Buildly Core Documentation

Buildly.io

Oct 14, 2022

Contents

buildly

Welcome to Buildly Core's documentation! Get started with the *Quickstart*. There is also a more detailed *Tutorial* that shows how to connect a service to Buildly Core. Microservice Architecture patterns are described in the *Patterns with Buildly* section. The rest of the docs describe each component of Buildly Core in detail.

Buildly Core depends on the *Docker* engine. The documentation for this service can be found at:

- Docker documentation https://docs.docker.com/

This technical communication document, is intended to give assistance to developers using a Buildly Core.

CHAPTER 1

Quickstart

This is an introduction to Buildly's Core.

1.1 Prerequisites

Docker version 19+

1.2 Buildly Core repo

Fork or clone the repository

https://github.com/buildlyio/buildly-core

1.3 Setting up Buildly Core

Make sure you have docker up and running then build the image:

```
docker-compose build # --no-cache to force dependencies installation
```

Next run the web server:

```
docker-compose up # -d for detached
```

Access the web server at http://127.0.0.1:8080

User: *admin* Password: *admin*.

To run the web server with Python debugger support:

```
docker-compose run --rm --service-ports buildly
```

1.4 Configuring the API authentication

All clients will interact with our API using the OAuth2 protocol, in order to configure it, go to *admin/oauth2_provider/application/* and add your new application there.

1.5 Generating RSA keys

To use JSON Web Token as the authentication method, you will need to configure public and private RSA keys.

To generate the public and private keys run the following commands:

```
openssl genrsa -out private.pem 2048
openssl rsa -in private.pem -outform PEM -pubout -out public.pem
```

The private key will stay in Buildly and the public one will be supplied to your microservices in order to verify the authenticity of the message.

1.6 Running the tests

To run the tests (without flake8) and have Python debugger open on error:

```
docker-compose run --entrypoint '/usr/bin/env' --rm buildly bash scripts/run-tests.sh␣
↪--keepdb
```

To run the tests with flake8:

```
docker-compose run --entrypoint '/usr/bin/env' --rm buildly bash scripts/run-tests.sh␣
↪--ci
```

For more tesing options enter:

```
pytest --help
```

CHAPTER 2

Tutorial

2.1 Connect your service to Buildly

2.1.1 Overview

This tutorial explains how to connect an existing service to Buildly.

Once you connect your service to Buildly, it will be able to communicate with all of your other services over a core authentication layer. All of its endpoints will be exposed as part of a single API that Buildly puts together from all of the services. You also have the option to use Buildly for managing permissions and users.

2.1.2 Requirements

There are no requirements for the language or framework used to code your service. It must only satisfy these conditions in order to connect to Buildly:

1. Your service must follow the OpenAPI (Swagger) spec.
2. You need to expose a *swagger.json* file at the */docs* endpoint.
3. Your service must use an OAuth2 library with support for JSON Web Tokens (JWTs). See the *Implement JWT authorization* section for more information.

2.1.3 Implement JWT authorization

Next, you need to implement Buildly's authorization method.

About Buildly authorization

For external requests to modules (e.g., Buildly UI users), Buildly uses an OAuth2 flow to issue JSON Web Tokens (JWTs) signed with RS256.

Buildly's **public key** should be exposed as the environment variable *JWT_PUBLIC_KEY_RSA_BUILDLY* inside the container where the service is deployed. The service must use this environment variable to decode requests from Buildly.

Buildly passes the JWT to the service in the *Authorization* HTTP header using the format *JWT {token}*. Example:

```
Authorization: JWT eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.
↪eyJpc3MiOiJiaWZyb3N0IiwiZXhwIjoxNTYwNjA0OTc2LCJpYXQiOjE1NjA1MTg1NzYsImNvcmVfdXNlcl91dWlkIjoiODJiZG
↪CV8PafWuGDZSpWRI5wC6btO6cyt9udI9P5uLBdnHzVhbbIY-
↪LH1o3qBgnRf0OAreUhRfl7zBTBMNO56pbyWeyg
```

Example using PyJwt

Here's an example using the PyJwt library. It takes the *encoded_jwt* from the HTTP header and decodes it with the *JWT_PUBLIC_KEY_RSA_BUILDLY* environment variable:

```
import jwt

jwt.decode(encoded_jwt, os.environ['JWT_PUBLIC_KEY_RSA_BUILDLY'], algorithms=['RS256
↪'])
```

The Buildly JWT payload looks like this:

- *core_user_uuid*: UUID of the [CoreUser](/model/permissions#coreuser) who initiated the request.
- *exp*: Datetime when the token expires.
- *iat*: Datetime when the token was issued.
- *iss*: Issuer of the JWT. This will always be *buildly*.
- *organization_uuid*: UUID of the [Organization](/model/permissions#organization) that contains the CoreUser who initiated the request.
- *scope*: Permission scopes granted in the request by Buildly.
- *username*: The username of the CoreUser who initiated the request.

2.1.4 Add Your Service to Buildly

After finishing the JWT authorization implementation, deploying your service somewhere and exposing it externally, you need to add it to Buildly.

To add your service to Buildly, make sure it meets the prerequisites, then navigate to the Buildly admin page at *https://<YOUR-BUILDLY-URL>/admin* and log into it. Then, under the section **Core** you will see **Logic Module**, click on it and add a new one with the following properties:

- Name: The name of your service for identification
- Endpoint: The host of your microservice e.g. https://my-amazing-service.com
- Endpoint Name: The API endpoint for your service (just type the name of the endpoint, for example, *amazing*

Now, your service will be accessed by Buildly and exposed following the structure *https://<BUILDLY-URL>/api/<ENDPOINT-NAME>*, so our service example above will be accessable via the URL *https://<YOUR-BUILDLY-URL>/api/amazing*.

To verify that this has worked, navigate to the Buildly docs at *https://<YOUR-BUILDLY-URL>/docs* and you should see the Swagger documentation for your service under Buildly's documentation.

2.1.5 (Optional) Implement Buildly permissions model

If you want to implement the Buildly *Permissions model* in your services, then you need to create **WorkflowLevels** for each of your data models and implement them in the models. We recommend creating WorkflowLevel1s for all top-level data models in your service and WorkflowLevel2s for defining any nested data relationships.

Use the following endpoints of your **app's API URL** to define WorkflowLevels:

- **POST /workflowlevel1: Create WorkflowLevel1**
  - Add the property *workflowleve1_uuid* to the data model and set the value to the UUID from the response.
- **POST /workflowlevel2: Create WorkflowLevel2**
  - Add the property *workflowleve2_uuid* to the data model and set the value to the UUID from the response.
  - If it's got a parent WorkflowLevel2, then add the property *parent_workflowlevel2* to the data model and set the value to the UUID of its parent WorkflowLevel2.

2.1.6 Appendix: Reserved endpoint names

The following endpoint names are reserved by Buildly and may not be implemented in your services' APIs:

- */admin*
- */oauth*
- */health_check*
- */docs*
- */complete*
- */disconnect*
- */static*
- */workflow*
- */core*
- */logicmodule*
- */milestone*
- */organization*

2.2 Deploy Buildly to Minikube

2.2.1 Overview

This tutorial explains how to deploy Buildly to an existing Minikube cluster.

Once you deploy Buildly to your cluster, it will be able to start receiving requests, and connecting with all of your other services.

2.2.2 Deploy Buildly with Helm

Requirements

The only requirements for this tutorial are:

1. A Minikube instance up and running in your local machine.
2. kubectl installed and configured to access your Minikube instance.
3. Helm also installed and configured.
4. Have a PostgreSQL database instance up and running.

Ensure minikube and kubectl are running by entering:

```
minikube status
kubectl cluster-info
kubectl get nodes
```

Ensure helm is initalized by running:

```
helm init
```

Deploy Buildly to Kubernetes clusters

The first thing you need to do is clone Buildly Helm Charts repository, where you can find the right charts to deploy Buildly to any Kubernetes cluster including local Minikube instances.

Run the following command to clone Buildly Helm Charts repository:

```
git clone https://github.com/buildlyio/helm-charts.git
```

Now, you need to create a namespace to deploy Buildly and you do it running:

```
kubectl create namespace buildly
```

The last but not least, you will execute the Helm charts but you need to pass the database connection information to Buildly when running the charts, otherwise, it won't work because it wasn't able to connect to the dabatase. You need to provide the database host, port(defaul=5432), username(base64 encrypted), and password(base64 encrypted). You run the following command replacing the fake data with your database connection info:

```
helm install . --name buildly-core --namespace buildly \
--set configmap.data.DATABASE_HOST="<db-host>" \
--set configmap.data.DATABASE_PORT="<db-port>" \
--set secret.data.DATABASE_USER="<db-user>" \
--set secret.data.DATABASE_PASSWORD="<db-pass>"
```

After that you should see a Buildly Core instance running in your Minikube dashboard. It has also created a ClusterIP and Ingress, so if you have a certificate manager and everything setup it should also be exposed externally. If you prefer to create a LoadBalancer instead, you can just delete the Ingress instance that was created.

This tutorial will walk you through connecting and configuring a web service to Buildly Core. Users will be able to see all endpoints in one OpenAPI specification, use only one URL to request data from different services, enable authentication and authorization.

It's assumed that you're already familiar with Docker, OpenAPI (old Swagger) and JWT.

While it's designed to give a good starting point, the tutorial doesn't cover all of Buildly Core's features.

CHAPTER 3

Models for Buildly

3.1 Permissions model

The Buildly permissions model follows the RBAC pattern.

Permissions are granted to CoreUsers by their **CoreGroups.** Each CoreGroup can be associated with one or more WorkflowLevels. "Permissions" are defined as the ability to execute CRUD operations on WorkflowLevels.

If a CoreGroup is given permissions to a WorkflowLevel, then those permissions will cascade down to all child WorkflowLevels.

By default, all CoreGroups can only have permissions defined to entities within their Organization. You can define a **global CoreGroup** that has permissions to all organizations by setting the *is_global* property to *true*.

3.1.1 Organization

The Organization is the top-level class of the Buildly permissions model. Everything contained within an organization—users, groups, and WorkflowLevels—can only be accessed by entities within the organization, with the exception of global CoreGroups (see below).

3.1.2 CoreUser

A CoreUser is a registered user of an application. Every CoreUser must belong to a **CoreGroup.**

3.1.3 CoreGroup

A CoreGroup defines a group of CoreUsers with specific permissions in the context of a given WorkflowLevel (1 or 2).

3.1.4 WorkflowLevels

Buildly allows you to define nested data hierarchies in your microservice architecture by using **WorkflowLevels.** There are two types of WorkflowLevels: **WorkflowLevel1** and **WorkflowLevel2.** WorkflowLevel2s can be nested within other WorkflowLevel2s as child objects, but they must always be associated with a parent WorkflowLevel1.

By creating WorkflowLevels for each model in your microservice architecture, you can enable them to share data by implementing their WorkflowLevel UUIDs as foreign keys.

You can define WorkflowLevels using the Buildly API once you have deployed your application.

3.1.5 Example implementation

Suppose you want to create a microservice application for managing factories that employ a variety of robots to manufacture a variety of products.

- The **Factory** is the top of the data hierarchy, so the Factory model would be considered the WorkflowLevel1.
- Each factory employs a set of **robots,** so the Robot model would be considered a WorkflowLevel2 associated with the "Factory" WorkflowLevel1.
- Each robot manufactures a set of **products,** so the Product model would be considered a WorkflowLevel2 with the Robot model as its parent.

Follow these steps to implement this data hierarchy in Buildly:

1. Create a **WorkflowLevel1** with the name "Factory" using your app's *POST /workflowlevel1* endpoint.
2. Add the *workflowlevel1_uuid* property to your Factory data model and set the value to the UUID of the Factory WorkflowLevel1 you just created.
3. Create a **WorkflowLevel2** with the name "Robot" using your app's *POST /workflowlevel2* endpoint. Set the value of *workflowlevel1* to the UUID of the Factory WorkflowLevel1.
4. Add the following properties to your Robot data model: - *workflowlevel1_uuid*: UUID of the Factory WorkflowLevel1. - *workflowlevel2_uuid*: UUID of the Robot WorkflowLevel2.
5. Create a **WorkflowLevel2** with the name "Product" using your app's *POST /workflowlevel2* endpoint. Set the value of *workflowlevel1* to the UUID of your Factory WorkflowLevel1, and set the value of *parent_workflowlevel2* to the UUID of your Robot WorkflowLevel2.
6. Add the following properties to your Product data model: - *workflowlevel1_uuid*: UUID of the Factory WorkflowLevel1. - *workflowlevel2_uuid*: UUID of the Robot WorkflowLevel2.

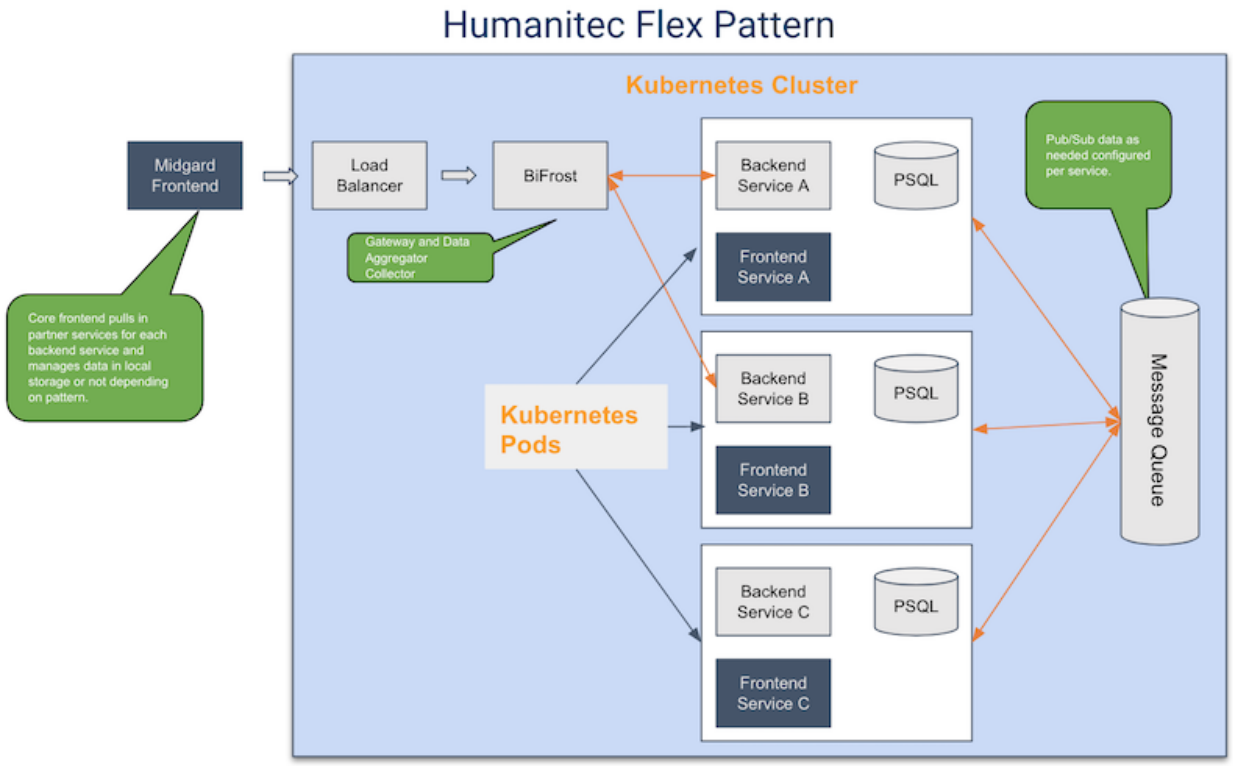
Associating your data models with WorkflowLevels enables them to be reused and dynamically swapped out. For example, if some of your factories employ humans to build the same products, you could create a **Human** WorkflowLevel2 and add the Human WorkflowLevel2 UUID as another parent to the Product data model.

In the attempt of removing the redundancy and repeatability when implementing a new application, Buildly Core has built-in Models. They are data structure and logic created for re-usability to facilitate developers life.

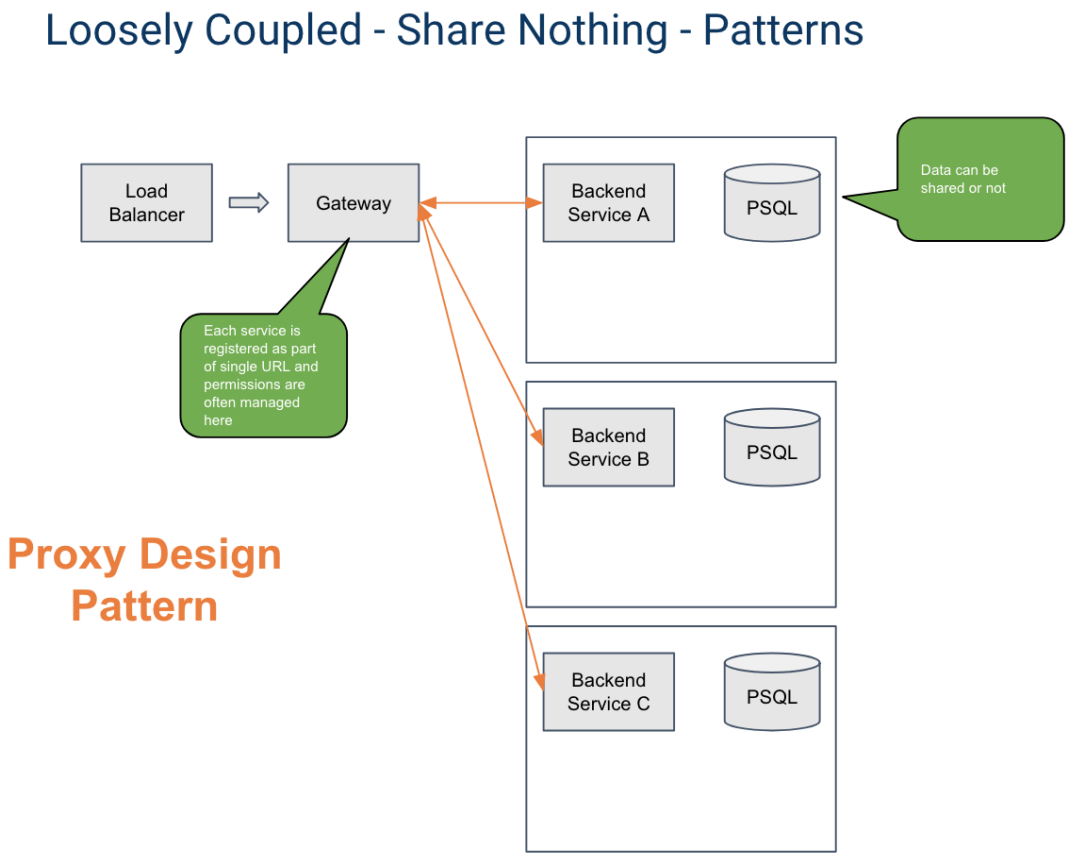
CHAPTER 4

Patterns with Buildly

Certain things are common enough that the chances are high you will find them in most web applications and Buildly Core makes it possible to implement a variety of common microservice architecture patterns.



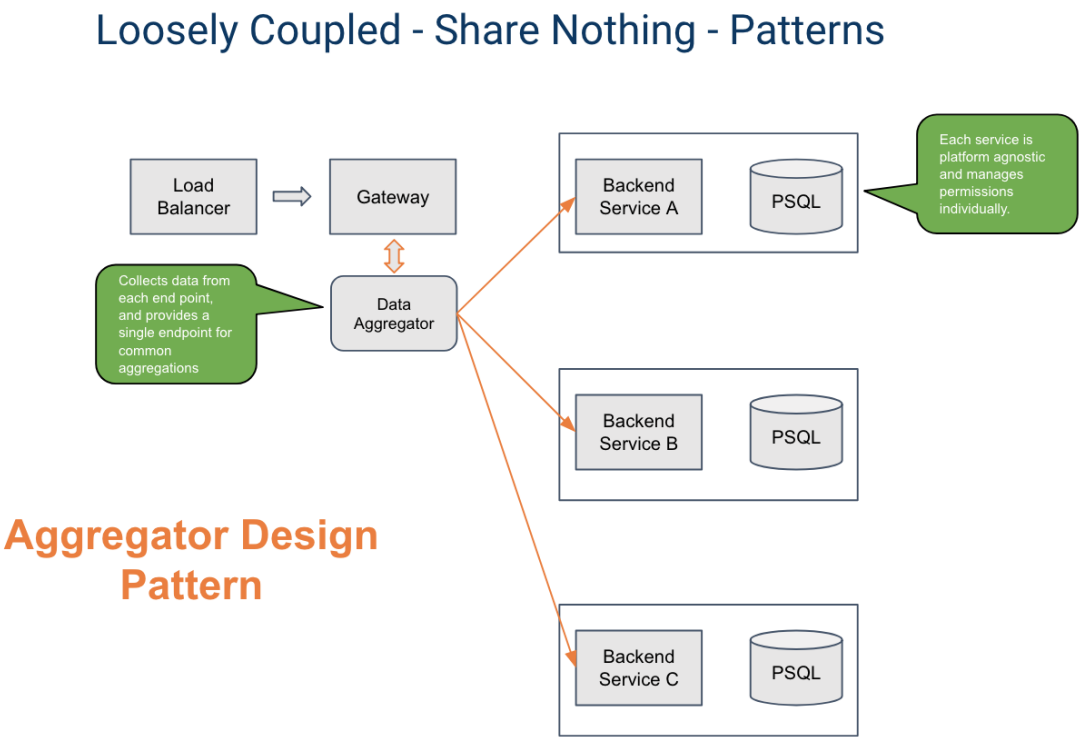
4.1 Proxy pattern



You can use Buildly's **API gateway** to implement a proxy pattern in your microservice architecture. Add all your services to Buildly via the API and it will run an auto-discovery process to identify all of the endpoints and combine them into a single API. All requests to and from the logic modules will be routed through Buildly. Authentication is handled with a JSON Web Token (JWT).

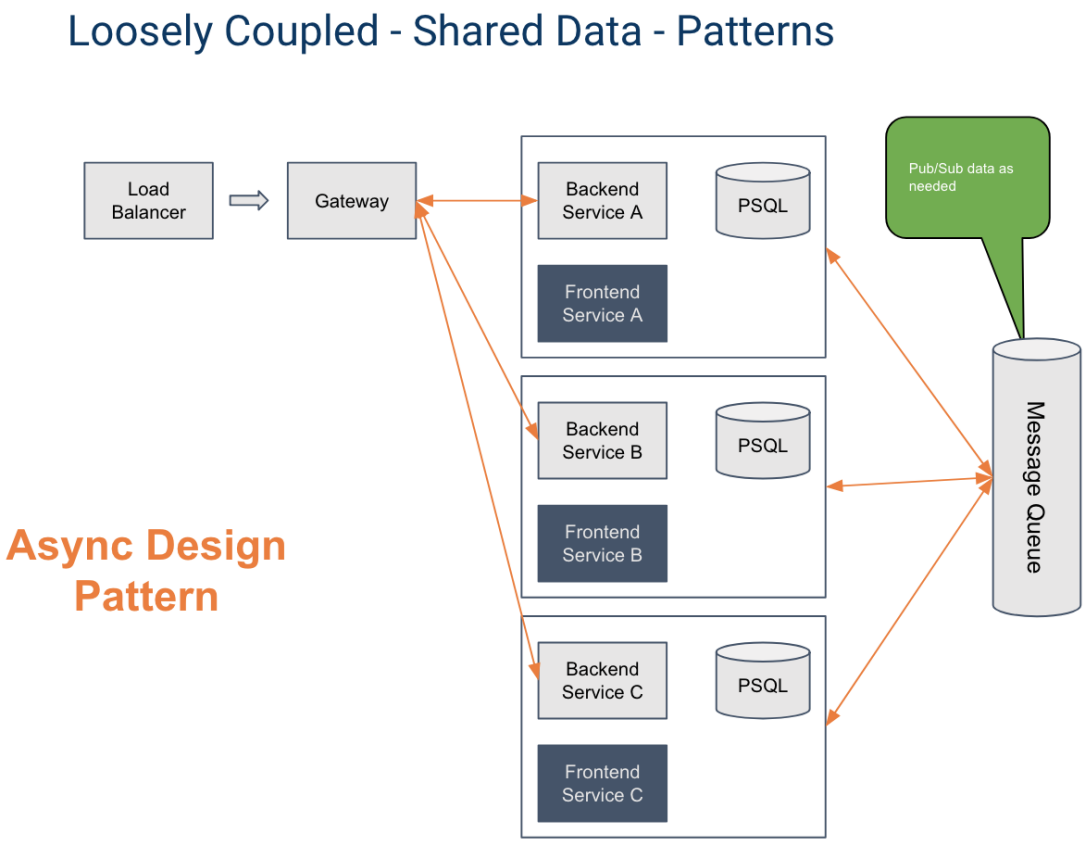
All you need to do to use the API gateway with your logic modules is to ensure that they expose a Swagger file (*swagger.json*) at the */docs* endpoint.

4.2 Aggregator pattern



Buildly includes a **data mesh** that can be used to implement an aggregator pattern. The data mesh is a service in Buildly running alongside the API gateway that contains a list of logic modules in the app and how they can be joined. It creates a lookup table of each of these connections. Then, the app frontend can query this table for each data type's unique ID, write the individual REST queries for each service, and then pull that service data back into one request object with a join of the data.

4.3 Async pattern



If you would prefer not to use the API gateway or data mesh, you can still use Buildly in an async pattern. Each service can publish data to a messaging queue like RabbitMQ and subscribe to the data it needs from other services. It requires a bit more upfront configuration for each service but provides the needed data and only the needed data to each endpoint.

CHAPTER 5

Additional Notes

Design notes, legal information and changelog are here.

5.1 Guidelines

The idea of this document is to make a guide for software source code quality. The guidelines appearing in this apply to any individual who create, alters, or reads the source code. This document is not a description of a complete software process and can be updated in the future by the maintainers and/or community.

5.1.1 Code Style

- Code is readable and maintainable. If you need to add too many comments to describe your code, then it is a sign of poor readability (this does not apply always). Code must be self-documenting.
- Should comply PEP8 with the following exceptions: - maximum line length is 120 characters (instead of 80 as in PEP 8 E501). - models.py, where it's allowed to break the 120 characters rule for the properties of a model.
- Variable naming is good and constant everywhere (coupon vs. voucher).
- Length and complexity of functions should be reduced as much as possible but don't create silly functions.
- Avoid magic constants or numbers. Incorrect: if len(password) > 7: error Correct: if len(password) > MAX_PASSWORD_SIZE: error
- Don't create a commit to undo anything from a previous commit, we should rebase the previous commit to undo the unneeded changes instead.

5.1.2 Design principles

- KISS - Simplicity should be a key goal in design, and unnecessary complexity should be avoided.
- Yagni - Don't add functionality until deemed necessary. Yagni only applies to capabilities built into the software to support a presumptive feature, it does not apply to efforts to make the software easier to modify.

- DRY - Remove duplication in logic via abstraction.

5.1.3 Pull Requests & Code Reviews

- Before merging, it's responsibility of the author to see if the branch is up-to-date (rebased on top of the latest commit) and the reviewer to confirm this . This way we avoid to merge two branches at the same time that can clash themselves and may leave potentially a red CI status.
- The title of the pull request should be descriptive but short. For example: Add validators to API endpoint /projects
- Use the pull request template to better explain how and which kind of changes you have done. It also should have the issue number inside for feature/change tracking.
- Commit message <= 70 characters, meaningful and written in imperative. Incorrect: "Fixing property". Correct: "Make User.email char type".
- Keep the scope of the ticket to solve and leave extra commits for a separate pull requests. It's fine to add small refactors but not the refactoring of an entire class, otherwise, you will be asked to create a task and another pull request for it.
- If the pull request is too big, consider to divide it in two or more parts. It's not recommended and comfortable to review very long pull requests. Divide and conquer!
- It fully complies the Acceptance Criteria of the ticket.
- How to QA should be in the body if it's not clear or detailed enough in the ticket.
- In case of DB and serializer changes, keep a special attention to migrations and backwards compatibility.
- Tested with unit and integration tests (test cover all possible cases).
- Check for efficiency, especially in database queries.

5.2 License

The license applies to all files in the Buildly Core repository and source distribution. This includes Buildly Core's source code, the examples, and tests, as well as the documentation.

To read the whole Buildly Core's license, click here!

5.3 Contribuiting

First off, thanks for taking the time to contribute!

To know more about how you can contribute to Buildly Core, click here!

5.4 Release Process

Buildly Core release numbering will work following the semantic versioning as defined in the following description that was extracted from the Semantic Versioning 2.0.0 web page. Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when we make incompatible API changes,

2. MINOR version when we add functionality in a backwards compatible manner, and
3. PATCH version when we make backwards compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

Extending the description, we're going to issue releases following this process:

- MAJOR.MINOR is the feature release version number.
- PATCH is the patch release version number, which is incremented for bugfix and security releases
- We make release candidate releases before a final feature release. These are of the form MAJOR.MINOR rc N, which means the Nth release candidate of version MAJOR.MINOR.

On GitHub, each Buildly feature release will have a branch called MAJOR.MINOR.x, so bugfix and security patches will be created from there. It will also have a signed tag indicating its version number.

5.4.1 Release timeline

Feature releases (MAJOR.MINOR) will happen following the roadmap, so it will depend on the prioritization and sprints. These releases will contain enhancements, new features, and so on.

Patch releases (MAJOR.MINOR.PATCH) will happen as needed, to security issues and/or fix bugs.

Some specific feature releases will be assigned as long-term support (LTS) releases, so they will still receive security and data loss fixes for a granted period of time, two years.

5.4.2 Deprecation policy

Certain features can be marked as deprecated by a feature release and if this happens in feature release MAJOR.X, the deprecated features will raise warnings but keep working in all MAJOR.X+n versions. Deprecated features will be removed in the MAJOR+1.0 release for features deprecated in the last MAJOR.X+n feature releases.

For example:

- MAJOR.0
- MAJOR.1
- MAJOR.2: Mark feature A, added in MAJOR.0, as deprecated
- MAJOR.3: Feature A still available and a warning is raised
- MAJOR+1.0: Remove feature A marked as deprecated in MAJOR.2