# Workflow

In Spotify Backstage, a **workflow** during component creation is a series of automated steps orchestrated by the **Scaffolder plugin** to generate and configure a new component. Each workflow defines the actions that need to be completed to move from an initial request to a fully functional and cataloged component.

**Key Elements of a Workflow in Backstage**

1. **Template-Based Definition**:
   * Each workflow is tied to a **template** (a YAML file) that specifies the series of actions or steps required to create the component.
   * Templates define parameters (like name, owner, type) and instructions for the creation process, using a standardized format that allows Backstage to create repeatable workflows.
2. **Stages of a Workflow**: A workflow consists of distinct stages that execute one after another, often including tasks like initializing a repository, setting up metadata, configuring CI/CD pipelines, and provisioning cloud resources.
3. **Steps in a Workflow**:
   * **Input Collection**: The workflow starts with collecting user-provided inputs based on the template requirements.
   * **Scaffolder Actions**: The workflow proceeds with automated actions, executed by the Scaffolder plugin. Common actions include:
     + **Repository Creation**: Initializes a repository and pushes initial files.
     + **File and Directory Setup**: Creates and organizes directories, configuration files, and starter code.
     + **Metadata and Catalog Registration**: Generates and registers the catalog-info.yaml file for the Service Catalog.
     + **CI/CD Setup**: Configures CI/CD pipelines, if specified, for automated building, testing, and deployment.
     + **Infrastructure Provisioning**: Provisions cloud resources if the template requires an infrastructure component, such as compute instances or databases.
4. **Workflow Execution Control**:
   * **Sequential Execution**: Each step in a workflow is executed sequentially. If a step fails, the workflow is typically halted, and the error is reported to the user.
   * **Logging and Feedback**: The workflow provides real-time feedback in the Backstage UI, showing the user the status of each step. Logs may include success/failure notifications, helping users debug issues if a workflow fails.
5. **Configuration and Customization**:
   * Organizations can create custom workflows by defining new templates, allowing them to tailor workflows to their own standards and tooling. For instance, a workflow could be customized to automatically apply specific security configurations or connect to proprietary systems.
6. **Post-Workflow Actions**:
   * Upon successful completion, the workflow links all relevant resources (like the repository, catalog entry, documentation, CI/CD pipelines) to the component’s Backstage page. This unifies everything needed to manage the component from its lifecycle initiation.

Example Workflow Definition:

apiVersion: backstage.io/v1alpha1

kind: Template

metadata:

name: service-template

title: Service Creation Template

spec:

steps:

- id: create-repo

name: Create Git Repository

action: backstage:github:repo:create

input:

repo: ${{ parameters.name }}

description: ${{ parameters.description }}

- id: setup-files

name: Set up Files

action: scaffolder:files:generate

input:

templatePath: ./templates/service

targetPath: ./${{ parameters.name }}

# Component Life Cycle

1. **Initiation**:

* **User Action**: A user clicks on "Create New Component" in the Backstage UI.
* **Template Selection**: The user selects a template for the component (e.g., microservice, library, documentation site) and provides initial configuration details, such as the component name, owner, and description.

1. **Input Validation**:

* **Validation Check**: The backend validates the user input to ensure it meets the template requirements (e.g., name uniqueness, owner assignment).
* **Error Handling**: If input doesn’t meet validation criteria, the user is prompted to correct it before proceeding.

1. **Template Processing**:

* **Scaffolder Plugin Execution**: The Scaffolder plugin initiates the component creation process by processing the selected template.
* **Variable Substitution**: The scaffolder plugin fills in placeholders in the template with user-provided values.

1. **Repository Creation**:

* **Source Control Integration**: The Scaffolder plugin connects with a source control provider (like GitHub, GitLab, or Bitbucket).
* **Repository Initialization**: A new repository is created, with an initial codebase and configuration files populated based on the selected template.
* **Push Initial Code**: The initial commit is pushed to the repository, establishing it as the starting point for the component.

1. **Catalog Registration**:

* **Catalog Metadata Creation**: The scaffolder generates a catalog-info.yaml file for the component. This file contains metadata such as name, type, owner, and tags.
* **Entity Registration**: Backstage’s backend registers the new component in the Service Catalog using the catalog-info.yaml, making the component visible in the catalog.

1. **CI/CD and Infrastructure Setup (Optional)**:

* **CI/CD Configuration**: If the template specifies, the Scaffolder may create CI/CD pipelines for build and deployment in tools like GitHub Actions, Jenkins, or CircleCI.
* **Infrastructure Provisioning**: If the template includes cloud resources, infrastructure is provisioned via Infrastructure as Code (IaC) tools like Terraform or ARM templates to support the new component’s runtime environment.

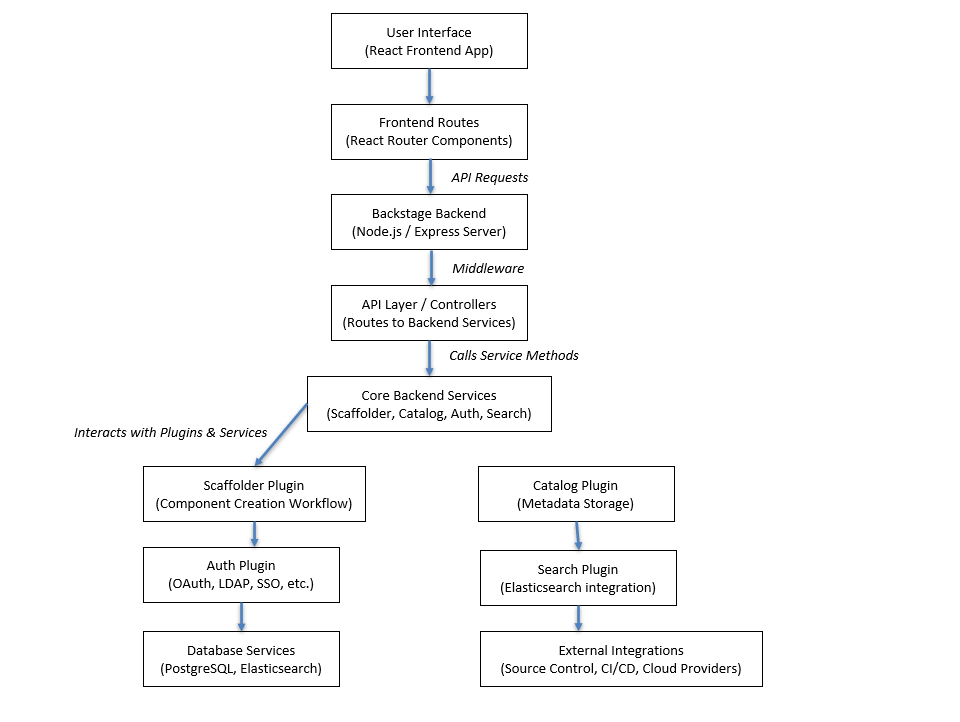
1. **Notification and Linking**:

* **Completion Notification**: The user receives a notification indicating successful component creation, with links to the repository, catalog entry, and any CI/CD pipelines.
* **Post-creation Linking**: Additional notifications or links to related documentation, Slack channels, or monitoring dashboards may be automatically created.

1. **Post-Creation Activities**:

* **Initial Build and Deployment**: CI/CD pipelines (if set up) trigger initial builds, tests, and deployments for the component.
* **Monitoring and Logging**: Relevant monitoring and logging integrations are configured to track the component's performance and status.
* **Maintenance and Updates**: The component’s metadata or structure can be updated as it evolves, and entity providers or catalog tools continuously sync updates.

# Backstage Source Code Flow Diagram (High Level)



**1. User Interface (React Frontend App)**

* **Role**: Backstage's UI is built with React and served from the backend. It serves as the entry point where users interact with Backstage’s core features.
* **Code**: Located under packages/app, the frontend includes components for navigating Backstage features (e.g., catalog browsing, component creation).

**2. Frontend Routes (React Router Components)**

* **Role**: Routes handle navigation and page rendering within the app, routing users to components based on URL paths.
* **Code**: Uses React Router, typically found in App.tsx or routes.ts files. These routes map URLs to individual components, which in turn make API requests to the backend.

**3. API Requests**

* **Role**: Frontend components make REST or GraphQL requests to the backend for data.
* **Code**: The frontend uses the @backstage/core or @backstage/plugin- libraries to interact with backend APIs. Requests are routed to the backend via HTTP calls using functions from @backstage/core-plugin-api.

**4. Backstage Backend (Node.js / Express Server)**

* **Role**: The backend server, built with Node.js and Express, manages all API requests from the frontend. It handles routing, authentication, and serves as the primary integration point for plugins.
* **Code**: Located under packages/backend, the server code configures routes and middleware, and orchestrates plugin usage.

**5. API Layer / Controllers**

* **Role**: The backend’s API layer (controllers) routes each request to the appropriate backend services or plugins.
* **Code**: Controllers are defined in packages/backend/src/plugins or routes. They include request-handling logic, authentication, and routing to services like catalog, auth, and scaffolder.

**6. Core Backend Services**

* **Role**: Core services represent the essential features of Backstage (e.g., scaffolder, catalog, auth). Each service has dedicated code to manage its functionality and interacts with data stores and plugins.
* **Code**: Core services have dedicated directories (e.g., packages/backend/src/plugins/scaffolder). Each service includes its data models, controllers, and business logic.

**7. Key Plugins (Scaffolder, Catalog, Auth, Search)**

* **Role**: Plugins extend Backstage functionality, with each plugin responsible for a specific area. For instance:
  + **Scaffolder Plugin**: Manages workflows for component creation and integrates with source control APIs to create repositories and CI/CD setups.
  + **Catalog Plugin**: Manages the Service Catalog, storing metadata about components.
  + **Auth Plugin**: Manages authentication and authorization (e.g., OAuth, SAML, LDAP).
  + **Search Plugin**: Provides search functionality, often using Elasticsearch.
* **Code**: Located in plugins/. Each plugin contains backend code for the service and frontend code to render related UI components.

**8. Database Services (PostgreSQL, Elasticsearch)**

* **Role**: Stores data needed by core plugins. PostgreSQL often holds catalog data, while Elasticsearch supports full-text search capabilities.
* **Code**: Database models and migration scripts are often defined within each plugin directory, usually in src/database.

**9. External Integrations**

* **Source Control**: Interacts with GitHub, GitLab, or Bitbucket for repository creation.
* **CI/CD**: Integrates with Jenkins, GitHub Actions, or other CI/CD platforms for automated testing and deployment.
* **Cloud Providers**: Optional integration with cloud platforms like AWS, GCP, or Azure, especially for resource provisioning through IaC (Infrastructure as Code).
* **Code**: Integrations are configured within individual plugins, like the scaffolder, which defines API clients and logic to interact with external services.