

Software Artifact Analyzer

Developer Guide

Version 1.0.0 July 2024

1 Introduction	2
1.1 Current System Architecture	2
1.2 Build & Maintenance	3
2. CASE Tool Integration Applications	3
2.1 Setting Up An GitHub Application	4
2.2 Setting Up An Atlassian Jira Application	4
3 Data Collection & Graph Database	5
3.1 Flask Server	5
3.2 UI	5
3.3 Data Retrieval	6
3.4 Data Cleaner	7
3.5 Graph Builder	7
3.6 Creating and Connecting to a Neo4j Database	8
3.6.1 Establishing a New Neo4j Database Instance	8
3.6.2 Preparing the NEO4J Database Instance	9
3.6.3 Connecting NEO4J Database Instance to the System	9
4 Custom Folder	10
4.1 config	11
4.2 customization-service	12
toolbar-customization.service.ts	12
navbar-customization.service.ts	13
cy-style-customization.service.ts	13
group-customization.service.ts	13
context-menu-customization.service.ts	13
4.3 operational-tabs	13
4.4 analyses	14
4.4 Database Connection	17
5 Starting Software Artifact Analyzer	17
5.1 Development Mode	17
5.2 Production Mode	17
6 Custom Procedure	18

1 Introduction

This guide will show you how to develop new features inside the Software Artifact Analyzer tool (SAA), designed to visually analyze your software artifacts and their relationships and improve the software development processes. SAA meticulously analyzes various software artifacts, including source code files, pull requests, issues, and commits, and their interconnected relationships with developers within a project. It uses an advanced drawing canvas with features such as a context menus, visual cues, and more to facilitate this comprehensive analysis. SAA uses a structured approach, and at its core is the open-source toolkit Visuall [1], which provides a solid foundation for its functionalities such as the user-friendly interface for developers to interact with the system. With this interface, developers can explore and make informed decisions by leveraging insights derived from the comprehensive artifact traceability graph. This guide details how a custom software analytics application can be integrated into SAA. We assume the reader has already gone over the [SAA User Guide](#).

As many features are inherited from the base toolkit Visuall [1], you might also want to refer to [Visuall Develop Guide](#) for details on customizations of generic features.

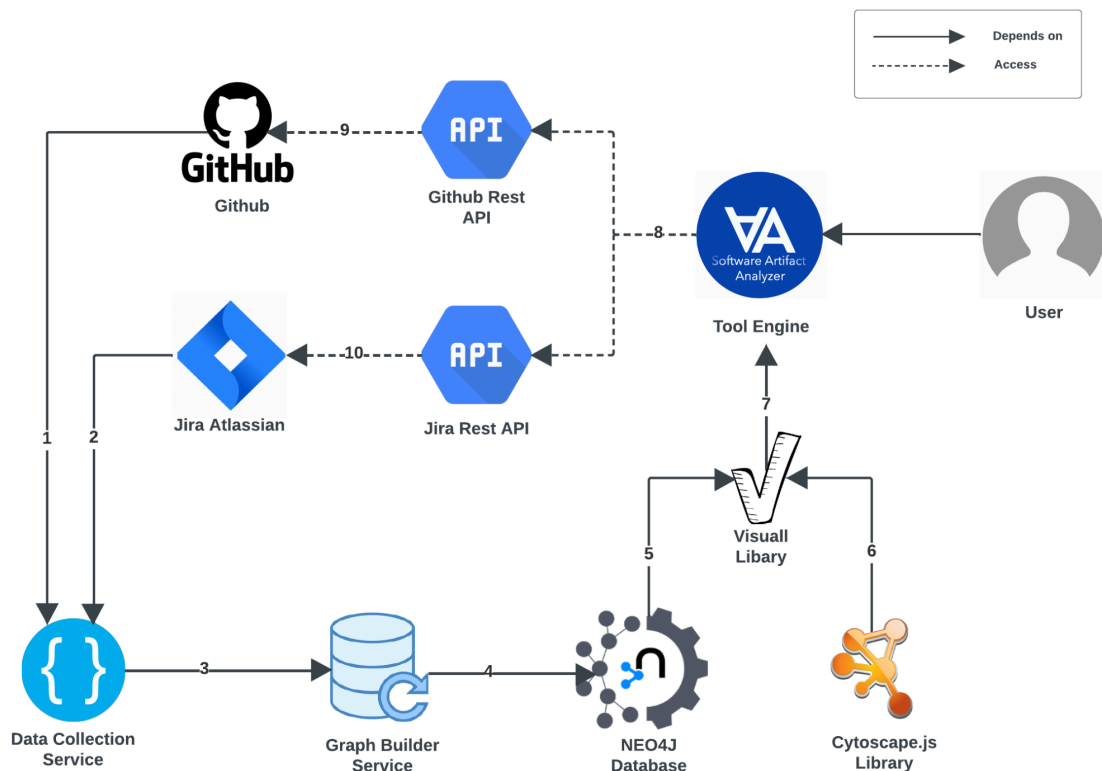


Figure 1 Current system architecture and flow of information

1.1 Current System Architecture

- 1, 2) Data Collection Service retrieves project data from respective backends as a JSON format.
- 3) The metadata is processed and cleaned by the data retrieval service to prepare it for use by the Graph Builder service.
- 4) Based on the prepared graph model, the Graph Builder Service creates and merges nodes and edges in the Neo4j database.
- 5) The database instance is connected to the Visuall [1] library.

- 6) Visuall [1] uses the Cytoscape.js library for web-based graphic and network visualization.
- 7) SAA engine is developed on top of the Visuall [1] toolkit and provides a user interface for developers to interact with the system.
- 8,9,10) The tool engine uses GitHub and Jira APIs, to access Github and Jira platforms to send data to these platforms through these APIs.

1.2 Build & Maintenance

To establish their development environment, developers can initiate the process by creating a fork of [software-artifact-analyzer-configuration](#) and [software-artifact-analyzer](#). It is recommended to fork all branches. Forking all branches ensures that developers have a complete replica of the original repository, including all development branches. This allows them to work on feature branches, bug fixes, or other changes without any limitations.

When needed, they can synchronize their forked repositories with the original repositories by executing the following commands:

```
# Add the original repository as a remote
git remote add base
https://github.com/iVis-at-Bilkent/software-artifact-analyzer.git
# Fetch all the commits from the original repository
git fetch base
git merge base/master
```

To selectively integrate specific commits from the original software-artifact-analyzer repository, developers can use the cherry-pick command:

```
# Fetch a specific commit
git cherry-pick -n 12ae1...
```

2. CASE Tool Integration Applications

SAA is designed to be used with tools like Jira and GitHub. In order for users to use SAA with their own projects, every SAA application should implement an integration flow for their own SAA implementations. This section will detail how developers can create GitHub and Atlassian applications for SAA that users can use to authorize and start using SAA applications with their own data. To get started, developers need to set up GitHub and Atlassian applications with appropriate OAuth configurations. Follow the steps below to complete the setup.



Figure 2: CASE tools integration flow

Prerequisites:

- A GitHub account
- An Atlassian account

2.1 Setting Up An GitHub Application

1. Create a GitHub App

- Go to [GitHub Developer Settings](#) and click on "New GitHub App".
- Fill in the necessary details for your application:
 - **GitHub App name:** Choose a name for your app.
 - **Homepage URL:** <http://localhost:4400>
 - **Callback URL:** <http://localhost:4400/?setup=Github>
 - **Webhook URL:** (Optional, depending on your needs)

Note: If your app runs on a server or different port, configure it accordingly

- Set the necessary **User permissions** and **Repository permissions**.
- Click on "Create GitHub App".

2. Configure OAuth 2.0

- Once the app is created, navigate to the app settings.
- Under "OAuth 2.0", generate new credentials for the app.
- Make sure to save the **Client ID** and **Client Secret**.

2.2 Setting Up An Atlassian Jira Application

1. Create an Atlassian App

- Go to the Atlassian Developer Console and click on "Create a new app".
- Select "OAuth 2.0 integration" and click "Next".
- Fill in the necessary details:
 - **App name:** Choose a name for your app.
 - **Callback URL:** <http://localhost:4400/?setup=Jira>
- Configure the permissions for the app. Make sure to select the appropriate scopes that your app will require.






API name	Scopes used	Action
 User identity API Get the profile details for the currently logged-in user, such as the Atlassian account ID and email.	2	Configure Documentation
 Confluence API Get, create, update, and delete content, spaces, and more.	0	Add Documentation
 BRIE API Create, cancel, and read backup and restore, retrieve and publish cloud details.	0	Add Documentation
 Jira API Get, create, update, and delete issues, projects, fields, and more.	11	Configure Documentation
 Personal data reporting API Report user accounts that an app is storing personal data for.	1	Configure Documentation

Figure 3: Atlassian app permissions

2. Configure the Classic Jira Platform REST API

- Navigate to the "Authorization" section in the app settings.
- Configure the Classic Jira platform REST API authorization URL.

3. Authorization Configuration in SAA


```
npm install
```

To run the Angular UI in development mode, you can use the Angular CLI's `ng serve` command. This command compiles the application, launches the development server, and opens your application in a default web browser <http://localhost:4200/>.

```
npm run ng serve
```

To prepare your Angular application for production, you need to build the application using the Angular CLI's `ng build` command. After building, you can use a simple Node.js server to serve the production-ready application.

- Build the Angular application:

```
npm run ng build
```

This will generate a `dist/` directory containing the production-ready files.

- Start the Node.js server:

```
npm start
```

By default, the server will start on port 4450. You can access your production-ready application by navigating to <http://localhost:4450/> in your web browser. You can access the production-ready application through the SAA user interface's top navbar by navigating to "Project | New..." The production-ready application is embedded as an `iframe` within the SAA user interface.

3.3 Data Retrieval

The `data retrieval` module serves as a critical component within the [software-artifact-analyzer-configuration](#) repository, seamlessly integrating Perceval backends for the three primary data sources: Git, GitHub, and Jira. The intricate process involves retrieving data from each of these sources and storing it meticulously in the `data` folder, specifically in JSON format. For the incorporation of an additional data source, the procedure is as follows: the corresponding backend must be introduced into the [data retrieval](#) module. This newly integrated backend should then undergo a transformation process, converting the retrieved data into JSON format. Subsequently, the processed data should find its place under the [data](#) folder.

```
/software-artifact-analyzer-configuration
|-- data
|   |-- git_commit.json
|   |-- github_pr.json
|   |-- jira_issue.json
|   |-- new_backend.json    # Example for the newly integrated backend
|-- data_retrieval
|   |-- data_retrieval.py
|   |-- perceval_git_commit.py
|   |-- perceval_github_pr.py
|   |-- perceval_jira_issue.py
```

```
| |-- new_backend.py      # Corresponding file for the new backend
```

3.4 Data Cleaner

The data cleaner module is primarily responsible for identifying similar developers across different sources and merging them into a unified developer profile. This module is designed to function seamlessly without requiring any customization.

3.5 Graph Builder

The Graph Builder module serves as the bridge between your application and the Neo4j database instance. It facilitates the integration by establishing a secure connection. To initiate this connection, you must specify the bolt address of your Neo4j instance along with the required authentication credentials. These credentials, including the username and password for the Neo4j driver, should be carefully defined within the [graphbuilder/GraphBuilder.py](#) file.

The [connectors/Neo4jConnection.py](#) is instrumental in handling and executing various tasks, such as merging or creating new nodes and edges. The merging process involves utilizing source classes and methods, which are specific to each backend. For every type of artifact in our project, there is a corresponding source class. These source classes are responsible for creating instances based on retrieved data and are then used to merge or create the artifact in the Neo4j database.

If you're introducing a new backend for a different type of artifact, follow these steps:

- 1. Define a new source class for the specific artifact type in the 'graph_builder/connectors/extra_source_classes.py' file. Let's call this class ArtifactType1. Define the structure and methods for handling ArtifactType1 data.*

```
# graph_builder/connectors/extra_source_classes.py
class ArtifactType1:
```

Import the newly created source class (ArtifactType1) into the Neo4jConnection.py file.

```
#graph_builder/connectors/Neo4jConnection.py
from graph_builder.connectors.extra_source_classes import
ArtifactType1
```

- 2. Update the classes dictionary in the addMultipleArtifacts method in Neo4jConnection*

```
classes = {
    "git": GitCommit,
    "jira": JiraIssue,
    "github-pr": GithubPr,
    "artifacttype1": ArtifactType1, # Add this line...}
```

3. Add a new method to the `Neo4jConnection` class that includes the logic for merging `ArtifactType1` and its respective edges into the graph.

```
def __addArtifcat1(self, tx, artifact1: ArtifactType1, projectId,
analysisId):
```

4. Update the methods dictionary in the `addMultipleArtifacts` method in `Neo4jConnection`

```
methods = {
    "git": self.__addCommit,
    "jira": self.__addIssue,
    "github-pr": self.__addPullRequest,
    "artifact1": self.__addArtifcat1, # Add this line
```

With these modifications, your `Neo4jConnection` class will be capable of handling the merging of the new artifact type (`ArtifactType1`) into the Neo4j database.

3.6 Creating and Connecting to a Neo4j Database

In SAA, we utilize Neo4j as our database management system. Developers may need to create their own Neo4j database instance for testing, development, or customization purposes. This section provides guidelines on how developers can establish a new Neo4j database instance and connect it to our system.

3.6.1 Establishing a New Neo4j Database Instance

Developers have the flexibility to create their own Neo4j database instances using various methods. Here, we discuss a common approach which is using Neo4j Desktop:

1. *Install Neo4j Desktop: If not already installed, download and install Neo4j Desktop from [here](#).*
2. *Create a New Project: Open Neo4j Desktop and create a new project by clicking on the "Add" button and selecting "New Project."*
3. *Add a New Database: Within the created project, click on the "Add Database" button. Choose the desired Neo4j version and configure the database settings as needed. SAA is compatible with Neo4j 4 and later versions.*
4. *Start the Database: Once configured, start the database instance by clicking on the "Start" button.*

3.6.2 Preparing the NEO4J Database Instance

Before connecting to the system, it is recommended to install essential plugins such as APOC. Additionally, if you are using custom procedures, ensure they are correctly configured and available within the Neo4j database instance.

Installing APOC (Awesome Procedures on Cypher): APOC [2] is a popular library of procedures and functions for Neo4j, providing additional functionalities beyond what is available in the core Cypher language. Since our SAA utilizes APOC functionalities, it is essential to install the APOC plugin before connecting to the system.

1. *Open Neo4j Desktop: Launch Neo4j Desktop and select the desired project.*
2. *Add Plugin: In the project overview, click on the "Add Plugin" button.*
3. *Select APOC: Choose APOC from the list of available plugins and click "Install." Neo4j Desktop will automatically handle the installation process.*
4. *Restart Database: After installation, restart the Neo4j database instance associated with the project to apply the changes.*

Installing Custom Procedures: If you are using custom procedures in your SAA project, ensure they are correctly configured and available within the Neo4j database instance. You can check how to add custom procedures from [Section 6](#).

3.6.3 Connecting NEO4J Database Instance to the System

Once the Neo4j database instance is up and running, developers can connect it to the [software-artifact-analyzer-configuration/graph-builder](#) system using the Bolt protocol and the [software-artifact-analyzer](#) system using the HTTP protocol. Here's how to establish the connection:

1. *Obtain Bolt Address: Determine the Bolt address of the Neo4j database instance. This typically follows the format `bolt://<host>:<port>`.*
2. *Obtain HTTP Address: Determine the HTTP address of the Neo4j database instance. This typically follows the format `http://<host>:<port>`.*
3. *Update Configuration:*
 - a. *For the software-artifact-analyzer-configuration, initiate this connection by specifying the Bolt address of your Neo4j instance along with the required*

authentication credentials. These credentials, including the username and password for the Neo4j driver, should be carefully defined within the [GraphBuilder.py](#) file.

- b. For the software-artifact-analyzer, update the environment files with the new HTTP address of the Neo4j database instance as described in [Section 4.4](#).

By following these steps, developers can establish connections between the Neo4j database instance and both the [software-artifact-analyzer-configuration/graph-builder](#) and [software-artifact-analyzer](#) systems, facilitating seamless data interaction and analysis.

Additional Resources

For more detailed instructions on creating and configuring Neo4j databases, developers can refer to the official Neo4j documentation:

- [Neo4j Desktop Documentation](#)
- [Establishing a New Neo4j](#)
- [Neo4j Documentation](#)

4 Custom Folder

SAA was created using Visuall [1]. Visuall is composed of three main Angular modules: AppModule, CustomizationModule, and SharedModule. To customize SAA to meet our specific requirements, we made changes to the CustomizationModule of the Visuall.

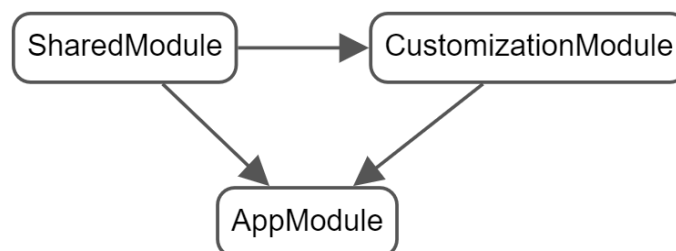


Figure 5: Basic Architecture of Visuall

The custom folder includes 3 subfolders and a `customization.module.ts` file. Below you can find the explanation about each folder and file.

```
/src
|-- app
|   |-- custom
|   |   |-- analyses
|   |   |-- customization-service
|   |   |-- config
|   |   |-- operational-tabs
|   |   |-- customization.module.ts
```

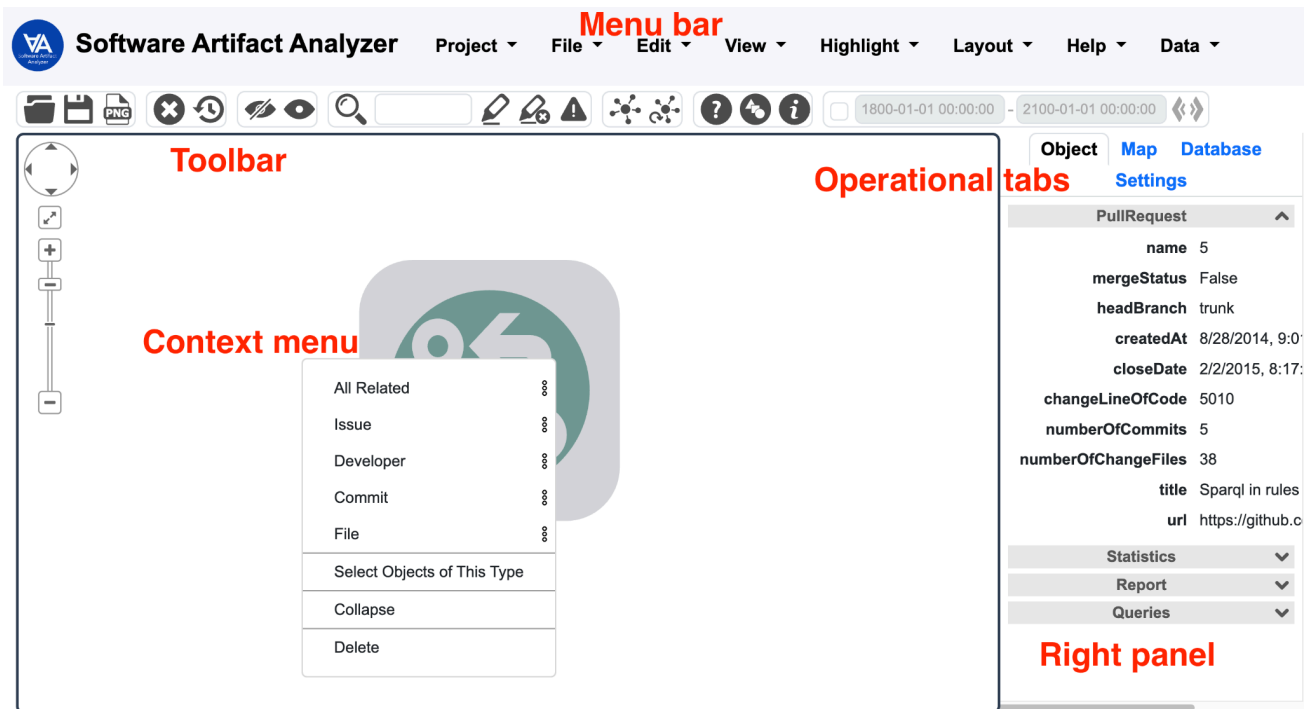


Figure 6: SAA UI

4.1 config

The `config` folder includes configuration files such as `app-description.json` file and `enum.json`.

```

/custom
|-- config
|   |-- app_description.json
|   |-- enum.json

```

The [app_description](#) file includes the description for each node and edge type. You can refer to the [Visual Developer Guide](#) for further explanation about the descriptions of graph elements.

To add a new node type or edge type:

1. Define a new class of node (e.g., "Method") under the "objects" section with associated properties and data types.
2. If necessary, define a new edge type under the "Relations" section, specifying the source and target nodes.
3. If the new node or edge requires specific styling, update the "Styling graph objects" section accordingly.
4. Define the time mapping for the newly added edge and node types
5. If the new node type involves an enum property, update the "Enumeration Mapping" section and add corresponding values to "enums.json." Ensure that the necessary

properties, data types, and styling configurations are properly set based on the requirements of the new node or edge type.

Once this file is prepared, the style generator reads the description file and the Cytoscape.js style file and modifies `index.html`, `styles.css`, `properties.json`, and `stylesheet.json` files using the information in the description file. To do so:

- First, navigate to the `src` folder
- Then, execute `node style-generator.js` (by default it will use [app_description.json](#))

Although it might not be necessary, it is recommended to always run this command after changing the description file.

Selecting Node Icon and Color

In the SAA drawing canvas, each node type is represented by an icon and assigned a specific color. The edges originating from nodes also adopt the same color scheme for consistency and clarity.

When selecting icons for your nodes, it's essential to consider the following points:

- Use SVG icon images
- Ensure that icons for nodes are as round as possible to facilitate perpendicular edge connections from any location.
- Opt for dark colors when selecting node colors to enhance visibility and contrast within the interface.
- Prioritize simplicity and legibility when choosing iconography to maintain clarity and prevent visual clutter.

4.2 customization-service

The `customization-service` folder houses files dedicated to tailoring features from the Visual [1], including the Menubar, toolbar, and context menu.

```
/custom
|-- analyses
|-- customization-service
|   |-- context-menu-customization.service.ts
|   |-- cy-style-customization.service.ts
|   |-- group-customization.service.ts
|   |-- navbar-customization.service.ts
|   |-- theoretic-properties-custom.service.ts
|   |-- toolbar-customization.service.ts
```

toolbar-customization.service.ts

For adding a new item to the toolbar menu or modifying the existing toolbar, this file is where you should make your changes.

navbar-customization.service.ts

To add new items to the navbar or modify existing ones, this file is where you would implement those changes.

cy-style-customization.service.ts

This service is responsible for customizing the styles of the visual elements. You can adjust the appearance of nodes, edges, and other graphical components to fit your needs.

group-customization.service.ts

For implementing new grouping functionalities, such as clustering by developer or other criteria, you can utilize this service. It allows for the creation and customization of groupings within the visual interface.

context-menu-customization.service.ts

The context menu stands as a core feature within the SAA, facilitating effortless navigation between artifacts and developer nodes. Each node type must possess a type-specific context menu. Therefore, when introducing a new node type, it's imperative to define its context menu in the [context-menu-customization.service.ts](#) file.

To integrate the context menu for a new node type and ensure navigation across different node types, follow step-by-step instructions:

- 1. Within the [context-menu-customization.service.ts](#) file, introduce a context menu item tailored for the new node type. This involves specifying the menu items, labels, and actions associated with the context menu for seamless user interaction.*
- 2. Go beyond merely adding the context menu for the new node type. Extend the other node types' context menus by seamlessly incorporating context menu options that establish meaningful connections with the new node type.*
- 3. Include queries that illuminate relationships between the new node type and other existing node types. This ensures that users can effortlessly comprehend the associations and dependencies between different elements in the system.*

4.3 operational-tabs

Operational tabs contain one subfolder for each operations tab. If developers wish to add a new sub-tab, they can place the relevant components in the corresponding folders within the operational tab where the new sub-tab is intended to be added. They also should add components into the subtab lists inside the customization.module.ts.

```
/custom
```

```
|-- operational-tabs
```

```
| |-- object-tab
| |-- map-tab
| |-- database-tab
```

4.4 analyses

The `analyses` folder serves as a repository for housing various analysis components within a system. These analysis components can be strategically organized into two main categories: "custom queries," and "object queries".

Custom Queries: Under the "custom queries" section of the `analyses` folder, developers have the flexibility to incorporate analysis components that cater to specific, user-defined queries or analytical processes. This section provides a dynamic space for the inclusion of custom-tailored analyses, allowing users to interact with the system in a way that aligns with their unique requirements and objectives. Custom queries often involve more generalized analyses that span across different node types, offering a broad spectrum of analytical capabilities within the application. For adding a query component under custom queries you should add it to the list in the query list in the [customization.module.ts](#).

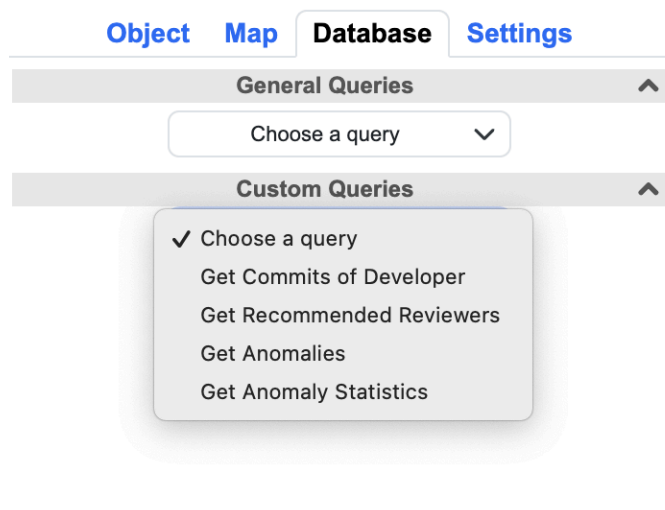


Figure 7: Custom Queries

Object Queries: The "object queries" section of the `analyses` folder is designated for analysis components that are intricately linked to specific node types within the system. These analyses focus on the characteristics, properties, and relationships associated with particular types of nodes. Placing these components under "object queries" enhances the organization and accessibility of node-specific analyses, streamlining the process for developers and users alike. Object queries provide a more targeted approach to analyzing and extracting insights from the data associated with individual node types, ensuring a more fine-grained understanding of the information present in the system. Object queries are specific to each node type and they are under the queries subtab on the Object operational tab.

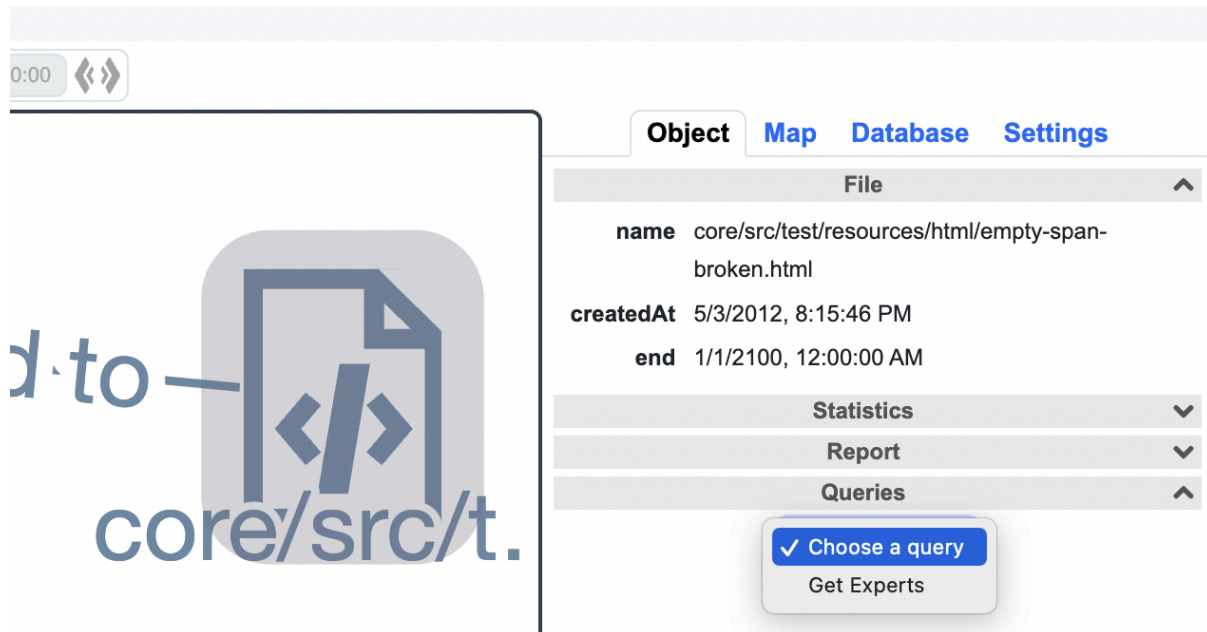


Figure 8: Object Queries

For adding a query component under object queries you should add it to the node type query list in the [object-queries.component.ts](#). To add a new analysis component you can implement the QueryComponent interface by following these steps.

1. Create an angular component inside the analyses folder

```
ng generate component analyses/MyQueryComponent
```

2. Define a Data Model: Before implementing the QueryComponent, define the data model (T) that represents the structure of the data you'll present as a table response.

```
interface MyDataModel {
  // Define the properties of your data model here
}
```

3. Implement the QueryComponent Interface: Create a class that implements the QueryComponent interface. This class will serve as your analysis component.

```
export class MyQueryComponent implements QueryComponent<MyDataModel> {
  tableInput: TableViewInput = /* Initialize your TableViewInput */;
  tableFilled = new Subject<boolean>();
  tableResponse: MyDataModel = null;
  graphResponse: GraphResponse = null;
}
```

```
clearTableFilter = new Subject<boolean>();

ngOnInit(): void {
    // Implement the OnInit lifecycle hook if needed
}

prepareQuery(): void {
    // Implement the query preparation logic
}

loadTable(skip: number, filter?: TableFiltering): void {
    // Implement the logic to load table data
}

loadGraph(skip: number, filter?: TableFiltering): void {
    // Implement the logic to load graph data
}

filterGraphResponse(x: GraphResponse): GraphResponse {
    // Implement the logic to filter graph response
}

fillTable(data: MyDataModel[], totalDataCount: number | null): void {
    // Implement the logic to fill the table with data
}

getDataForQueryResult(e: any): void {
    // Implement the logic to get data for a query result
}

filterTable(filter: TableFiltering): void {
    // Implement the logic to filter the table
}

filterTableResponse(x: MyDataModel[], filter: TableFiltering):
MyDataModel[] {
    // Implement the logic to filter the table response
}
}
```


4. *Customize Query Logic: Customize the `prepareQuery`, `loadTable`, and `loadGraph` methods according to your specific data source and query requirements. Update the GraphQL queries, REST API calls, or other data retrieval methods accordingly.*
5. *Handle Data Processing: Customize the `fillTable`, `filterGraphResponse`, `filterTable`, and `filterTableResponse` methods to handle the processing and manipulation of the retrieved data based on your application's needs.*
6. *Designing the HTML template and style it according to your analysis*

You can easily reference the existing analysis components, such as [UnassignedBugsComponent](#), to gain insights into the implementation details. By making minor modifications to the methods and structure, you can tailor the component to meet the specific requirements of your analysis.

4.4 Database Connection

To connect to the database, Software Artifact Analyzer uses [environment variables](#). For each different environment you would like to use, you should add a new environment file. You can make your own files similar to these: [environment.ts](#) and [environment.heroku.ts](#).

5 Starting Software Artifact Analyzer

5.1 Development Mode

Before running SAA in development mode, make sure to install the project dependencies using;

```
npm install
```

To run SAA in development mode, you can use the Angular CLI's `ng serve` command. This command compiles the application, launches the development server, and opens your application in a default web browser <http://localhost:4200/>.

```
npm run ng serve
```

5.2 Production Mode

To prepare your Angular application for production, you need to build the application using the Angular CLI's `ng build` command. After building, you can use a simple Node.js server to serve the production-ready application.

- Build the Angular application:

```
npm run ng build
```

This will generate a `dist/` directory containing the production-ready files.

- Start the Node.js server:

```
npm start
```

By default, the server will open on port 4400. You can access your production-ready application by navigating to <http://localhost:4400/> in your web browser.

For both the development mode and the production mode, you should also start the Software-Analyzer-Configuration application flask server and UI as described in Sections [2.1](#) and [2.2](#), respectively.

6 Custom Procedure

A user-defined procedure is a mechanism that enables you to extend Neo4j by writing customized code, which can be invoked directly from Cypher. Currently, for deploying custom queries for SAA we add them into the [saa-advanced-query](#) project. The [saa-advanced-query](#) repository is a fork of the original [visuall-advanced-query](#) repository, which serves as a foundation for advanced query capabilities in the context of the Visuall [1] toolkit. The primary objective of `saa-advanced-queries` is to extend the functionality provided by the original repository to cater to the specific requirements of the SAA and its team of developers. This extension includes the addition of custom Java-based Neo4j procedures tailored to the needs of SAA. Before beginning to write your custom procedure please review the existing NEO4J document [here](#) to familiarize yourself with the process.

Here's a guideline for incorporating custom Java-based procedures into the `saa-advanced-queries` repository for Neo4j:

1. Set Up Your Development Environment: Ensure that your development environment is set up properly for Java development, including IDE setup and dependency management.
2. Clone the Repository: Clone the [saa-advanced-queries](#) repository to your local machine. (we are currently using the unstable version)
3. Create a Feature Branch: Create a new feature branch from the main branch where you will add your custom procedures. Make sure the branch name is descriptive of the feature you're adding.
4. Implement Your Custom Procedure: Implement your custom Java-based procedure following the standards and conventions of Neo4j procedures. Refer to the existing procedures and the [Neo4j documentation](#) for guidance.

5. Test Your Procedure: Thoroughly test your procedure to ensure it works as expected. This includes unit tests as well as integration tests with Neo4j. You can write your test inside the [AdvancedQueryTest.java](#)
6. Update README and Documentation: Update the [README](#) file and any other relevant documentation to include information about your new procedure. This ensures that other developers are aware of its existence and how to use it.
7. Create an Executable Jar File: Build your project to create an executable JAR file containing your custom procedure along with any dependencies.

```
mvn clean install
```

8. Add executable JAR files containing custom procedures as plugins to Neo4j:
 - a. Find the plugins directory within your Neo4j installation. This directory is typically named plugins and is located within the Neo4j installation directory.
 - b. Copy the JAR File: Copy the executable JAR file containing your custom procedures into the plugins directory.
 - c. Restart Neo4j: Restart the Neo4j database server to allow it to detect and load the newly added plugin. This step is essential for Neo4j to recognize and make use of the custom procedures contained within the JAR file.
9. Verify Installation: Once Neo4j has restarted, verify that the custom procedures from your JAR file have been successfully loaded and are available for use. You can do this by checking the Neo4j logs for any errors during startup and testing the custom procedures' functionality through Cypher queries.
10. Usage in Cypher Queries: With the custom procedures successfully installed as plugins, you can now use them in Cypher queries within Neo4j. Invoke the procedures using the CALL syntax followed by the procedure name and any required parameters.

References

- [1] i-Vis Research Laboratory. 2021. "Visuall: A tool for convenient construction of a web-based visual analysis component", Bilkent University, Ankara, Turkey.
- [2] APOC Library, "Awesome Procedures for Neo4j 5.21.0.x (Extended)," 2024. [Online]. Available: <https://github.com/neo4j/apoc>. [Accessed: 07-Jul-2024].