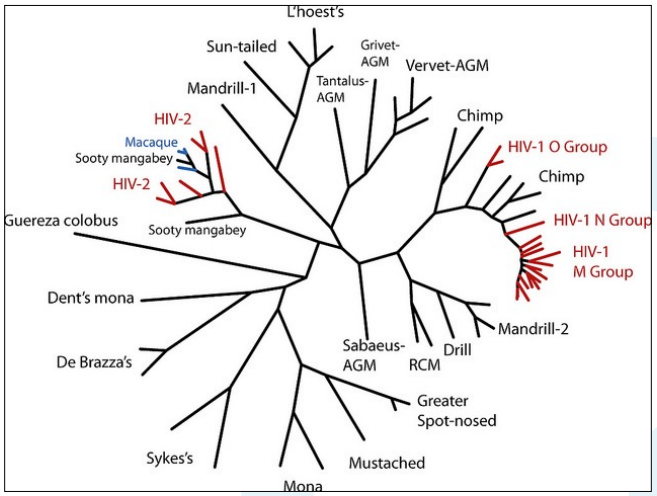
Good morning. I'm going to tell you a story about adaptation, both in a metaphorical and in a very literal sense. Our story is about Plone, an open source content management system and a community I've been involved with for the last ten years.

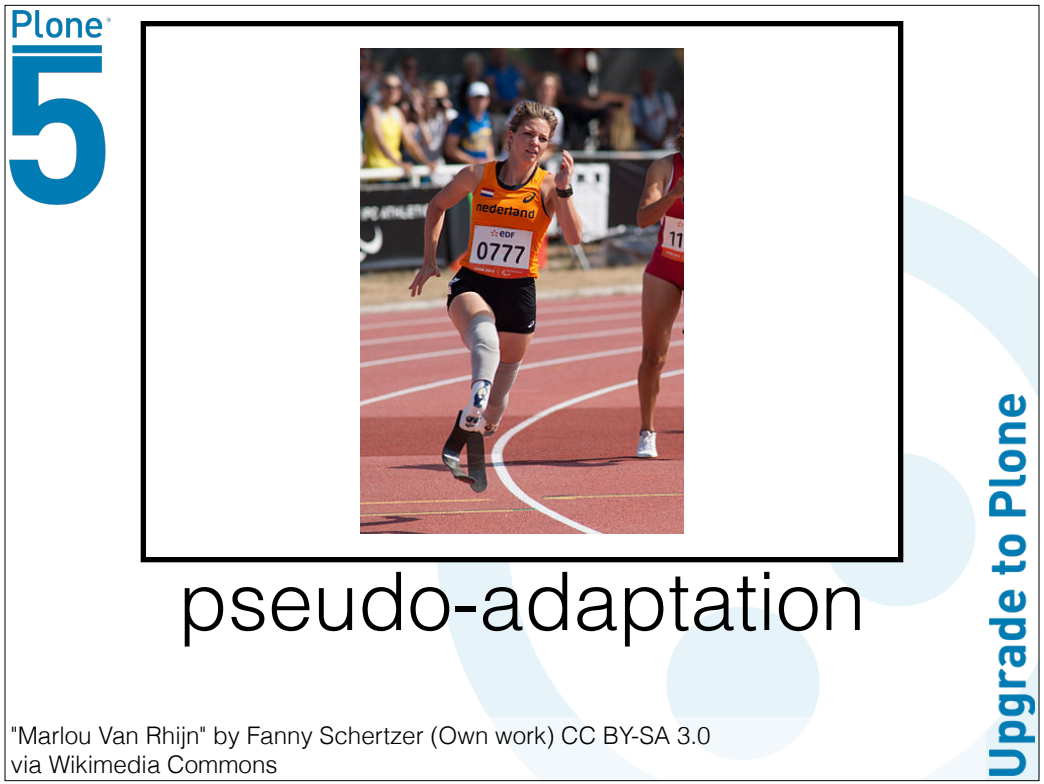


Adaptation

"Phylogeny Figures" by pat.0020015.g001, CC-BY
https://www.flickr.com/photos/123636286@N02/14138677365

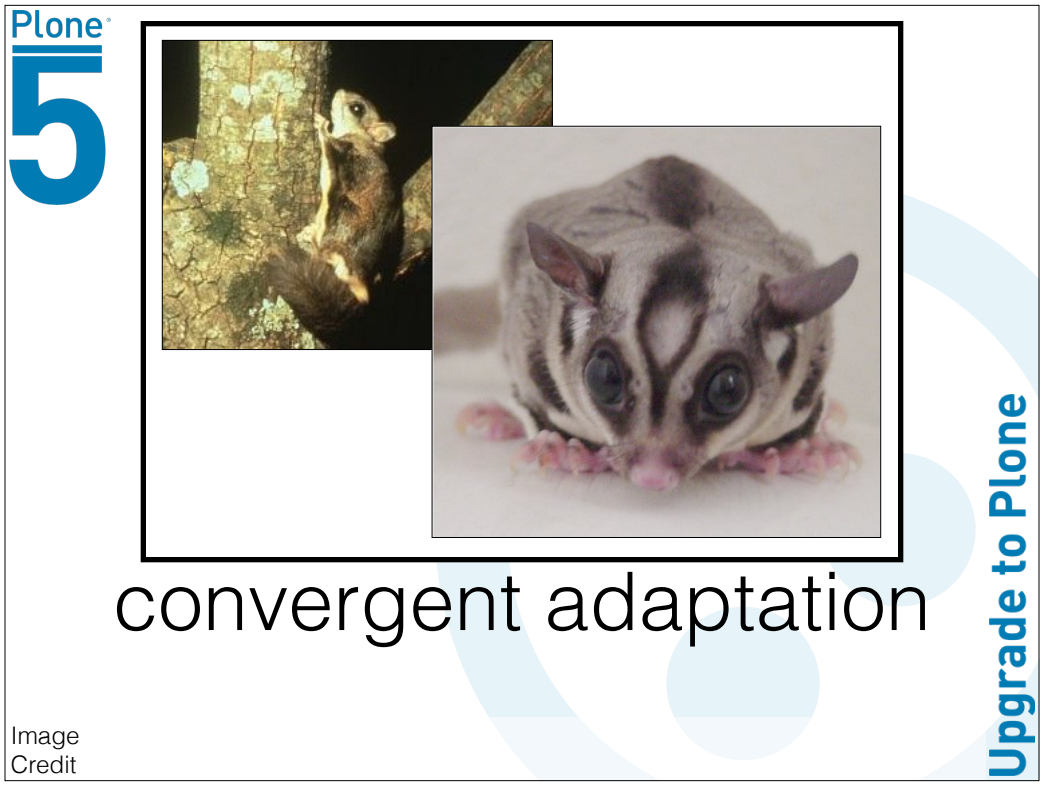
Upgrade to Plone

There are a few possible meanings of adaptation, but I'm thinking of it here in the biological/evolutionary sense.

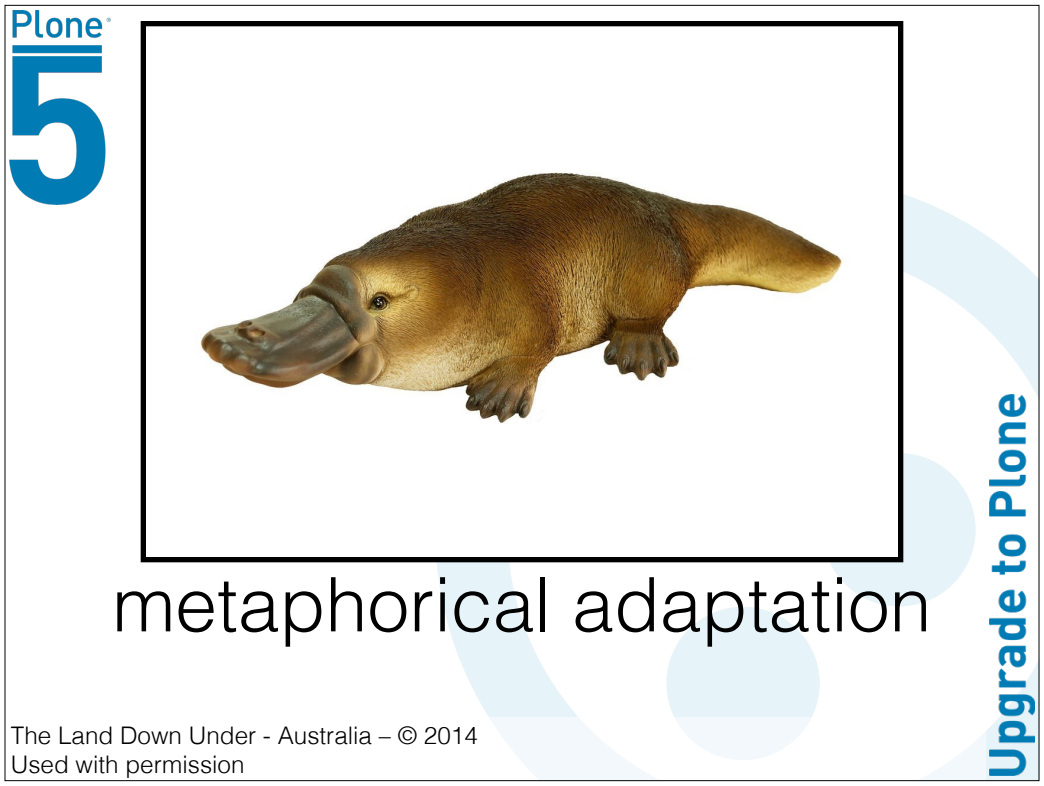


I'll be taking a look at the environment challenges that Plone has faced, and then at a number of different ways in which the idea of adaptation has served Plone well over time.

We'll start with something I'm calling "pseudo-adaptation". The idea is that a behavioral change can mimic an evolutionary adaptation. It's an adaptation, but not a "true" one, in the evolutionary sense. For example, consider this athlete. Her prostheses enable her "adaptation" to an environment that requires running. But the change is not biological or evolutionary.



We'll also look at an example that is similar to convergent adaptation, where different species undergo adaptations that make them similar. A good example of this can be found in the flying squirrel, a placental mammal found here in North America and the sugar glider, a marsupial found in Australia. Both have developed the ability to glide despite starting from widely divergent branches of the tree of life.



Finally, we'll see some "metaphorical" adaptations. In these cases, there's no real adaptation happening, but the idea of it is useful in thinking about how a system works. Honestly, I've got no reason to use a Platypus here, except that it's a bit of an evolutionary mystery (and it seems you can't talk about programming approaches in an Object Oriented system without it coming up).

“A Famous Historian” from Monty Python and the Holy Grail

We’ll get to the technical stuff soon enough, but let’s begin with some history. It will help you to understand where Plone comes from and about the forces that have driven change in the system and the community.

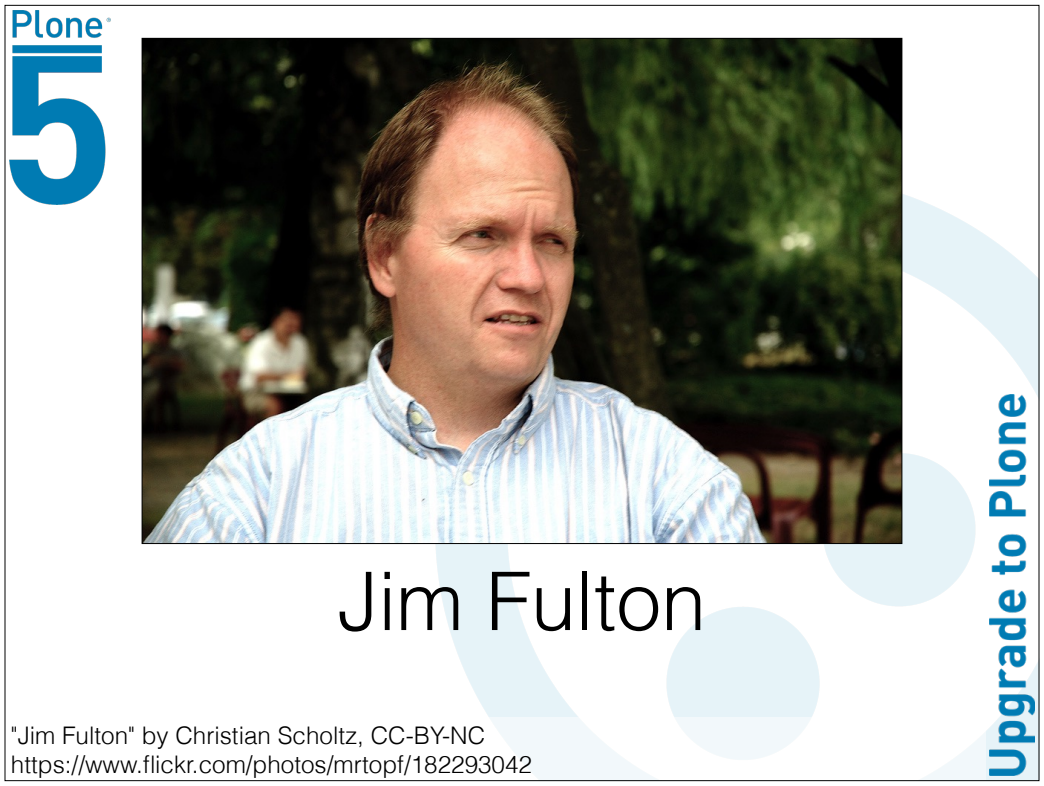
Plone 5

Let's Go Way Back

"Peabody's Wayback Machine" by Twechie CC-BY-SA-2.0
https://www.flickr.com/photos/twechie/5410720755

Upgrade to Plone

We'll begin our story today by taking a trip way back in time.



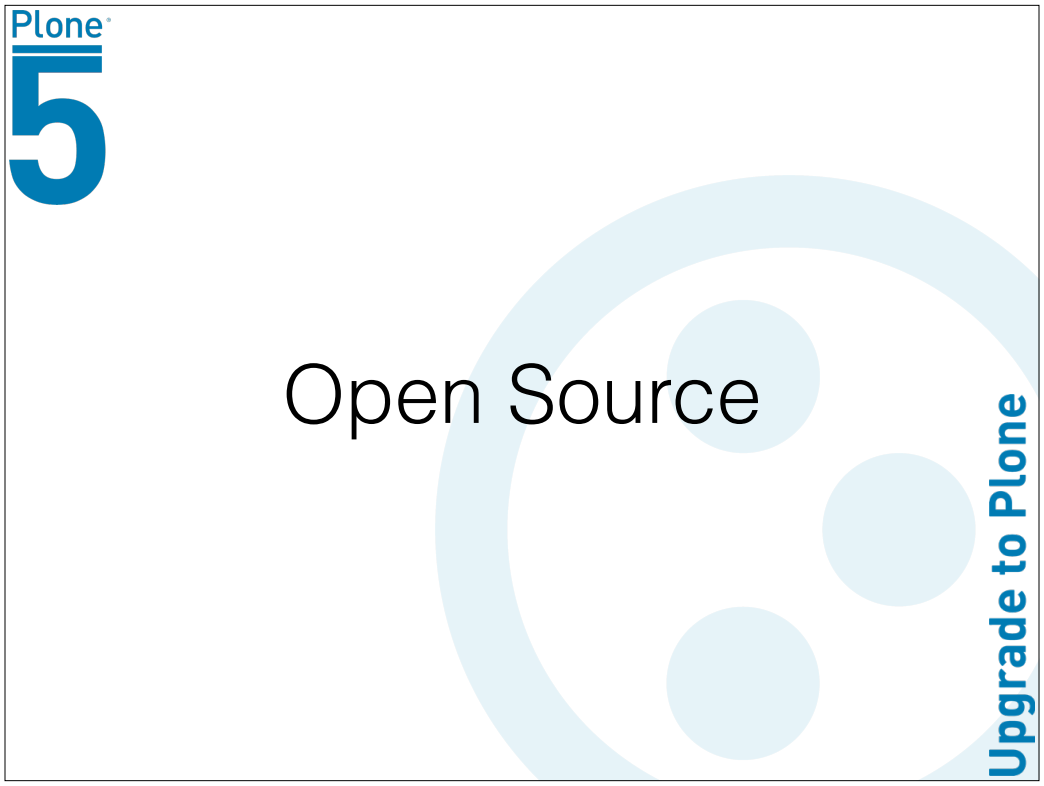
The year is 1996. This man, the CTO of Digital Creations in Fredericksburg, VA is on a plane to the International Python Conference in California. He's scheduled to give a tutorial about CGI programming and so he spends the flight learning the specification. He's got some ideas about how to improve on it, and on the flight home he designs what would become Bobo, the first Web Object Publishing System.

Principia

Image by Paul Everitt, used with permission

Upgrade to Plone

Bobo, along with Document Template (a package supporting dynamic tempting) and Bobopos (an Object Database that would become the ZODB) form the core of what Digital Creations called “Principia” a commercial Python Application Server.



In 1998, the largest investor in Digital Creations convinced the CEO Paul Everitt to release the Principia software as Open Source Software.

Principia became the Z Object Publishing Environment, Digital Creations became the Zope Corporation, and Zope was born.

In 1998, the largest investor in Digital Creations convinced the CEO Paul Everitt to release the Principia software as Open Source Software.

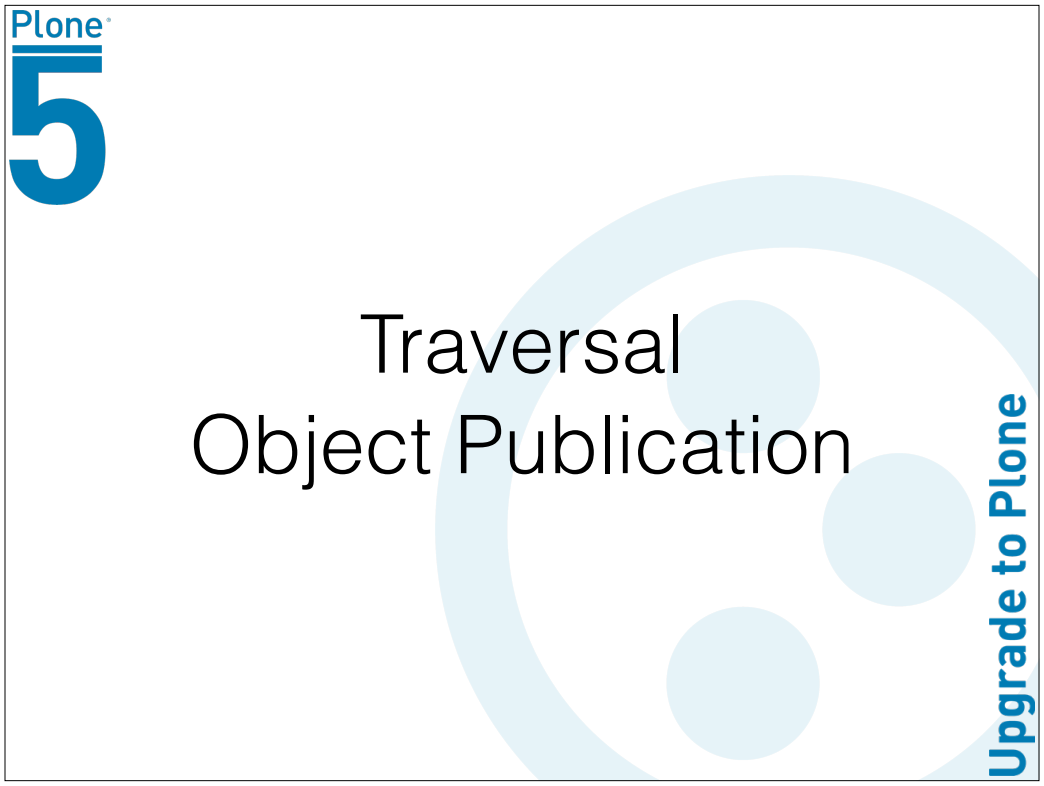
Principia became the Z Object Publishing Environment, Digital Creations became the Zope Corporation, and Zope was born.

Zope brought a number of technological innovations to the table, but the key ideas were

* traversal
* object publication

Zope brought a number of technological innovations to the table, but the key ideas were

* traversal
* object publication



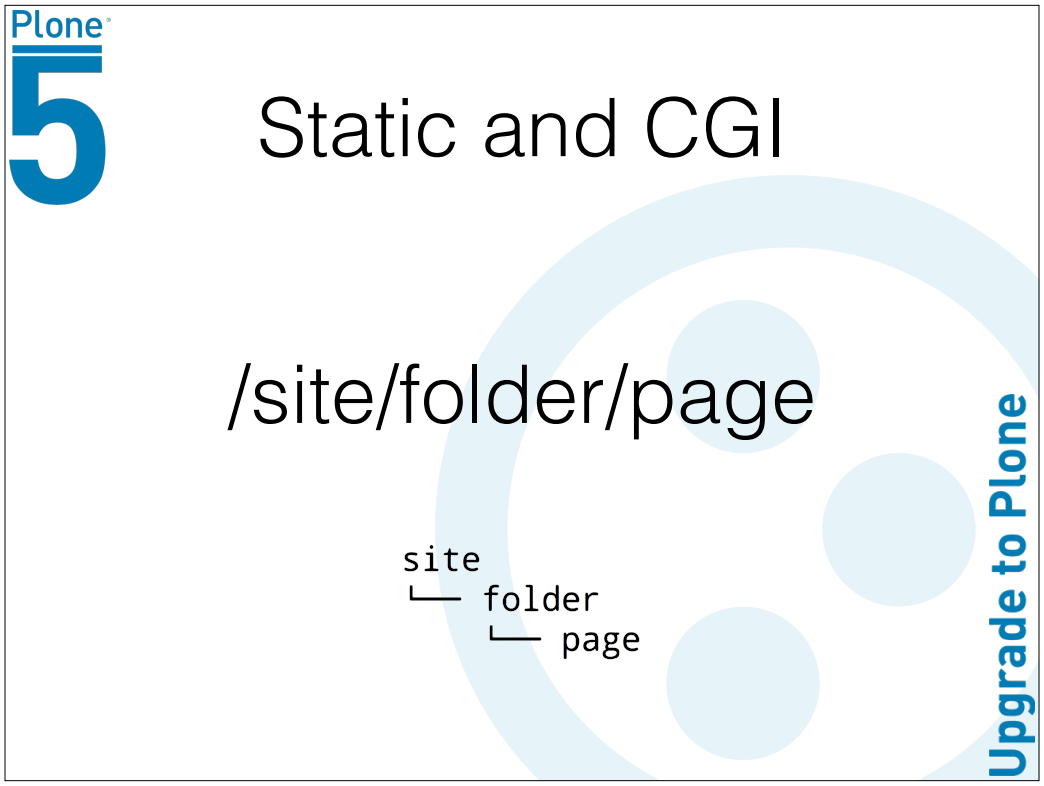
Zope brought a number of technological innovations to the table, but the key ideas were

* traversal
* object publication

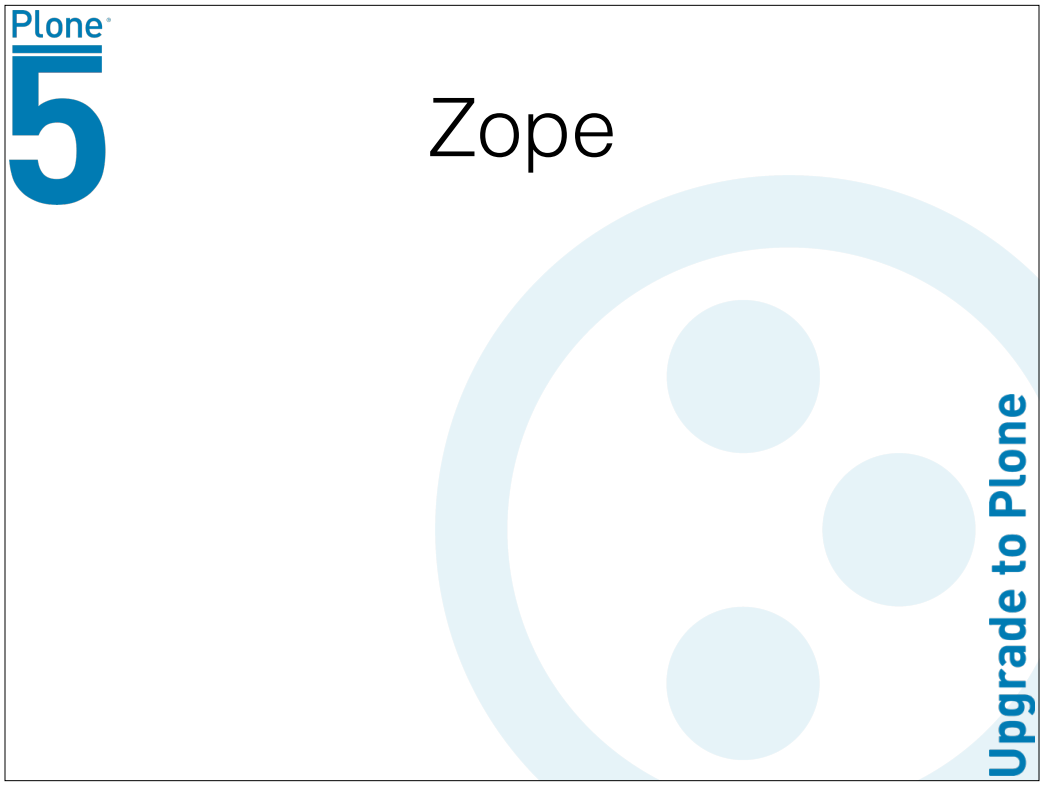
Traversal is a powerful idea with a simple origin.

It starts with a URL

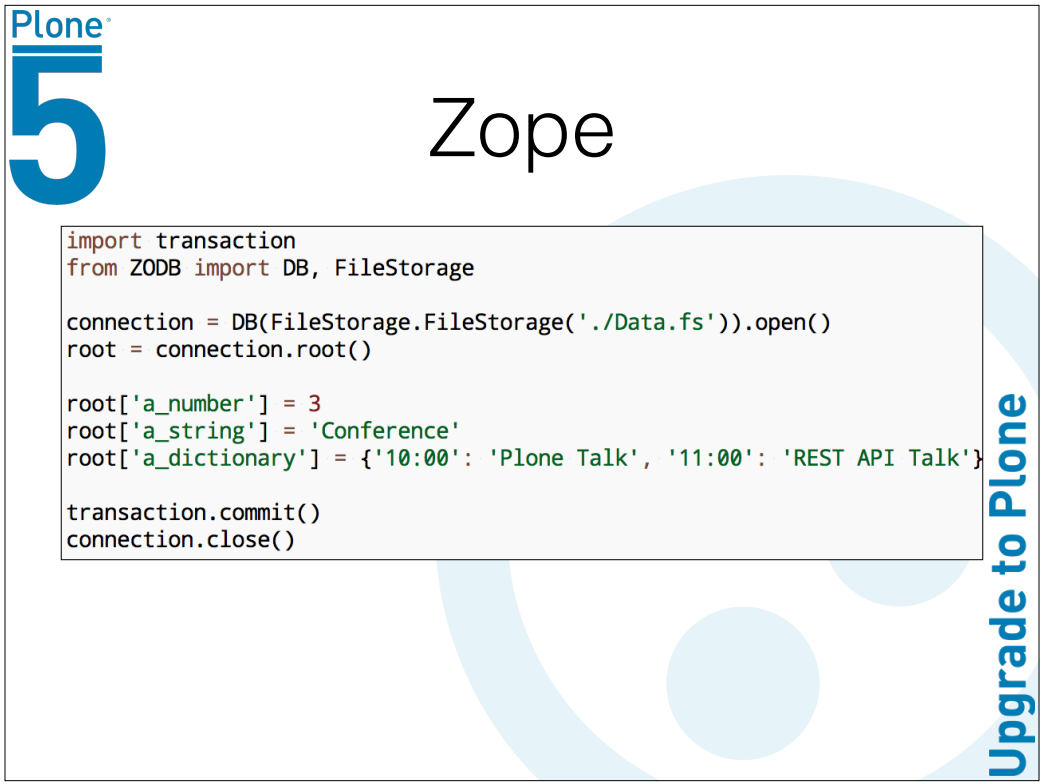
* This part of the URL is called the “path”. Notice how it looks a lot like a filesystem path.
* Static Web servers like Apache or Nginx serve static content by walking the filesystem, following these paths and returning the item at the end of the path as an HTTP response.
* CGI, the dominant dynamic web technology of the day works the same way, except that the path ends in an executable script that generates HTTP headers and a response body.



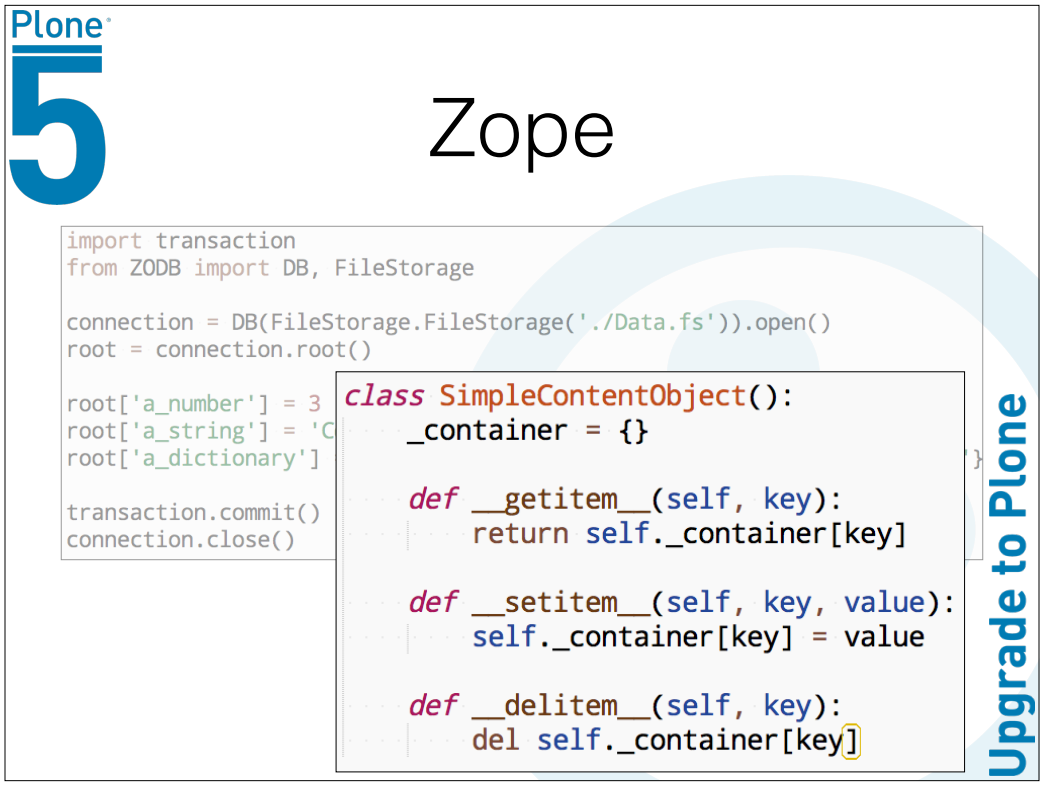
* This part of the URL is called the “path”. Notice how it looks a lot like a filesystem path.
* Static Web servers like Apache or Nginx serve static content by walking the filesystem, following these paths and returning the item at the end of the path as an HTTP response.
* CGI, the dominant dynamic web technology of the day works the same way, except that the path ends in an executable script that generates HTTP headers and a response body.



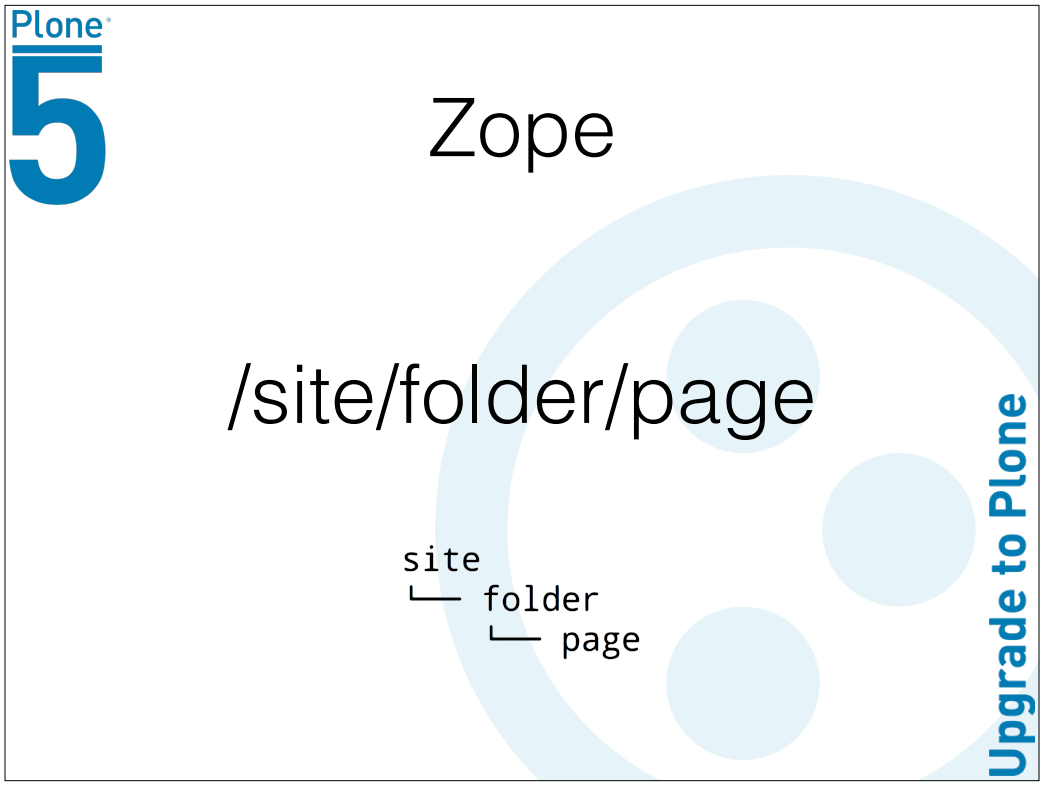
Jim Fulton, on that airplane ride back in ’96, asked himself a question: “Could we treat Python objects the same way?
If we have a database that allows us to store Python objects,
And we combine that with objects that can behave like Python dicts,



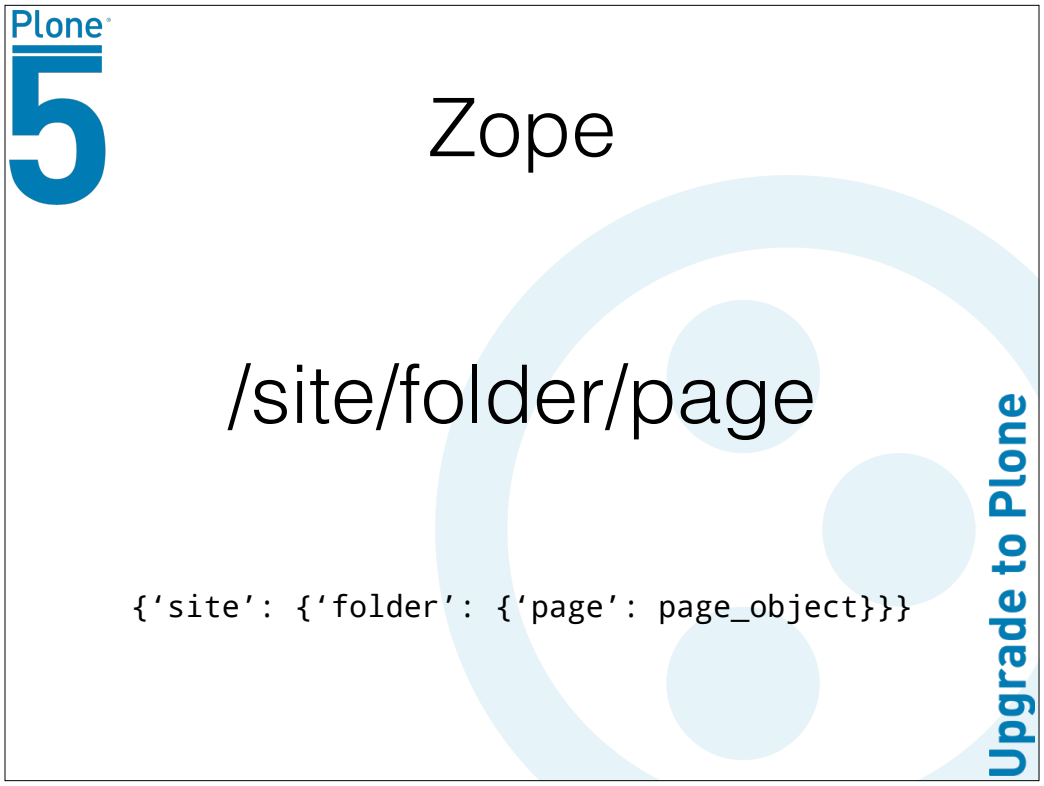
Jim Fulton, on that airplane ride back in ’96, asked himself a question: “Could we treat Python objects the same way?
If we have a database that allows us to store Python objects,
And we combine that with objects that can behave like Python dicts,



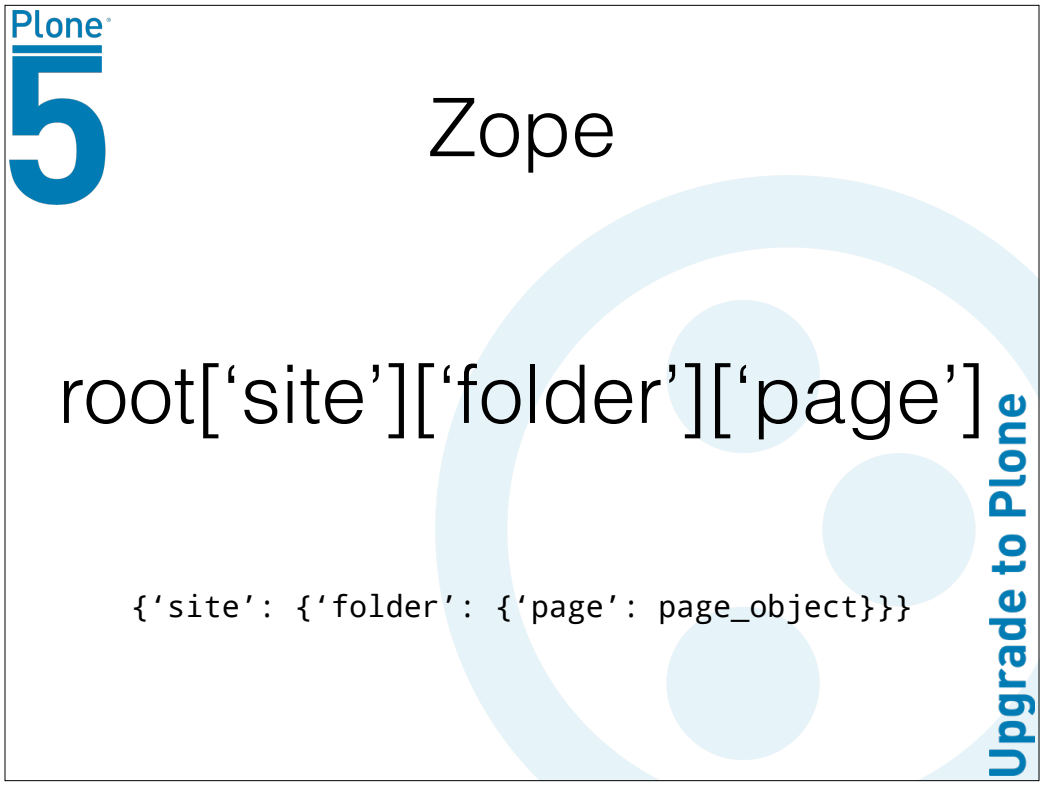
Jim Fulton, on that airplane ride back in ’96, asked himself a question: “Could we treat Python objects the same way?
If we have a database that allows us to store Python objects,
And we combine that with objects that can behave like Python dicts,



Could we not, then, transform this filesystem hierarchy into a series of nested objects?



Could we not, then, transform this filesystem hierarchy into a series of nested objects?



Treating path segments like keys, would allow us to walk the chain of contained objects just like walking a filesystem.

Since Zope is an Object Publishing system, then the part that remains is to let objects publish themselves.

* We find the objects using traversal.
* Then we *call* the object, passing in the request (which contains environmental information) to generate a publishable representation of itself.
* Finally, we use that representation as the response we send back to the client.

Since Zope is an Object Publishing system, then the part that remains is to let objects publish themselves.

* We find the objects using traversal.
* Then we *call* the object, passing in the request (which contains environmental information) to generate a publishable representation of itself.
* Finally, we use that representation as the response we send back to the client.

Since Zope is an Object Publishing system, then the part that remains is to let objects publish themselves.

* We find the objects using traversal.
* Then we *call* the object, passing in the request (which contains environmental information) to generate a publishable representation of itself.
* Finally, we use that representation as the response we send back to the client.

Since Zope is an Object Publishing system, then the part that remains is to let objects publish themselves.

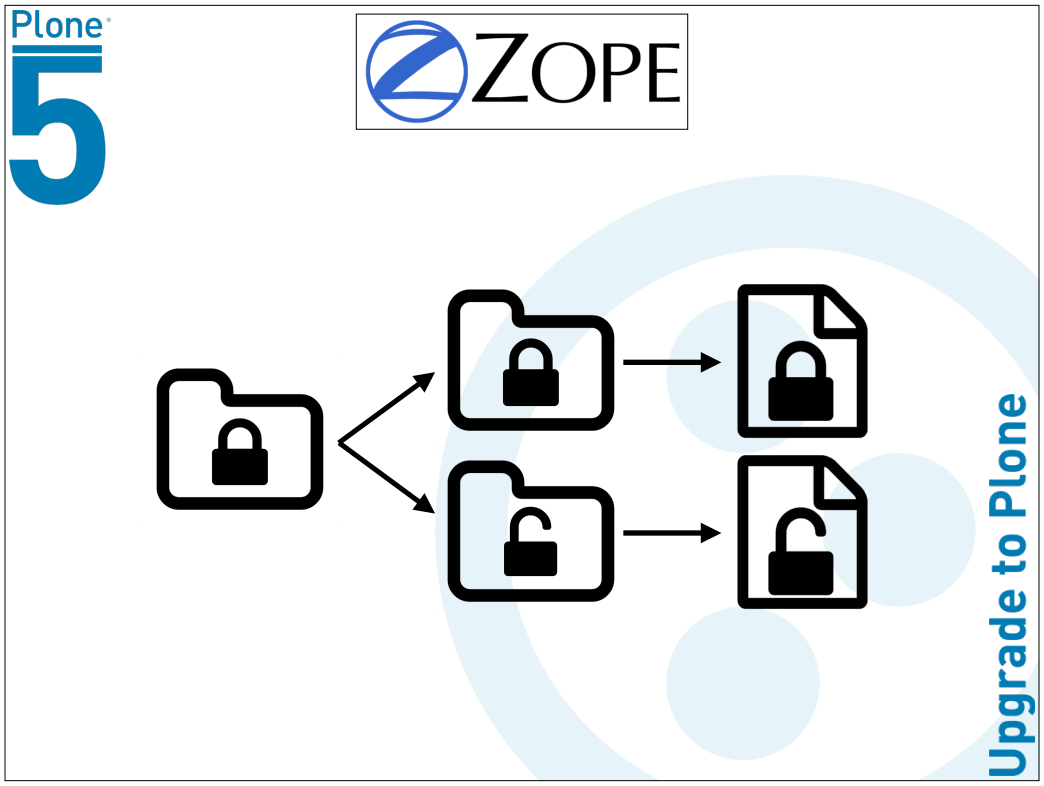
* We find the objects using traversal.
* Then we *call* the object, passing in the request (which contains environmental information) to generate a publishable representation of itself.
* Finally, we use that representation as the response we send back to the client.

While Zope was powered by traversal and object publication, there were a few other attributes that helped it to succeed.

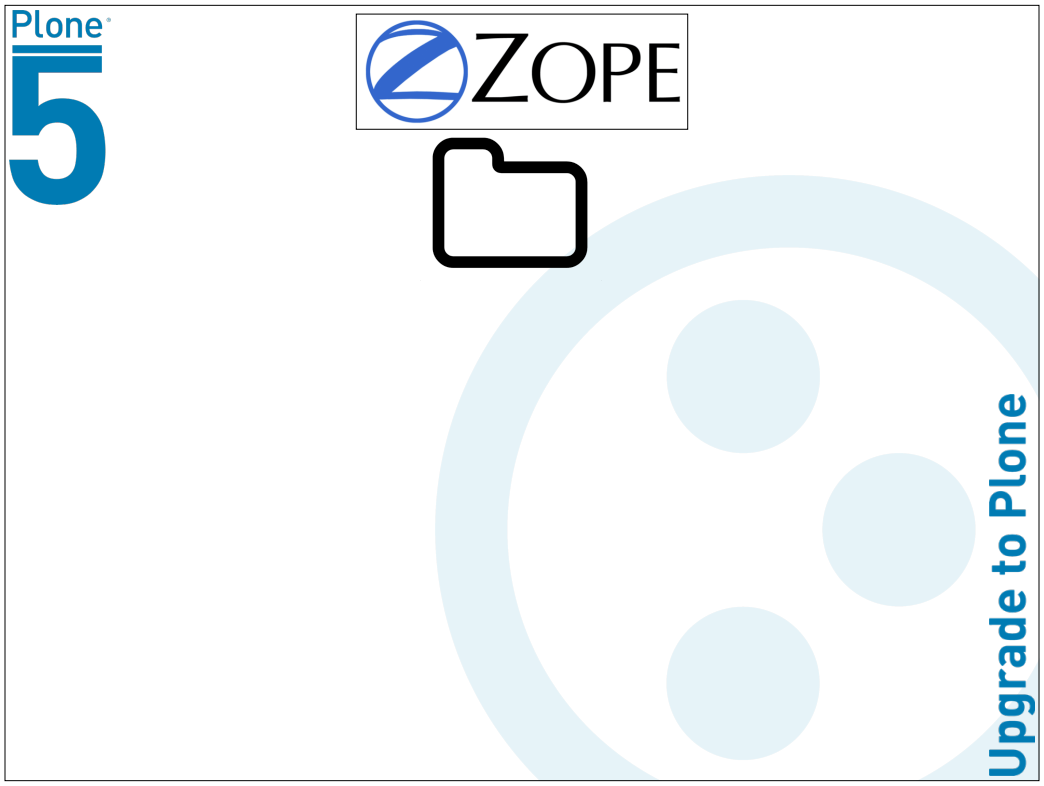
* Security was baked directly into the objects of Zope, not added as an extra layer
* Using a persistent graph of objects made sites with mixed types of content simple to build
* Containment allowed for flexible and fine-grained access controls

But perhaps most important was the idea of Through-The-Web development. This allowed new developers to build powerful applications with only a browser. It lowered the bar to get started in web development. You could explore the system and learn experientially.

Zope benefitted greatly from this easy on-ramp, much as Python itself benefits from it's simplicity. And like Python itself, once you learned, the system was plenty powerful and secure to build really outstanding projects.



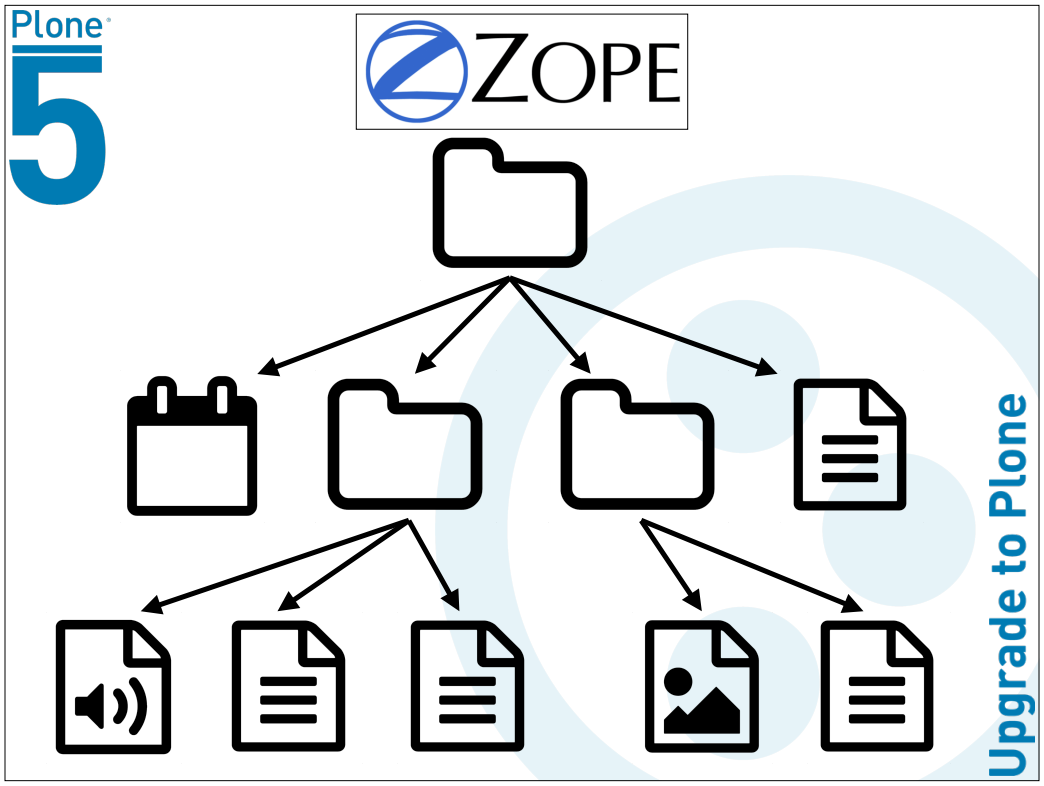
This combined with object containment to allow for flexible and fine-grained access controls



Using a persistent graph of Python objects made building sites with mixed content easy.

But perhaps most important was the idea of Through-The-Web development. This allowed new developers to build powerful applications with only a browser. It lowered the bar to get started in web development. You could explore the system and learn experientially.

Zope benefitted greatly from this easy on-ramp, much as Python itself benefits from it's simplicity. And like Python itself, once you learned, the system was plenty powerful and secure to build really outstanding projects.



Using a persistent graph of Python objects made building sites with mixed content easy.

But perhaps most important was the idea of Through-The-Web development. This allowed new developers to build powerful applications with only a browser. It lowered the bar to get started in web development. You could explore the system and learn experientially.

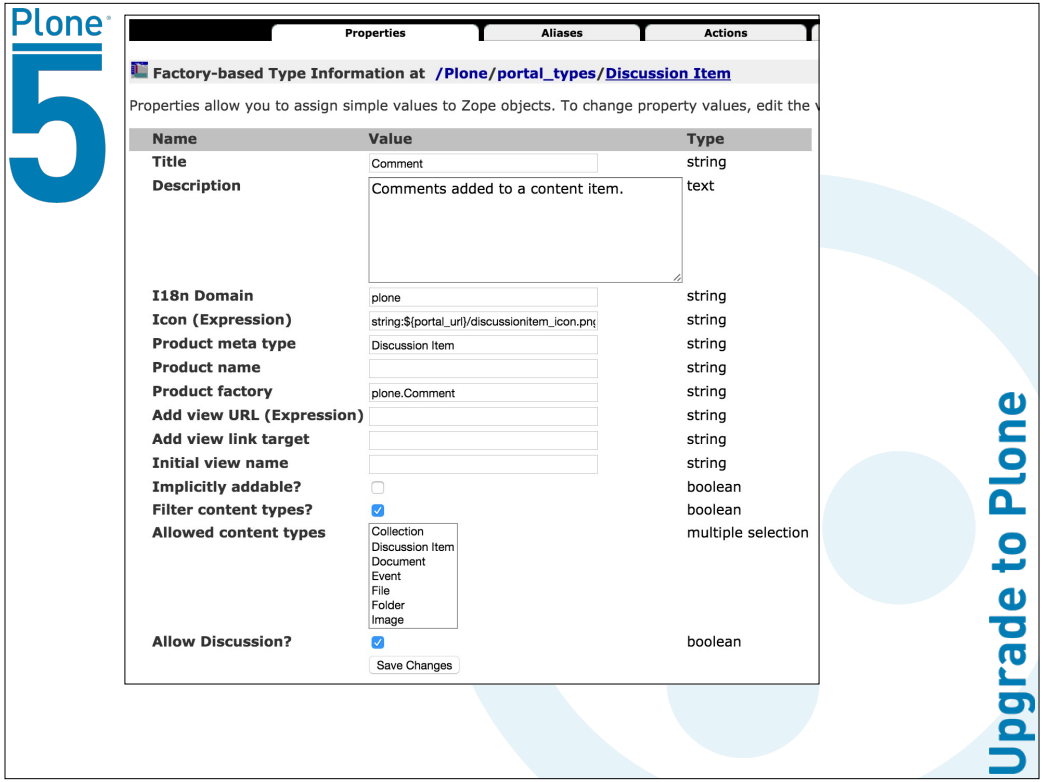
Zope benefitted greatly from this easy on-ramp, much as Python itself benefits from it's simplicity. And like Python itself, once you learned, the system was plenty powerful and secure to build really outstanding projects.

But perhaps most important was the idea of Through-The-Web development. Zope allowed new developers to build powerful applications with only a browser. It lowered the bar to get started in web development.

Zope benefitted greatly from this easy on-ramp, much as Python itself benefits from it's simplicity. And like Python the system was plenty powerful and secure to build really outstanding projects.
Soon after its open source release in 1998, Zope was being called "Python's killer app". Numerous applications were built on top of it.

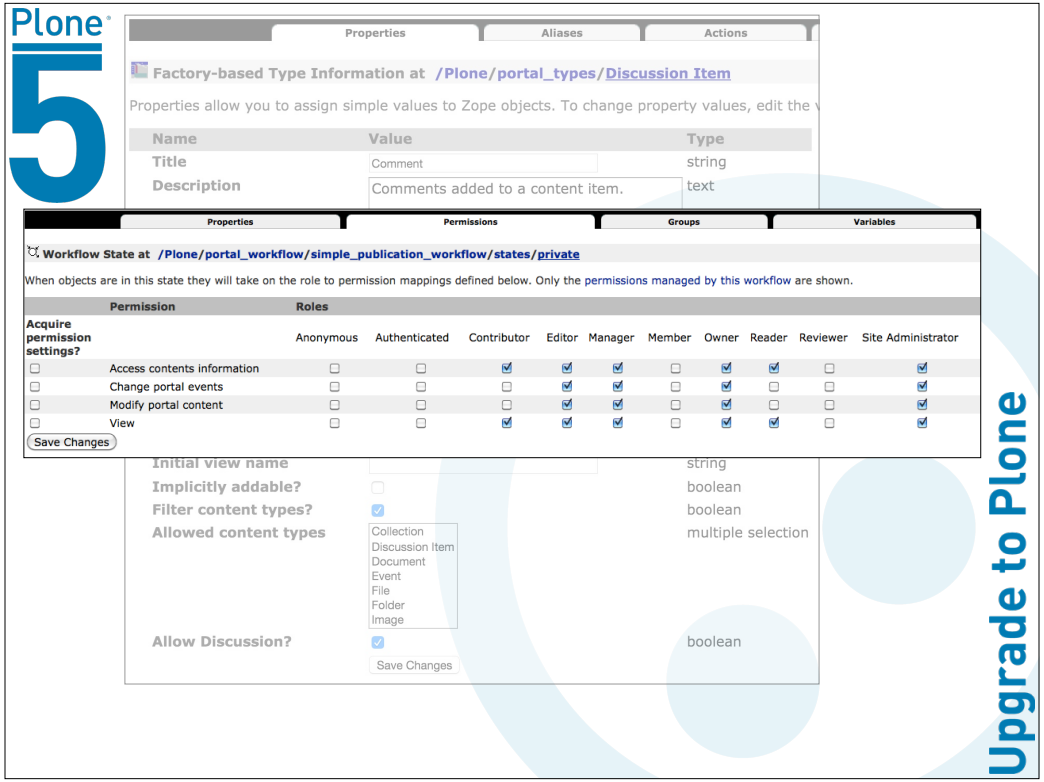
One of these was the Portal Toolkit, or Content Management Framework (aka CMF) architected by Tres Seaver. The CMF provided all sorts of great tools to create content, control its publication, set its display, add interactivity via user input and theme the resulting web application.

But the CMF wasn't particularly nice to look at, particularly for non-technical users.



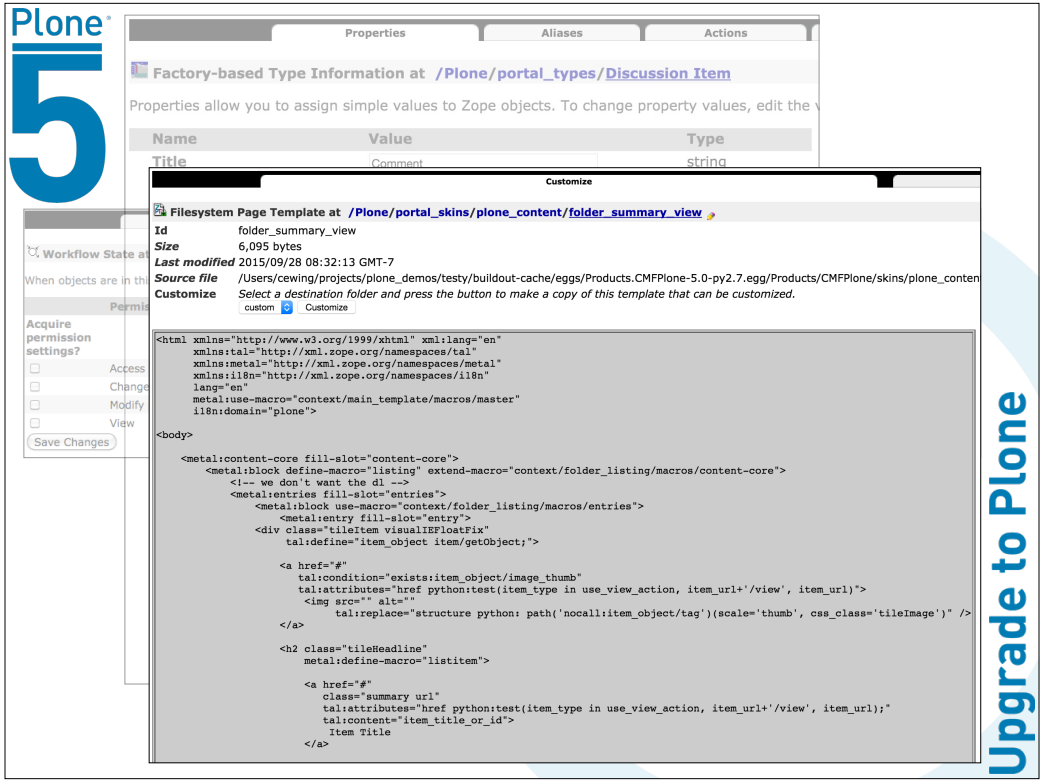
One of these was the Portal Toolkit, or Content Management Framework (aka CMF) architected by Tres Seaver. The CMF provided all sorts of great tools to create content, control its publication, set its display, add interactivity via user input and theme the resulting web application.

But the CMF wasn’t particularly nice to look at, particularly for non-technical users.



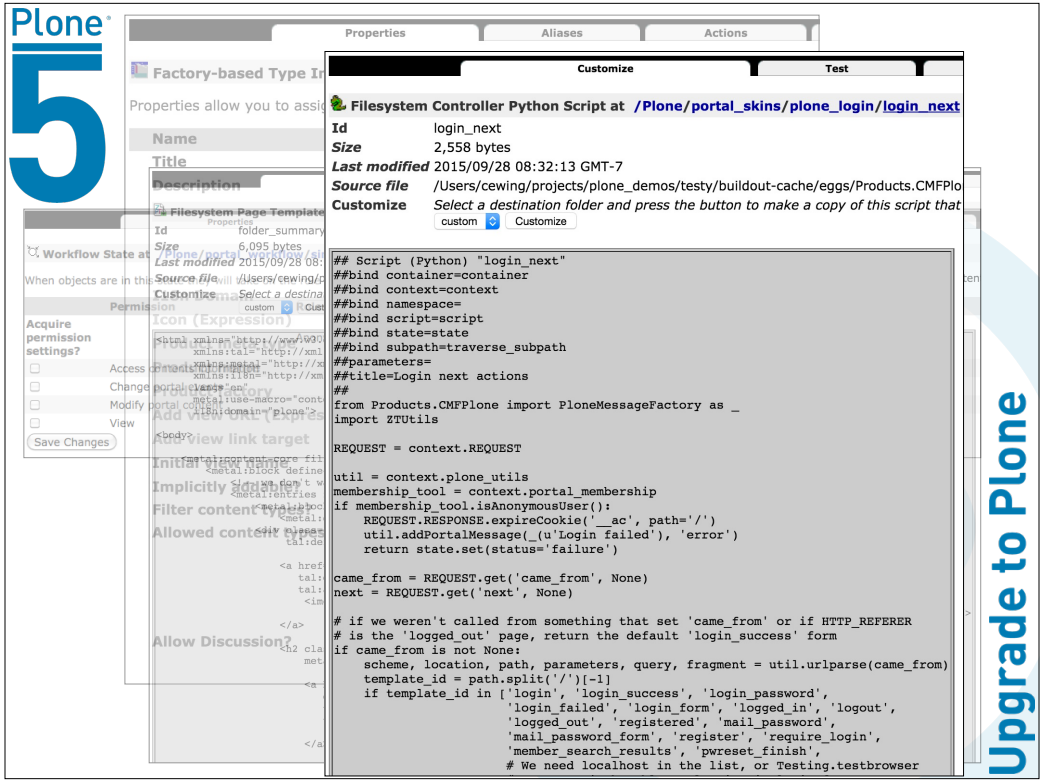
One of these was the Portal Toolkit, or Content Management Framework (aka CMF) architected by Tres Seaver. The CMF provided all sorts of great tools to create content, control its publication, set its display, add interactivity via user input and theme the resulting web application.

But the CMF wasn’t particularly nice to look at, particularly for non-technical users.



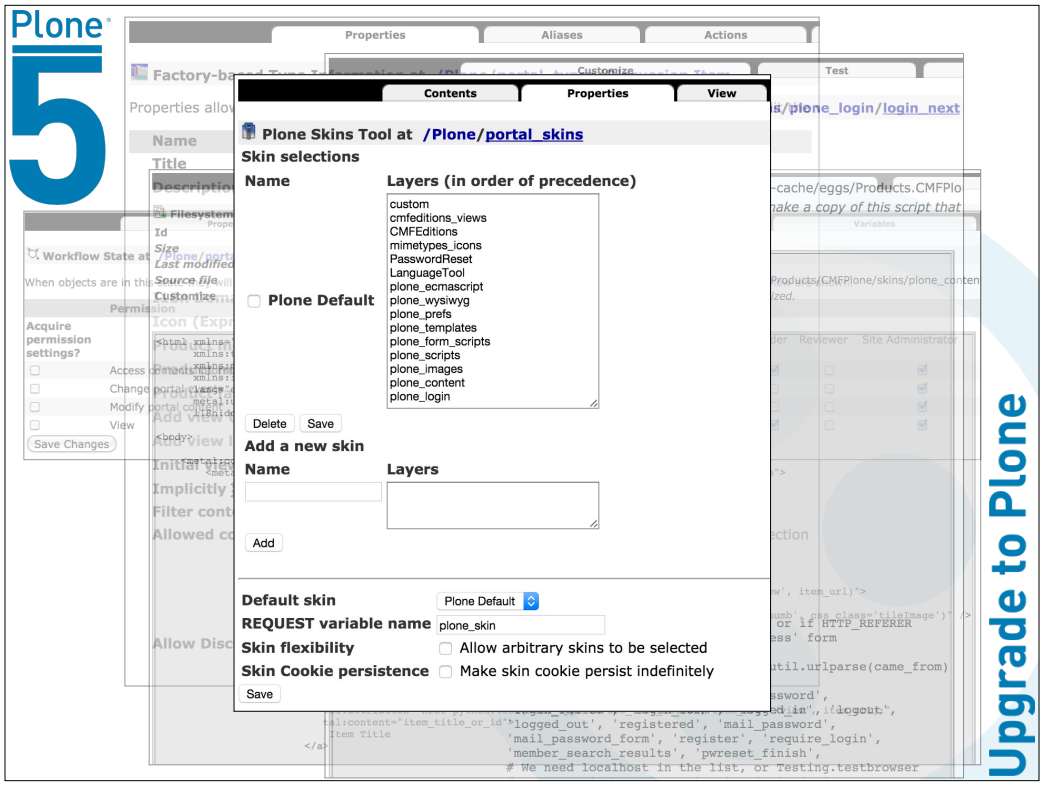
One of these was the Portal Toolkit, or Content Management Framework (aka CMF) architected by Tres Seaver. The CMF provided all sorts of great tools to create content, control its publication, set its display, add interactivity via user input and theme the resulting web application.

But the CMF wasn’t particularly nice to look at, particularly for non-technical users.



One of these was the Portal Toolkit, or Content Management Framework (aka CMF) architected by Tres Seaver. The CMF provided all sorts of great tools to create content, control its publication, set its display, add interactivity via user input and theme the resulting web application.

But the CMF wasn't particularly nice to look at, particularly for non-technical users.



One of these was the Portal Toolkit, or Content Management Framework (aka CMF) architected by Tres Seaver. The CMF provided all sorts of great tools to create content, control its publication, set its display, add interactivity via user input and theme the resulting web application.

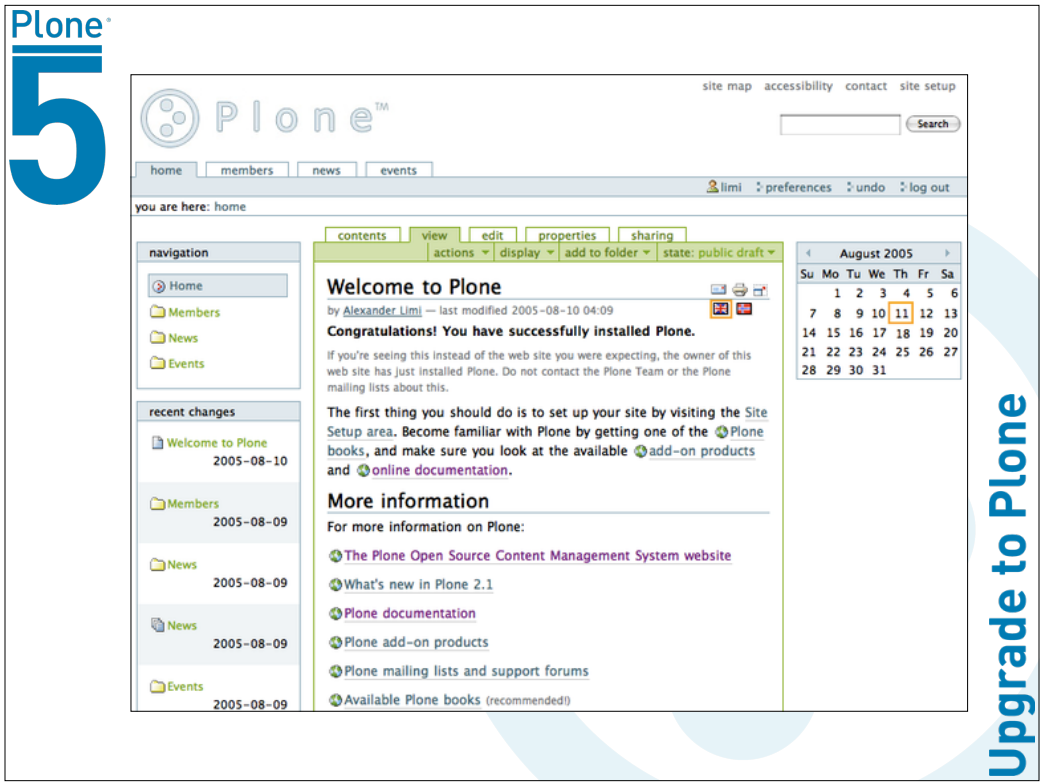
But the CMF wasn't particularly nice to look at, particularly for non-technical users.

In 1999 two guys (alex limi and alan runyan) met in the Zope IRC channel. Both were interested in improving the usability of the Content Management Framework (CMF). They created CMFPlone as a CMF theme package with this goal in mind.

They must have done a good job, because after it's first public release in October of 2001, Plone quickly gained users and mindshare. It's most distinguishing feature was in-place content creation. Users could navigate with their browser to the place they wanted an item, and then

* add it
* edit it
* change how it looked
* allow access to it
* and publish it right there.

There was no "backend" to learn, which made it easy for the average person to learn. The strong security model Plone inherited from Zope allowed websites to mix private and public content. This allowed organizations to combine their separate intranet and extranet into a single seamless website.

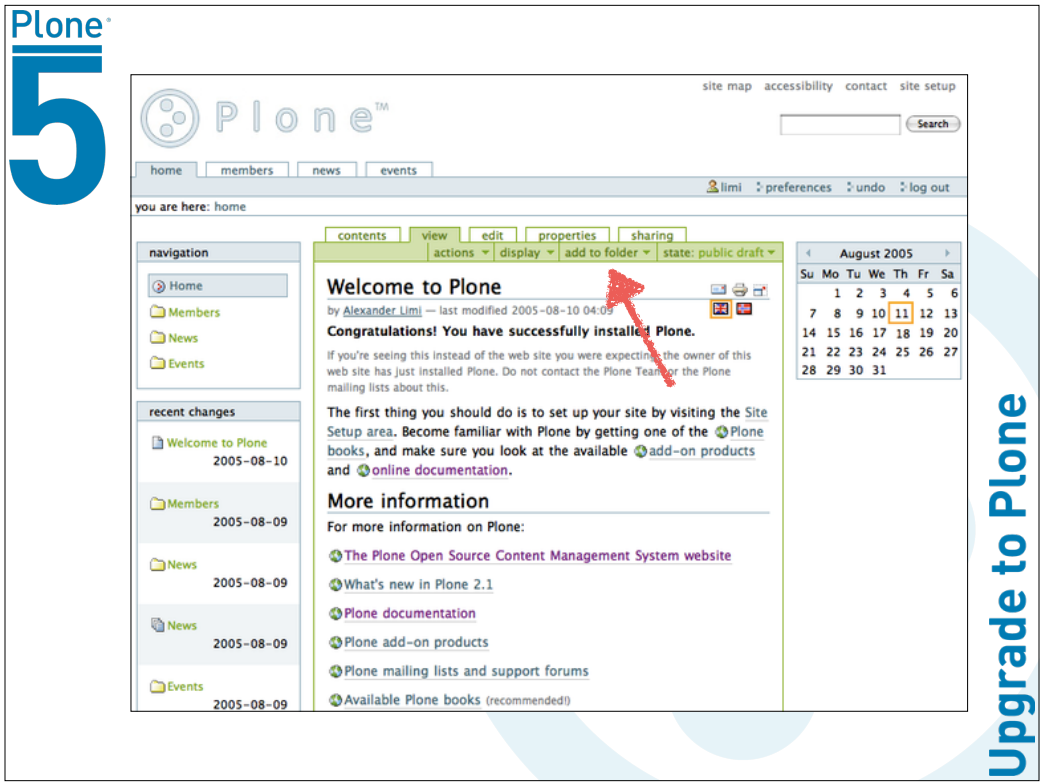


In 1999 two guys (alex limi and alan runyan) met in the Zope IRC channel. Both were interested in improving the usability of the Content Management Framework (CMF). They created CMFPlone as a CMF theme package with this goal in mind.

They must have done a good job, because after it's first public release in October of 2001, Plone quickly gained users and mindshare. It's most distinguishing feature was in-place content creation. Users could navigate with their browser to the place they wanted an item, and then

* add it
* edit it
* change how it looked
* allow access to it
* and publish it right there.

There was no "backend" to learn, which made it easy for the average person to learn. The strong security model Plone inherited from Zope allowed websites to mix private and public content. This allowed organizations to combine their separate intranet and extranet into a single seamless website.

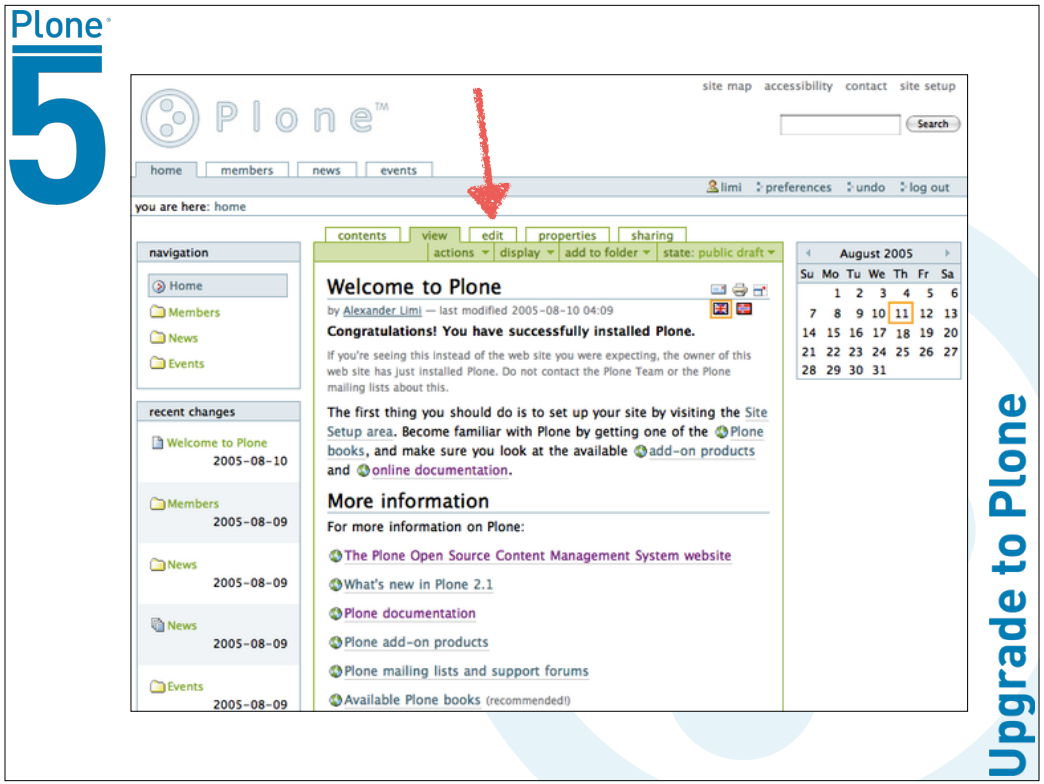


In 1999 two guys (alex limi and alan runyan) met in the Zope IRC channel. Both were interested in improving the usability of the Content Management Framework (CMF). They created CMFPlone as a CMF theme package with this goal in mind.

They must have done a good job, because after it's first public release in October of 2001, Plone quickly gained users and mindshare. It's most distinguishing feature was in-place content creation. Users could navigate with their browser to the place they wanted an item, and then

* add it
* edit it
* change how it looked
* allow access to it
* and publish it right there.

There was no "backend" to learn, which made it easy for the average person to learn. The strong security model Plone inherited from Zope allowed websites to mix private and public content. This allowed organizations to combine their separate intranet and extranet into a single seamless website.

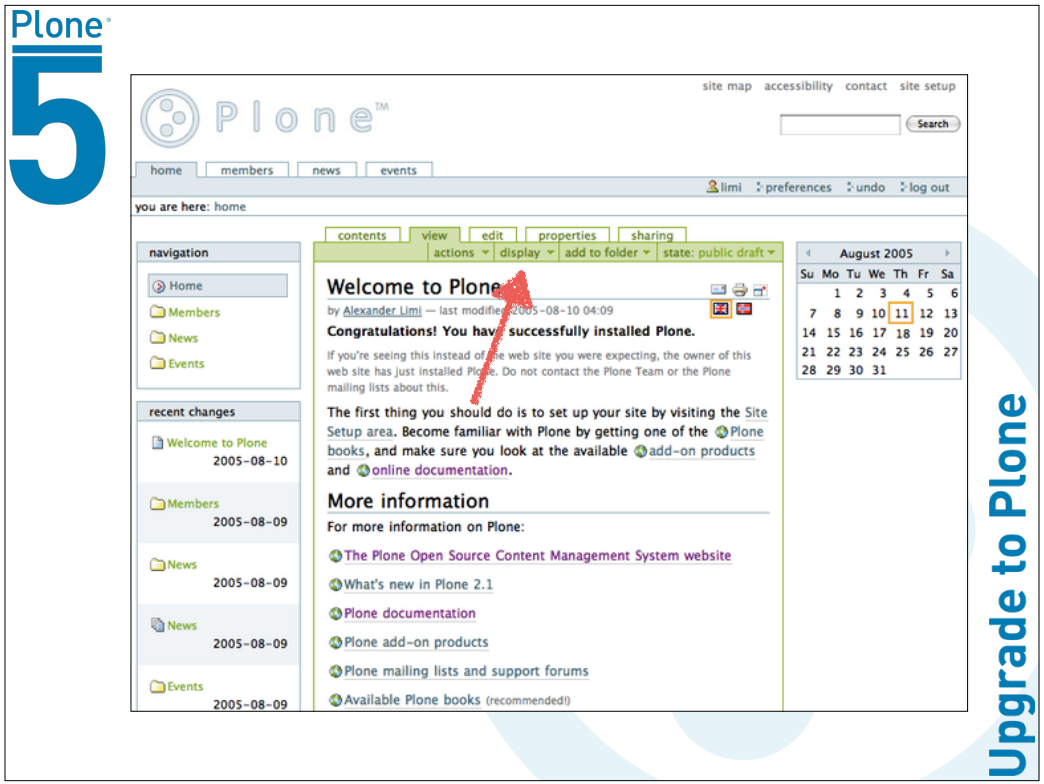


In 1999 two guys (alex limi and alan runyan) met in the Zope IRC channel. Both were interested in improving the usability of the Content Management Framework (CMF). They created CMFPlone as a CMF theme package with this goal in mind.

They must have done a good job, because after it's first public release in October of 2001, Plone quickly gained users and mindshare. It's most distinguishing feature was in-place content creation. Users could navigate with their browser to the place they wanted an item, and then

* add it
* edit it
* change how it looked
* allow access to it
* and publish it right there.

There was no "backend" to learn, which made it easy for the average person to learn. The strong security model Plone inherited from Zope allowed websites to mix private and public content. This allowed organizations to combine their separate intranet and extranet into a single seamless website.

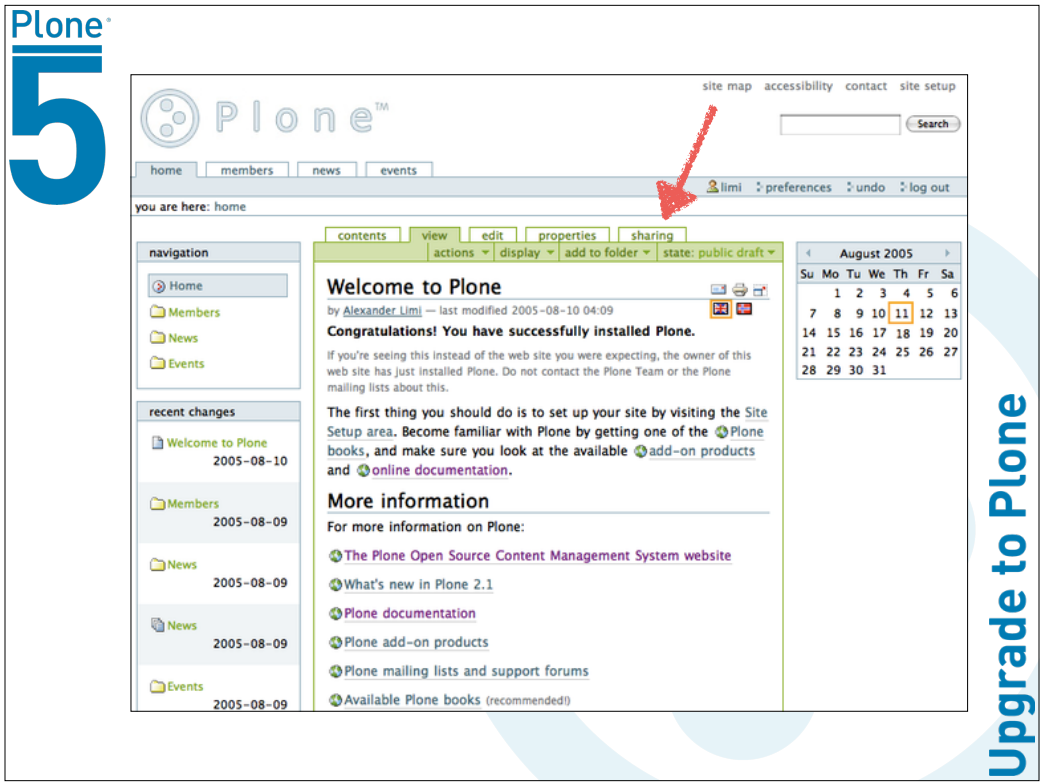


In 1999 two guys (alex limi and alan runyan) met in the Zope IRC channel. Both were interested in improving the usability of the Content Management Framework (CMF). They created CMFPlone as a CMF theme package with this goal in mind.

They must have done a good job, because after it's first public release in October of 2001, Plone quickly gained users and mindshare. It's most distinguishing feature was in-place content creation. Users could navigate with their browser to the place they wanted an item, and then

* add it
* edit it
* change how it looked
* allow access to it
* and publish it right there.

There was no "backend" to learn, which made it easy for the average person to learn. The strong security model Plone inherited from Zope allowed websites to mix private and public content. This allowed organizations to combine their separate intranet and extranet into a single seamless website.

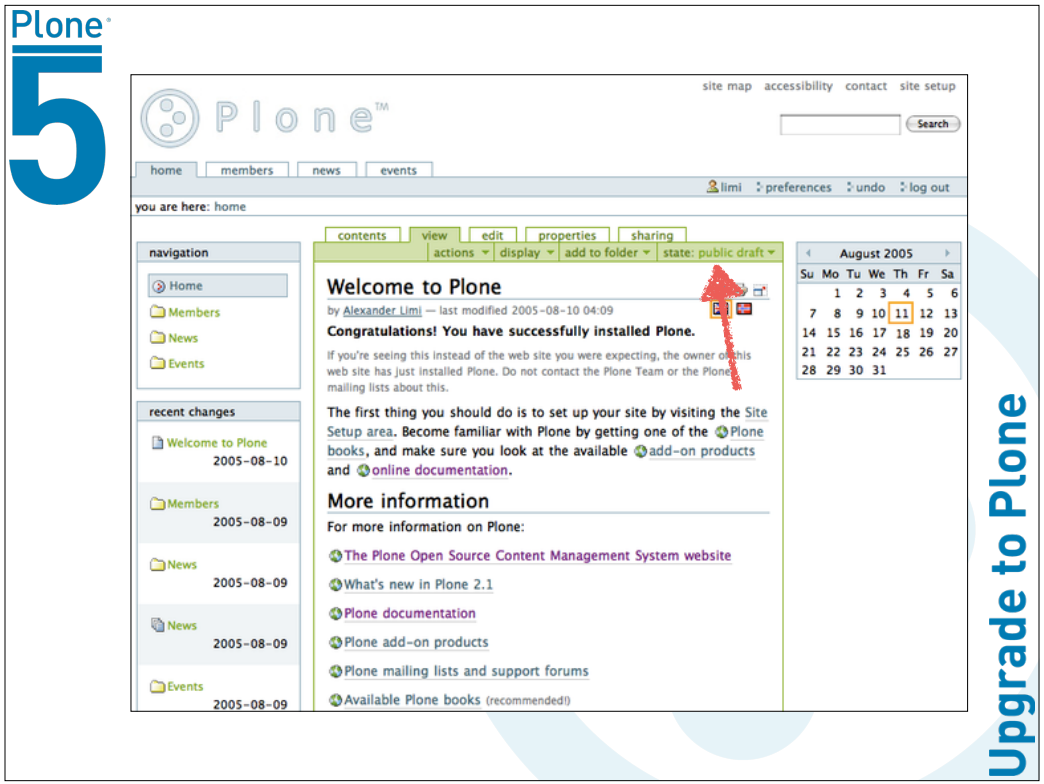


In 1999 two guys (alex limi and alan runyan) met in the Zope IRC channel. Both were interested in improving the usability of the Content Management Framework (CMF). They created CMFPlone as a CMF theme package with this goal in mind.

They must have done a good job, because after it's first public release in October of 2001, Plone quickly gained users and mindshare. It's most distinguishing feature was in-place content creation. Users could navigate with their browser to the place they wanted an item, and then

* add it
* edit it
* change how it looked
* allow access to it
* and publish it right there.

There was no "backend" to learn, which made it easy for the average person to learn. The strong security model Plone inherited from Zope allowed websites to mix private and public content. This allowed organizations to combine their separate intranet and extranet into a single seamless website.

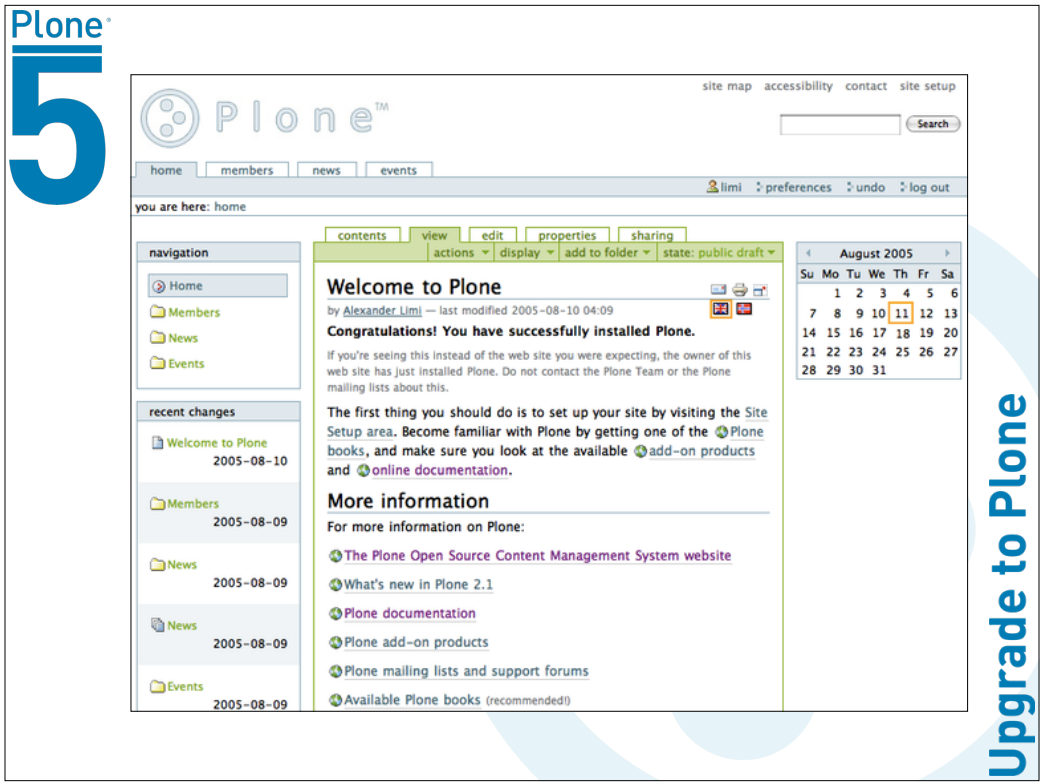


In 1999 two guys (alex limi and alan runyan) met in the Zope IRC channel. Both were interested in improving the usability of the Content Management Framework (CMF). They created CMFPlone as a CMF theme package with this goal in mind.

They must have done a good job, because after it's first public release in October of 2001, Plone quickly gained users and mindshare. It's most distinguishing feature was in-place content creation. Users could navigate with their browser to the place they wanted an item, and then

* add it
* edit it
* change how it looked
* allow access to it
* and publish it right there.

There was no "backend" to learn, which made it easy for the average person to learn. The strong security model Plone inherited from Zope allowed websites to mix private and public content. This allowed organizations to combine their separate intranet and extranet into a single seamless website.

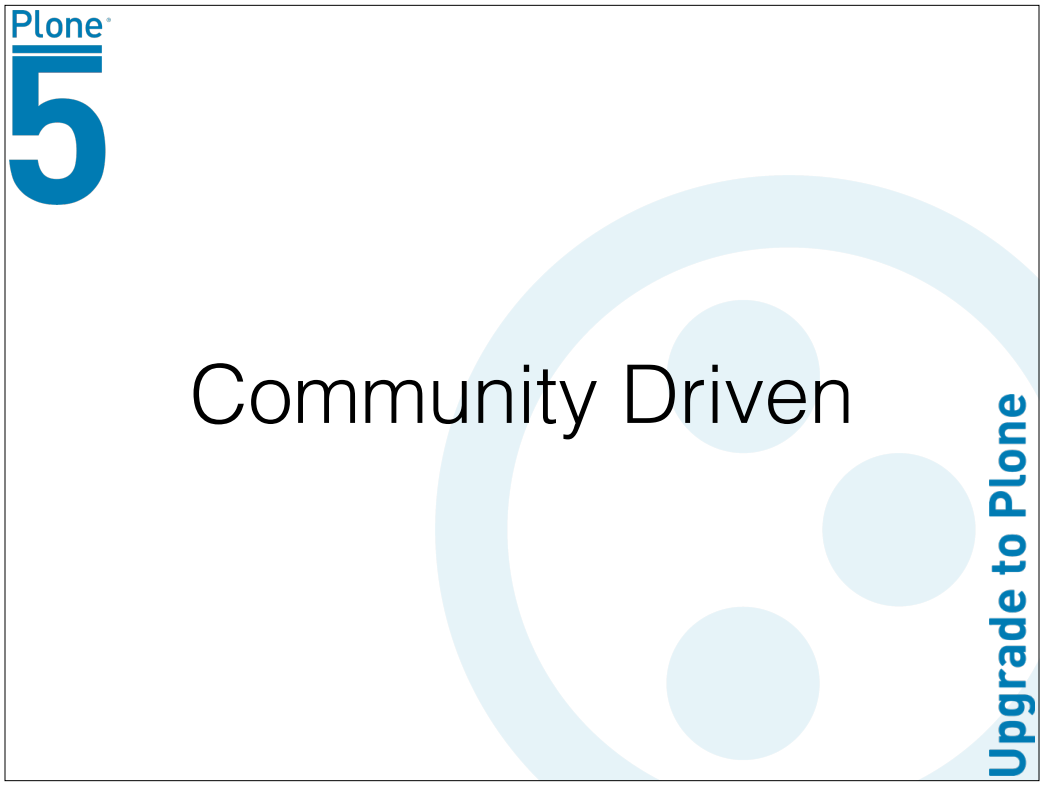


In 1999 two guys (alex limi and alan runyan) met in the Zope IRC channel. Both were interested in improving the usability of the Content Management Framework (CMF). They created CMFPlone as a CMF theme package with this goal in mind.

They must have done a good job, because after it's first public release in October of 2001, Plone quickly gained users and mindshare. It's most distinguishing feature was in-place content creation. Users could navigate with their browser to the place they wanted an item, and then

* add it
* edit it
* change how it looked
* allow access to it
* and publish it right there.

There was no "backend" to learn, which made it easy for the average person to learn. The strong security model Plone inherited from Zope allowed websites to mix private and public content. This allowed organizations to combine their separate intranet and extranet into a single seamless website.



A few months after the first release of Plone in early 2002, Zope Corporation held the first-ever community development sprint. The goal was to build internationalization support for Zope applications. Participants in the sprint went home afterward with increased enthusiasm for Zope.

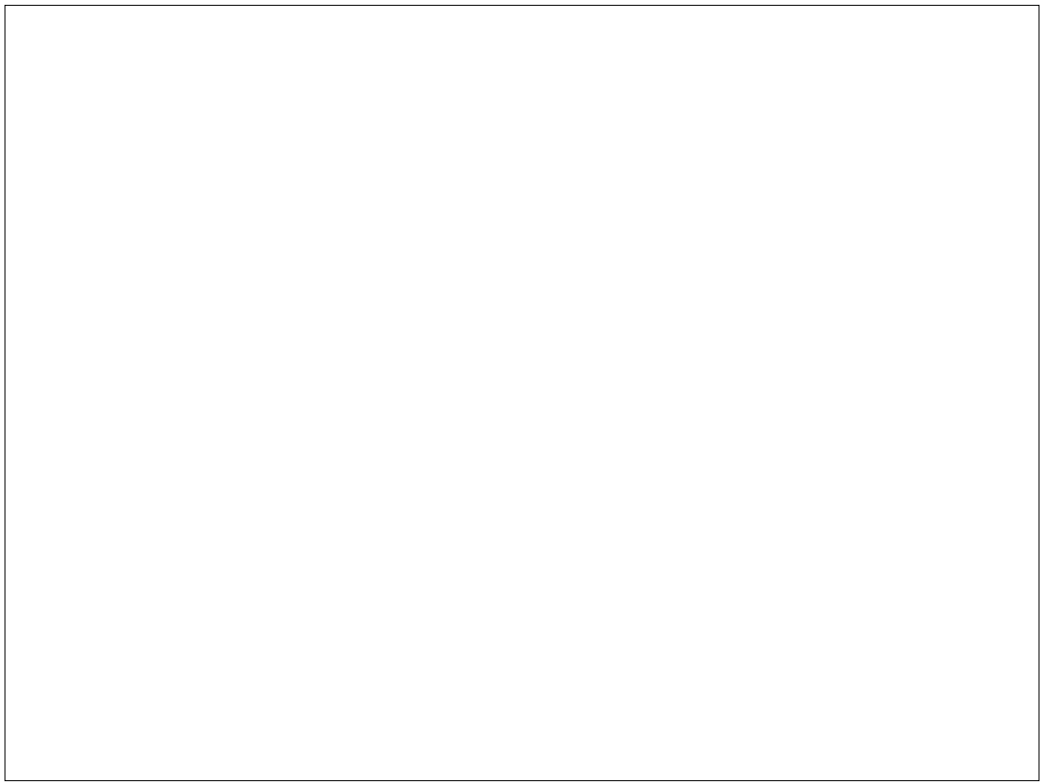
The Plone community saw this and followed suit, holding the first Plone sprint in Berne, Switzerland the next year, 2003. This began a long tradition of community driven development. Held in fantastic locations like a Norwegian archipelago, an alpine mountaintop, an Italian village, coastal beach houses and even an Austrian castle, these sprints have served to move Plone forward and keep the community vital.

A few months after the first release of Plone in early 2002, Zope Corporation held the first-ever community development sprint. The goal was to build internationalization support for Zope applications. Participants in the sprint went home afterward with increased enthusiasm for Zope.

The Plone community saw this and followed suit, holding the first Plone sprint in Berne, Switzerland the next year, 2003. This began a long tradition of community driven development. Held in fantastic locations like a Norwegian archipelago, an alpine mountaintop, an Italian village, coastal beach houses and even an Austrian castle, these sprints have served to move Plone forward and keep the community vital.

A few months after the first release of Plone in early 2002, Zope Corporation held the first-ever community development sprint. The goal was to build internationalization support for Zope applications. Participants in the sprint went home afterward with increased enthusiasm for Zope.

The Plone community saw this and followed suit, holding the first Plone sprint in Berne, Switzerland the next year, 2003. This began a long tradition of community driven development. Held in fantastic locations like a Norwegian archipelago, an alpine mountaintop, an Italian village, coastal beach houses and even an Austrian castle, these sprints have served to move Plone forward and keep the community vital.



2003 also saw the first annual Plone conference. The conference combined sprints with training and talks, and offered the community of Plonistas a way to keep abreast of the latest developments, share knowledge, connect with each-other, and add new Plonistas.

It’s a tradition that continues to this day.

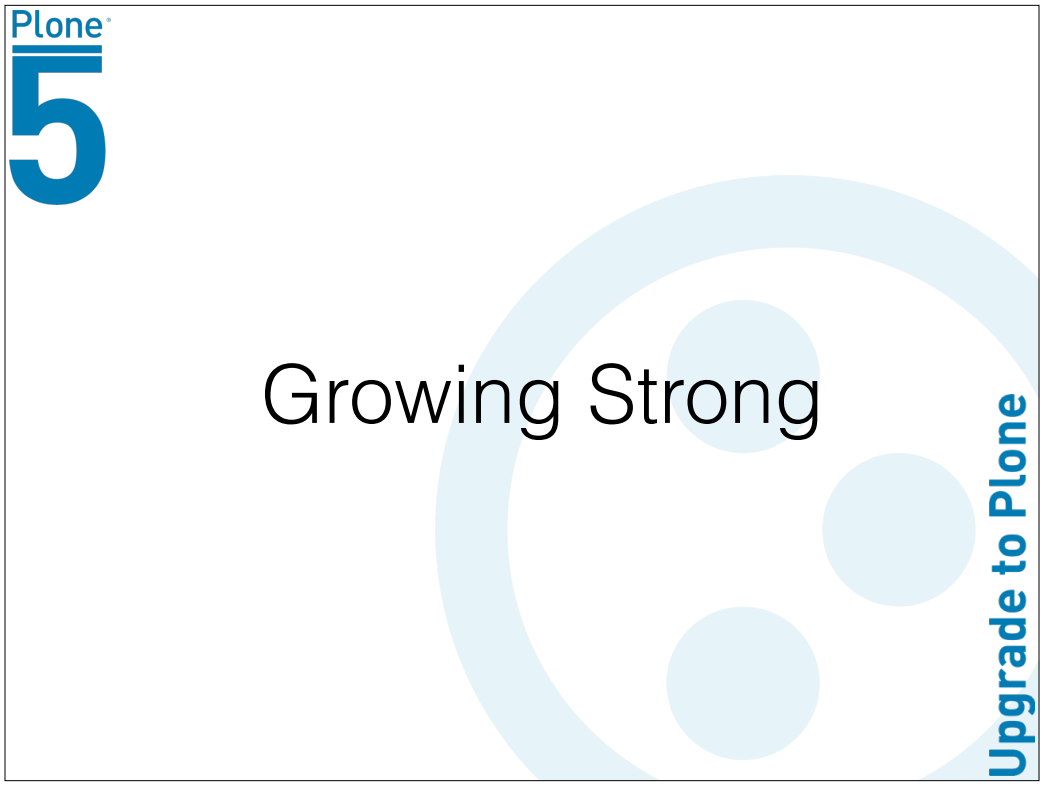
2003 also saw the first annual Plone conference. The conference combined sprints with training and talks, and offered the community of Plonistas a way to keep abreast of the latest developments, share knowledge, connect with each-other, and add new Plonistas.

It’s a tradition that continues to this day.

2003 also saw the first annual Plone conference. The conference combined sprints with training and talks, and offered the community of Plonistas a way to keep abreast of the latest developments, share knowledge, connect with each-other, and add new Plonistas.

It’s a tradition that continues to this day.

There's no better place than the Plone conference, in Boston this October, to learn about the exciting system we're discussing today. Perhaps if you're interested in the technologies we discuss here, you might join us.



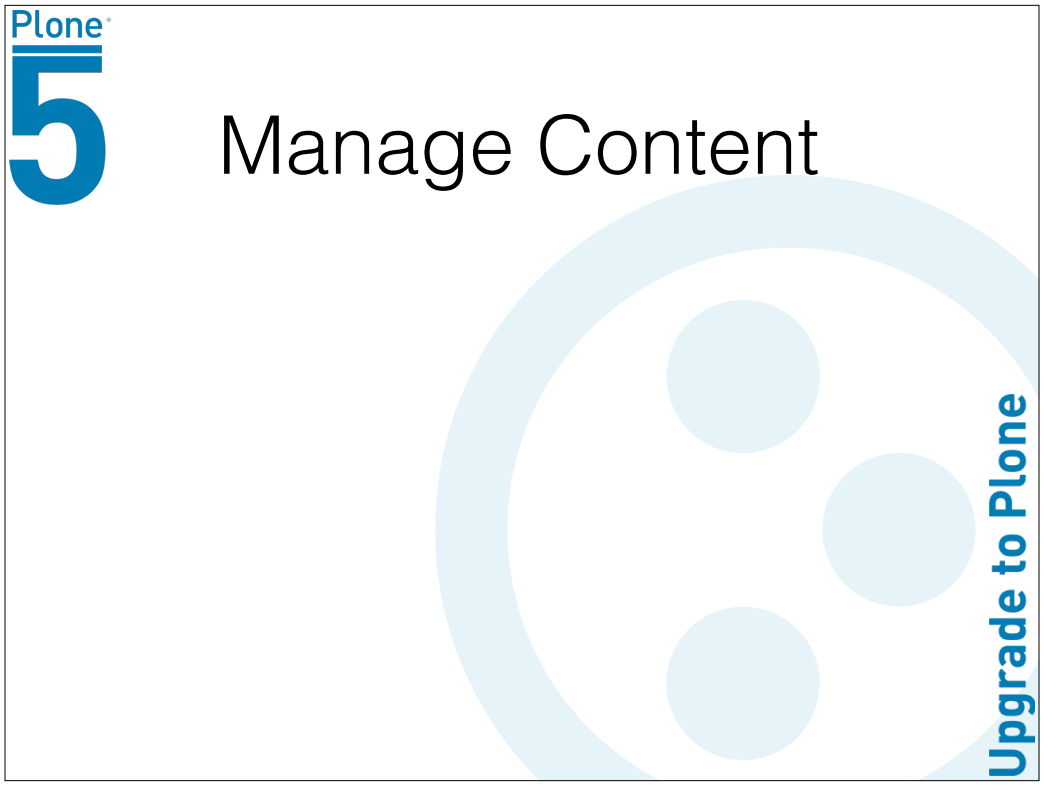
We Plonistas combine frequent sprints and annual conferences with an open policy toward contributions and a commitment to inclusion. This has helped the community to grow strong. More than 900 developers, designers, UI experts, and documentation writers have committed to core Plone. It is among the largest open source projects in the world.

We very much appreciate the opportunity to come here show you our pride and joy, and we hope you'll try us out when you need managed content.

We Plonistas combine frequent sprints and annual conferences with an open policy toward contributions and a commitment to inclusion. This has helped the community to grow strong. More than 900 developers, designers, UI experts, and documentation writers have committed to core Plone. It is among the largest open source projects in the world.

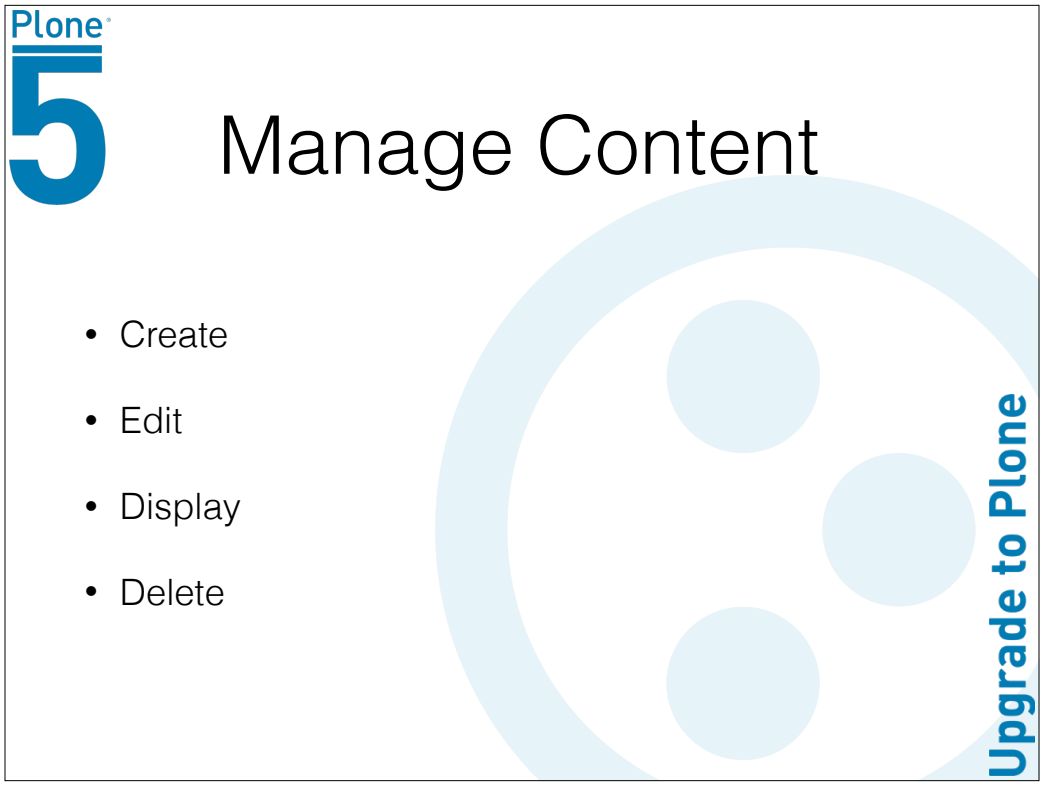
We very much appreciate the opportunity to come here show you our pride and joy, and we hope you'll try us out when you need managed content.

Plone is a Content Management System, or CMS. But what exactly does that mean?



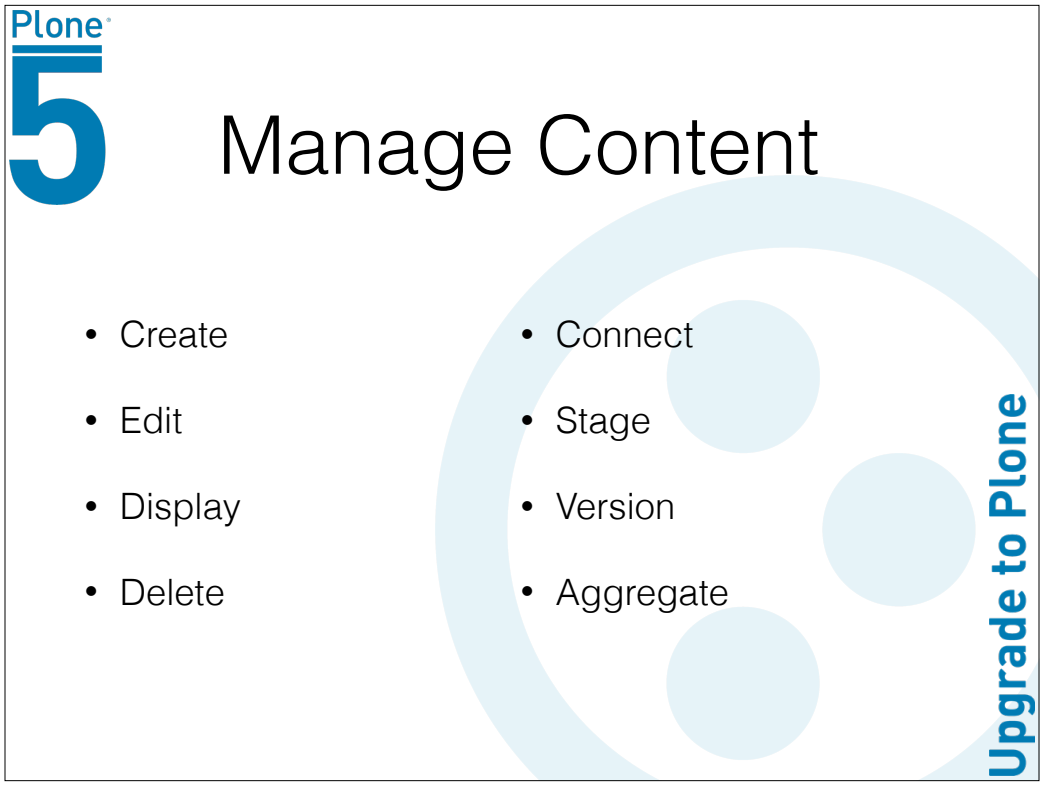
At a basic level, a CMS should provide tools to Create, Edit, Display and Delete content.

A more complex CMS might offer tools to interconnect content, to stage and version items or to aggregate things.



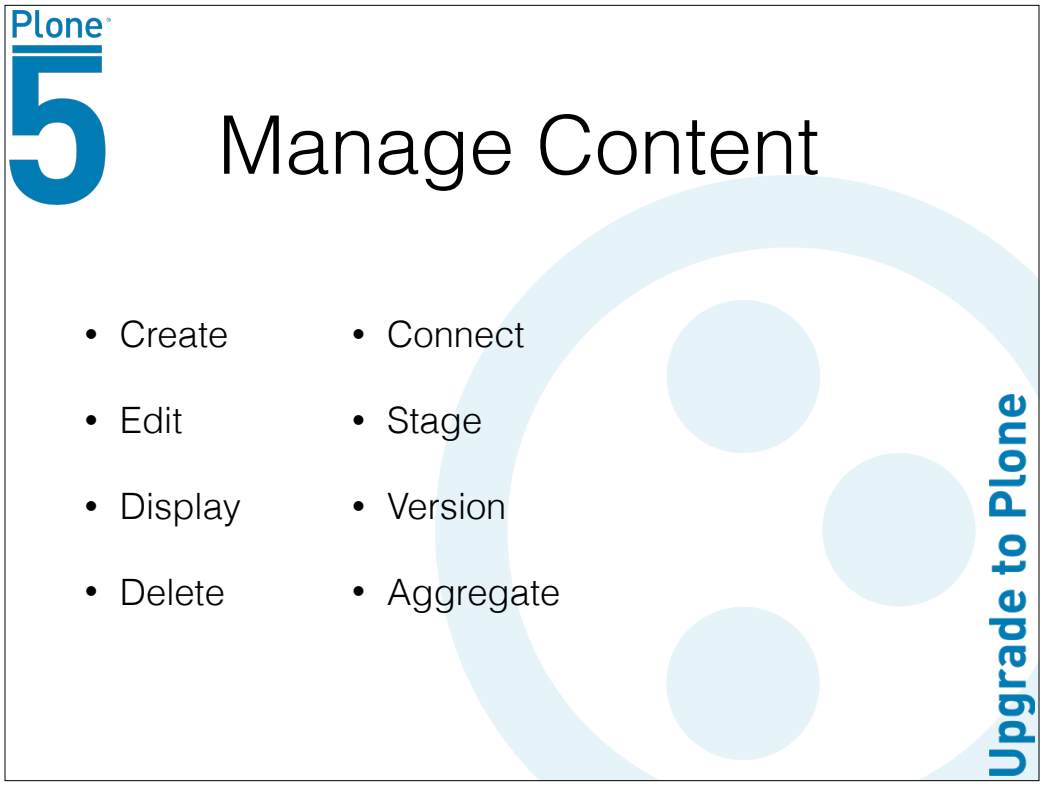
At a basic level, a CMS should provide tools to Create, Edit, Display and Delete content.

A more complex CMS might offer tools to interconnect content, to stage and version items or to aggregate things.

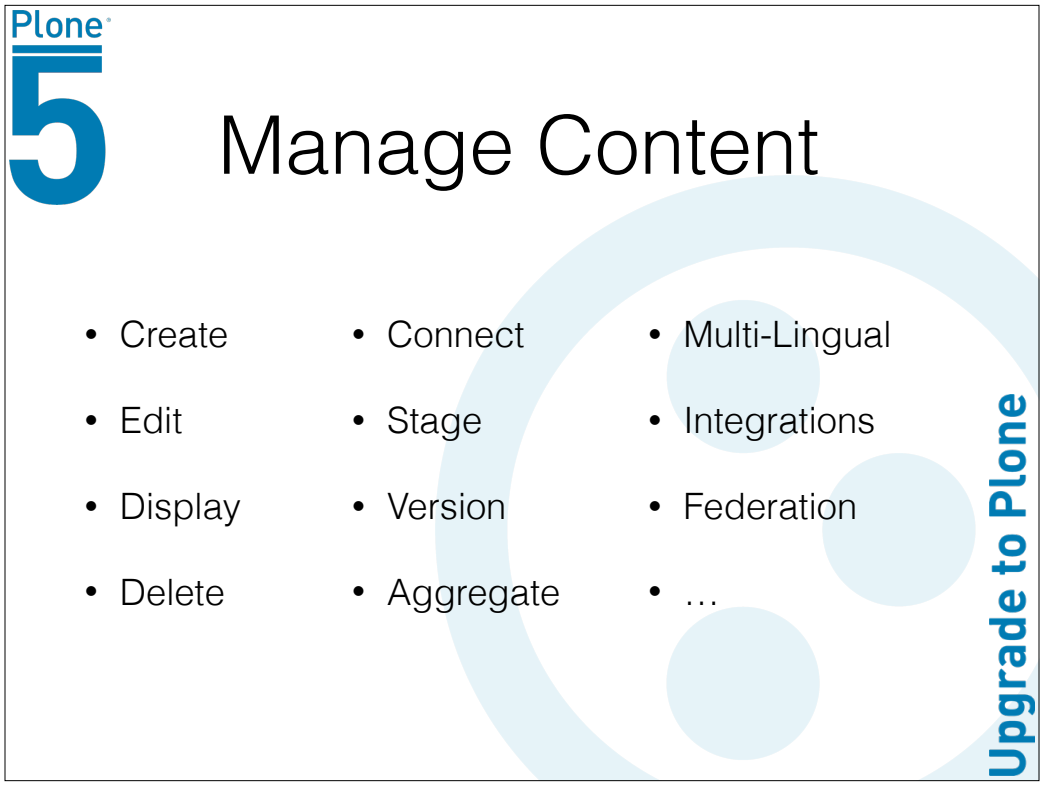


At a basic level, a CMS should provide tools to Create, Edit, Display and Delete content.

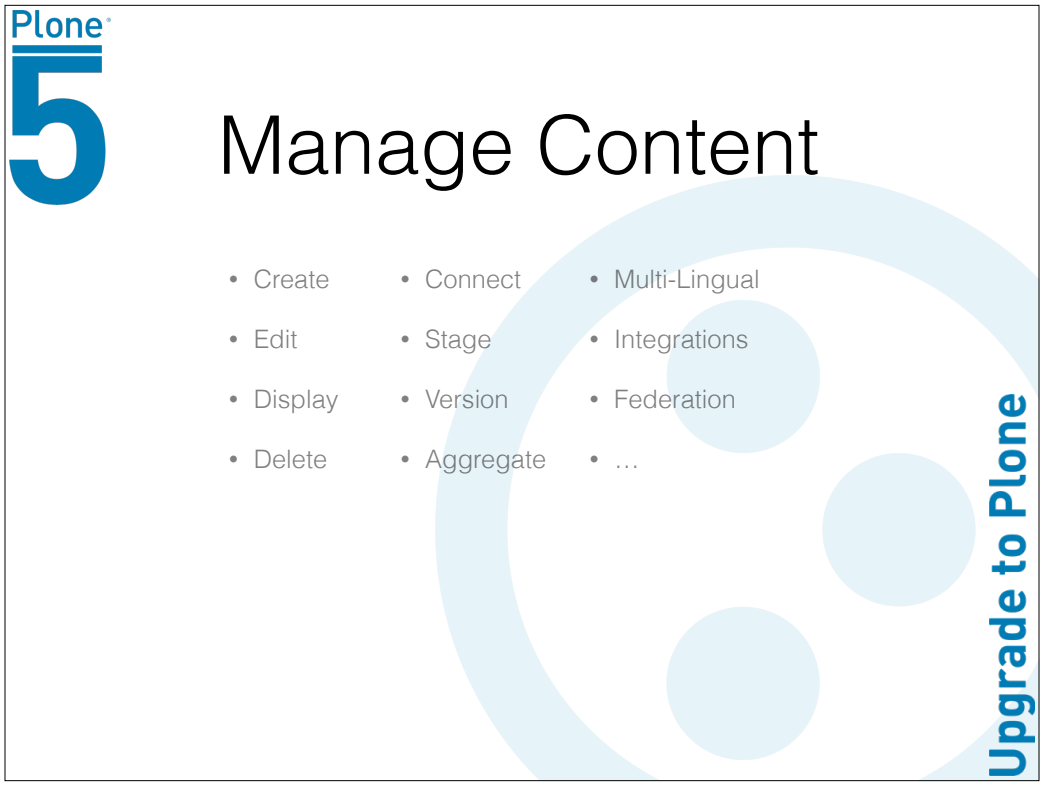
A more complex CMS might offer tools to interconnect content, to stage and version items or to aggregate things.



A truly powerful CMS will offer more: translations, integrations, federated authentication, the options are nearly endless.

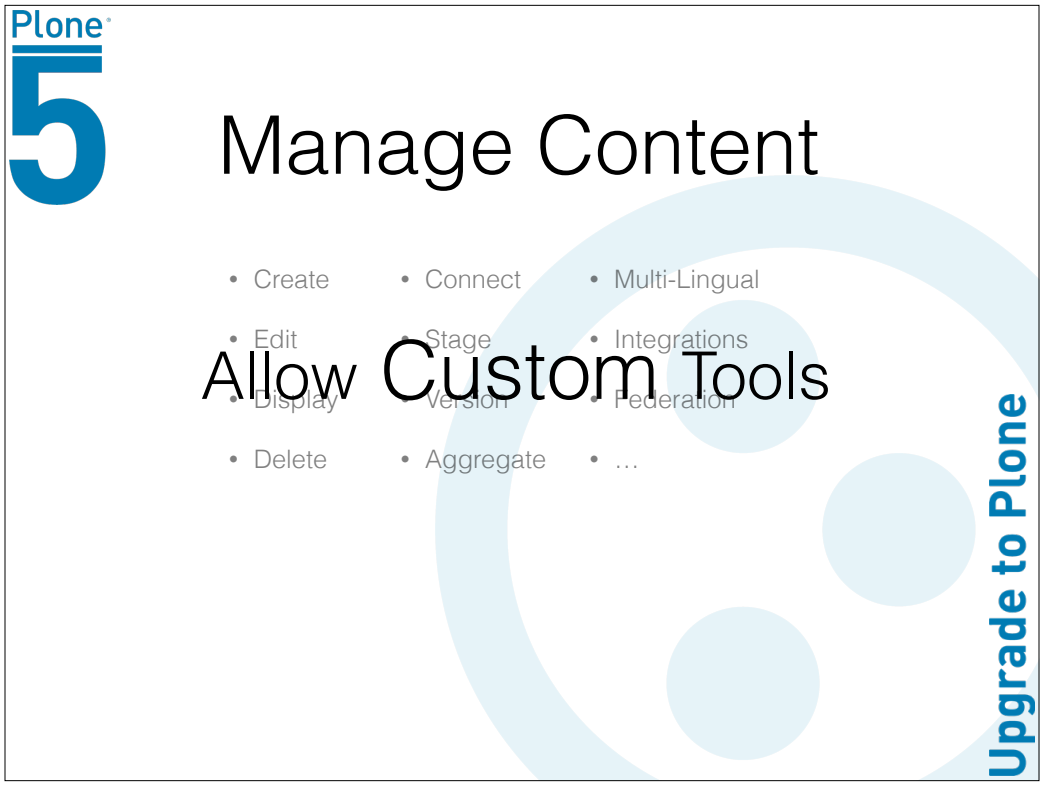


A truly powerful CMS will offer more: translations, integrations, federated authentication, the options are nearly endless.



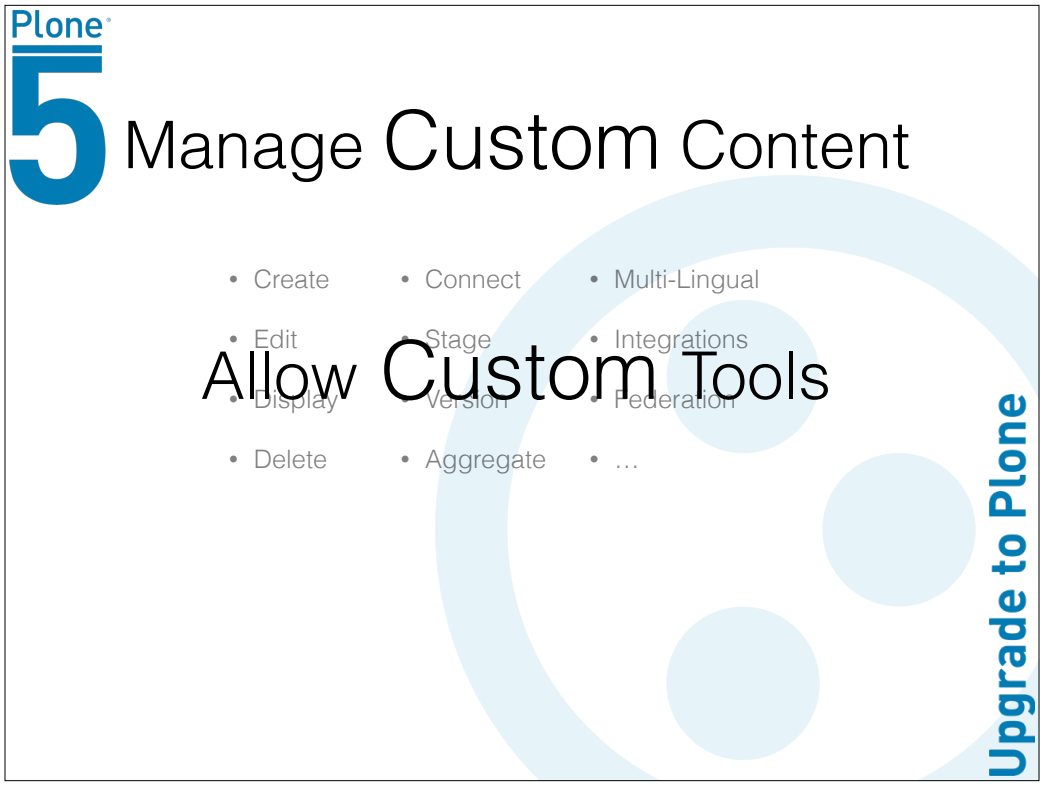
A CMS needs to be easily pluggable, so you can add the tools you need when you need them.

And how about that content you're managing? A good CMS will provide some common content types, but there's no real way to provide for all possible use cases.



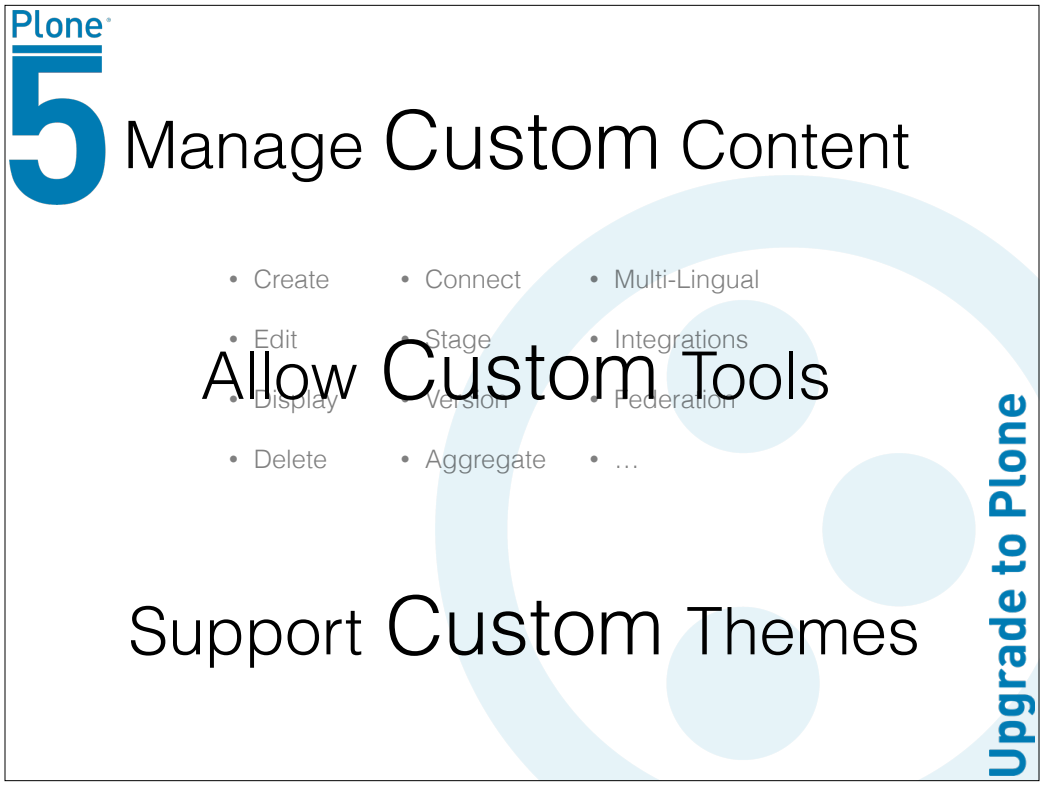
A CMS needs to be easily pluggable, so you can add the tools you need when you need them.

And how about that content you're managing? A good CMS will provide some common content types, but there's no real way to provide for all possible use cases.



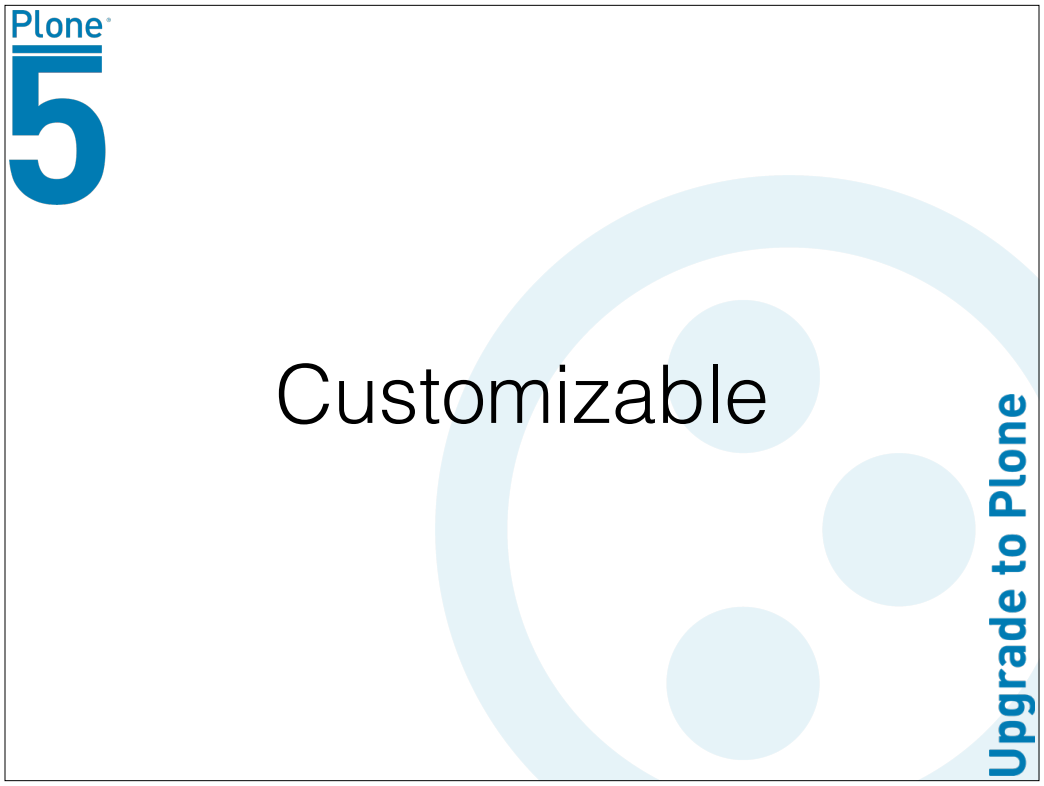
So a good system should allow easily creating custom content types.

And hey, since it's *your* content we are talking about here, you probably want some control over how it looks, so a good CMS has to support easily creating custom themes.



So a good system should allow easily creating custom content types.

And hey, since it's *your* content we are talking about here, you probably want some control over how it looks, so a good CMS has to support easily creating custom themes.



In short, you want a system that's easily customizable.

Having been built on top of Zope and the CMF, Plone provided this. You could even make changes through the web.

"warning sign" by Windell Oskay
https://www.flickr.com/photos/oskay/237442629

But there were problems. They started, as weaknesses often do, at the core of Plone's strength, this support for working through-the-web.

The issue is that when you customized Plone through the web, your changes were stored in the database. They couldn't be tested or versioned properly. It was difficult to document them. And though site managers always mean to move the changes to filesystem code, real life tended to get in the way. It rarely actually happened.

"Not Me" by Caleb Roenigk, CC-BY
https://www.flickr.com/photos/crdot/6134455594

If making untested, unversioned, and undocumented changes causes problems, we should discourage or even prevent that behavior. As Plone moved through version 2 toward version 3, Plonistas were given new marching orders: All changes should be made on the filesystem.

Around this same time, tremendous advances were underway in the world of Python packaging, the creation of the Python Package Index and setuptools enabled more flexible ways to distribute and install software. Zope and Plone moved firmly to using eggs for core and plugin packages.

"Today is a good day" by Paul Downey, CC-BY
https://www.flickr.com/photos/psd/3545657028

The benefits of these changes were clear. Repeatable builds, reduced surprises, and modifications that were tested, versioned, and documented.

"Portaloo -- It's a trap!" by Spider.Dog CC-BY-2.0
https://www.flickr.com/photos/spiderdog/5934335847

However, while it was still easy to *use* Plone, to extend the system you *had* to be a programmer.

"Dagwood" by Cindy Funk CC-BY-2.0
https://www.flickr.com/photos/cindyfunk/3396982815

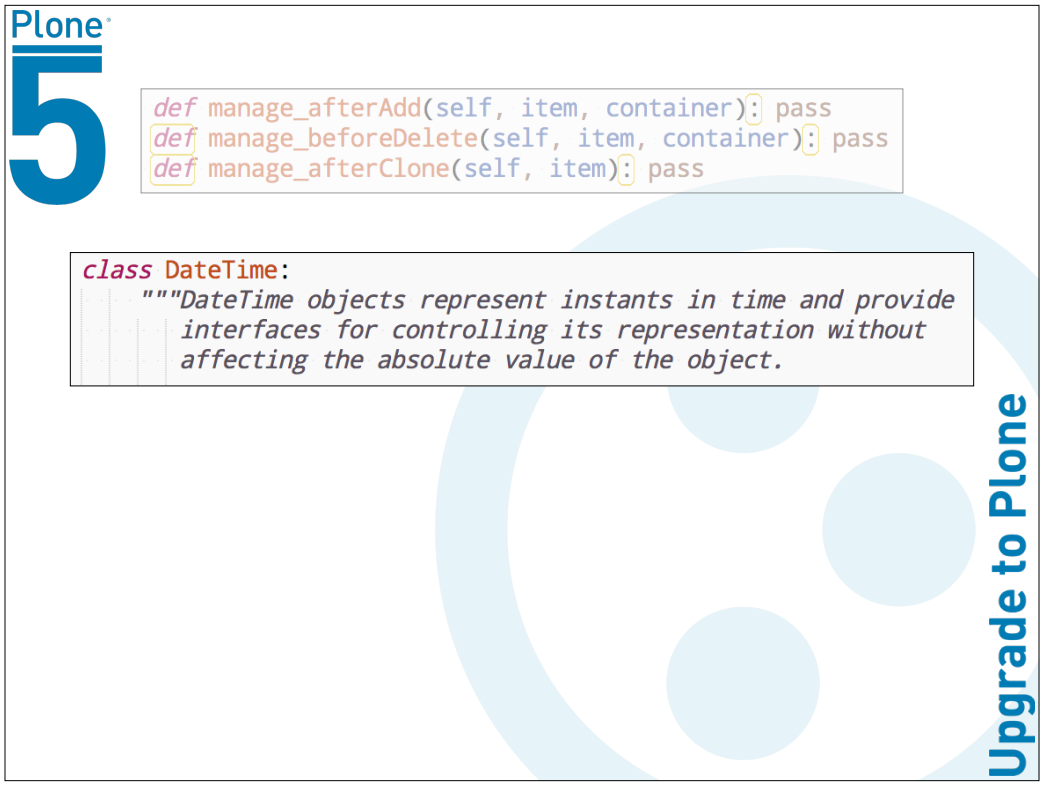
To make matters worse, Plone was built on top of the CMF, which was built on top of Zope.

As one of the earliest large systems built in Python, Zope was not particularly Pythonic.

* It had quirky method names and code style (pep8 was added five years after Zope started).
* It had oddly redundant classes (the datetime module arrived seven years after Zope).
* And perhaps most problematic, it made heavy use of Multiple Inheritance and mixins to share functionality.

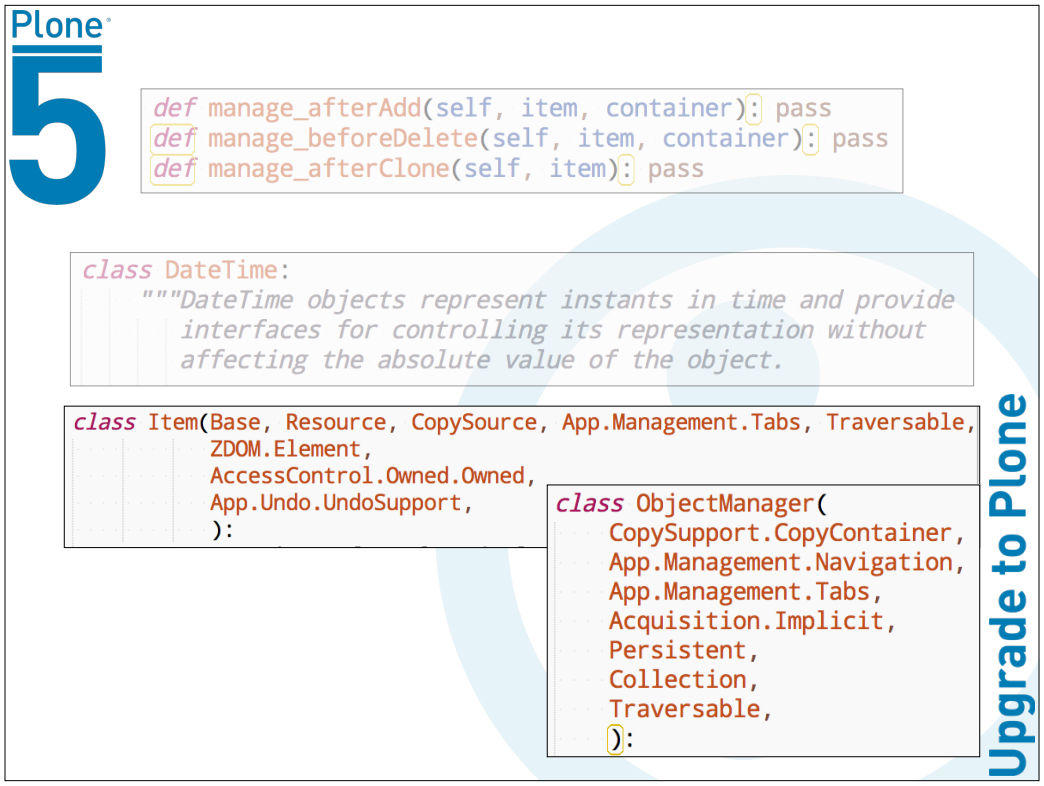
As one of the earliest large systems built in Python, Zope was not particularly Pythonic.

* It had quirky method names and code style (pep8 was added five years after Zope started).
* It had oddly redundant classes (the datetime module arrived seven years after Zope).
* And perhaps most problematic, it made heavy use of Multiple Inheritance and mixins to share functionality.



As one of the earliest large systems built in Python, Zope was not particularly Pythonic.

* It had quirky method names and code style (pep8 was added five years after Zope started).
* It had oddly redundant classes (the datetime module arrived seven years after Zope).
* And perhaps most problematic, it made heavy use of Multiple Inheritance and mixins to share functionality.

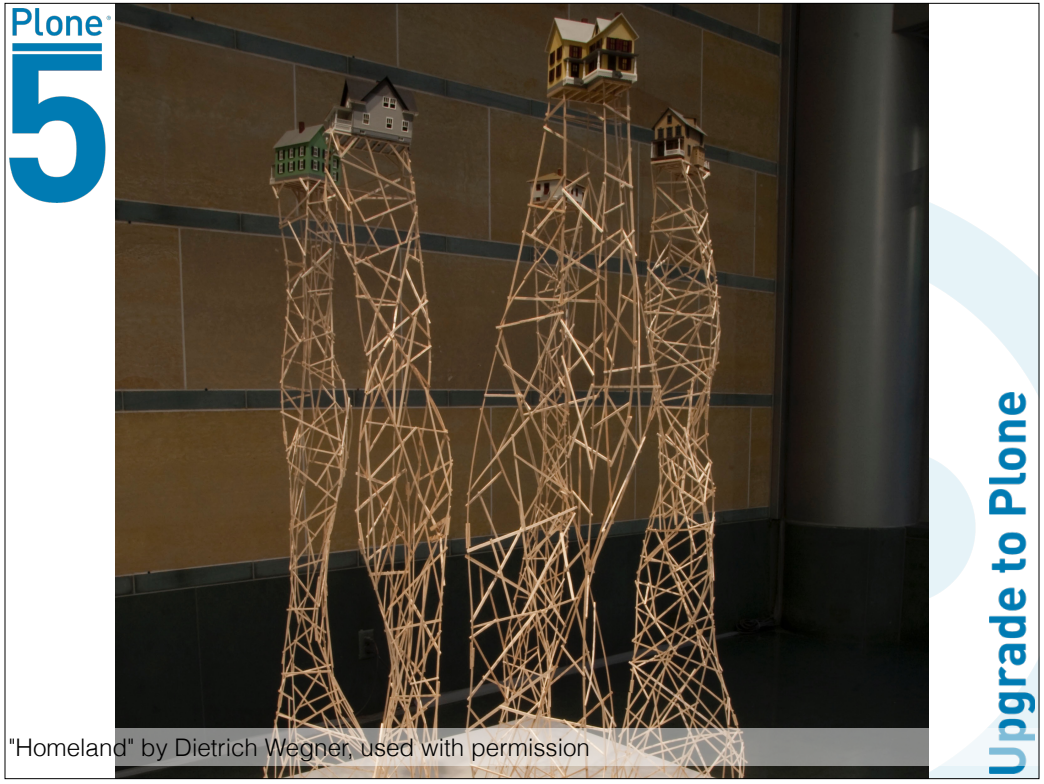


As one of the earliest large systems built in Python, Zope was not particularly Pythonic.

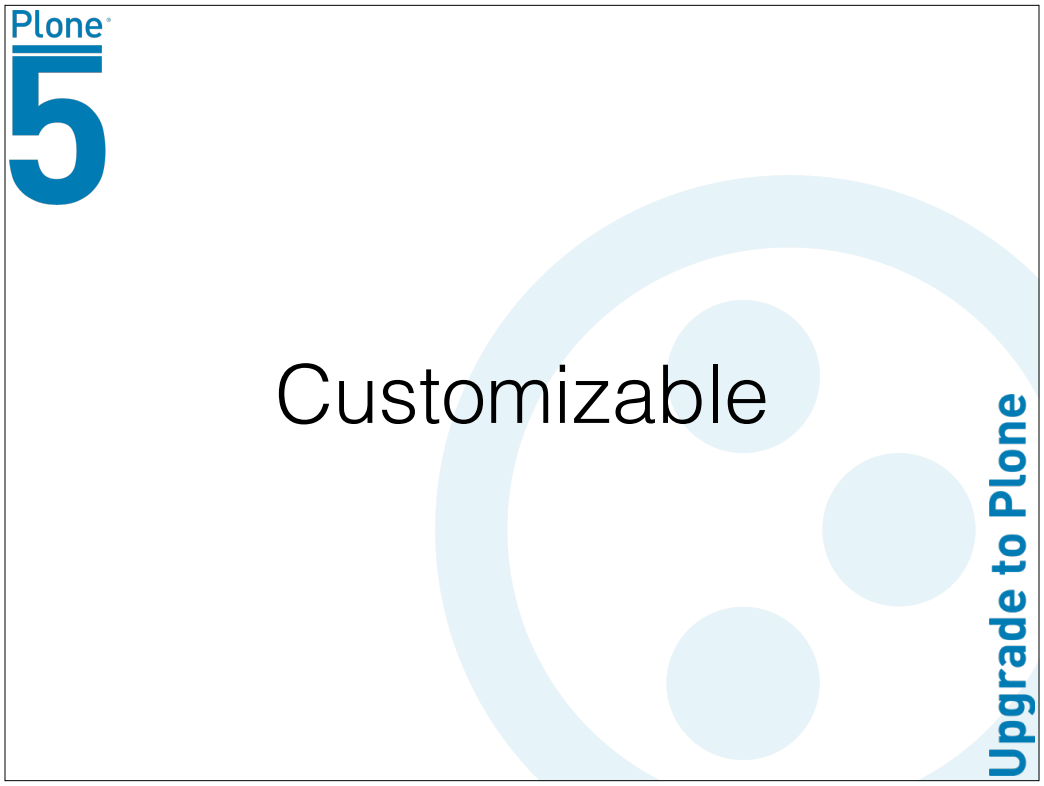
* It had quirky method names and code style (pep8 was added five years after Zope started).
* It had oddly redundant classes (the datetime module arrived seven years after Zope).
* And perhaps most problematic, it made heavy use of Multiple Inheritance and mixins to share functionality.

"Homeland" by Dietrich Wegner, used with permission

The end result meant that to have a hope of being able to customize Plone, you needed to be more than just a Python programmer.



You had to be a Plone programmer.

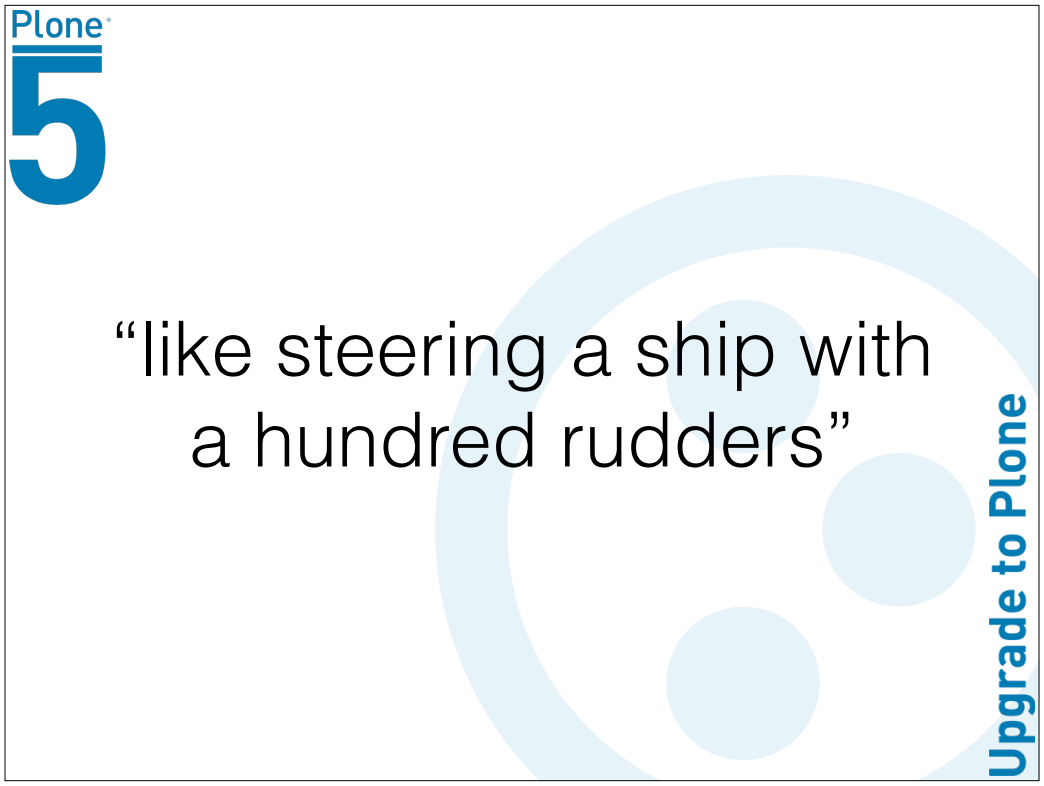


Plone was no longer quite so easily customizable.

A change was needed.

Plone was no longer quite so easily customizable.

A change was needed.



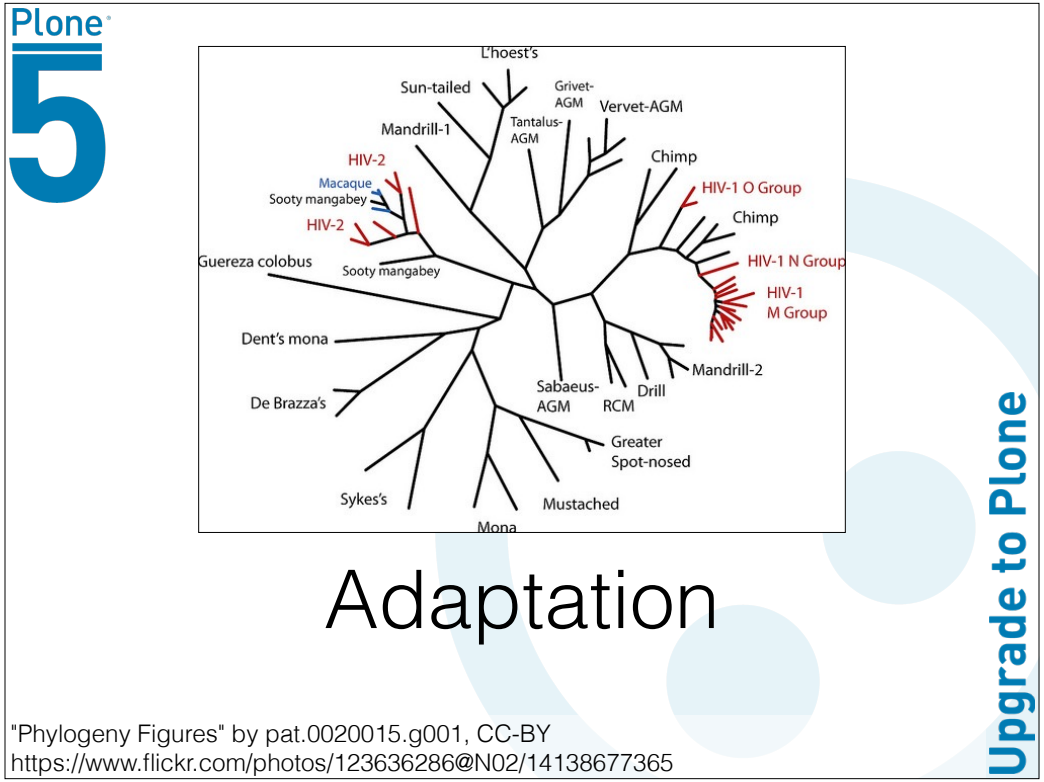
But with hundreds of core contributors, directed change can be difficult.

PAUSE

"Yin & Yang martini" by Andrew Magill CC-BY-2.0
https://www.flickr.com/photos/amagill/197789505

Weakness can rise from apparent strength, …. and strength can also rise from apparent weakness.

Plone’s large and diverse community has tried out a number of different approaches to solving the problems it faces.



Winners have risen out of this evolutionary process.

Plone 5, released almost 14 years to the day after the first public release of the system, brings those winning adaptations to you.

Plone 5

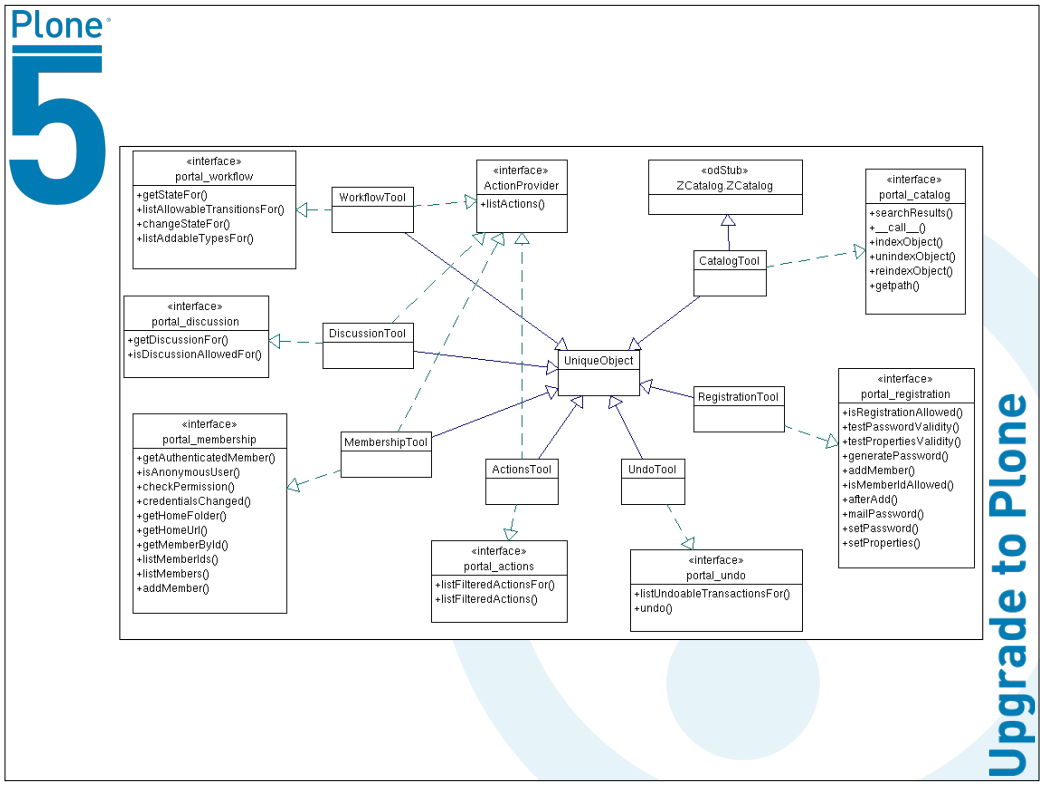
pseudo-adaptation

"Marlou Van Rhijn" by Fanny Schertzer (Own work) CC BY-SA 3.0
via Wikimedia Commons

Upgrade to Plone

But don’t just take my word for it, let’s take a look at some of the problems that Plone faced and the adaptations that have been made to solve them.

We’ll start with an adaptation that isn’t, exactly, an adaptation.



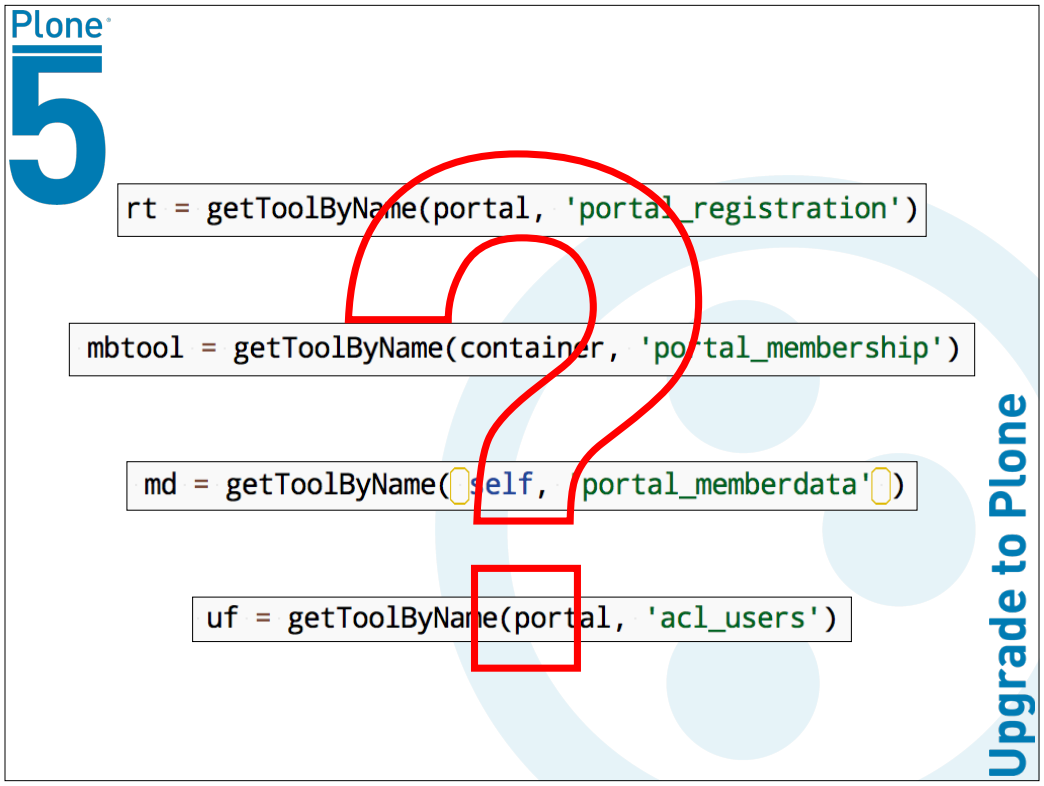
One of the highest barriers to entry in Plone development has been the messy API.

Plone provides access to core systems like registration, content creation and indexing, workflow and theming through “tool” objects. Developers of plugins could use the APIs of these tools to integrate their code with core functionality.

The problem is that just to get hold of these tools, you had to be in possession of arcane knowledge.

beyond that, the APIs of the tools were not always clear. For example, let's say you wanted to get some information about a Plone user. Perhaps you need their full name.

Which of the four tools with APIs related to users would *you* choose?



Which of the four tools with APIs related to users would *you* choose?

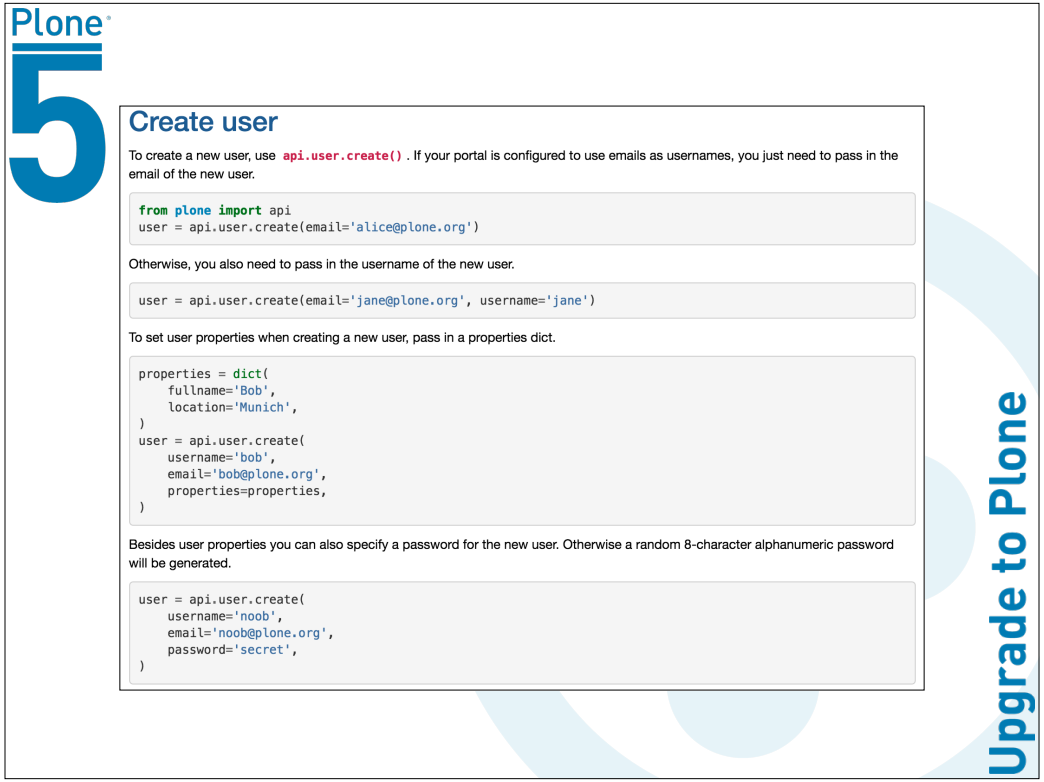
For years, little changed. Until a group of electrical engineering students got together for a Plone training in Eastern Europe.

Plone 5

Photo by hobvias sudoneighm - CC-BY
http://www.flickr.com/photos/striatic/2192192956/

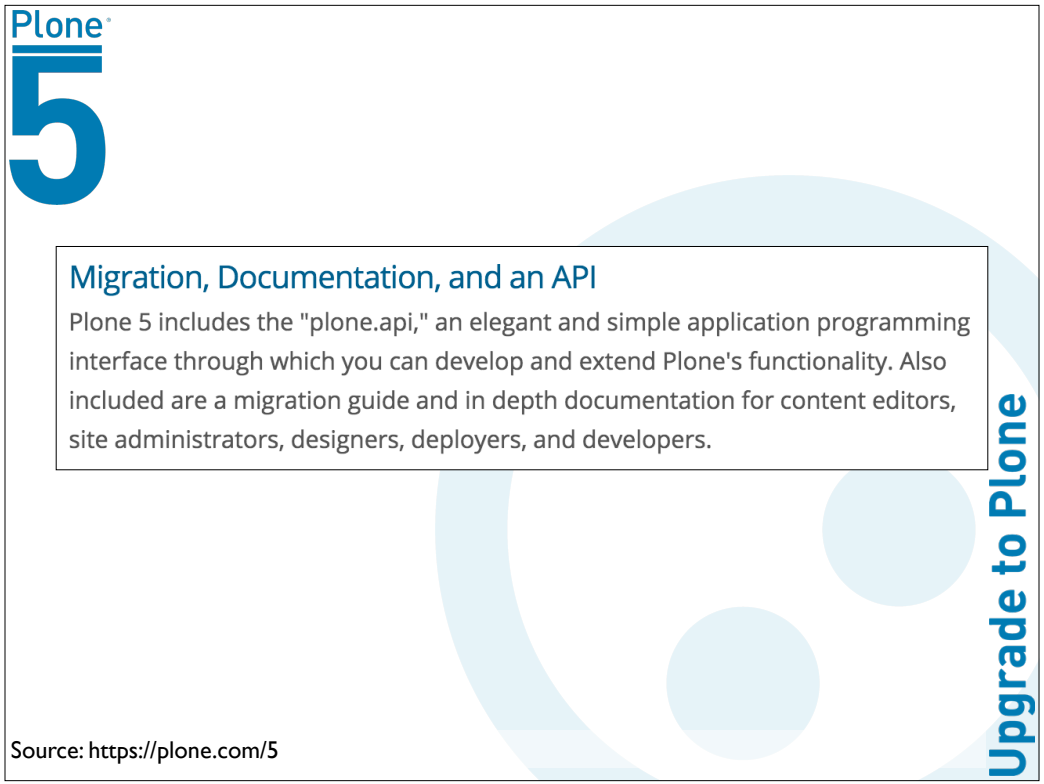
Upgrade to Plone

After a few hours of trying, and failing, to figure out how to accomplish simple tasks, most of them gave up.

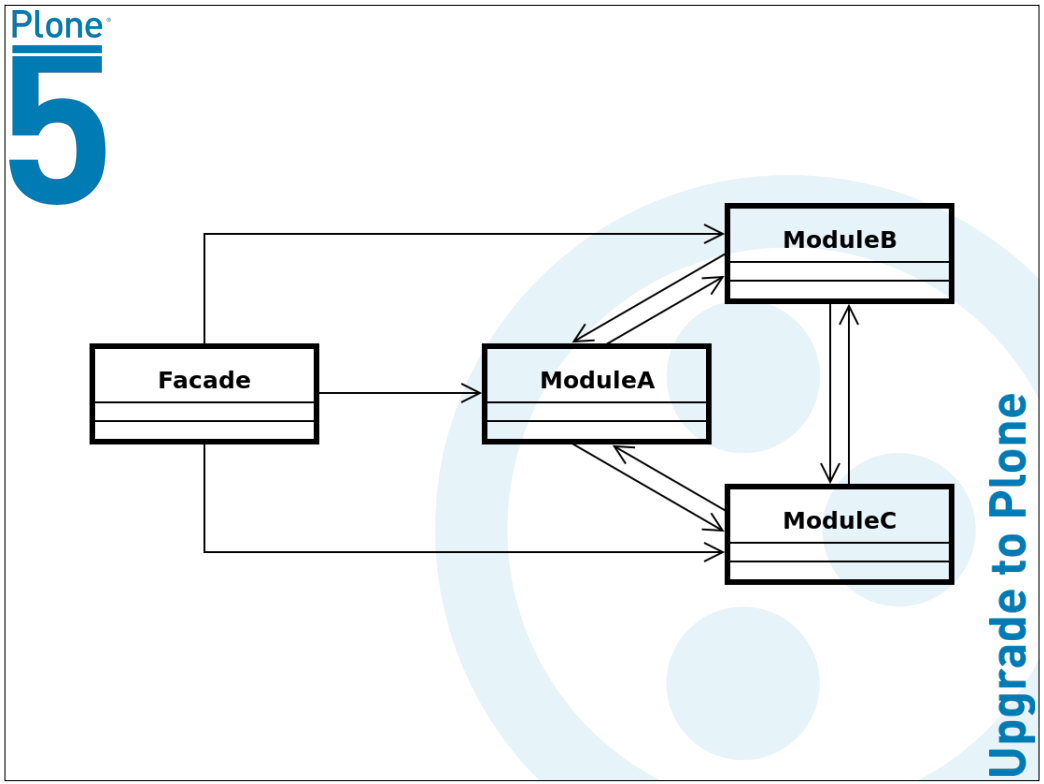


A few, however, switched gears, and began writing the documentation for the API they wish Plone had. They consciously chose not to write any code, instead focusing first on defining what they wanted to do.

Eventually, after a number of sprints to perfect the idea, the plone.api package was born, providing a clear, well documented facade for the most common actions needed by Plone developers.



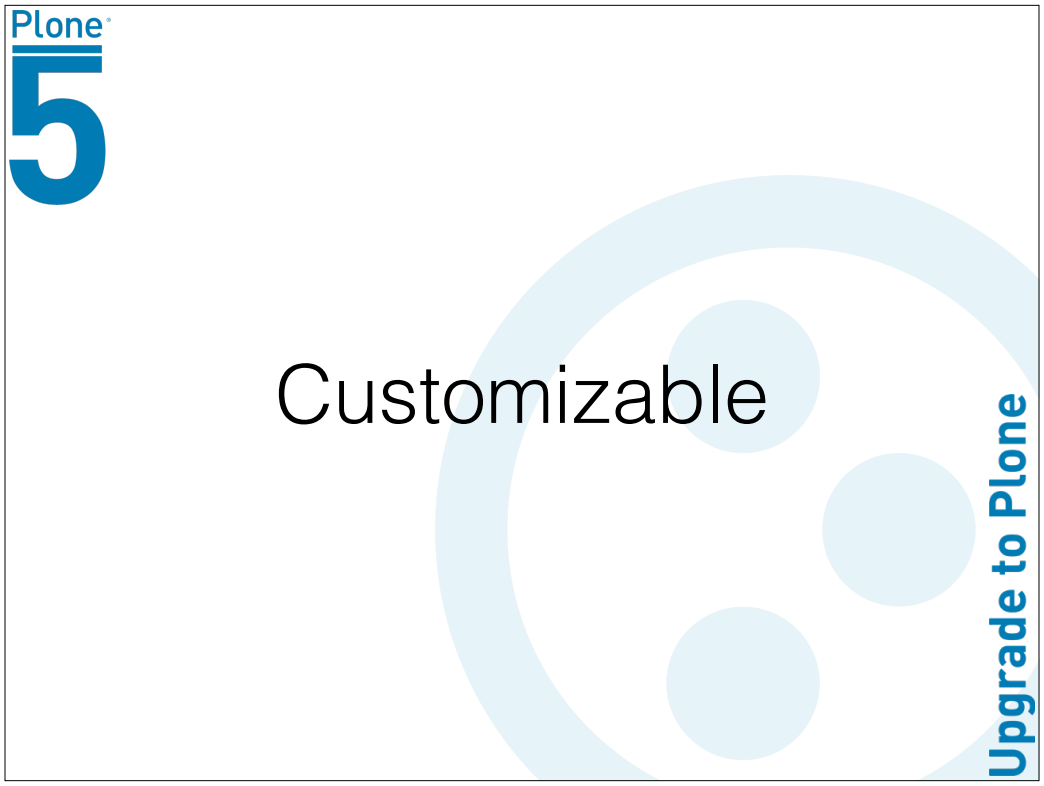
The new package spent several years as an add-on, growing stronger and adapting to the real-world requirements of developers. Now, in Plone 5, it’s become part of the core of Plone.



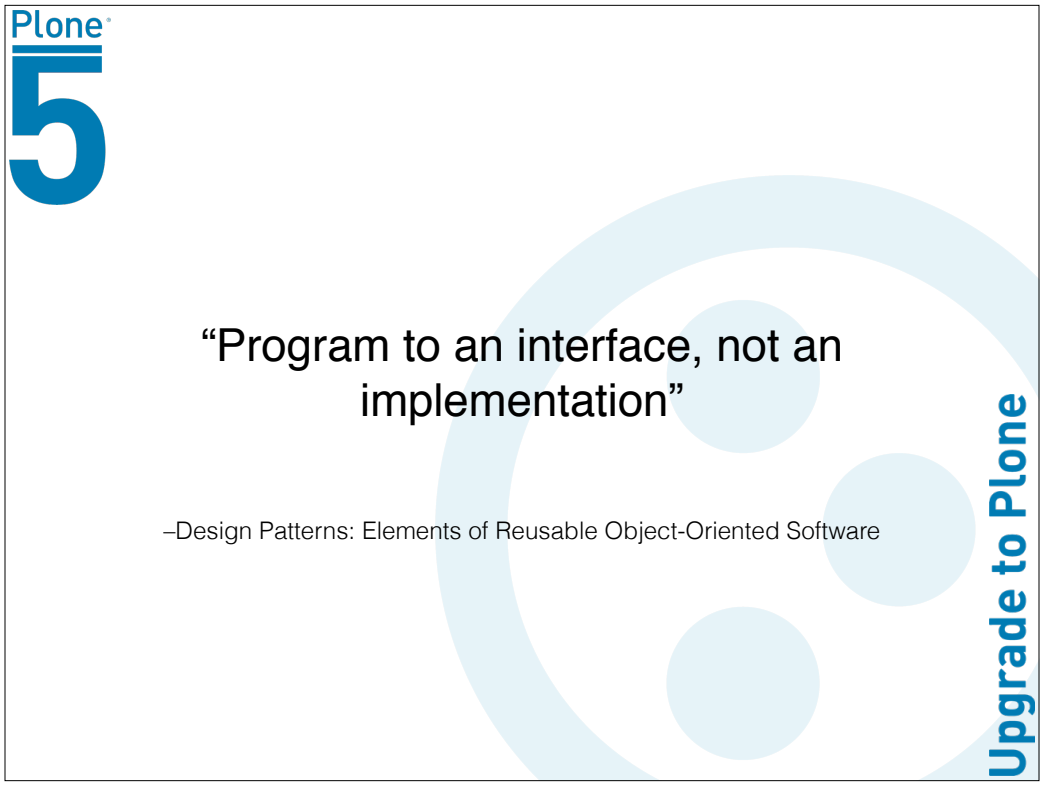
plone.api is a great example of the “facade pattern” in software design. It adapts the complicated APIs provided by the tools of Plone, the CMF and Zope into a clean, easy-to-use API. Developers can use this API without needing to worry about the mess behind the facade.

(And work can begin to clean up that mess without disturbing these plugins)

This facade has made it possible to write plugin code that is clean, modern and Pythonic.



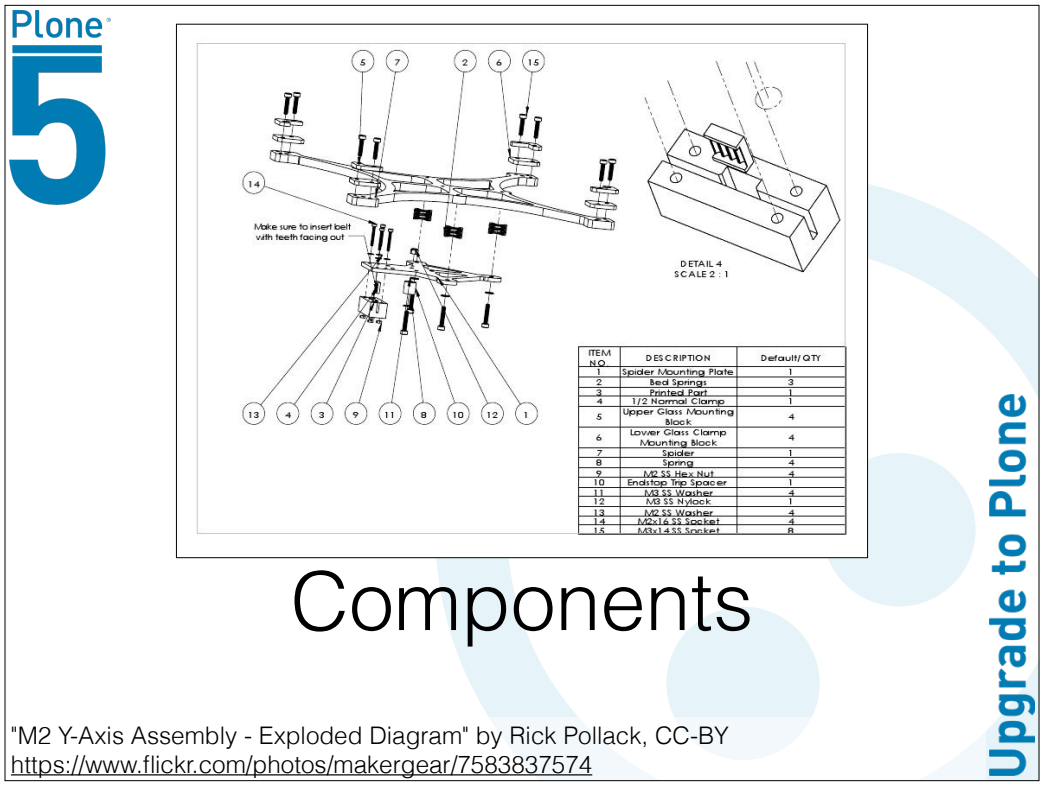
This facade has made it possible to write plugin code that is clean, modern and Pythonic.



The facade pattern is one example of a way of programming to an interface.

An interface is a kind of a contract that specifies the external behavior of an object. Any object that exhibits that behavior can be said to “provide” the interface.

the benefit of such an approach to software design is that programmers need not concern themselves about exactly “how” a particular part of the system works. They can create components that are more truly independent of each-other.



"M2 Y-Axis Assembly - Exploded Diagram" by Rick Pollack, CC-BY
https://www.flickr.com/photos/makergear/7583837574

In 2001, the folks at Zope Corporation were quite interested in this idea. They had learned about the problems of using Multiple Inheritance and mixin classes.

They started work on a new approach, a Component Architecture for Zope (we call it the ZCA). It consisted of a Python implementation of the idea of an interface and a registry called a “Component Manager”.

Programmers could create software components using a number of different patterns and register them with this component manager.

"Adapter Frenzy" by Andrew Turner CC-BY-2.0
https://www.flickr.com/photos/ajturner/3736209089

One of the design patterns implemented by the Zope Component Architecture is the Adapter pattern.

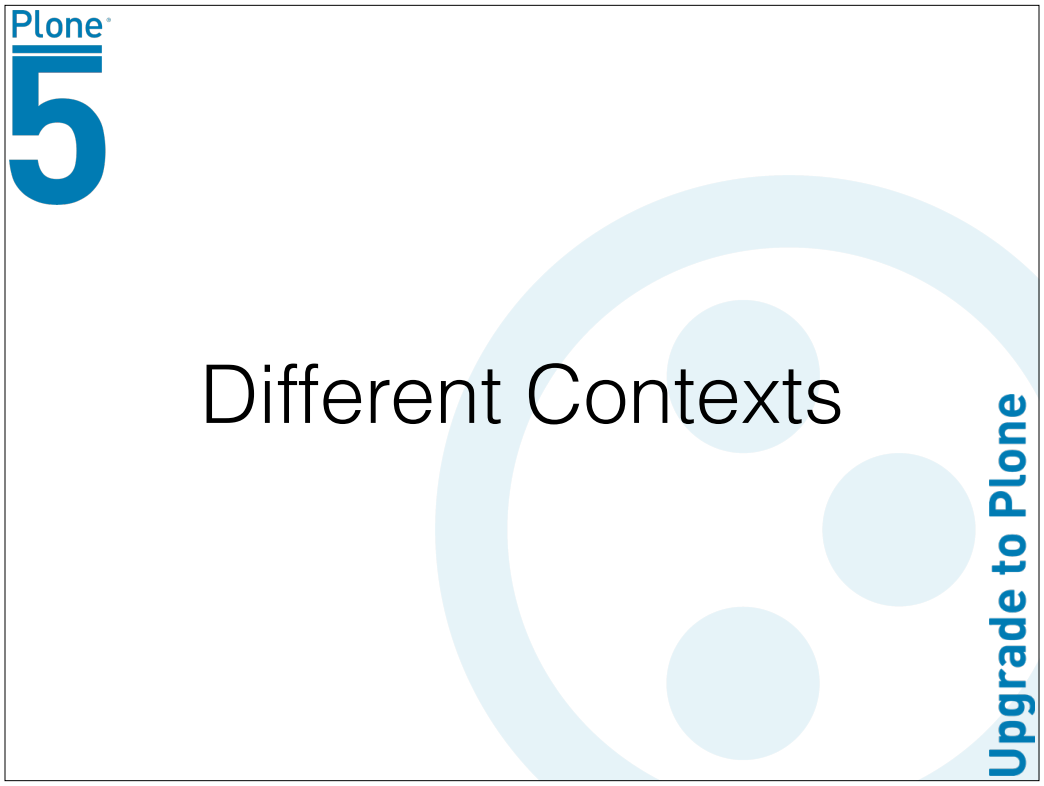
It’s easiest to describe the adapter pattern in real world terms, so let’s consider one example:

Plone 5

https://commons.wikimedia.org/wiki/File:Kleindle.jpg (CC-0)

Upgrade to Plone

Most buildings that have running water have a hot and cold water feed. Two pipes, one provides hot water, the other provides cold. You can think of this as the “interface” of a water supply.



You can use this water supply in a variety of ways:

* To wash your dirty dishes
* To rinse off after a dip in the local pool
* Or to do your laundry (though you might want to shoo the cat out before you begin).

The sink faucet, the showers, the cat hideout are all “adapters” of the interface provided by the water supply. The make use of the hot and cold water in a variety of ways.

If we wanted to create a code representation of this using the ZCA, it might go something like this:

You can use this water supply in a variety of ways:

* To wash your dirty dishes
* To rinse off after a dip in the local pool
* Or to do your laundry (though you might want to shoo the cat out before you begin).

The sink faucet, the showers, the cat hideout are all “adapters” of the interface provided by the water supply. The make use of the hot and cold water in a variety of ways.

If we wanted to create a code representation of this using the ZCA, it might go something like this:

You can use this water supply in a variety of ways:

* To wash your dirty dishes
* To rinse off after a dip in the local pool
* Or to do your laundry (though you might want to shoo the cat out before you begin).

The sink faucet, the showers, the cat hideout are all “adapters” of the interface provided by the water supply. The make use of the hot and cold water in a variety of ways.

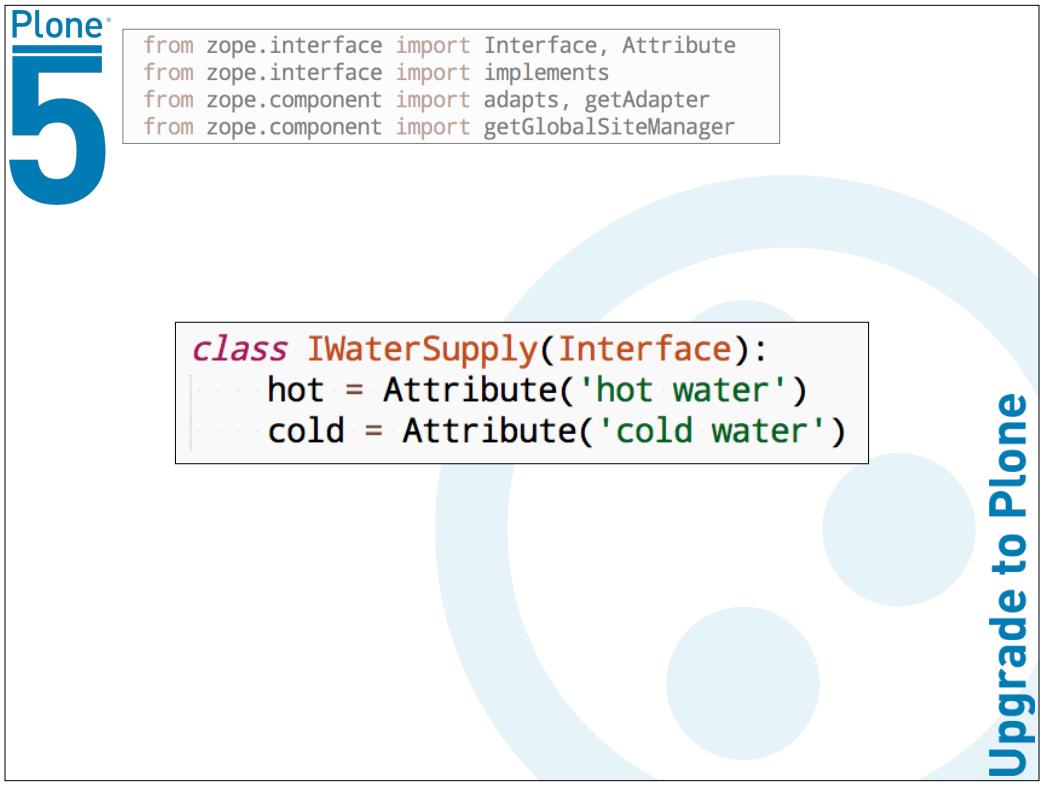
If we wanted to create a code representation of this using the ZCA, it might go something like this:

You can use this water supply in a variety of ways:

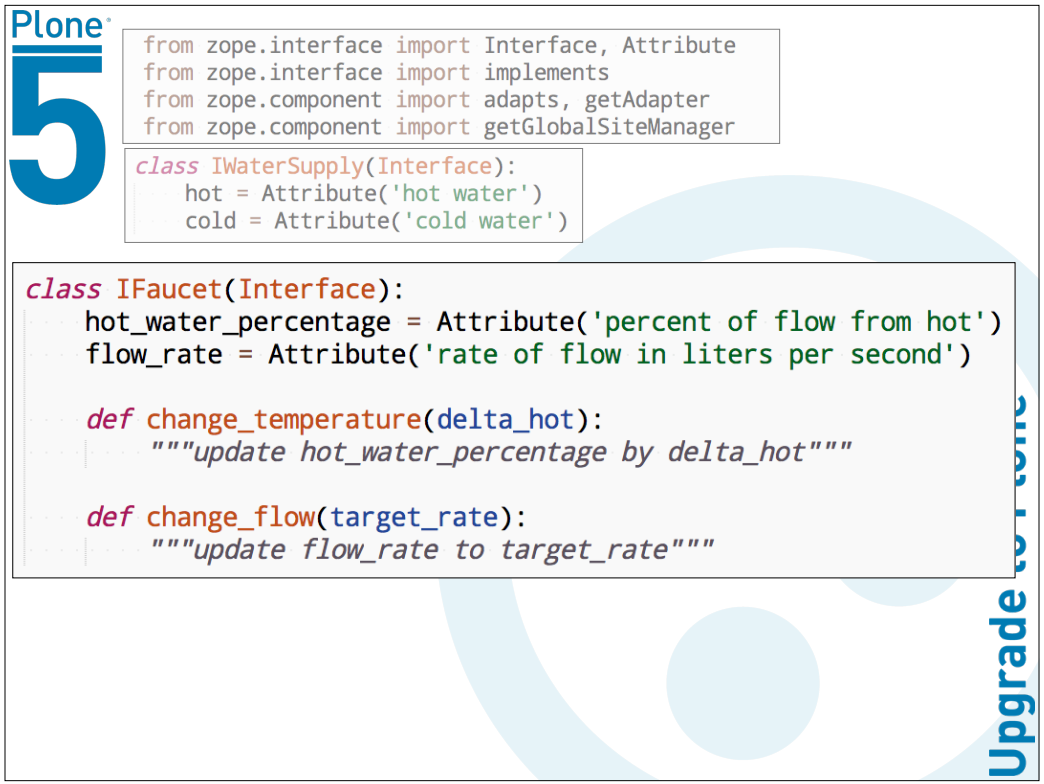
* To wash your dirty dishes
* To rinse off after a dip in the local pool
* Or to do your laundry (though you might want to shoo the cat out before you begin).

The sink faucet, the showers, the cat hideout are all “adapters” of the interface provided by the water supply. The make use of the hot and cold water in a variety of ways.

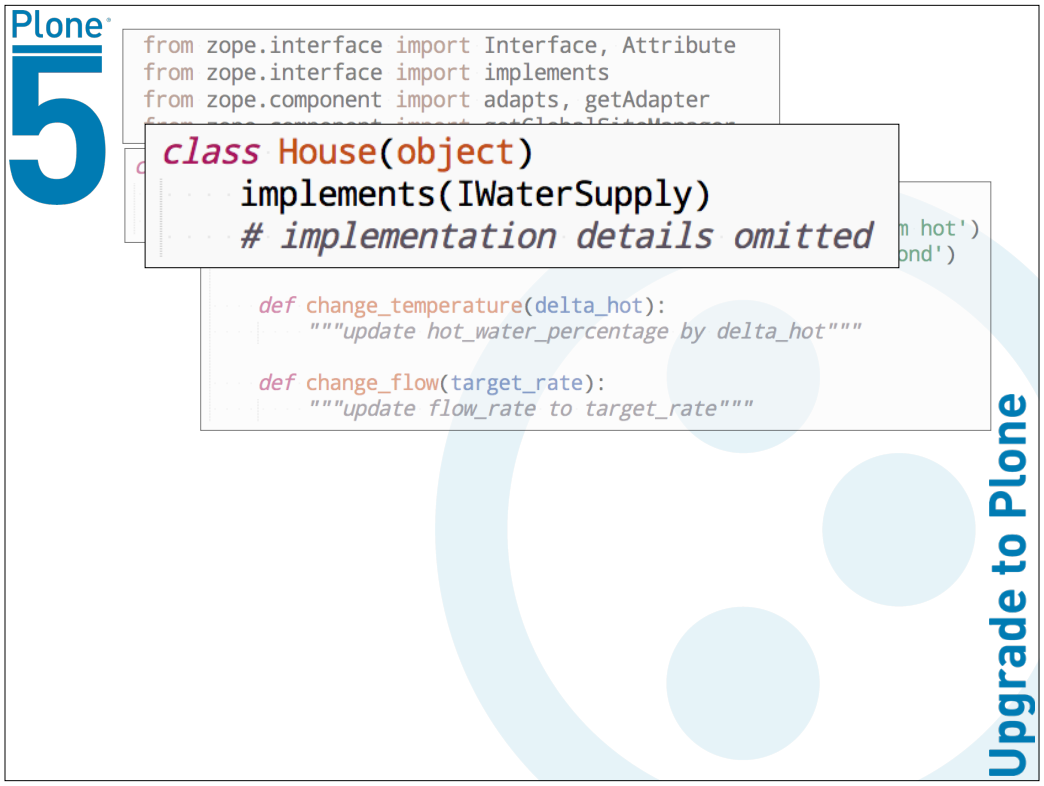
If we wanted to create a code representation of this using the ZCA, it might go something like this:



We can use the core packages of the ZCA to set up interfaces for our water supply. This represents the thing we will adapt. Notice that we don't have any implementation details, just names for attributes and descriptions of them for documentation purposes.

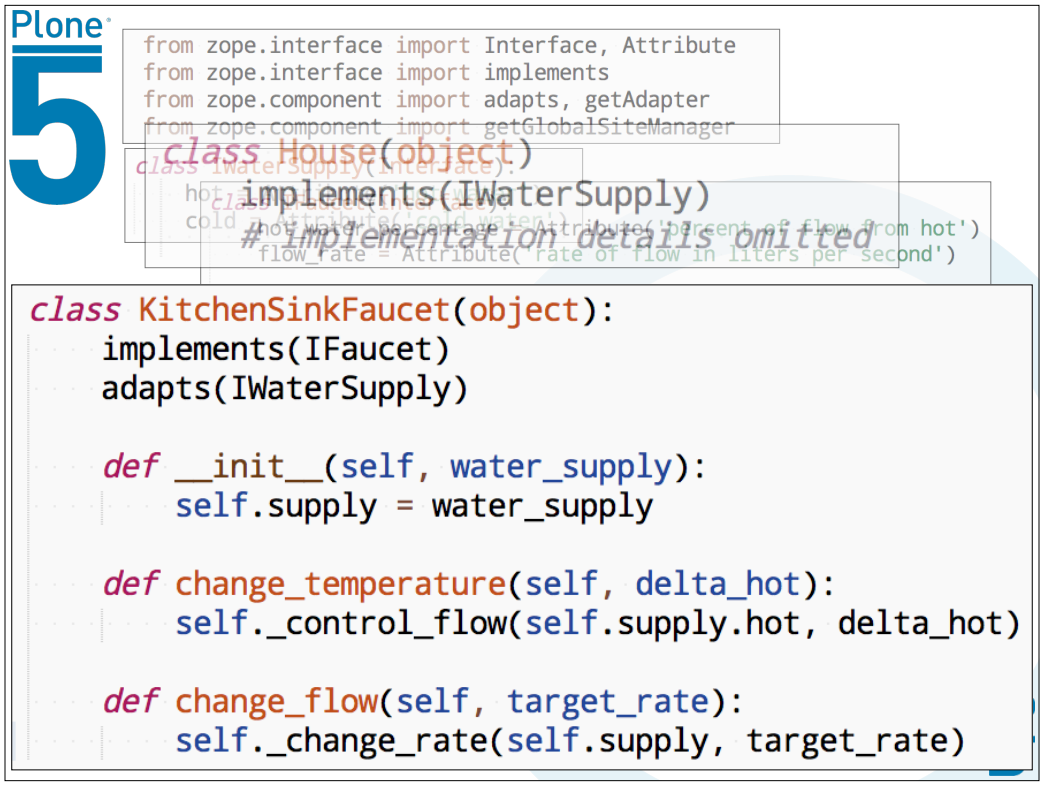


The next step is to create interfaces for our adapters. We'll use use the faucet for our sink. Again, notice that we specify methods and attributes but provide no implementation.



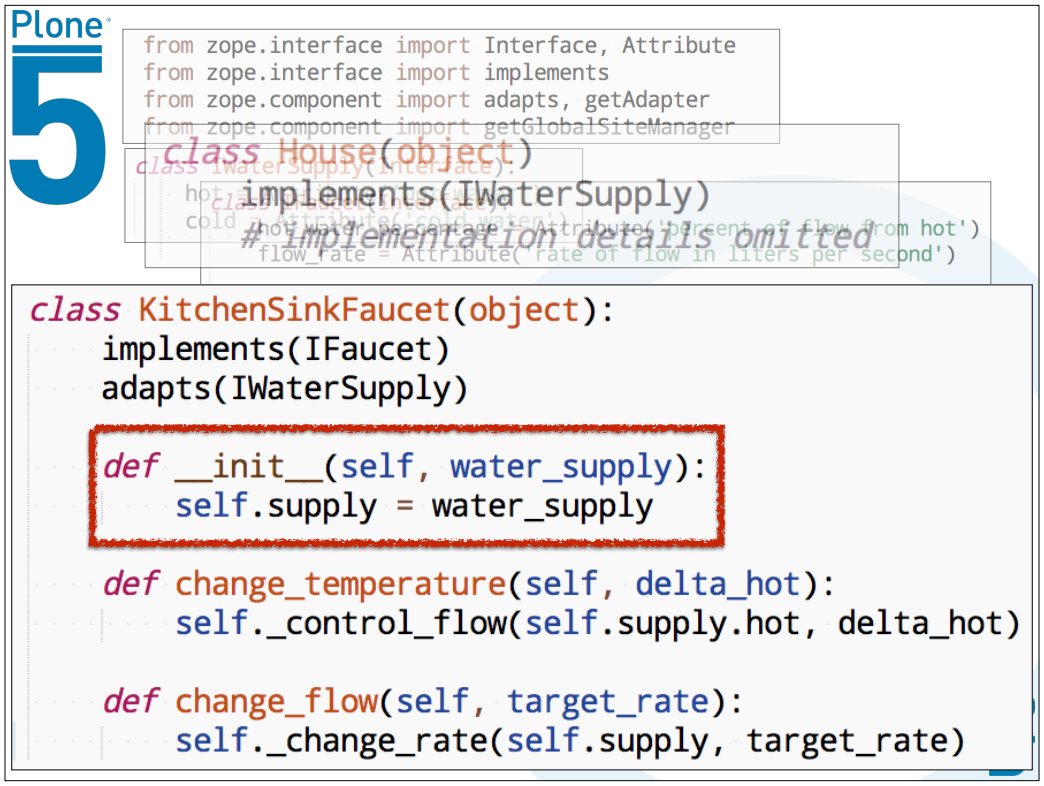
Now we can start to implement classes. We'll set up a house that implements our water supply interface. And we can create a kitchen sink faucet so we can finally get to those dirty dishes.

There's one important thing to note about our sink faucet. Take a look at the init special method. Notice that it takes a water supply as an argument and then sets that supply as an instance attribute. This is an example of the OOP approach called "containment" or "composition"



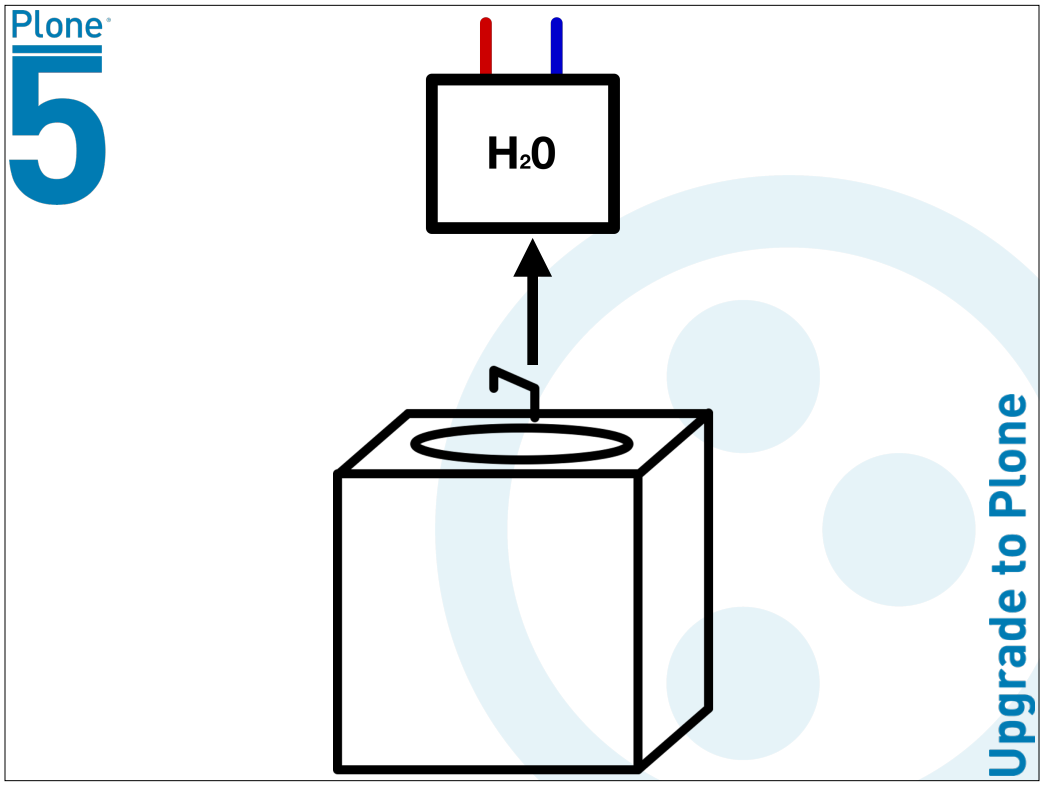
Now we can start to implement classes. We'll set up a house that implements our water supply interface. And we can create a kitchen sink faucet so we can finally get to those dirty dishes.

There's one important thing to note about our sink faucet. Take a look at the init special method. Notice that it takes a water supply as an argument and then sets that supply as an instance attribute. This is an example of the OOP approach called "containment" or "composition"

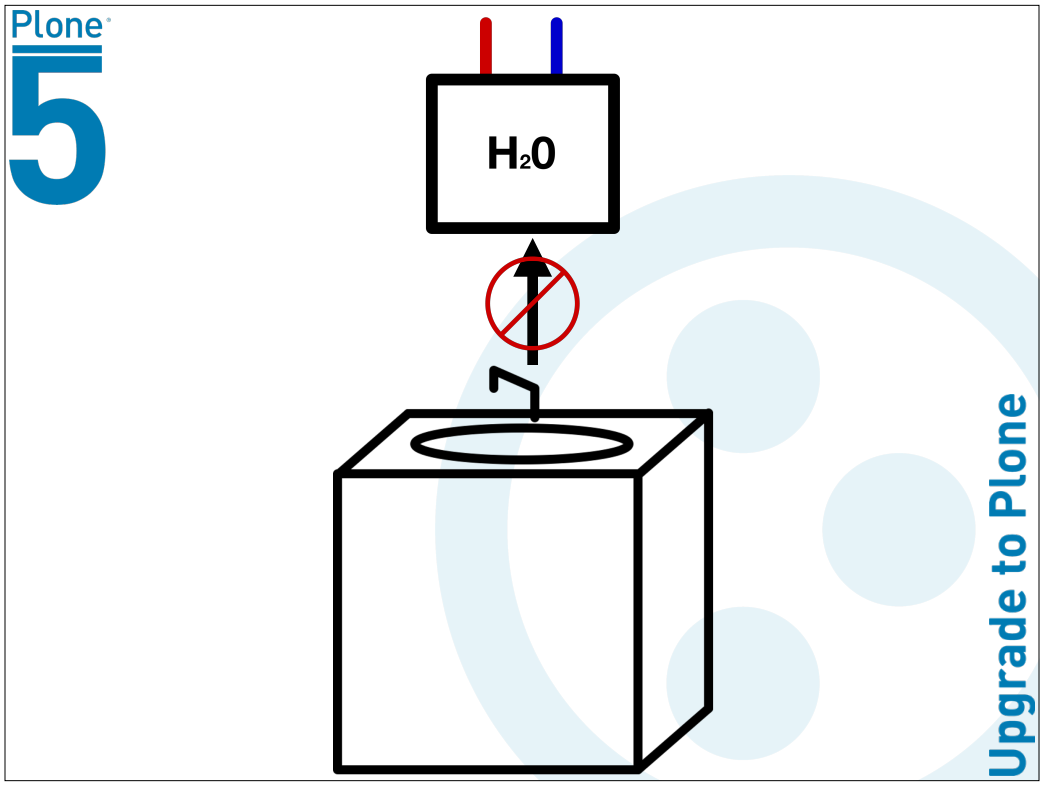


Now we can start to implement classes. We'll set up a house that implements our water supply interface. And we can create a kitchen sink faucet so we can finally get to those dirty dishes.

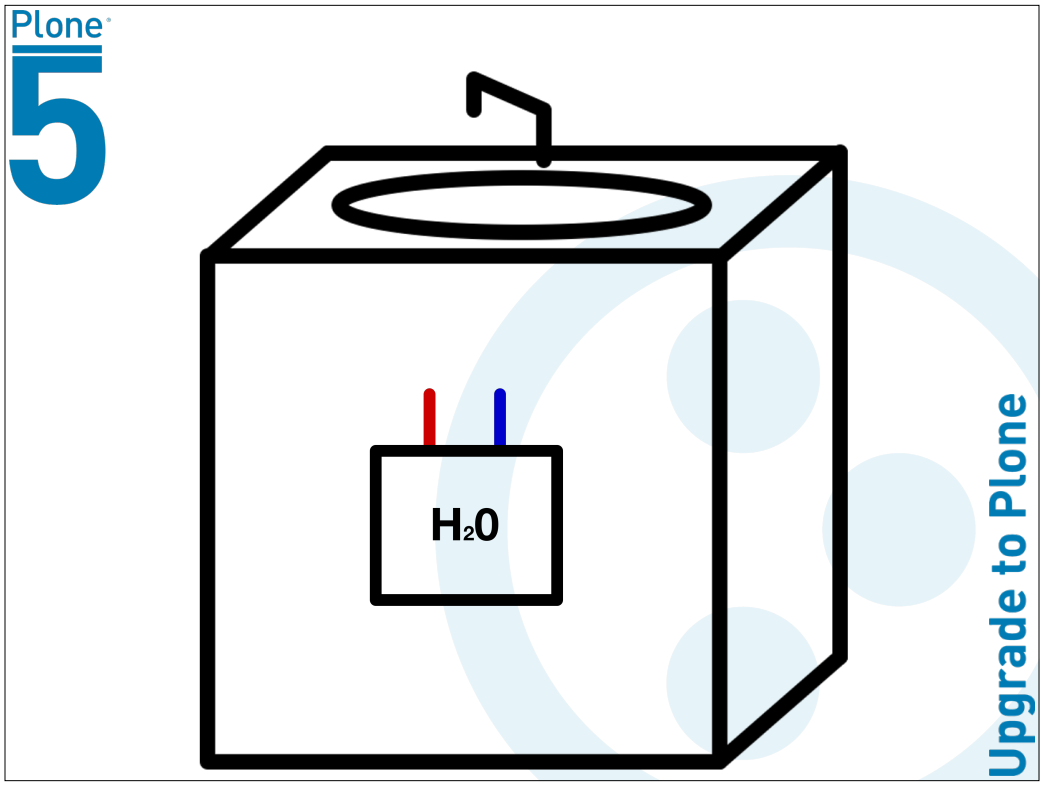
There's one important thing to note about our sink faucet. Take a look at the init special method. Notice that it takes a water supply as an argument and then sets that supply as an instance attribute. This is an example of the OOP approach called "containment" or "composition"



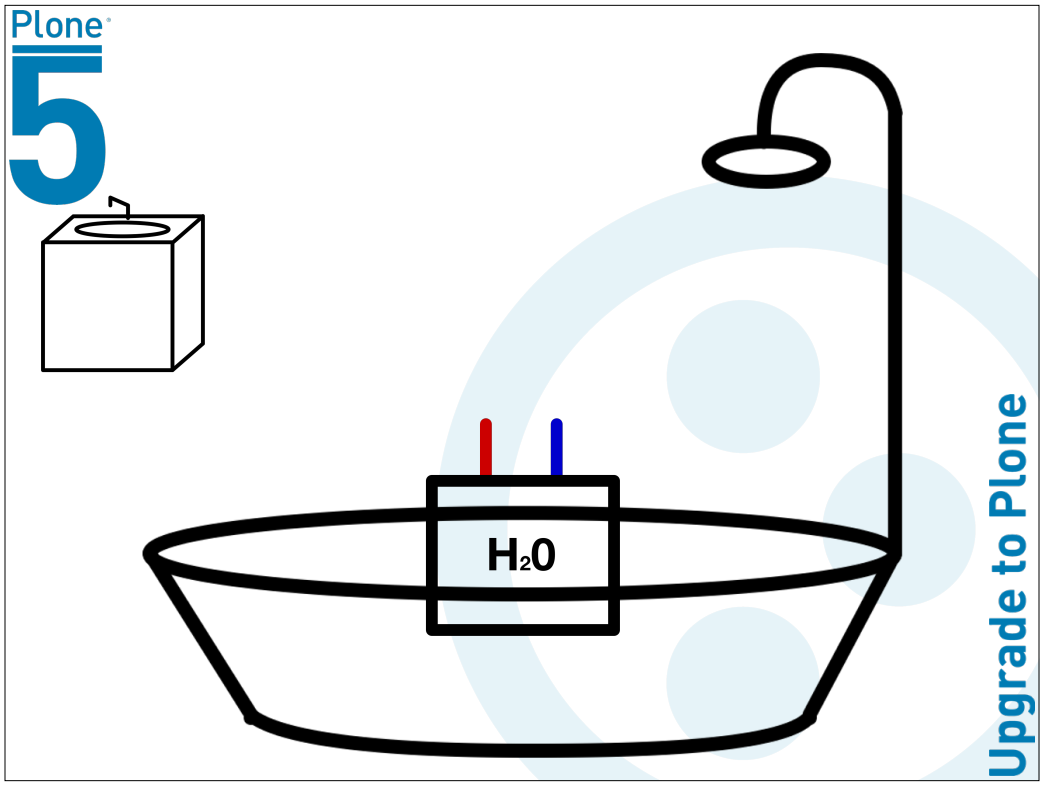
Often we reach for containment when we are dealing with a situation of “has-a” rather than “is-a”. A kitchen sink is not itself a special kind of water supply, but it does have one in it.



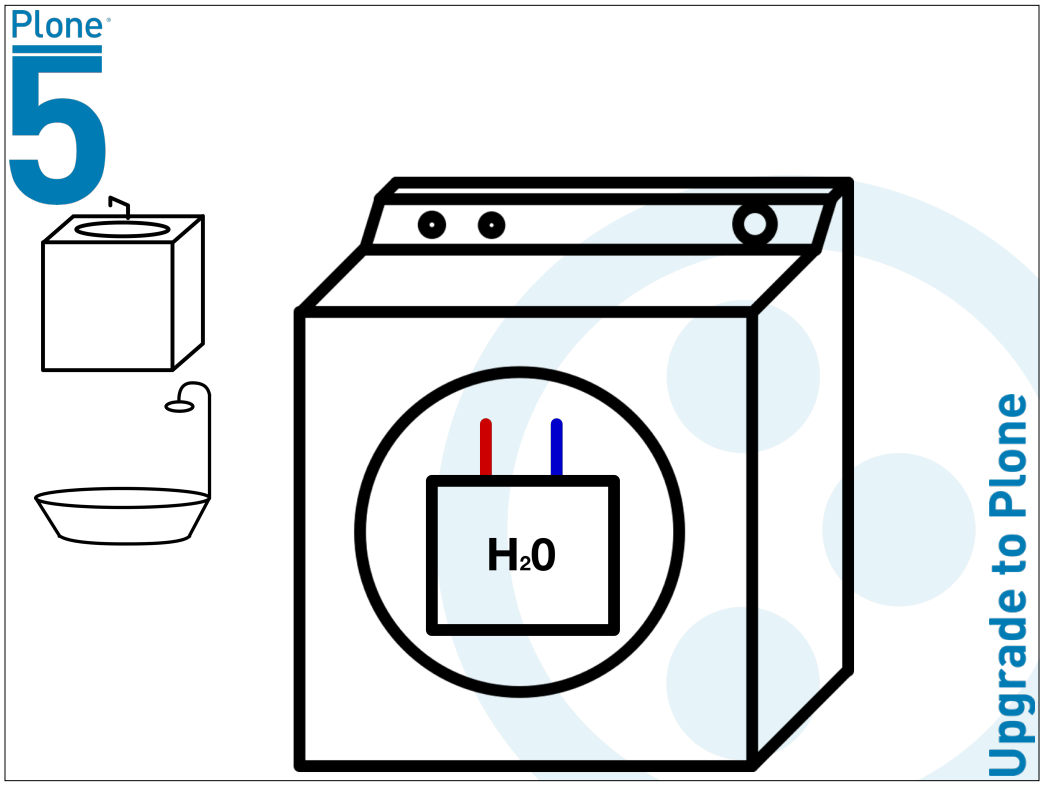
Often we reach for containment when we are dealing with a situation of “has-a” rather than “is-a”. A kitchen sink is not itself a special kind of water supply, but it does have one in it.



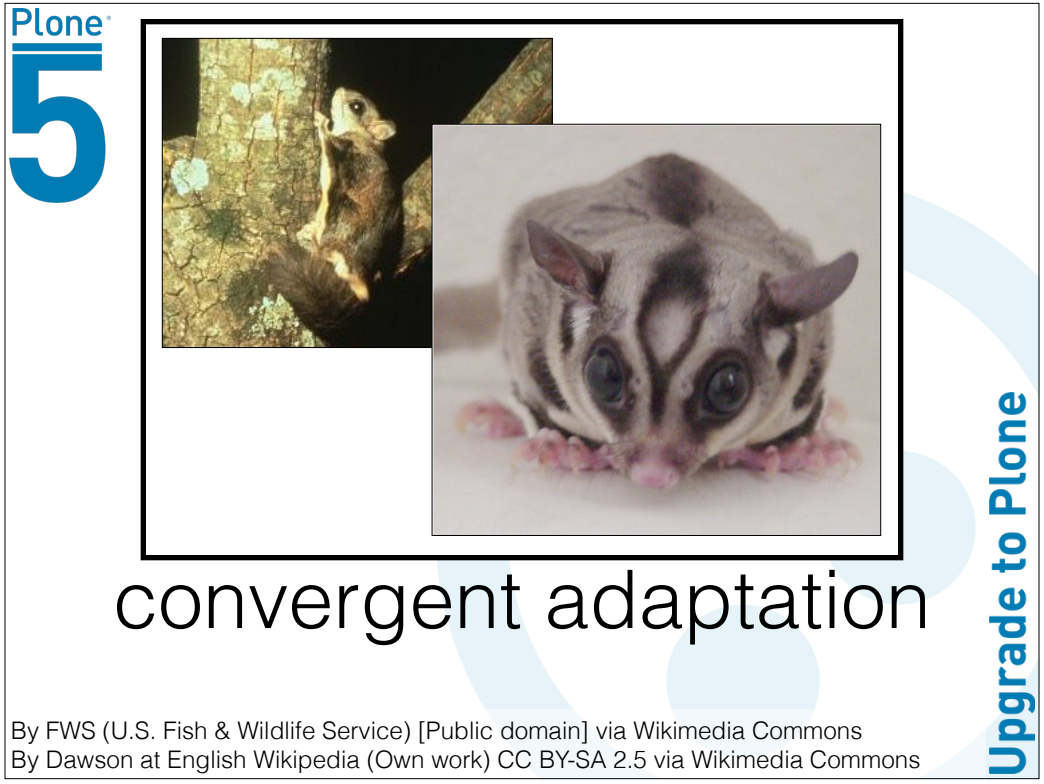
Often we reach for containment when we are dealing with a situation of “has-a” rather than “is-a”. A kitchen sink is not itself a special kind of water supply, but it does have one in it.



At its heart, the adapter pattern is about containment. The adapted object is contained in the adapter, and so any of its attributes and methods can be used to fulfill the interface of the adapter, our sink, the shower, or the washing machine.



At its heart, the adapter pattern is about containment. The adapted object is contained in the adapter, and so any of its attributes and methods can be used to fulfill the interface of the adapter, our sink, the shower, or the washing machine.



As it turns out, this design pattern is quite useful in a large, complex system like Plone. It's helped to solve a number of thorny problems we've faced in trying to deliver simple and reliable customizability.

One of the ways adapters have helped is in allowing us to treat a number of different objects as if they shared a certain attribute. For example, the ability to be published.

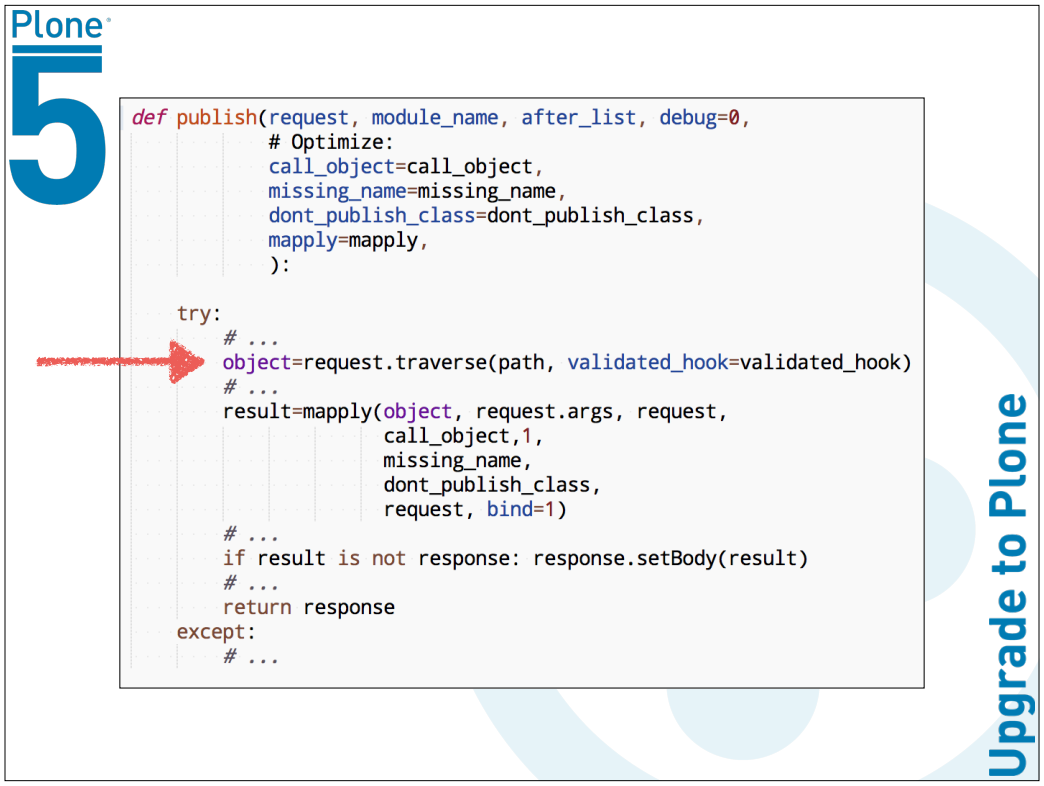
Remember that one of the key features of Zope is the idea of “object publication”: displaying an object directly as an HTTP response.

Remember that one of the key features of Zope is the idea of “object publication”: displaying an object directly as an HTTP response.

If you remember, Zope uses traversal to find the object we want to see.

The object that is found is supposed to publish itself, returning a representation that can be sent back as an HTTP response.

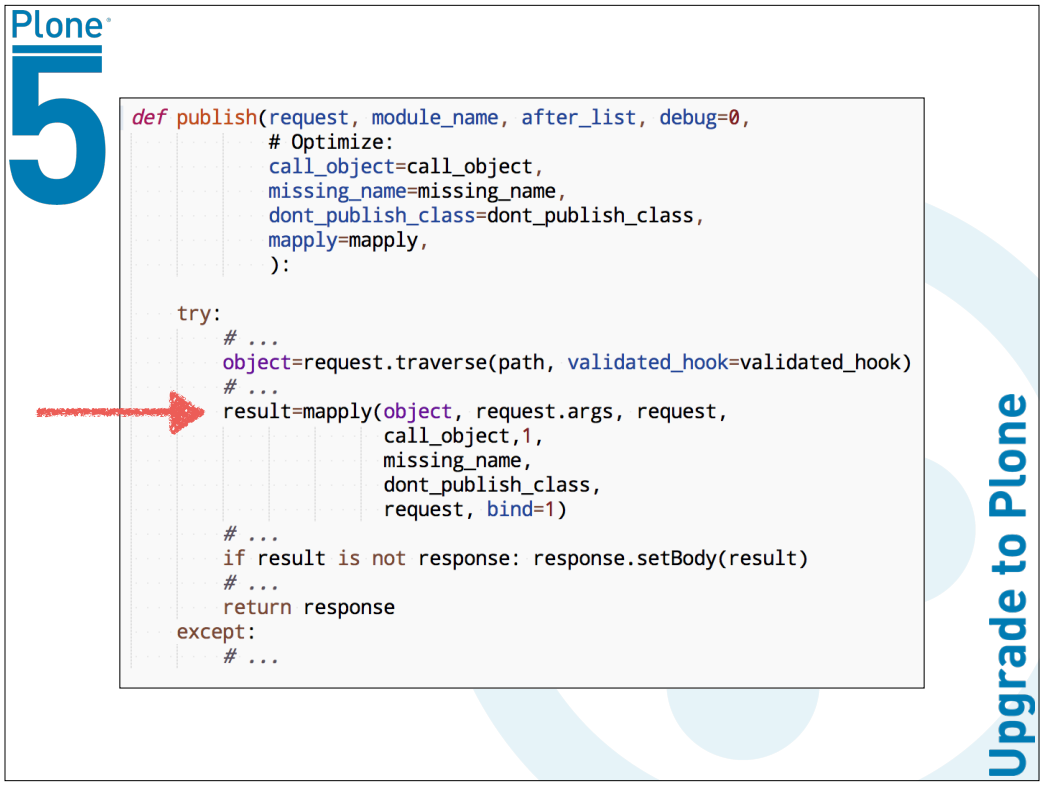
Well, what happens in there?



If you remember, Zope uses traversal to find the object we want to see.

The object that is found is supposed to publish itself, returning a representation that can be sent back as an HTTP response.

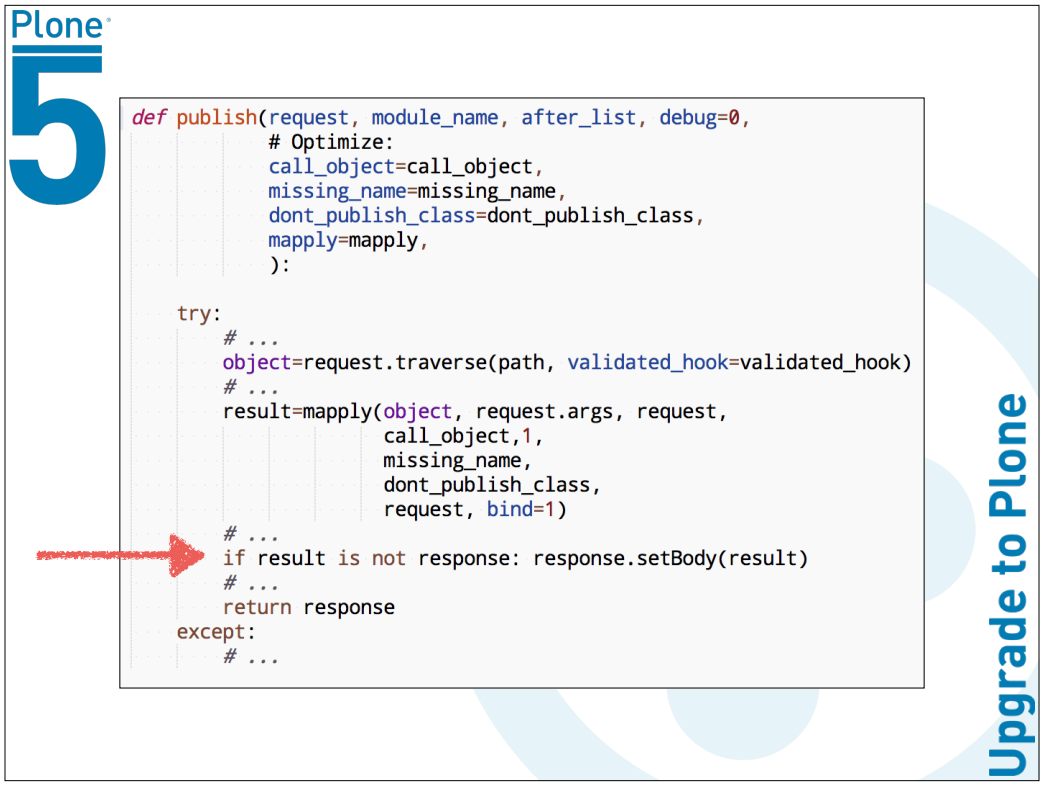
Well, what happens in there?



If you remember, Zope uses traversal to find the object we want to see.

The object that is found is supposed to publish itself, returning a representation that can be sent back as an HTTP response.

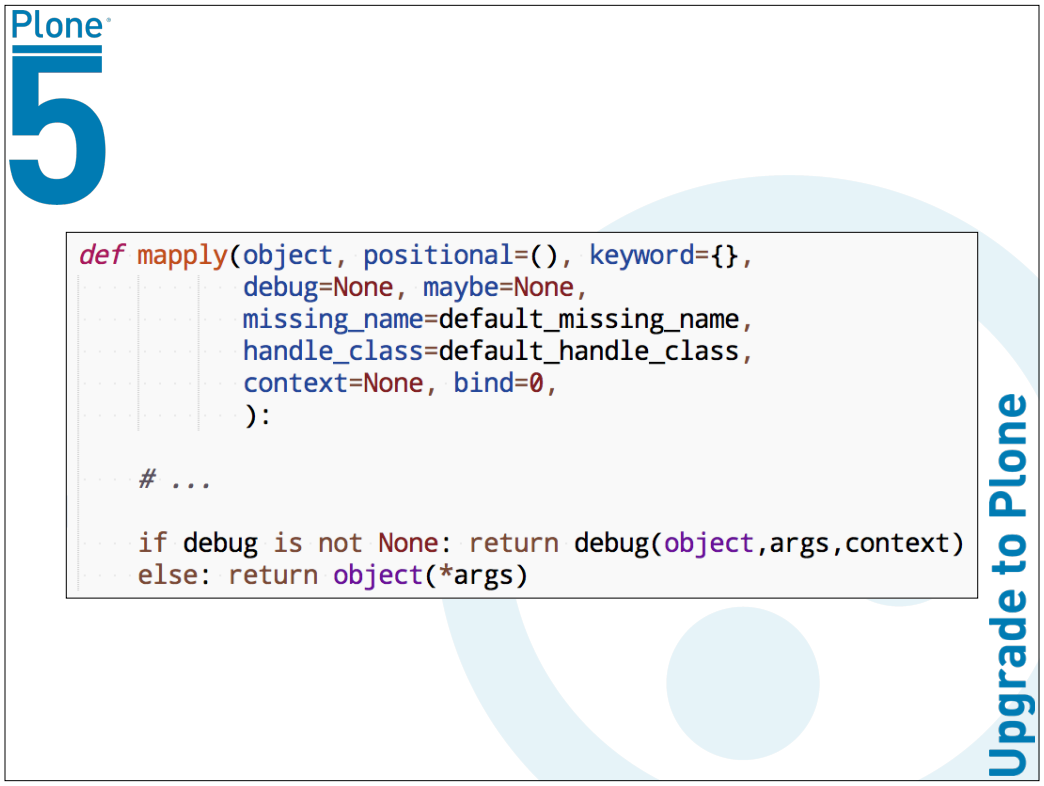
Well, what happens in there?



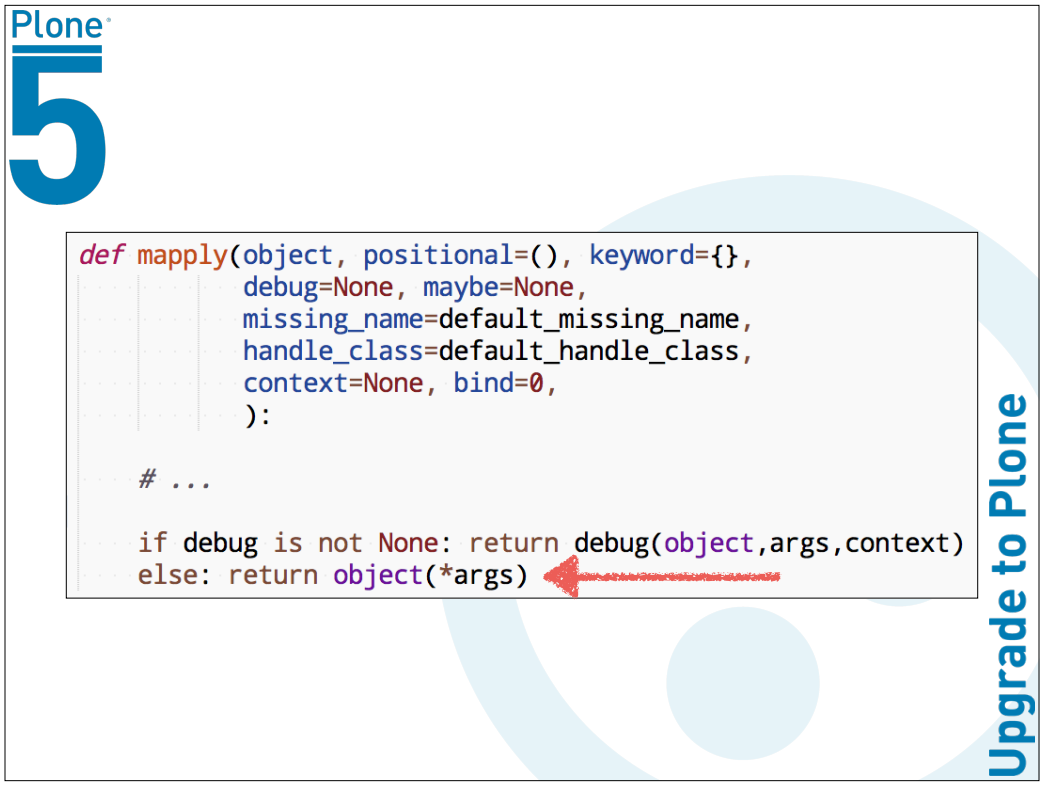
If you remember, Zope uses traversal to find the object we want to see.

The object that is found is supposed to publish itself, returning a representation that can be sent back as an HTTP response.

Well, what happens in there?

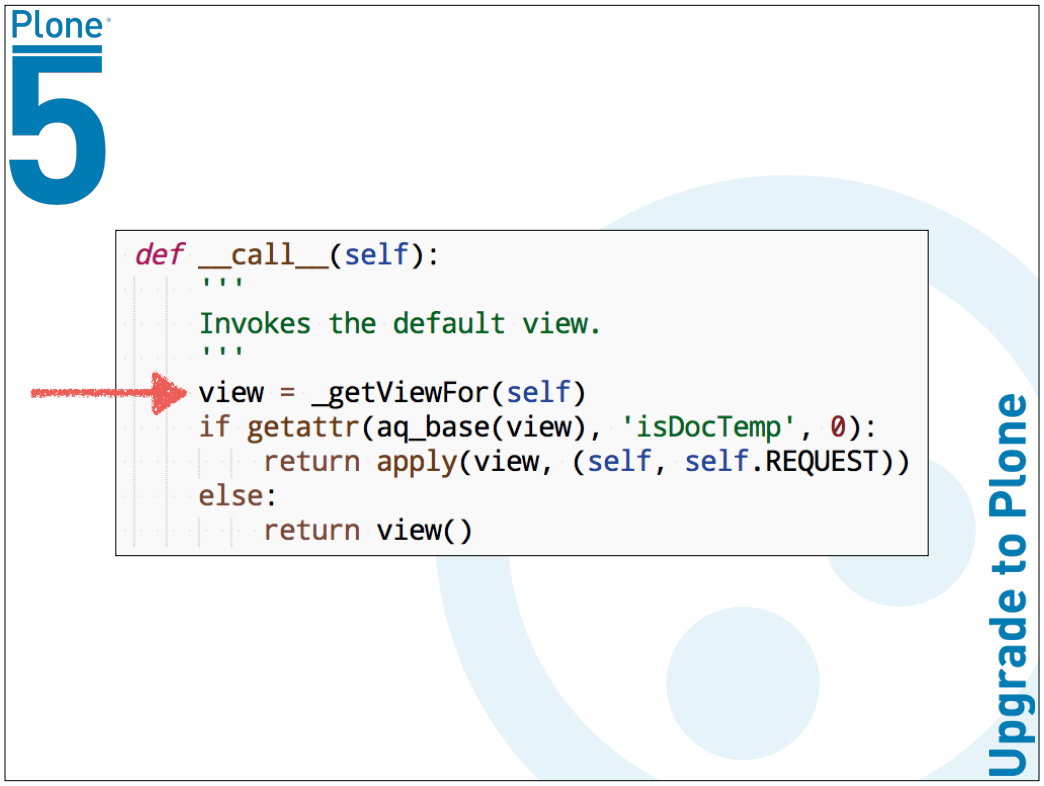


If we dig into the `mapply` function, we can see that after doing some other work, it returns the result of calling the object.



If we dig into the `mapply` function, we can see that after doing some other work, it returns the result of calling the object.

* For Content Objects in the CMF (and thus in Plone), The call method uses a function called _getViewFor to find the appropriate view for that object.
* Once the right “view” is found, it is called in turn to produce a result, which is returned to build the HTTP response body



* For Content Objects in the CMF (and thus in Plone), The call method uses a function called _getViewFor to find the appropriate view for that object.
* Once the right “view” is found, it is called in turn to produce a result, which is returned to build the HTTP response body

* For Content Objects in the CMF (and thus in Plone), The call method uses a function called _getViewFor to find the appropriate view for that object.
* Once the right “view” is found, it is called in turn to produce a result, which is returned to build the HTTP response body

Plone 5

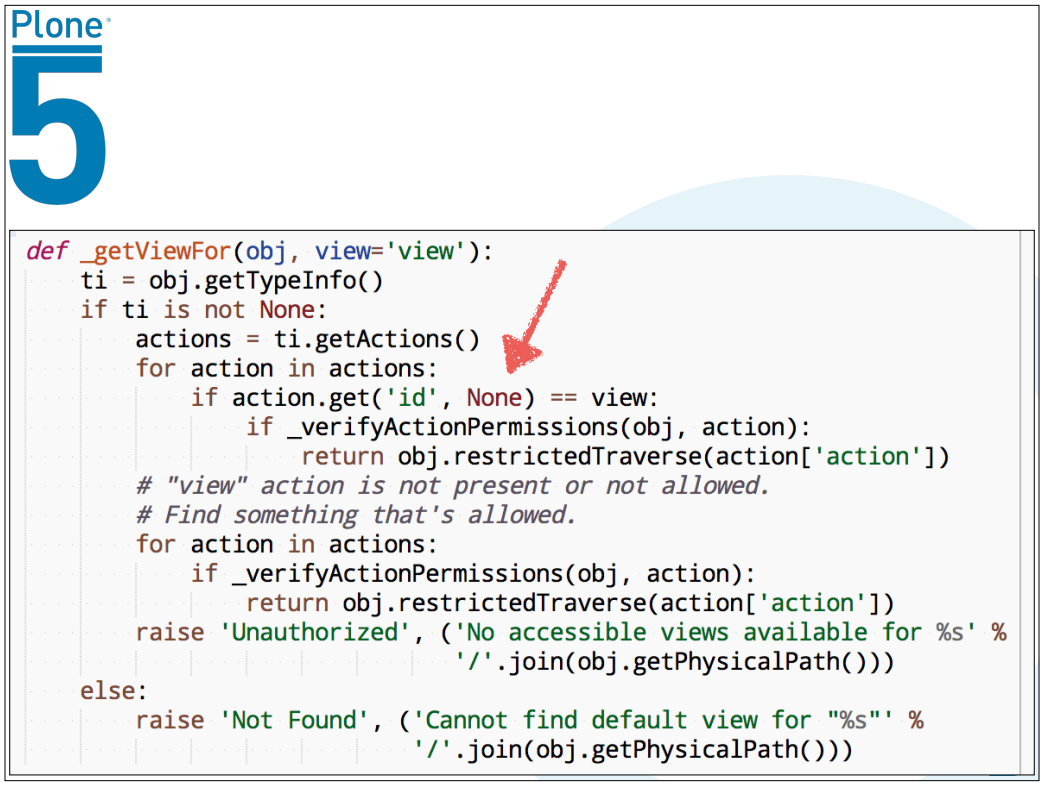
```
def _getViewFor(obj, view='view'):
    ti = obj.getTypeInfo()
    if ti is not None:
        actions = ti.getActions()
        for action in actions:
            if action.get('id', None) == view:
                if _verifyActionPermissions(obj, action):
                    return obj.restrictedTraverse(action['action'])
        # "view" action is not present or not allowed.
        # Find something that's allowed.
        for action in actions:
            if _verifyActionPermissions(obj, action):
                return obj.restrictedTraverse(action['action'])
        raise 'Unauthorized', ('No accessible views available for %s' %
                               '/'.join(obj.getPhysicalPath()))
    else:
        raise 'Not Found', ('Cannot find default view for "%s"' %
                            '/'.join(obj.getPhysicalPath()))
```

If we drill a bit farther, we find that our object is supposed to provide some sort of “type info”.
That type info is supposed to provide us with an action who’s ID matches the name of the view we want.
Then we are supposed to traverse on from the object to find this action by name. That’s the thing we return so it can be called to publish our object.

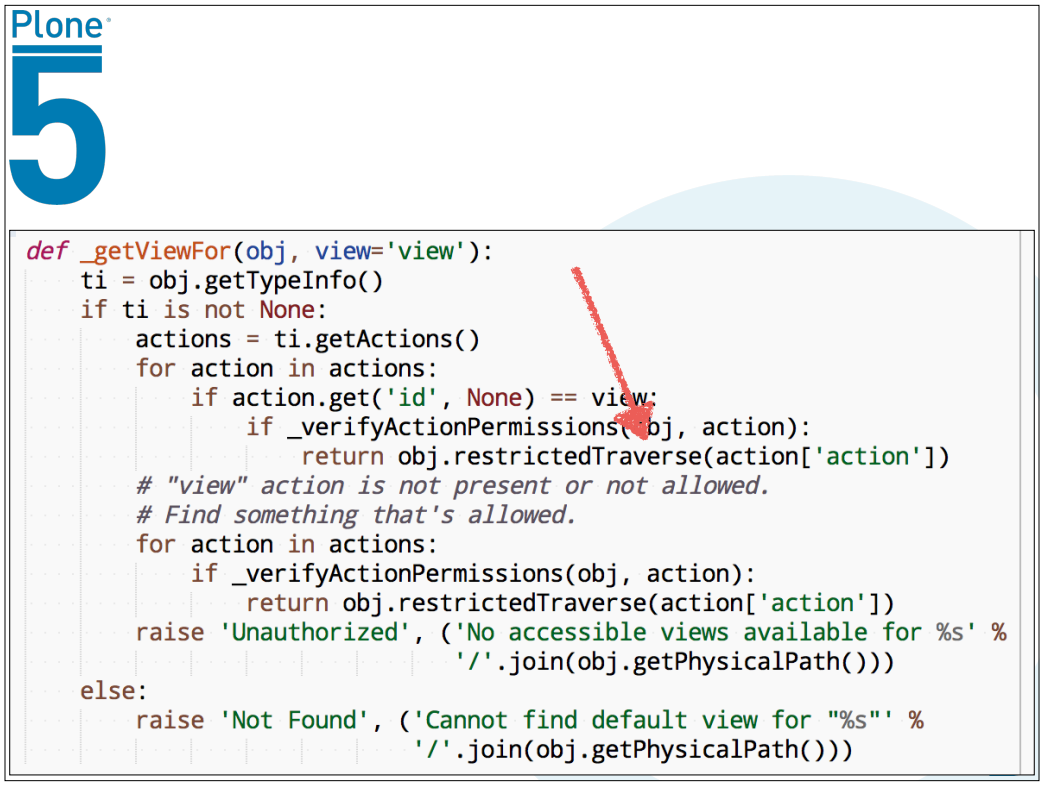
Plone 5

```
def _getViewFor(obj, view='view'):
    ti = obj.getTypeInfo()
    if ti is not None:
        actions = ti.getActions()
        for action in actions:
            if action.get('id', None) == view:
                if _verifyActionPermissions(obj, action):
                    return obj.restrictedTraverse(action['action'])
        # "view" action is not present or not allowed.
        # Find something that's allowed.
        for action in actions:
            if _verifyActionPermissions(obj, action):
                return obj.restrictedTraverse(action['action'])
        raise 'Unauthorized', ('No accessible views available for %s' %
                               '/'.join(obj.getPhysicalPath()))
    else:
        raise 'Not Found', ('Cannot find default view for "%s"' %
                            '/'.join(obj.getPhysicalPath()))
```

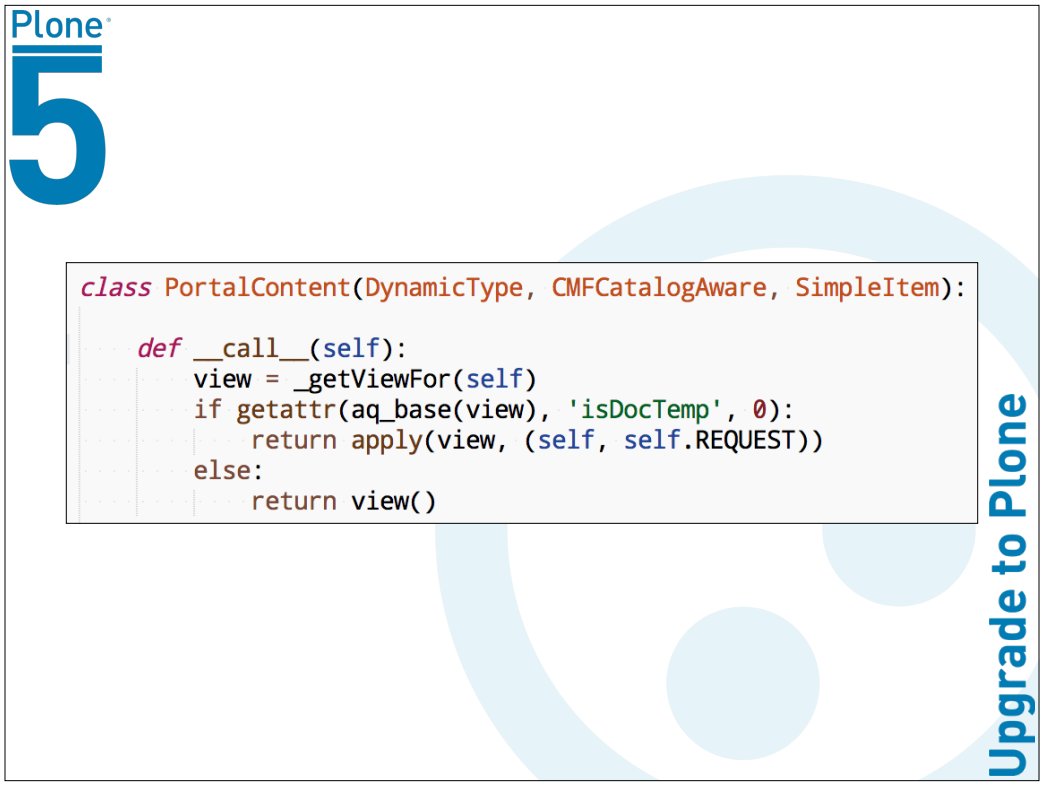
If we drill a bit farther, we find that our object is supposed to provide some sort of “type info”.
That type info is supposed to provide us with an action who’s ID matches the name of the view we want.
Then we are supposed to traverse on from the object to find this action by name. That’s the thing we return so it can be called to publish our object.



If we drill a bit farther, we find that our object is supposed to provide some sort of “type info”.
That type info is supposed to provide us with an action who’s ID matches the name of the view we want.
Then we are supposed to traverse on from the object to find this action by name. That’s the thing we return so it can be called to publish our object.



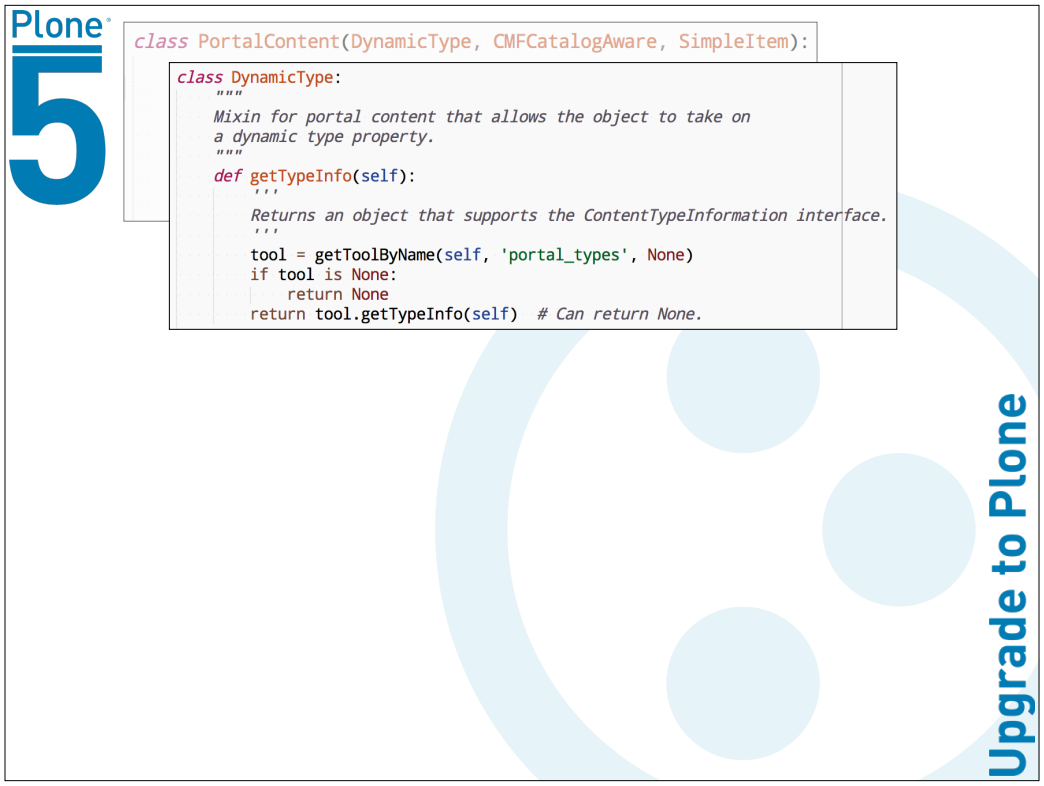
If we drill a bit farther, we find that our object is supposed to provide some sort of “type info”.
That type info is supposed to provide us with an action who’s ID matches the name of the view we want.
Then we are supposed to traverse on from the object to find this action by name. That’s the thing we return so it can be called to publish our object.



So our objects, in order to be published, need to provide this call special method

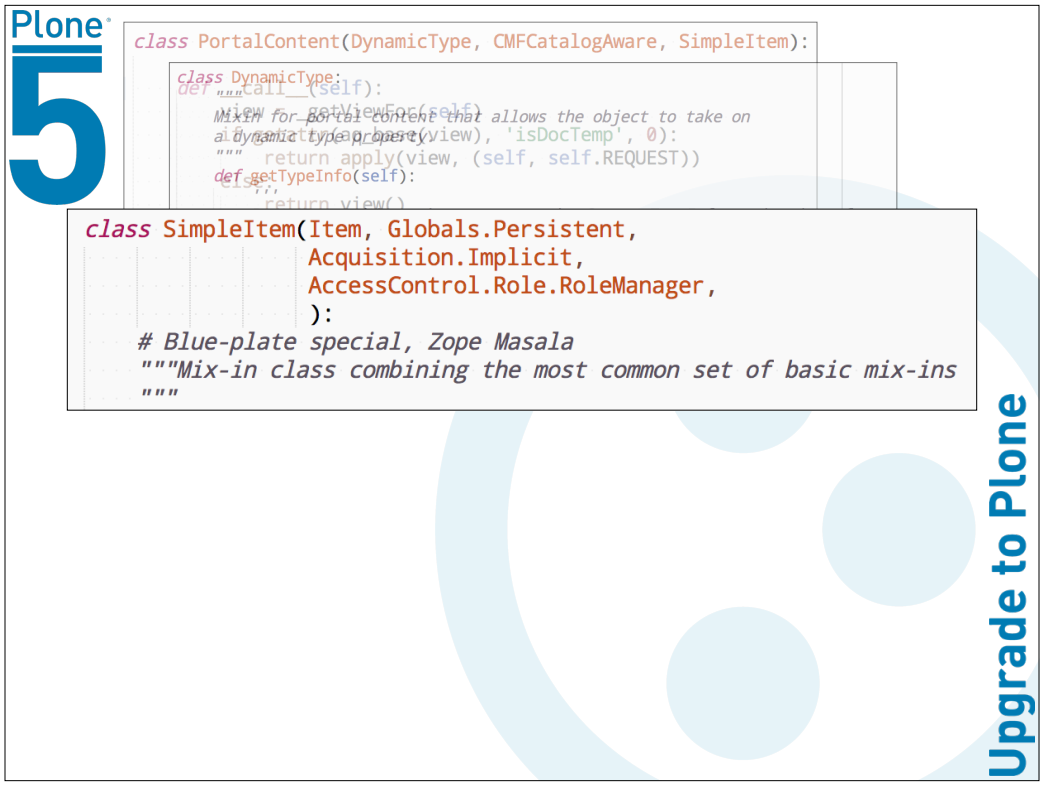
They also have to be able to provide “type info” so we can get the right action. They can get that from this mixin.

They also have to be able to provide “type info” so we can get the right action. They can get that from this mixin.



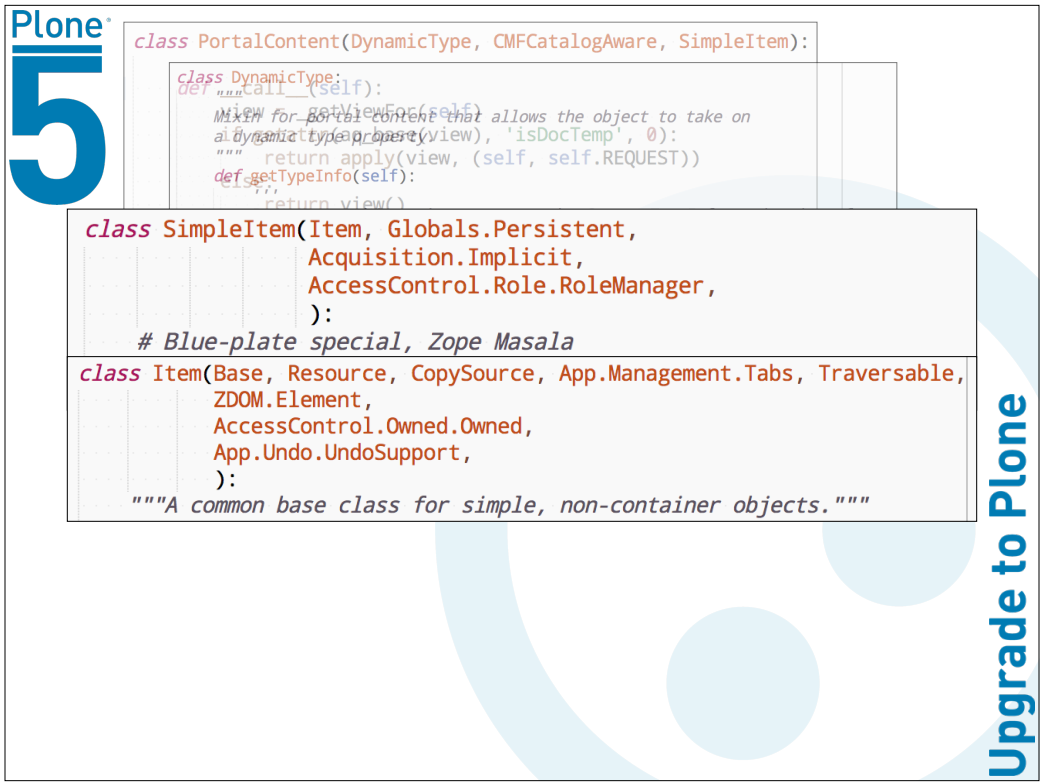
In addition, they have to be “traversable” so we can get to the action we find from the type info.

To get that, we look at “SimpleItem”,
which inherits from “Item”,
which inherits from the “Traversable” mixin.



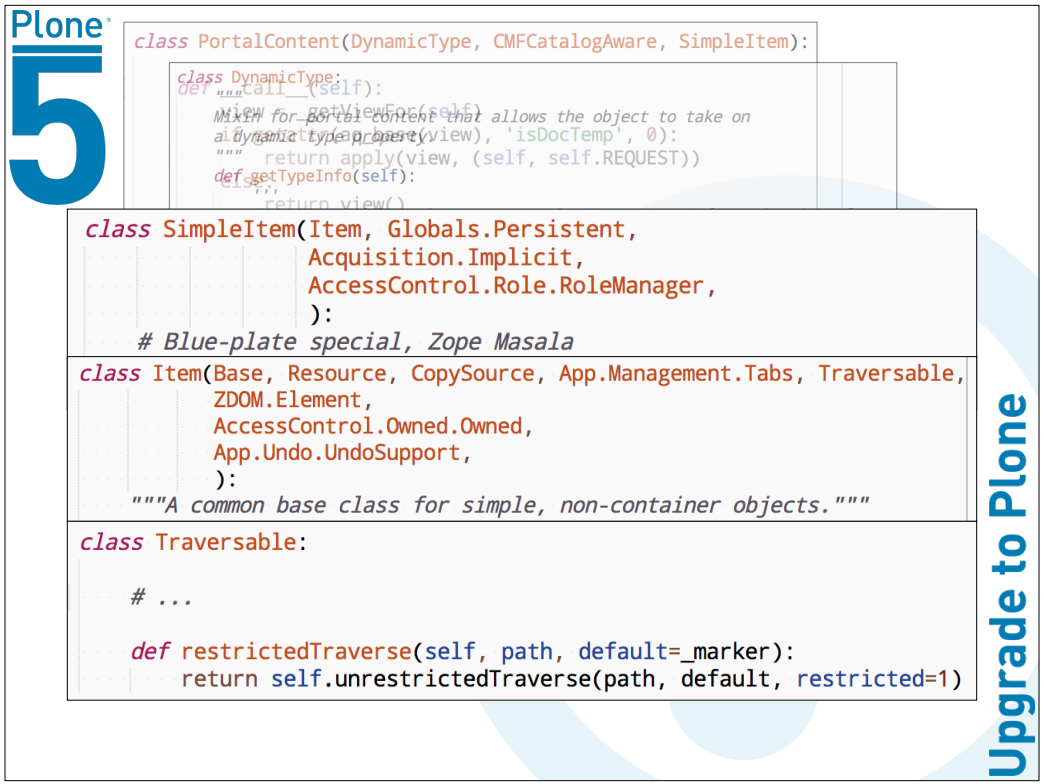
In addition, they have to be “traversable” so we can get to the action we find from the type info.

To get that, we look at “SimpleItem”,
which inherits from “Item”,
which inherits from the “Traversable” mixin.



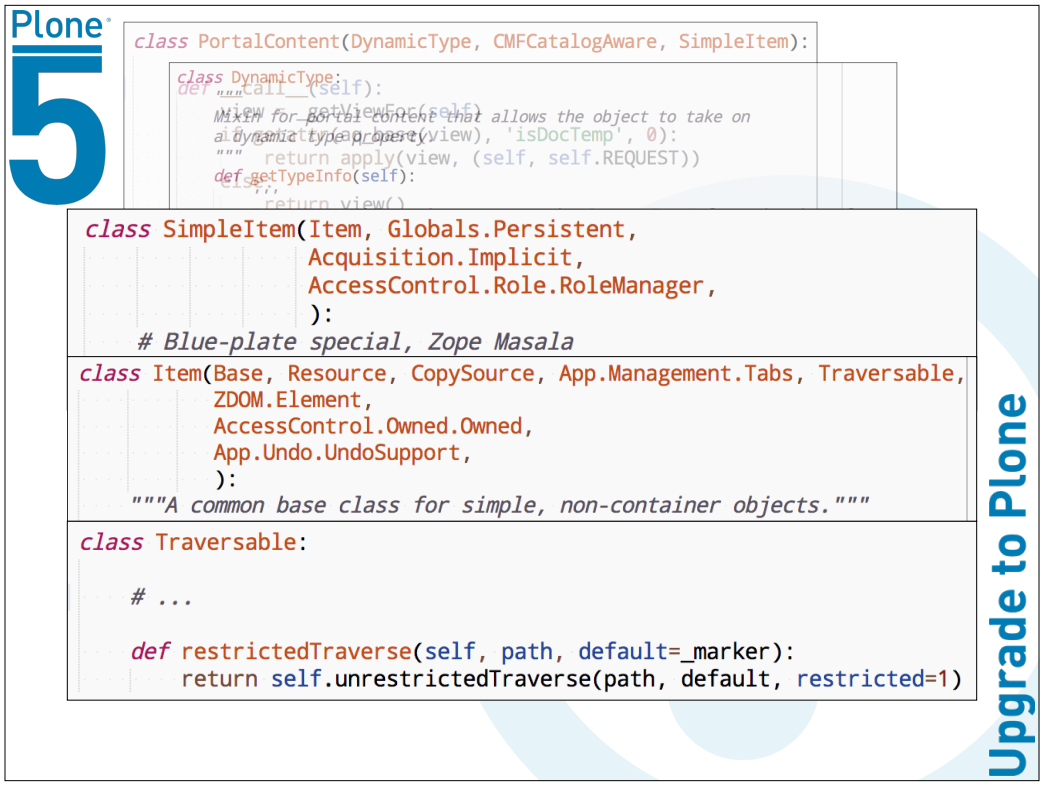
In addition, they have to be “traversable” so we can get to the action we find from the type info.

To get that, we look at “SimpleItem”,
which inherits from “Item”,
which inherits from the “Traversable” mixin.

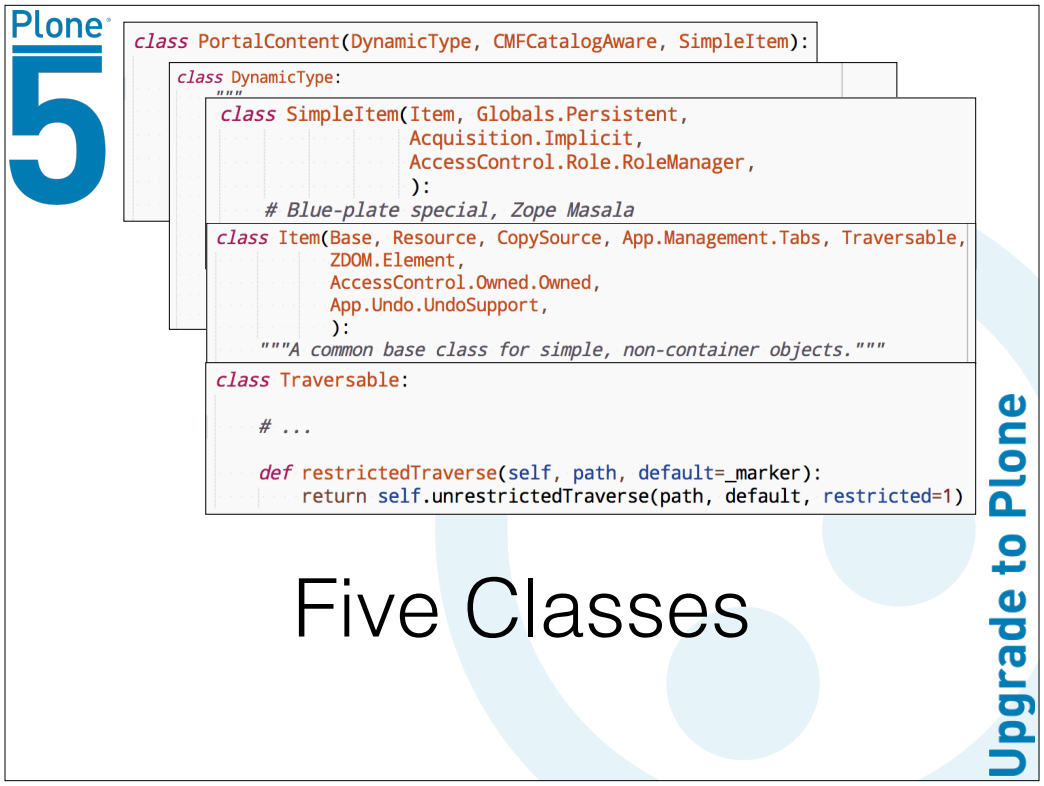


In addition, they have to be “traversable” so we can get to the action we find from the type info.

To get that, we look at “SimpleItem”,
which inherits from “Item”,
which inherits from the “Traversable” mixin.

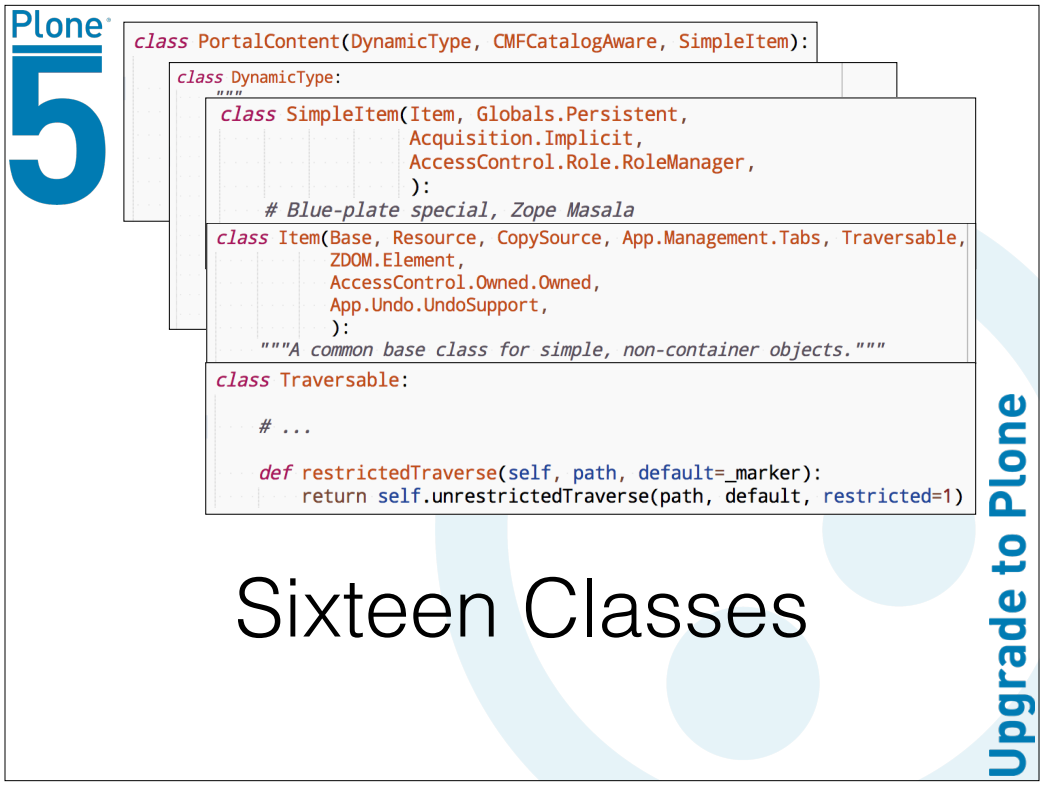


So to implement the ability to publish an object,

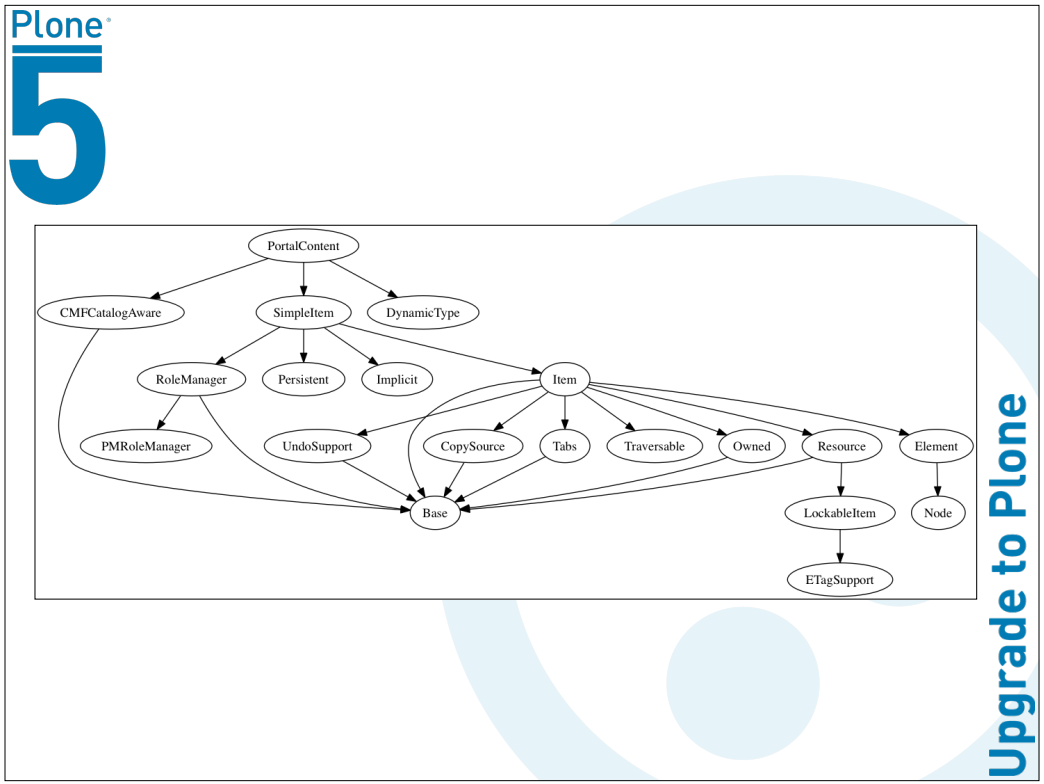


we use 5 classes.

But if we count all the inherited classes, which provide other required behaviors like Persistence, Access Control, Indexing, and so on:



the count of inherited classes comes to 16



And if we follow on to the entire dependency tree for this simple content object, we end up involving 20 separate classes.

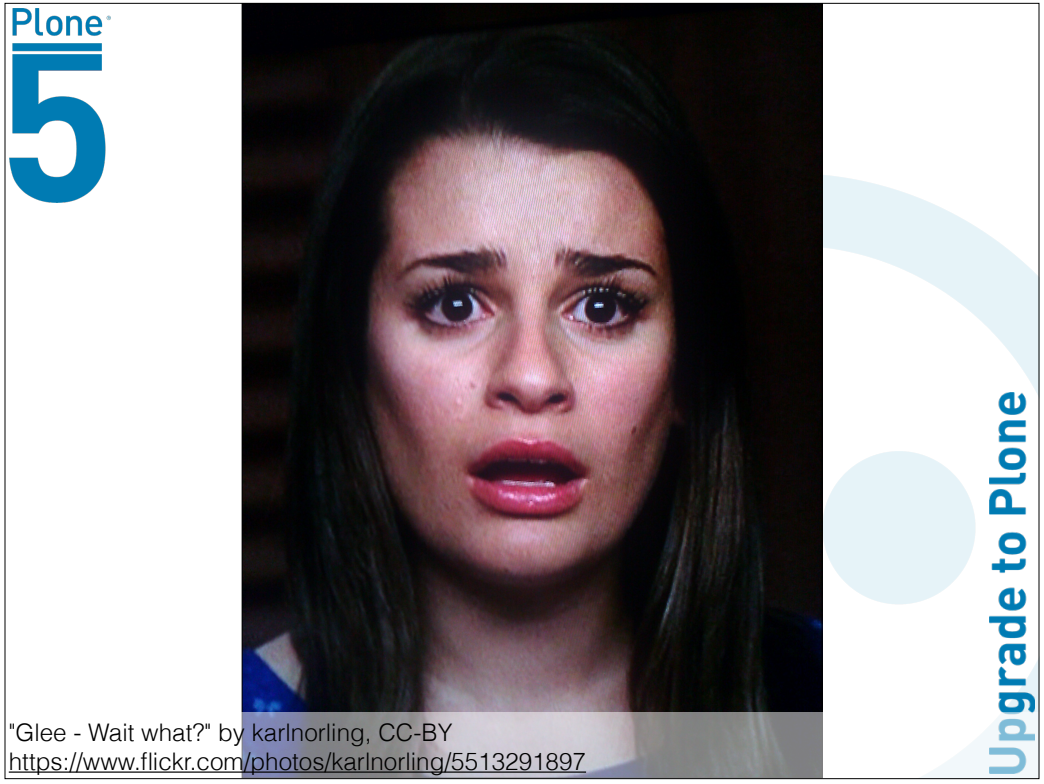
"Swiss Army Knife" by Dave Taylor CC-BY-2.0
https://www.flickr.com/photos/askdavetaylor/4261149346

Think about all the methods on all those classes.

Objects with a lot of responsibilities end up being big and hard to use.

And this is before we even begin to get up to the actual content types used in Plone.

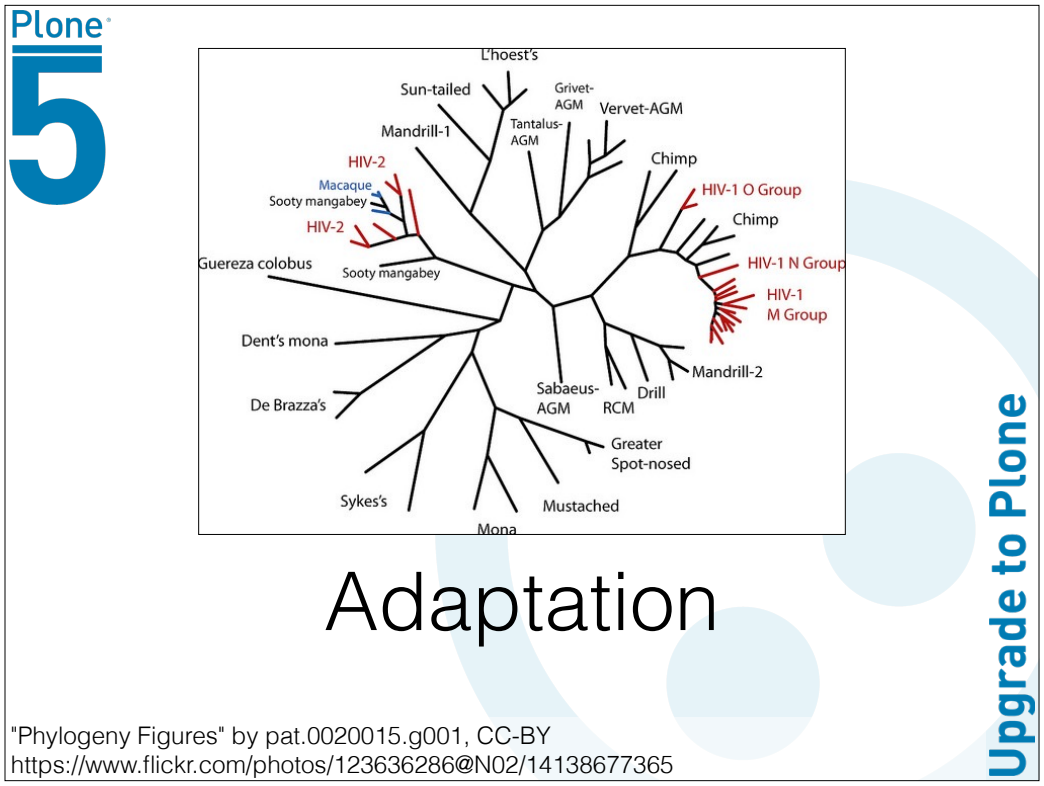
But before you get too down on Zope for this, look carefully at your own favorite frameworks. Providing the ability to use multiple inheritance makes it really easy for any reasonably complex system to end up in this kind of tangle.



"Glee - Wait what?" by karlnorling, CC-BY
https://www.flickr.com/photos/karlnorling/5513291897

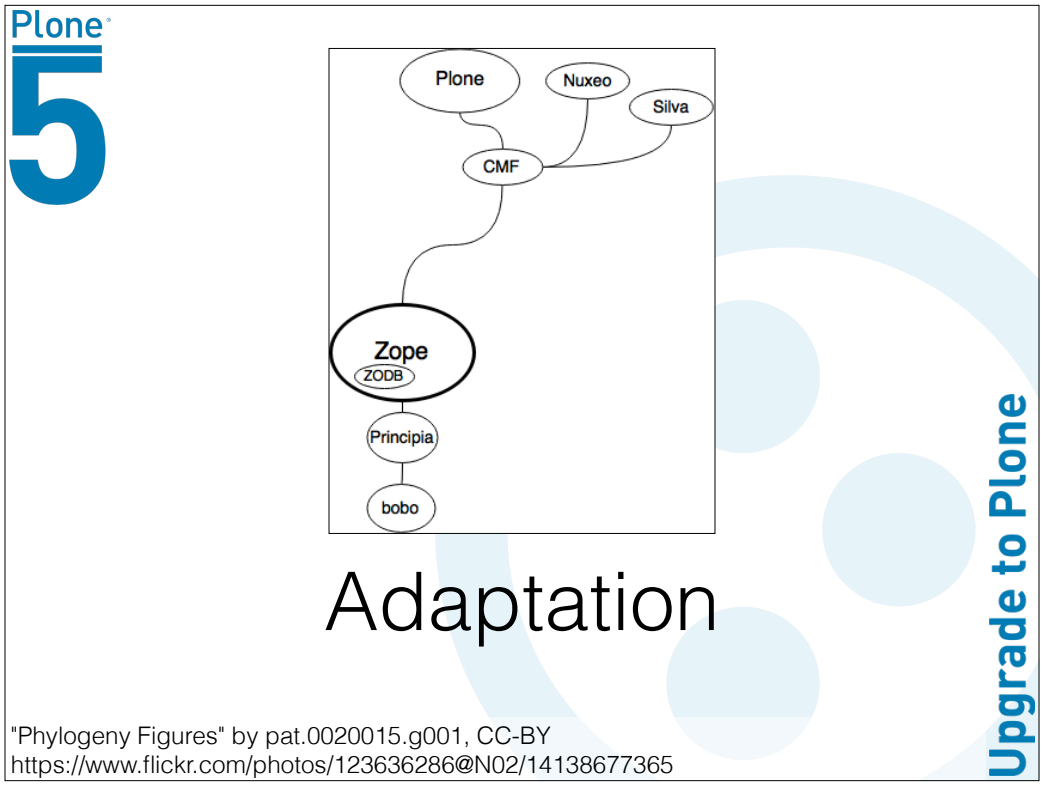
And this is before we even begin to get up to the actual content types used in Plone.

But before you get too down on Zope for this, look carefully at your own favorite frameworks. Providing the ability to use multiple inheritance makes it really easy for any reasonably complex system to end up in this kind of tangle.



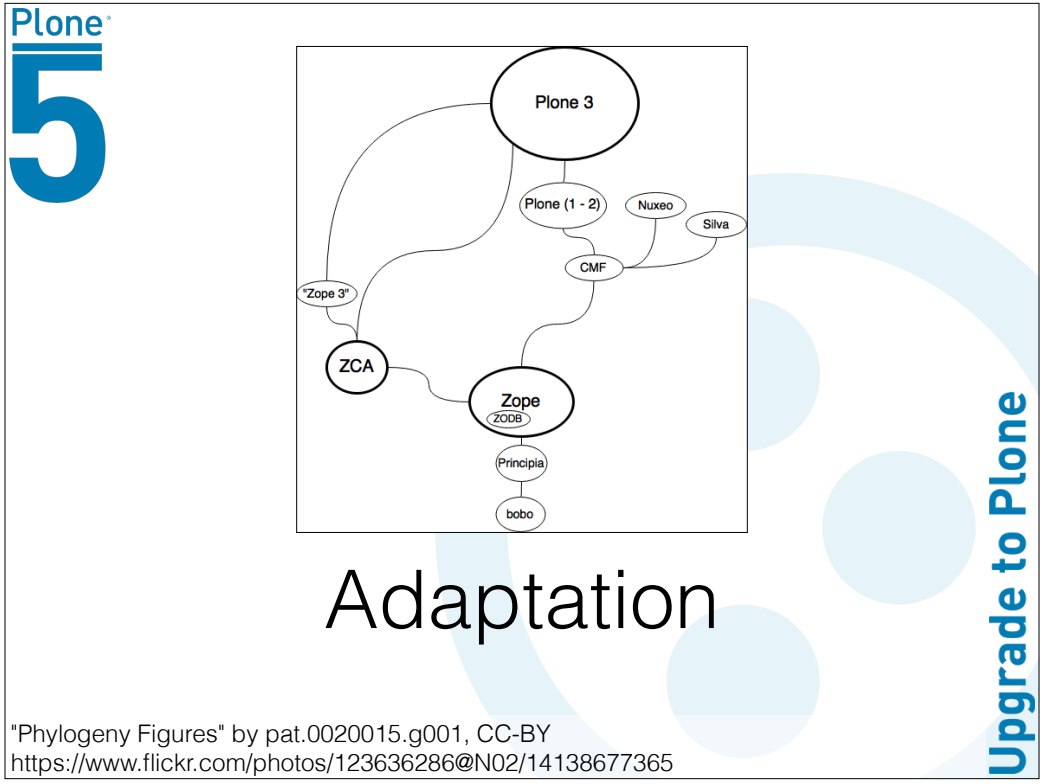
Plone inherited this problem from Zope and the CMF.

But the work that Zope did on the ZCA gave the key to fixing it as well.



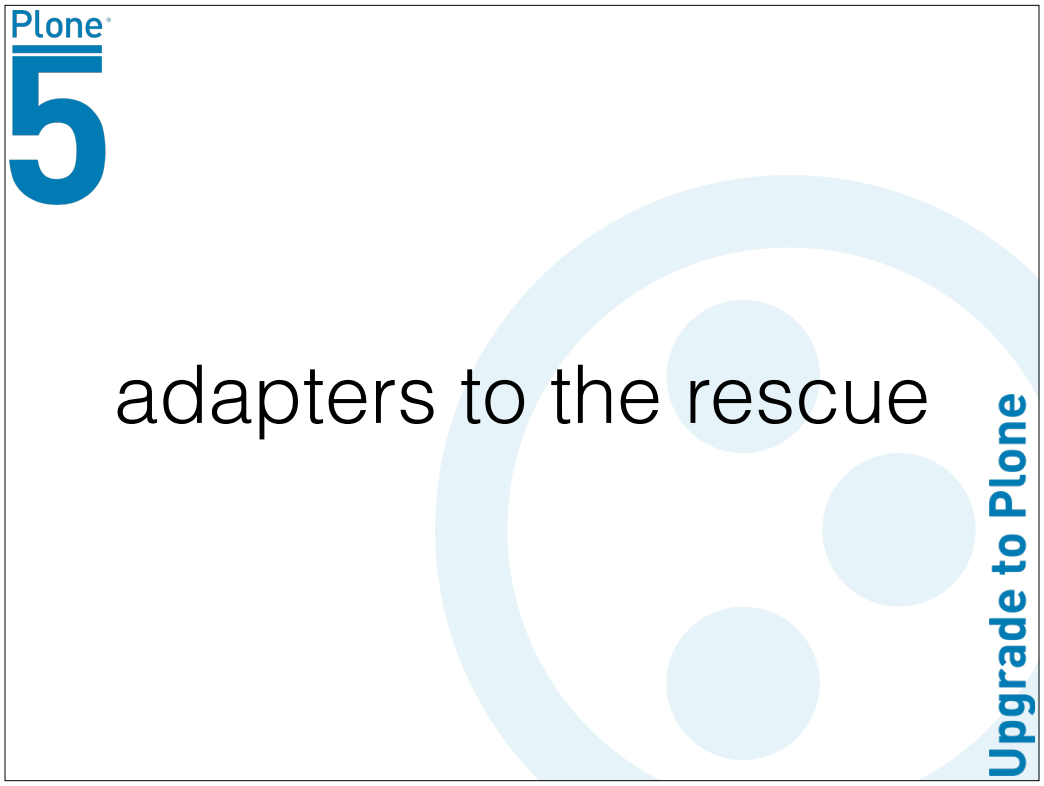
Plone inherited this problem from Zope and the CMF.

But the work that Zope did on the ZCA gave the key to fixing it as well.



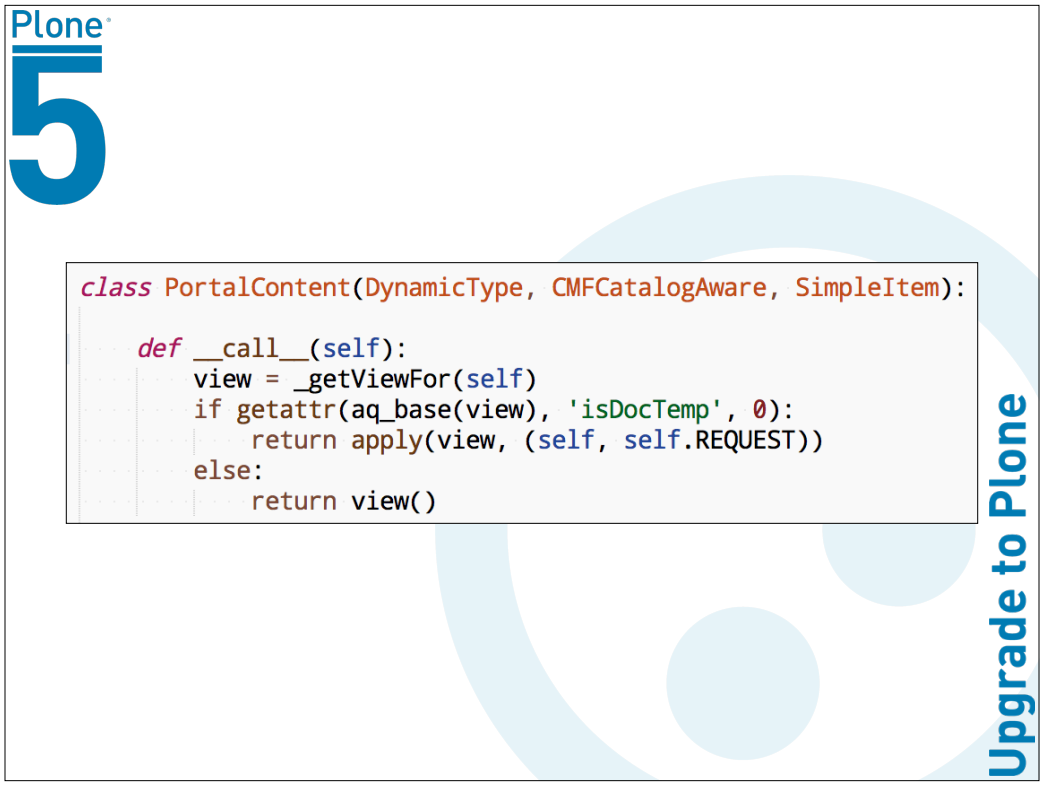
Plone inherited this problem from Zope and the CMF.

But the work that Zope did on the ZCA gave the key to fixing it as well.



After all, in a world with adapters, do we really need to have objects publish themselves directly?

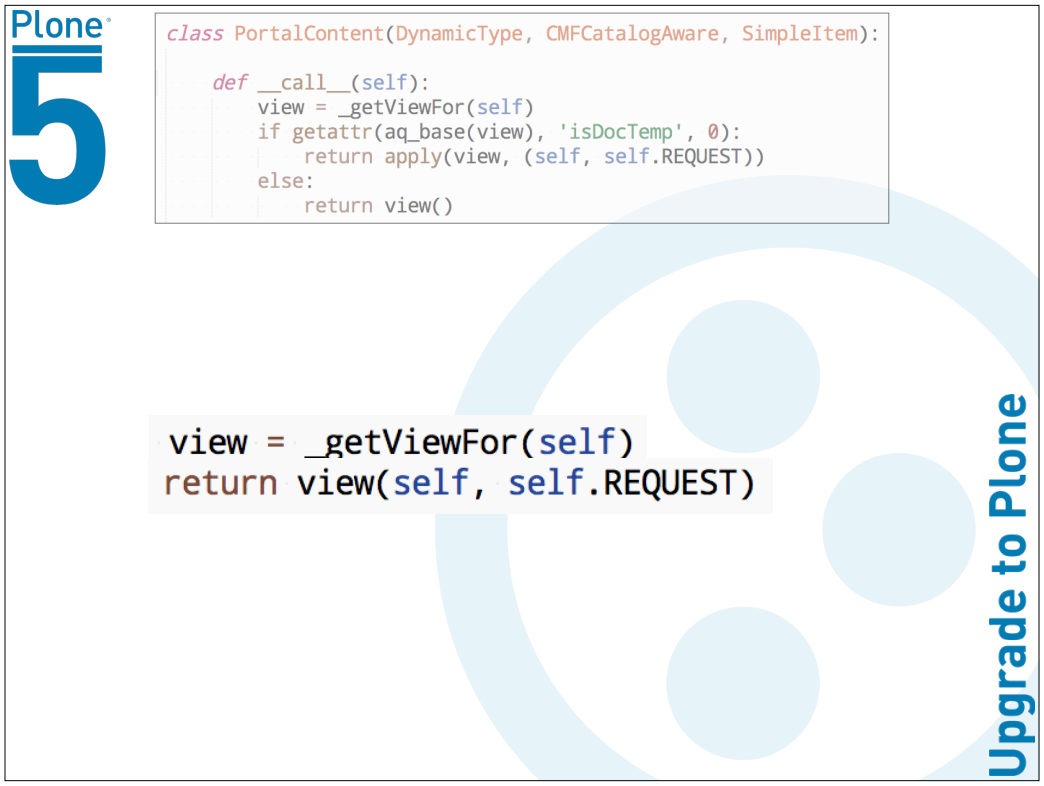
remember that call method from our basic content class?
Take another, closer look at what's going on here.



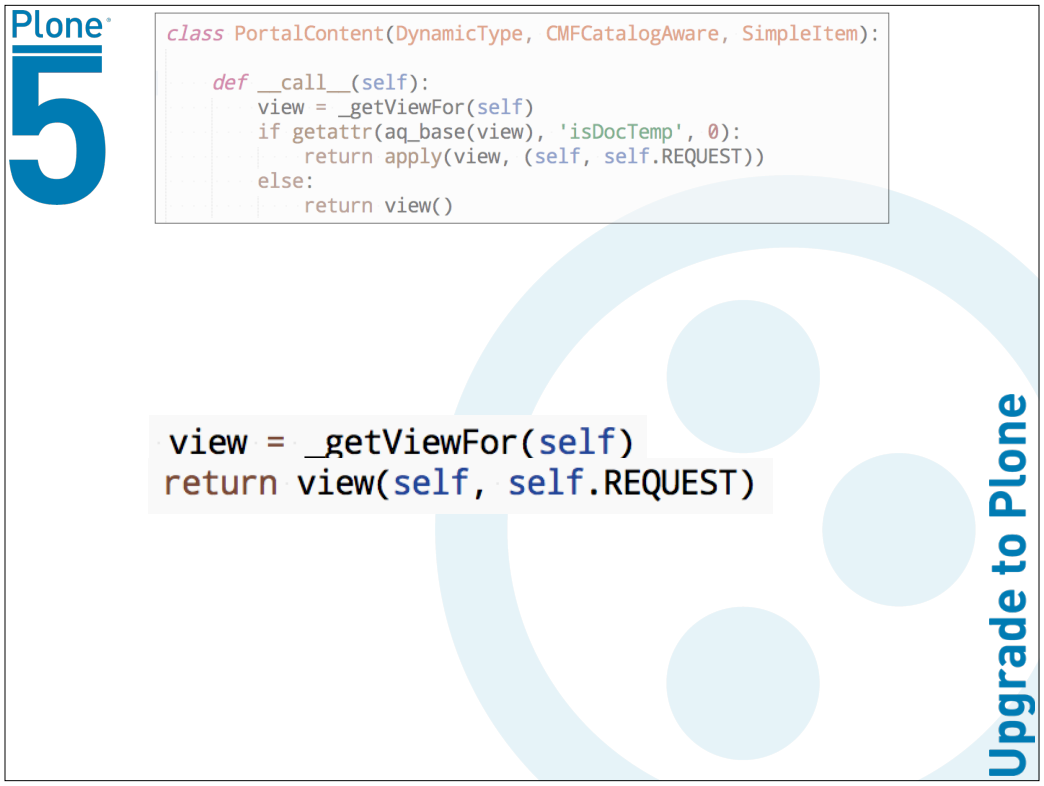
After all, in a world with adapters, do we really need to have objects publish themselves directly?

remember that call method from our basic content class?
Take another, closer look at what's going on here.

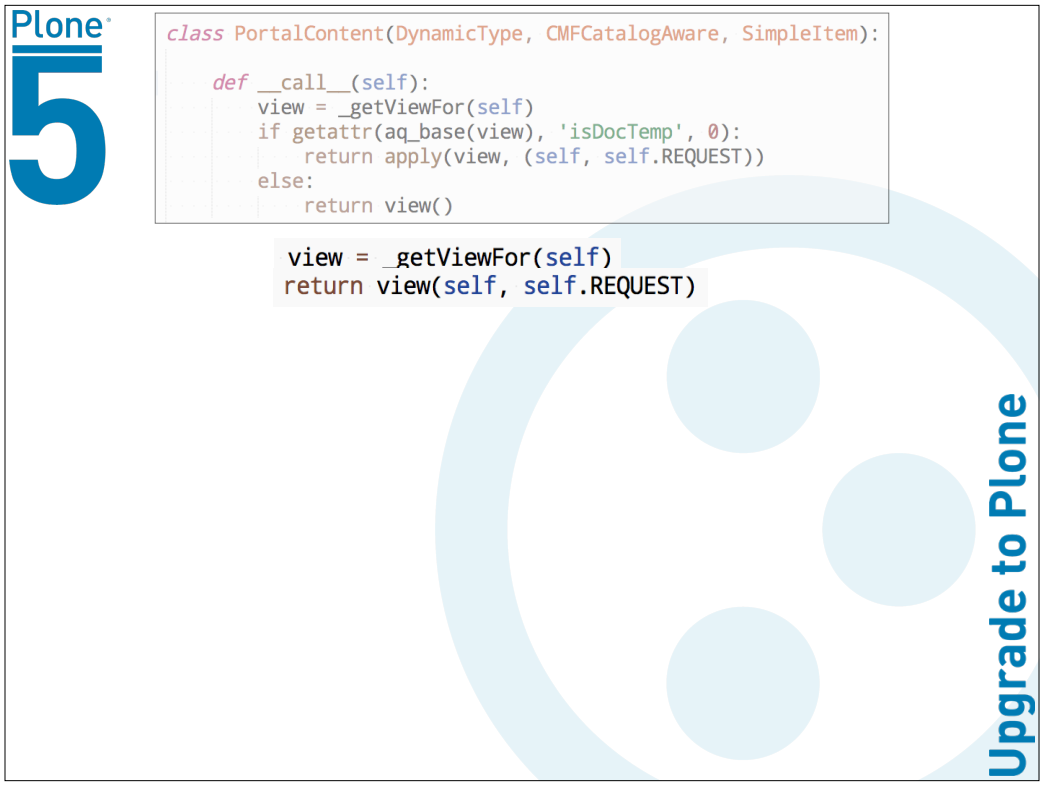
At the core, we are looking up a callable we'll name 'view', and then using apply to call it with the content object and the request as arguments.
Simplified, it might look like this:



At the core, we are looking up a callable we'll name 'view', and then using apply to call it with the content object and the request as arguments.
Simplified, it might look like this:



What if we rethink how this works.



What if this thing we are calling ‘view’ is a callable class, one that *contains* the content object and the request?

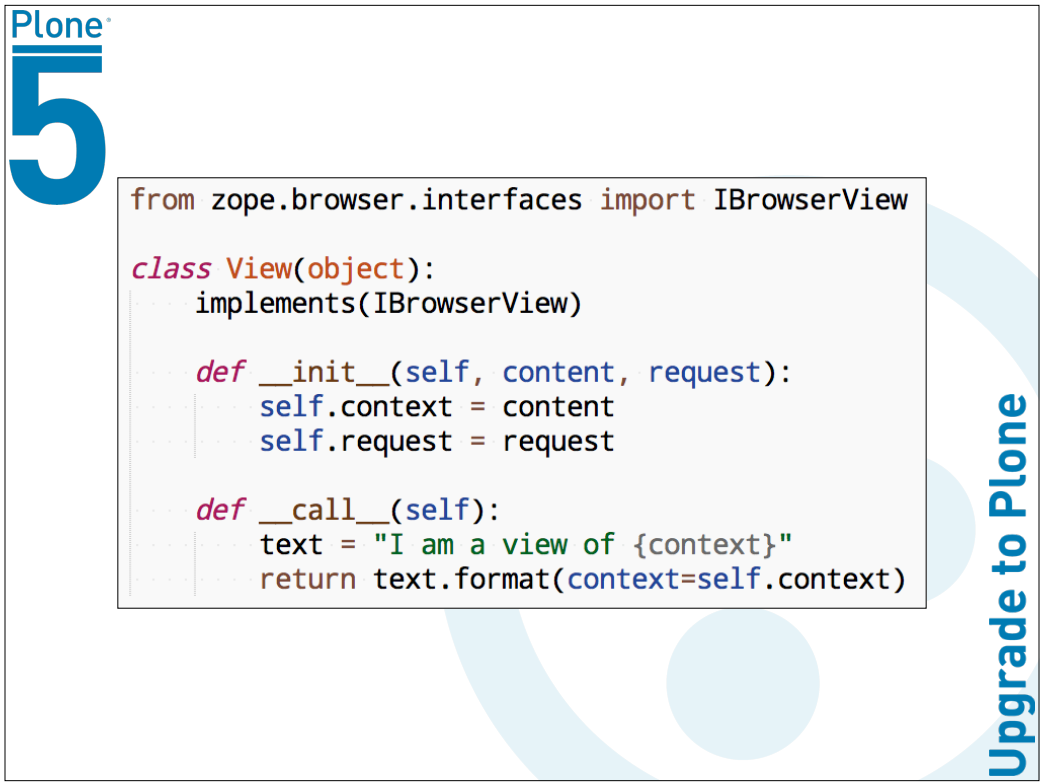
For those of you in the audience who use other web frameworks, this should start to look pretty familiar. It looks an awful lot like a very simple class-based view.

But Plone doesn’t have url-based dispatch to be able to look up class-based views. Plone uses traversal and object publication.

What if this thing we are calling ‘view’ is a callable class, one that *contains* the content object and the request?

For those of you in the audience who use other web frameworks, this should start to look pretty familiar. It looks an awful lot like a very simple class-based view.

But Plone doesn’t have url-based dispatch to be able to look up class-based views. Plone uses traversal and object publication.

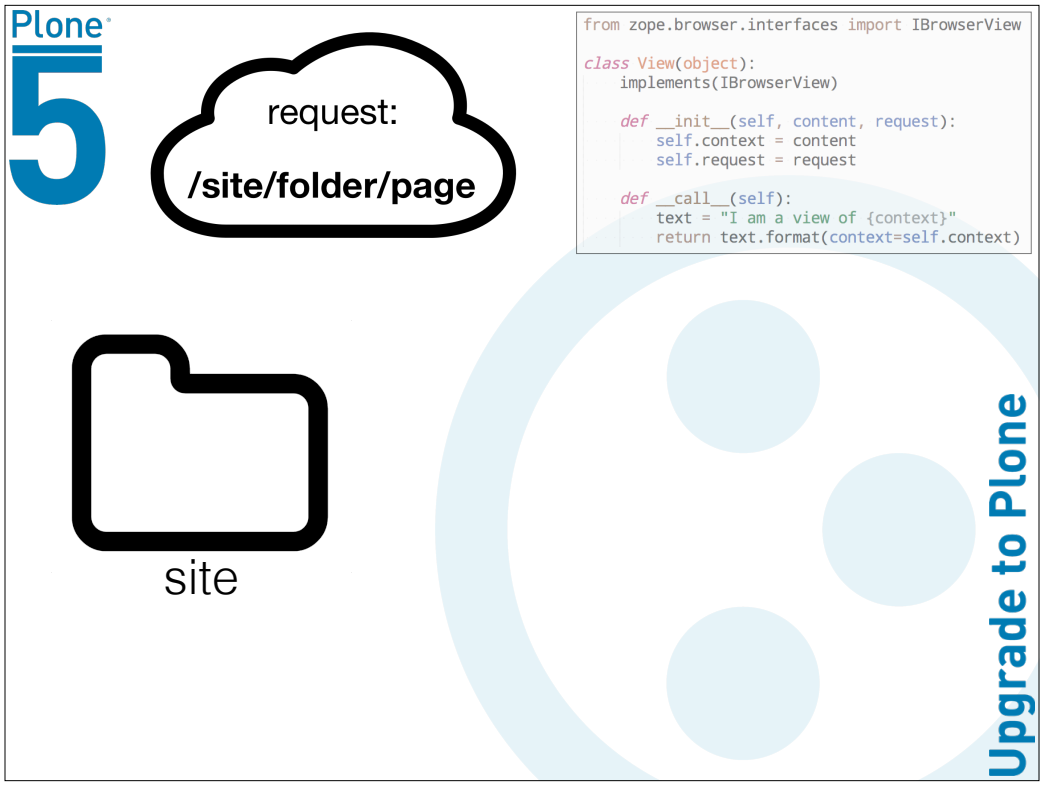


Plone does have adapters and interfaces, though.

We can make a minor update to our view class claiming that it implements a “BrowserView” interface. Notice that this “adapter” adapts both a “context” and a “request”.

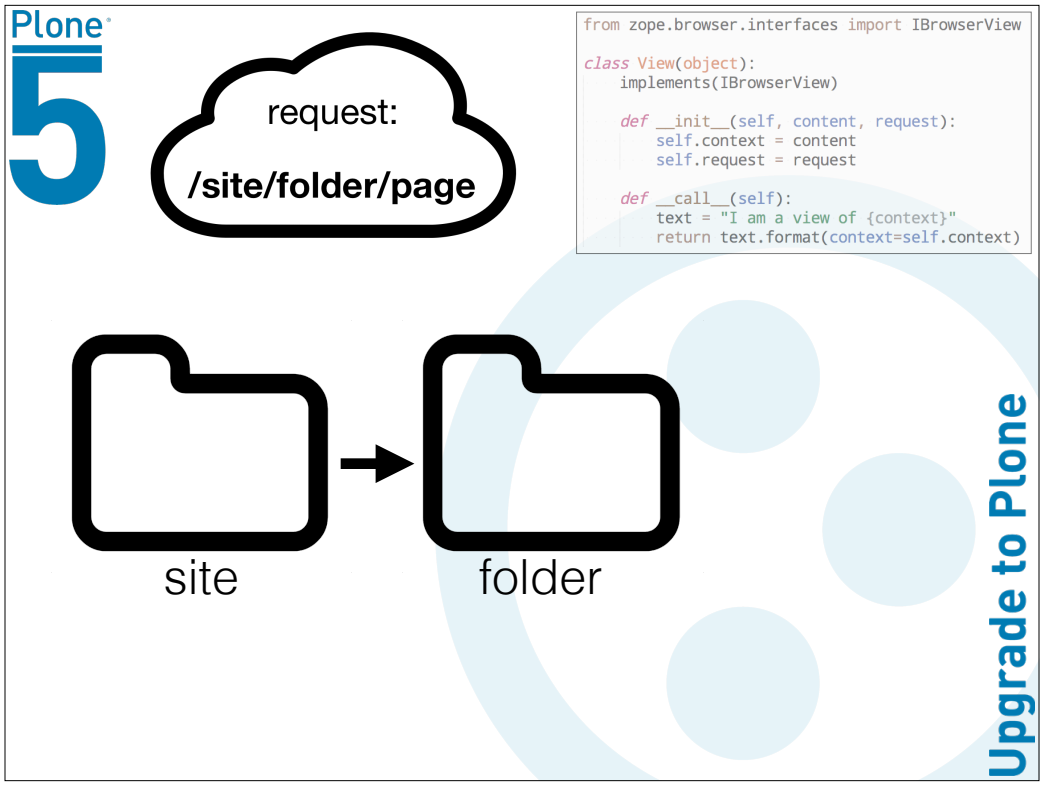
When a request comes to our site now, we use traversal to find the object as before.

But now, instead of calling it and having it publish itself we can look up a “browser view” that can adapt this object and request.



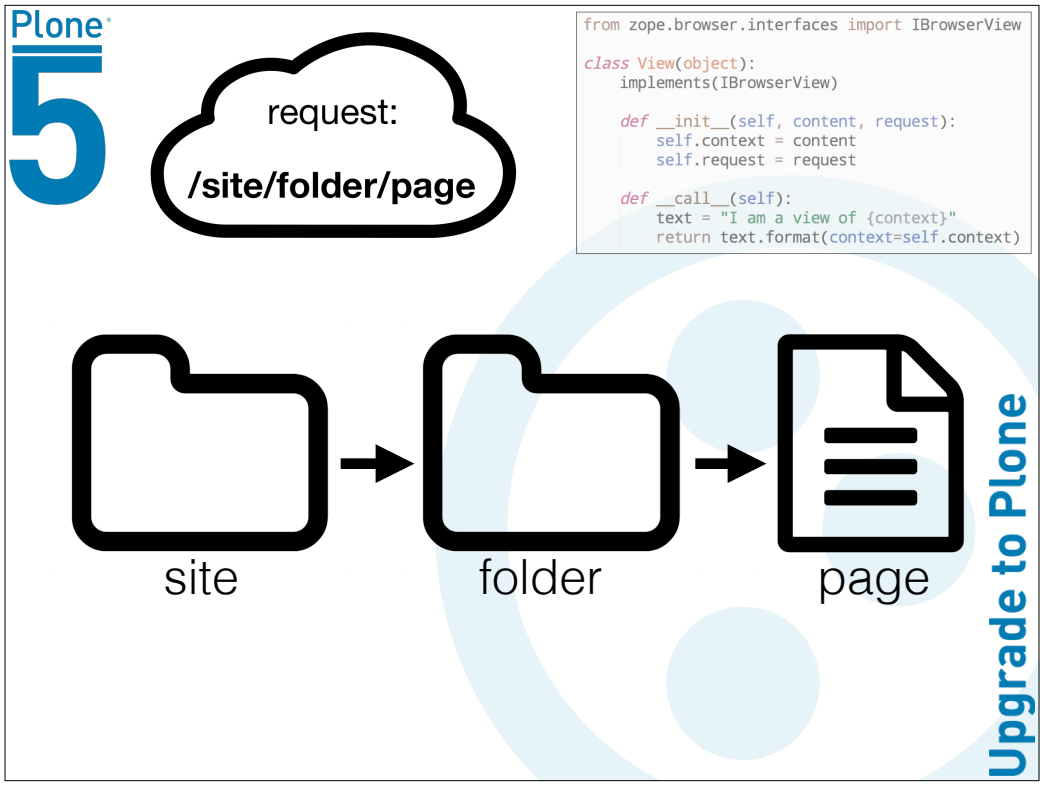
When a request comes to our site now, we use traversal to find the object as before.

But now, instead of calling it and having it publish itself we can look up a “browser view” that can adapt this object and request.



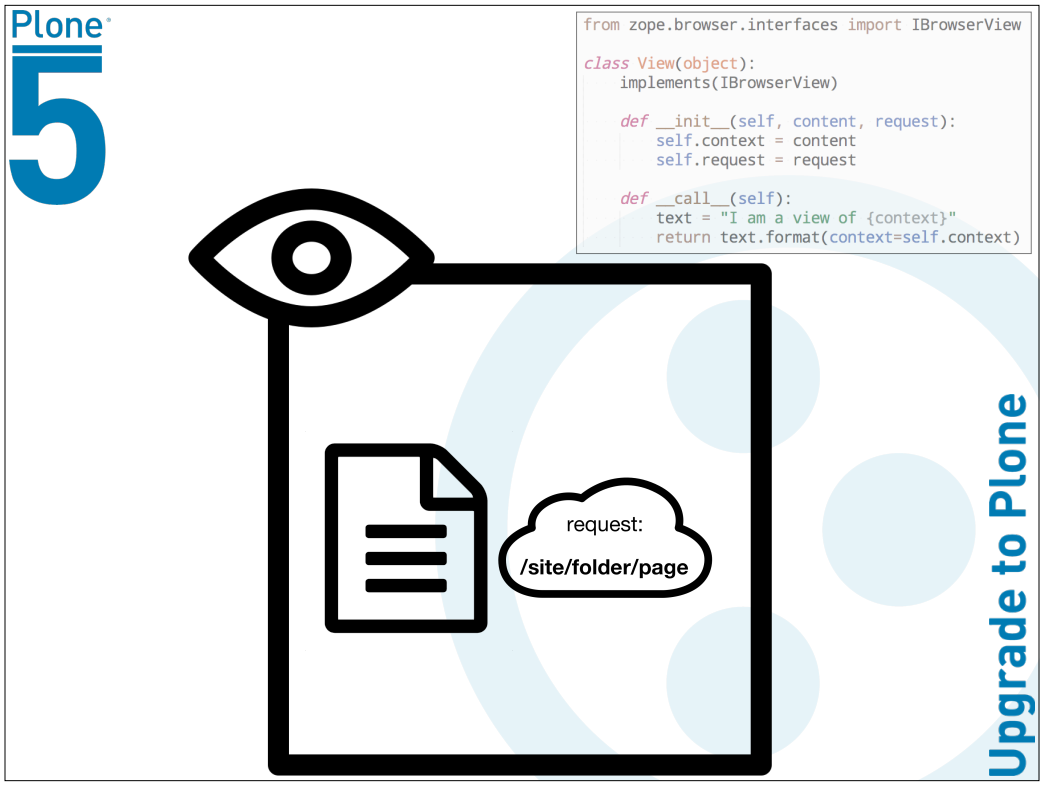
When a request comes to our site now, we use traversal to find the object as before.

But now, instead of calling it and having it publish itself we can look up a “browser view” that can adapt this object and request.

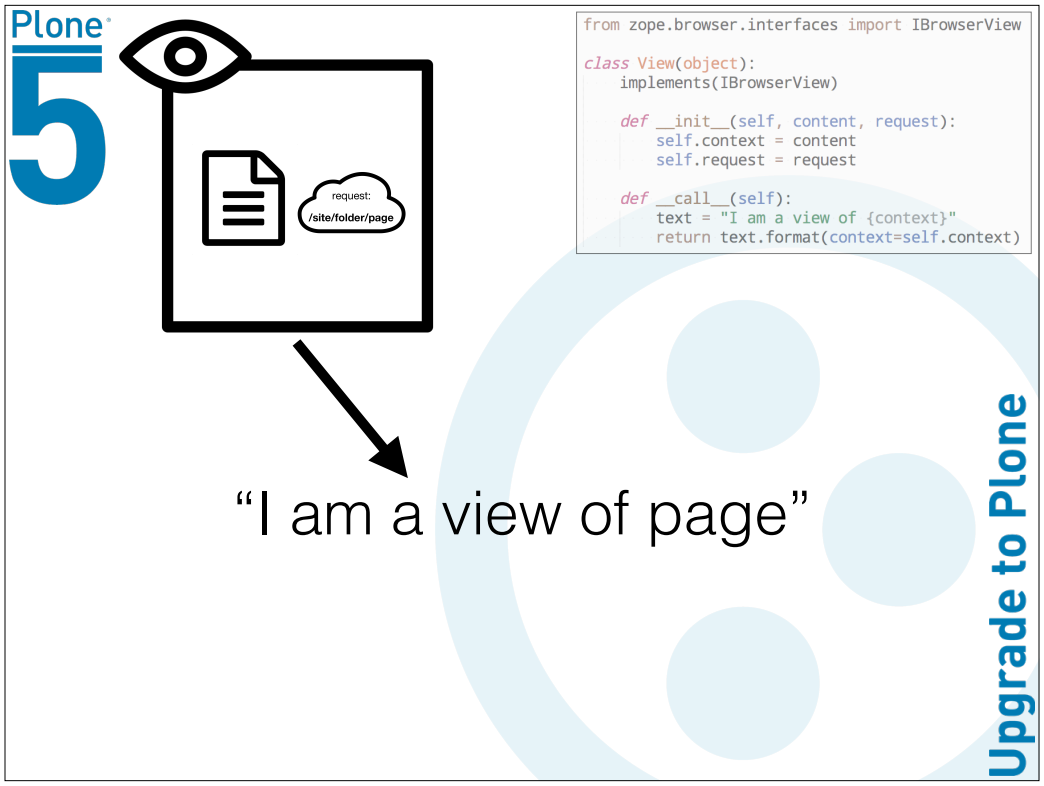


When a request comes to our site now, we use traversal to find the object as before.

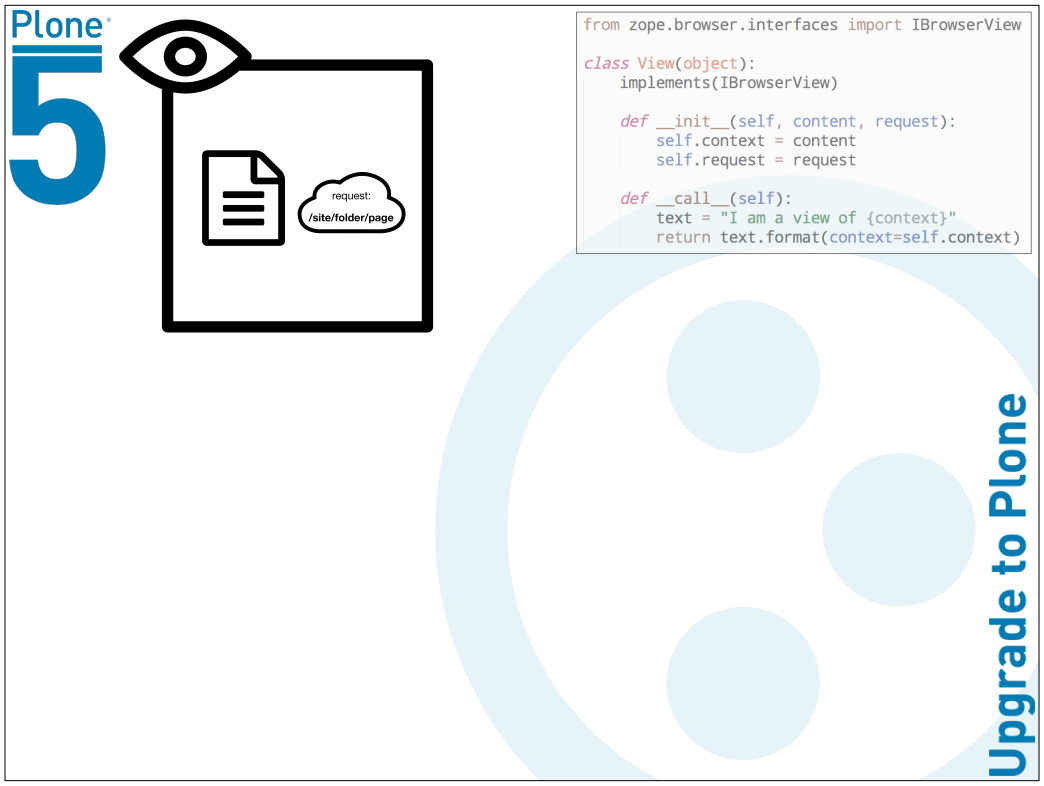
But now, instead of calling it and having it publish itself we can look up a “browser view” that can adapt this object and request.



If we then call that object.

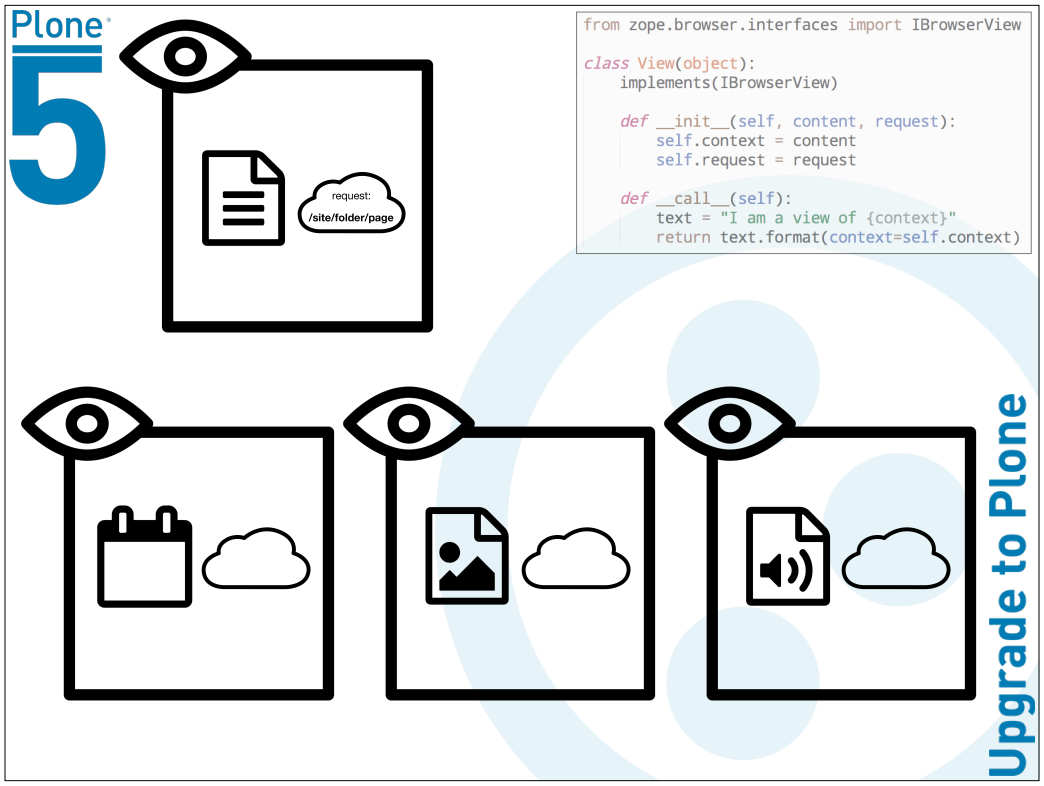


we get back the representation it produces



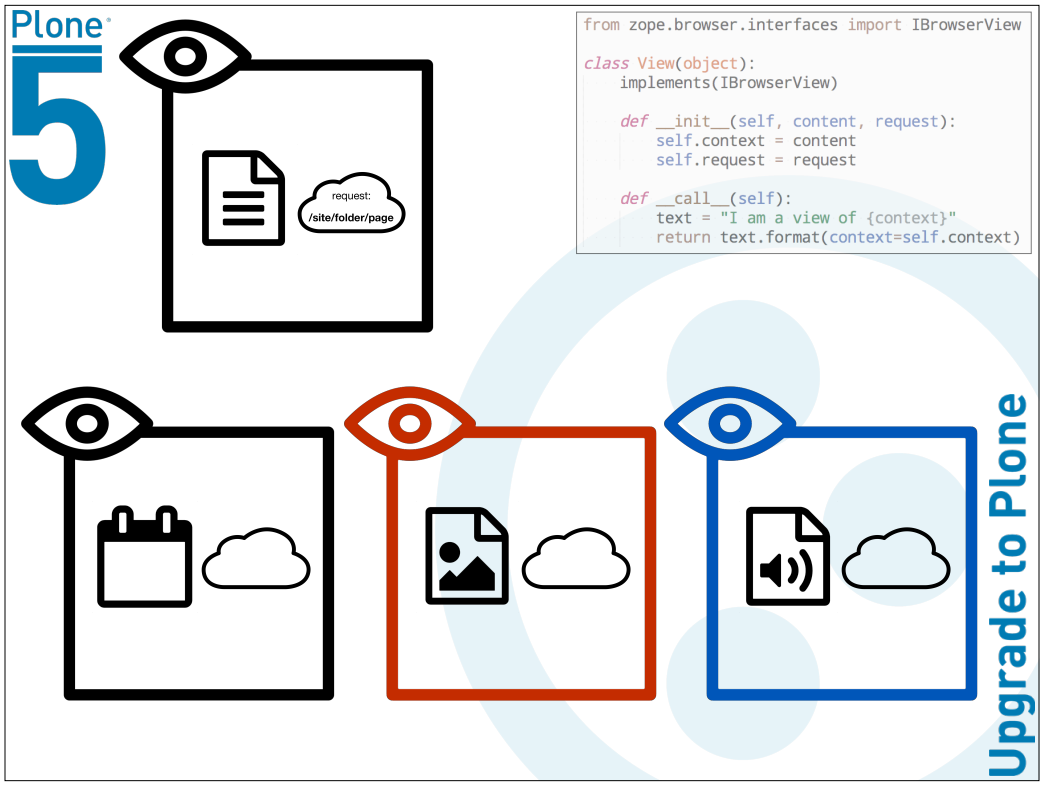
The same default view can be used for different types of content.

Or, we can have different adapters found for different types of content.



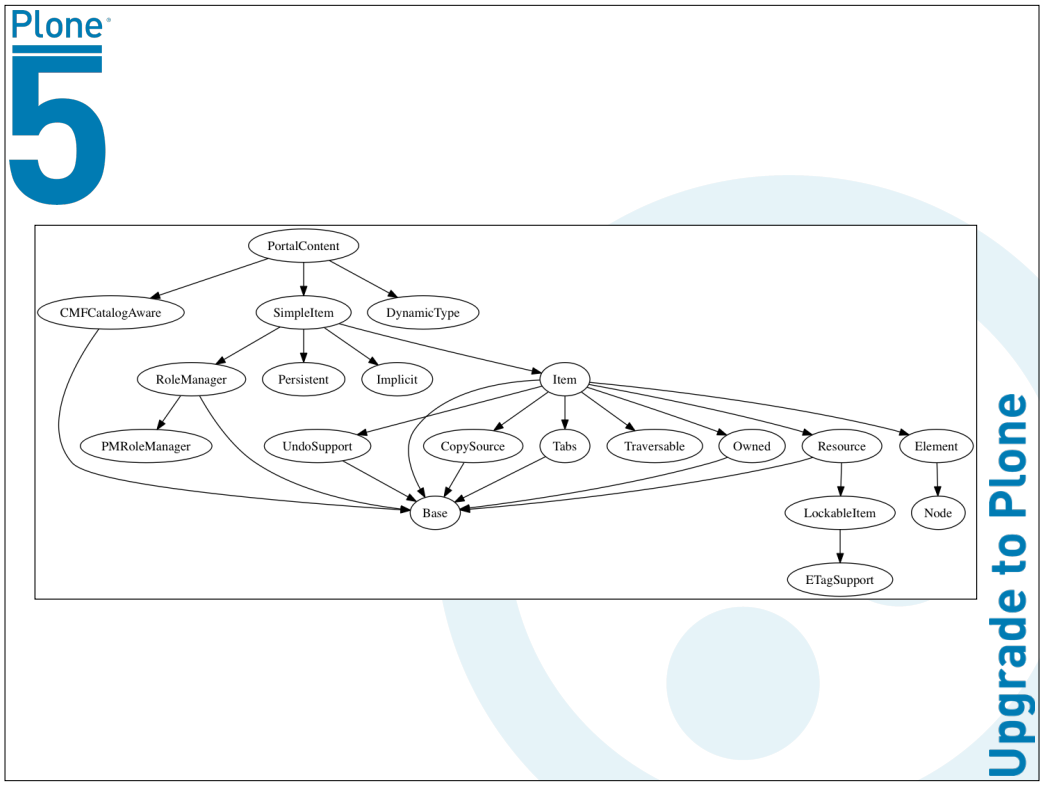
The same default view can be used for different types of content.

Or, we can have different adapters found for different types of content.



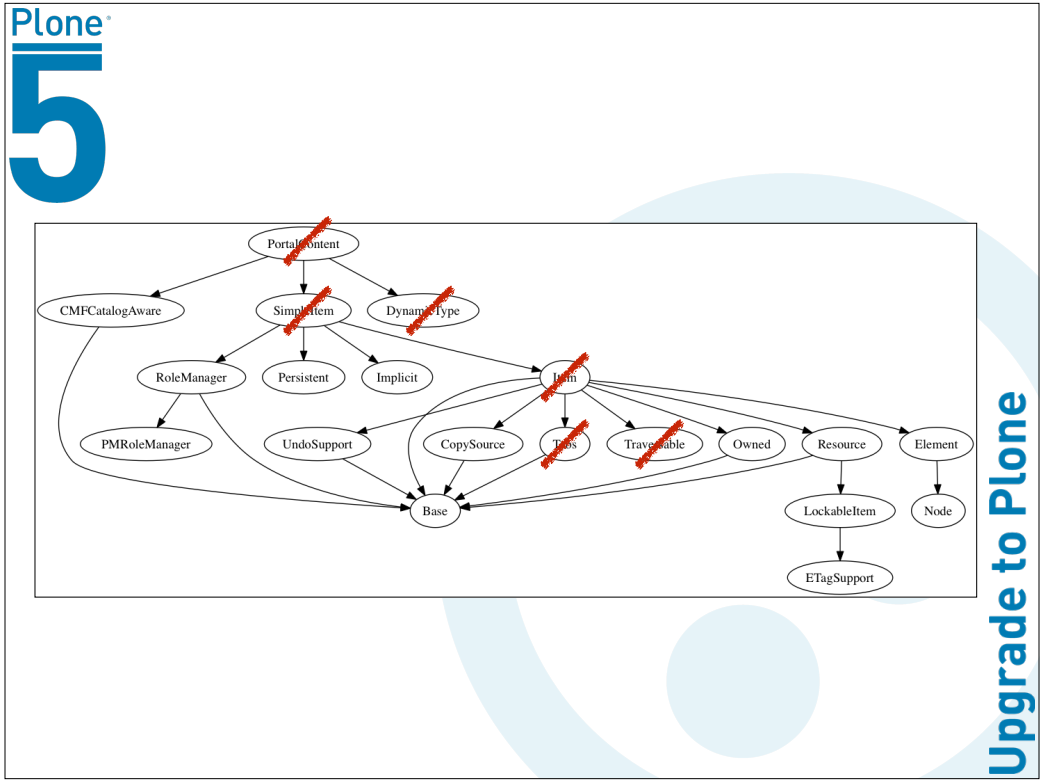
The same default view can be used for different types of content.

Or, we can have different adapters found for different types of content.



The result of this new approach is a profound change.

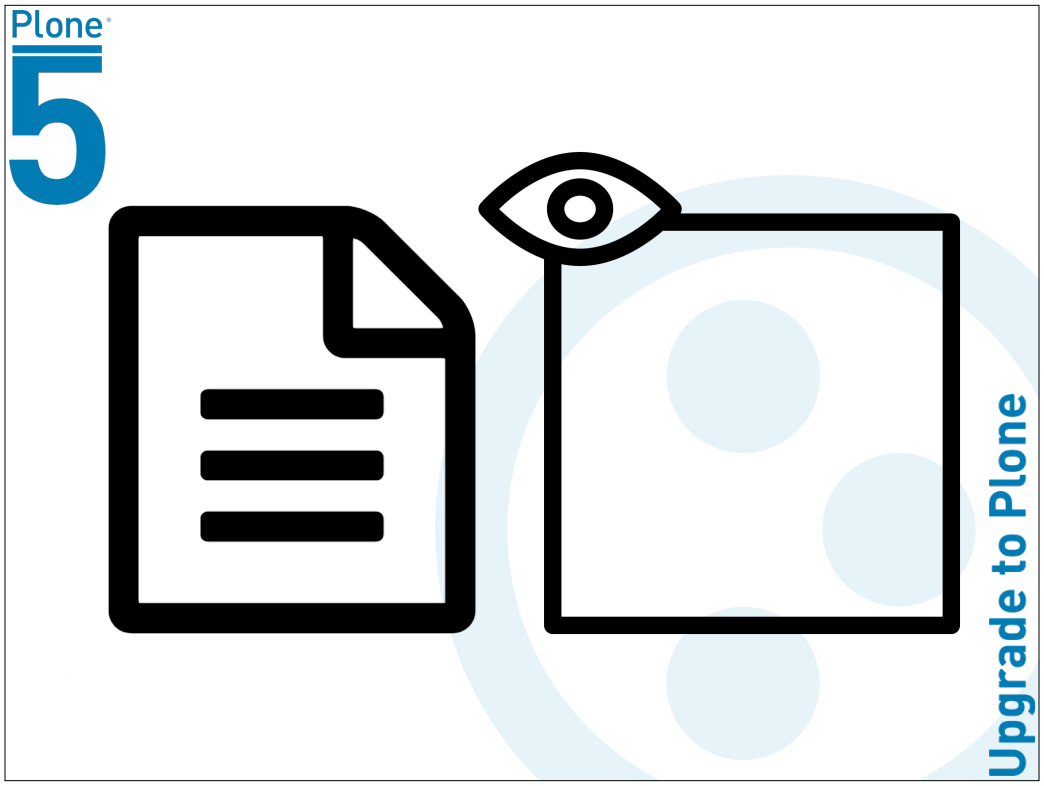
Publishing functionality that was needed by our content objects is now delegated to this adapter. Our objects get simpler



The result of this new approach is a profound change.

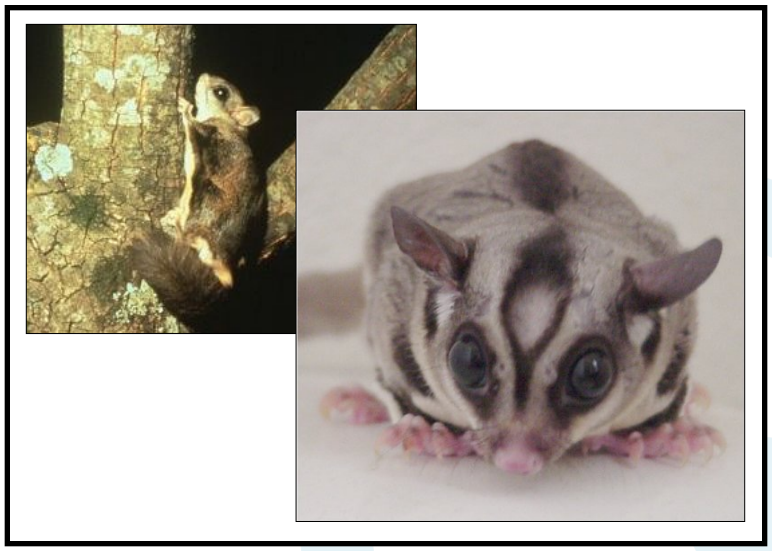
Publishing functionality that was needed by our content objects is now delegated to this adapter. Our objects get simpler

better yet, we can now think of content objects and our view of them separately.



better yet, we can now think of content objects and our view of them separately.

Plone 5



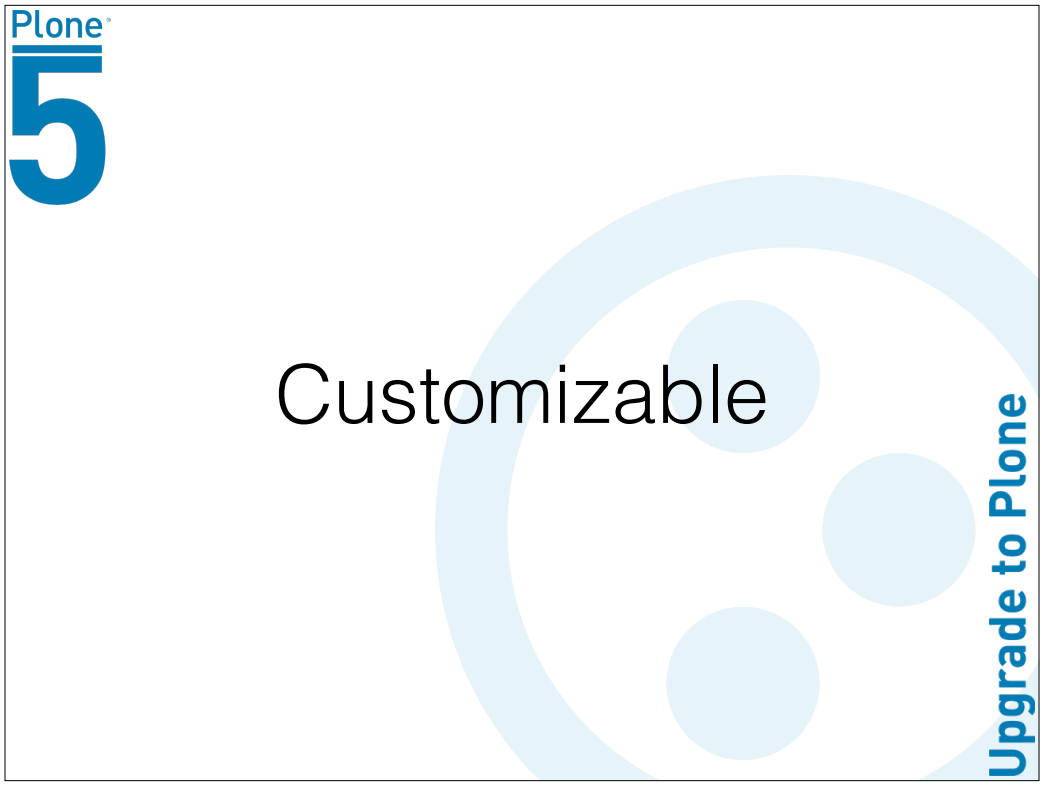
convergent adaptation

By FWS (U.S. Fish & Wildlife Service) [Public domain] via Wikimedia Commons
By Dawson at English Wikipedia (Own work) CC BY-SA 2.5 via Wikimedia Commons

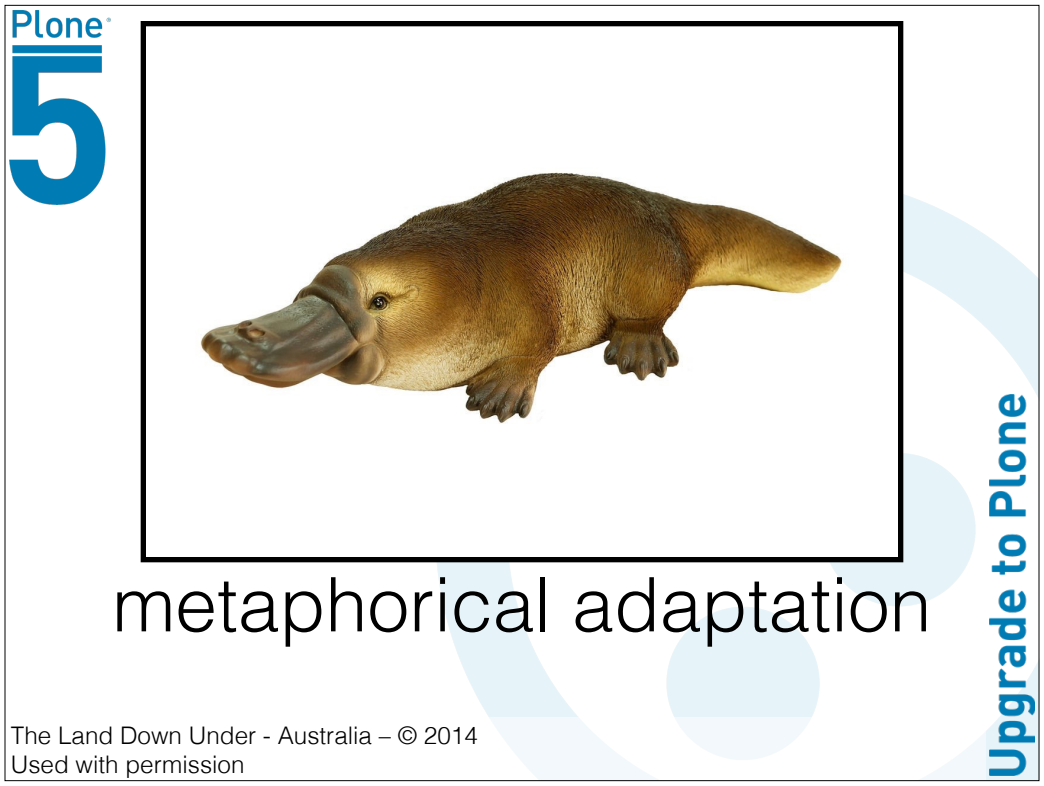
Upgrade to Plone

and we can treat any content object the same.

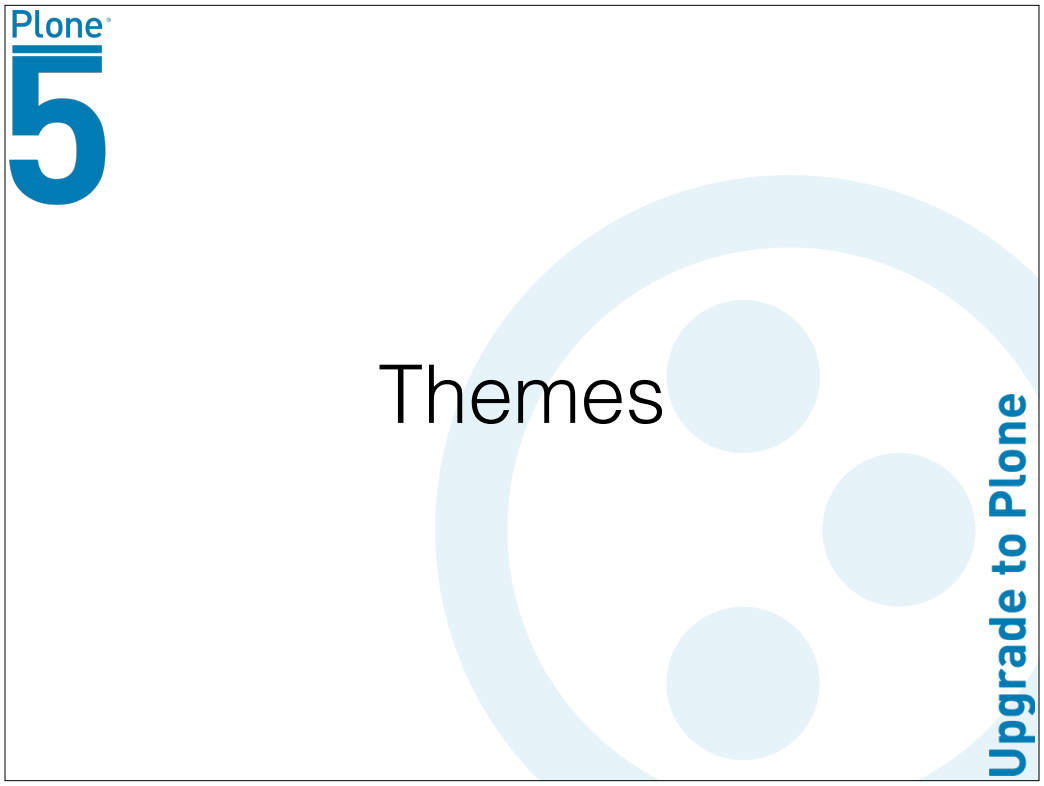
Adapters have made publishing objects simpler, and more easily customizable.



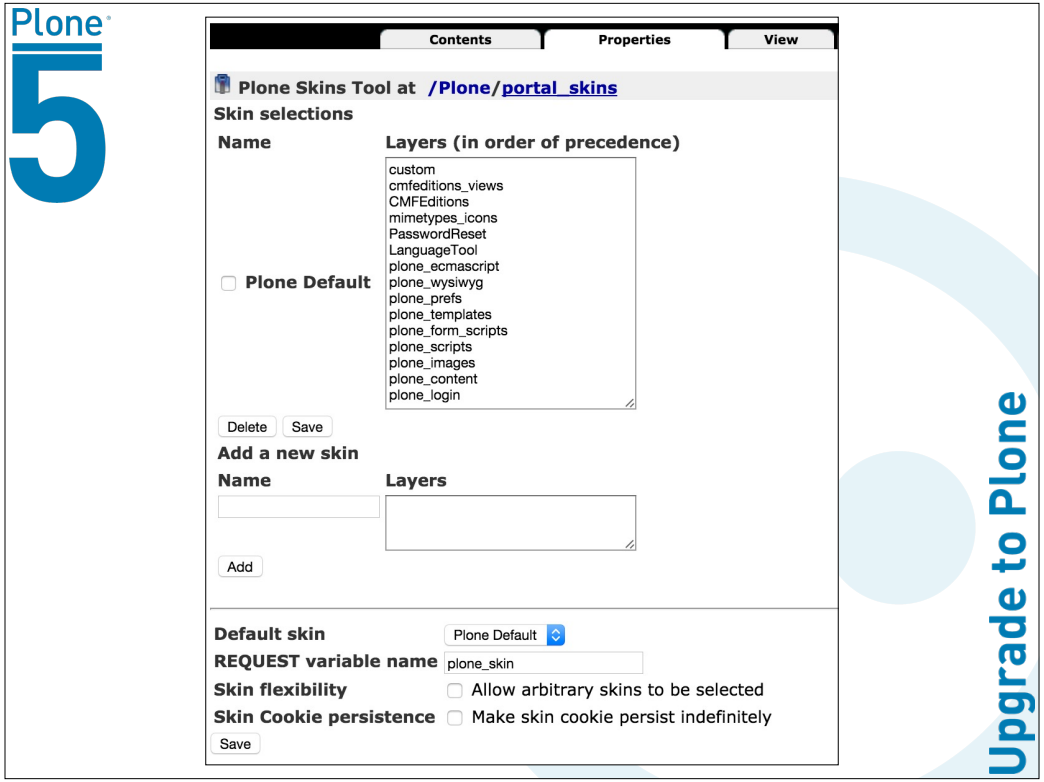
Adapters have made publishing objects simpler, and more easily customizable.



Moving beyond literal adaptation, I'd like to consider a more metaphorical version. Here, there's not really any adaptation going on at all. But the idea of adaptation helped to lead us to the solution to another thorny problem.

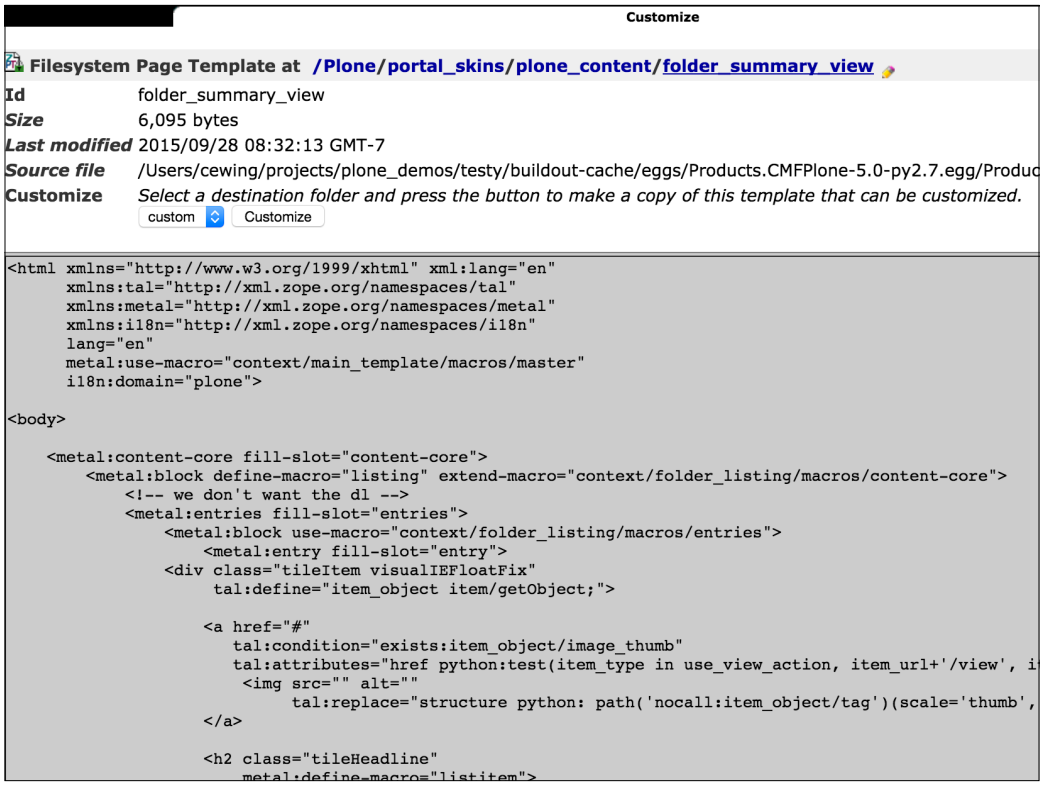


This time, the source of the pain was creating Themes for Plone.

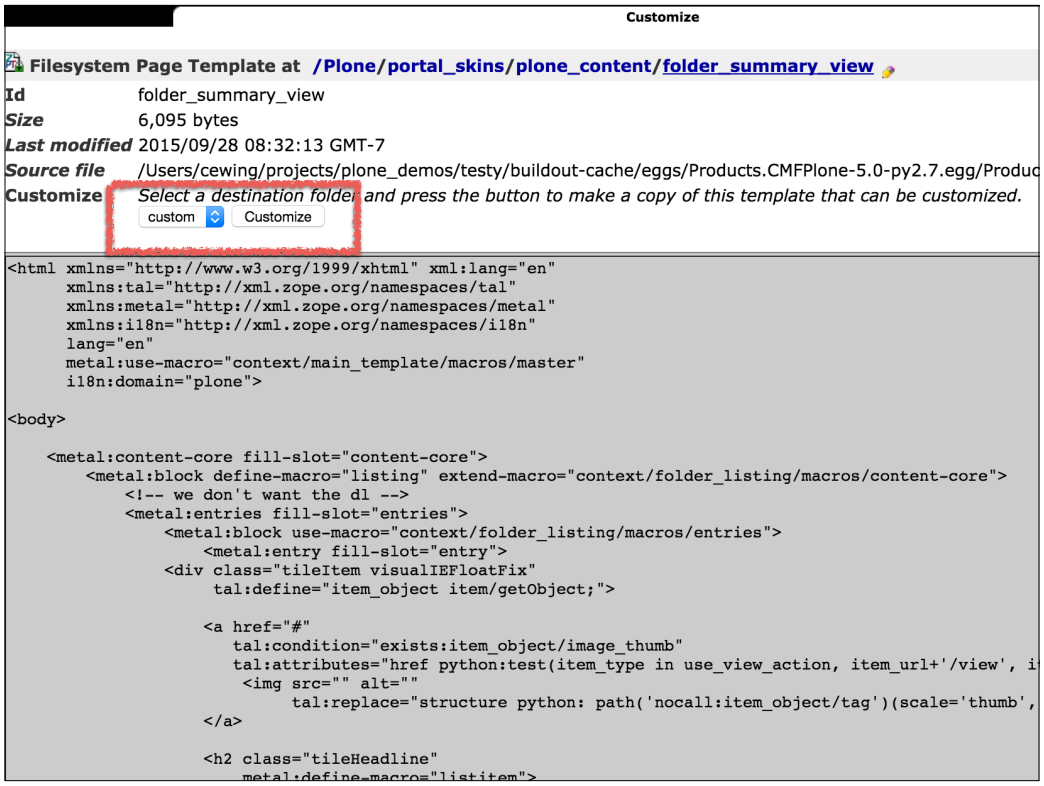


Remember that back in the beginning, the CMF gave us the idea of grouping an ordered series of folders together to make a “theme”. Each folder would contain page templates, javascript, css, images, whatever was needed to make the site look good.

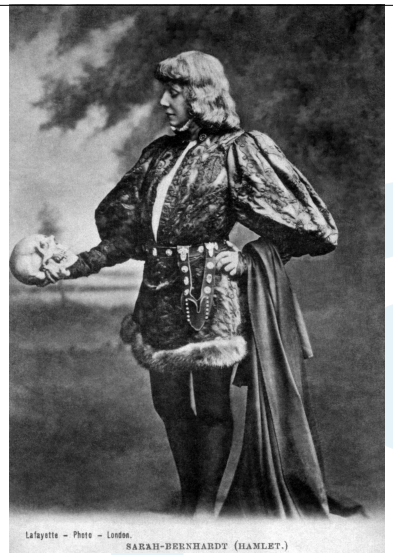
To customize these themes, you only needed to insert a new item with the same name as an existing item into a folder that was closer to the top of the pile.



This was easy to do through the web, because you could find the original items and click on this button here to make a copy in the “custom” folder (which was always the folder right at the top of the list). This copied version was stored in the database, and could be edited through the web.



This was easy to do through the web, because you could find the original items and click on this button here to make a copy in the “custom” folder (which was always the folder right at the top of the list). This copied version was stored in the database, and could be edited through the web.



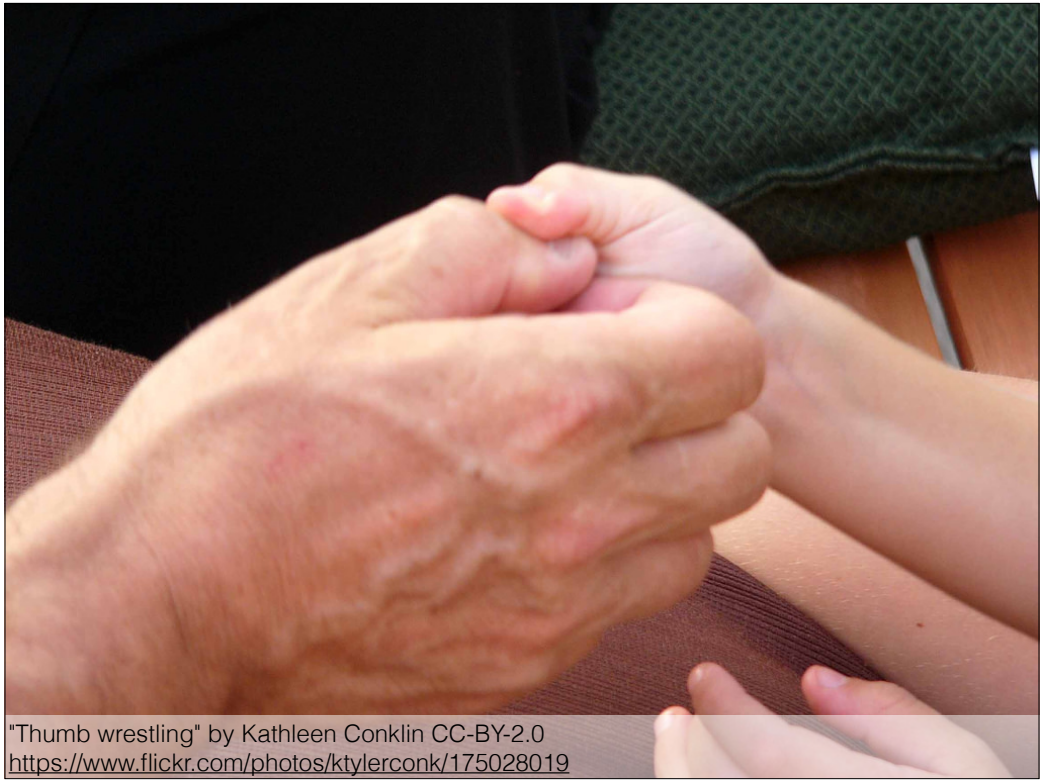
aye, there's the rub…

Sarah Bernhardt as Hamlet, with Yorick's skull
(Photographer: James Lafayette, c. 1885–1900)

Upgrade to Plone

Of course, page templates have all sorts of expectations built into them, about what values will be present in the context, about the names of available macros, and so on.

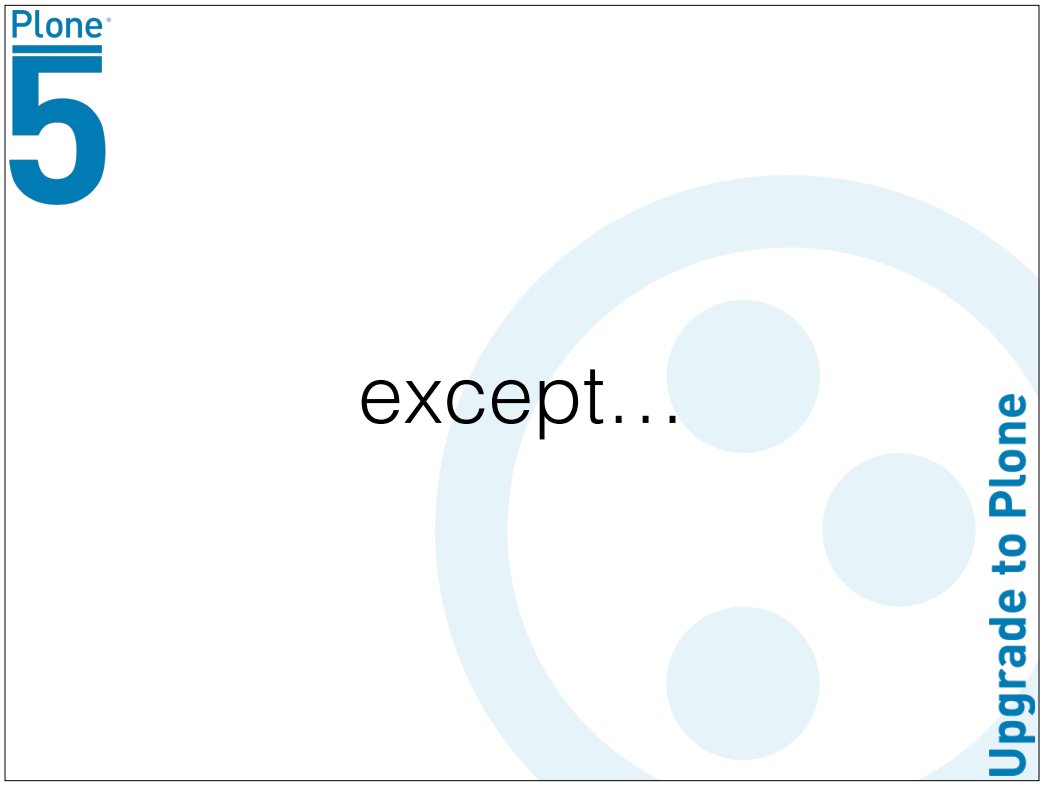
Here's a place where upgrades began to bite. If the expectations of page templates change between versions of Plone, then on upgrade, all those customized page templates you've created are now broken, and you can't tell until you've upgraded the site. Finding and fixing the problems is hard, slow and expensive. Not a happy situation.



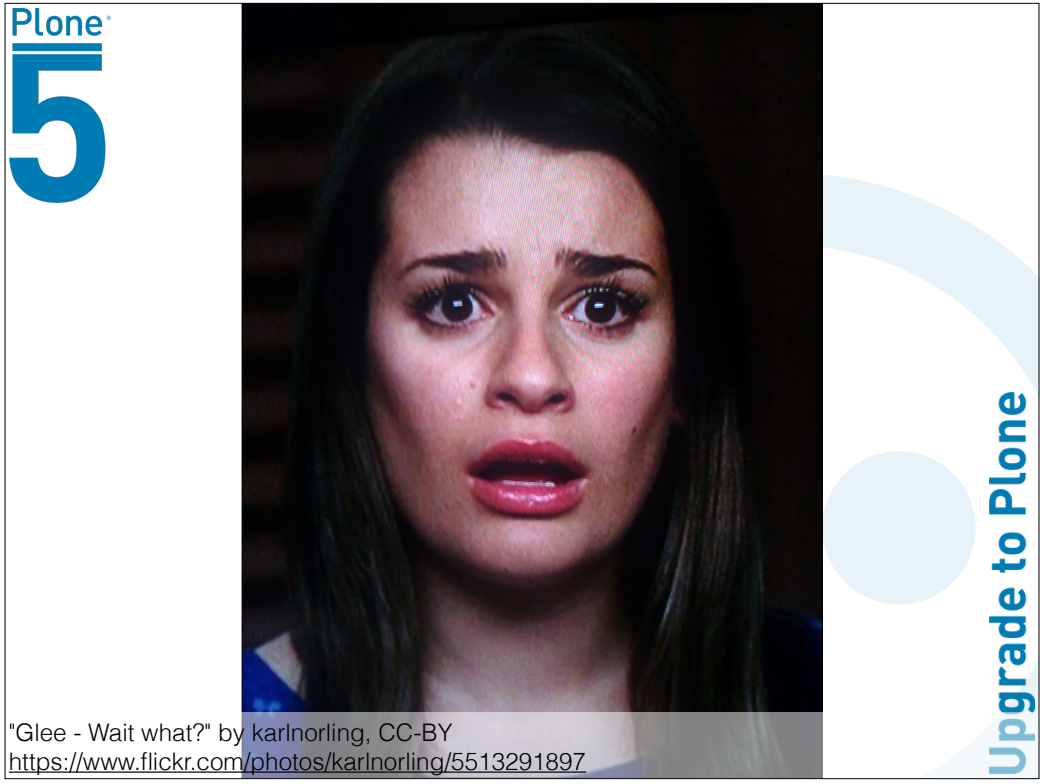
"Thumb wrestling" by Kathleen Conklin CC-BY-2.0
https://www.flickr.com/photos/ktylerconk/175028019

Another hazard is that this type of manual ordering is inherently brittle.
Even if you move your customization out to the filesystem, it's possible that some other plugin you add to your site later might override the same template in a different way, and install its theme folder higher on the stack than yours.

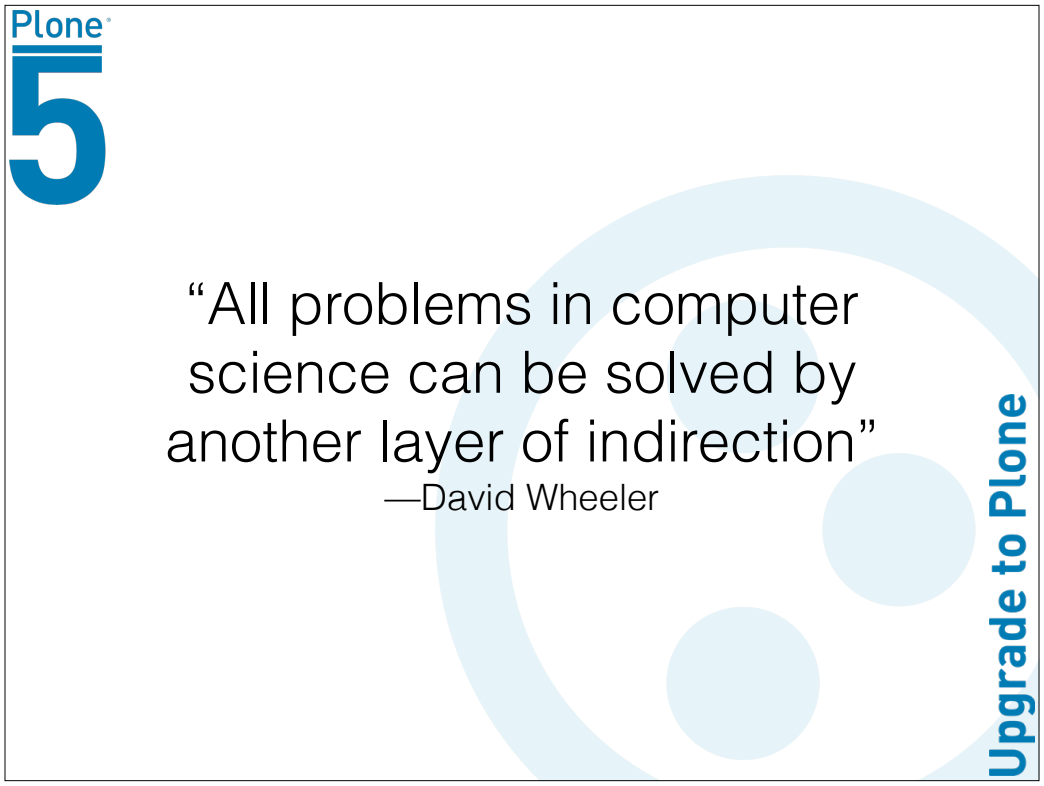
The rise of BrowserViews I described earlier helped to make things less hazardous. Page templates associated with browser views are fetched from the file system, not from them layer folders.



But to create browser views for simple theme customizations was tremendous overkill. To theme Plone you had to know HTML, JavaScript, CSS, Python, the Zope Page Template language, and XML.

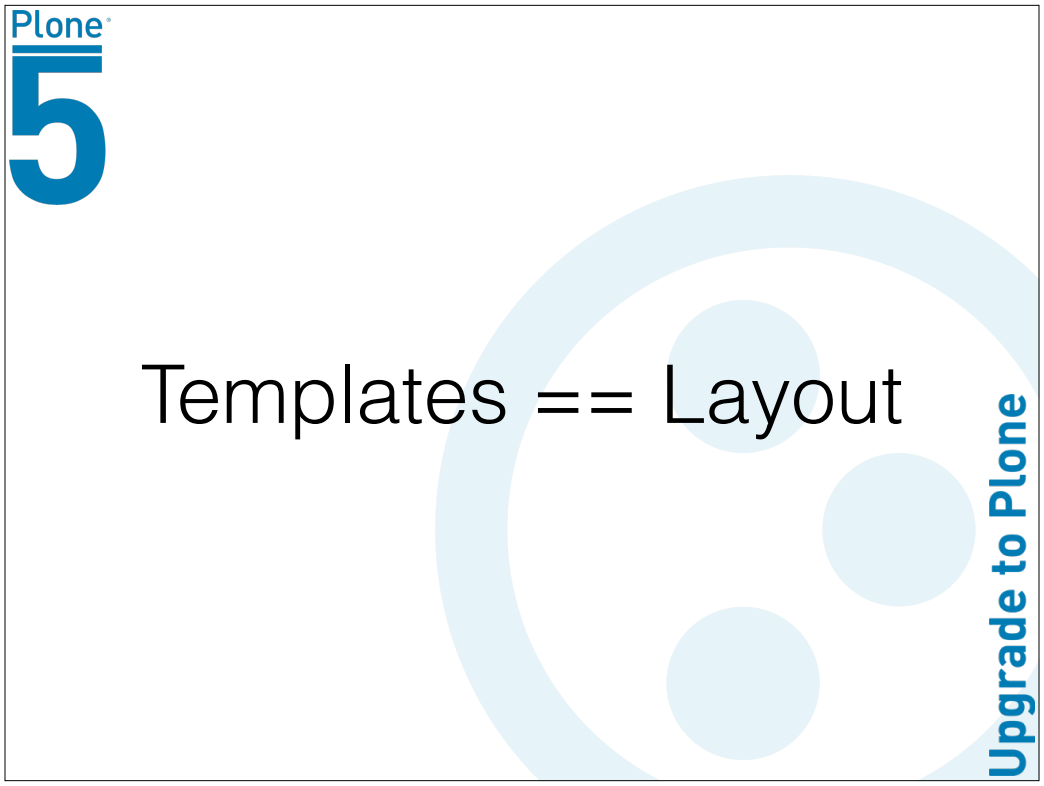


But to create browser views for simple theme customizations was tremendous overkill. To theme Plone you had to know HTML, JavaScript, CSS, Python, the Zope Page Template language, and XML.



So some folks in the community asked a pretty simple question:

“Why do I have to override page templates at all?”

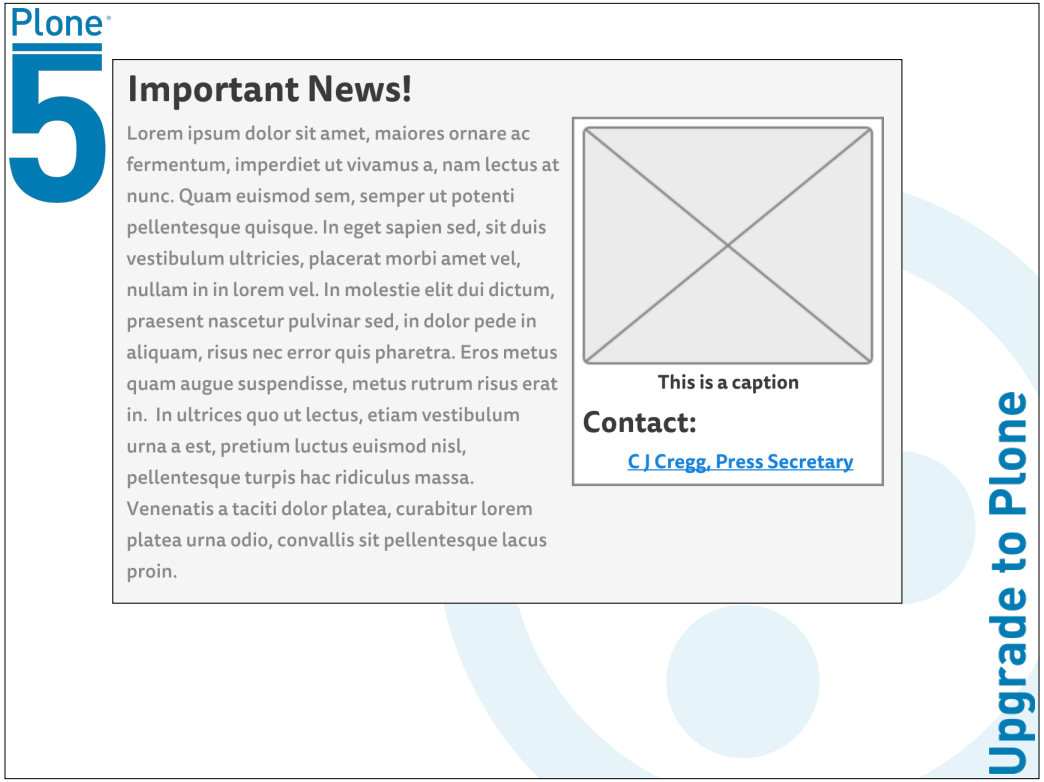


Page templates provide specific layout for the visible data of a site.
Generally in the web world, we've overridden templates in order to provide different layout.

Consider a “news item” with a title, some body text, a lead image and a contact.

the default layout for a news item might look like this:

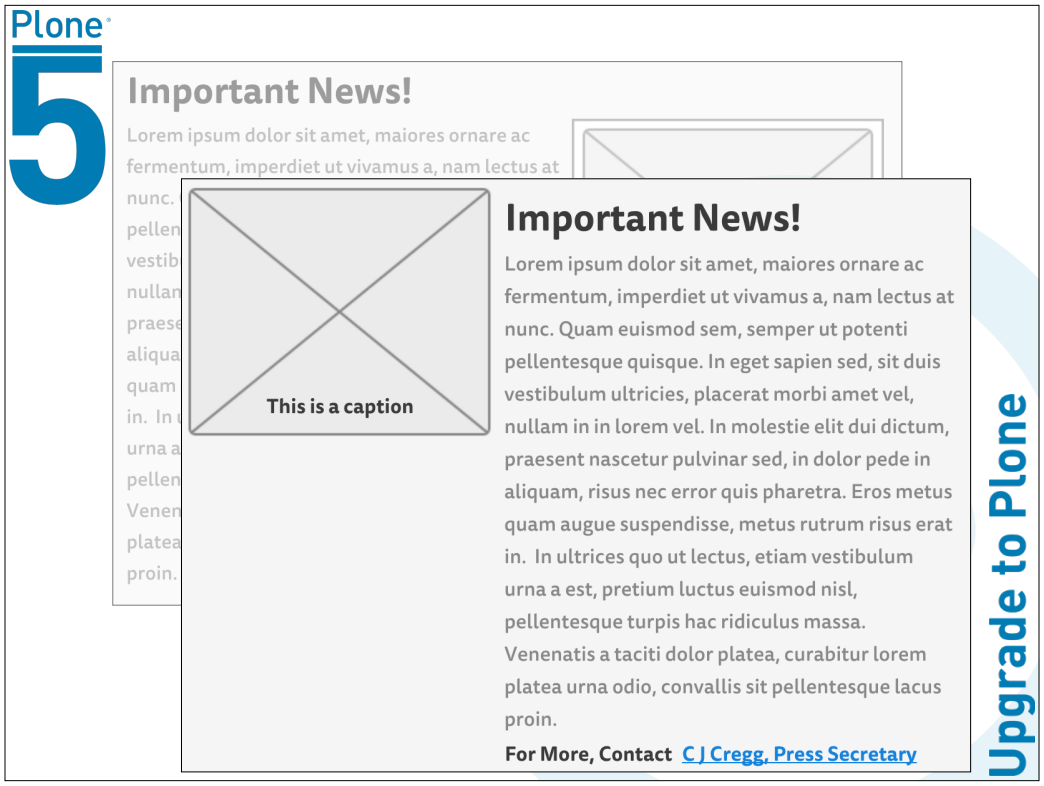
Maybe you want it to look like this, instead:



Consider a “news item” with a title, some body text, a lead image and a contact.

the default layout for a news item might look like this:

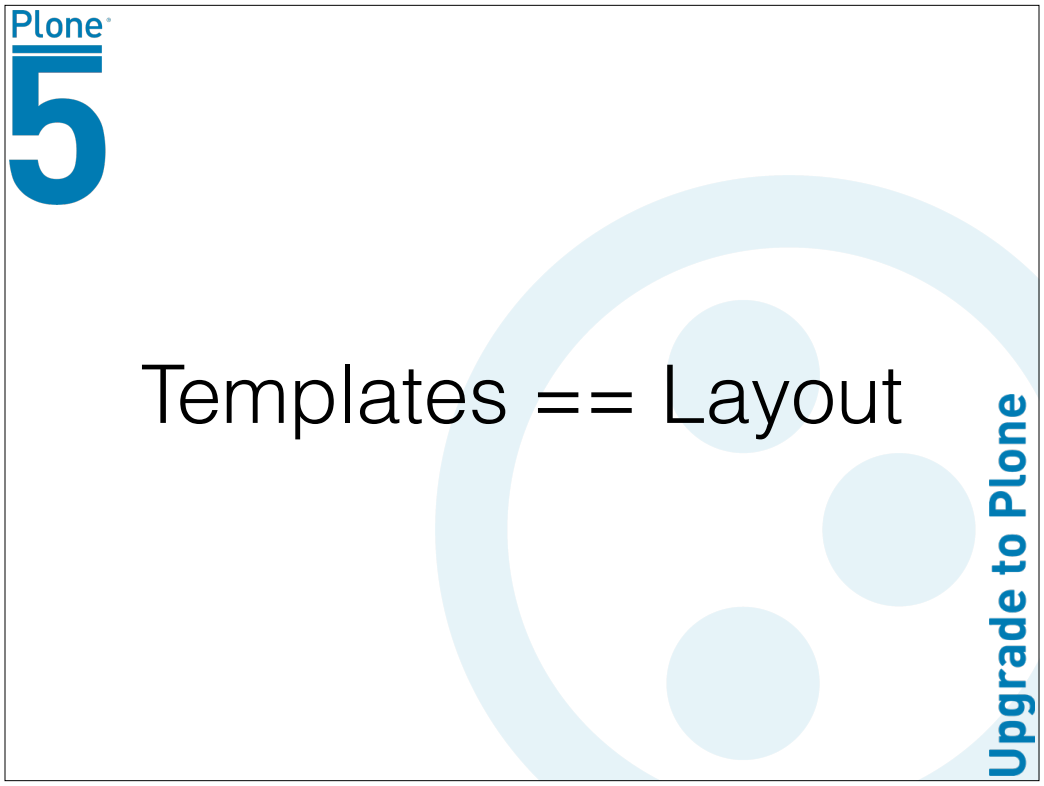
Maybe you want it to look like this, instead:



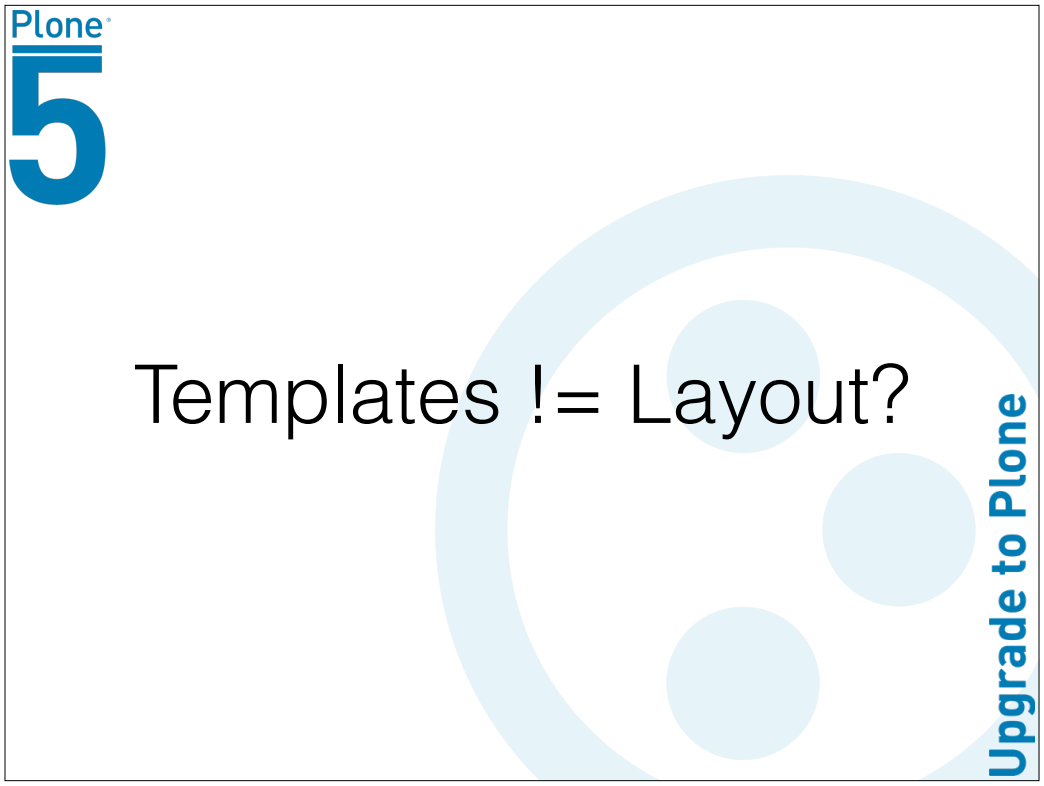
Consider a “news item” with a title, some body text, a lead image and a contact.

the default layout for a news item might look like this:

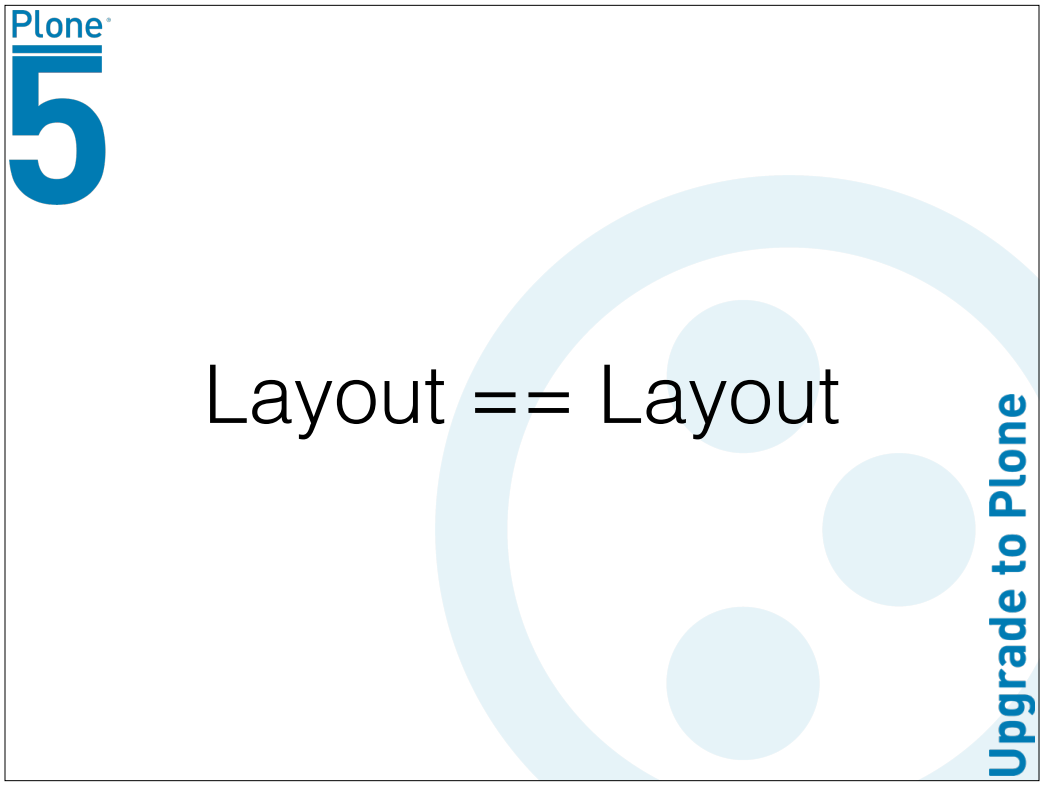
Maybe you want it to look like this, instead:



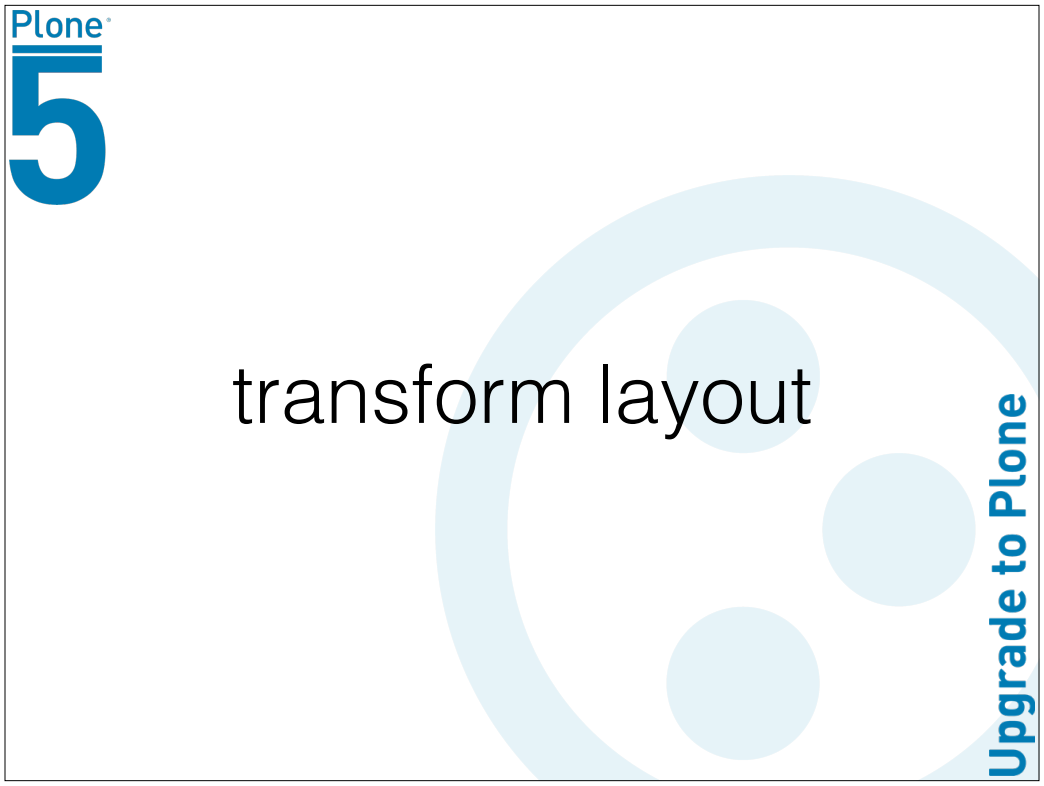
To do so, you override the old template with the new one.



But is this really the only way things can be?



maybe layout is just… layout?



What if there were a way to “adapt” a layout, to magically transform one layout into another?

Well, there is. The technology has been around for years…

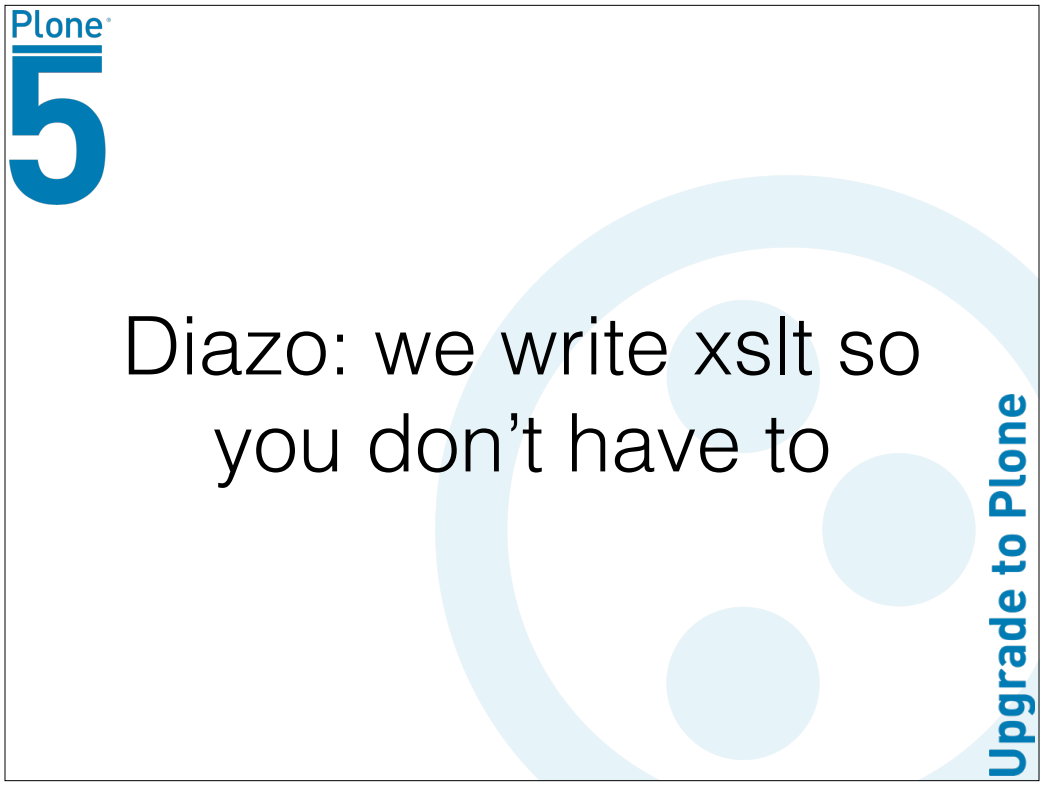
Plone 5

xslt

XSLT (Extensible Stylesheet Language Transformations)

a language for transforming XML documents into other XML documents

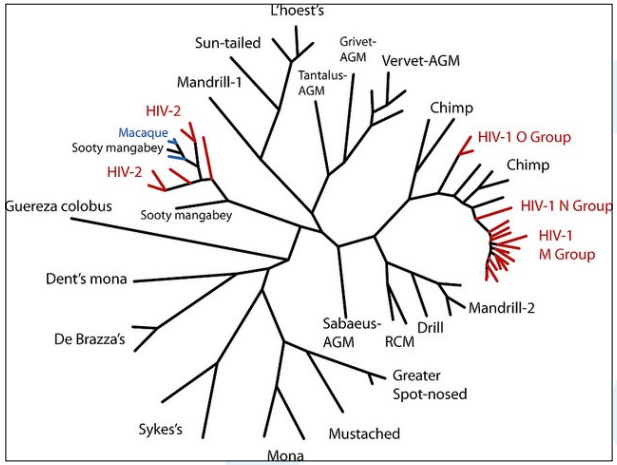
Upgrade to Plone



Thus was born “Diazo”

PAUSE

Plone 5 Diazo



Adaptation

"Phylogeny Figures" by pat.0020015.g001, CC-BY
https://www.flickr.com/photos/123636286@N02/14138677365

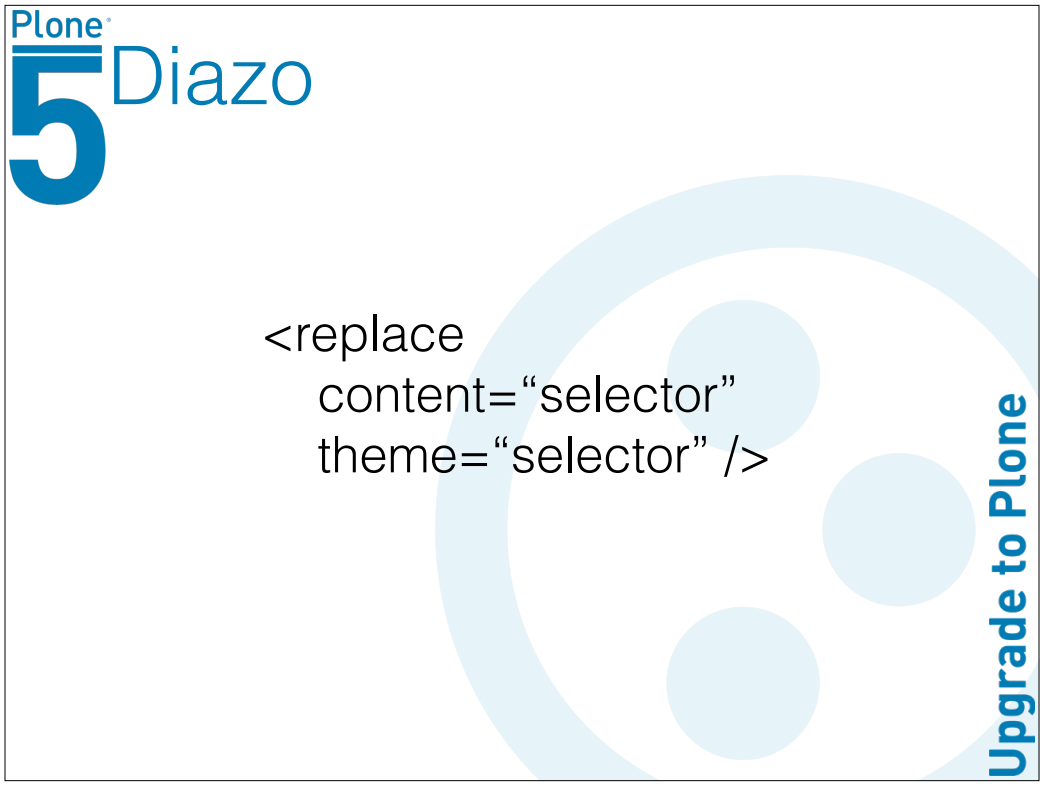
Upgrade to Plone

Diazo started life as two separate packages: Deliverance and XDV. The codebase for xdv eventually won out and became diazo, but some of the ideas of Deliverance were adopted as well.

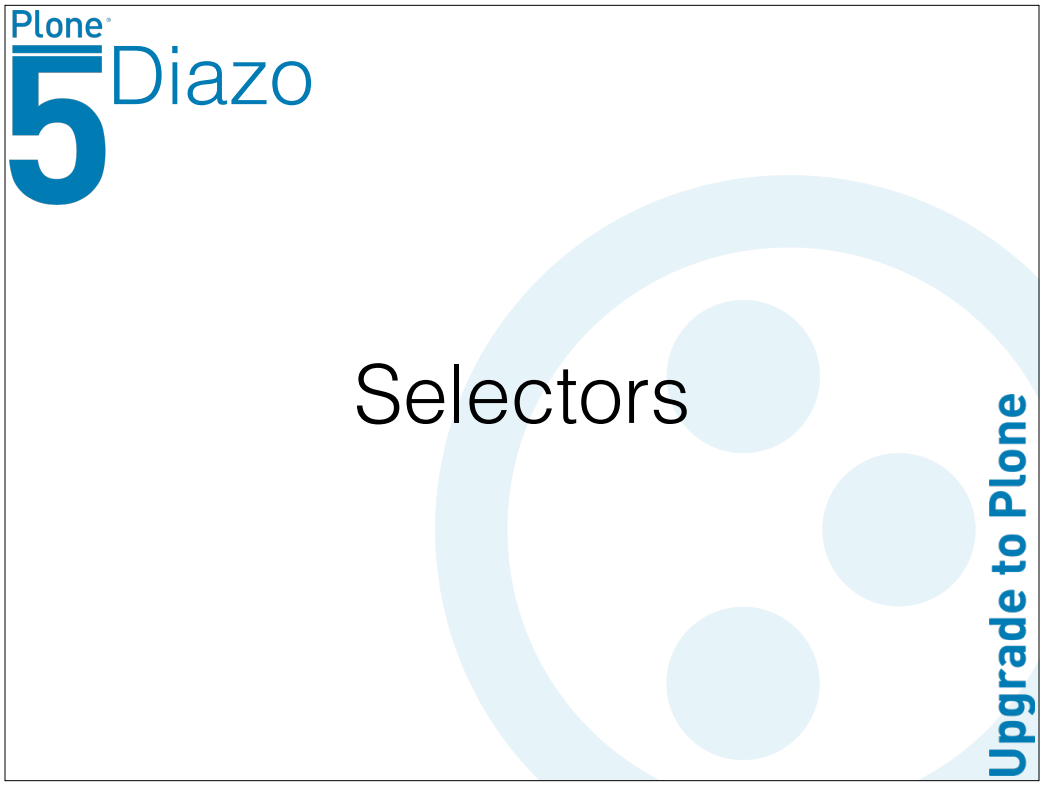
Diazo works with a “theme” and the “content”.

It consists of a 8 rules.

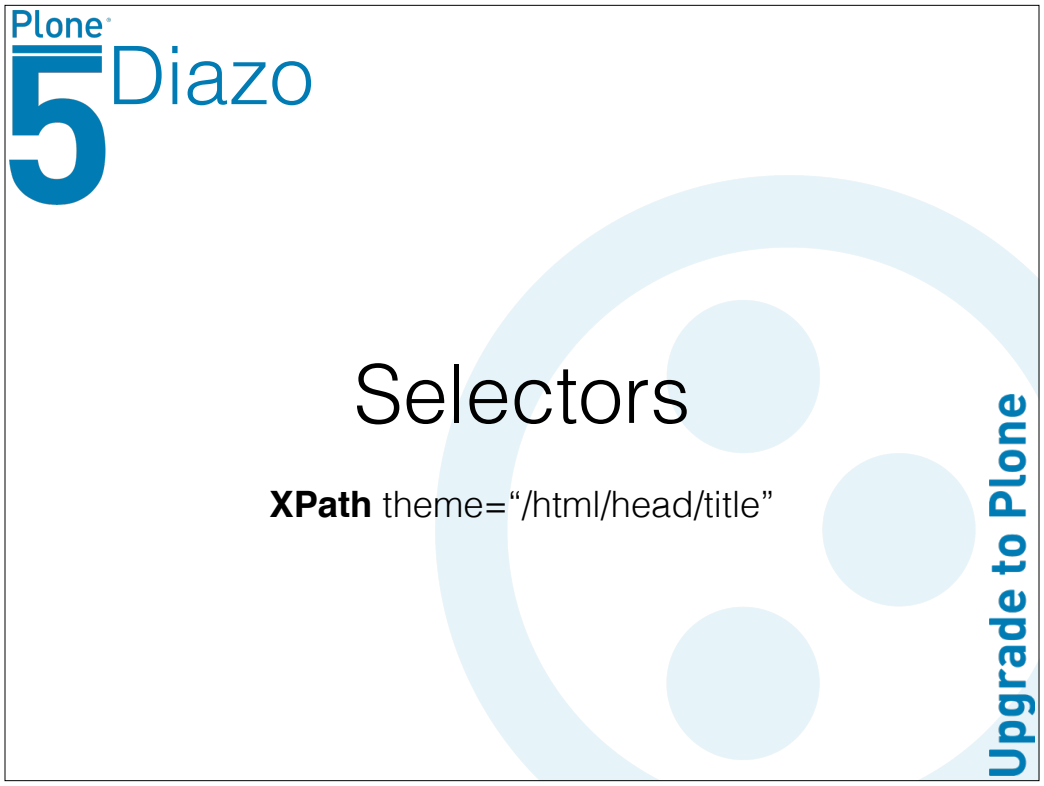
the first rule is the “theme” rule, which indicates a plain html page to be used as the “theme” layout.



All the other rules use “selectors” to find HTML elements in either the target theme or the source content.



The selectors used by diazo to find elements in the theme or in the content can be either xpath selectors or css selectors like those used by jQuery and most other common front-end libraries.

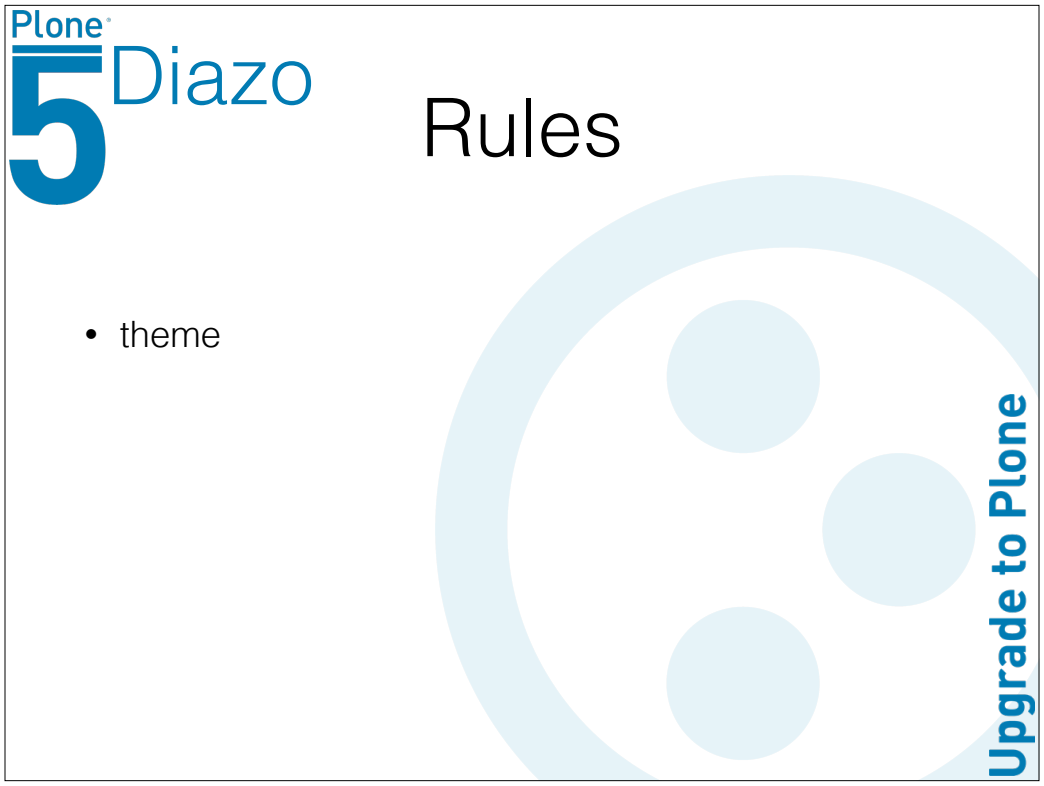


The selectors used by diazo to find elements in the theme or in the content can be either xpath selectors or css selectors like those used by jQuery and most other common front-end libraries.

The selectors used by diazo to find elements in the theme or in the content can be either xpath selectors or css selectors like those used by jQuery and most other common front-end libraries.

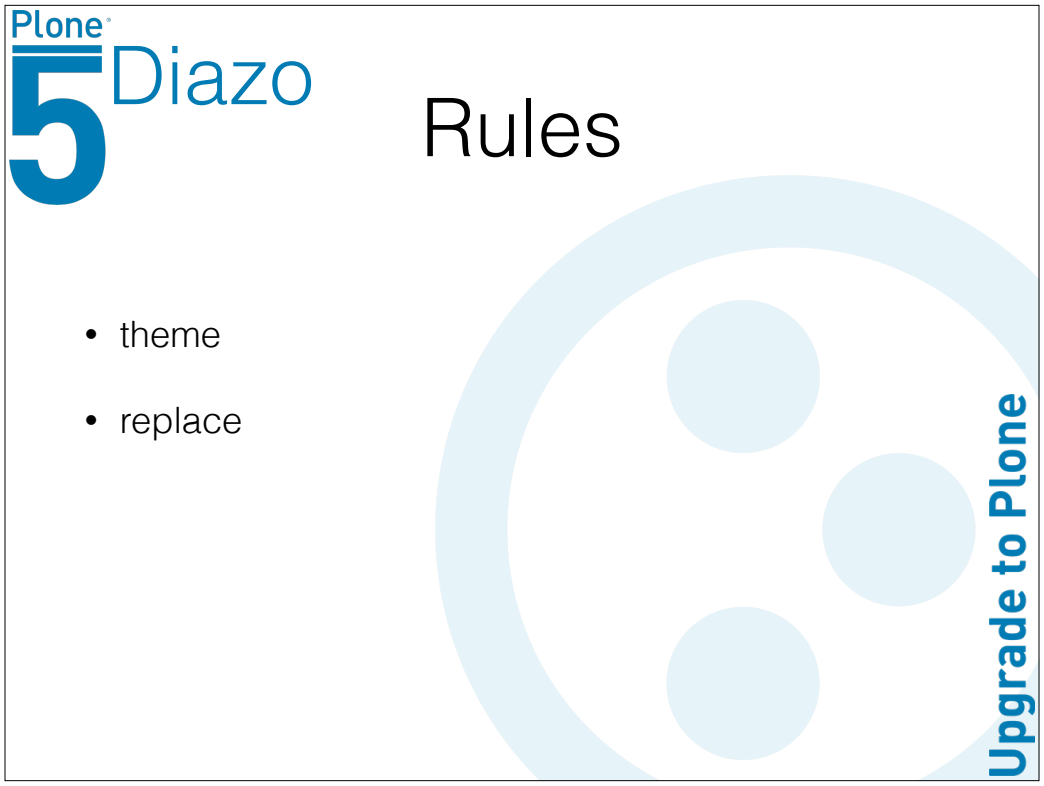
In addition to the theme rule, there are rules to

* replace a theme element with one from the content
* insert a content element before or after a theme element
* drop an element (and it's children) from the theme or the content
* strip an element from the theme or content, leaving its children
* merge attributes from a content element to the same attributes on the theme element
* and copy attributes from a content element onto the theme element.



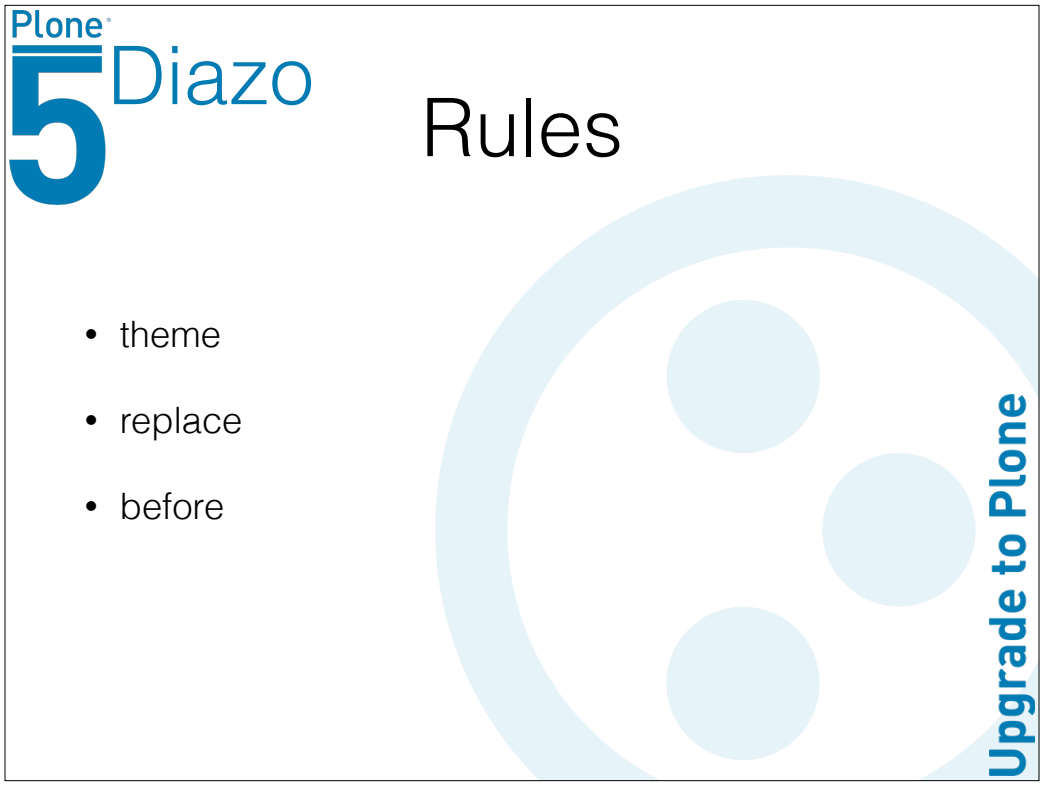
In addition to the theme rule, there are rules to

* replace a theme element with one from the content
* insert a content element before or after a theme element
* drop an element (and it's children) from the theme or the content
* strip an element from the theme or content, leaving its children
* merge attributes from a content element to the same attributes on the theme element
* and copy attributes from a content element onto the theme element.



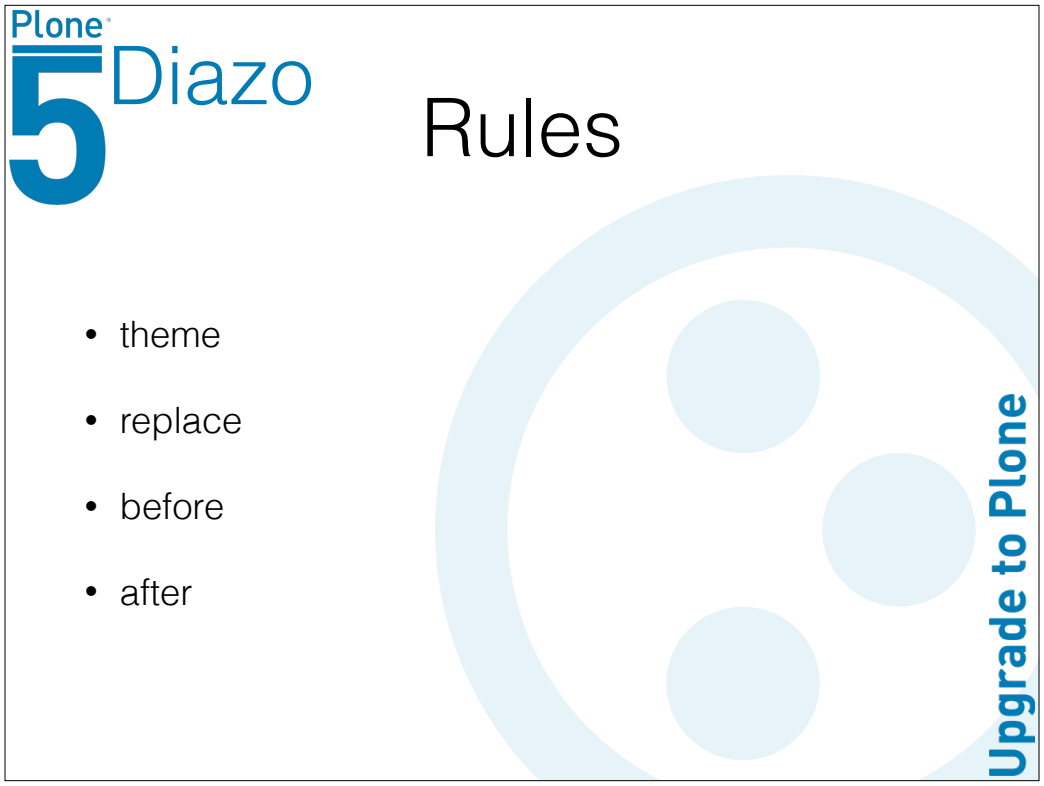
In addition to the theme rule, there are rules to

* replace a theme element with one from the content
* insert a content element before or after a theme element
* drop an element (and it's children) from the theme or the content
* strip an element from the theme or content, leaving its children
* merge attributes from a content element to the same attributes on the theme element
* and copy attributes from a content element onto the theme element.



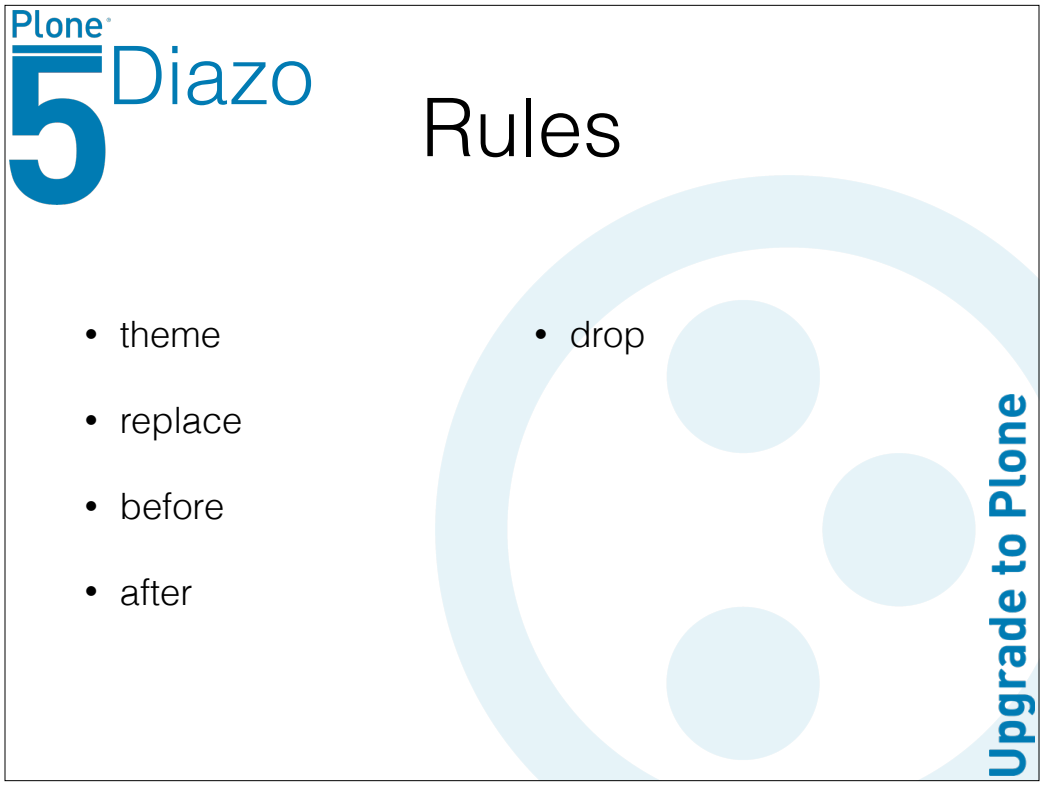
In addition to the theme rule, there are rules to

* replace a theme element with one from the content
* insert a content element before or after a theme element
* drop an element (and it's children) from the theme or the content
* strip an element from the theme or content, leaving its children
* merge attributes from a content element to the same attributes on the theme element
* and copy attributes from a content element onto the theme element.



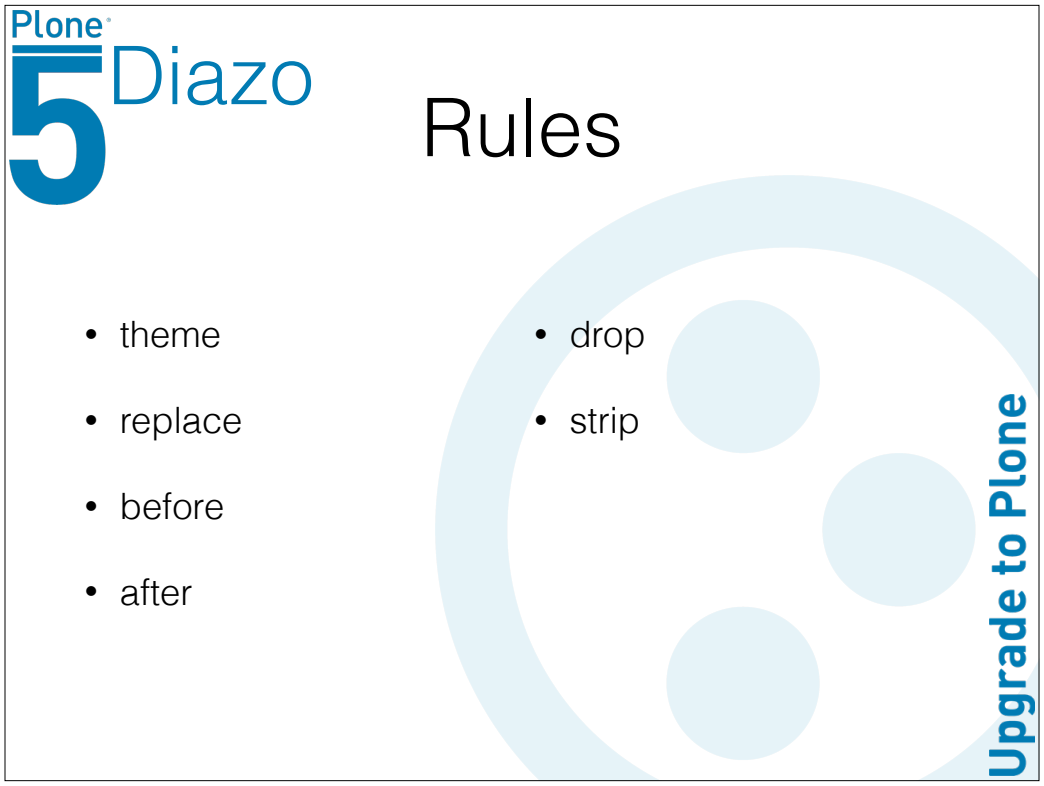
In addition to the theme rule, there are rules to

* replace a theme element with one from the content
* insert a content element before or after a theme element
* drop an element (and it's children) from the theme or the content
* strip an element from the theme or content, leaving its children
* merge attributes from a content element to the same attributes on the theme element
* and copy attributes from a content element onto the theme element.



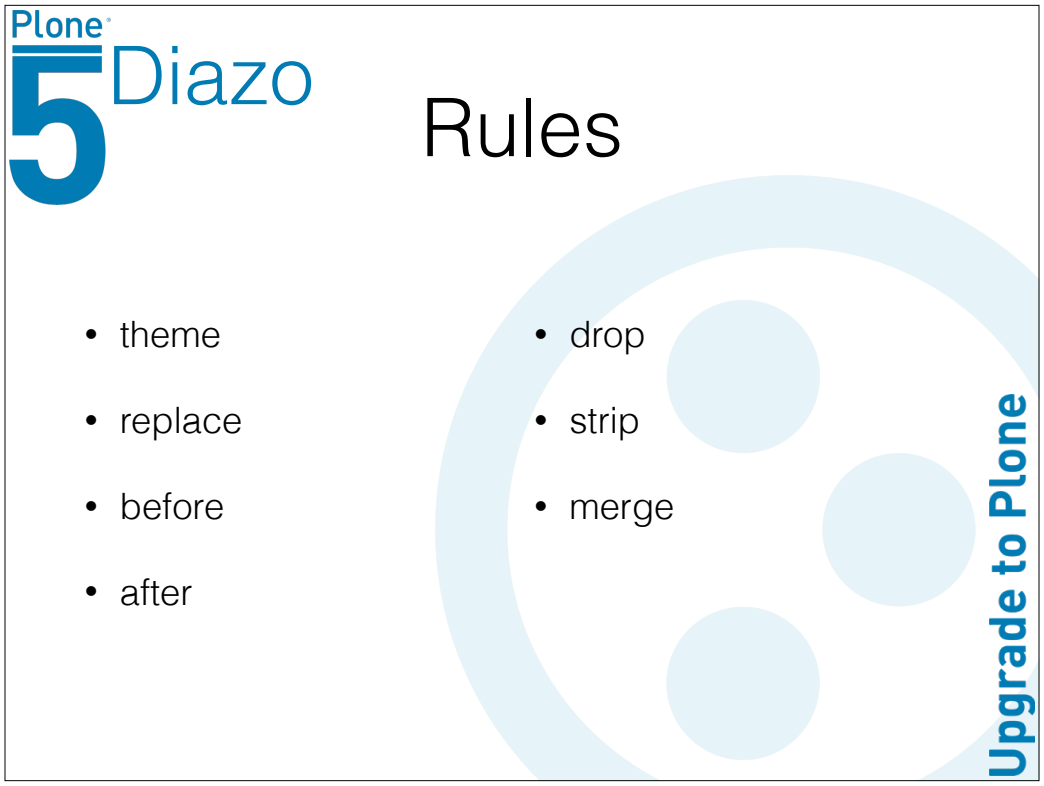
In addition to the theme rule, there are rules to

* replace a theme element with one from the content
* insert a content element before or after a theme element
* drop an element (and it's children) from the theme or the content
* strip an element from the theme or content, leaving its children
* merge attributes from a content element to the same attributes on the theme element
* and copy attributes from a content element onto the theme element.



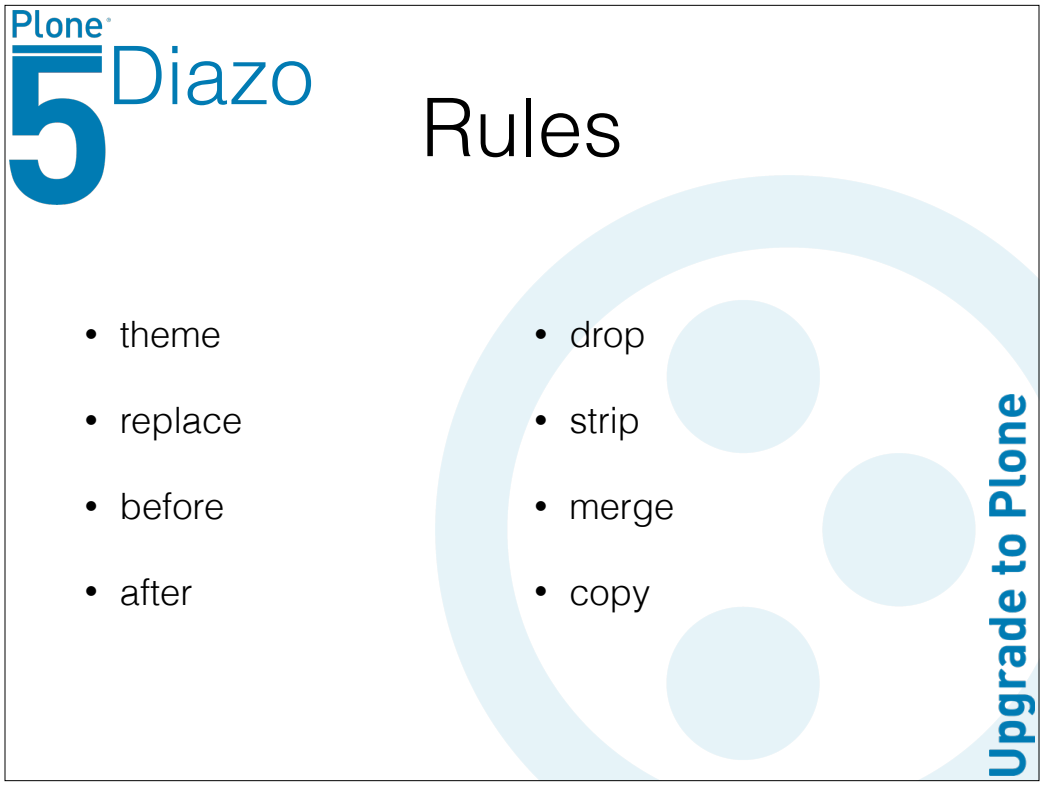
In addition to the theme rule, there are rules to

* replace a theme element with one from the content
* insert a content element before or after a theme element
* drop an element (and it's children) from the theme or the content
* strip an element from the theme or content, leaving its children
* merge attributes from a content element to the same attributes on the theme element
* and copy attributes from a content element onto the theme element.



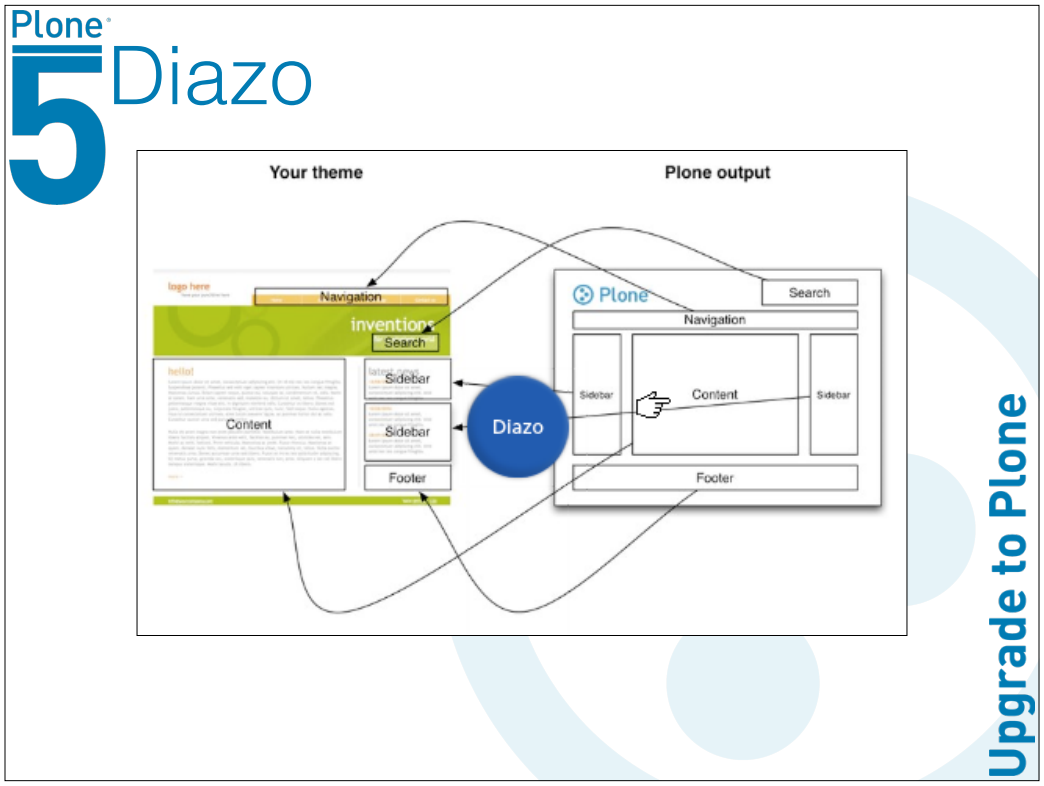
In addition to the theme rule, there are rules to

* replace a theme element with one from the content
* insert a content element before or after a theme element
* drop an element (and it's children) from the theme or the content
* strip an element from the theme or content, leaving its children
* merge attributes from a content element to the same attributes on the theme element
* and copy attributes from a content element onto the theme element.

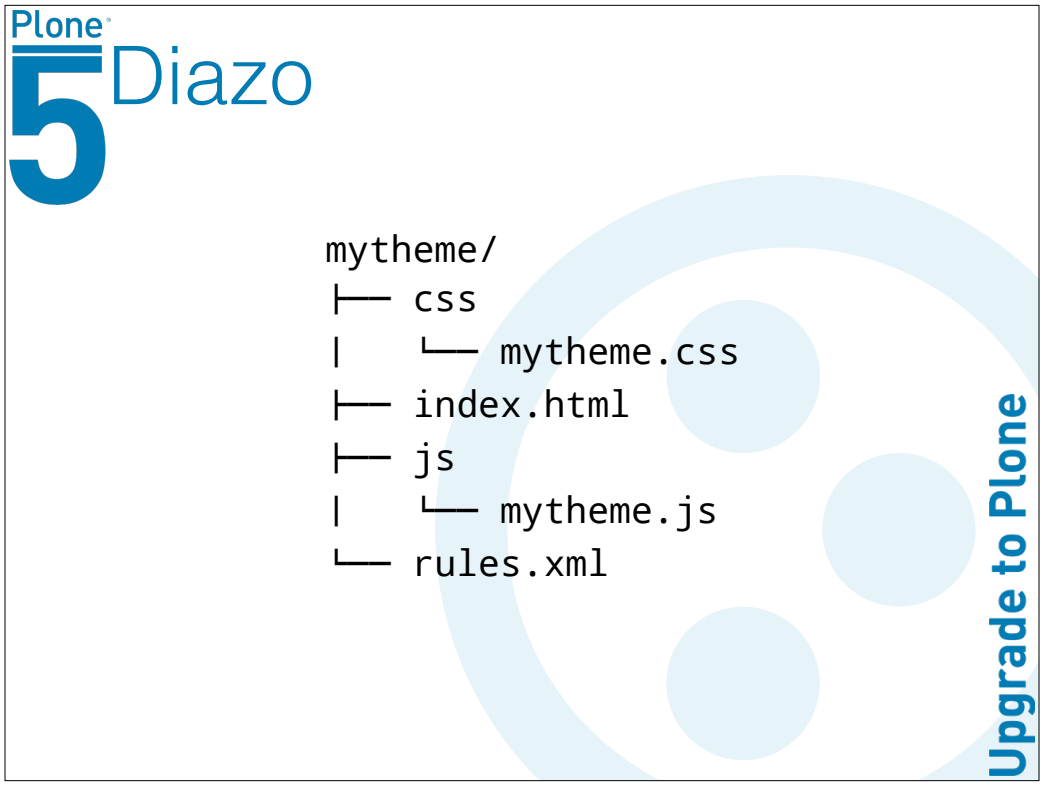


In addition to the theme rule, there are rules to

* replace a theme element with one from the content
* insert a content element before or after a theme element
* drop an element (and it's children) from the theme or the content
* strip an element from the theme or content, leaving its children
* merge attributes from a content element to the same attributes on the theme element
* and copy attributes from a content element onto the theme element.



Diazo takes the rules you write and precompiles them to an XSLT file which transforms the HTML produced directly by Plone into the HTML of your theme.



A theme for Plone 5, then, consists of one or more purely HTML layouts, the CSS and javascript needed to make them work, and a rules file that determines where the content that Plone produces will be placed in the layout(s) provided by the theme.

If you have a designer you work with, they can use their own tools to build a theme for you. If they prefer Grunt and sass, great. If they like yeoman, wonderful. The latest javascript and css tooling is no problem to adapt to because all of this happens outside of Plone.

You add a rules file to whatever they give you, zip it up and upload it to your website and you are off to the races.

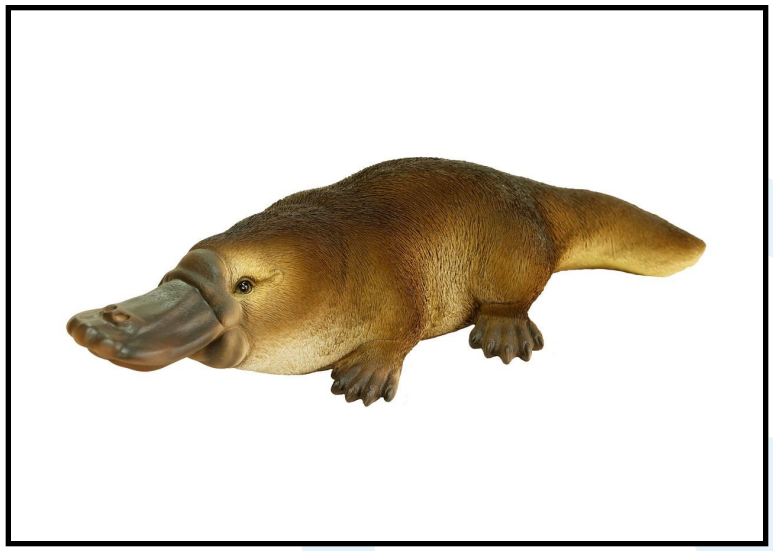
I know, I know. It's okay. Diazo rules use XML syntax, and if you use Diazo with other web applications (either as a plugin to your web server or as a WSGI middleware layer), that's how you must write them, but with Plone you get MORE.

A Visual Editor built right into your plone site.


Plone
5
Diazo
Theming Demo
Upgrade to Plone

As you can see, Diazo helps make theming Plone 5 a snap.

Plone 5



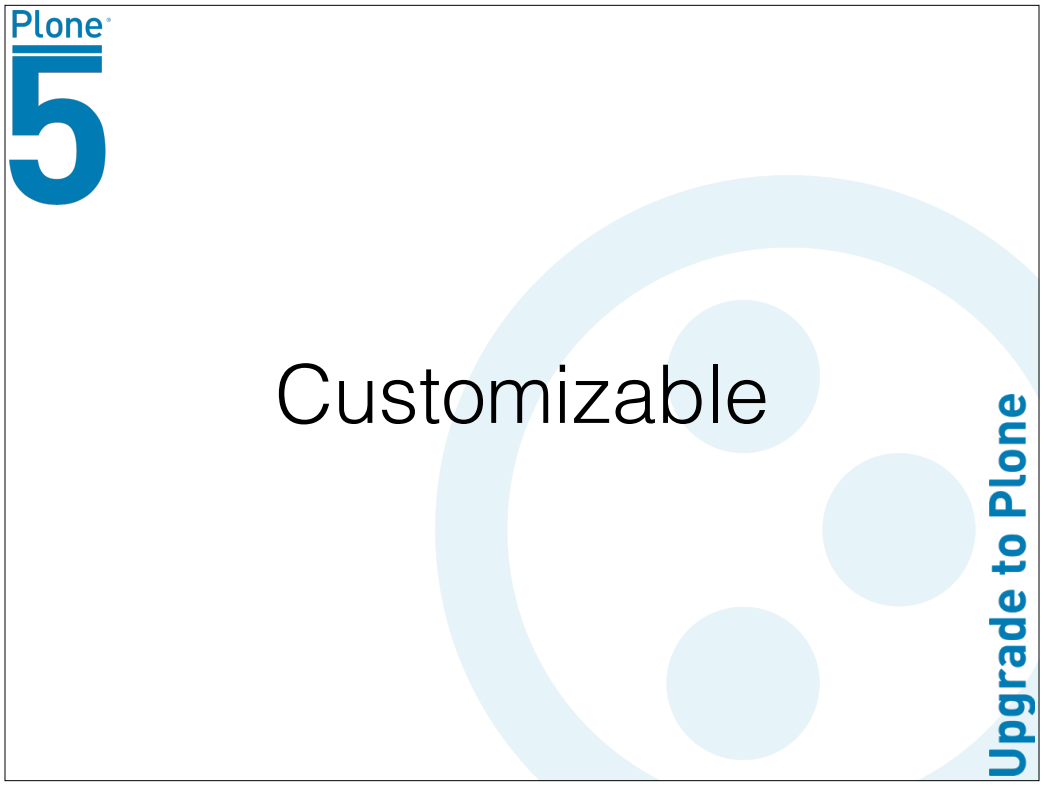
metaphorical adaptation

The Land Down Under - Australia – © 2014
Used with permission

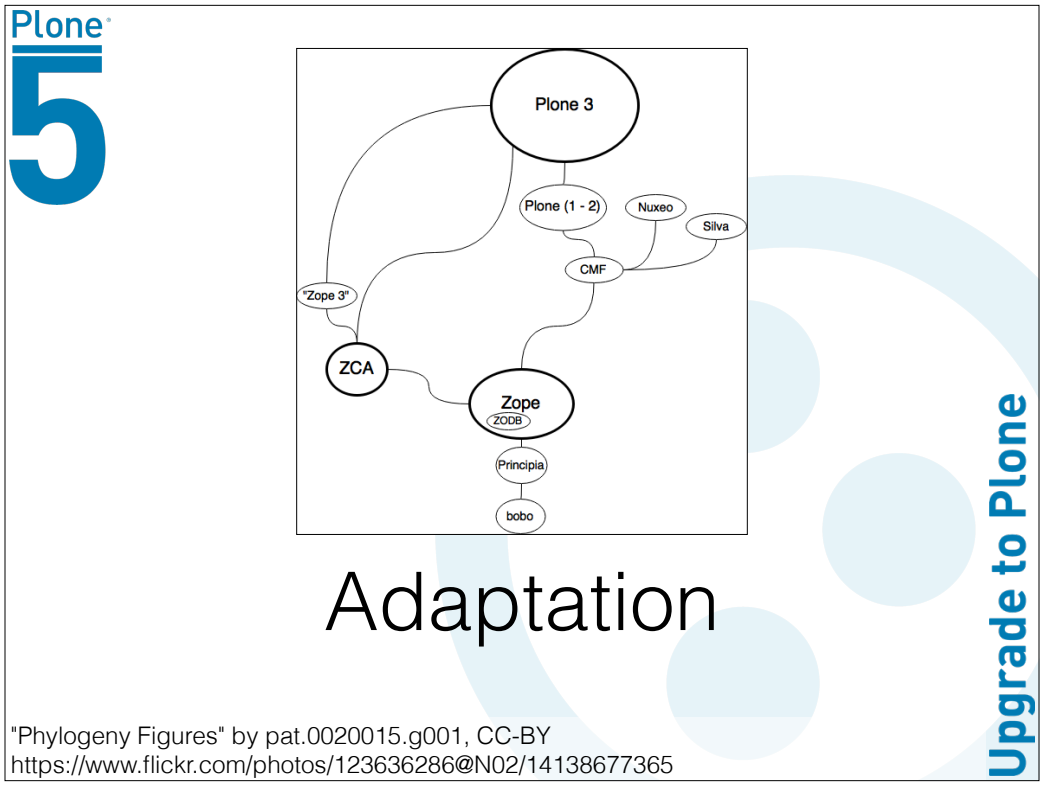
Upgrade to Plone

The idea of adaptation helped lead Plone developers to this new idea.

And it’s helped to improve the story of customizing Plone themes.



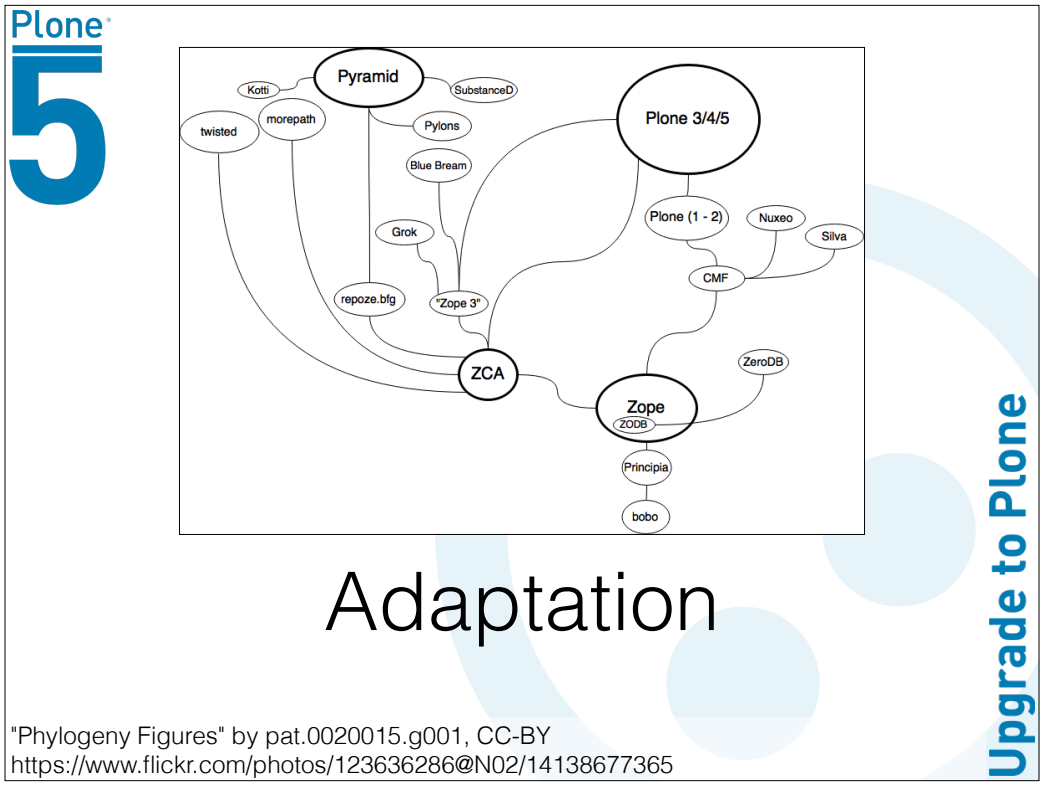
And it’s helped to improve the story of customizing Plone themes.



Adaptation has proven key in overcoming many of the challenges Plone has faced over time. And we inherited this idea from the work that Zope did on the Zope Component Architecture.

And other systems have taken these ideas even farther. Pyramid has used the power of the ZCA to build a phenomenally customizable system that's remarkably easy to work with.

Adaptation is not limited to web systems, you can use these powerful concepts in your software as well. The approach allows you to adapt fluidly to changing needs.



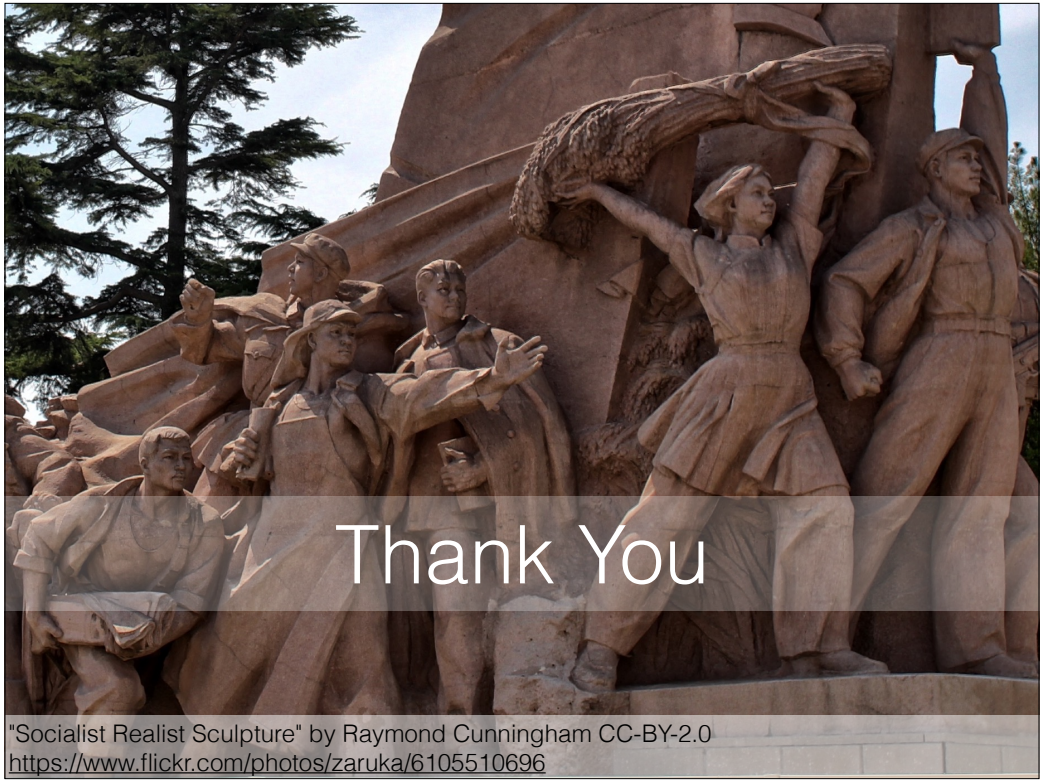
Adaptation has proven key in overcoming many of the challenges Plone has faced over time. And we inherited this idea from the work that Zope did on the Zope Component Architecture.

And other systems have taken these ideas even farther. Pyramid has used the power of the ZCA to build a phenomenally customizable system that's remarkably easy to work with.

Adaptation is not limited to web systems, you can use these powerful concepts in your software as well. The approach allows you to adapt fluidly to changing needs.

"Socialist Realist Sculpture" by Raymond Cunningham CC-BY-2.0
https://www.flickr.com/photos/zaruka/6105510696

The world of the web continues to change. RESTful APIs are replacing server bound systems. Client-side apps are becoming the norm. The “headless CMS” is an idea whose time has come. Powered by adaptation, Plone will continue to be a vital part of this future. We’d love it if you join us there.



"Socialist Realist Sculpture" by Raymond Cunningham CC-BY-2.0
https://www.flickr.com/photos/zaruka/6105510696

The world of the web continues to change. RESTful APIs are replacing server bound systems. Client-side apps are becoming the norm. The “headless CMS” is an idea whose time has come. Powered by adaptation, Plone will continue to be a vital part of this future. We’d love it if you join us there.

Plone 5

Additional Credits

"the kitchen sink" by Alan Cleaver, CC-BY
https://www.flickr.com/photos/alancleaver/3727870484

"Shower at Victoria Park Swimming Pool" by Malesworldoa, CC-BY-SA
https://commons.wikimedia.org/wiki/File:HK_CWB_%E7%B6%AD%E5%A4%9A%E5%88%A9%E4%BA%9E%E5%85%AC%E5%9C%92%E6%B8%B8%E6%B3%B3%E6%B1%A0_old_Victoria_Park_Swimming_Pool_%E7%94%B7%E6%9B%B4%E8%A1%A3%E5%AE%A4_Changing_Room_shower_bathing_heads_Sept-2013.JPG

"Cat in a washing machine" by James Cridland, CC-BY
https://www.flickr.com/photos/jamescridland/2663785397

Font Awesome by Dave Gandy - http://fontawesome.io

Upgrade to Plone


Plone
5
Cris Ewing
@crisewing
cris@crisewing.com
cewing
Upgrade to Plone