namely compilation to code, is practically ignored. Such a solution is much simpler – you just need to create a library of components that can be used for the specified target technology and create an editor that works with the primitives (components). Still, this leads to a huge number of additional subtleties, and choosing which of the approaches will be better must rely on the business requirements.

The component code in the library implements the functionality of the component itself, and the editor comprises an additional configuration file so that the editor may understand what manipulations are allowed with this component and what parameters it corresponds to in the components of the library.

The following features of this approach can be noted:

- Importing external code will be the most difficult since you have to specify a common component format that covers the maximum functionality.
- Newer (more extended) versions of the real-time component libraries may not support older ones, which may lead to support issues.
- Faster time-to-market since individual components are easy to test with further expansion of the functionality. In fact, such a solution will still be more limited and rely on the capacity of the team supporting this platform.

- Higher reliability
- A small/large performance drawdown (depends on the target platform and the quality of the real-time code of the component library). The components are not compiled, but work in real-time, as a wrapper library for native ones, using plug-in libraries.

For example, FlutterFlow uses this approach.

We have investigated three main approaches to implementing the main part of most no-code and low-code platforms. This will help you in choosing the right development direction.



**Read also:**

[Web Application Architecture in 2022: Moving in the Right Direction](#)

## What It Takes to Build Your Own No-code / Low Code Platform

A couple of additional points that can play a decisive role in the development of your particular product are also worth considering.

The first one is why importing external code, templates or snippets can play a decisive role in your business related to no-code or low-code platforms.

Everything is pretty basic: such platforms are used mostly at the POC or [MVP](#) stages to test a business idea or enter the market as quickly as possible. It is also used for building niche solutions, if the use case is straightforward and there is no need for extra features.

In the subsequent stages, a crucial role is played by a limited set of components or functionality (ready-made modules), which prevents implementing any unique features or gaining a competitive advantage.

The huge cost of supporting backend logic is also important, though it may be minimized in the case of the proper design. To resolve a problem with a limited set of functionality, it's possible to import an external code – the npm model with closed and open components serves as an example. Such a low-code platform will be supported by the community, introducing positive changes in the business.

Unfortunately, if we talk about the implementation time of such platforms, then it significantly exceeds standard projects such as social networks or ERP systems. This must be taken into account in the planning stage.

Finally, a small idea as a bonus to the article: you can use a text description to generate the scaffold of your application. For example, [Natural Language Processing \(NLP\)](#), to parse the text into entities objective and subjective, form the structure of your application, and then generate the code. This won't be an ideal solution, and with the current state of technology it's still too early to talk about this approach for no-code platforms, yet as a key feature of your new low-code platform, it may be quite valid in the near future.

If you are interested in other implementation aspects of now-code/low-code systems, feel free to contact our company and discuss all the details.



**CONTACT US**  
**[info@mobidev.biz](mailto:info@mobidev.biz)**

**[mobidev.biz](http://mobidev.biz)**