plone.server Documentation

Release 1.0

Ramon Navarro Bosch & Asko Soukka & Nathan Van Gheem

Apr 09, 2023

Contents

Contents:

CHAPTER 1

Quickstart

How to quickly get started using `plone.server`.

This tutorial will assume usage of virtualenv. You can use your own preferred tool for managing your python environment.

Setup the environment:

```
virtualenv .
```

Install `plone.server`:

```
./bin/pip install plone.server
```

Generate configuration file:

```
./bin/pcreate configuration
```

Finally, run the server:

```
./bin/pserver
```

The server should now be running on http://0.0.0.0:8080

Then, use Postman, curl or whatever tool you prefer to interact with the REST API.

Modify the configuration in config.json to customize server setttings.

1.1 Creating default content

Once started, you will require to add at least a Plone site to start fiddling around:

```
  curl -X POST -H "Accept: application/json" --user root:root -H "Content-Type:␣
↪application/json" -d '{
    "@type": "Site",
    "title": "Plone 1",
    "id": "plone",
    "description": "Description"
  }' "http://127.0.0.1:8080/zodb1/"
```

and give permissions to add content to it:

```
  curl -X POST -H "Accept: application/json" --user root:root -H "Content-Type:␣
↪application/json" -d '{
    "prinrole": {
        "Anonymous User": ["plone.Member", "plone.Reader"]
    }
  }' "http://127.0.0.1:8080/zodb1/plone/@sharing"
```

and create actual content:

```
  curl -X POST -H "Accept: application/json" --user root:root -H "Content-Type:␣
↪application/json" -d '{
    "@type": "Item",
    "title": "News",
    "id": "news"
  }' "http://127.0.0.1:8080/zodb1/plone/"
```

CHAPTER 2

Configuration

`plone.server` and it's addon define global configuration that is used throughout the `plone.server`. All of these settings are configurable by providing a JSON configuration file to the start script.

By default, the startup script looks for a `config.json` file. You can use a different file by using the `-c` option for the script script like this `./bin/pserver -c myconfig.json`.

2.1 Databases

To configure available databases, use the `databases` option. Configuration options map 1-to-1 to ZODB setup:

```
{
        "databases": [{
    "zodb1": {
                        "storage": "ZODB",
                        "path": "Data.fs"
                },
                "zodb2": {
                        "storage": "ZEO",
                        "address": "127.0.0.1",
      "port": 8090,
                        "configuration": {
                "pool_size": 100,
                "cache_size": 100
                   }
                }
        }]
}
```

2.2 Static files

```
{
  "static": [
    {"favicon.ico": "static/favicon.ico"}
  ]
}
```

2.3 Server port

```
{
        "address": 8080
}
```

2.4 Root user password

```
{
        "root_user": {
                "password": "root"
        }
}
```

2.5 CORS

```
{
        "cors": {
                "allow_origin": ["*"],
                "allow_methods": ["GET", "POST", "DELETE", "HEAD", "PATCH"],
                "allow_headers": ["*"],
                "expose_headers": ["*"],
                "allow_credentials": true,
                "max_age": 3660
        }
}
```

2.6 Async utilities

```
{
        "utilities": [{
    "provides": "plone.server.interfaces.ICatalogUtility",
    "factory": "pserver.elasticsearch.utility.ElasticSearchUtility",
    "settings": {}
  }]
}
```

CHAPTER 3

Production

3.1 Nginx front

Its so common to add the api with a nginx with a proxy_pass in front so there is the option define which is going to be the url for the generated urls inside the api:

Adding the header:

```
X-VirtualHost-Monster https://example.com/api/
```

Will do a rewrite of the urls.

Some configuration on nginx :

```
    location /api/ {
        proxy_set_header X-VirtualHost-Monster $scheme://$http_host/api/
        proxy_pass http://api.plone.server.svc.cluster.local:80/;
    }
```

CHAPTER 4

Security

`plone.server` implements robust ACL security.

An overview of our security features are:

- Users are given roles and groups
- Roles are granted permissions
- Groups are granted roles
- Roles can be granted to users on specific objects

4.1 Requests security

By default request has participation of anonymous user plus the ones added by auth pluggins

4.2 Databases, Application and static files objects

Databases and static files has an specific permission system. They don't have roles by default and the permissions are specified to root user

- plone.AddPortal
- plone.GetPortals
- plone.DeletePortals
- plone.AccessContent
- plone.GetDatabases

Anonymous user has on DB/StaticFiles/StaticDirectories/Application object :

- plone.AccessContent

4.3 Roles in plone.server Site objects

Defined at:

- plone/plone.server/src/plone.server/plone/server/permissions.zcml
- plone/plone.server/src/plone.server/plone/server/security.zcml

4.4 Content Related

4.4.1 plone.Anonymous

- plone.AccessPreflight

4.4.2 plone.Member

- plone.AccessContent

4.4.3 plone.Reader

- plone.AccessContent
- plone.ViewContent

4.4.4 plone.Editor

- plone.AccessContent
- plone.ViewContent
- plone.ModifyContent
- plone.ReindexContent

4.4.5 plone.Reviewer

4.4.6 plone.Owner

- plone.AccessContent
- plone.ViewContent
- plone.ModifyContent
- plone.DeleteContent
- plone.AddContent
- plone.ChangePermissions
- plone.SeePermissions
- plone.ReindexContent

4.5 Site/App Roles

4.5.1 plone.SiteAdmin

- plone.AccessContent
- plone.ManageAddons
- plone.RegisterConfigurations
- plone.WriteConfiguration
- plone.ReadConfiguration
- plone.ManageCatalog

4.5.2 plone.SiteDeleter

- plone.DeletePortal

4.6 Default roles on Plone Site

They are stored in anontations using IRolePermissionMap.

Created objects set the plone.Owner role to the user who created it.

4.7 Default groups on Plone Site

4.7.1 Managers

RootParticipation

There is a `root` user who has permissions to all site:

DB/APP permissions are defined on factory.py

Plone permissions because belongs to Managers group auth/participation.py

CHAPTER 5

Security

Security information for every operation is checked against three informations:

- Code definitions
- Local definitions
- Global definitions

Order or priority :

- Local
- Global
- Code

Locally can be defined :

- A user/group has a permission in this object and its not inherit
- A user/group has a permission in this object and its going to be inherit
- A user/group has a forbitten permission in this object and its inherit
- A user/group has a role in this object and its not inherit
- A user/group has a role in this object and its inherit
- A user/group has a forbitten role in this object and its inherit
- A role has a permission in this object and its not inherit
- A role has a permission in this object and its inherit
- A role has a forbitten permission in this object and its inherit

Globally :

- This user/group has this Role
- This user/group has this Permission

Code :

- This user/group has this Role
- This user/group has this permission
- This role has this permission

CHAPTER 6

Roles

There is two kind of roles : Global and Local. Ones that are defined to be local can't be used globally and viceversa. On indexing the global roles are the ones that are indexed for security plus the flat user/group information from each resource.

CHAPTER 7

Python helper functions

```
# Code to get the global roles that have access_content to an object
from plone.server.auth import get_roles_with_access_content
get_roles_with_access_content(obj)

# Code to get the user list that have access content to an object
from plone.server.auth import get_principals_with_access_content
get_principals_with_access_content(obj)


# Code to get all the security info
from plone.server.auth import settings_for_object
settings_for_object(obj)

# Code to get the Interaction object ( security object )
from zope.security.interfaces import IInteraction

interaction = IInteraction(request)

# Get the list of global roles for a user and some groups
interaction.global_principal_roles(principal, groups)

# Get if the authenticated user has permission on a object
interaction.check_permission(permission, obj)
```

CHAPTER 8

REST APIS

8.1 Get all the endpoints and its security

[GET] APPLICATION_URL/@apidefinition (you need plone.GetPortals permission)

8.2 Get the security info for a resource (with inherited info)

[GET] RESOURCE/@sharing (you need plone.SeePermissions permission)

8.3 Modify the local roles/permission for a resource

[POST] RESOURCE/@sharing (you need plone.ChangePermissions permission)

```
{
        "type": "Allow",
        "prinrole": {
                "principal_id": ["role1", "role2"]
        },
        "prinperm": {
                "principal_id": ["perm1", "perm2"]
        },
        "roleperm": {
                "role1": ["perm1", "perm2"]
        }
}
```

The different type of types are :

- Allow : you set it on the resource and the childs will inherit
- Deny : you set in on the resource and the childs will inherit

- AllowSingle : you set in on the resource and the childs will not inherit
- Unset : you remove the setting

CHAPTER 9

Applications

Applications are used to provide additional functionality to plone.server.

9.1 Community Addons

Some useful addons to use in your own development:

- pserver.elasticsearch: Index content in elastic search
- pserver.zodbusers: Store and authenticate users in the database
- pserver.mailer: async send mail

9.2 Creating

An application is a python package that implements an entry point to tell plone.server to load it.

If you're not familiar with how to build python applications, please read documentation on building packages before you continue on.

In your setup.py file, include an entry point like this for your application:

```
setup(
  entry_points={
    'plone.server': [
        'include = pserver.myaddon',
    ]
})
```

In this example, `pserver.myaddon` is your package module.

9.3 Initialization

Creating the `plone.server` entry point only tells `plone.server` that your application is available to be used. Your `config.json` file will also need to provide the application name in the `applications` array for it to be initialized.

```
{
  "applications": ["pserver.elasticsearch"]
}
```

9.4 Configuration

Once you create a `plone.server` application, there are three primary ways for it to hook into `plone.server`.

9.4.1 Call includeme function

Your application can provide an `includeme` function at the root of the module and `plone.server` will call it with the instance of the `root` object.

```
def includeme(root):
  # do initialization here...
  pass
```

9.4.2 Load app_settings

If an `app_settings` dict is provided at the module root, it will automatically merge the global `plone.server` app_settings with the module's. This allows you to provide custom configuration.

9.4.3 ZCML

If you're application is activated and has a `configure.zcml` file in it, it will automatically be loaded.

CHAPTER 10

Add-ons

Addons are integrations that can be installed or uninstalled against a Plone site. `plone.server` applications can provide potentially many addons. If you have not read the section on applications, please read that before you come here. The only way to provide addons is to first implement a `plone.server` application.

10.1 Creating an add-on

Create an addon installer class in an `install.py` file in your `plone.server` application:

```
from plone.server.addons import Addon
from plone.server import configure

@configure.addon(
    name="myaddon",
    title="My addon")
class MyAddon(Addon):

    @classmethod
    def install(self, request):
        # install code
        pass

    @classmethod
    def uninstall(self, request):
        # uninstall code
        pass
```

Scanning If your service modules are not imported at run-time, you may need to provide an additional scan call to get your services noticed by `plone.server`.

In your application `__init__.py` file, you can simply provide a `scan` call.

```
from plone.server import configure

def includeme(root):
    configure.scan('my.package.addon')
```

10.2 Layers

Your addon can also install layers for your application to lookup views and adapters from:

```
from plone.server.addons import Addon
from plone.server.registry import ILayers

LAYER = 'pserver.myaddon.interfaces.ILayer'

@configure.addon(
    name="myaddon",
    title="My addon")
class MyAddon(Addon):

    @classmethod
    def install(self, request):
      registry = request.site_settings
      registry.for_interface(ILayers).active_layers |= {
          LAYER
      }

    @classmethod
    def uninstall(self, request):
      registry = request.site_settings
      registry.for_interface(ILayers).active_layers -= {
        LAYER
      }
```

CHAPTER 11

Services

Services provide responses to api endpoint requests. A service is the same as a "view" that you might see in many web frameworks.

The reason we're using the convention "service" is because we're focusing on creating API endpoints.

11.1 Defining a service

A service can be as simple as a function in your application:

```
from plone.server import configure
from plone.server.interfaces import ISite

@configure.service(context=ISite, name='@myservice', method='GET',
                   permission='plone.AccessContent')
async def my_service(context, request):
    return {
        'foo': 'bar'
    }
```

The most simple way to define a service is to use the decorator method shown here.

As long as your application imports the module where your service is defined, your service will be loaded for you.

In this example, the service will apply to a GET request against a site, `/zodb/plone/@myservice`.

Scanning If your service modules are not imported at run-time, you may need to provide an additional scan call to get your services noticed by `plone.server`.

In your application `__init__.py` file, you can simply provide a `scan` call.

```
from plone.server import configure

def includeme(root):
    configure.scan('my.package.services')
```

11.2 class based services

For more complex services, you might want to use class based services.

The example above, with the class based approach will look like:

```
from plone.server import configure
from plone.server.interfaces import ISite
from plone.server.api.service import Service


@configure.service(context=ISite, name='@myservice', method='GET',
                   permission='plone.AccessContent')
class DefaultGET(Service):
    async def __call__(self):
      # self.context
      # self.request
      return {
          'foo': 'bar'
      }
```

11.3 special cases

11.3.1 I want that my service is accessible no matter the content

you can define in the Service class the **allow_access** = True

```
@service(
    context=IResource, name='@download',
    method='GET', permission='plone.Public')
class DefaultGET(DownloadService):

    __allow_access__ = True
```

CHAPTER 12

Content types

Content types allow you to provide custom schemas and content to your services.

OOTB, `plone.server` ships with simple `Folder` and `Item` content types. The `Folder` type allowing someone to add items inside of it. Both types only have simple dublin core fields.

12.1 Defining content types

A content type consists of a class and optionally, a schema to define the custom fields you want your class to use.

A simple type will look like this::

```
from plone.server import configure
from plone.server.content import Folder
from plone.server.interfaces import IItem
from zope import schema

class IMySchema(IItem):
    foo = schema.Text()

@configure.contenttype(
    portal_type="MyType",
    schema=IMySchema,
    behaviors=["plone.server.behaviors.dublincore.IDublinCore"])
class MyType(Folder):
    pass
```

This example creates a simple schema and assigns it to the `MyType` content type.

Scanning If your service modules are not imported at run-time, you may need to provide an additional scan call to get your services noticed by `plone.server`.

In your application `__init__.py` file, you can simply provide a `scan` call.

```
from plone.server import configure

def includeme(root):
    configure.scan('my.package.content')
```

CHAPTER 13

Behaviors

Besides having static content type definition with its schema there is the concept of behaviors that provide us cross-content type definitions with specific marker interface to create adapters and subscribers based on that behavior and not the content type.

13.1 Definition of a behavior

If you want to have a shared behavior based on some fields and operations that needs to be shared across different content you can define them on a zope.schema interface:

```
from plone.server.interfaces import IFormFieldProvider
from zope.interface import Interface
from zope.interface import provider
from zope.schema import Textline

@provider(IFormFieldProvider)
class IMyLovedBehavior(Interface):
    text = Textline(
        title=u'Text line field',
        required=False
    )

    text2 = Textline(
        title=u'Text line field',
        required=False
    )
```

Once you define the schema you can define a specific marker interface that will be applied to the objects that has this behavior:

```
class IMarkerBehavior(Interface):
    """Marker interface for content with attachment."""
```

Finally the instance class that implements the schema can be defined in case you want to enable specific operations or you can use plone.behavior.AnnotationStorage as the default annotation storage.

For example in case you want to have a class that stores the field on the content and not on annotations:

```
from plone.server.behaviors.properties import ContextProperty
from plone.server.behaviors.instance import AnnotationBehavior
from plone.server.interfaces import IResource
from plone.server import configure

@configure.behavior(
    title="Attachment",
    provides=IMyLovedBehavior,
    marker=IMarkerBehavior,
    for_=IResource)
class MyBehavior(AnnotationBehavior):
    """If attributes
    """
    text = ContextProperty(u'attribute', ())
```

On this example text will be stored on the context object and text2 as a annotation.

13.2 Static behaviors

With behaviors you can define them as static for specific content types:

```
from plone.server import configure
from plone.server.interfaces import IItem
from plone.server.content import Item

@configure.contenttype(
    portal_type="MyItem",
    schema=IItem,
    behaviors=["plone.server.behaviors.dublincore.IDublinCore"])
class MyItem(Item):
    pass
```

Scanning If your service modules are not imported at run-time, you may need to provide an additional scan call to get your services noticed by `plone.server`.

In your application `__init__.py` file, you can simply provide a `scan` call.

```
from plone.server import configure

def includeme(root):
    configure.scan('my.package.services')
```

13.2.1 Create and modify content with behaviors

On the deserialization of the content you will need to pass on the POST/PATCH operation the behavior as a object on the JSON.

CREATE an ITEM with the expires : POST on parent:

```
{
    "@type": "Item",
    "plone.server.behaviors.dublincore.IDublinCore": {
        "expires": "1/10/2017"
    }
}
```

MODIFY an ITEM with the expires : PATCH on the object:

```
{
    "plone.server.behaviors.dublincore.IDublinCore": {
        "expires": "1/10/2017"
    }
}
```

13.2.2 Get content with behaviors

On the serialization of the content you will get the behaviors as objects on the content.

GET an ITEM : GET on the object:

```
{
    "@id": "http://localhost:8080/zodb/plone/item1",
    "plone.server.behaviors.dublincore.IDublinCore": {
        "expires": "2017-10-01T00:00:00.000000+00:00",
        "modified": "2016-12-02T14:14:49.859953+00:00",
    }
}
```

13.3 Dynamic Behaviors

plone.server offers the option to have content that has dynamic behaviors applied to them.

13.3.1 Which behaviors are available on a context

We can know which behaviors can be applied to a specific content.

GET CONTENT_URI/@behaviors:

```
{
    "available": ["plone.server.behaviors.attachment.IAttachment"],
    "static": ["plone.server.behaviors.dublincore.IDublinCore"],
    "dynamic": [],
    "plone.server.behaviors.attachment.IAttachment": { },
    "plone.server.behaviors.dublincore.IDublinCore": { }
}
```

This list of behaviors is based on the for statement on the zcml definition of the behavior. The list on static are the ones defined on the content type definition on the zcml. The list on dynamic are the ones that have been assigned.

13.3.2 Add a new behavior to a content

We can add a new dynamic behavior to a content by a PATCH operation on the object with the @behavior attribute or in a small PATCH operation to @behavior entry point with the value to add.

MODIFY an ITEM with the expires : PATCH on the object:

```
{
    "plone.server.behaviors.dublincore.IDublinCore": {
        "expires": "1/10/2017"
    }
}
```

MODIFY behaviors : PATCH on the object/@behaviors:

```
{
    "behavior": "plone.server.behaviors.dublincore.IDublinCore"
}
```

13.3.3 Delete a behavior to a content

We can add a new dynamic behavior to a content by a DELETE operation to @behavior entry point with the value to remove.

DELETE behaviors : DELETE on the object/@behaviors:

```
{
    "behavior": "plone.server.behaviors.dublincore.IDublinCore"
}
```

CHAPTER 14

Interfaces

`plone.server` uses interfaces to abstract and define various things including content. Interfaces are useful when defining api contracts, using inheritance, defining schema/behaviors and being able to define which content your services are used for.

In the services example, you'll notice the use of `context=ISite` for the service decorator configuration. In that case, it's to tell `plone.server` that the service is only defined for a site object.

14.1 Common interfaces

Interfaces you will be interested in defining services for are:

- `plone.server.interface.IDatabase`: A database contains the site objects
- `plone.server.interface.ISite`: Site content object
- `plone.server.interface.IResource`: Base interface for all content
- `plone.server.interface.IContainer`: Base interface for content that can contain other content
- `plone.server.interface.IRegistry`: Registry object interface
- `plone.server.interface.IDefaultLayer`: Layers are an interface applied to the request object. IDefaultLayer is the base default layer applied to the request object.

CHAPTER 15

Migrations

plone.server provides an interface to run migrations for itself and the applications it currently has activated.

Migrations are run against applications. If applications define addons for sites, those application migration steps need to check for the installation of their addon in the migration step.

15.1 Running migrations

plone.server provides a command line utility to manage and run migrations against your entire install or against a particular site.

Here is a minimalistic example using the command:

```
./bin/pmigrate
```

By default, the `pmigrate` command will migrate all sites on all available databases.

15.2 pmigrate command options

- dry-run: do test running the migration but not commit to the database
- site: path to site to run the command against
- report: report current versions site(s) are migrated to and see available migrations
- app: run command for a particular application
- to-version: run migrations to a provided version

Advanced command usages example:

```
./bin/pmigrate --site=/zodb/plone --app=pserver.elasticsearch --to-version=1.0.1 --
↪dry-run
```

15.3 Defining migrations

To define migrations in your own applications, `plone.server` provides a simple decorator::

```
from plone.server.migrations import migration
@migration('my.app', to_version='1.0.1')
def migrate_stub(site):
    # my migration code...
    pass
```

CHAPTER 16

Commands

You can provide your own CLI commands for plone.server through a simple interface.

16.1 Available commands

- pserver: run the http rest api server
- pmigrate: run available migration steps
- pcli: command line utility to run manually RUN API requests with
- pshell: drop into a shell with root object to manually work with
- pcreate: use cookiecutter to generate plone.server applications

16.2 Creating commands

plone.server provides a simple API to write your own CLI commands.

Here is a minimalistic example:

```
from plone.server.commands import Command
class MyCommand(Command):

    def get_parser(self):
        parser = super(MyCommand, self).get_parser()
        # add command arguments here...
        return parser

    def run(self, arguments, settings, app):
        pass
```

Then, in your setup.py file, include an entry point like this for your command:

```
setup(
  entry_points={
    'console_scripts': [
          'mycommand = my.package.commands:MyCommand'
    ]
})
```

CHAPTER 17

Application Configuration

`plone.server` handles configuration application customizations and extension mostly with decorators in code.

This page is meant to be a reference to the available decorators and options to those decorators.

17.1 service

`@configure.service`

- *context*: Content type interface this service is registered against. Example: ISite: *required*
- *method*: HTTP method this service works against. Default is `GET`
- *permission*: Permission this service requires. Default is configure default_permission setting
- *layer*: Layer this service is registered for. Default is `IDefaultLayer`
- *name*: This is used as part of the uri. Example `@foobar` -> `/mycontent/@foobar`. Leave empty to be used for base uri of content -> `/mycontent`.

17.2 content type

`@configure.contenttype`

- *portal_type*: Name of the content type: *required*
- *schema*: Interface schema to use for type: *required*
- *add_permission*: Permission required to add content. Defaults to `plone.AddContent`
- *allowed_types*: List of types allowed to be added inside this content assuming it is a Folder type. Defaults to allowing all types.

17.3 behavior

`@configure.behavior`

- *title*: Name of behavior
- *provides*: Interface this behavior provides
- *marker*: Marker interface to apply to utilized instance's behavior
- *for_*: Content type this behavior is available for

17.4 addon

`@configure.addon`

- *name*: *required*
- *title*: *required*

17.5 adapter

`@configure.adapter`

- *for_*: Type or list of types this adapter adapts: *required*
- *provides*: Interface this adapter provides: required
- *name*: Your adapter can be named to be looked up by name
- *factory*: To use without decorator syntax, this allows you to register adapter of class defined elsewhere

17.6 subscriber

`@configure.subscriber`

- *for_*: Type or list of types this subscriber is for: *required*
- *handler*: A callable object that handles event, this allows you to register subscriber handler defined elsewhere
- *factory*: A factory used to create the subscriber instance
- *provides*: Interface this adapter provides–must be used along with factory

17.7 utility

`@configure.utility`

- *provides*: Interface this utility provides
- *name*: Name of utility
- *factory*: A factory used to create the subscriber instance

17.8 permission

configure.permission

- *id*
- *title*
- *description*

17.9 role

configure.role

- *id*
- *title*
- *description*

17.10 grant

configure.grant

- *role*: ID of role
- *principal*: ID of principal to grant to
- *permission*: ID of permission to grant
- *permissions*: List of permission IDs to grant to

17.11 grant_all

configure.grant_all

- *principal*: ID of principal
- *role*: ID of role

CHAPTER 18

Design

This section is meant to explain and defend the design of `plone.server`.

18.1 JavaScript application development focus

One of the main driving factors behind the development of `plone.server` is to streamline the development of custom CMS-like web applications.

Some of the technologies we support in order to be a great web application development platform are:

- Everything is an API endpoint
- JWT
- Web sockets
- Configuration is done with JSON
- URL to object-tree data model

18.2 CMS

`plone.server` is a API-only, CMS framework. It is for building CMS-like applications.

It uses the ZODB, a database well suited for CMS.

18.3 Speed

A primary focus of `plone.server` is speed. We take shortcuts and may use some ugly or less-well conceptually architected solutions in some areas in order to gain speed improvements.

Some of the decisions we made affect how applications and addons are designed. Mainly, we try to stay light on the amount of data we're loading from the database where possible and we try to lower the number of lookups we do in certain scenarios.

That being said, `plone.server` is not a barebones framework. It provides a lot of functionality so it will never be as fast as say Pyramid.

"There are no solutions. There are only trade-offs." - Thomas Sowell

18.4 Asynchronous

`plone.server` is asynchronous from the group up, built on top of `aiohttp` using Python 3.5's asyncio features.

Practically speaking, being built completely on asyncio compatible technologies, `plone.server` does not block for network IO to the database, index catalog, redis, etc or whenever you'd integrated.

Additionally, we have support for async utilities that run in the same async loop and async content events.

Finally, the web server can also support web sockets OOTB.

18.5 Tooling

I've talked some about it but these are the basic technologies `plone.server` is built with:

- aiohttp
- ZODB
- ZCA

18.6 Security

`plone.server` uses the same great security infrastructure that has made Plone such a great product for the past 15 years.

18.7 Style

Stylistically, currently the project isn't extremely coherent right now so I'll speak to what we'd like to work toward stylistically long term::

- JSON configuration
- No ZCML
- Pyramid-like idioms and syntax where it makes sense
- Functions + decorators over classes

18.8 ZODB

`plone.server` uses the ZODB as its database engine.

The ZODB database server maps very nicely onto a CMS application where content is stored in a tree data structure.

CHAPTER 19

Indices and tables

- genindex
- modindex
- search