

PAVEL VLASOV

Beyond Diagrams



CAPTURE YOUR THOUGHTS IN DIAGRAMS,
GENERATE DOCUMENTATION,
MAKE THEM EXECUTABLE

Copyright © 2024 Pavel Vlasov

All rights reserved.

No portion of this book may be reproduced in any form without written permission from the publisher or author, except as permitted by U.S. copyright law.

Table of Contents

Introduction.....	5
Prerequisites.....	8
Resources.....	8
Diagramming problem domain.....	9
Text/Label.....	11
Shape.....	11
Icon.....	11
Border.....	12
Connection.....	12
Size.....	13
Geometry.....	13
Color.....	13
Opacity.....	14
Gradient.....	14
Z-Order.....	14
Draw.io.....	15
Non-Draw.io data sources.....	18
Documentation sites.....	21
Demo.....	22
System Context Diagram.....	24
Container Diagram.....	25
API Application Component Diagram.....	27
Mainframe Banking System Code Diagram.....	28
MainframeBankingSystemFacadeImpl Documentation.....	29
Generation.....	30
GitHub Actions.....	30
Nasdanika CLI.....	30
Java.....	31
Configuration.....	31
Search.....	32
Navigation panel filter.....	32
Search page.....	33
Glossary.....	34
Summary.....	35
Sites.....	37

Action models.....	37
Diagrams.....	38
Diagramming ecosystem.....	38
Semantic search & Chat.....	39
Markdown.....	41
Embedded images.....	41
PNG resource.....	41
JPEG resource.....	41
PNG.....	41
JPEG.....	41
Embedded diagrams.....	42
Drawio.....	42
PlantUML.....	42
Loading from a resource.....	42
Inline.....	42
Sequence.....	43
Component.....	44
Mermaid.....	45
Extensions.....	45
Full book overview.....	47

Introduction

This book is about communication. More specifically – visual and non-visual communication using diagrams and artifacts derived (generated) from them. The book is written from a standpoint of a software developer. However, I¹ believe that the concepts explained in this book can be applied outside of software development/corporate IT.

Diagrams is a fundamental form of communication. They've been used by humanity since prehistoric times and predate writing. You may ask – if the diagrams have been around for so long, why to write a book on the subject which surely should be well-understood and well-documented?

According to my experience, there is a gap in the diagramming continuum and in how people approach diagramming in general. Similar to writing, diagramming is not an innate skill – the fact that a person can connect two boxes with an arrow does not make them a good diagrammer². The goal of this book is to help readers to start “thinking in diagrams” in order to become better diagrammers and, therefore, communicators.

Wikipedia defines a [diagram](#) as

A symbolic **representation** of information using visualization techniques

with [representation](#) in turn defined as:

A **reference** (to an object) conveyed through pictures

Wikipedia also defines a “specific meaning” for the term “diagram”:

The genre that shows qualitative data with shapes that are connected by lines, arrows, or other visual links

1 Some people believe that using “I” in books is a bad style and “the author” shall be used instead. I, personally, think that “the author” is too impersonal and 10 times more letters (including the space) and, therefore, mental burden on the reader. Because this book is about effective and efficient communication, using an ineffective construct would go against the purpose of the book. So, first person it is!

2 A person who authors diagrams. Program -> programmer, diagram -> diagrammer

In this book the term diagram is used in this specific meaning extended to include quantitative data.

There are many diagrams which can be divided into two major categories:

- Just representation. Such diagrams leave the job on deducing what a particular shape on the diagram means to the beholder. Examples of tools which produce such diagrams include "Draw.io" (diagrams.net) - covered in details below, MS Visio, MS PowerPoint.
- Representations referencing objects belonging to a specific problem domain expressed in a domain language. Examples: UML, BPMN.

Enterprise IT is dominated by the first type of diagrams³ with Draw.io and PowerPoint being predominant formats.

Despite the fact that UML has been around for about 30 years, it is not being used widely and is very often misused/abused. This fact was recognized by the author of "The C4 model for visualising software architecture"⁴ Simon Brown:

As an industry, we do have the Unified Modeling Language (UML), ArchiMate and SysML, but asking whether these provide an effective way to communicate software architecture is often irrelevant because many teams have already thrown them out in favour of much simpler "boxes and lines" diagrams. Abandoning these modeling languages is one thing but, perhaps in the race for agility, many software development teams have lost the ability to communicate visually.

The reason for this is that enterprises have their own "languages", often not explicitly defined. They don't "speak" in, say, UML or BPMN.

Tools like [Eclipse Sirius](#) make it rather easy to create specialized diagramming tools, but they are not widely used either.

I always wanted a solution which would allow me to:

- Create a diagram

³ Here and below "in my opinion" or "in my experience" is implied unless explicitly stated otherwise

⁴ <https://bit.ly/nsd-c4>

- Share it with others without requiring them to have a specialized viewer
- Document diagram elements using rich text
- Create sub-diagrams for diagram elements with an infinite level of nesting - as many as needed
- Associate behavior with diagram elements—make them executable

I'm proud to say that now I have it! This is what this book is about – it explains how to navigate the diagramming continuum from "representation only"/"bare" diagrams to diagrams from which you can generate documentation sites, executable diagrams, and diagrams mapped to domain models. With this knowledge you will be able to maximize effectiveness and efficiency of your diagramming efforts by choosing the right point on the continuum to minimize the total amount of effort needed to communicate (produce and consume).

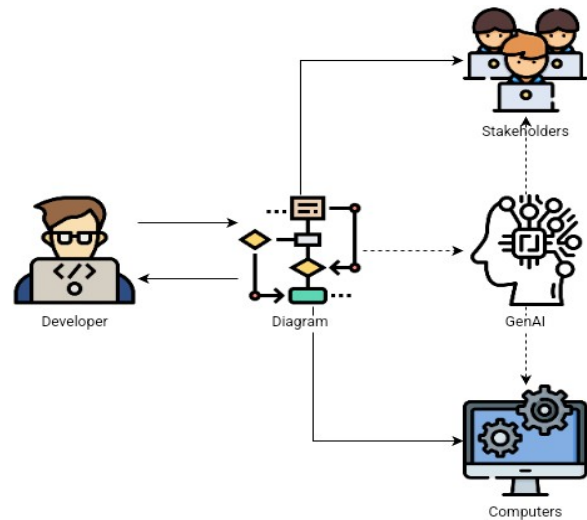


Figure 1: Diagrams as a communication vehicle

This is not a recipe book, rather a menu – it explains WHAT can be done with diagrams and provides samples (demos) and links to recipes explaining the HOW part.

The book focuses on the four types of communication, as shown on Figure 1:

- **Self**
 - *Present* – to get a grasp of the problem at hand and break it down into pieces.
 - *Future* - "The faintest ink is better than the strongest memory" – your future self may forget what was obvious for your present self.

- **Other humans** – to communicate intent, gather feedback or instruct.
- **Computers** – to instruct them what to do. Either make diagrams executable, use them as configuration resources, or generate executable code from them.
- **GenAI** – in this time and age it is everywhere. It might be used to generate descriptions and summaries for humans or as a chat interface, so humans can have a dialog with your diagrams or a specific diagram element. It may also be used to generate code for computers to execute.

The book is structured to maximize reader's return on time investment. It starts with explaining the problem domain of diagramming and diagram expressive means. Reading just that shall help you with your diagramming efforts. Setting up a web site generated from a diagram is a matter of a few minutes on GitHub. Executable diagrams and semantic mapping would require more time investment, you may consider it a reinvestment of time you saved by using site generation and more effective diagrams.

Prerequisites

Techniques explained in this book are implemented in Java as Open Source Maven libraries available on Maven Central under the terms of Eclipse Public License 2. If you are a Java developer, you can incorporate them into your solutions.

Some of the techniques are wrapped into Nasdanika CLI⁵ commands. Nasdanika CLI requires Java 17.

Generation of a documentation site from a Draw.io diagram is wrapped into a GitHub pages action. To use it you need to have a GitHub account to create a new repository from a template and then adjust it to your needs.

Resources

Links to external resources are available at this page: <https://bit.ly/m/beyond-diagrams>

5 <https://bit.ly/nsd-cli>

Diagramming problem domain

In this book Draw.io is used to author and view diagrams. However, most of functionality is implemented in a generic fashion with a layer of Draw.io specific functionality on top of it. So, before getting into the specifics of Draw.io, let's take a look at diagramming in general – how it be used with any tool, including tools like paper and pencils or and whiteboard and chalks.

Diagrams visually represent information about “things” and their relationships using shapes connected with lines and arrows. We will call shapes *Nodes*, lines/arrows *Connections*, and “things” *Entities*. All of them can have associated information. We will call this information *Features*. Entities can “do stuff”, we will call it *Operations*.

You can think of entities as nouns, features as adjectives, and operations as verbs. “*The quick brown fox jumps over the lazy dog*”:

- Fox, Dog – entities
- Quick, Brown, Lazy – features
- Jumps over – operation⁶

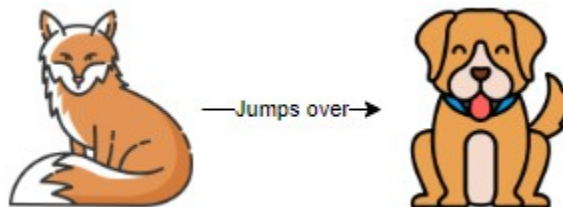


Figure 3 shows a pseudo-UML diagram of *Figure 2: Operation* our diagramming problem domain. A *Diagram* contains *Diagram Elements* and describes a *Domain*. *Diagram Elements* represent *Domain Entities*, *Entity Features* or *Entity Operations*. *Diagram Elements* can be of two types – *Nodes* and *Connections*. *Nodes* can have outgoing and incoming *Connections* and can contain other *Diagram Elements*. *Connections* can be connected to source and target *Nodes*. Please note that it is possible to have a dangling connection with no source Node, target Node, or both.

⁶ Jump might be an operation and over might be its argument: `fox.jump(dog, over)`

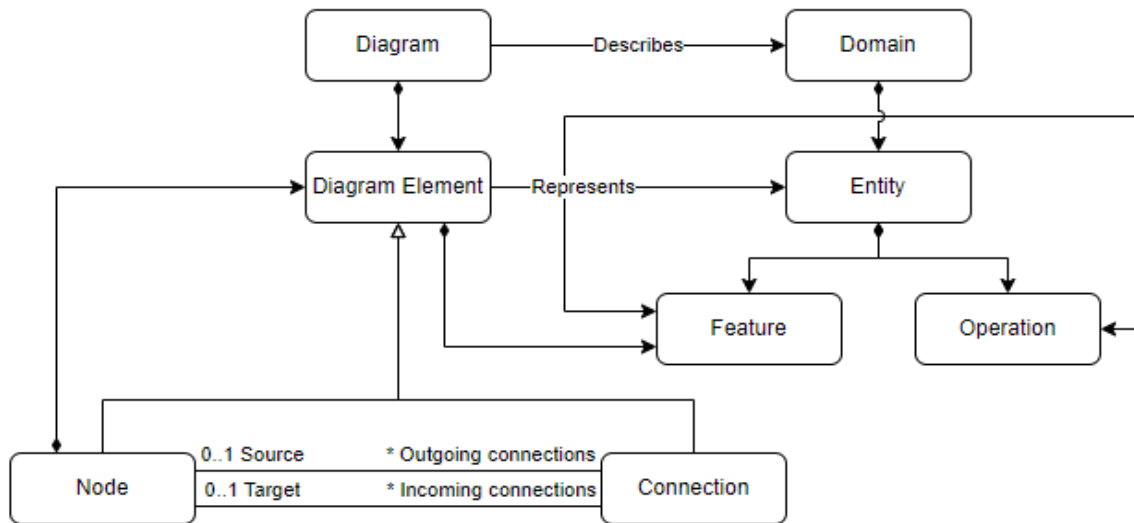


Figure 3: Diagramming problem domain

A *Diagram Element* can represent one *Entity* or a group of *Entities*. Some *Diagram Elements* may not represent anything and be purely visual elements.

Domain is a collection of *Entities* which have *Features* and *Operations*.

Entity features can be either:

- *Attributes* or *properties* with scalar or collection of scalars values:
 - Name: Jane
 - Color: Brown
 - Age: 10
- *References* – single or multiple entity values:
 - Jane father: Joe
 - USA constituents: Florida, Texas

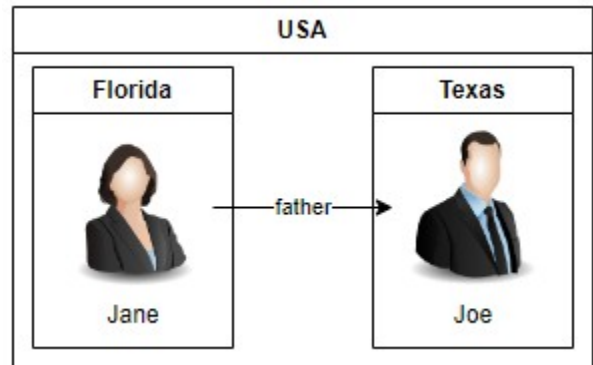


Figure 4: Features

- Florida residents: Jane

Diagram element features can be divided into visual and non-visual. Many diagram editors do not support non-visual features.

Let's take a quick look at typical visual features using Figure 5 as an example.

Text/Label

Diagram elements may have text labels. These labels can communicate both qualitative and quantitative information. Labels can be plain text or formatted – bold, italic, colors, ...

Shape

There are many shapes – circle, rhombus, rectangle, ellipse, ...

Shape can represent both qualitative and quantitative information. For example, circles represent apples and rectangles represent oranges. Ellipse major/minor axis ratio and rectangle width/height ratio may represent quantitative information.

Shapes can have background color, border, and opacity. All of them can be used to display both qualitative and quantitative information – see below.

Icon

An icon can be used instead of a shape. There are millions of icons available free with attribution or for a moderate subscription fee from multiple sources⁷. Similar to shapes,

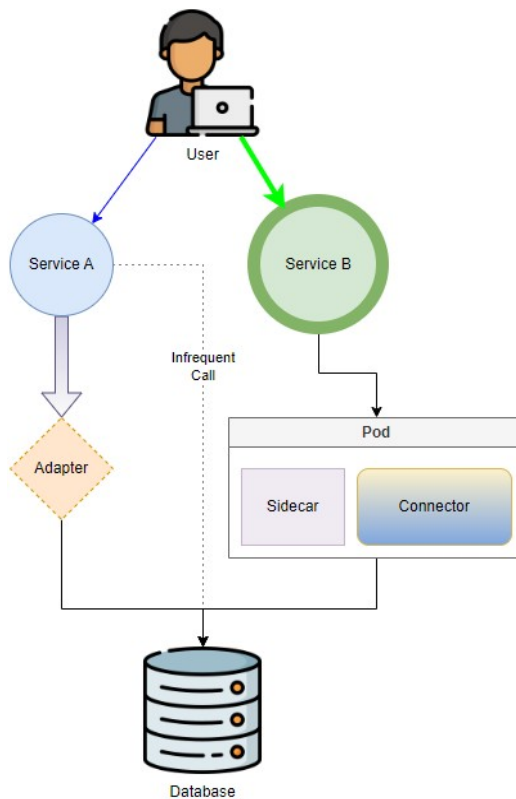


Figure 5: Visual features

⁷ <https://bit.ly/nsd-icons>

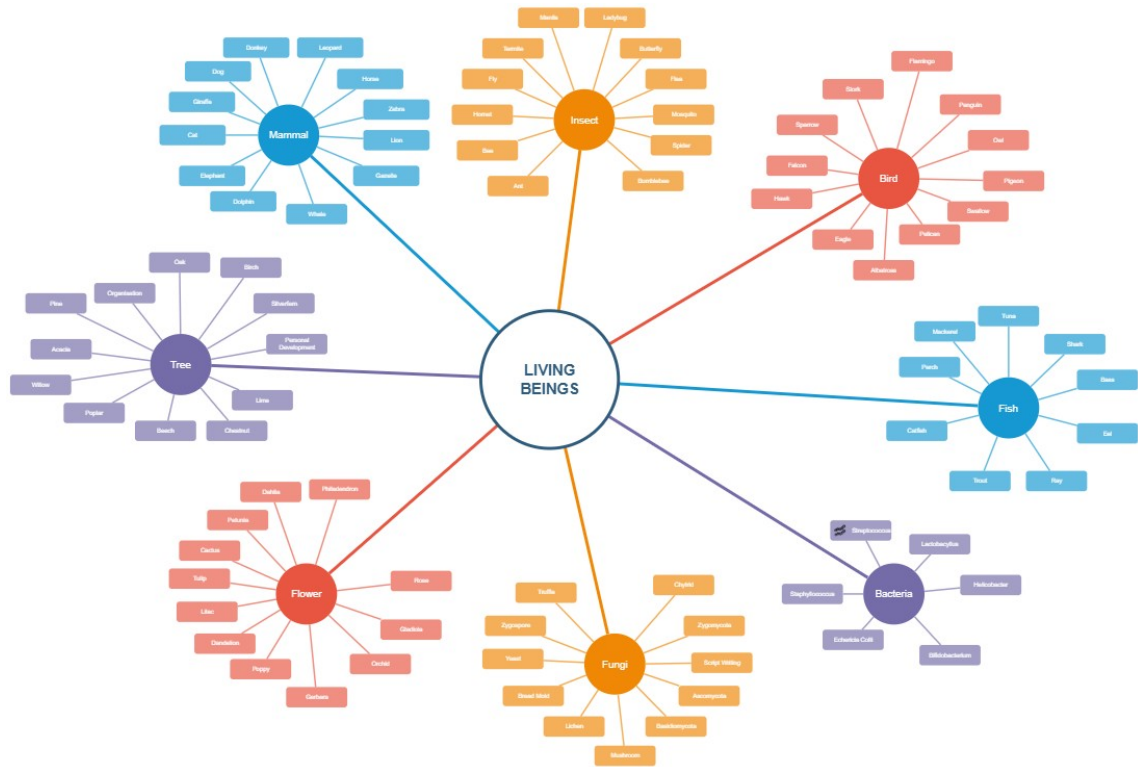


Figure 6: Living beings

icons can have borders and opacity to communicate qualitative or quantitative information.

Border

Shape or icon borders can be of different width and color. They may also have different patterns – solid, dashed, dotted. All of these features may represent qualitative or quantitative information.

Connection

Similar to border, connections may have width, color, and pattern. They may also have start/end decorators and multiple labels.

Size

Shape size may be used to communicate quantitative information – number of lines of code, account balance, transaction volume.

Geometry

Geometry includes nodes relative position and distance. Relative position can be used to order nodes and distance may represent some numeric value. Relative position can be used to compute Cartesian or angular order.

Figure 6 shows a diagram where nodes can be ordered based on their angle relative to another (base) node. For example, bacteria can be ordered clockwise so Streptococcus comes before Lactobacillus.

Cartesian ordering would use horizontal and vertical positions to compare/order elements with 2 coordinates and 2 directions in each. For example, right-down ordering would compare nodes by their horizontal order first with nodes on the left being smaller and then by vertical order with higher nodes being smaller.

Color

Color can be used to communicate both quantitative and qualitative information.

Let's take a look at two ways of how colors can be represent information – ordering and categorization (bucketing).

For ordering we'd need a way to compare two colors. For example, for spectral ordering blue shall be greater than red. The HSV/HSB representations can be used for this – you can use

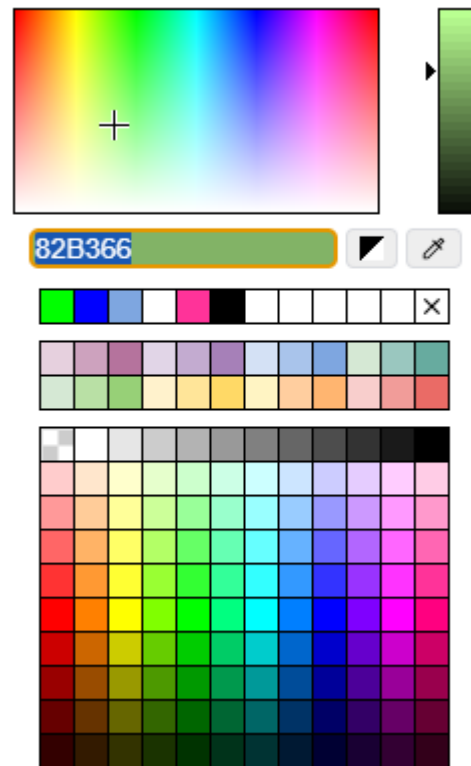


Figure 7: Color visual feature

the hue value for sorting/ordering. You'd need to decide what to do with saturation and brightness. One option is to break the hue into several ranges and then order by saturation and brightness within the range bucket – similar to the color picker in the Drawio editor shown on Figure 7

You may also define a list of base colors instead of ranges and map a specific color to a base color based on distance in the color space – here you might need to choose your distance definition. This way you may have, say, blue being before green because this is how color coding is set up in Jira – blue means in progress, green means done.

Categorization would work similar to the above – map color to a value, use exact match (can easily break) or least distance. Say. if it is more green than blue – then it is Done, not In Progress.

Opacity

Opacity may also represent information. For example, a translucent shape or connection may be of less importance or indicate future availability.

Gradient

You can use gradient to indicate that a shape belongs to two categories identified by gradient colors.

Z-Order

Z-order is important when shapes overlap. It may be used for ordering and be combined with geometric ordering.

Draw.io

There are many diagram editors, so why Draw.io? For a number of reasons listed below!

Draw.io is a cross-platform graph drawing software developed in HTML5 and JavaScript. Here is a quick summary of Drawio capabilities:

- Available on multiple platforms:
 - Web App,
 - Linux
 - macOS
 - Windows
- Available as a plugin in:
 - Confluence
 - VS Code
- Can be self-hosted:
 - WAR file
 - Docker container

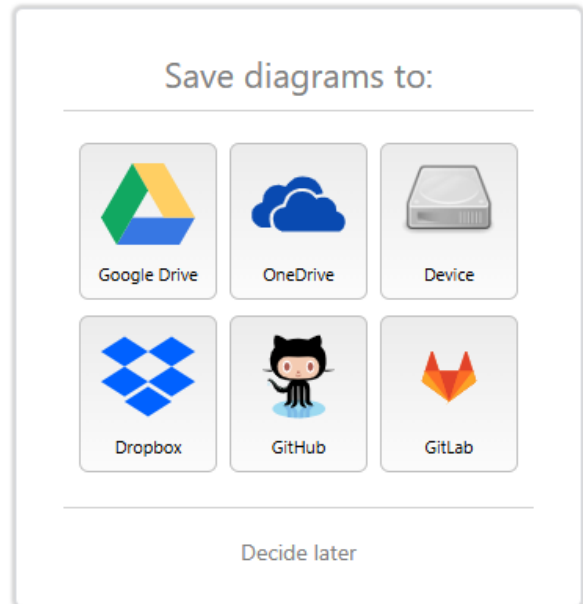


Figure 8: Draw.io supported storage

- Can load and save diagrams from/to GoogleDrive, OneDrive, local device (disk), Dropbox, GitHub, and GitLab as shown on Figure 8
- Has a large library of shapes and images – more than 2000
- Images can be copied/pasted from external sources, including icon sources mentioned above.
- Features an in-browser diagram viewer which allows to interact with the diagram – click on elements, go full screen, show/hide tags and layers, and switch to the editor.

- Tooltips can be used to provide descriptions for diagram elements.
- Diagram elements have unique editable IDs and user-defined text properties as shown in Figure 9.
- A diagram document may contain multiple pages with diagram elements linking to pages. It allows to create a page hierarchy.
- Java API to read, manipulate, and write diagrams – Nasdanika Drawio API.
 - The API extends the concept of linking to pages to cross-document linking and linking to diagram elements. This allows to create a multi-resource network of diagrams.
 - It also supports loading of documents from arbitrary URIs using a URI resolver. It allows to use logical URIs in links which are resolved to URLs at load time.
- Users can:
 - Create libraries of diagram elements and groups of elements (diagram fragments)
 - Reuse diagrams as templates

Figure 9: Element properties dialog

All of the above is free and open source!

Draw.io is great, but it is not enough! As I've mentioned above, I need to be able to associate a great deal of details with diagram elements starting with long formatted descriptions – something that exceeds tooltip capabilities.

reveal details during a presentation or use different levels of details with different audiences.

We may say that we have a "language" with six "parts of speech" - document, page, layer, node, connection, and tag.

However, this language has quite a large "vocabulary" of nodes and connections – dozens of styles and millions of images.

Elements of the Draw.io language represent/reference objects/concepts (by the definition of a diagram). A reference is established using graphics, text (labels and tooltips), links, and properties.

In the subsequent chapters we will see how to:

- Generate HTML documentation sites from diagrams
- Make diagrams executable by associating Java and script (e.g. Groovy) code with diagram elements
- Use properties to map diagram elements to elements of a domain (semantic) model expressed in EMF Ecore.

Non-Draw.io data sources

It is worth reiterating that generation of documentation, diagram execution and mapping to semantic models is implemented in a generic way in abstract classes with Draw.io specific classes extending those abstract classes and implementing abstract methods. If for some reason you want to use the above mentioned techniques with a different diagram format – you can make a copy of Draw.io specific classes and rewrite them to support your diagram format. This is what is you'd need to implement:

- Retrieval of a diagram element properties. In Draw.io element properties are used, but it is also possible to load properties from an external source such as a YAML/JSON/MS Excel file or a database.
- Retrieval of element parent, children, and source and target for connections.

- For documentation generation – generation of a representation to embed into documentation pages. Draw.io implementation embeds diagram pages. For some other format you may embed, say, PNG with an image map. Another option is a PlantUML or Mermaid diagram spec.

Essentially a data source for documentation generation etc. doesn't have to be a diagram at all! It can be a database schema or a cloud or Kubernetes environment. You may also implement a data source where some properties are loaded from a diagram and some from external sources.

Documentation sites

So far we've been talking about "bare" diagrams – they are quite powerful, but sooner than later they reach their communication capacity. In this chapter we'll take the first step beyond diagrams – generate a documentation site from a diagram.

Imagine you created a diagram or a set of diagrams to capture your idea or architecture. You pitched it to the powers that be and one of the three things happened:

- You've got an approval to proceed with your idea/architecture. Now you need to build it – work with multiple groups, delegate ownership of components for further elaboration, ...
- You've got an approval to proceed, but a critical dependency will not be available in the near future, so you need to put your effort on hold and resume at a later point in time
- Your idea was rejected, but you think it might be useful for other people or the circumstances may change in the future and you might get back to the idea.

In either case you need to communicate your idea – to other people in the first case and last case and to the future self in the last two cases. In the first case you'd also need to incorporate input of other people.

One way to do it is to document diagram elements and generate HTML documentation.

Demo

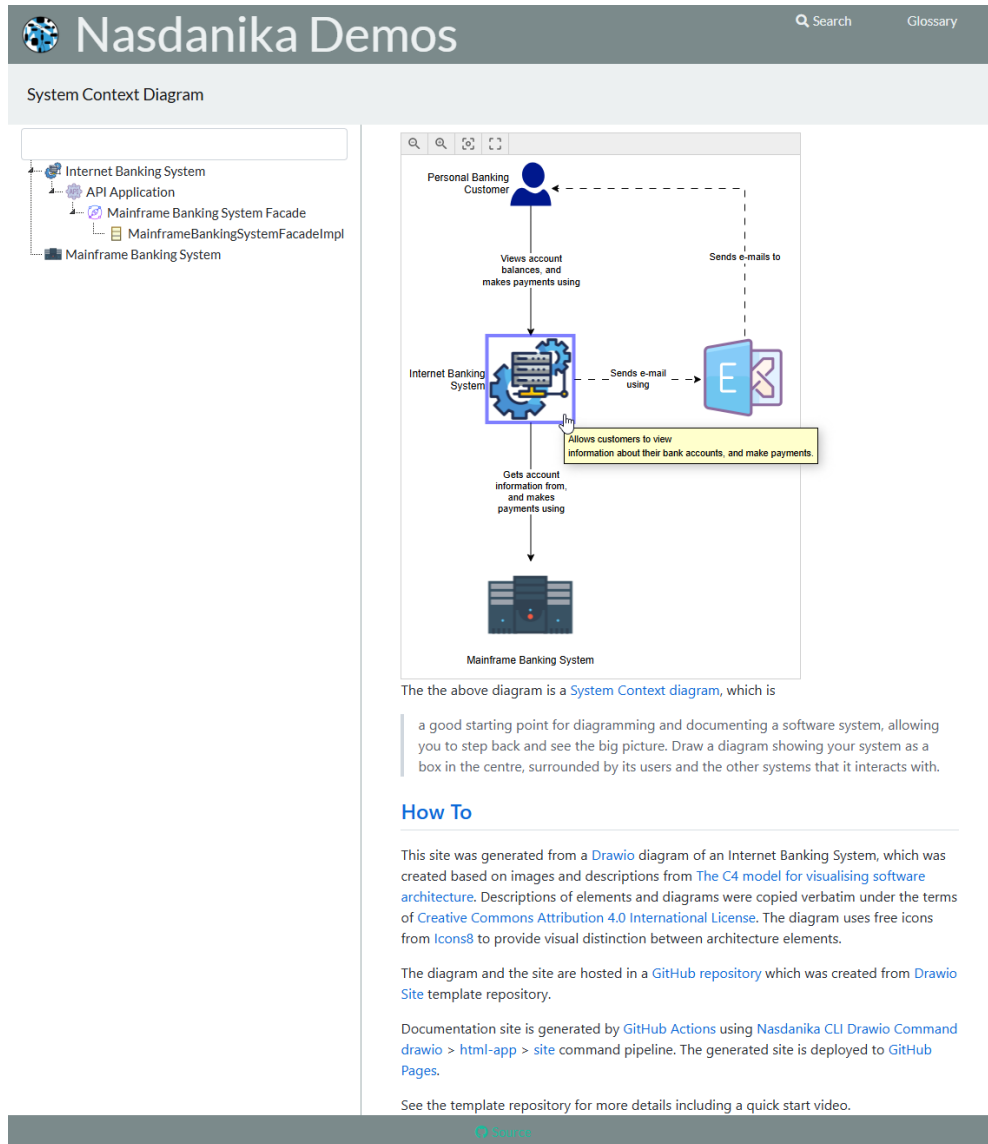


Figure 11: Internet Banking System – System Context Diagram

First we will take a look at the Internet Banking System demo⁹. Its home page is shown on Figure 11.

The diagram file has 4 pages:

[9https://bit.ly/nsd-ibs-drawio](https://bit.ly/nsd-ibs-drawio)

- System Context Diagram
- Container Diagram
- API Application Component Diagram
- Mainframe Banking System Diagram

System Context Diagram

On this diagram all elements have tooltips. “Internet Banking System” is linked to the “Container Diagram” page and has doc-ref property referencing a documentation file:

Because “Internet Banking System” is linked to the “Container Diagram” page, the container diagram is automatically added to the “Internet Banking System” documentation page.

“Send e-mail using” links also has doc-ref property referencing a documentation file.

“Mainframe Banking System” documentation is provided in the documentation property.

So, out of 8 diagram elements (4 nodes and 4 connections) on the page 3 are documented and get HTML documentation pages generated for them. The documentation pages are shown in the left panel. Also, documented diagram elements are linked to their documentation pages – clicking one an element navigates to its documentation page.

With this approach you can document what matters (now) and leave the rest for a later time, for other people or for never – what you have might be good enough AS-IS.

Figure 12 shows the properties dialog for the 'Internet Banking System'. The fields are:

- ID: internet-banking-system
- doc-ref: internet-banking-system.md
- link: data:page/id,fufg5MHGldTB2NhlnXvk
- tooltip: Allows customers to view information about their bank accounts, and make payments.

Buttons: Enter Property Name, Add Property, Placeholders (unchecked), Cancel, Export, Apply.

Figure 12: Internet Banking System properties

Figure 13 shows the properties dialog for the 'Mainframe Banking System'. The fields are:

- ID: mainframe-banking-system
- documentation: Stores all of the core banking information about customers, accounts, transactions, etc.
- tooltip: (empty)

Buttons: Enter Property Name, Add Property, Placeholders (unchecked), Cancel, Export, Apply.

Figure 13: Mainframe Banking System properties

If you don't document anything you'll get a page with the diagram – a good way to share your diagrams – virtually everybody has a web browser, but not everybody has a Draw.io editor. Illustrations to this book are published using this approach, a good deal of them are undocumented.

Container Diagram

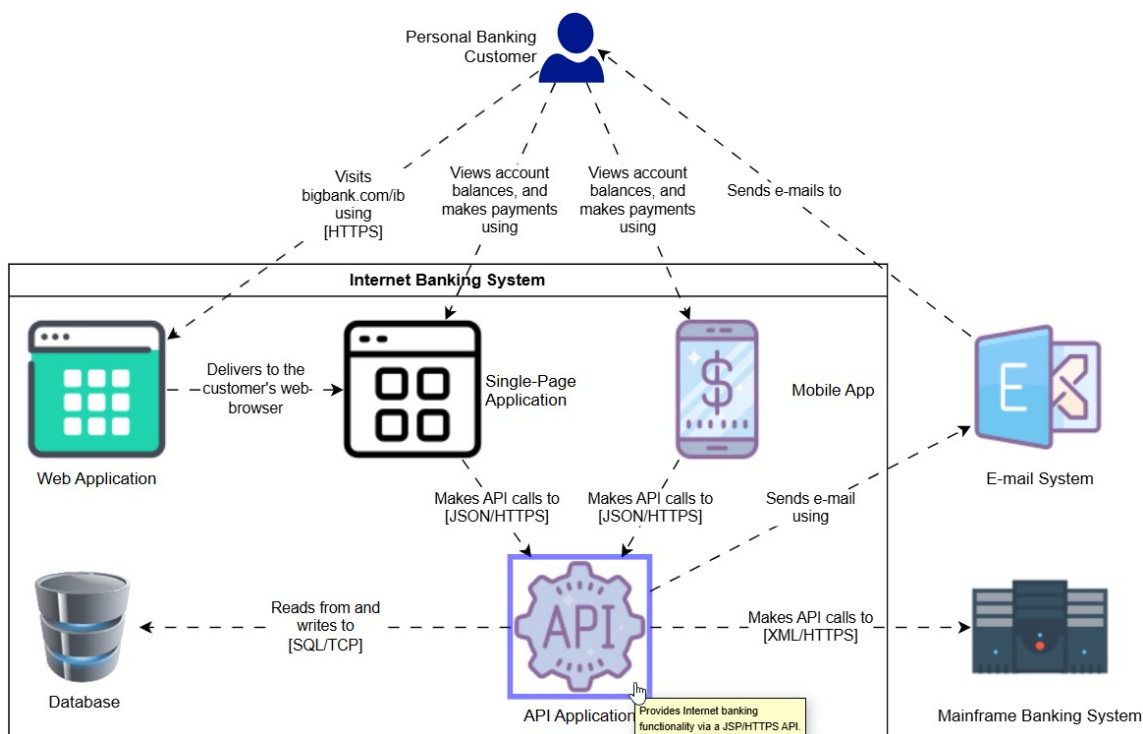


Figure 14: Container Diagram

One the Container Diagram the “Internet Banking System” container is linked to the “Internet Banking System” node on the “System Context Diagram” using the extended link syntax using page and element IDs:

`data:element/id,id,Ht1M8jgEwFfnCIfoTk4-/internet-banking-system`. Similarly, the “Mainframe Banking System” node is linked to the “Mainframe Banking System” node on the “System Context Diagram”, but it uses page name and element ID:
`data:element/id,name,System+Context+Diagram/mainframe-banking-system`.

“API Application” is documented and linked to the “API Application Component Diagram” page. As a result, “API Application” documentation page contains both the “API Application Component Diagram” and documentation.

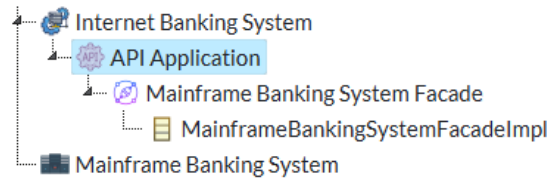


Figure 15: Page hierarchy

Because “API Application” is contained by “Internet Banking System”, its documentation page becomes a child of the “Internet Banking System” page:

On Figure 16 you can see that `doc-ref` property value is `%id%.md`. Because “Placeholders” checkbox on, `%id%` get expanded to `api-application`, so the final name of the documentation resource is `api-application.md`.

ID: api-application
 doc-ref: %id%.md
 link: data:page/id,OfHEa-7rySiIrvDJSLU1
 tooltip: Provides Internet banking functionality via a JSP/HTTPS API.

Enter Property Name Add Property

☒ Placeholders Cancel Export Apply

Figure 16: API Application properties

API Application Component Diagram

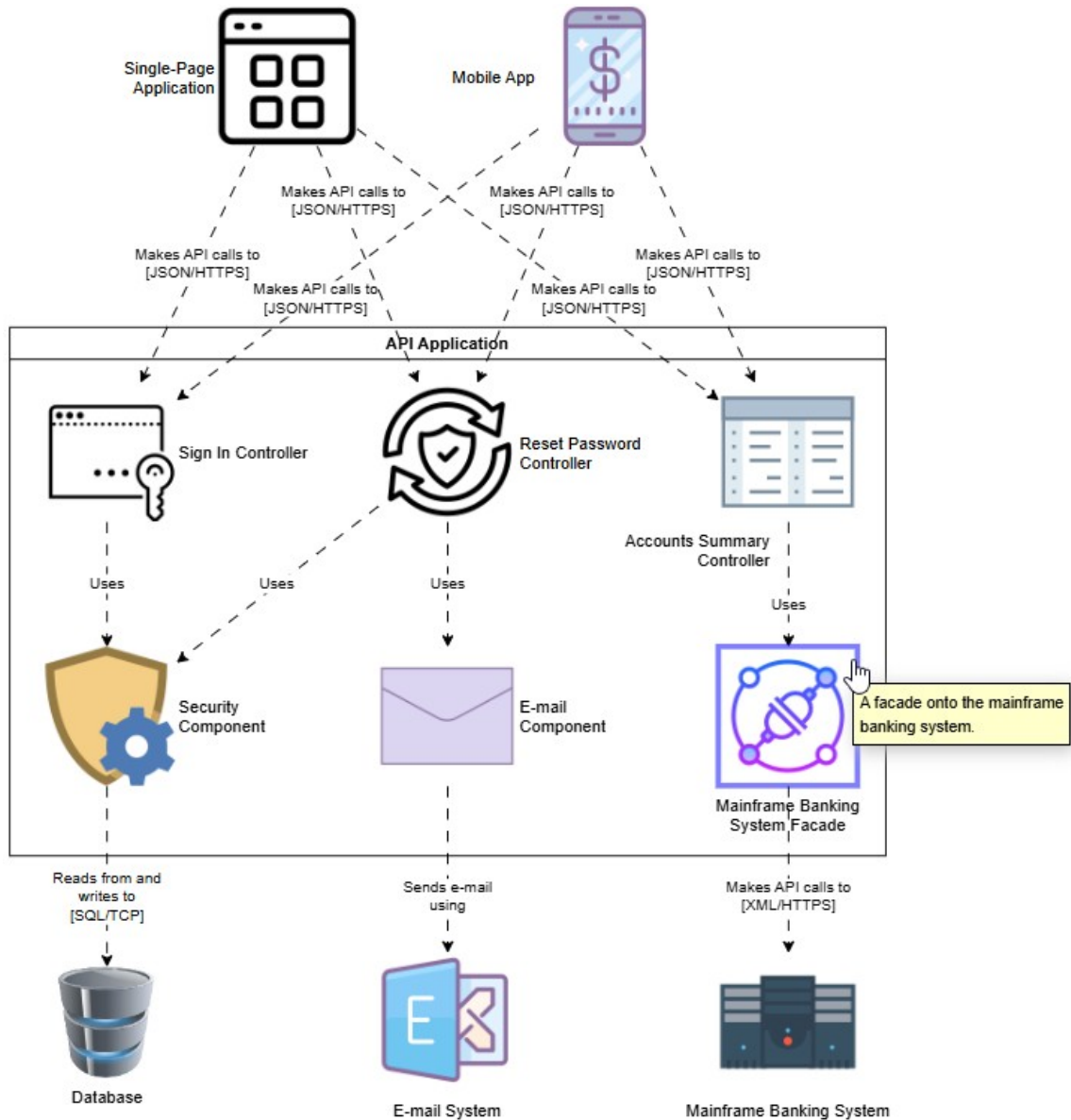


Figure 17: Component Diagram

The component diagram is structured similarly to the container diagram.

- “Mainframe Banking System” is linked to the “Mainframe Banking System” on the system diagram.
- “API Application” container is linked to “API Application” on the container diagram.
- “Mainframe Banking System Facade” is linked to “Mainframe Banking Facade Code” page.
- The rest of the diagram elements have tooltips.

Mainframe Banking System Code Diagram

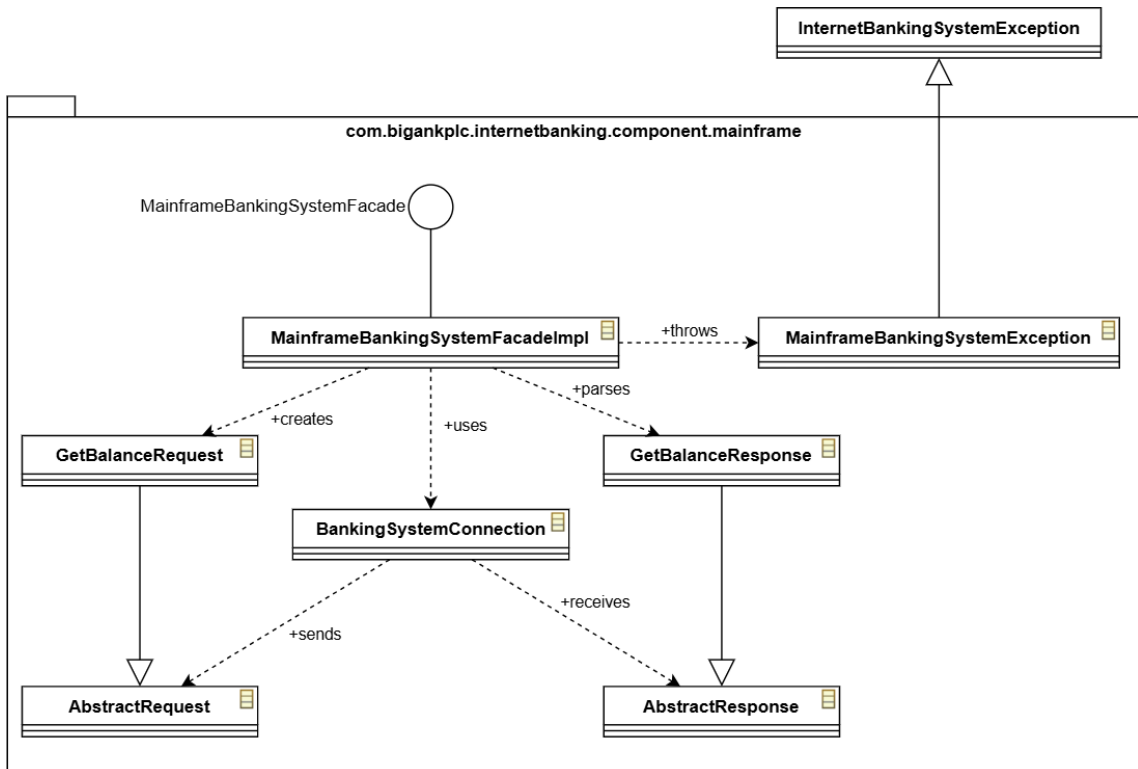


Figure 18: Code Diagram

On the code diagram the package container is linked to “Mainframe Banking System Facade” on the component diagram. **MainframeBankingSystemFacadeImpl** class is

documented in a markdown file. The rest of the diagram elements are neither documented nor linked.

MainframeBankingSystemFacadeImpl Documentation

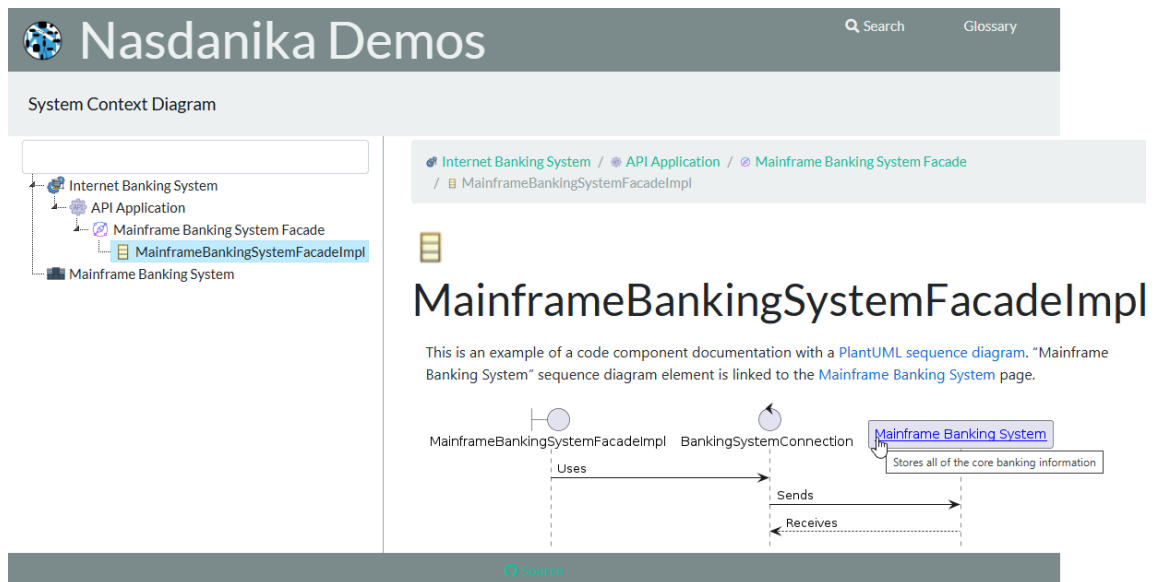


Figure 19: *MainframeBankingSystemFacadeImpl* documentation

`MainframeBankingSystemFacadeImpl` Markdown documentation features a PlantUML sequence diagram. The “Mainframe Banking System” participant has a tooltip and is linked to “Mainframe Banking System” documentation page.

Below is the sequence diagram UML fenced block:

```
```uml
hide footnote

boundary MainframeBankingSystemFacadeImpl
control BankingSystemConnection

participant "[[.././.././../mainframe-banking-system/index.html{Stores all of the core banking information} Mainframe Banking System]]" as MBS
```

```

MainframeBankingSystemFacadeImpl -> BankingSystemConnection: Uses
BankingSystemConnection -> MBS : Sends
BankingSystemConnection <-- MBS : Receives
...

```

## Generation

In the previous chapter we took a brief look at what documentation site generation is and how it works. In this and the next chapters we will take a more detailed look at the “How” part. Detailed documentation is available at the Draw.io site template repository<sup>10</sup>.

## GitHub Actions

The easiest way to start with generating documentation from Draw.io diagrams is to use GitHub actions. Create a new repository from the “Draw.io site” template repository mentioned above. Follow instructions on the template repository site.

## Nasdanika CLI

If you want to generate sites locally – use Nasdanika CLI<sup>11</sup>  
`drawio > html-app > site command12`

To generate a site with a single top-level diagram:

```

nsd drawio <diagram file> html-app -r <root action>
--add-to-root site -r=-1 -F <page template> <output
dir>

```

If you have more than one top-level diagram:

```

nsd drawio <diagram file> html-app -r <root-action>
site -r=-1 -F <page-template> <output dir>

```

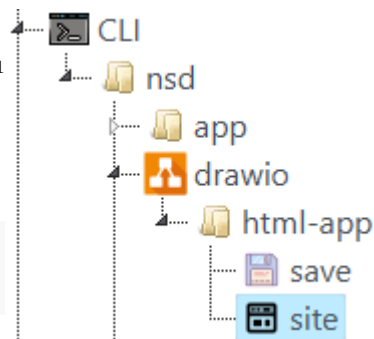


Figure 20: Site command

<sup>10</sup> <https://bit.ly/nsd-drawio-site>

<sup>11</sup> <https://bit.ly/nsd-cli>

<sup>12</sup> <https://bit.ly/nsd-cli-drawio-site>

## Java

To generate a site programmatically you can build the command hierarchy and the command line programmatically and then pass the command line to the root command. You can copy `Launcher`<sup>13</sup> class `main` method and add programmatic arguments construction.

You can also combine the code of `drawio`, `html-app`, and `site` commands into a method or class – this is a cleaner approach, but more involved.

## Configuration

This chapter is a quick reference of diagram elements configuration properties. There are the following properties:

- `documentation` – documentation text in documentation format
- `doc-format` – explicitly specified documentation format for `documentation` and `doc-ref`: `markdown` (default), `html`, or `text`.
- `doc-ref` – URL of a documentation resource resolved relative to the URL of the diagram file. Documentation format is derived from the URL extension, defaulting to `markdown`. Use `doc-format` to override.
- `icon` – diagram element icon URL resolved relative to the diagram file. If there is no slash (/) in the icon name then it is treated as a CSS style, e.g. `fas fa-user`. For image diagram elements icons are derived from element images as it is done in the demo. It is recommended to use SVG 20x20 pixels for icons because are also used in page titles and PNG images get blurry when scaled up.
- `parent` – Connection property with values `source` or `target`. Use to generate documentation from mind maps where parent/child relationship is defined by connections, not by containment.
- `prototype` & `proto-ref` – With prototypes you can:
  - Generate complex site pages (actions) with children, navigation, sections, ...

---

<sup>13</sup> <https://bit.ly/nsd-cli-launcher>

- Reuse action models. For example, generate an action model from one diagram and use it as a prototype for an element of another diagram. Or generate an action model for CLI or Ecore documentation.
- **role** – action/page role<sup>14</sup>:
  - **anonymous** (default for connections)
  - **child** (default for nodes)
  - **navigation**
  - **section**
- **sort-key** – By default generated pages (actions) are sorted alphabetically by title. This property can be used to customize sorting. If it is set then pages are sorted first by the property value and then by page title.
- **title** – By default the element label is used as page title (action text). Use this property to explicitly set the page title. For example, for elements with long labels.

## Search

Sites generated from diagrams provide 3 search facilities, all of them use information extracted from diagrams.

### ***Navigation panel filter***

Navigation panel filter uses page text, diagram element labels and tooltips to match text in the search box.

Figure 21 shows filtering by the word **mobile**. On the matching pages this word appears only on diagrams.



*Figure 21: Navigation panel filter*

---

<sup>14</sup> More about roles in the HTML Application model chapter

## Search page

The search page shown on Figure 22 provides the same search functionality as the navigation panel filter – finding pages by searching full page text and diagram text. It

**Nasdanika Demos** Search Glossary

System Context Diagram

Internet Banking System  
Mainframe Banking System

**Search**

Filter: mobile

You can use wildcards, e.g. 'foo\*' or '\*foo'; title or content fields, e.g. 'title:foo' bar'; boosts, e.g. 'foo\*10 bar'; fuzzy matches, e.g. 'foo~1'; and term presence, e.g. '+foo bar -baz'

Nasdanika Demos  
Children  
System Context Diagram  
Children  
Internet Banking System  
Children  
API Application

**Internet Banking System**

... via their web browser. Makes API calls to [JSON/HTTPS] **Mobile** App Provides a limited subset of the Internet banking functionality to customers via their **mobile** device. Reads from and writes to [SQL/TCP] API Application ...

**API Application**

... API calls to [JSON/HTTPS] Makes API calls to [JSON/HTTPS] **Mobile** App Reads from and writes to [SQL/TCP] Sends e-mail ...

Internet Banking System/API Application

Source


*Figure 22: Search page*

displays results in cards with text fragments around matches with matched text highlighted. It also highlights matched pages in the site content tree. Unlike the left navigation tree, the site content tree contains all pages including navigation and anonymous<sup>15</sup>.

<sup>15</sup> See “HTML Application Model” chapter for more information about page (action) roles.

## Glossary



Glossary is an alphabetically ordered searchable table with page icons, names, and descriptions (diagram elements tooltips). Filtering is done by page name and description.



# Nasdanika Demos

[Search](#)
[Glossary](#)

System Context Diagram






 Internet Banking System
  Mainframe Banking System


## Glossary

Filter
[Clear](#)

You can use wildcards, e.g. 'foo\*' or '\*o'; title or content fields, e.g. 'title:foo' bar'; boosts, e.g. 'foo^10 bar'; fuzzy matches, e.g. 'foo~1'; and term presence, e.g. '+foo bar -baz'

☐ Identifier(s)
 ☐ Hide UUID

Element	Description
 API Application	Provides Internet banking functionality via a JSP/HTTPS API.
<a href="#">Glossary</a>	
 Internet Banking System	Allows customers to view information about their bank accounts, and make payments.
 Mainframe Banking System	
 Mainframe Banking System Facade	A facade onto the mainframe banking system.
 MainframeBankingSystemFacadeImpl	



Nasdanika

Demos

[Search](#)

[→ Sends e-mail using](#)

[Source](#)

[System Context Diagram](#)

[Source](#)

Figure 23: Glossary



## Summary

Generation of HTML documentation from Draw.io diagrams aims to improve efficiency and effectiveness of human communication- with your present and future self and others:

- **Efficiency** – less time spent by diagram authors to communicate with diagram consumers. For example, making diagrams self-service and not having to explain meaning of diagrams elements over and over again.
- **Effectiveness** – less time spent by diagram consumers to get to information they need:
  - *Self-service* again – no need to contact diagram authors for explanations.
  - *Interactive diagrams* – hover mouse over for tooltips, click to navigate to details.
  - *Search* – find what you need including text in diagrams.

It was designed to be:

- **Simple** – just 7 configuration properties, you may use one most of the time,
- **Robust** – generation does not fail on incorrect configuration. Instead, it reports errors on generated pages.
- **Scalable** – sites can be generated from multiple cross-referencing diagram files, including circular references. Prototypes can be used to link (mount) pages (actions) which were either manually created or generated – from other diagrams, models, source code, ...

In the subsequent chapters we will see how to use diagrams to communicate with computers – make diagrams executable. In the remainder of this chapter we will use the Internet Banking System demo to take a look at how to scale human-to-human communications with diagrams and sites generated from them.

The Internet Banking System demo shows only one vertical of diagrams: Internet Banking System > API Application > Mainframe Banking System Facade >

MainframeBankingSystemFacadeImpl. In a fully elaborated documentation there would be many more diagrams and pages. For example, The Internet Banking System has 5 containers and API Application has 6 components. Maintaining diagrams for all these elements in a single file will most likely lead to diagram editing becoming a bottleneck if done sequentially (exclusive edits), or edit and merge conflicts if done concurrently.

In addition to this, low-level documentation such as class and sequence diagrams and database schema documentation, can be generated from sources – maintaining it manually is tedious and will likely to lead to outdated documentation.

Let's assume that there are three teams maintaining four types of diagrams in three repositories as shown on Figure 24.

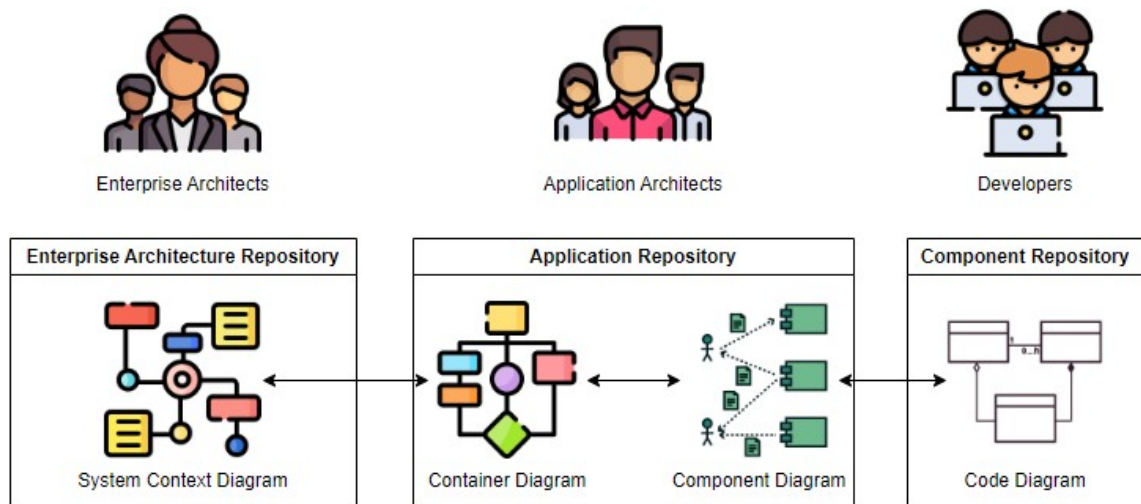


Figure 24: Federation

Information from the diagrams can be federated into a holistic body of architectural knowledge in the following ways:

- Sites
- Action models
- Diagrams

## Sites

In this scenario each team generates and publishes documentation sites and then cross-reference them.

The enterprise architects may publish a site with high level documentation of diagram elements. When the application architects publish more detailed documentation, the enterprise architects may add a link to the site with detailed documentation to their high level documentation or they may remove their high level documentation and link a diagram element to the detailed documentation.

At the same time, application architects can link the header of the Container Diagram site to the System Context Diagram site. This would allow bi-directional navigation between sites.

Each site would have its own navigation tree and search – it might be desired in some situations, but can be a drawback in some others. For example, you might keep sites with a lot of low-level documentation such as code diagrams and code documentation separately from the higher-level sites. At the same time you may want to have System, Container, and Component diagrams in one site so you can search through them and quickly find elements in the Glossary.

## Action models

You can generate action models instead of sites from diagrams and then use them as prototypes. For example, application architects would generate a container diagram action model and share it with the enterprise architects. They may publish action models to a Maven repository or to a web site. The enterprise architects would update their diagrams with `proto-ref` properties pointing to locations of action models published by the application architects. This way information from both system and container diagrams will be published to the site. This approach can be extended to component and code diagrams.

A drawback of this approach is that cross-references between elements would have to be maintained manually. Also, action models might have to be processed to include relative resources.

The advantage of this approach is that you may use action models generated not only from diagrams, but from other sources as well.

## Diagrams

Diagrams can be federated using the extended link syntax which supports cross-document linking. Diagrams stored in GitHub can be referenced using their JSDelivr URLs. Diagrams in GitLab can be cross-referenced using GitLab URIs `gitlab://<project>/<ref>/<path>` supported by `GitLabURIHandler`<sup>16</sup>.

You can also keep and edit diagrams on Confluence and load them using Confluence REST API. This way high-level information can be provided on Confluence pages with detailed information in a version control system. This can be a good starting point if you already have your diagrams in Confluence. For this approach to work a URI handler would be needed to make calls to Confluence with an authorization HTTP header. While this functionality is not available yet, it is easy to add once needed.

Logical URIs with URL to URL mapping and property placeholders (e.g. `%env%`) can be used in cross-links to link to different elements for different environments or versions.

## Diagramming ecosystem

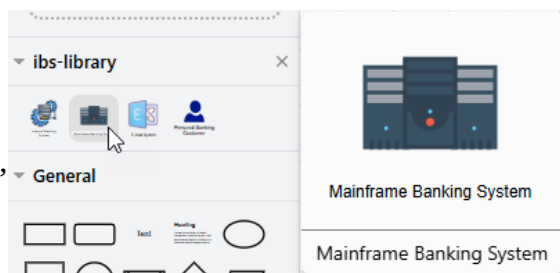
You can also establish a diagramming ecosystem<sup>17</sup> with libraries of reusable preconfigured diagram elements with names, tooltips, links to documentation pages.

Figure 25 shows a demo library for the

Internet Banking System. The library URL:

<https://nasdanika-demos.github.io/internet-banking-system/ibs-library.xml>.

More about a diagramming ecosystem later in the book.



*Figure 25: Internet Banking System Library*

<sup>16</sup> <https://bit.ly/nsd-gitlab-uri-handler>

<sup>17</sup> <https://bit.ly/nsd-diagramming-ecosystem>

## Semantic search & Chat

Generated sites have `search-documents.js` file used for search. This file contains a map of site pages URLs to page information, which includes page content in plain text. Page content includes documentation and text generated from diagrams – element labels and tooltips.

`search-documents.js` file can be used to build semantic search or chat. Imagine being able to have a conversation with your architecture site! Chat can be made contextual. For example, chatting with the Mainframe Banking System would use information more relevant to the Mainframe Banking System.

If published sites are versioned, then chats would also be version-specific. It might be important in situations when different systems and different releases of a particular system use different versions of a specific component, say Mainframe Banking System Facade.

## Markdown

This chapter explains advanced capabilities of Markdown documentation.

### Embedded images

You can embed PNG and JPEG using fenced blocks.

#### PNG resource

```
```png-resource
isa.png
```
```

Resource location is resolved relative to the diagram file location.

#### JPEG resource

```
```jpeg-resource
my.jpeg
```
```

#### PNG

```
```png
<Base 64 encoded png>
```
```

#### JPEG

```
```jpeg
<Base 64 encoded jpeg>
```
```

## Embedded diagrams

You can also embed PlantUML, Draw.io, and Mermaid diagrams using fenced blocks.

### Drawio

```
```drawio-resource
aws.drawio
```
```

Resource location is resolved in the same way as for image files as explained above.

### PlantUML

PlantUML diagrams can be defined inline or loaded from resources.

#### *Loading from a resource*

```
```uml-resource
sequence.plantuml
```
```

### *Inline*

The following language specifications (dialects) are supported:

- **uml** - for the following diagram types:
  - Sequence
  - Use Case
  - Class
  - Activity
  - Component
  - State

- Object
- Deployment
- Timing
- Network
- **wireframe**
- **gantt**
- **mindmap**
- **wbs**

Below are two examples of PlantUML diagrams.

## Sequence

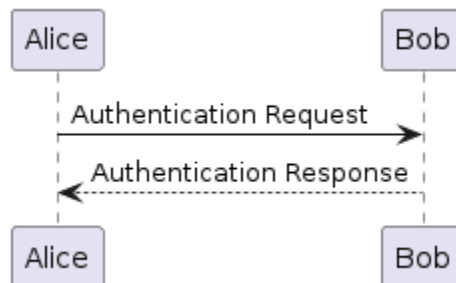
Fenced block:

```

```uml
Alice -> Bob: Authentication Request
Bob --> Alice: Authentication Response
```

```

Diagram:



*Figure 26: PlantUML Sequence Diagram*



## Component

A component diagram with links to component pages.

Fenced block:

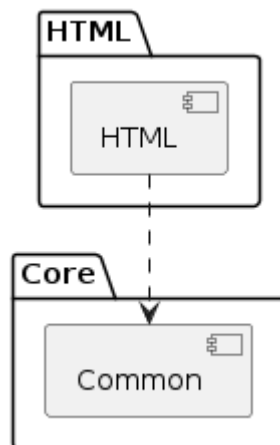
```

```uml
package Core {
    component Common [[https://github.com/Nasdanika/core/tree/master/common]]
}

package HTML {
    component HTML as html
    [[https://github.com/Nasdanika/html/tree/master/html]]
    [html] ..> [Common]
}
```

```

Diagram:



*Figure 27: PlantUML Component Diagram*

## Mermaid

You can define Mermaid diagrams in mermaid fenced blocks:

```
```mermaid
flowchart LR
    Alice --> Bob & Chuck --> Deb
```
```

results in a diagram shown on Figure 28

It is also possible to load a diagram definition from a resource resolved relative to the diagram resource:

```
```mermaid-resource
sequence.mermaid
```
```

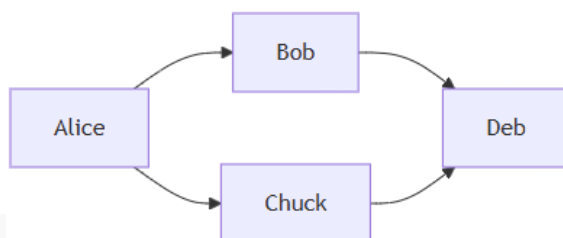


Figure 28: Mermaid Flowchart Diagram

## Extensions

Below are a few useful extensions<sup>18</sup>:

- **Table of contents:** add ``[TOC]`` to the document as explained in the documentation. This extension will create a table of contents from markdown headers.
- **Footnotes**
- **Strikethrough:** ~~strikethrough~~
- **Subscript:** H<sub>2</sub>O
- **Superscript:** 2<sup>5</sup> = 32

<sup>18</sup> <https://bit.ly/nsd-markdown-extensions>

## Full book overview

This is a sample book. It contains explains how to generate documentation sites from diagrams with minimal effort and without any programming experience.

The remainder of the book focuses on more advanced concepts most of which require Java development experience:

- **Executable diagrams** chapter explains how to make diagrams executable and use them from Java code as regular Java objects using dynamic proxies.
- **EMF Ecore** chapter provides a brief introduction into EMF Ecore used in semantic mapping.
- **Semantic mapping** chapter explains how to map diagrams to semantic (problem domain) models such as Architecture model, C4 Model, ...
- **Models** chapters introduce readers to a few models which can be used as semantic mapping targets:
  - Family
  - HTML Application
  - Graph
  - Architecture
  - C4 Modeling
  - Function flow
- **Custom model** chapter explains how you can create your own model based on pre-existing models of from scratch.
- **Diagramming ecosystem** chapters introduces participants, activities, and artifacts of the diagramming process.

- **GenAI** chapter sheds some light on how information stored in diagrams can be made available to GenAI and how GenAI can be integrated with documentation sites.
- There are several chapters covering important technologies on which capabilities explained in this book are built:
  - Capability framework
  - Nasdanika CLI
  - Invocable URIs
- Diagramming as explained in this book is done to improve effectiveness and efficiency of information exchange. Effectiveness focuses on reducing consumption cost and efficiency focuses on reducing production cost. **Economy of diagramming** chapter takes a closer look at different aspects of the costs and how to minimize them.

## Index

|                                       |    |
|---------------------------------------|----|
| Entity.....                           | 10 |
| Internet Banking System.....          | 23 |
| Code Diagram.....                     | 28 |
| Component Diagram.....                | 27 |
| Container Diagram.....                | 25 |
| Mainframe Banking System Diagram..... | 24 |
| System Context Diagram.....           | 24 |
| representation.....                   | 5  |
| Representation.....                   | 5  |

## Table of Figures

|                                                                  |    |
|------------------------------------------------------------------|----|
| Figure 1: Diagrams as a communication vehicle.....               | 7  |
| Figure 2: Operation.....                                         | 9  |
| Figure 3: Diagramming problem domain.....                        | 10 |
| Figure 4: Features.....                                          | 10 |
| Figure 5: Visual features.....                                   | 11 |
| Figure 6: Living beings.....                                     | 12 |
| Figure 7: Color visual feature.....                              | 13 |
| Figure 8: Draw.io supported storage.....                         | 15 |
| Figure 9: Element properties dialog.....                         | 16 |
| Figure 10: Draw.io metamodel.....                                | 17 |
| Figure 11: Internet Banking System – System Context Diagram..... | 23 |
| Figure 12: Internet Banking System properties.....               | 24 |
| Figure 13: Mainframe Banking System properties.....              | 24 |
| Figure 14: Container Diagram.....                                | 25 |
| Figure 15: Page hierarchy.....                                   | 26 |
| Figure 16: API Application properties.....                       | 26 |
| Figure 17: Component Diagram.....                                | 27 |
| Figure 18: Code Diagram.....                                     | 28 |
| Figure 19: MainframeBankingSystemFacadeImpl documentation.....   | 29 |
| Figure 20: Site command.....                                     | 30 |
| Figure 21: Navigation panel filter.....                          | 32 |
| Figure 22: Search page.....                                      | 33 |
| Figure 23: Glossary.....                                         | 34 |
| Figure 24: Federation.....                                       | 36 |
| Figure 25: Internet Banking System Library.....                  | 38 |
| Figure 26: PlantUML Sequence Diagram.....                        | 43 |
| Figure 27: PlantUML Component Diagram.....                       | 44 |
| Figure 28: Mermaid Flowchart Diagram.....                        | 45 |