The Complete Guide to Creating Components

Introduction

Components are one of the most important aspects of low-code platforms, which enable reusability of code and repositories of component libraries. As a result, developers can build high-quality app components and share them with others, boosting application development time and code fragmentation.

This article includes a companion checklist that you can print and refer to at any time.

What is a Component?

A component is a reusable object that speeds up application creation and delivery. It can be either an application or a module that provides additional features. It should be easy to use and understand, and the focus should be main or common use cases.

After you create a component, there are three possible outcomes:
• It can **modularize** your code, to be reused within your application (Don't Repeat Yourself - [DRY](https://success.outsystems.com/Documentation/Best_Practices/Development/The_Complete_Guide_to_Creating_Components)).

• It can act as an infrastructure piece for your **factory**, to be reused in more than one application.

• It can be **published** to the Forge, where it will serve several apps and developers.

The successful creation of a component encompasses:

• Deciding its **purpose**.

• Designing an **API** and **logic** that support the main scenarios.

• Defining the **presentation** by giving it the right names and icons, for example.

• Following a set of best practices that enhance the **experience of reuse and maintenance**.

• Making sure you cover **non-functional requirements**, such as performance, security, and scalability.

• Planning the **release** and let your component see the light of day.

Keep in mind that your components will fit into the [architecture of the 4 Layer Canvas](https://success.outsystems.com/Documentation/Best_Practices/Development/The_Complete_Guide_to_Creating_Components) methodology used to build OutSystems applications.

## Component Types

Components can be of five types:


• **Widgets/ Blocks**: Widgets or blocks are independent components that provide **additional and reusable features**, usually related to UI or events, for example, the [Cropper](https://success.outsystems.com/Documentation/Best_Practices/Development/The_Complete_Guide_to_Creating_Components) widget. To become components, blocks need a set of definitions, such as parameters, and variables. See the complete properties of a block.

• **Libraries**: A JavaScript library lets you develop faster. See as an example the [Google Maps](https://success.outsystems.com/Documentation/Best_Practices/Development/The_Complete_Guide_to_Creating_Components) library. These libraries are wrapped around blocks and actions that provide an **easy-to-interact interface**. See the complete properties of JavaScript.

• **Connectors**: Connectors allow **integrations** without the need to write custom code, significantly reducing time and effort and eliminating errors. Broadly speaking, connectors allow the integration
with other applications, for example, Facebook, or Paypal.

- **Plugins**: A plugin is an application module acting as a wrapper for an Apache Cordova plugin. Depending on the Apache Cordova used in the plugin, it can support the use of the native mobile capabilities for iOS and Android or only for one of them. See the list of supported plugins provided by OutSystems.

### Creating a Component: Thinking it

Before you start building it, you need to think about your component. What do you want from your component? What **options** will it have? What should be the **default behavior** out of the box for a quick use? What are the most **important use cases**, and how will your component offer them upfront?

A **component is usually a set of blocks and actions** that needs to be distributed as a module and/or application. For example, if you want to modularize your own code, most likely your component will be a new block. However, if you’re planning on feeding your factory or have a public release of your component, it will be inside an application, as a module.

If you’re creating a component with only logic, consider creating an empty module, so it doesn’t have any UI dependencies. This will make your component light and simpler to reuse, without additional dependencies.

On the other hand, if your component will have UI elements, start by creating a module with a template. This will provide you with a structure optimized for that purpose.

Consider your scenarios to build for success from the start.

### Creating a Component: Building it

Upon defining the purpose of your component, you can start building it, by defining any relevant UI elements, actions, and logic.
API Design to Offer the Best Options

There is no point in owning a great car if you don't know how to ride it. The same applies to a component. During implementation, keep in mind the developer experience and their skill set. As such:

- The API should be small and simple to understand. For example, instead of offering a 360-degree input for alignment, simply offer top, bottom, right and left.

- The API should focus on the main use cases and offer the best defaults. Other options can come through an extensibility mechanism.

- Consider limiting the number of inputs only to the ones strictly necessary for the component's default behavior.

One trick to expand your component configuration options without adding many inputs is to add only one more that receives all extended options as JSON. Make sure you explain the available JSON options on both the input and component descriptions.

If you have a lot of advanced scenarios, consider having a page dedicated to explaining those extensible options. In case you’re publishing your component to the Forge, this can feed the Try Now option.

CSS and JavaScript

**Style Classes to Simplify Customization**

Make sure you take advantage of expressions in the Style Classes property. This will generate code that changes the "class" attribute instead of using the "style" one, preventing unnecessary data on your DOM and simplifying customization. Keep in mind that everything that is static but conditional can be a CSS class, so use the inline style attribute only when you have a dynamic style. You can also consider leaving a comment explaining the math behind it.
CSS Selectors for Enhanced Reusability

By only using classes in your CSS selectors, the reusability of your component will be limited because you will only be able to reference one instance of it, limiting its multiple usages on the same screen. Follow the steps below to create CSS selectors in relation to a specific instance of the component, instead of using only classes:

1. Add a wrapper div and set the "Name" property.

2. In the action, where you will have the JavaScript node with the selector, add an input parameter and set it as the MyBlockWrapper element ID.

3. Build your selector with a CSS class, related to the ID you passed as an input. In this example, the element where the CSS class exists is a child of the Wrapper element, but it could even be the same element.

The `selected` class is static CSS conditionally applied.

```css
Style Classes

If(IsSelected,"selected",""")

Attributes

style

= "transform:scale(" + MyList.Current.Size * Multiplier + ");"

This `style` itself is dynamic, generated at runtime.

Updated: Wed, 22 Apr 2020 23:34:15 GMT
Balancing Limits and Customization

When writing your code, especially CSS and JavaScript, be careful not to limit the developer. Keep in mind the possibility of changing the UI and UX elements of your component.

- **Avoid the "!important" CSS tag.** Even though this could be the right choice in some cases, you might not want to use it on a component because it will override the developer's custom CSS, therefore, limiting customization.

- **Avoid setting colors and buttons** as well, as they aren't easily overridden. If you have UI elements to trigger actions, like buttons, consider replacing them with placeholders, so that developers can add their own UI elements and decide what actions or logic to execute.

Creating for Extensibility

When designing the component's API, consider if it provides enough mechanisms for the developer to extend its functionality. For example, a component that created JavaScript Objects should have a function that provides access to them. This way, the developer can add features, creating a component that wraps another.
It is very important to understand the lifecycle of screens and blocks while developing a component in order to be able to control its data and behavior. Either to define default values for variables, or delete data that becomes irrelevant. Having complete knowledge of the lifecycle behavior allows you to predict issues and optimize performance.

Check the documentation for Screen and Block Lifecycle Events. Below, we'll detail the events more relevant for components.

**OnInitialize to Define Variables' Default Values**

Imagine that you have a screen with a local boolean variable with the default value *False*. After the screen logic is executed, the values change to *True*, and you navigate to another screen. In runtime, variables' values (defined as model) are saved, so that we can restore it when there's a Back Navigation also known as *Previous Screen Navigation*.

After this navigation, the variable will be *True*, but you might want to reset the screen state. So when you're using variables with default values always consider these navigation scenarios and if needed, assign the default values on the OnInitialize event handler.
OnDestroy to Prevent Memory Leaks

Some components create global variables, event listeners, or other data during their usage. Consider removing/deleting that data on the **OnDestroy** event.

OnRender for Rendering and Data Changes

Note that if your component includes UI, the **OnRender** event is also important, because it runs each time the Screen or Block is rendered. For example, it runs whenever there is a data change on a screen.

Creating a Component: Presenting it

**Reusability** and **simplicity** are key when creating a component. The component should be easy to understand, and it should focus on the main use cases. **Names, descriptions** and **visual cues**, such as **icons**, become fundamental to easily grasp the purpose of the component, and quickly know how to use it.

Names and Descriptions

**Names and descriptions** help people understand what is the goal of your component and how to use it. Names should be clear and clarify what the component is about. The public-facing parts of the component should have comprehensive, front-loaded and succinct descriptions.

To have a complete overview of the component, your description should include things such as: libraries URL, API version, and references to external resources.

Keep in mind these naming conventions in particular:

- Use **meaningful names** (for example, "Cropper", instead of "Cppr").
- Use **PascalCase** (for example, "FirebaseReceiver", instead of "Firebasereceiver").
- Use **event names that start with "On"** (for example, "OnReady", instead of "Ready").

Descriptions provide visual cues during development, enhancing that experience. For example:

- Hover over application dependencies to see their description.
- Check a module's description, by hovering in the Manage Dependencies popup window.

Updated: Wed, 22 Apr 2020 23:34:15 GMT
• The **search box** allows you to search for the name and the description. Note that only components with an icon will appear in the toolbox, on the left.


**Icons for Visual Reference**

Visual programming environments have **graphical or iconic elements** to be used in an interactive way. Icons are important because developers will actually see them in the development environment.

The first thing to keep in mind is that you should **use images that identify the component**. For example, the [Firebase Connector](https://success.outsystems.com/Documentation/Best_Practices/Development/The_Complete_Guide_to_Creating_Components) has, as an icon, the Firebase company logo.

The second important thing to consider is that you must **use the same image from the Application level all the way to the Actions level**. This will provide consistency and allow developers to easily identify all your component pieces.

**Application Icons**

Setting an application icon defines the **main icon of your component**. Users will relate everything that has this icon to the component. The application icon will be displayed throughout the component and its references. For example, it could even be used in the component's documentation.

Application icons should be in `.png` format to allow [transparencies](https://success.outsystems.com/Documentation/Best_Practices/Development/The_Complete_Guide_to_Creating_Components) and their size should be 1024 x 1024 px. Keep in mind the file size, as big files will slow down screen loading.
Module Icons

Setting icons for the modules allows developers to relate the modules to your application in the Manage Dependencies window.

The size of module icons should be 1024 x 1024 px.

---

Block Icons

Settings icons for blocks helps to easily identify to which component a UI element belongs to. So when you search for elements, you will be able to immediately see the blocks related to your component.

The size of block icons (.ico format) should be 32 x 32 px.
**Action Icons**

Setting image icons for actions helps to easily identify to which component an action belongs to in a flow.

![Diagram of Action Icons]

**Client and Server Action Icons**

Service Studio makes a visual distinction between client actions and server actions. This visual cue helps users understand not only the context of the action but also the relevant scope.

For example, if you're working on a client side form for an offline application you'll want to use client side validations and avoid server actions. Your component should follow this distinction if possible and applicable.
If you're implementing both client and server actions, consider using different styles of the same icon. OutSystems already provides you with the default icons for client and server actions. If you want to change them, their size should be 32 x 32 px.

Enhancing the Developer Experience

Configuring a Good Preview

For a better development experience, ensuring a great preview inside Service Studio is key. Often, there are UI elements that have certain styles that we want to avoid when developing applications. For this, there are a set of preview improvements that can be done.

**False Conditions for Better Previews**

When using an If widget with *False* as condition, developers are presented with a **friendly** UI that allows them to **hide placeholders** that occupy a large amount of UI space, for example, a side panel, or a bar.

You can also use it to display an image that replaces a runtime UI or adds UI to a component that doesn't have one, for example, a chart that is generated by JavaScript or a plugin block that propagates events.

Take a look at this example of the Silk Ui Mobile FloatingActions pattern. Its purpose is to provide a clickable circle that expands to provide extra options:
Service Studio CSS Tags for a Better Development Experience

CSS properties prefixed with `-servicestudio-` can be used to improve the preview and consequently the development experience. This allows you to adapt the layout during development time. For example, when you have a placeholder with a position "absolute" that you want to keep static, so the developer can add content. Since the editor renders all CSS instructions, the elements might become inaccessible in the editor.

Using `-servicestudio-` tags helps developers set the values without compromising other features:


Updated: Wed, 22 Apr 2020 23:34:15 GMT
Sample Content inside Placeholders

A placeholder should have sample content if it is directly connected to a behavior or structure. This way, the developer will know what the component is expecting in that placeholder. For example, both Silk UI Search and AnimatedLabel blocks have an input as sample because it is required for the block to work properly.

You should avoid using sample content for simple things like texts and images because it will only overwhelm and force the developer to delete them. Instead, add sample content only if it is mandatory or if it helps the developer.
Maintaining a Component

You should keep your code tidy and organized. This is important for when you come back to it months later, and to make it easier for other developers to reuse it. When published in the Forge, it will be imperative that different users understand and work with the same code.

Comment changes to code between versions, so people can better understand what changed and why.
Notes and Comments

If you have logic-related code in your component that is not clear, then you should add comments or notes explaining its purpose. This is essential to make sure other developers can follow and improve the code. However, even if it’s not that complex, you should always add a simple label, to make the code more understandable.

Have a look at this example, of JavaScript nodes, or complex assigns:

Because different people have different mindsets, you should use the Expression Editor Comments. Leave comments to explain complex code but also to sum it up. Consider how useful this is for a developer with limited knowledge of JavaScript, but still wants to understand the feature behind a JavaScript Node.

```javascript
IsNotSystemMessage Condition

//If this is not a system message we want to show the sender name
```

Non-Functional Requirements
Scalability

Keep in mind that you should build thinking of the future. So make sure that your component will be able to support **multiple blocks on the same screen**.

Consider **concurrency scenarios**. OutSystems already helps you with that in the background, keeping each block separated, but you will still need to be aware of actions triggered by the users. For example, several fast clicks on a button, triggering multiple calls to the server.

If the **component fetches data**, consider allowing the developer to **fetch it outside of the component** (for example, a screen), and provide that data (to the component) as an input parameter, instead of having the component fetch the data.

When using **external resources** (for example, external libraries, or JSON files), consider reusing them instead of fetching them again. For example, the System's RequireScript action will use the cached file if you already requested it.

Read the complete description on how OutSystems handles scalability, in a way that's adjustable to specific requirements.

Security

When it comes to security, you need to consider that any device can be compromised. You should, therefore, focus on keeping your **data safe**.

Since everything you have on the **client side can be accessed and is more vulnerable**, make sure you **protect any sensitive data**. You can use this [plugin](https://success.outsystems.com/Documentation/Best_Practices/Development/The_Complete_Guide_to_Creating_Components) to cipher your data.

For any server-side communication, you must ensure its safety by implementing **server-side validations**, especially permission roles.


Read the complete description on how OutSystems handles security with a set of pre-built features, to ensure the safety of your data during the entire lifecycle of mobile and web apps.
Performance

OutSystems ensures that there are no performance bottlenecks with design-time validation. When it generates the source code for applications, OutSystems optimizes code performance at all application layers.

Still, you'll do well in testing your component against more than just a few sample records. Set up a scenario as close as possible to the real one, and test your component against probable conditions.

Go the extra mile and consider peaking the number of records or conditions, or test scenarios with thousands of concurrent users. If your component is reaching the server, there might be performance impacts.

Publishing a Component

If you think your component can help other developers, consider publishing it.

The Forge is a repository of components, libraries, and connectors that speeds delivery by providing existing modules that can be reused in applications.

As regards to licensing, components in Forge are distributed under BSD license, but some components or plugins have different licensing models, that can have different requirements, like attribution in the app. Check compliance with third-party licenses.

See how you can use a component made by the community.

Demo Module

To prepare your component for publication, add one new module to the application, with a demonstration of the most common use case. Its name should include the component's name and "Demo", or "Example", since users will search for the component's name when they need help.

This demo should be guided and straightforward, with one single or central screen if possible, and without login.
Once your component is ready, follow the steps below to prepare and publish your component to the Forge:

1. Manage dependencies to remove all unused elements and ensure the final Module (.oml) is not too big. If it is, consider compressing any used image files or resources.

2. Go to Service Center and download your component. Your component can be downloaded as an Application (.osp) or as a Module/eSpace (.oml). If possible, you should download it as an Application, but the Forge will support the Module format as well.

3. Log into the Forge and Click Publish New Project.

4. Select the component's file you downloaded in step 1. If applicable, the Forge will warn you of any dependencies.

5. Continue to edit your component's Details. By default, the name, icon, and descriptions are the ones defined during the development of your component. These descriptions can be changed even after publication.

6. Add images that best illustrate your component's features and the main use cases and behavior. You should select screenshots of the application, with the size 500/600 x 280 px.

7. In the Preview URL, enter the URL that will feed the Try Now option. This URL should link to either sample, demo, or tutorial screens, that can also be made available inside the downloadable module. Your link should be hosted in public server, such as a personal environment.

8. Select at least one Category that best matches what your component is. This categorization will help users to find your component.

9. Enter the Project Version. Keep in mind that numbers are unique. The Forge will suggest one, but you can change it. Since this number will be visible in the versions of your component, you should assign incremental numbers to each new version update, so people easily know which is the most recent version.

10. The Stable checkbox exists to allow unfinished or untested projects. However, if your component is finished, with all features completely implemented and fully tested, and with the best performance possible, then you should let other users know and select the Stable checkbox.

11. Under Requirements, the Forge suggests a Platform Version. You can also select the Supported Database and Supported Stack. This will help users know if they meet all the conditions to download and use your component.
12. Add any relevant Tags. Tags help users find your component, and they also allow to filter published components.

13. Enter the Detailed Description of your component. Make sure it fully describes your component's features and options. Consider following the best practices described above: your descriptions should be clear and explain use cases and behaviors.

14. Click Publish.

Keep it Relevant

Following these guidelines will ensure your success, so many users might start using your component. As the owner of the component you are responsible for it, and you will probably receive queries on its functionality or implementation. Do your best to be helpful. If you don't have time to respond or iterate on the component, consider inviting its users to join the team.

Every once in a while, a new version of OutSystems is released. These released often bring new features. Keep your component fresh by upgrading to new platform releases and making use of the new capabilities.