Backstage Way of Working: Creating Templates for Software and Building Blocks

Introduction

Backstage provides a powerful developer portal designed to standardize and automate how teams create, manage, and deploy software components. Within the context of the Building Blocks project, Backstage serves as the central tool for defining reusable, configurable templates that allow architects and developers to rapidly spin up new services, microservices, authentication portals, or other reusable building blocks.

This document outlines a recommended way of working for transforming any piece of software—ranging from simple components to full microservices—into a reusable Backstage software template. This guidance ensures consistency, scalability, and ease of maintenance across all building blocks.

Template Limitations in Backstage

Although Backstage is highly flexible, there are several architectural and practical limitations when creating templates:

1. Templating is File-Based

Backstage Scaffolder templates operate by rendering files using values provided by the user. This means that:

- It cannot dynamically generate logic that depends on runtime environments.
- Templates cannot conditionally include or exclude files unless explicitly implemented using templating expressions.

2. No Complex Execution Logic

Templates can run Scaffolder actions (e.g., fetch:template, publish:github, catalog:register), but they cannot perform:

- Heavy computation,
- Long-running scripts,
- Multi-stage CI/CD logic.

Such operations must instead be part of the generated repository itself.

3. UI Form Limitations

Parameters exposed to the user must follow JSON Schema formats:

- No nested dynamic fields,
- No computed defaults,
- No conditional form layout.

4. Repository-Centric Workflows

Templates assume that the end result is a repository. They do not directly deploy software unless deployment automation is baked into the generated project.

How the Templating Can Be Done

Below is a simplified workflow for transforming a piece of software into a Backstage template. For concrete examples, the structure references the uploaded Keycloak template (filecite turn1file0).

1. Identify What Should Be Parameterized

Determine which parts of the software should be configurable:

- Environment variables
- Deployment configuration (Docker Compose, Helm chart)
- Authentication values
- Database options
- Repository metadata

These will become the parameters section in your Backstage template.

2. Prepare a Base Template Repository

This repository contains:

- The raw code of the building block (microservice, library, Keycloak setup, etc.)
- Files that will be templated using ${{ values.* }} syntax
- Any scripts or Docker files required for deployment

The Backstage template will **fetch** this repo using the Scaffolder's fetch:template action.

3. Create the Template Definition (template.yaml)

A template follows this general structure:

- Metadata (name, title, description)
- Parameters
- Steps
- Output links

Example sections from the uploaded template:

- The **parameters** define realm, roles, redirect URIs, GitHub repo info.
- The **steps** include fetching the base template, publishing it, and registering it.

4. Add Template Expressions

Anywhere you want to inject user input, use:

${{ parameters.<name> }}

Examples include:

- Docker Compose files
- Keycloak config files
- README templates
- CI/CD pipeline files

5. Test the Template

Steps for validation:

- Verify that required fields are enforced
- Check that all template expressions render correctly
- Ensure GitHub publish step works
- Confirm the generated project builds & deploys

6. Register the Template in Backstage

The template must be added to Backstage via:

- The catalog-info.yaml for templates

- The Backstage app's app-config.yaml under catalog.locations

This ensures the template becomes visible in the Backstage UI.

Troubleshooting Syntax in Templating

Templating issues are common, especially with YAML and Scaffolder expressions. Below are frequent issues and how to fix them.

1. Expression Parsing Errors

Backstage uses Nunjucks-style syntax. Errors occur when:

- Indentation is wrong
- Quotes are missing
- Arrays are nested incorrectly

Fix: Validate each expression individually and ensure YAML indentation follows strict spacing.

2. Unparsed Variables in Output Files

If ${{ parameters.x }} appears verbatim in the generated repo, it means Backstage didn't process that file.

Fix: Ensure the file is inside the fetch:template processing scope.

3. Incorrect Step References

Referencing outputs incorrectly—for example:

${{ steps.publish.output.repoContentsUrl }}

must match the action's actual outputs.

Check the action documentation or inspect error logs.

4. Missing Required Parameters

When a parameter is marked as required but missing, Backstage fails before rendering.

Fix: Always set default values or ensure form fields are visible to the user.

5. Wrong Data Types

Backstage is strict about parameter types:

- Arrays must be explicitly typed
- Objects must follow schema

Check JSON schema compatibility.

Conclusion

Backstage templates provide a powerful, standardized way to automatically scaffold software, building blocks, and infrastructure components. By following a consistent workflow—identifying parameters, preparing a base repository, defining template.yaml, testing thoroughly, and documenting the process—teams can drastically improve their development speed and reliability.

This document should serve as both a methodology guide and a practical reference for creating new Backstage templates that support the Building Blocks initiative at Fontys. Additional research (projectdescription.txt, filecite turn1file1) indicates the need for maintainability and long-term consistency, which this workflow directly supports.