

Introduction to IT Systems for Beginners

Maciej Gowin @ [CoderBrother](#)

All rights reserved

API (Application Programming Interface)

Acts as interface for communication which enabled different systems or applications to communicate with each other.

For example, when you use Ryanair app, it communicates with web services which serves data via an API to loads a list of available routes.

API (Application Programming Interface)

The term API can be used in the context of various communication channels:

- Web API
- Library API
- Operating System API

API evolution

- API evolves and may require changes
- API requires incremental changes

API example

```
GET https://api.coderbrother.pl/customers/abcd-1234
```

```
{  
  "id": "abcd-1234",  
  "name": "Maciej Gowin"  
}
```

API evolution

Consider changes:

- What if we want to add date of birth to an existing API?
- What if we want to split the name and return separated first and last name?

API backward compatibility

Add date of birth to the model - backward compatibility met

Additive changes are acceptable as long as they do not break existing contract.

```
GET https://api.coderbrother.pl/customers/abcd-1234
```

```
{  
  "id": "abcd-1234",  
  "name": "Maciej Gowin",  
  "dateOfBirth": "1986-11-21"  
}
```

API backward compatibility

Replace name with first and last name - backward compatibility not met

Breaking changes define new model.

```
GET https://api.coderbrother.pl/newCustomers/abcd-1234
```

```
{  
  "id": "abcd-1234",  
  "firstName": "Maciej",  
  "lastName": "Gowin"  
}
```

API backward compatibility

For incremental changes, we will implement API versioning when a new model is defined for an existing resource.

As an example:

```
GET https://api.coderbrother.pl/v1/customer/abcd-1234  
GET https://api.coderbrother.pl/v2/customer/abcd-1234
```

API versioning

URI Versioning

```
/v1/customer/abcd-1234
```

Query Parameter Versioning

```
/customer/abcd-1234?version=1
```

API versioning

Header Versioning

```
/customer/abcd-1234
```

With a header:

```
Accept: application/vnd.api.v1+json
```

Subdomain Versioning

```
v1.api.coderbrother.pl/customer/abcd-1234
```

```
v2.api.coderbrother.pl/customer/abcd-1234
```

Choosing a Versioning Strategy

Clarity	How easily can users understand the versioning scheme?
Flexibility	Does the versioning method allow for easy changes and updates?
Backward Compatibility	How well does the versioning approach support clients using older versions of the API?
Simplicity	Is the versioning method simple to implement and maintain?

When API Can Be Deprecated?

- different approach driven by the consumer: mobile vs web
- unused versions can be removed
- use of tracing tools to verify the API usage

API vs Sprints

The API must be defined before frontend work begins, which undermines the principle of delivering functionality from the ground up during sprints.

Incremental API

Backend ahead of Frontend

API implementation starts one sprint before.

Model agreement

The team agrees on the API model and works in accordance with it. Model can be defined using tools such as Swagger.

Mocking

Quick API mocks representing the model. Mocks are delivered by tools such as Mockoon or JSON Server.

APIs vs Agile Methodologies

API planning and versioning for incremental changes are crucial from a technical point of view.

However, there are other challenges to API work in the context of Agile methodologies.

API challenges

1. Defining Clear Requirements

Challenge

Translating business requirements into API user stories is tricky since APIs often don't interface directly with users.

Solution

Collaborate with product owners to create technical stories and clarify the business value.

API challenges

2. Incremental Delivery

Challenge

Delivering usable API increments each sprint is hard, especially when endpoints are interdependent.

Solution

Prioritize high-value endpoints and start with a minimal viable API (MVA).

API challenges

3. Versioning and Backward Compatibility

Challenge

Frequent changes can break existing clients, making backward compatibility a concern.

Solution

Use versioning strategies and plan for backward compatibility in every sprint.

API challenges

4. Managing Dependencies

Challenge

Coordinating between teams (frontend, backend) can delay progress.

Solution

Use API contracts and mock services to unblock teams.

API challenges

5. Non-functional Requirements (NFRs)

Challenge

Addressing performance, security, and scalability in short sprints can be tough.

Solution

Plan NFRs into sprint work and conduct performance and security tests regularly.

API challenges

6. Testing and Automation

Challenge

Thoroughly testing and automating API tests in each sprint is time-consuming.

Solution

Automate unit, integration, and performance tests using CI/CD pipelines.

API challenges

7. Documentation

Challenge

Keeping API documentation up-to-date in fast-paced sprints is difficult.

Solution

Use automated tools (Swagger) to generate documentation alongside development.

API challenges

8. Managing Technical Debt

Challenge

Rapid delivery can lead to technical debt, degrading API quality over time.

Solution

Allocate time in sprints for refactoring and technical improvements.

API challenges

9. Coordinating Release Cycles

Challenge

Releasing new API versions must align with consumers (frontend, mobile) to avoid disruptions.

Solution

Use feature flags and communicate with teams to synchronize releases.

API challenges

10. Changing Requirements

Challenge

Constant requirement changes can lead to design instability.

Solution

Use flexible API design patterns and maintain clear communication with stakeholders.

Design patterns

Design patterns are general, reusable solutions to common problems in software design.

They provide a **standardized approach** to solving recurring challenges, making code more robust, flexible, and maintainable.

Design patterns Key Characteristics

- **Proven Solutions:** Derived from best practices in software development.
- **Language Agnostic:** Can be applied in any programming language.
- **Categorized for Reusability:** Patterns are grouped based on the type of problem they solve.

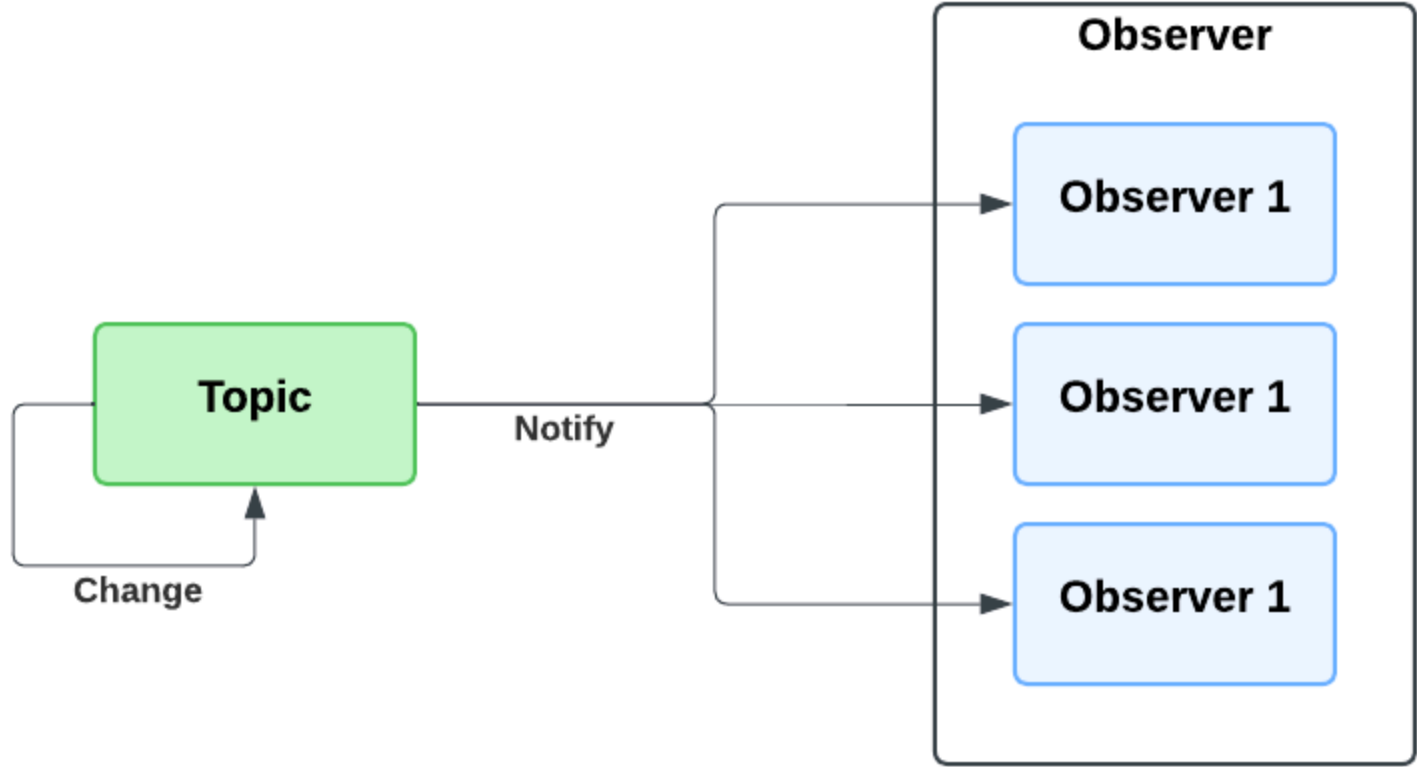
Design patterns example

Observer Pattern

Defines a subscription mechanism to notify multiple objects about any events occurring in the observed object.

Example:

A subject (e.g., news publisher) notifies observers (e.g., subscribers) whenever there is new content.



DRY (Don't Repeat Yourself)

The **DRY Principle** is a software development principle that **emphasizes reducing repetition** of code or logic throughout a system.

Ensures that every piece of knowledge or logic exists in a single, unambiguous place.

DRY Benefits

Improved Maintainability: Changes are easier to implement since code needs to be updated in only one location.

Reduced Errors: Decreases the likelihood of inconsistencies or bugs caused by duplicated code.

Enhanced Readability: Cleaner, more concise code that is easier to understand and navigate.

KISS Principle (Keep It Simple, Stupid)

The **KISS Principle** is a design philosophy that emphasizes simplicity in software development and problem-solving.

Keep systems simple and straightforward to minimize complexity and potential issues. Coined by the U.S. Navy in 1960, promoting the idea that most systems work best if they are kept simple and avoid unnecessary complexity.

KISS Principle Benefits

"Simplicity is the ultimate sophistication."

- **Easier Maintenance:** Simple code is easier to understand, maintain, and extend.
- **Reduced Errors:** Minimizing complexity lowers the chance of bugs.
- **Faster Development:** Streamlined, simple solutions reduce development time.

SOLID Principles of Object-Oriented Design

The **SOLID principles** are a set of five design principles intended to make software designs more understandable, flexible, and maintainable.

Acronym for five key principles:

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion

Version control

During software development, the source code undergoes constant changes. Additionally, the software development process is not linear, and often more than one person works on its development.

To streamline change management, version control systems are introduced to support the development process.

There are several version control systems available, such as:

- CVS
- SVN
- Mercurial
- Git

Git

Git is a Distributed Version Control System (DVCS) designed to **track changes** in source code during software development.

Key Features

- **Distributed:** Each copy of the repository contains the full history.
- **Branching:** Enables independent workstreams.

Git: Key terminology

Local directory/working space

A directory containing files where we make changes.

Staging area

A space where changes from the working directory are added before being committed to the repository as a revision. Essentially, it acts as a buffer for changes.

Git: Key terminology

Repository

A space that contains all files, the complete history of changes, and other data related to Git. Repositories are categorized as local (located on a specific workstation) or remote (external to the given workstation).

Revision

A revision (also known as a commit) is a single entry in the repository that contains all files along with specific changes. Each revision is assigned a unique identifier.

Git: Key terminology

Branch

A branch in Git is a separate environment that contains its own version of the code, including its own working directory state, staging area, and revision history. Every repository has at least one branch. The default branch is typically named master (or main in newer conventions).

Merge

In Git, merging is the process of combining changes from two different branches. The simplest form of merging is Fast Forward, where changes follow a linear progression without creating a new merge commit.

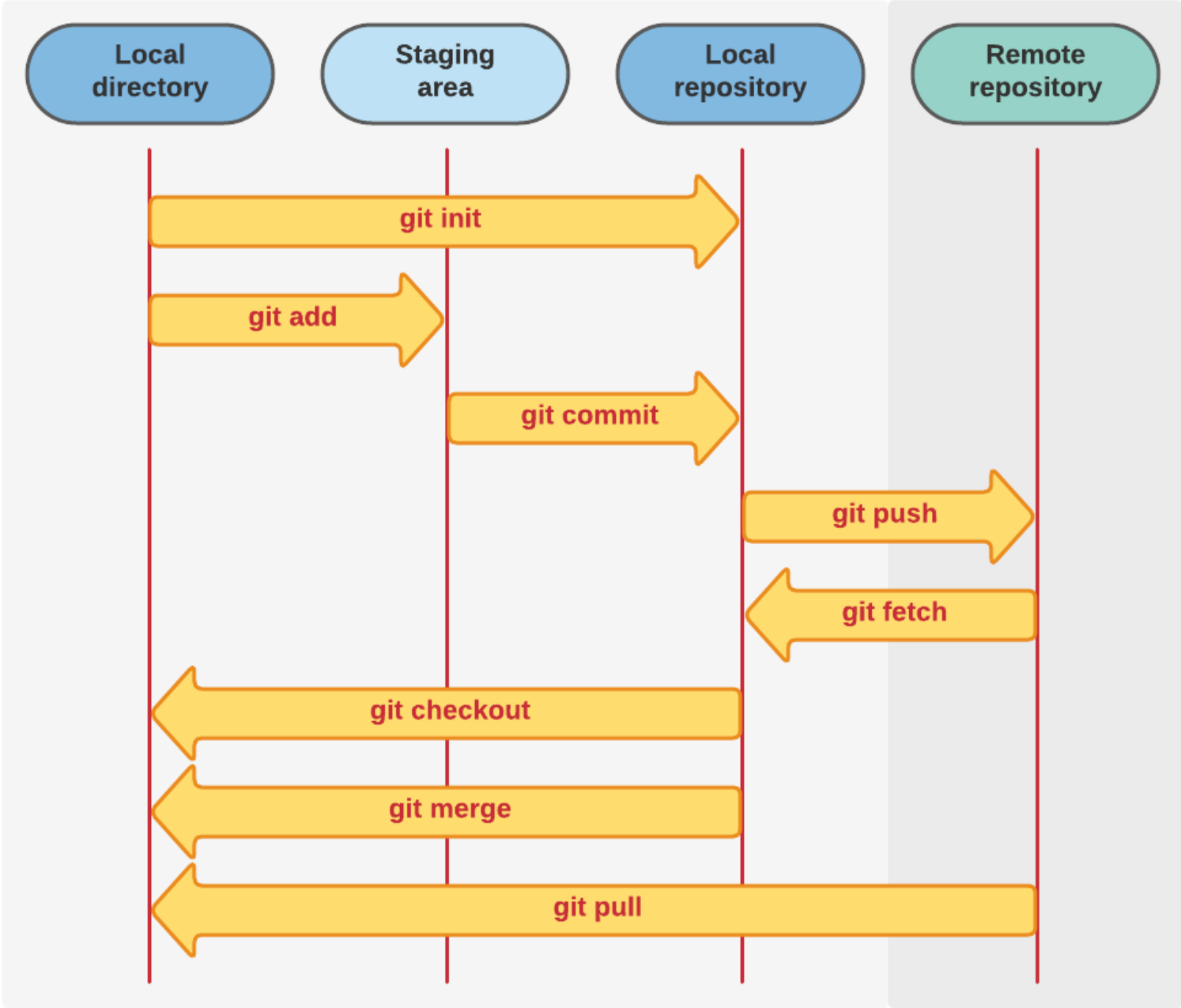
Git: Key terminology

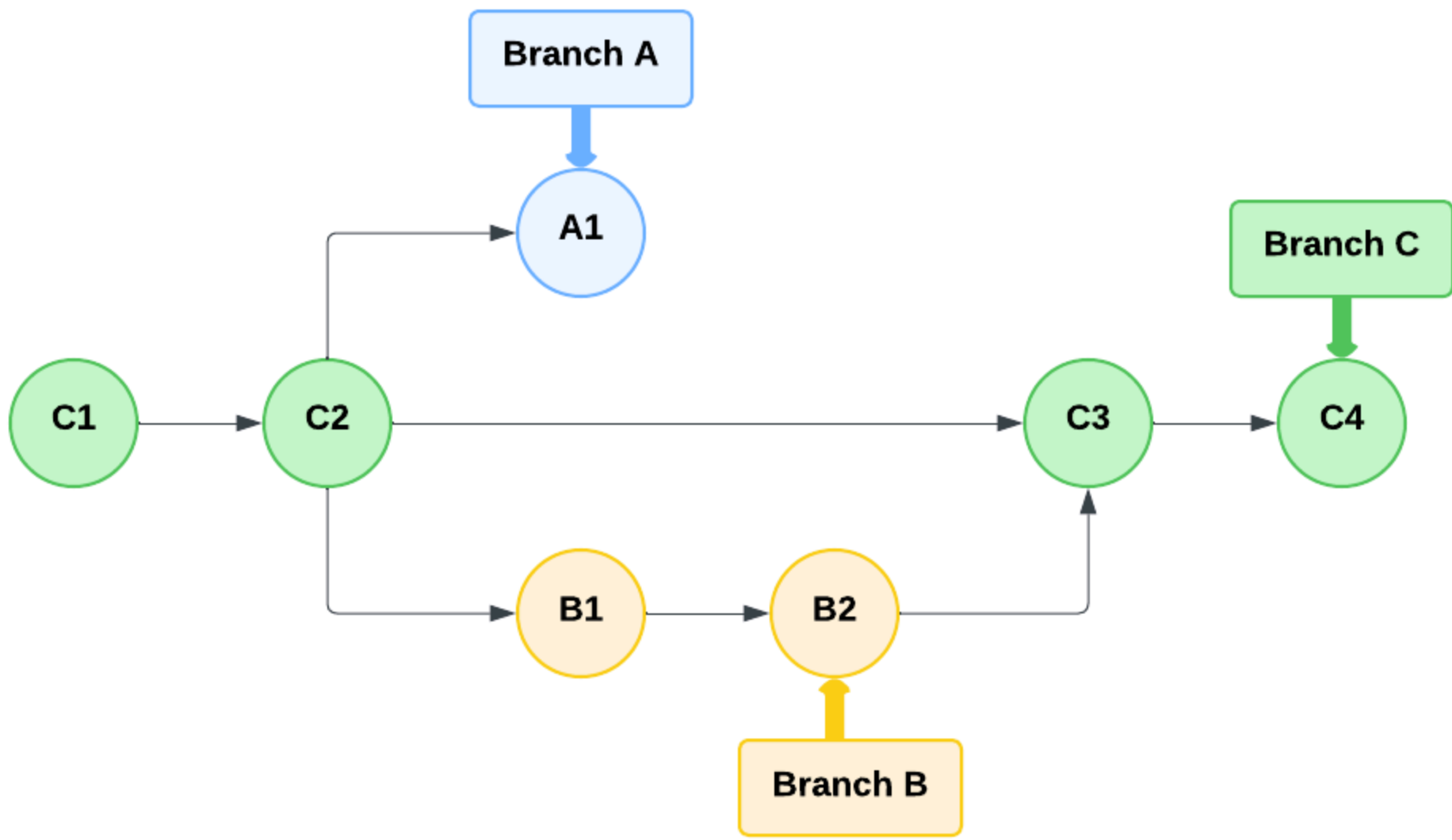
HEAD

In Git, HEAD is a pointer to the latest revision of the currently active branch. In other words, it indicates the current location within the repository's history.

origin

In Git, origin is the default name for a remote repository. It refers to the original version of the repository, typically the one from which the local repository was cloned.





Why do we use Git?

- **Collaboration:** Allows multiple developers to work on the same project.
- **Version History:** Tracks every change, enabling rollback if needed.
- **Efficiency:** Handles projects of any size with speed.
- **Flexibility:** Works for solo and team projects of any size
- **Integration:** Seamlessly integrates with tools like CI/CD pipelines, IDEs, and more.