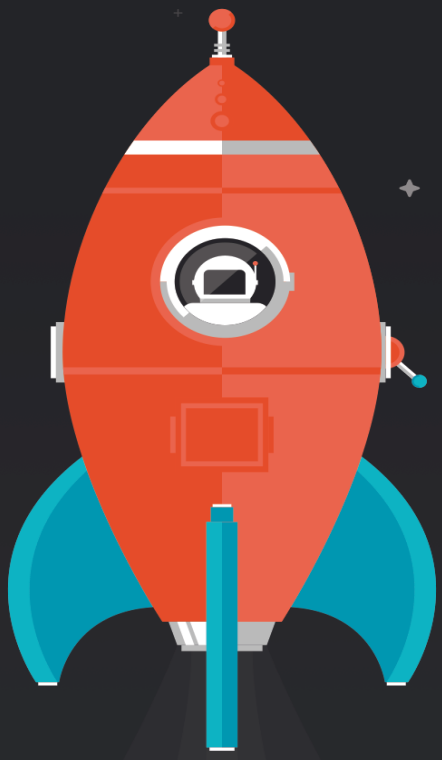
استكشف فلاسك



كتابة: روبرت بيكارد

ترجمة: أحمد نورالله

أنجزت هذا الكتاب آملاً أن يكون سنداً في تعلّم إطار فلاسك بين المبرمجين العرب، وساعياً لإثراء المحتوى العربي الشحيح حول هذه التقنيّة اليسيرة، وسائلاً اللّٰه أن يكون في ميزات حسناتي. آمل أن يساعدك هذا الكتاب كما ساعدني، وأشكر الكاتِب (روبرت بيكارد) على هذا العمل المفيد.

أود أيضاً أن أنوِّه أنَّ في الكتاب العديد من المعلومات منتهيّة الصلاحيّة. مما يعني أنَّك قد تجد مُفسِرك لا يتعرَّف على بعض الشيفرات. أعدك أنَّك لن تقع في الكثير من هذه الحالات، فإطار فلاسك لم يتغيّر لهذه الدرجة، إضافةً إلى أنَّ الكتاب يسعى إلى تقديم مشورة عامة في معظم فصوله.

أُنجِزَ هذا العمل بفضل اللّٰه وبفضل أفراد فريق أوروك، فريق يسعى لنشر البرمجيات الحرَّة وتطويرها. تفضل بزيارة موقعنا للمزيد من التفاصيل.

نسخة الويب

حُرِّرَ في 9 رمضان 1439هـ

جدول المحتويات

المقدمة

افتراضات

بهدف تزويدك بنصائح أكثر تحديداً، قمت بكتابة هذا الكتاب بناءً على بضعة افتراضات أساسيّة. فمن المهم أن تضع هذه الافتراضات بعين الاعتبار خلال قراءتك للكتاب وتطبيق الاقتراحات المذكورة فيه على مشاريعك.

الجمهور المُستهدَف

تم بناء هذا الكتاب اعتماداً على المعلومات الموجودة في التوثيقات الرسميّة. أنصحك بشدّة أن تقرأ دليل المُستخدِم وهذه الدورة. ستعلمك المصادر السابقة المصطلحات الأساسيّة المتعلقة بالإطار. فيجب عليك فهم دوال العرض (Views) ومحرك القوالب جينجا (Jinja) وغيرها من المفاهيم الأساسيّة المهمة للمبتدئين. حاولتُ قدر الإمكان تجنب إعادة المعلومات الموجودة في دليل المستخدم، لذلك إن قرأت هذا الكتاب أولاً - من دون قراءة المصدرين السابقين - هناك احتمال كبير أن تتشتت.

بالرغم مما قلنا، مواضيع هذا الكتاب ليست متقدمة جداً. فالهدف الرئيسي هو عنونة الممارسات والأساليب المثلى التي تجعل عمليّة التطوير أسهل باتباعها. وعلى الرغم من أنني أحاول عدم إعادة المعلومات الموجود في التوثيق الرسمي، قد تجد أنني قمت بتكرار مفاهيم معينة أحياناً لأتأكد من معرفتك لها. لذلك ليس عليك أن تفتح دورة للمبتدئين تزامناً مع قراءتك لهذا الكتاب.

الإصدارات

بايثون الإصدار الثاني مقارنةً مع بايثون الإصدار الثالث

في أثناء كتابتي لهذه الكلمات، يعد مجتمع بايثون في خضم الانتقال من الإصدار الثاني إلى الثالث. أما موقف مؤسسة برمجيات بايثون فهو كالتالي:

> بايثون 2 يمثل الوضع الحالي، أما بايثون 3 فهو يمثل مستقبل وغَد اللغة.[1]

بدءً من الإصدار 0.10، أصبح فلاسك قادراً على العمل مع بايثون الإصدار 3.3. عندما سألت أرمين روناشر عمّا إذا كان على مستخدمي فلاسك البدء باستخدام بايثون الإصدار 3.3 في تطبيقاتهم، أجابني حينها قائلاً:

> أنا لا أستخدم هذا الإصدار حالياً، وأنا لا أوصي الناس عادةً باستخدام أشياء لا أثق بفعاليتها، لذلك أنا حريص جداً حول مسألة التوصيّة باستخدم الإصدار الثالث من اللغة حالياً.
>
> — أرمين روناشر، مُخترع إطار فلاسك[2]

أحد أسباب التمهل في استخدام الإصدار الثالث هو أن العديد من الاعتماديات الشائعة لا تعمل على هذا الإصدار بعد. فأنت بالتأكيد لا ترغب ببناء مشروع باستخدام بايثون 3 لتنتظر بضعة شهور لأنك لا تستطيع استخدام الحزمة الفلانيّة. من الممكن أن يوصي

(1) موسوعة بايثون

(2) محادثتي مع أرمين روناشر

فلاسك في نهاية الأمر باستخدام الإصدار الثالث في المشاريع الجديدة، ولكن حالياً الإصدار الثاني هو المُسيطر.

> **ملاحظة**
>
> يقوم هذا الموقع بتتبع الحزم الرئيسيّة التي أصبحت تعمل مع الإصدار الثالث.

وبما أن الكتاب يهدف إلى تقديم مشورة عملية، فإنه من المنطقي أن استخدم الإصدار الثاني فيه. على وجه التحديد، سأستخدم في هذا الكتاب إصدار بايثون 2.7. قد تؤدي التحديثات المستقبليّة إلى تطور مجتمع فلاسك من هذه الناحية، ولكن حالياً الإصدار 2.7 هو ما سنستخدمه في كتابنا.

فلاسك الإصدار 0.10

حالياً، وفي أثناء كتابتي لهذه الكلمات، الإصدار 0.10 هو الإصدار الأحدث من فلاسك (الإصدار 0.10.1 لأكون دقيقاً). معظم الدروس في هذا الكتاب لن تتأثر بالتحديثات الطفيفة التي سيتم إجراءها على الإطار لاحقاً، ومع ذلك هذا شيء يؤخذ بعين الاعتبار.

وثيقة حيّة!

سيتم تحديث محتوى هذا الكتاب على الطاير (بشكل مُتقطِّع)، وليس بإصدارات دوريّة. تلك إحدى فوائد وضع المحتوى مجاناً بدلاً من إخفاءه خلف حديقة مسوّرة. فشبكة الإنترنت هي قناة توزيع أكثر مرونة من الطباعة وحتى من الكتب الإلكترونيّة (PDF).

محتوى الكتاب مُستضاف على موقع جيتهاب حيث ستتم جميع أعمال "التطوير" (محتوى النسخة العربية موجود في هذا المستودع). المساهمات والأفكار دائماً مرحب بها!

الطرق التنسيقيّة المُستخدمة في الكتاب

الفصول منفصلة

يعتبر كل فصل في هذا الكتاب درساً منفصلاً. الكثير من الكتب والدورات تُكتَب على شكل درس طويل. وهذا يعني عموماً أن يتم إنشاء مثال واستخدامه في جميع فصول الكتاب لشرح المفاهيم والدروس. بدلاً من ذلك، سنستخدم مثالاً منفصلاً في كل درس لشرح مفهوم معين، وذلك لا يعني أنه سيتم جمع تلك الأمثلة من الفصول المختلفة في مشروع واحد كبير في نهاية الكتاب.

التنسيق

سيتم استخدام الحواشي السفليّة للاستشهاد بمصادر حتى لا تظن أنني أختلق الأمور.[1]

سيتم استخدام النص *المائل* للدلالة على اسماء الملفات.

سيتم استخدام النص **العريض** للدلالة على مصطلح مهم أو جديد.

(1) هل رأيت؟ بهذه الطريقة ستظهر!

> **تحذير**
> سوف تُعرَض الأخطاء الشائعة التي من الممكن أن تسبب مشاكل كبيرة في مربع كهذا.
>
> **ملاحظة**
> سوف تُعرَض المعلومات التكميليّة في مربعات كهذه.

بيضات الفصح

تم نثر وترميز أسماء لستة متبرعين من حملة Kickstarter (حملة تبرعات لمشاريع) في أنحاء الكتاب. إذا وجدت الأسماء الستة وأرسلت بريد إلكتروني لي بمواقعهم سأرسل لك جائزة متواضعة للغاية. لا تلميحات!

الخلاصة

- يحتوي هذا الكتاب على توصيات (نصائح) لاستخدام إطار العمل فلاسك.
- افترض أنك قرأت دورة فلاسك الرسميّة.
- استخدم في هذا الكتاب بايثون الإصدار 2.7.
- استخدم في هذا الكتاب فلاسك الإصدار 0.10.
- سأبذل ما في وسعي لإبقاء محتويات هذا الكتابة مُحدَثِة.
- الكتاب يتألف من فصول منفصلة.

- يوجد بضعة طرق سأستخدمها في التنسيق لإيصال معلومات إضافية عن المحتوى.
- ستُستخدَم الملخصات كقوائم موجزة لمعلومات الفصول.

الفصل الأوَّل: أعراف كتابة الشيفرة البرمجيّة

يوجد العديد من الأعراف في مجتمع بايثون لتوجيهك إلى طريقة تنسيق التعليمات البرمجية خاصتك. قد تكون مُلِماً ببعض هذه الأعراف إن كانت لك تجربة مع اللغة من قبل. سأبقي معلومات هذا القسم مُختصرة قدر الإمكان، كما سأضع بعض الروابط للتمكن من إيجاد معلومات موسّعة حول حيثيّات الأمور.

دعنا نتعرف على هذه الأعراف!

يُقصد بمصطلح **PEP**: "مقترحات بايثون التعزيزية" (Python Enhancement Proposal). هذه المُقترحات مُفهرسة ومُستضافة على موقع بايثون الرسمي، وتُقسم إلى عدد من التصنيفات، منها: meta-PEPs، وهي مقترحات نظريّة أكثر من تقنيّة. أما على الجانب الآخر، فتصف المقترحات التقنيّة عادةً أشياء مثل تحسين الأجزاء الداخلية للغة.

يوجد بضعة مقترحات مثل PEP 8 و PEP 257 تهدف إلى إرشادك إلى الطريقة التي نكتب بها التعليمات البرمجية. يحتوي المُقترح PEP 8 على دليل تنسيق التعليمات البرمجية. أمّا المُقترح PEP 257 فيحتوي على دليل حول السلاسل التوثيقية (Docstrings)، وهي الطريقة المقبولة عموماً لتوثيق التعليمات البرمجية.

المُقترح PEP 8: دليل تنسيق الشيفرات البرمجية في لغة بايثون

يوفر هذا المُقترح دليل رسمي للشيفرة البرمجية الخاصة ببايثون. أنصحك بقراءته وتطبيق توصياته على مشاريعك بإطار فلاسك (وجميع مشاريعك الأخرى باللغة عامةً). بحيث ستصبح شيفرتك أكثر قبولاً عندما يبدأ مشروعك بالنمو ليصبح متكوناً من مئات، أو آلاف الأسطر البرمجية. فجميع توصيات هذا المُقترح تتمحور حول جعل التعليمات البرمجية مقروءة بشكل أفضل. إضافةً لذلك، إن كان مشروعك مفتوح المصدر، فإنَّ المساهمين المحتملين غالباً ما سيكونون أكثر ارتياحاً بالتعامل مع تعليمات برمجية مكتوبة وفق توصيات هذا المُقترح.

إحدى التوصيات المهمة تحديداً في هذا المُقترح هي استخدام أربعة مسافات في كل مستوى إزاحة (indentation level)، وبدون الاستعانة بزر الجدولة (الزر TAB). ستُحدِث عبئاً عليك وعلى الآخرين عند التنقل بين المشاريع في حال عدم امتثالك لهذه التوصية. عدم اعتماد طريقة موحدة للإزاحة أمر مُزعِج في أي لغة، ولكنه مهم بشكل خاص في بايثون. لذلك التنقل بين استخدام زر الجدولة واستخدام المسافات (ضغط أربع مرات على زر المسطرة) قد يُنتِج العديد من الأخطاء التي من المُزعِج إصلاحها.

المُقترح PEP 257: السلاسل التوثيقيّة

يغطي هذا المُقترح مفهوم قياسي آخر في اللغة ألا وهو **السلاسل التوثيقيّة** (docstrings). يمكنك قراءة تعريف هذه التوصية والتوصيات الأخرى في الفهرس، ولكن أدناه يوجد مثال لإعطاءك فكرة حول كيف تبدو هذه السلاسل التوثيقية:

```
def launch_rocket():
   """Main launch sequence director.

   Locks seatbelts, initiates radio and fires engines.
   """
   # [...]
```

تُستخدم هذه الأنواع من السلاسل التوثيقية بواسطة برمجيات مثل Sphinx لتوليد ملفات توثيقية بصيغة HTML و PDF وغيرها من الصيغ الأخرى. كما تجعل هذه السلاسل التعليمات البرمجية أكثر سهولة من ناحية الفهم.

> **ملاحظة**
>
> – المُقترح PEP 8
>
> – المُقترح PEP 257
>
> – برمجية Sphinx، مُولِد توثيقات مكتوب بواسطة نفس الأشخاص الذين اخترعوا إطار فلاسك

الاستيرادات النسبية

يجعل الاستيراد النسبي الأمور أسهل عند تطوير التطبيقات التي تستخدم إطار فلاسك. هذه التقنية بسيطة وسهلة لأبعد الحدود، وإليك هذا المثال للتوضَّح أكثر: دعنا نقول أنك تريد استيراد الوحدة `User` من الملف *myapp/models.py*. قد تظن أنك يجب أن تكتب اسم الحزمة بهذا الشكل `myapp.models`، ولكن باستخدام الاستيرادات النسبية يمكنك تحديد موقع الوحدة الهدف تبعاً لمكان تواجدها (مصدرها). للقيام بهذا نستخدم النقطة، بحيث أول نقطة تحدد الدليل (directory) الحالي وكل نقطة لاحقة تمثل الدليل الأب التالي (الدليل الحاوي للدليل الحالي). المثال أدناه يوضح الفرق بين طريقة كتابة الاستيراد المطلق والنسبي.

```
# myapp/views.py

# طريقة الاستيراد المطلق لجلب الوحدة User
from myapp.models import User

# طريقة الاستيراد النسبي، وهي تفعل نفس الشيء
from .models import User
```

إحدى محاسن هذه الطريقة أنَّ الحزمة تصبح أكثر معيارية. فيمكنك الآن إعادة تسمية الحزمة وإعادة استخدام وحداتها من مشروع آخر دون الحاجة لتحديث تعليمات الاستيراد.

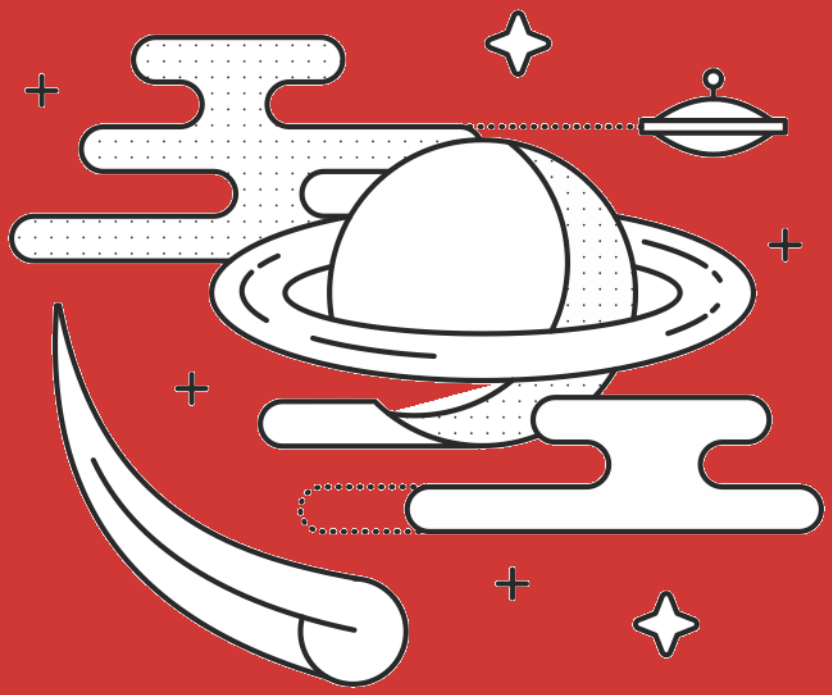
وجدتُ تغريدةً، خلال بحثي، توضِّح فائدة الاستيرادات النسبية، تقول:

> استغرق الأمر دقيقة واحدة لإعادة تسمية الحزمة بأكملها، يا لروعة الاستيراد النسبي!
> — David Beazley, @dabeaz
>
> **ملاحظة**
> يمكنك قراءة المزيد حول الصيغة الكتابية للاستيرادات النسبية في القسم المخصص لها في المُقترح PEP 328

الخلاصة

- حاول اتباع أعراف تنسيق التعليمات البرمجية الموضوعة في المُقترح PEP 8.
- حاول توثيق تطبيقك بالسلاسل التوثيقية المشروحة في المُقترح PEP 257.
- استخدم الاستيراد النسبي لاستيراد الوحدات الداخليّة في تطبيقك.

الفصل الثاني: بيئة العمل



من المُرجَّح أنَّ تطبيقك سيحتاج الكثير من الاعتماديات ليؤدي عمله. فإذا لم يتطلب على الأقل حزمة إطار فلاسك، فغالباً أنت تقرأ الكتاب الخطأ. **بيئة عمل** تطبيقك هي كل الأشياء التي يحتاجها عندما يعمل. من حسن حظنا، يوجد عدد من الأشياء التي يمكننا القيام بها لجعل إدارة بيئة العمل أقل تعقيداً.

استخدم برمجية virtualenv لإدارة بيئتك

برمجية virtualenv هي أداة تُستخدَم لعزل تطبيقاتك في ما يسمى **بالبيئة الافتراضيّة** (virtual environement). البيئة الافتراضيّة هي الدليل الذي يحوي البرمجيات التي يعتمد عليها تطبيقك (الاعتماديات). كما تقوم البيئة الافتراضية أيضاً بتغيير بعض متغيّرات البيئة خاصتك للحفاظ على بيئتك التطويريّة مُحتواة (تحت السيطرة). فبدلاً من تنزيل الحزم أمثال فلاسك في دلائل الحزم الموجودة في فضاء النظام - أو فضاء المستخدم - يمكننا تنزيلهم في دليل معزول مُستخدم من التطبيق الحالي فقط. هذا يجعل من السهل تحديد أي إصدار بايثون ليتم إستخدامه وتحديد الاعتماديات التي نريدها لكل مشروع.

تتيح لنا هذه البرمجيّة أيضاً استخدام إصدارات مختلفة لنفس الحزمة في مشاريع مختلفة. هذه المرونة قد تكون مهمة إن كنت تعمل على نظام قديم بمشاريع مختلفة تملك متطلبات بإصدارات مختلفة.

عندما تستخدم هذه البرمجية، سيصبح لديك فقط بضعة حزم بايثون مثبتة على النظام ككل. إحدى هذه الحزم ستكون أداة virtualenv ذاتها. يمكنك تثبيت حزمة `virtualenv` باستخدام `pip`.

حالما تمتلك الأداة على نظامك، يمكنك البدء بإنشاء البيئات الافتراضيّة مباشرةً. قم بالذهاب إلى دليل المشروع ونفِذ الأمر `virtualenv`. يأخذ هذا الأمر معامل واحد، والذي يمثل اسم الدليل الهدف للبيئة الافتراضيّة. المثال أدناه يُظهِر عملية إنشاء بيئة افتراضية باستخدام هذه الأداة.

```
$ virtualenv venv
New python executable in venv/bin/python
Installing Setuptools...........[...].....done.
Installing Pip.................[...].....done.
$
```

سيقوم هذا الأمر بإنشاء دليل جديد حيث يمكنك تثبيت اعتماديات مشروعك.

ينبغي عليك تفعيل البيئة الافتراضيّة بعد إنشائها عن طريق استدعاء البرمجيّة *bin/activate* التي تم إنشأها داخل البيئة الافتراضية.

```
$ which python
/usr/local/bin/python
$ source venv/bin/activate
(venv)$ which python
/Users/robert/Code/myapp/venv/bin/python
```

تقوم البرمجية *bin/activate* بإجراء بعض التغييرات على متغيّرات الصدفة (shell) خاصتك؛ ليشير كل شيء (الخدمات والبرمجيات) إلى البيئة الافتراضية الحالية بدلاً من النظام. بعد تفعيل البيئة، سيشير الأمر `python` إلى مُفسِر البايثون الموجود داخل البيئة الافتراضية، كما والاعتماديات المثبتة باستخدام `pip` سيتم تنزيلها في البيئة الافتراضية بدلاً من النظام.

قد تلاحظ أيضاً أن مُحث الصدفة (shell prompt) قد تغيّر؛ وهذا لأنّ برمجية virtualenv تقوم بإضافة اسم البيئة الافتراضية المُفعلة إليه حتى تتمكن من معرفة أنّك لا تعمل على صدفة النظام.

يمكنك تعطيل البيئة الافتراضية باستخدام الأمر `deactivate`.

```
(venv)$ deactivate
$
```

برمجية virtualenvwrapper

تُستخدَم الحزمة virtualenvwrapper لإدارة البيئات الافتراضية المُنشأة باستخدام أداة virtualenv. لم أرغب بذكر هذه الأداة قبل رؤيتك لأساسيات virtualenv حتى تتمكن من فهم ما الذي تضيفه هذه الأداة ولمَ ينبغي عليك استخدامها.

البيئة الافتراضية التي أنشأناها في المثال السابق تضيف فوضى إلى مستودع مشروعك. فأنت لا تتعامل معها سوى عندما تقوم بتفعيلها كما لا ينبغي تتبعها بنظام التحكم بالإصدارات، وبالتالي لا داعي لوجودها في مستودع المشروع أصلاً. الحل لهذا هو

استخدام أداة virtualenvwrapper. تقوم هذه الحزمة بإبقاء جميع البيئات الافتراضية في دليل واحد، والذي هو افتراضياً في *~/.virtualenvs*.

قم باتباع الإرشادات الموجودة في توثيقها الرسمي لتثبيت الأداة.

> **تحذير**
> تأكد من تعطيل كل البيئات الافتراضية قبل تثبيت virtualenvwrapper، فأنت تريد تثبيتها في النظام وليس في بيئة افتراضيّة.

والآن قم باستخدام الأمر `mkvirtualenv` لإنشاء بيئة افتراضية جديدة بدلاً من استخدام `virtualenv`:

```
$ mkvirtualenv rocket
New python executable in rocket/bin/python
Installing setuptools.....[...].....done.
Installing pip.....[...].....done.
(rocket)$
```

سيقوم الأمر `mkvirtualenv` بإنشاء دليل في مجلد البيئات الافتراضية خاصتك وسيقوم بتفعيل البيئة لك. كما سابقاً، سيشير مُفسِر `python` والبرمجية `pip` إلى البيئة الافتراضية بدلاً من النظام. لتفعيل بيئة افتراضية معينة استخدم الأمر `workon [اسم البيئة]`.

مايزال الأمر `deactivate` يُستخدم لتعطيل البيئة.

تتبع الاعتماديات

مع نمو مشروعك، ستجد أنَّ قائمة الاعتماديات تنمو معه. فإنَّه ليس من غير المألوف أنْ تحتاج إلى عشرات الحزم لتشغيل تطبيق فلاسك. الطريقة الأسهل لإدارة هذا هو استخدام ملف نصي بسيط. تحتوي برمجية `pip` على خاصيّة تولِّد ملف نصي يسرد جميع الحزم المُثبتة. كما تتمكن من قراءته أيضاً لتثبيت كل حزمة مسرودة فيه على نظام آخر، أو في بيئة طازجة.

تعليمة pip freeze

يُستخدم الملف النصي *requirements.txt* من قبل العديد من تطبيقات فلاسك لسرد جميع الحزم المطلوبة لتشغيل هذا التطبيق. المثال أدناه يشرح كيفيّة إنشاء هذا الملف، كما يشرح المثال الذي يليه كيفيّة استخدام هذا الملف النصي لتثبيت الاعتماديات في بيئة جديدة.

```
(rocket)$ pip freeze > requirements.txt
```

```
(rocket)$ pip freeze > requirements.txt

$ workon fresh-env
(fresh-env)$ pip install -r requirements.txt
[...]
Successfully installed flask Werkzeug Jinja2 itsdangerous
markupsafe
Cleaning up...
(fresh-env)$
```

التتبع اليدوي للاعتماديات

مع نمو مشروعك، قد تجد بعض الحزم المُدرجة بواسطة الأمر `pip freeze` ليست في الواقع ضروريّة لتشغيل التطبيق. حيث سيكون لديك أيضاً الحزم المُثبتة للاستخدام في عملية التطوير فقط. الأمر `pip freeze` لا يميّز بين الإثنين، فهو يقوم فقط بإدراج الحزم المُثبتة حالياً. كنتيجة لذلك، قد ترغب بتتبع اعتمادياتك يدوياً عند إضافتها. يمكنك فصل بين الحزم الضروريّة لتشغيل التطبيق والحزم الضروريّة لتطويره ووضعهم في ملفين منفصلين (بالاسم *require_run.txt* و *require_dev.txt* على سبيل المثال).

التحكم بالإصدارات

اختر نظام تحكم بالإصدارات واستخدمه. انصحك باستخدام جيت (Git). مما رأيته، جيت هو الخيار الأكثر شعبيّة للمشاريع الجديدة هذه الأيام. أن تكون قادراً على حذف التعليمات البرمجية دون الحاجة إلى القلق من القيام بأخطاء لا رجعة فيها شيءٌ لا يقدر بثمن. حيث ستكون قادراً على الحفاظ على مشروعك خالياً من التعليقات، لأنّه يمكنك حذف ما تريد الآن والرجوع عن التغيير إذا دعت الحاجة. إضافةً لذلك، ستملك نسخ إحتياطية لمشروعك على جيتهاب، أو Bitbucket، أو خادم Gitolite خاص بك.

ماذا يجب أن أُبقي خارج نظام التحكم بالإصدارات؟

عادةً ما أبقي ملف خارج نظام التحكم بالإصدارات لسببٍ من سببين: إما لأنّه خردة (غير ضروري)، أو لأنّه سري. ملفات *pyc*. ومجلدات البيئات الافتراضية - إن كنت لا تستخدم

virtualenvwrapper لسببٍ ما - أمثلة على الخردة. وذلك لأنَّه لا داعي لأن يكونوا في نظام التحكم بالإصدارات؛ فمن الممكن إعادة إنشائهم باستخدام ملفات *py.* و *requirement.txt* على التوالي.

مفاتيح الواجهات البرمجية، والمفاتيح السرية للتطبيق، وبيانات اعتماد قاعدة البيانات هي أمثلة على الأسرار. لأنه لا ينبغي وجودهم في نظام التحكم بالإصدارات؛ فكَشفُهُم للعامة سيكون خرقاً أمنياً كبيراً.

> **ملاحظة**
> عندما اتخذ قرارات متعلقة بالأمن، غالباً ما أفترض أن مستودعي سيصبح ظاهراً للعموم في مرحلةٍ ما. وهذا يعني الإبقاء على الأشياء السرية وعدم الافتراض أنه لن يتم العثور على ثغرة أمنية؛ لأنَّ مبدأ افتراض أن "من سيستطيع تخمين أنها تتمكن من فعل ذلك؟" معروف في المجال الأمني بالإبهام (obscurity) وهي سياسة سيئة للاعتماد عليها.

عند استخدام جيت، يمكنك إنشاء ملف خاص يدعى *gitignore.* في مستودعك. قم بسرد أنماط في هذا الملف تستخدم حروف بدل (Wildcard) لتطابق أسماء ملفات معينة. أي ملف يتطابق مع أحد تلك الأنماط سيتم تجاهله بواسطة جيت. أنصحك باستخدام ملف *gitignore.* الموضح أدناه للتوضح الفكرة.

```
*.pyc
instance/
```

تُستخدَم مجلدات *instance* لجعل متغيّرات الإعداد السرية متاحة لتطبيقك بطريقة أكثر أماناً. سنتحدث المزيد عنهم لاحقاً.

> **ملاحظة**
>
> يمكنك قراءة المزيد حول ملف *gitignore*. على http://git-scm.com/docs/gitignore

التنقيح (تصحيح الأخطاء)

وضع التنقيح

يأتي إطار فلاسك مع ميزة مفيدة تدعى "وضع التنقيح". عليك تعيين `debug = True` لتشغيلها في إعدادات التطوير خاصتك. عندما يكون هذا الخيار قيد العمل، يقوم الخادم بإعادة التشغيل عند إجراء تغييرات على الشيفرة البرمجية، كما وتظهر الأخطاء في رصة تتبع (stack trace) ووحدة تحكم تفاعلية (interactive console).

> **تحذير**
>
> انتبه من تشغيل وضع التنقيح في وضع الإنتاج؛ فتتيح وحدة التحكم التفاعلية تنفيذ تعليمات برمجية مما سيؤدي إلى هشاشة أمنية كبيرة إذا تُرِكت هذه الخاصية قيد العمل في الموقع الحي (المرفوع على الشابكة).

أداة Flask-DebugToolbar

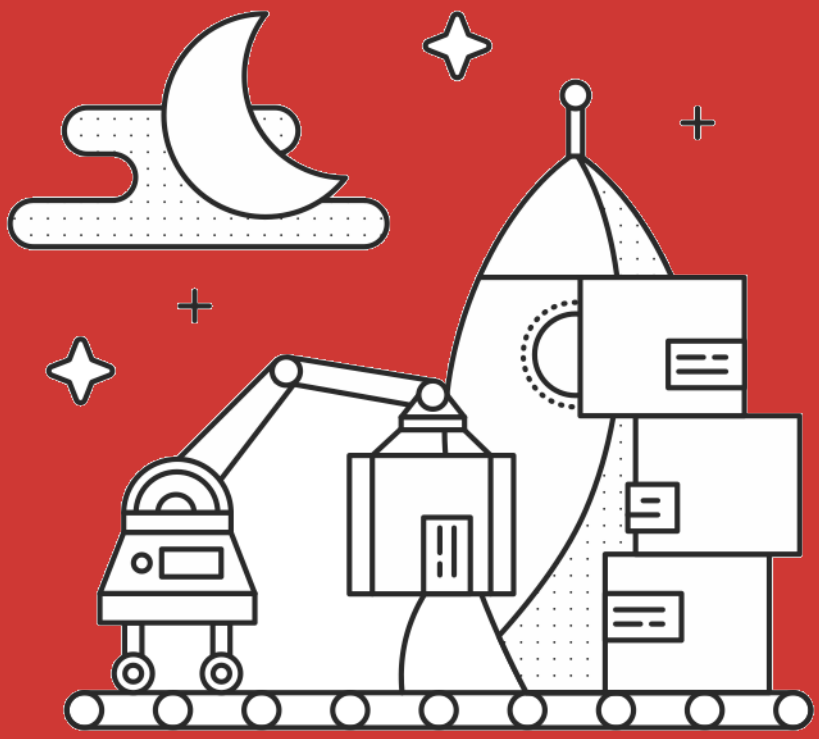
برمجية Flask-DebugToolbar هي أداة رائعة أخرى لتنقيح المشاكل الموجودة في تطبيقك. في وضع التنقيح، ستقوم الأداة بإضافة شريط جانبي في كل صفحة في تطبيقك. هذا الشريط يوفِر معلومات حول استعلامات SQL، والتسجيل (logging)، والإصدارات، والقوالب، والإعدادات، وغيرها من الأشياء الجميلة التي ستجعل تتبع الأخطاء أسهل.

> ملاحظة
> – خذ نظرة على قسم البداية السريعة مع وضع التنقيح.
> – يوجد بعض المعلومات الجيدة حول التعامل مع الأخطاء، والتسجيل، والعمل مع المنقحات الأخرى في توثيقات فلاسك.

الخلاصة

- استخدم أداة virtualenv لإبقاء اعتماديات تطبيقك في مكان واحد.
- استخدم أداة virtualenvwrapper لإبقاء بيئاتك الافتراضية في مكان واحد.
- تتبَّع الاعتماديات باستخدام ملف نصي أو أكثر.
- استخدم نظام تحكم بالإصدارات. أنصحك باستخدم جيت.
- استخدم ملف gitignore. لإبقاء الفوضى والأسرار خارج نظام التحكم بالإصدارات.
- يمنحك وضع التنقيح معلومات حول المشاكل في عملية التطوير.
- ستمنحك إضافة Flask-DebugToolbar المزيد من المعلومات المفيدة في عملية التنقيح.

الفصل الثالث: تنظيم المشروع



يترك فلاسك خيار تنظيم التطبيق على عاتقك. تلك إحدى الأسباب التي جعلتني أحب فلاسك كمبتدئ، ولكن هذا يعني أنك يجب أن تولي بعض الاهتمام لطريقة هيكلة الشيفرة البرمجية. يمكنك وضع التطبيق كاملاً في ملف واحد، أو يمكنك تقسيمه إلى عدة حزم. توجد بضعة أساليب تنظيميّة يمكننا اتباعها تجعل عملية التطوير والنشر أسهل.

تعريفات

دعنا نُعرِف بعض المصطلحات قبل أن نبدأ بهذه الوحدة.

مستودع (Repository): المجلد الأساسي حيث يتوضَّع تطبيقك. يشير هذا المصطلح تقليدياً إلى أنظمة التحكم بالإصدارات، والتي يجب أن تستخدمها. عندما سأذكر هذا المصطلح في هذا الكتاب، سيكون مقصدي الحديث عن الدليل الجذر لمشروعك (دليل المستوى الأول). على الأرجح أنَّك لن تقوم بالخروج من هذا الدليل عند عملك على تطبيقك.

حزمة (Package): يشير هذا المصطلح إلى حزمة البايثون التي تحتوي على الشيفرة البرمجية لتطبيقك. سأتحدث أكثر حول إعداد تطبيقك كحزمة في هذا الفصل، ولكن حالياً كل ما عليك معرفته هو أنَّ الحزمة تمثل دليل فرعي من المستودع.

وحدة (Module): تمثل الوحدة ملف بايثون واحد يُمكن استيراده من أي ملف بايثون آخر. الحزمة بالأساس هي عدة وحدات مُحزمِة معاً.

> ملاحظة
> – اقرأ المزيد حول الوحدات في دورة بايثون الرسميّة.
> – نفس الصفحة السابقة تحتوي على قسم حول الحزم.

أساليب التنظيم

وحدة مفردة

العديد من أمثلة فلاسك التي ستراها تضع الشيفرة في ملف واحد، وغالباً ما يسمى *app.py*. هذه طريقة رائعة لاستخدامها في المشاريع البسيطة والسريعة (مثل التي تُستخدَم في الدورات)، حيث فقط تحتاج لاستخدام بعض الموجهات وكتابة أقل من بضعة مئات السطور.

```
app.py
config.py
requirements.txt
static/
templates/
```

شيفرة التطبيق ستوضع في الملف *app.py* كما هو موضح في المثال أعلاه.

حزمة

عندما تعمل على مشروع أكثر تعقيداً بقليل، التنظيم باستخدام وحدة مفردة لن يفي بالغرض. فستحتاج لتعريف صنوف (Classes) للنماذج (Models) والاستمارات (Forms)، وسينمزج كل هذا مع الموجهات والإعدادات. كل هذا يُمكن أن يثبط عملية التطوير. يمكنك جمع المكونات المختلفة لتطبيقك في مجموعة من الوحدات المترابطة، أي في حزمة، لحل هذه المشكلة.

```
config.py
requirements.txt
run.py
instance/
    config.py
yourapp/
    __init__.py
    views.py
    models.py
    forms.py
    static/
    templates/
```

يسمح لك هذا التنظيم - الموضح أعلاه - جمع المكونات المختلفة لتطبيقك بطريقة منطقية. بحيث يتم جمع تعريفات الصنوف للنماذج في الملف *models.py*، وتعريفات الموجهات في الملف *views.py*، ويتم تعريف الاستمارات في الملف *forms.py* (يوجد فصل كامل حول الاستمارات لاحقاً).

يوفر الجدول أدناه الخلاصة الأساسية للمكونات التي ستجدها غالباً في تطبيقات فلاسك. من المرجح أنك ستتعامل مع ملفات أخرى في مستودعك، ولكن هذه المكونات شائعة في معظم التطبيقات.

الملف	الشرح
runs.py	هذا هو الملف الذي يتم استدعاؤه لبدء خادم التطوير. يحصل هذا الملف على نسخة من التطبيق من حزمتك ويقوم بتشغيلها. لن يتم استخدام هذا الملف في وضع الإنتاج، ولكنك سترى الكثير من الأميال منه في وضع التطوير.
requirements.txt	يسرد هذا الملف جميع الحزم التي يعتمد عليها تطبيقك. قد تملك ملفات منفصلة لاعتماديات وضع التطوير والإنتاج.
config.py	يحتوي هذا الملف على متغيرات الإعداد التي يحتاجها تطبيقك.
/instance/config.py	يحتوي هذا الملف على متغيرات الإعداد التي لا ينبغي أن يتم تتبعها باستخدام نظام التحكم بالإصدارات. بما فيها: مفاتيح الواجهات البرمجية وروابط قاعدة البيانات التي تحتوي على كلمات المرور. كما يحتوي أيضاً على متغيرات خاصة بحالة معينة من تطبيقك. على سبيل المثال، قد تستخدم `DEBUG = False` في ملف config.py، بينما تستخدم `DEBUG = True` في ملف instance/config.py على جهازك المحلي لعملية التطوير. وفي حين أن هذا الملف يتم قراءته بعد قراءة الملف config.py، فسيتم تجاهل القيمة الموجودة في الملف config.py واستبدالها ب `DEBUG = True`.
/yourapp/	هذه هي الحزمة التي تحتوي على مكونات تطبيقك.

الملف	الشرح
/yourapp/__init__.py	هذا هو الملف الذي يستهل مشروعك ويجمع جميع المكونات المختلفة معاً.
/yourapp/views.py	هنا حيث يتم تعريف الموجهات. قد يتم فصل الموجهات إلى حزمة بحد ذاتها (*yourapp/views*) تحتوي على دوال عرض متعلقة مجموعة معاً في وحدات.
/yourapp/models.py	هنا حيث تُعرَّف نماذج تطبيقك. قد يتم فصل هذا الملف إلى وحدات متعددة بنفس طريقة فصل الملف views.py.
yourapp/static/	الدليل الذي يحوي ملفات CSS و Javascript والصور وغيرها من الملفات التي تريدها أن تظهر للعموم بواسطة تطبيقك. يمكن الوصول لهذه الملفات من الرابط yourapp.com/static افتراضياً.
/yourapp/templates	هنا حيث سيتم وضع قوالب جينجا الخاصة بتطبيقك.

المُخطّطات

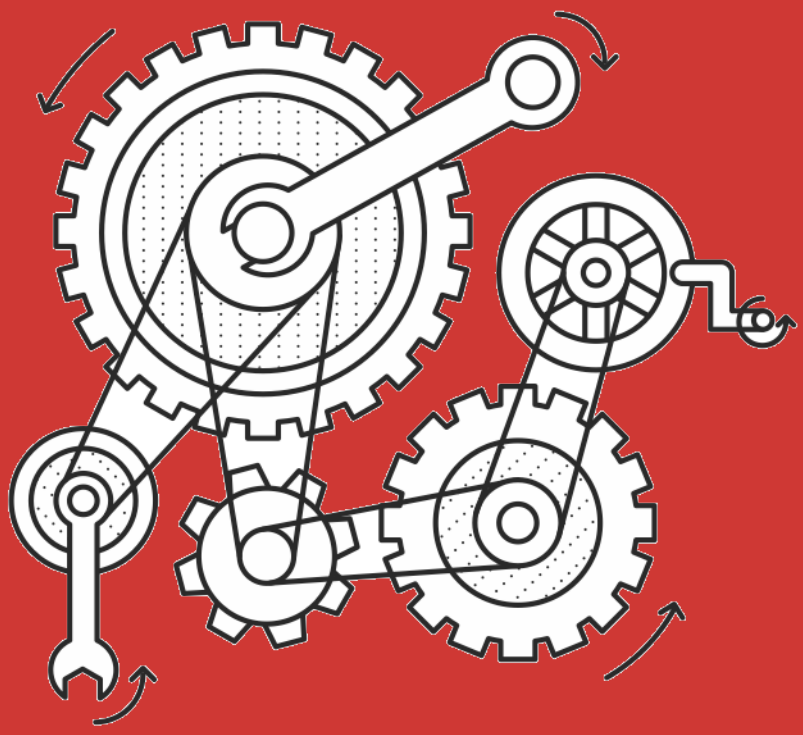
في هذه المرحلة قد تجد أنك تملك العديد من الموجهات المتعلقة. إذا كنت تفكر مثلي فستفكر بدايةً بفصل الملف *views.py* إلى حزمة وجمع دوال العرض هذه في وحدات. عندما تكون في هذه المرحلة، قد يكون حان الوقت لاستخدام المُخطّطات في تطبيقك.

المُخطّطات بشكل أساسي هي مكونات من تطبيقك مُعرفة بشكل مستقل إلى حدٍ ما. تعمل هذه الأجزاء كتطبيقات مستقلة بداخل تطبيقك. فقد تملك مُخطّطات مختلفة للوحة تحكم المشرف، ولواجهة الموقع، وللوحة تحكم المُستخدم. توفر لك هذه التقنية إمكانيّة تجميع دوال العرض، والملفات الساكنة (Static Files)، والقوالب بحسب المكونات، مع السماح لك بمشاركة النماذج، والاستمارات، والجوانب الأخرى من تطبيقك بين هذه المكونات. سنتحدث قريباً حول استخدام المُخطّطات لتنظيم التطبيقات.

الخلاصة

- استخدام الوحدة المفردة لتنظيم تطبيقك هي فكرة جيدة للمشاريع السريعة.
- استخدام حزمة لتنظيم مشروعك هي فكرة جيدة للمشاريع المكونة من دوال عرض، ونماذج، واستمارات، وغيرها من المكونات.
- تعد المُخطّطات طريقة رائعة لتنظيم المشاريع المؤلفة من مكونات مختلفة.

الفصل الرابع: الإعداد



عندما ستبدأ بتعلم فلاسك، ستجد أن الإعداد يبدو سهلاً للغاية. فكل ما عليك فعله هو تعريف بعض المتغيرات في الملف *config.py* وكل شيء سيعمل على ما يرام. هذه السهولة سرعان ما ستزول عندما تبدأ بإدارة الإعدادات لوضع الإنتاج. فقد تحتاج لحماية المفاتيح السرية للواجهات البرمجية أو استخدام ملفات ضبط مختلفة لبيئات مختلفة (بيئة التطوير والإنتاج على سبيل المثال). في هذا الفصل سنتحدث عن بعض ميزات فلاسك المُتقدمة التي تجعل إدارة إعدادات تطبيقك أسهل.

الحالة البسيطة

قد لا تحتاج التطبيقات البسيطة إلى أي من هذه الميزات المعقدة. قد تحتاج فقط لوضع الملف *config.py* في جذر مستودعك واستدعاءه من ملف *app.py* أو من *yourapp/__init__.py* (بحسب طريقة تنظيمك للمشروع).

يجب أن يحوي الملف *config.py* على عملية إسناد قيمة لمتغير في كل سطر. عند استهلال التطبيق، تُستخدم تلك المتغيرات لضبط تطبيق فلاسك وإضافاته، ويمكن الوصول إليها باستخدام القاموس `app.config`، مثال: `app.config["DEBUG"]`.

```
DEBUG = True # تمكين ميزة التنقيح
BCRYPT_LOG_ROUNDS = 12 # Flask-Bcrypt خيار لإضافة
MAIL_FROM_EMAIL = "robert@example.com" # للاستخدام في رسائل البريد
الإلكتروني للتطبيق
```

يُمكن استخدام إعدادات الضبط (التكوين) بواسطة إطار فلاسك، أو الإضافات، أو حتى بواسطتك. في هذا المثال، يمكننا استخدام `app.config["MAIL_FROM_EMAIL"]` كلما احتجنا القيمة الإفتراضية لعنوان "المرسل" لاستخدامها في المعاملات البريدية، مثال: إعادة تعيين كلمة المرور. وضع معلومات كهذه في متغير إعداد يجعل من السهل تغيير قيمتها في المستقبل.

```
# app.py أو app/__init__.py
from flask import Flask

app = Flask(__name__)
app.config.from_object('config')

# app.config["VAR_NAME"] يمكننا الآن الوصول لهذه المتغيرات باستخدام.
```

المتغير	الشرح	نصيحة
DEBUG	يعطيك هذا الخيار بعض الأدوات المفيدة لتنقيح الأخطاء. هذا يتضمن رصة تتبع ووحدة تحكم تفاعلية مبنية على الويب.	ينبغي إسناد القيمة `True` له في وضع التطوير والقيمة `Flase` في وضع الإنتاج.
SECRET_KEY	هذا هو المفتاح السري المستخدم من فلاسك لتوقيع الكعكات. كما يستخدم أيضاً من إضافات مثل Flask-Bcrypt. ينبغي عليك تعريفه داخل المجلد `instance` لإبقاءه خارج نظام التحكم بالإصدارات. يمكنك قراءة المزيد حول مجلد `instance` في	ينبغي أن يُسند لهذا الخيار قيمة عشوائية معقدة.

المتغير	الشرح	نصيحة
BCRYPT_LOG_ROUNDS	الفقرة التالية. إذا كنت تستخدم الإضافة Flask-Bcrypt لتهشير كلمات مرور المستخدمين فعليك تحديد عدد المرات التي سيتم تنفيذ فيها الخوارزمية على كلمة المرور. إن لم تكن تستخدم هذه الإضافة فعليك البدء باستخدامها. فكلما زادت عدد الجولات، زادت معها صعوبة تخمين كلمة المرور من قبل المُهاجم. وينبغي ازدياد عدد الدورات بمرور الوقت بزيادة القدرة الحاسوبية.	سنناقش لاحقاً في الكتاب أفضل الاستخدامات لهذه الإضافة في تطبيقك.

> **تحذير**
> تأكد من تعيين قيمة المتغير `DEBUG` إلى `False` في وضع الإنتاج. فتركه مُفعلاً يسمح للمستخدمين بتشغيل تعليمات بايثون تعسفياً على خادمك.

المجلد الحالِي

ستحتاج أحياناً لتعريف متغيرات تكوين تحتوي معلومات حساسة، وبالتالي ستحتاج لفصل هذه المتغيرات عن المتغيرات الموجودة في الملف *config.py* وإبقائها خارج المستودع. حيث ربما أنت تقوم بإخفاء بعض الأمور السريّة مثل كلمات مرور قاعدة البيانات ومفاتيح الواجهة البرمجية، أو ربما تقوم بتعريف متغيرات مخصصة لجهاز معين. لجعل الأمر سهل، يوفر فلاسك خاصية تدعى **المجلدات الحالِية**. المجلد الحالِي (

instance folder) هو دليل فرعي من المستودع الجذر يحوي على ملفات إعداد مخصصة لوضع التطبيق الحالي. فنحن لا نريد إيداع هذه الملفات في نظام التحكم بالإصدارات.

```
config.py
requirements.txt
run.py
instance/
  config.py
yourapp/
  __init__.py
  models.py
  views.py
  templates/
  static/
```

استخدام المجلد الحالِي

نستخدم التعليمة `app.config.from_pyfile()` لاستدعاء متغيرات التكوين من هذه المجلدات. إذا قمنا بتعيين `instance_relative_config=True` عند إنشاء تطبيقنا باستدعاء الصنف `Flask()` ستقوم التعليمة `app.config.from_pyfile()` بشحن الملف المُحدد من الدليل *instance/*.

```
# app.py أو app/__init__.py

app = Flask(__name__, instance_relative_config=True)
app.config.from_object('config')
app.config.from_pyfile('config.py')
```

الآن أصبح بإمكاننا تعريف المتغيرات في الملف *instance/config.py* كما كنا نفعل في الملف *config.py*. ينبغي عليك أيضاً إضافة المجلد الحالِي إلى قائمة التجاهل الخاصة بنظام التحكم بالإصدارات. للقيام بهذا باستخدام جيت، كل ما عليك هو إضافة `/instance` في الملف *.gitignore*.

المفاتيح السرية

الطبيعة الخاصة للمجلد الحالِي تجعله مكاناً مناسباً لتعريف المفاتيح التي لا تريد تتبعها بواسطة نظام التحكم بالإصدارات. هذه المفاتيح قد تتضمن مفاتيح تطبيقك السرية أو مفاتيح واجهات برمجية لطرف ثالث. هذه الخطوة مهمة جداً خاصةً إن كان تطبيقك مفتوح المصدر، أو سيصبح مفتوح المصدر في مرحلة ما في المستقبل. عادةً نطلب من المستخدمين والمساهمين الآخرين استخدام مفاتيحهم الخاصة.

```
# instance/config.py

SECRET_KEY = 'Sm9obiBTY2hyb20ga2lja3MgYXNz'
STRIPE_API_KEY = 'SmFjb2IgS2FwbGFuLU1vc3MgaXMgYSBoZXJv'
SQLALCHEMY_DATABASE_URI= \
"postgresql://user:TWljaGHFgiBCYXJ0b3N6a2lld2ljeiEh@localhost/datab
asename"
```

الإعداد بناءً على فرق صغير بين البيئات

إذا كان الفرق بين بيئة الإنتاج وبيئة التطوير بسيط فقد ترغب باستخدام المجلد الحالي للتعامل مع تغيير الإعدادات. حيث أن المتغيرات المُعرفة في *instance/config.py* يمكن أن تستبدل قيم تلك المعرفة في *config.py*. كل ما عليك القيام به هو استدعاء `()app.config.from_pyfile` بعد `()app.config.from_object`. إحدى الطرق التي يمكنك الاستفادة فيها من هذا هي تغيير طريقة إعداد تطبيقك على الأجهزة المختلفة.

```
# config.py

DEBUG = False
SQLALCHEMY_ECHO = False


# instance/config.py
DEBUG = True
SQLALCHEMY_ECHO = True
```

بهذه الطريقة، في وضع الإنتاج، يمكننا ترك المتغيرات كما هي في *instance/config.py* فستستبدل قيمتها تلقائياً بتلك المُعرفة في *config.py*.

> **ملاحظة**
>
> – اقرأ المزيد حول مفاتيح إعداد إضافة Flask-SQLAlchemy.

الإعداد بناءً على متغيرات البيئة

المجلد الحالِي لا ينبغي أن يكون في نظام التحكم بالإصدارات. هذا يعني أنك لن تكون قادراً على تتبع التغييرات في ملفات الضبط في هذا المجلد. قد لا يبدو الأمر مهماً لو كنت تملك متغير أو إثنين، ولكن إن كنت تملك إعدادات مضبوطة بدقة لبيئات مختلفة (بيئة الإنتاج، والتهيئة، والتطوير، إلخ.) فأنت بالتأكيد لا تريد حدوث ذلك.

يعطينا فلاسك القدرة على اختيار ملف التكوين بناءً على قيمة متغير في البيئة. هذا يعني أنه يمكننا امتلاك عدة ملفات إعداد في المستودع واختيار المناسب دائماً. طالما أننا نملك عدة ملفات إعداد، فيمكننا نقلهم إلى دليل `config` مخصص لهم.

```
requirements.txt
run.py
config/
  __init__.py # هذا الدليل يأثون أن يُستخدم فقط لإخبار بايثون أن ملف فارغ، حزمة
  default.py
  production.py
  development.py
  staging.py
instance/
  config.py
yourapp/
  __init__.py
  models.py
  views.py
  static/
  templates/
```

في الجدول أدناه يوجد شرح لملفات التكوين المُستخدمة.

ملف التكوين	الشرح
config/default.py	يحوي على القيم الافتراضية لتُستخدم على جميع البيئات أو يتم استبدالها بمتغيرات البيئة المناسبة. على سبيل المثال، قد يتم تعيين `DEBUG = False` في الملف config/default.py ويتم استبداله ب `DEBUG = True` في الملف config/development.py.
config/development.py	يحتوي على القيم التي تُستخدم في عملية التطوير. فقد تضع هنا عنوان قاعدة البيانات ليشير إلى خادم محلي.
config/production.py	يحتوي على القيم التي تستخدم في وضع الإنتاج. فقد تضع هنا عنوان قاعدة البيانات ليشير إلى خادم بعيد، على عكس العنوان المُستخدم في وضع التطوير.
config/staging.py	اعتماداً على عملية النشر، فقد تحتاج لخطوة تهيئة حيث تقوم باختبار التغييرات المجراة على تطبيقك على خادم يحاكي بيئة الإنتاج. على الأرجح ستحتاج لاستخدام قاعدة بيانات مختلفة هنا، كما قد تحتاج لاستبدال قيم أخرى.

نقوم باستدعاء التعليمة `app.config.from_envvar()` لاختيار ملف الإعداد الذي نريد تحميله.

```
# yourapp/__init__.py

app = Flask(__name__, instance_relative_config=True)

# تحميل ملف الإعداد الافتراضي
app.config.from_object('config.default')

# تحميل ملف الإعداد من المجلد الحالِي
app.config.from_pyfile('config.py')

# APP_CONFIG_FILE تحميل ملف الإعداد المحدد بواسطة متغير البيئة
# سيتم استبدال القيم المُعرفة في ملف الإعداد الافتراضي بتلك المعرفة في
ملف الإعداد المُحدد بمتغير البيئة
app.config.from_envvar('APP_CONFIG_FILE')
```

يجب أن تكون القيمة المسندة لمتغير البيئة هي المسار الكامل لملف الإعداد.

تعيين قيمة متغير البيئة يعتمد على المنصة التي يعمل عليها التطبيق. فإذا كنا نعمل على خادم لينُكس يمكننا كتابة ملف نصي (shell script) لتعيين قيمة متغير البيئة وتشغيل الملف *run.py*.

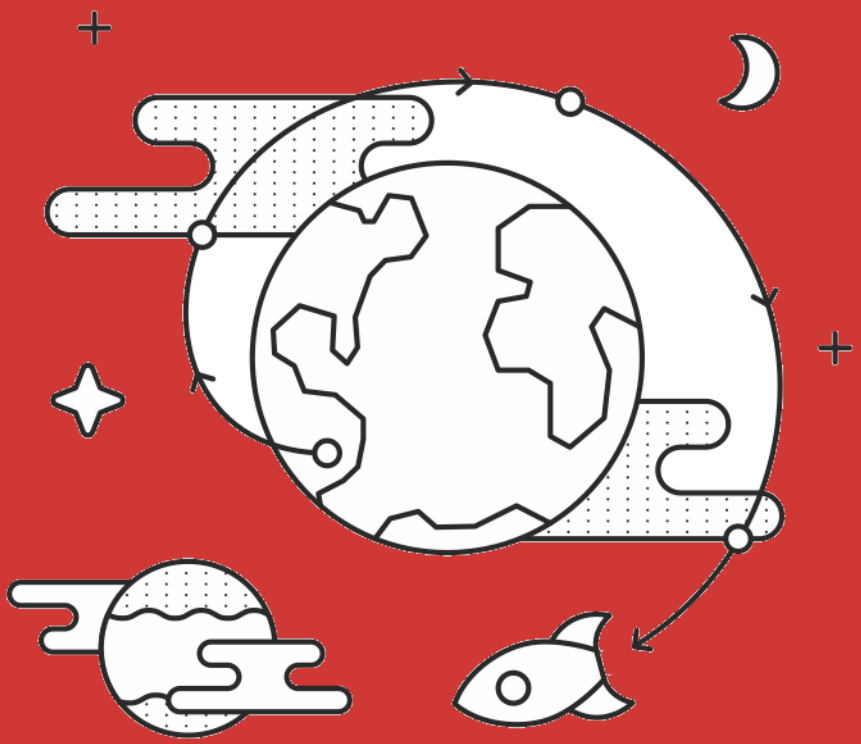
```
# start.sh

export APP_CONFIG_FILE=/var/www/yourapp/config/production.py
python run.py
```

يكون الملف *start.sh* مُتغيّر تبعاً للبيئة. لذلك ينبغي عدم تتبعه بنظام التحكم بالإصدارات. على منصة هيروكو، يمكننا استخدام أداة هيروكو لتعيين قيمة متغيرات البيئة. نفس الشيء يُمكن تطبيقه على منصات PaaS (منصة كخدمة) الأخرى.

الخلاصة

- التطبيق البسيط قد يحتاج فقط إلى ملف إعداد واحد: *config.py*.
- مجلدات الحالة تساعدنا على إخفاء قيم التكوين السرية.
- مجلدات الحالة يُمكن أن تُستخدم لاستبدال إعدادات التطبيق في بيئة محددة.
- يجب استخدام متغيرات البيئة والتعليمة `app.config.from_envvar()` للتعامل مع نظام إعدادات أكثر تعقيداً قائم على نوع البيئة.

الفصل الخامس: استخدامات متقدمة لدوال العرض والتوجيه



مزخرفات بايثون هي دوال تُستخدم لتغيير هيئة (وظيفة أو عمل) دوال أخرى. عندما يتم تنفيذ الدالة المُزخرَفِة، يُنَفَذ المُزخرِف بدلاً منها. يتمكن المُزخرِف من تنفيذ عمليات، أو تعديل المُعطيات (arguments)، أو إيقاف التنفيذ، أو استدعاء الدالة الأصلية (المُزخرَفِة). يمكننا استخدام المُزخرِفات للإحاطة بدوال العرض (الدوال التي تُستخدم لعرض المحتوى) بشيفرة برمجية نود تشغيلها قبل تنفيذ دوال العرض هذه.

```
@decorator_function
def decorated():
    pass
```

إذا قمت باتباع دورة فلاسك الرسميّة فقد تبدو لك صيغة الشيفرة السابقة مألوفة. يُستخدم المُزخرِف `@app.route` لتحديد عناوين دوال العرض في تطبيقات فلاسك.

دعونا نأخذ نظرة على بعض المُزخرِفات الأخرى التي يمكننا استخدامها في تطبيقات فلاسك.

المصادقة

تجعل إضافة Flask-Login من السهل إضافة نظام تسجيل إلى تطبيقك. بالإضافة إلى أنها تُمكِنُك من التعامل مع تفاصيل مصادقة المستخدمين. توفر لنا هذه الإضافة مُزخرِف يقوم بحصر وصول المستخدمين المسجلين فقط إلى دوال عرض معينة وهو المُزخرِف `@login_required`.

```python
# app.py

from flask import render_template
from flask_login import login_required, current_user


@app.route('/')
def index():
    return render_template("index.html")

@app.route('/dashboard')
@login_required
def account():
    return render_template("account.html")
```

> **تحذير**
>
> يجب وضع المُزخرِف `@app.route` في مقدمة المُزخرِفات دائماً (أول مُزخرِف عند استدعاء المُزخرِفات).

فقط المستخدمين المسجلين (المصادقين) سيتمكنون من الولوج إلى الرابط *./dashboard*. يمكننا إعداد الإضافة لتوجه المستخدمين الغير مسجلين إلى صفحة التسجيل، أو لإرجاع رمز الحالة 401، أو للقيام بأي شيء آخر نريده.

> **ملاحظة**
>
> اقرأ المزيد حول استخدام إضافة Flask-Login في توثيقاتها الرسميّة.

التخزين المؤقت

تخيّل أنَّ مقالة ما ذكرت تطبيقاً لك ونُشرِت على موقع CNN وبعض المواقع الإخبارية الأخرى. من المؤكَّد أنَّك ستحصل على آلاف الزيارات في كل ثانية بعد الخبر. وبما أن الصفحة الرئيسية في موقعك تقوم بعدة استعلامات في قاعدة البيانات في كل زيارة، سيؤدي هذا إلى بطء استجابة موقعك. فكيف يمكننا تسريع الأشياء عندها حتى لا يترك كل هؤلاء الزوار موقعنا؟

يوجد العديد من الإجابات الجيدة، ولكن هذا القسم هو حول التخزين المؤقت، لذا دعنا نتحدث عنه. سنستخدم إضافة Flask-Cache للقيام بذلك. توفر لنا هذه الإضافة مُزخرِف يُمكن استخدامه على صفحتنا الرئيسية لتخزين الاستجابة لفترة من الزمن.

يُمكن أن يتم إعداد هذه الإضافة مع باقة من قواعد بيانات التخزين المؤقت المختلفة. الخيار الشعبي هو Redis، والذي هو سهل الاستخدام والتثبيت. لنفترض أن إضافة Flask-Cache تم إعدادها بالفعل مع قاعدة البيانات المُختارة. تُظهِر التعليمات البرمجية أدناه كيف ستبدو دالة العرض المُزخرَفِة خاصتنا.

```python
# app.py

from flask_cache import Cache
from flask import Flask

app = Flask()

# سنقوم هنا بإضافة الإعدادات التي نريدها خلال الاستدعاء
cache = Cache(app)

@app.route('/')
@cache.cached(timeout=60)
def index():
    [...] # هنا يتم القيام ببعض عمليات الاستعلام التي نحتاجها
    return render_template(
        'index.html',
        latest_posts=latest_posts,
        recent_users=recent_users,
        recent_photos=recent_photos
    )
```

الآن ستعمل الدالة فقط كل ستين ثانية (عندما تنتهي مدة التخزين). سيتم حفظ الرد في ذاكرة التخزين المؤقت وعرضه عندما تطرأ أية طلبات (زيارات للصفحة).

> **ملاحظة**
>
> توفر لنا هذه الإضافة أيضاً خاصية لحفظ الدوال، أو بمعنى آخر، تخزين نتيجة دالة يتم استدعاءها بمعاملات محددة. يمكنك حتى تخزين قوالب جينجا.

مُزخرِفات مُخصصة

دعنا نتخيل أنك تملك تطبيق يفرض رسوماً على المستخدمين كل شهر. إذا إنتهت صلاحية حساب المستخدم سنقوم بتوجيهه إلى صفحة الدفع ونعلمه أنه عليك ترقية حسابه.

```
# myapp/util.py

from functools import wraps
from datetime import datetime

from flask import flash, redirect, url_for

from flask_login import current_user

def check_expired(func):
    @wraps(func)
    def decorated_function(*args, **kwargs):
        if datetime.utcnow() > current_user.account_expires:
            flash("Your account has expired. Update your billing
info.")
            return redirect(url_for('account_billing'))
        return func(*args, **kwargs)

    return decorated_function
```

السطر الشرح

10 عندما يتم زخرفة دالة باستخدام `@check_expired`، سيتم استدعاء `()` `check_expired` وستمرر الدالة المُزخرَفِة كمعامل.

11 يوجد في هذا السطر مُزخرِف (`@warps`) يقوم ببعض الأمور حتى تبدو الدالة `()` `decorated_function` كالدالة `func()` لأغراض توثيقية وتنقيحية. هذا يجعل من سلوك الدالة أكثر طبيعيةً بقليل.

12 ستقوم الدالة `decorated_function` بجمع كل المعاملات المفتاحية (kwargs) والمعاملات الصفيّة (args) المُمررة لدالة العرض الأصليّة `func()`. في هذا السطر سنقوم بالتأكد إذا كان حساب المستخدم مُنتهي الصلاحية. إذا كان كذلك، سنرسل رسالة له ونقوم بإعادة توجيهه إلى صفحة الدفع.

عندما نرص (نُكدِس) المُزخرِفات، المُزخرِف الأعلى (الذي في المقدمة) هو من يعمل أولاً، ومن ثم يقوم باستدعاء الذي يليه بالسطر، سواءً كان دالة عرض أو مُزخرِف آخر. الصيغة الكتابية للمُزخرِفات تحوي القليل من التجميل اللغوي (اقرأ المزيد عن التجميل اللغوي).

```
# هذه التعليمات البرمجية:
@foo
@bar
def one():
    pass

r1 = one()

# هل هذه نفس التعليمة:
def two():
    pass

two = foo(bar(two))
r2 = two()

r1 == r2 # النتيجة ستكون صحيحة منطقياً (True)
```

الشيفرة أدناه تعرض مثال حول استخدام المُزخرِف المخصص الذي قمنا بإنشائه مع المُزخرِف `@login_required` من الإضافة Flask-Login. نستطيع استخدام كلا المُزخرِفين عن طريق رصهم (تكديسهم) أسفل بعضهم البعض.

```
# myapp/views.py

from flask import render_template

from flask_login import login_required

from . import app
from .util import check_expired

@app.route('/use_app')
@login_required
@check_expired
def use_app():
    """Use our amazing app."""
    # [...]
    return render_template('use_app.html')

@app.route('/account/billing')
@login_required
def account_billing():
    """Update your billing info."""
    # [...]
    return render_template('account/billing.html')
```

عندما يحاول المستخدم الآن الدخول إلى */use_app*، سيتأكد المُزخرِف `()check_expired` قبل تشغيل دالة العرض أن حساب هذا المستخدم غير منتهي الصلاحية.

> **ملاحظة**
>
> اقرأ المزيد حول ما تقوم به الدالة `warps()` في توثيقات بايثون.

محولات الروابط

المحولات القياسية (المبنية في الإطار)

عندما تقوم بتعريف موجه بفلاسك، يمكنك تحديد أجزاء من الرابط ليتم تحويلها إلى متغيرات وتمريرها كمعاملات إلى دالة العرض.

```
@app.route('/user/<username>')
def profile(username):
    pass
```

أي شيء سيتم كتابته مكان الجزء في العنوان المسمى بـ `<username>` سيتم تمريره إلى دالة العرض كمعامل يسمى username. يمكنك أيضاً تحديد محول لتصفية (تحويل قيمة) المتغير قبل تمريره إلى دالة العرض.

```
@app.route('/user/id/<int:user_id>')
def profile(user_id):
    pass
```

في الشيفرة أعلاه، الدخول إلى العنوان *http://myapp.com/user/id/Q29kZUxlc3NvbiEh* سيعيد رمز الحالة 404 (أي الصفحة غير موجودة). هذا لأن الجزء الأخير من الرابط الذي كان من المفترض أن يكون رقماً صحيحاً تم إدخال مكانه سلسلة نصية.

يمكننا أيضاً تعريف دالة عرض أخرى تستخدم محوِّل يبحث عن سلسلة نصيّة. يمكن استدعاء تلك الدالة عبر الرابط */user/id/Q29kZUxlc3NvbiEh/*، بينما الأولى ينبغي أن تُستدعى عبر */user/id/124*.

يظهر الجدول أدناه محولات الروابط القياسية الموجودة في إطار فلاسك.

المحول	الشرح
string	يقبل أي مُدخل نصي بدون مائلة (المحول الافتراضي).
int	يقبل الأعداد الصحيحة.
float	مثل المحول int ولكنه يقبل القيم ذات الفاصلة العشرية.
path	مثل المحول string ولكنه يقبل المائلة.

محولات مخصصة

يمكننا أيضاً صنع محولات خاصة لتناسب احتياجاتنا. على موقع ريديت (موقع مشهور لمشاركة الروابط) يصنع المستخدمون والمدراء مجتمعات للمناقشة ومشاركة الروابط بناءً على الموضوع. بعض الأمثلة حول روابط المجتمعات: */r/python* و */r/flask*. يوجد ميزة مثيرة في هذا الموقع وهي أنه يمكنك عرض منشورات من مجتمعين مختلفين عن طريق الفصل بين أسماء المجتمعات بإشارة مساواة في الرابط، على سبيل المثال: *.reddit.com/r/python+flask*

يمكننا استخدام محول مخصص لإضافة هذه الميزة إلى تطبيقنا. سنأخذ عدداً عشوائياً من العناصر المفصولة بعلامة زائد، وسنقوم بتحويلها إلى قائمة باستخدام الصنف `ListConverter` وسنمرر هذه القائمة إلى دالة العرض.

```
# myapp/util.py

from werkzeug.routing import BaseConverter

class ListConverter(BaseConverter):

    def to_python(self, value):
        return value.split('+')

    def to_url(self, values):
        return '+'.join(BaseConverter.to_url(value)
                        for value in values)
```

في المثال أعلاه، قمنا بتعريف دالتين: `to_python()` و `to_url()`. كما يشير الاسم، تُستخدم الدالة `to_python()` لتحويل الرابط إلى كائن بايثون ليتم تمريره إلى دالة العرض، وتُستخدم الدالة `to_url()` من قبل `url_for()` لتحويل المعاملات إلى رابط.

لاستخدم المحول `ListConverter`، علينا بدايةً إخبار فلاسك أنه موجود.

```
# /myapp/__init__.py

from flask import Flask

app = Flask(__name__)

from .util import ListConverter

app.url_map.converters['list'] = ListConverter
```

> **ملاحظة**
>
> هناك احتمالية للوقوع ببعض مشاكل الاستيراد الحلقية إذا احتوى ملف util خاصتك على السطر `from . import app`. لهذا السبب انتظرت حتى يتم استهلال التطبيق حتى أقوم باستيراد `ListConverter`.

الآن يمكننا استخدام المحول الذي قمنا بإنشائه بالطريقة نفسها التي نستخدم بها المحولات القياسية (المبنية في الإطار). قمنا بتحديد نوع المفتاح في القاموس على أنه "قائمة" لذلك المثال أدناه يوضح كيف يمكننا استخدام هذا المحول في المُزخرِف `()` `@app.route`.

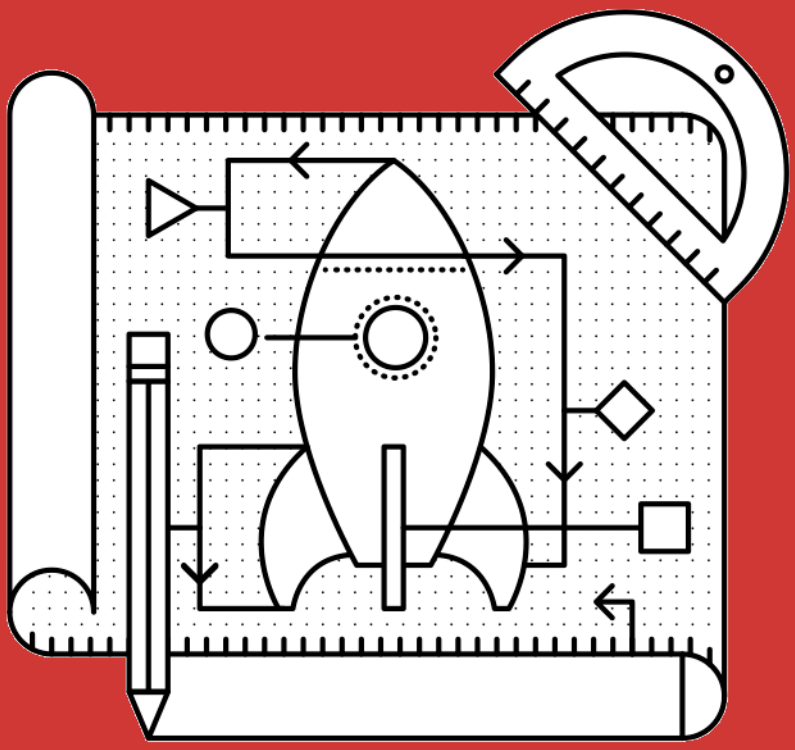
```
# myapp/views.py

from . import app

@app.route('/r/<list:subreddits>')
def subreddit_home(subreddits):
    """Show all of the posts for the given subreddits."""
    posts = []
    for subreddit in subreddits:
        posts.extend(subreddit.posts)

    return render_template('/r/index.html', posts=posts)
```

يجب أن يعمل المثال أعلاه كما في موقع ريديت. يُمكن استخدام نفس الطريقة لصنع أي محول روابط نريده.

الخلاصة

- يساعدنا المُزخرِف `@login_required` في الإضافة Flask-Login على حصر وصول المستخدمين المسجلين فقط لبعض دوال العرض.
- توفر الإضافة Flask-Cache مجموعة من المُزخرِفات لتنفيذ طرق متنوعة من التخزين المؤقت.
- يُمكننا تطوير مُزخرِفات عرض مُخصصة لتساعدنا على تنظيم الشيفرة البرمجية وللالتزام بمبدأ "لا تكرر نفسك".
- يُمكن أن تكون محولات الروابط المخصصة طريقة رائعة لتنفيذ ميزات إبداعية متعلقة بالروابط.

الفصل السادس: المُخطّطات



ما هي المُخطّطات؟

يُعرِّف المُخطّط مجموعة من دوال العرض، والقوالب، والملفات الساكنة (Static Files) وغيرها من العناصر التي يُمكن استخدامها في التطبيق. على سبيل المثال، دعنا نتخيل أننا نملك مُخطّط للوحة تحكم مشرف. هذا المُخطّط سيُعرِّف دوال عرض لموجهات مثل *admin/login/* و *admin/dashboard/*. قد يُضمِّن هذا المُخطّط أيضاً القوالب والملفات الساكنة التي سنستخدمها في هذه الموجهات. يمكننا بعد ذلك استخدام المُخطّط لإضافة لوحة التحكم تلك لتطبيقنا، سواءً كان هذا التطبيق شبكة تواصل اجتماعي لرواد الفضاء أو موقع لإدارة مبيعات الصواريخ.

لِمَ عليك استخدام المُخطّطات؟

حالة الاستخدام الأكثر شعبية للمُخطّطات هي تنظيم التطبيق إلى مكونات مستقلة. فمثلاً لموقع شبيه بتويتر، قد نحتاج لإنشاء مُخطّط لصفحات الموقع (مثل صفحة *index.html* و *about.html*). ومن ثم سنحتاج إلى مُخطّط آخر من أجل لوحة تحكم المستخدم المُسجل حيث ستعرض جميع المنشورات الجديدة هناك، كما سنحتاج لواحد آخر من أجل لوحة تحكم المُشرِف. حيث كل قسم مستقل من الموقع يُمكِن أن يُجزَّأ إلى قسم مستقل من التعليمات البرمجية أيضاً. مما يتيح لنا تنظيم تطبيقنا وتقسيمه إلى "تطبيقات" مُصغرة كلٌ منها يؤدي وظيفة معينة.

> **ملاحظة**
>
> يمكنك قراءة المزيد حول فوائد استخدام المُخطّطات في قسم "لِمَ المُخطّطات" في التوثيق الرسمي لإطار فلاسك.

أين يجب أن توضع هذه المُخطّطات؟

ككل شيء في فلاسك، يوجد العديد من الطرق لتنظيم التطبيق باستخدام المُخطّطات. باستخدام هذه التقنية يمكننا الاختيار ما بين استخدام التنظيم الوظيفي أو التقسيمي.

التنظيم الوظيفي

في التنظيم الوظيفي، نقوم بتنظيم أجزاء التطبيق بناءً على ما تفعله. بحيث تُجمَع القوالب في دليل، والملفات الساكنة في آخر، ودوال العرض في ثالث.

```
yourapp/
    __init__.py
    static/
    templates/
        home/
        control_panel/
        admin/
    views/
        __init__.py
        home.py
        control_panel.py
        admin.py
    models.py
```

باستثناء الملف *yourapp/views/__init__.py*، كل ملف يحمل اللاحقة *.py* في الدليل */yourapp/views* (كما هو موضح أعلاه) يُمَثِل مُخطّط. في الملف *yourapp/__init__.py* سنقوم باستيراد هذه المُخطّطات و**تسجيلهم** في الكائن `Flask()`. سنتعمق أكثر قليلاً حول كيفية تطبيق الطريقة التنظيمية هذه لاحقاً في هذا الفصل.

> **ملاحظة**
>
> في أثناء كتابة هذه الكلمات، يستخدم موقع فلاسك هذه الهيكلية التنظيمية. يمكنك أخذ نظرة بنفسك في موقع جيتهاب.

التنظيم التقسيمي

في التنظيم التقسيمي، نقوم بتنظيم أجزاء التطبيق بناءً على أي جزء من التطبيق تساهم فيه (تدخل في تركيبته). بحيث جميع القوالب، ودوال العرض، والملفات الساكنة الخاصة بلوحة تحكم المُشرِف ستوضع في دليل واحد، ونظيراتها الخاصة بلوحة تحكم المستخدم ستوضع في دليل آخر.

```
yourapp/
    __init__.py
    admin/
        __init__.py
        views.py
        static/
        templates/
    home/
        __init__.py
        views.py
```

```
        static/
        templates/
    control_panel/
        __init__.py
        views.py
        static/
        templates/
    models.py
```

باستخدام التنظيم التقسيمي (كما هو موضح أعلاه)، كل دليل مُحتوى بداخل الدليل *yourapp/* يُمثِل مُخطّطاً مُنفصِلاً. جميع المُخطّطات سيتم تسجيلها في الملف *__init__.py* (الموجود في الدليل الجذر).

أيهم أفضل؟

يعد اختيار الهيكلية التنظيمية لمشروعك خيار شخصي بحت. فالفرق الوحيد هو بطريقة تمثيل التسلسل الهرمي - أي يمكنك هندسة (تنظيم) التطبيق بكلا الطريقتين - لذلك يجب أن تختار الطريقة الأكثر منطقية بالنسبة لك.

إذا كان تطبيق يملك أجزاء مستقلة والتي تتشارك فقط أشياء مثل النماذج (Models) والإعدادات فقد تكون الطريقة التقسيمية هي المناسبة. دعنا نأخذ تطبيق (SaaS) يتيح للمستخدمين بناء مواقع على سبيل المثال. يمكنك إنشاء مُخطّطات للصفحة الرئيسية، ولوحة التحكم، وموقع المستخدم، ولوحة تحكم المُشرِف وتنظيمها تقسيمياً. هذه الأقسام قد تملك ملفات ساكنة ونماذج (layouts) مختلفة عن بعضها لحدٍ كبير. إذا كنت تفكر

بفصل هذه المُخطّطات وتحويلها إلى إضافات أو استخدامها في مشاريع أخرى فاستخدام التنظيم التقسيمي سيكون أفضل وأسهل بكثير.

من ناحية أخرى، إذا كانت مكونات تطبيقك تتشارك معاً أكثر بقليل فمن الأفضل تمثيلها باستخدام الهيكلية الوظيفية. لنأخذ الفيس بوك على سبيل المثال. إذا كان الفيس بوك يستخدم فلاسك فمن الممكن أن يكون هناك مُخطّط للصفحات الساكنة (أي للصفحة الرئيسية، ولصفحة التسجيل، ولصفحة حول، ... إلخ.)، وللوحة التحكم (أي لصفحة آخر الأخبار)، ولصفحة الملفات الشخصية (*/robert/about* و */robert/photos*)، ولصفحة الإعدادات (*/settings/security* و */settings/privacy*)، وللعديد من الصفحات الأخرى. المكونات السابقة تشترك بالتخطيط (layout) وبالتنسيقات العامة، ولكن لكل منها تخطيطها (تصميمها) الخاص أيضاً. المثال أدناه يوضح كيف ستبدو النسخة المُختصرة من فيس بوك لو كان مبني بإطار فلاسك.

```
facebook/
    __init__.py
    templates/
        layout.html
        home/
            layout.html
            index.html
            about.html
            signup.html
            login.html
        dashboard/
            layout.html
```

```
            layout.html
            news_feed.html
            welcome.html
            find_friends.html
        profile/
            layout.html
            timeline.html
            about.html
            photos.html
            friends.html
            edit.html
        settings/
            layout.html
            privacy.html
            security.html
            general.html
    views/
        __init__.py
        home.py
        dashboard.py
        profile.py
        settings.py

    static/
        style.css
        logo.png
    models.py
```

ستكون المُخطّطات في الدليل *facebook/views/* مجموعة من دوال العرض بدلاً من مكونات مستقلة كاملة. ستستخدم معظم دوال العرض الموجودة في المُخطّطات نفس

الملفات الساكنة. ستستخدم (ستستمد) معظم القوالب النموذج الرئيسي (الموجود في الدليل *templates/* باسم *layout.html*). لذلك الهيكلة التنظيمية وسيلة رائعة لتنظيم المشاريع المشابهة.

كيف تُستخدَم المُخطّطات؟

الاستخدام المبدئي

دعنا نلقي نظرة على شيفرة إحدى المُخطّطات من مثال "الفيس بوك" السابق.

```
# facebook/views/profile.py

from flask import Blueprint, render_template

profile = Blueprint('profile', __name__)

@profile.route('/<user_url_slug>')
def timeline(user_url_slug):
    # ننفذ بعض الأشياء هنا
    return render_template('profile/timeline.html')

@profile.route('/<user_url_slug>/photos')
def photos(user_url_slug):
    # ننفذ بعض الأشياء هنا
    return render_template('profile/photos.html')
```

```
@profile.route('/<user_url_slug>/about')
def about(user_url_slug):
    # ننفذ بعض الأشياء هنا
    return render_template('profile/about.html')
```

لإنشاء كائن المُخطّط، نقوم باستيراد الصنف `Blueprint()` ونقوم بتهيئته بالمعاملات `name` و `import_name`. عادةً تُسنَد القيمة `__name__` للمعامل `import_name` وحسب. تُعَد هذه القيمة متغيّر بايثون من النوع الخاص يحوي اسم الوحدة الحالية.

لقد استخدمنا في مثال "الفيس بوك" الهيكلة الوظيفية. فإذا كنا استخدمنا الهيلكة التقسيمية لكُنّا أخبرنا فلاسك أنَّ المُخطّط يمتلك دليل للملفات الساكنة وآخر للقوالب خاص به. تظهر الشيفرة البرمجية أدناه كيف يتم ذلك.

```
profile = Blueprint('profile', __name__,
                    template_folder='templates',
                    static_folder='static')
```

بعد أن قمنا بتعريف المُخطّط، يجب علينا تسجيله في التطبيق.

```
# facebook/__init__.py

from flask import Flask
from .views.profile import profile

app = Flask(__name__)
app.register_blueprint(profile)
```

الآن الموجهات المُعرَفِة في الملف *facebook/views/profile.py* (على سبيل المثال `/<user_url_slug>`) أصبحت مُسجلة في التطبيق وستعمل كما لو كانت مُعرفِة باستخدام المُزخرِف `@app.route()`.

استخدام بادِئة روابط ديناميكية

بالإكمال مع مثال الفيس بوك، نلاحظ كيف أن جميع الموجهات المتعلقة بالملف الشخصي تبدأ بالجزء `<user_url_slug>` وتقوم بتمرير تلك القيمة لدالة العرض. نريد الآن أن يكون المستخدم قادراً على الدخول للملف الشخصي عن طريق الذهاب إلى عنوان مثل *https://facebo-ok.com/john.doe*. يمكننا التوقف عن تكرار ذلك الجزء في الموجهات عن طريق تعريف بادئة ديناميكية لجميع موجهات المُخطّط.

تتيح لنا المُخطّطات تعريف بادئات ثابتة وديناميكية. حيث يمكننا إخبار فلاسك أن جميع الموجهات في المُخطّط ينبغي أن تبدأ بـ */profile* على سبيل المثال، وبالتالي تسمى هذه البائدة بالثابتة. في حالة مثال الفيس بوك، ستتغير البادئة بناءً على الملف الشخصي الذي يريد المستخدم تصفحه. بحيث أياً كان النص الذي سيختاره فهو سبيكة رابط الملف الشخصي الذي يجب أن نعرضه، وبالتالي تسمى هذه بالبادئة الديناميكية.

الخيار متروك لنا في مكان تعريف البادئة. حيث يمكننا تعريفها في موضع من إثنين: إما عندما نقوم بتهيئة الصنف `Blueprint()` أو عندما نقوم بتسجيله في `()` `app.register_blueprint`.

```
# facebook/views/profile.py

from flask import Blueprint, render_template

profile = Blueprint('profile', __name__,
url_prefix='/<user_url_slug>')

# [...]
```

```
# facebook/__init__.py

from flask import Flask
from .views.profile import profile

app = Flask(__name__)
app.register_blueprint(profile, url_prefix='/<user_url_slug>')
```

في حين أنه لا توجد أية حدود تقنيَّة لكلتا الطريقتين، فمن الجميل أن تكون جميع البادئات المتوفرة في نفس الملف. هذا يجعل من السهل التحكم بالأشياء من مستوى عالٍ (مكان واحد للتحكم بكل البادئات). لهذا السبب، أنصح بإعداد البادِئات عند تسجيل المُخطّطات.

يمكننا استخدام المحولات لإنشاء بادئة ديناميكية، كما كُنَّا نفعل عند استدعاء المُزخرِف `route()`. هذا يتضمن أي محول مُخصص قمنا بإنشاءه. عند استخدام المحولات، يمكننا معالجة القيمة المُعطاة قبل تسليمها إلى دالة العرض. في هذه الحالة سنحتاج إلى تحديد المستخدم بناءً على سبيكة العنوان المُمررة إلى المُخطّط الخاص بالملفات الشخصية.

يمكننا القيام بهذا عن طريقة زخرَفِة دالة باستخدام `url_value_preprocessor()`.

```
# facebook/views/profile.py

from flask import Blueprint, render_template, g

from ..models import User

# facebook/__init__.py البادئة معرفة عند التسجيل في ملف
profile = Blueprint('profile', __name__)

@profile.url_value_preprocessor
def get_profile_owner(endpoint, values):
    query = User.query.filter_by(url_slug=values.pop('user_url_slug'))
    g.profile_owner = query.first_or_404()

@profile.route('/')
def timeline():
    return render_template('profile/timeline.html')

@profile.route('/photos')
def photos():
    return render_template('profile/photos.html')
```

```
@profile.route('/about')
def about():
    return render_template('profile/about.html')
```

استخدمنا الكائن `g` لتخزين مالك الملف الشخصي. يتوفر الكائن `g` في سياق قالب (Template Context) جينجا 2. مما يعني أن كل ما علينا القيام به هو عرض القالب وستكون المعلومات التي نحتاجها فيه.

```
{# facebook/templates/profile/photos.html #}

{% extends "profile/layout.html" %}

{% for photo in g.profile_owner.photos.all() %}
    <img src="{{ photo.source_url }}" alt="{{ photo.alt_text }}" />
{% endfor %}
```

> **ملاحظة**
>
> يوجد في توثيقات فلاسك درس رائع حول استخدام البادئات لتدويل الروابط.

استخدام نطاق فرعي ديناميكي

توفر العديد من تطبيقات SaaS (برمجية كخدمة) هذه الأيام نطاق فرعي لمستخدميها ليتمكنوا من الوصول لبرمجياتهم. تطبيق هارفست (Harvest) على سبيل المثال، والذي هو تطبيق مراقبة (تتبع) الوقت للاستشاريين، يسمح لك بالوصول للوحة التحكم من

الرابط yourname.harvestapp.com. سنعرض في هذا القسم كيفية التعامل مع نطاق فرعي موَّلد تلقائياً كهذا.

ساستخدم في هذا الفصل مثال لتطبيق يتيح لمستخدميه إنشاء مواقع خاصة بهم. تخيل أن تطبيقنا ذلك فيه ثلاث مُخطّطات لأقسام مستقلة: الصفحة الرئيسية حيث يلِج (يقومون بتسجيل الدخول) المستخدمون، ولوحة تحكم المستخدم حيث يبني المستخدمون مواقعهم وحيث تُستضاف. وبما أن هذه الأقسام غير مرتبطة ببعضها، فسنقوم بتنظيم الموقع باستخدام الهيكلة التقسيمية.

```
sitemaker/
    __init__.py
    home/
        __init__.py
        views.py
        templates/
            home/
        static/
            home/
    dash/
        __init__.py
        views.py
        templates/
            dash/
        static/
            dash/
```

```
site/
    __init__.py
    views.py
    templates/
        site/
    static/
        site/
models.py
```

يشرح الجدول أدناه المُخطّطات المُستخدمة في هذا التطبيق.

الرابط	الموجه	الشرح
sitemaker.com	*sitemaker/home*	مُخطّط عادي. يحوي دوال العرض، والقوالب، والملفات الساكنة لصفحة *index.html*، و *about.html* ولصفحة *pricing.html*.
bigdaddy.sitemaker.com	*sitemaker/site*	مُخطّط يستخدم نطاق فرعي ديناميكي ويُضمِّن عناصر موقع المُستخدم. سنتكلم لاحقاً عن التعليمات المُستخدمة لتنفيذ هذا المُخطّط.
bigdaddy.sitemaker.com/admin	*sitemaker/dash*	يُمكِن أن يستخدم هذا المُخطّط كلاً من النطاق الفرعي الديناميكي وبادئة الرابط من خلال دمج التقنيات في هذا القسم مع التقنيات في القسم السابق.

نستطيع تعريف النطاق الفرعي بنفس الطريقة التي عرفنا بها البوادئ سابقاً. كلا الخيارين متاحين (سواءً عن طريق التعريف في دليل المُخطّط أو في الملف *__init__.py* الموجود في الدليل الجذر)، ولكن مجدداً سنقوم بالتعريف في الملف *sitemaker/__init__.py*.

```
# sitemaker/__init__.py

from flask import Flask
from .site import site

app = Flask(__name__)
app.register_blueprint(site, subdomain='<site_subdomain>')
```

وفي حين أننا نستخدم الهيكلية التقسيمية، فسنقوم بتعريف المُخطّط في الملف *sitemaker/site/__init__.py*.

```
# sitemaker/site/__init__py

from flask import Blueprint

from ..models import Site

site = Blueprint('site', __name__)

@site.url_value_preprocessor
def get_site(endpoint, values):
    query =
Site.query.filter_by(subdomain=values.pop('site_subdomain'))
    g.site = query.first_or_404()

from . import views
```

الآن نحن نملك المعلومات من قاعد البيانات التي سنستخدمها لعرض موقع المستخدم للزوار الذين يقومون بالدخول إلى النطاق الفرعي.

ليتمكن فلاسك من العمل مع النطاقات الفرعية، يجب عليك تعريف متغير الإعداد SERVER_NAME.

```
# config.py

SERVER_NAME = 'sitemaker.com'
```

> **ملاحظة**
> منذ بضعة دقائق، عندما كنت أصيغ هذا القسم، قال أحدهم في قناة IRC أن نطاقاته الفرعية كانت تعمل في وضع التطوير ولكنها توقفت عن العمل في وضع الإنتاج. سألته عمَّا إذا قام بتعريف المتغير `SERVER_NAME`، فتبيّن أنه كان قد عرَّف هذا المتغير في وضع التطوير ولكنه لم يفعل ذلك في وضع الإنتاج. وبعدها قام بتعريفه في وضع الإنتاج وانتهت المشكلة.
>
> يمكنك رؤية المحادثة التي دارت بيني (اسمي imrobert في السجل) وبين aplavin (الاسم المُستعار لصاحب الاستفسار) من هنا.
>
> كانت هذه المصادفة كافية لجعلي أظن أنه يجب إدراج هذه النقطة في الفصل.
>
> **ملاحظة**
> يمكنك استخدام كلاً من النطاق الفرعي والبادئة معاً. فكَّر في كيف يمكننا

> إعداد المُخطّط في الدليل *sitemaker/dash* ليستخدم بنيَّة الرابط المبيّنة في الجدول السابق.

إعادة هيكلة (تنظيم) التطبيقات الصغيرة لتستخدم المُخطّطات

أود أن أتقدم بمثال مُختصَر عن الخطوات التي يمكننا اتخاذها لتحويل أحد التطبيقات ليستخدم المُخطّطات. سنبدأ مع إعادة هيكلة تطبيق فلاسك نموذجي (عادي).

```
config.txt
requirements.txt
run.py
U2FtIEJsYWNr/
  __init__.py
  views.py
  models.py
  templates/
  static/
tests/
```

يحتوي ملف *views.py* على ما يزيد عن 10,000 سطر من التعليمات البرمجية! لقد قمنا بتأخير عملية إعادة الهيكلة، ولكن الوقت قد حان. يحتوي الملف على دوال العرض لكل جزء من الموقع. الأقسام هي: الصفحة الرئيسية، ولوحة تحكم المُستخدم، ولوحة تحكم المُشرِف، والواجهة البرمجية (API)، ومدونة الشركة.

الخطوة الأولى: هل نستخدم الهيكلة الوظيفية أم التقسيمية؟

التطبيق مصنوع من أجزاء منفصلة عن بعضها بشكل كبير. حيث أنَّ القوالب والملفات الساكنة من المُرجَّح أنها لن يتُم مُشاركتُها بين لوحة تحكم المُستخدم ومدونة الشركة، على سبيل المثال. لذلك سنستخدم الهيكلة التقسيمية.

الخطوة الثانية: نقل بعض الملفات

> **تحذير**
> قم بإيداع كل شيء إلى نظام التحكم بالإصدارات قبل القيام بأية تغييرات في تطبيقك، فأنت لا تريد حذف أي شيء عن طريق الخطأ.

سنقوم الآن بإنشاء هيكلية الدلائل لتطبيقنا الجديد. يمكننا البدء بإنشاء مجلد لكل مُخطّط بداخل دليل الحزمة. ومن ثم سنقوم بنسخ الملف *views.py* والدليل */static* و */templates* إلى كل دليل من الذين أنشأناهم. وبعدها يمكننا حذفهم من جذر دليل الحزمة.

```
config.txt
requirements.txt
run.py
U2FtIEJsYWNr/
  __init__.py
  home/
    views.py
```

```
    static/
    templates/
  dash/
    views.py
    static/
    templates/
  admin/
    views.py
    static/
    templates/
  api/
    views.py
    static/
    templates/
  blog/
    views.py
    static/
    templates/
  models.py
tests/
```

الخطوة الثالثة: الدخول في الجديَّة

الآن سندخل إلى كل مُخطّط وسنحذف دوال العرض والقوالب والملفات الساكنة الغير متعلقة بالمُخطّط. هذه الخطوة معتمدة بشكل كبير على كيفية قيامك بتنظيم تطبيقك منذ البداية.

في النهاية يجب أن يحتوي كل مُخطّط على ملف *views.py* يحتوي جميع دوال العرض المتعلقة بالمُخطّط. لا ينبغي أن يُعرِف مُخطّطان دالة عرض لنفس المُوجِه. كل دليل/

templates يجب أن يحتوي فقط على القوالب الخاصة بدوال العرض في المُخطّط. كل دليل *static/* يجب أن يحتوي فقط على الملفات الساكنة التي ستُعرَض بواسطة هذا المُخطّط.

> **ملاحظة**
>
> قم في هذه المرحلة بإزالة جميع الاستيرادات الغير ضرورية. فمن السهل نسيانهم، ولكن في أفضل الأحوال سيقومون بتلويث شيفرتك وفي أسوئها سيقومون بإبطاء تطبيقك.

الخطوة الرابعة: تعريف المُخطّطات

في هذه المرحلة سنقوم بتحويل الدلائل التي أنشأناها إلى مُخطّطات. بدايةً، دعنا نلقي نظرة على تعريف مُخطّط الواجهة البرمجية.

```
# U2FtIEJsYWNr/api/__init__.py

from flask import Blueprint

api = Blueprint(
    'site',
    __name__,
    template_folder='templates',
    static_folder='static'
)

from . import views
```

بعد ذلك يمكننا تسجيل هذا المُخطّط في الملف *__init__.py* الجذر للحزمة U2FtIEJsYWNr.

```
# U2FtIEJsYWNr/__init__.py

from flask import Flask
from .api import api

app = Flask(__name__)

# ننشئ نطاق فرعي لوضع الواجهة البرمجية للتطبيق على الرابط
api.U2FtIEJsYWNr.com
app.register_blueprint(api, subdomain='api')
```

تأكد أن جميع الموجهات الآن مُسجلة في المُخطّط بدلاً من الكائن `app` (أو أياً يكن ما سميت كائن التطبيق).

```
# U2FtIEJsYWNr/api/views.py

from . import api

@api.route('/search')
def search():
    pass
```

```
# U2FtIEJsYWNr/views.py

from . import app

@app.route('/search', subdomain='api')
def api_search():
    pass
```

الخطوة الخامسة: استمتع

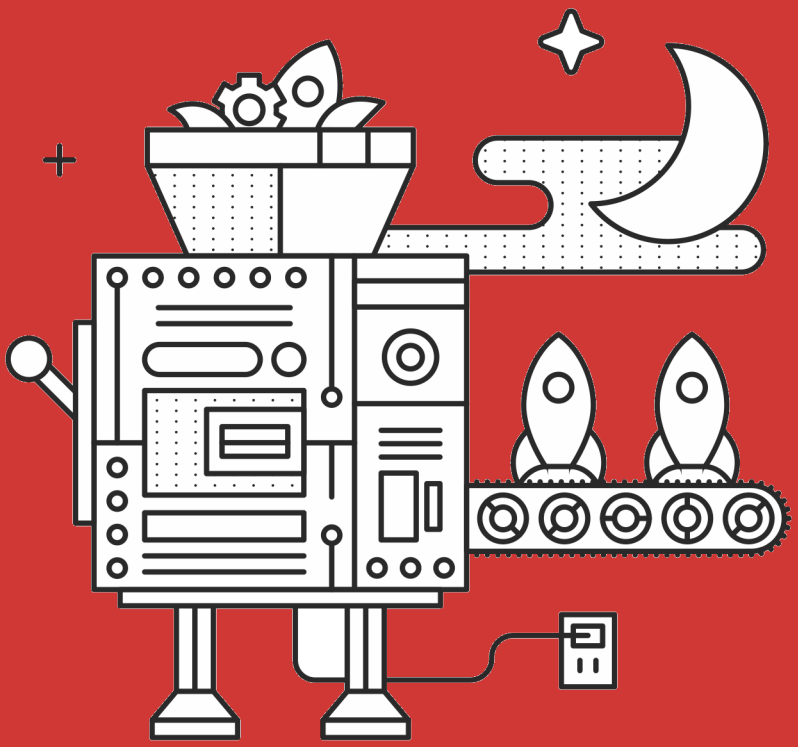
الآن تطبيقنا أكثر معياريَّة بكثير مما كان عليه عندما كان يستخدم ملف views.py واحد ضخم. تعريفات الموجهات الآن أكثر بساطة؛ لأنه بإمكاننا جمعها معاً في مُخطّطات وإعداد أشياء مثل النطاقات الفرعية وبوادئ العنواين في كل مُخطّط.

الخلاصة

- المُخطّطات هي مجموعة من دوال العرض، والقوالب، والملفات الساكنة، وغيرها من الملحقات الأخرى التي يمكن إضافتها للتطبيق.
- تعد المُخطّطات طريقة رائعة لتنظيم تطبيقك.
- في الهيكلية التقسيمية، كل مُخطّط هو مجموعة من دوال العرض، والقوالب، والملفات الساكنة التي تُشكِل قسم معين من تطبيقك.
- في الهيكلية الوظيفية، كل مُخطّط هو مجرد مجموعة من دوال العرض. فجميع القوالب توضع معاً، كما والملفات الساكنة.

- لتستخدم مُخطّط، عليك تعريفه ومن ثم تسجيله في التطبيق عن طريق استدعاء `Flask.register_blueprint()`.
- يمكنك تعريف بوادئ روابط ديناميكية والتي سيتم تطبيقها على جميع موجهات المُخطّط.
- يمكنك أيضاً تعريف نطاق فرعي ديناميكي لجميع الموجهات في المُخطّط.
- يتم إعادة هيكلة تطبيق لاستخدام المُخطّطات في خمس خطوات صغيرة نسبياً.

الفصل السابع: القوالب



في حين أنَّ فلاسك لا يجبرك على استخدام أيَّة لغة قوالب محددة، فهو يفترض أنك ستستخدم لغة جينجا (Jinja). معظم المطورين في مجتمع فلاسك يستخدمون جينجا، وأنصحك بأن تفعل المِثل. يوجد بضعة إضافات (ملحقات) تتيح لنا استخدام لغات قوالب أخرى، مثل Flask-Genshi و Flask-Mako. استخدم الخيار الافتراضي (جينجا) ما لم يكن لديك سبب جيد لتستخدم شيء آخر. عدم معرفتك للبنية الكتابية لجينجا ليس بالسبب الجيد! فالسوف توفر الكثير من الوقت والصداع.

> **ملاحظة**
>
> معظم المواقع تَقصُد Jinja2 في حين أنها تكتب "Jinja" وحسب. حيث كان هناك إصدار أول من هذه اللغة (Jinja1)، ولكننا لن تعامل معه هنا. عندما سأذكر جينجا في هذا الكتب، فسأكون أقصد هذا الإصدار.

مقدمة سريعة حول جينجا

يؤدي توثيق جينجا عملاً جيداً في شرح البنية الكتابية وميزات هذه اللغة. لن أقوم بإعادة المعلومات المذكورة هناك، ولكن أريد أن أحرص على رؤيتك لهذه الملاحظة الهامة:

> يوجد نوعان من المحددات في اللغة: `{% ... %}` و `{{ ... }}`. يُستخدم الأول لتنفيذ الجمل (Statements) مثل حلقة for أو لإسناد القيم، أما الثاني فيُستخدم لطباعة نتيجة تعبير إلى القالب.
>
> — توثيق مُصمم قوالب جينجا

كيفية تنظيم القوالب

أين المكان المناسب لوضع القوالب في تطبيقنا؟ إذا كنت قد تابعت الدورة منذ البداية، فمن المؤكد أنك لاحظتَ أنَّ إطار فلاسك يملك مرونة عالية فيما يتعلق بتنظيم الأشياء. ولربما لاحظت أيضاً أنَّ هناك عادةً مكان موصى به لوضع تلك الأشياء. يوصى بوضع القوالب في دليل الحزمة.

```
myapp/
    __init__.py
    models.py
    views/
    templates/
    static/
run.py
requirements.txt
```

```
templates/
    layout.html
    index.html
    about.html
    profile/
        layout.html
        index.html
    photos.html
    admin/
        layout.html
        index.html
        analytics.html
```

يماثل تنظيم الدليل *templates* تنظيم موجهاتنا. حيث أنَّ القالب الذي سيُعرض في العنوان *myapp.com/admin/analytics* هو *templates/admin/analytics.html*. كما يوجد بعض القوالب الإضافية التي ستُستَمَد ولكن لن يتم عرضها بشكل مباشر. بحيث أنَّ ملفات *layout.html* سيتم وراثتها (استمدادها/تضمينها) من قوالب أخرى.

الوراثة

يعتمد دليل القوالب المُنظم بشكل جيد بشدة على الوراثة. فيُعرِّف **القالب الأب** عادةً الهيكل العام الذي سترثه جميع **القوالب الأبناء**. في مثالنا السابق، يعد القالب *layout.html* هو القالب الأب، والملفات البقيّة التي تحمل الإمتداد *html*. هي القوالب الأبناء.

ستملك عادةً ملف *layout.html* رئيسي (ذا مستوى أول) يُعرِّف التخطيط (التصميم/التنظيم) العام لتطبيقك، وملف آخر لكل قسم من موقعك. إذا قمت بأخذ نظرة على الدليل أعلاه، ستجد أنَّ هناك قالب رئيسي (*myapp/templates/layout.html*) كما توجد قوالب رئيسية فرعية مثل *myapp/templates/profile/layout.html* و *myapp/templates/admin/layout.html*. القالبان الفرعيان يرثان ويعدلان الأول (القالب ذو المستوى الأول).

يتم تنفيذ الوراثة باستخدام الوسمان `{% extends %}` و `{% block %}`. حيث يمكننا تعريف كتل في القالب الأب ليتم ملؤها (الكتابة بداخلها) بواسطة القوالب الأبناء.

```
{# _myapp/templates/layout.html_ #}

<!DOCTYPE html>
<html lang="en">
    <head>
        <title>{% block title %}{% endblock %}</title>
    </head>
    <body>
    {% block body %}
        <h1>This heading is defined in the parent.</h1>
    {% endblock %}
    </body>
</html>
```

الآن يمكننا استمداد (extend) القالب الأب في القالب الابن وتعريف محتوى (ملء) هذه الكتل.

```
{# _myapp/templates/index.html_ #}

{% extends "layout.html" %}
{% block title %}Hello world!{% endblock %}
{% block body %}
    {{ super() }}
    <h2>This heading is defined in the child.</h2>
{% endblock %}
```

تتيح لنا الدالة `super()` تضمين أياً كان ما بداخل هذه الكتلة في القالب الأب.

> قم بزيارة وثيقة جينجا عن الوراثة في القوالب للمزيد من المعلومات.

إنشاء وحدات ماكرو

يمكننا تطبيق مبدأ لا تكرر نفسك في القوالب عن طريق استخلاص المقاطع البرمجية التي ستُستخدم بشكل متكرر وتحويلها إلى **وحدات ماكرو** (Macros). على سبيل المثال، إن كنا نعمل على الشريط العلوي لتطبيقنا، فقد نرغب بإعطاء صنف (Class) مختلف للرابط "النشط" (أي رابط الصفحة الحالية). من دون استخدام وحدات الماكرو، سنضطر إلى استخدام العبارة `if ... else` للتحقق من كل رابط لإيجاد النشط منهم.

توفر وحدات الماكرو طريقة لتجزئة الشيفرة البرمجية، فهي تعمل كالدوال. دعنا نلقي نظرة على طريقة يمكننا استخدامها لتحديد الرابط النشط باستخدام الماكرو.

```
{# myapp/templates/layout.html #}

{% from "macros.html" import nav_link with context %}
<!DOCTYPE html>
<html lang="en">
    <head>
    {% block head %}
        <title>My application</title>
    {% endblock %}
    </head>
    <body>
        <ul class="nav-list">

            {{ nav_link('home', 'Home') }}
            {{ nav_link('about', 'About') }}
            {{ nav_link('contact', 'Get in touch') }}
```

```
        </ul>
    {% block body %}
    {% endblock %}
    </body>
</html>
```

ما قُمنا به في هذا المثال هو استدعاء ماكرو غير مُعرَّف (وهو الماكرو `nav_link`) وتمرير معاملين له: الأول اسم الدالة الهدف (أي اسم الدالة لدالة العرض المستهدفة)، والثاني النص الذي نريد إظهاره.

> **ملاحظة**
>
> قد تكون لاحظت أننا استخدامنا `with context` في عبارة الاستدعاء. يتكون **سياق** (Context) جينجا من مجموعة المعاملات المُمررة إلى دالة `()` `rendrer_template` وإلى سياق بيئة جينجا (Jinja environment context) من الشيفرة البرمجية. تُصبح هذه المتغيرات متوفرة في القالب الذي يجري عرضه.
>
> بعض المتغيرات تُمرر بشكل جلي بواسطتنا، مثل `render_template("index.html", color="red")`، ولكن يوجد بعض المتغيرات والدوال تُضمَّن تلقائياً بواسطة إطار فلاسك، مثل `request` و `g` و `session`. فعندما نستخدم `{% from ... import ... with context %}` نحن نخبر مُحرِّك جينجا أن يجعل جميع هذه المتغيرات متوفر أيضاً في الماكرو.

> **ملاحظة**
>
> – يمكنك إيجاد جميع المتغيرات العمومية (global) المُمرّرة إلى سياق جينجا تلقائياً من هنا.
>
> – يمكننا تعريف دوال ومتغيرات نريد دمجها في سياق جينجا باستخدام معالجات السياق.

الآن حان وقت تعريف الماكرو `nav_link` الذي استخدمناه في ذلك القالب.

```
{# myapp/templates/macros.html #}

{% macro nav_link(endpoint, text) %}
{% if request.endpoint.endswith(endpoint) %}
    <li class="active"><a
href="{{ url_for(endpoint) }}">{{text}}</a></li>
{% else %}
    <li><a href="{{ url_for(endpoint) }}">{{text}}</a></li>
{% endif %}
{% endmacro %}
```

قمنا بتعريف الماكرو في المسار *myapp/templates/macros.html*. في هذا الماكرو قمنا باستخدام الكائن `request` - والذي يكون متوفر إفتراضياً في سياق جينجا - للتأكد مما إذا كان الطلب الحالي قد وُجِّهَ إلى الدالة المُمرّرة إلى الماكرو `nav_link`. إن كان، فبالتالي نحن في تلك الصفحة ويمكننا تعليم رابطها كرابط نشط.

> **ملاحظة**
>
> عبارة `from x import y` تأخذ القيمة x كمسار نسبي. بحيث إذا كان قالبنا

> في المسار *myapp/templates/user/blog.html* فسنستخدم `from "../macros.html" import nav_link with context`.

مُرشِّحات مخصصة

مُرشِّحات جينجا هي عبارة عن دوال يُمكِن تطبيقها على نتائج العبارة في المحدد `{{ ... }}`. بحيث تُطبَّق هذه المُرشِّحات قبل طباعة هذه النتائج في القالب.

```
<h2>{{ article.title|title }}</h2>
```

في المثال أعلاه، المُرشِّح `title` سيأخذ `article.title` ويعيد النسخة المعنونة حرفياً (title cased) منه، والتي من ثم ستُطبَع إلى القالب. تشبه آلية عمل هذه الخاصية إلى حدٍ كبير آلية عمل "ضخ (piping)" مُخرج برنامج إلى مدخلات برنامج آخر في أنظمة يونكس.

> **ملاحظة**
>
> توجد حفنة من المُرشِّحات المبنيّة ضمنياً في جينجا من مثل `title`. يمكنك رؤية القائمة الكاملة في توثيقات جينجا.

يمكننا تعريف مُرشِحات خاصة بنا لاستخدامها في قوالب جينجا. كمثال، سنقوم بتنفيذ مُرشِّح بسيط باسم `caps` يقوم بجعل جميع أحرف السلسلة النصيّة كبيرة.

> **ملاحظة**
>
> يملك محرك جينجا بالفعل مُرشِّح يسمى `upper` وهو يؤدي تلك الوظيفة، كما

> يملك المُرشِّح `capitalize` الذي يقوم بجعل الحرف الأوّل بالحالة الكبيرة وباقي الأحرف بالحالة الصغيرة. تتعامل هذه المُرشِّحات أيضاً مع تحويلات اليونيكود (Unicode)، ولكننا سنبقي المثال بسيطاً لتسليط الضوء على الفكرة.

سنقوم بتعريف المُرشِّح في وحدة تتوضع في المسار *myapp/util/filters.py*. أي سيصبح لدينا الحزمة `util` والتي سنضع فيها الوحدات الأخرى المتنوعة.

```
# myapp/util/filters.py

from .. import app

@app.template_filter()
def caps(text):
    """Convert a string to all caps."""
    return text.uppercase()
```

في هذا المثال قمنا بتسجيل الدالة كمُرشِّح جينجا باستخدام المُزخرِف `@app.template_filter()`. يأخذ المُرشِّح اسمه الافتراضي من اسم الدالة، ولكن يمكننا تغيير ذلك عن طريق تمرير معامل للمُزخرِف.

```
@app.template_filter('make_caps')
def caps(text):
    """Convert a string to all caps."""
    return text.uppercase()
```

الآن يمكننا استدعاء المُرشِّح بالاسم `make_caps` في القالب بدلاً من استخدام `caps` كالتالي: `{{ hello world!"|make_caps" }}`.

لجعل المشرح متاحاً في القوالب، كل ما علينا القيام به هو استدعاءه في ملف *__init__.py* الرئيسي.

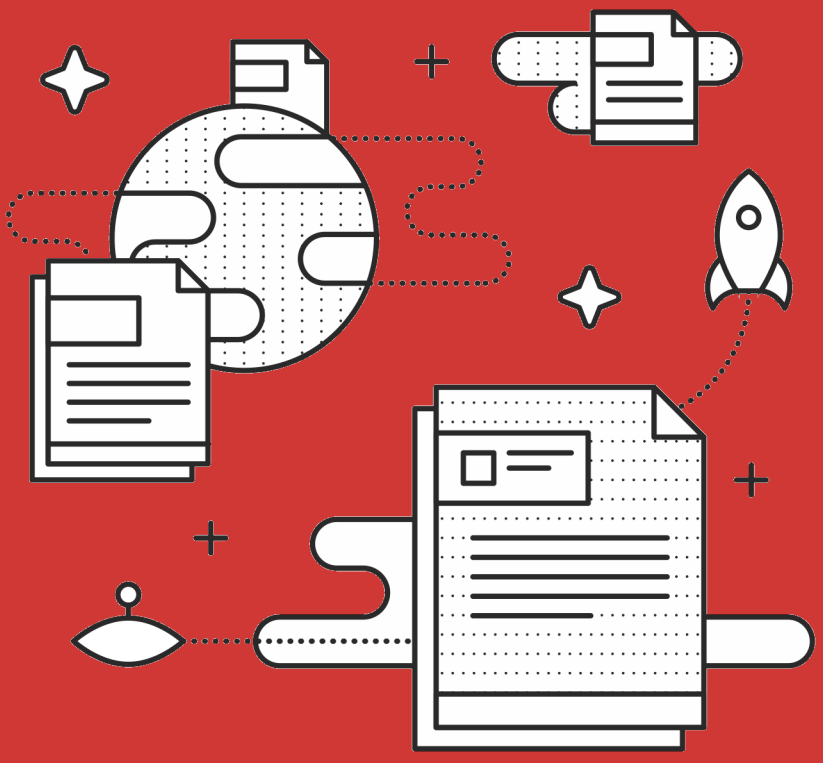
```
# myapp/__init__.py

# تأكد أنه تم استهلال التطبيق أولاً لمنع مشاكل الاستيرادات الحلقية.
from .util import filters
```

الخلاصة

- استخدم جينجا للتعامل مع القوالب.
- يوجد نوعان من المحددات في محرك جينجا: `{% ... %}` و `{{ ... }}`. يُستخدم الأول لتنفيذ الجمل (Statements) مثل حلقة for أو لإسناد القيم، أما الثاني فيُستخدم لطباعة نتيجة تعبير إلى القالب.
- ينصح بأن توضع القوالب في الدليل */myapp/templates* أي في دليل بداخل حزمة التطبيق.
- أنصح بأن تعكس هيكلة الدليل */templates* هيكلة عناوين الروابط في التطبيق.

- أنصح بأن تملك ملف *layout.html* رئيسي في الدليل *myapp/templates* وملف خاص لكل قسم في الموقع. وعليك أن تحرص على أن يستدعي الملف المخصص لقسم معين الملف الرئيسي.
- وحدات الماكرو هي مثل الدوال مصنوعة من شيفرة القالب.
- المُرشِّحات هي دوال مصنوعة من شيفرة بايثون وتستخدم في القوالب.

الفصل الثامن: الملفات الساكنة



كما يوضِّح اسمها، الملفات الساكنة هي الملفات التي لا تتغيّر. في تطبيقك العادي، ستجد ملفات ساكنة مثل ملفات CSS وملفات الجافاسكربت والصور. قد يتضمن تطبيقك أيضاً ملفات الصوت وغيرها من الأشياء المشابهة.

تنظيم الملفات الساكنة

سننشئ دليلاً لملفاتنا الساكنة وسنسميه *static* بداخل حزمة تطبيقنا.

```
myapp/
    __init__.py
    static/
    templates/
    views/
    models.py
run.py
```

كيفيّة تنظيم الملفات بداخل الدليل /static هي تفضيل شخصي بحت. شخصياً، أنزعج قليلاً من انمزاج مكتبات الطرف الثالث (مثل جيكوري وبوتستراب) مع ملفات CSS والجافاسكربت خاصتي. لتجنب ذلك أنصح بفصل مكتبات الطرف الثالث ووضعها بداخل المجلد *lib/* في دليل مناسب لها. بعض المشاريع تستخدم الدليل *vendor/* بدلاً من *lib/*.

```
static/
    css/
        lib/
            bootstrap.css
```

```
        style.css
        home.css
        admin.css
    js/
        lib/
            jquery.js
        home.js
        admin.js
    img/
        logo.svg
        favicon.ico
```

عرض أيقونة المفضلة

ستُعرَض الملفات الساكنة بداخل الدليل *static* من الرابط *example.com/static/*. تتوقع متصفحات الويب وغيرها من البرمجيات وجود أيقونة المفضلة (favicon) في الرابط *example.com/favicon.ico* افتراضياً. لإصلاح هذا التضارب، يمكننا إضافة السطر التالي إلى قسم `<head>` في قالب موقعنا.

```
<link rel="shortcut icon"
    href="{{ url_for('static', filename='img/favicon.ico') }}">
```

إدارة الأصول (الملفات) الساكنة باستخدام إضافة Flask-Assets

إضافة Flask-Assets هي إضافة متخصصة بإدارة الملفات الساكنة. تتيح هذه الإضافة أداتان مفيدتان بشكل كبير. تسمح لك الأداة الأولى بتعريف **رزم** (bundles) من الأصول

في شيفرة بايثون خاصتك يُمكِن إدراجها معاً في قالب موقعك. أما الأداة الثانية فتسمح لك **بالمعالجة المسبقة** (pre-process) لتلك الملفات. هذا يعني أنه يمكنك مزج وتصغير ملفات الـ CSS والجافاسكربت خاصتك بحيث لن يكون على المستخدم سوى تحميل ملفين مُصغرين من دون إجبارك على تطوير خط ضخ معقد. يمكنك حتى تجميع (compile) ملفاتك من مصادر Sass، و LESS، و CoffeeScript وغيرها من المصادر الأخرى.

```
static/
    css/
        lib/
            reset.css
        common.css
        home.css
        admin.css

    js/
        lib/
            jquery-1.10.2.js

            Chart.js
        home.js
        admin.js

    img/
        logo.svg
        favicon.ico
```

تعريف الرزم

يملك تطبيقنا قسمين: الموقع العمومي ولوحة تحكم المشرف، ويُشار إليهم بـ "home" و "admin" على التوالي في تطبيقنا. سنُعرِف أربع رزم تغطي ملفات CSS والجافاسكربت لكل قسم. سنُعرِف الرزم في الوحدة *assets* بداخل الحزمة `util`.

```
# myapp/util/assets.py

from flask_assets import Bundle, Environment
from .. import app

bundles = {

    'home_js': Bundle(
        'js/lib/jquery-1.10.2.js',
        'js/home.js',
        output='gen/home.js'),

    'home_css': Bundle(

        'css/lib/reset.css',
        'css/common.css',
        'css/home.css',
        output='gen/home.css'),

    'admin_js': Bundle(

        'js/lib/jquery-1.10.2.js',
        'js/lib/Chart.js',
        'js/admin.js',
```

```
        output='gen/admin.js'),

    'admin_css': Bundle(
        'css/lib/reset.css',
        'css/common.css',
        'css/admin.css',
        output='gen/admin.css')
}

assets = Environment(app)

assets.register(bundles)
```

تقوم الإضافة بمزج ملفاتك حسب الترتيب الذي تم سردهم به. فإذا كان الملف *admin.js* يتطلب الملف *jquery-1.10.2.js* احرص على وضع ملف الجيكوري أولاً.

قمنا بتعريف الرزم في دليل لجعل تسجليهم أسهل. الحزمة webassets، والتي تعتمد عليها إضافة Flask-Assets، تتيح لنا تسجيل الرزم بعدة طرق، ومنها تمرير قاموس كالذي استخدمناه في المقتطف السابق.[1]

في حين أننا نُسجِل رزمنا في `util.assets`، فكل ما علينا القيام به هو استيراد تلك الوحدة في الملف *__init__.py* بعد استهلال تطبيقنا.

(1) يمكنك رؤية كيف تعمل عملية تسجيل الرزم في الشيفرة المصدريّة للإضافة.

```
# myapp/__init__.py

# [...] هنا نقوم باستهلال التطبيق

from .util import assets
```

استخدام الرزم

لاستخدام الرزم الخاصة بقسم لوحة المشرف، سنقوم بإدراجهم في القالب الأب لقسم المشرف: *admin/layout.html*.

```
templates/
    home/
        layout.html
        index.html
        about.html
    admin/
        layout.html
        dash.html
        stats.html
```

```
{# myapp/templates/admin/layout.html #}

<!DOCTYPE html>
<html lang="en">
    <head>

        {% assets "admin_js" %}
            <script type="text/javascript"
src="{{ ASSET_URL }}"></script>
```

```
        {% endassets %}
        {% assets "admin_css" %}
            <link rel="stylesheet" href="{{ ASSET_URL }}" />
        {% endassets %}
    </head>
    <body>
    {% block body %}
    {% endblock %}
    </body>
</html>
```

يمكننا القيام بنفس الشيء لرزم القسم "home" في الملف *templates/home/layout.html*.

استخدام المُرشِّحات

يمكننا استخدام المُرشِّحات للقيام بمعالجة مسبقة لملفاتنا الساكنة. هذا مفيد بشكل خاص لتصغير رزم CSS والجافاسكربت خاصتنا.

```
# myapp/util/assets.py

# [...]

bundles = {

    'home_js': Bundle(
        'lib/jquery-1.10.2.js',
        'js/home.js',
```

```
        output='gen/home.js',
        filters='jsmin'),

    'home_css': Bundle(
        'lib/reset.css',
        'css/common.css',
        'css/home.css',
        output='gen/home.css',
        filters='cssmin'),

    'admin_js': Bundle(
        'lib/jquery-1.10.2.js',
        'lib/Chart.js',
        'js/admin.js',
        output='gen/admin.js',
        filters='jsmin'),

    'admin_css': Bundle(
        'lib/reset.css',
        'css/common.css',
        'css/admin.css',
        output='gen/admin.css',
        filters='cssmin')
}

# [...]
```

> **ملاحظة**
>
> لاستخدام المُرشِّحات `jsmin` و `cssmin`، احرص على تثبيت الحزمتين `jsmin` و `cssmin` (على سبيل المثال باستخدام `pip install jsmin cssmin`). تأكد من إضافتهم إلى ملف المتطلبات (*requirements.txt*) أيضاً.

ستقوم إضافة Flask-Assetes بدمج وضغط ملفاتنا في أول مرة يُعرَض فيها القالب، وستقوم تلقائياً بتحديث الملفات المضغوطة عندما يتم تحديث أحد ملفات المصدر.

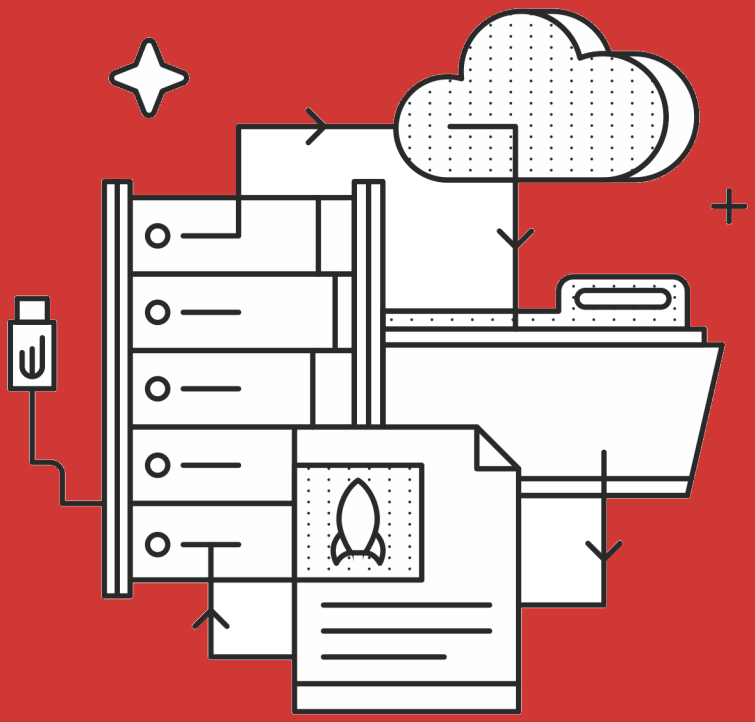
> **ملاحظة**
>
> إذا قمت بإعطاء الخيار `ASSETS_DEBUG` القيمة `True` في ملف الإعداد خاصتك فستقوم الإضافة بإخراج كل ملف مصدر مفرداً بدلاً من دمجهم.

> **ملاحظة**
>
> ألقِ نظرة على بعض المُرشِّحات الأخرى التي يمكنك استخدامها مع إضافة Flask-Assets.

الخلاصة

- توضع الملفات الساكنة في الدليل */static*.
- قم بفصل مكتبات الطرف الثالث عن الملفات الساكنة خاصتك.
- حدد موقع أيقونة المفضلة في قوالبك.
- استخدم إضافة Flask-Assets لإدراج الملفات الساكنة في قوالبك.
- تتمكن إضافة Flask-Assets من تجميع، ودمج، وضغط ملفاتك الساكنة.

الفصل التاسع: تخزين البيانات



معظم تطبيقات فلاسك ستتعامل مع تخزين البيانات في مرحلةٍ ما. يوجد العديد من الطرق المختلفة لفعل ذلك. إيجاد الطريقة الأفضل يعتمد بشكل كبير على نوع البيانات الذي تريد تخزينه. فإذا كنت تريد تخزين بيانات علائقية (مثال: مستخدم لديه منشورات، والمنشورات لديها مستخدم أنشأها)، فاستخدام قاعدة بيانات علائقية من المرجح سيكون أفضل (أليس هذا جلياً!). الأنواع الأخرى من البيانات قد يلائمها أن تُخزَّن في قاعدة بيانات NoSQL، مثل MongoDB.

لن أخبرك بطريقة حتى تختار فيها محرك قاعدة البيانات المناسب لتطبيقك. هناك بعض الأشخاص سيخبروك أن قاعدة بيانات NoSQL هي الطريقة الوحيدة المناسبة لك والبعض سيقولون المثل عن قواعد البيانات العلائقية. كل ما سأخبرك به حيال هذا الموضوع هو إن لم تكن مُتأكِداً، فقاعدة البيانات العلائقية (مثل MySQL أو PostgreSQL) ستعمل - تقريباً - مع أياً ما تقوم به.

إضافة لذلك، إذا استخدمت قاعدة بيانات علائقية، فسيمكنك العمل مع إضافة SQLAlchemy وهذه الإضافة ممتعة.

إضافة SQLAlchemy

إضافة SQLAlchemy هي مُطابِقة علاقات كائنية (ORM - Object Relational Mapper). هي بالأساس طبقة تجريد (abstraction layer) مبنية على رأس استعلامات SQL الخام التي يتم إجراؤها على قاعدة بياناتنا. توفر هذه الإضافة واجهة برمجية متوافقة مع قائمة كبيرة من محركات قواعد البيانات. المحركات الأكثر شعبية منها

MySQL و PostgreSQL و SQLite. هذا يجعل من السهل نقل البيانات بين نماذجنا (models) وقواعد البيانات ويجعل من السهل حقاً القيام بأمور أخرى مثل التنقل بين محركات قواعد البيانات وترحيل (migrate) مخططات (schemas) قواعد البيانات.

يوجد إضافة رائعة لفلاسك تجعل استخدام SQLAlchemy أسهل تدعى بـ Flask-SQLAlchemy. تقوم هذه الإضافة بإعداد العديد من الإعدادات الإفتراضية لـ SQLAlchemy. كما تتولى بعض أعمال إدارة الجلسة وبالتالي لن نضطر إلى التعامل مع أمور الحماية في شيفرة تطبيقك.

دعنا نتعمق بكيفية استخدام هذه الإضافة في الشيفرة. سنقوم بتعريف بعض النماذج ومن ثم إعداد SQLAlchemy. ستوضع النماذج في الملف *myapp/models.py*، ولكن بدايةً سنقوم بتعريف قاعدة البيانات في الملف *myapp/__init__.py*.

```
# ourapp/__init__.py

from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__, instance_relative_config=True)

app.config.from_object('config')
app.config.from_pyfile('config.py')

db = SQLAlchemy(app)
```

اولاً، قمنا باستهلال وإعداد التطبيق (الكائن `app`) ومن ثم استخدمناه لاستهلال معالج قاعدة بيانات SQLAlchemy. سنقوم باستخدام المجلد الحالِي لإعداد قواعد البيانات وبالتالي ينبغي علينا استخدام الخيار `instance_relative_config` عند استهلال التطبيق واستدعاء `app.config.from_pyfile` لتحميل المجلد. ومن ثم يمكننا تعريف نماذجنا.

```
# ourapp/models.py

from . import db

class Engine(db.Model):

    # Columns

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)

    title = db.Column(db.String(128))

    thrust = db.Column(db.Integer, default=0)
```

الأصناف `Column`، و `Integer`، و `String`، و `Model` وغيرها من أصناف (classes) إضافة SQLAlchemy متوافرة بواسطة الكائن `db` المنُشأ (constructed) من إضافة FLask-SQLAlchemy. قمنا بتعريف نموذج لتخزين الحالة الحالية لمحركات مركبتنا الفضائية. كل محرك يملك رقم تعريف (id)، وعنوان (title)، ومستوى دفع (thrust level).

مازلنا بحاجة لإضافة بعض معلومات قاعدة البيانات إلى ملف إعداداتنا. نحن نستخدم المجلد الحالِي لإبقاء متغيرات الإعداد السرية خارج نظام التحكم بالإصدارات، لذلك سنضع تلك المعلومات في الملف *instance/config.py*.

```
# instance/config.py

SQLALCHEMY_DATABASE_URI =
"postgresql://user:password@localhost/spaceshipDB"
```

> **ملاحظة**
>
> يختلف عنوان قاعدة بياناتك بناءً على المحرك الذي تستخدمه ومكان تواجده. طالع توثيق SQLAlchemy حول الصيغة الكتابية لعنوان قاعدة البيانات.

إنشاء قاعدة البيانات

الآن بعد أن قمنا بإعداد قاعدة البيانات وتعريف النموذج، يمكننا إنشاء قاعد البيانات.

تتضمن هذه الخطوة إنشاء قاعدة البيانات اعتماداً على مخطط النماذج المُعرفِة.

عادةً قد تكون هذه العملية مزعجة. ولكن لحسن الحظ، توفر إضافة SQLAlchemy أمر رائع يقوم بكل شيء لنا.

دعنا نفتح مُفسر بايثون من الطرفية في جذر المستودع.

```
$ pwd
/Users/me/Code/myapp
$ workon myapp
(myapp)$ python
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from myapp import db
>>> db.create_all()
>>>
```

الآن - شكراً لإضافة SQLAlchemy - تم إنشاء جميع الجداول في قاعدة البيانات المحددة في ملف الإعداد.

أداة Alembic

مخطط قاعدة البيانات ليس جامداً (لا يتغير). على سبيل المثال، قد ترغب بإضافة العمود `last_fired` إلى محرك الجدول. إذا كنت لا تملك أية بيانات، فيمكنك تحديث النموذج وتشغيل الأمر `db.create_all()` مجدداً. ولكن ماذا لو كنت نملك بيانات مُسجلة من ستة أشهر في ذلك الجدول؟! أنت غالباً لا تريد حذف كل ذلك والبدء من الصفر. لذلك جاءت آلية الترحيل لحل تلك المشكلة.

برمجية Alembic هي أداة ترحيل قواعد بيانات أُنشئَت خصيصاً للعمل مع SQLAlchemy. تتيح لنا هذه الأداة الاحتفاظ بسجل يُصدَر من مخطط قاعدة بيانات حتى نتمكن من تحديث المخطط لاحقاً أو حتى الرجوع إلى إصدار أقدم.

تمتلك أداة Alembic دورة شاملة لتبدأ معها، لذلك سأعطيك لمحة سريعة فقط وسأشير إلى بضعة أشياء عليك تعلمها.

سننشئ "بيئة الترحيل (migration environment)" باستخدام الأمر `alembic init`. حالما نقوم بتشغيل هذا الأمر في جذر المستودع سينتُج دليل جديد باسم خَلاَّق (واضح) وهو *alembic*. سيصبح مستودعنا بعد ذلك مماثل للمثال أدناه (المأخوذ من دورة Alembic).

```
ourapp/
    alembic.ini
    alembic/
        env.py
        README
        script.py.mako
        versions/
            3512b954651e_add_account.py
            2b1ae634e5cd_add_order_id.py
            3adcc9a56557_rename_username_field.py
    myapp/
        __init__.py
        views.py
        models.py
        templates/
    run.py
    config.py
    requirements.txt
```

يحوي الدليل *alembic/* برمجيات تقوم بترحيل بياناتنا بين الإصدارات. كما يوجد أيضاً الملف *alembic.ini* والذي يحوي معلومات الإعداد.

> **ملاحظة**
>
> أضف الملف *alembic.ini* إلى الملف *gitignore*. ستقوم بالاحتفاظ ببيانات اعتماد قاعدة البيانات في هذا الملف، لذلك أنت **لا** تريد بالتأكيد وضعه في نظام التحكم بالإصدارات.

عندما يحين موعد تحديث المخطط فهناك بضعة خطوات علينا القيام بها. أولاً يجب علينا تنفيذ الأمر `alembic revision` لتوليد برمجية الترحيل. ومن ثم علينا فتح ملف بايثون المولد حديثاً في المسار *myapp/alembic/versions/* وملء دوال `upgrade` و `downgrade` باستخدام الأدوات التي يوفرها الكائن `op`.

حالما تصبح برمجية الترحيل جاهزة، يمكننا تشغيل الأمر `alembic upgrade head` لترحيل بياناتنا إلى الإصدار الأحدث.

> **ملاحظة**
>
> للمزيد من التفاصيل حول إعداد أداة Alembic، وإنشاء برمجية الترحيل، وتشغيل عملية الترحيل، طالع دورة Alembic.

> **تحذير**
>
> لا تنسى وضع خطة عملية لعمل نسخة احتياطية لبياناتك. هذا الموضوع خارج إطار الكتاب، ولكن ينبغي دائماً أن تكون قاعدة بياناتك محفوظة بطريقة آمنة وقوية.

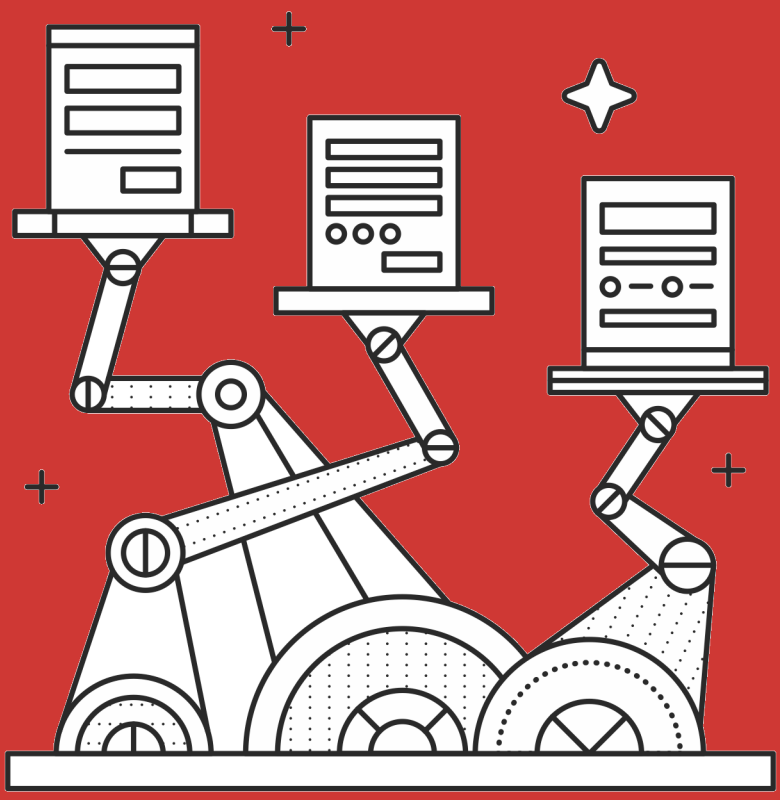
> **ملاحظة**
>
> قواعد بيانات NoSQL أقل استقراراً مع فلاسك، ولكن طالما يملك محرك قواعد البيانات الذي اخترته مكتبة لبايثون، فينبغي أن تكون قادراً على

> استخدامه. يوجد حتى عدة إضافات فلاسك في سجل إضافات فلاسك تساعدك على استخدام محركات NoSQL مع فلاسك.

الخلاصة

- استخدم SQLAlchemy للعمل مع قواعد البيانات العلائقية.
- استخدم إضافة Flask-SQLAlchemy للعمل مع SQLAlchemy.
- تساعدك أداة Alembic على ترحيل بياناتك بين تغييرات المخططات.
- يمكنك استخدام قواعد بيانات NoSQL مع فلاسك، ولكن الطرق والأدوات متباينة تبعاً للمحرك المستخدم.
- احرص على إنشاء نسخة احتياطية لبياناتك.

الفصل العاشر: التعامل مع الاستمارات



الاستمارة (form) هي عنصر أساسي يتيح للمستخدمين التفاعل مع تطبيقات الويب. لا يقدم إطار فلاسك - ضمنياً - أي شيء ليساعدنا في التعامل مع الاستمارات، ولكن يوجد في بحر إضافات فلاسك الغني، كالمُعتاد، إضافة Flask-WTF التي توفر لنا إمكانيّة استخدام حزمة WTForms الشهيرة في تطبيقاتنا. تجعل هذه الحزمة تعريف الاستمارات والتعامل مع بياناتها سهلاً وبسيطاً.

إضافة Flask-WTF

أول ما سنقوم به مع هذه الإضافة (بعد تثبيتها طبعاً) هو استخدامها لتعريف استمارة بسيطة في الحزمة myapp.forms.

```
# ourapp/forms.py

from flask_wtf import Form
from wtforms import StringField, PasswordField
from wtforms.validators import DataRequired, Email

class EmailPasswordForm(Form):
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired()])
```

> **ملاحظة**
>
> وفَرَت إضافة Flask-WTF أغلفتها (wrappers) الخاصة، لحين قدوم الإصدار الثاني، للتعامل مع حقول إدخال (fields) ومصادقات (validators) حزمة WTForms. وبالتالي، قد تجد في الكثير من البرمجيات أنَّه يتم استدعاء

> `TextField`، و `PasswordField` وغيرها من مكتبة `flask_wtforms` بدلاً من `wtforms`.
>
> بدءً من الإصدار 0.9 ينبغي علينا استيراد كل تلك الأشياء من حزمة `wtforms` مباشرةً.

الاستمارة التي عرفناها هي لتسجيل دخول المستخدمين. كان بإمكاننا تسميتها `SigninForm`، ولكن من خلال الابتعاد عن إطلاق تسميات محددة سيكون بإمكاننا استخدام نفس الصنف في أشياء أخرى، كاستخدامه في استمارة إنشاء الحساب مثلاً. سيودي بنا تعريف أصناف استمارات لأهداف محددة إلى إنشاء العديد من الأصناف المتشابهة الغير ضروريّة. من الأفضل تسميّة صنوف الاستمارات بناءً على محتواها، فهذا ما يميّزُها عن غيرها. بالرغم من ذلك، سنحتاج في بعض الأحيان إلى إنشاء أصناف محددة الغرض، وحينها لن يكون اختيار اسم يصف وظيفتها مشكلة.

ستقوم استمارة تسجيل الدخول بعدة أمور عند عملها. حيث ستقوم بحماية تطبيقنا من ثغرات تزوير الطلبات (CSRF)، وتنقيح مدخلات المستخدم بناءً على القالب الذي حددناه (فمثلاً البريد الإلكتروني يجب أن يكون له شكل محدد وكذلك رقم الهاتف).

التحقق من هجمات تزوير الطلبات والوقاية منها

يعد المصطلح CSRF اختصاراً لـ "Cross Site Request Forgery" (ثغرة تزوير الطلبات). تنطوي عمليّة الهجوم على قيام طرف ثالث بتزوير طلب (كإرسال بيانات استمارة)

وإرساله لخادم التطبيق. يعتقد الخادم المُصاب بهذه الثغرة أنَّ ذلك الطلب - المزوَّر - قادم من استمارة مُستضافة عليه، فيتصرف وكأنَّه طلب عادي.

إليك هذا المثال الذي يوضِّح خطورة الهجمة: لنقل أنَّك مُسجَّل لدى مزود خدمة بريد إلكتروني، وهذا المزود يتيح لك حذف حسابك عبر استمارة بسيطة. تُرسِل تلك الاستمارة طلباً من النوع POST إلى الدالة `account_delete` التي تحذف الحساب الذي كان مُسجلاً عند إرسال بيانات الاستمارة. الآن دعنا نُنشئ استمارةً جديدة، ولتكن هذه الاستمارة موجودة على خادم آخر غير ذاك. ستُرسِل هذه الاستمارة، أيضاً، طلباً من النوع POST إلى الدالة `account_delete` الموجودة على استمارة التطبيق الأصلي. ماذا لو أقنعنا الآن شخصاً بالدخول إلى استمارتنا وضغط زر الإرسال (أو القيام بذلك تلقائياً عند الدخول إلى الصفحة باستخدام الجافاسكربت)؟ سيُحذَف حسابه المُسجَّل (إذا كانت لديه جلسة في المُتصفِح) عند مزود البريد الإلكتروني ذاك. كان يمكن تفادي ذلك لو كان موقع مزود تلك الخدمة أذكى ويستطيع التمييز بين الطلبات القادمة من الاستمارات الموجودة لديه أو الطلبات القادمة من مواقع أخرى.

كيف يمكننا إذاً إيقاف ذلك الوثوق الأعمى لتطبيقاتنا بجميع الطلبات من النوع POST؟ تجعل إضافة WTForms ذلك مُمكناً عبر توليد رمز مميز (token) عند تصيير (rendering) الاستمارات. يُرسَل ذلك الرمز إلى الخادم مع بيانات الطلب ويتم التحقق منه قبل قبول تلك البيانات. الفكرة الأساسيّة لهذا الرمز أنَّه يُضاف إلى متغيّر في جلسة المُستخدِم وتنتهي صلاحيته بعد مقدار مُعين من الوقت (بعد ثلاثين دقيقة افتراضياً). وبهذه الطريقة الوحيد من يستطيع إرسال الاستمارة هو مَنْ فَتَحَها (أو شخص ما على

نفس الحاسب)، ولديه أيضاً مهلة ثلاثين دقيقة (أو أي وقت تحدده أنت) لإرسال البيانات بعد فتح الصفحة.

> **ملاحظة**
>
> – اقرأ المزيد حول كيفيّة توليد إضافة WTForms لتلك الرموز المميزة في توثيقات الإضافة.
>
> – اقرأ المزيد عن ثغرة تزوير الطلبات في موسوعة أواسب.

علينا بدايةً إنشاء دالة صفحة التسجيل قبل استخدام إضافة Flask-WTF للوقاية من ثغرة تزوير الطلبات:

```
# ourapp/views.py

from flask import render_template, redirect, url_for

from . import app
from .forms import EmailPasswordForm

@app.route('/login', methods=["GET", "POST"])
def login():
    form = EmailPasswordForm()
    if form.validate_on_submit():

        # كود التحقق من كلمة مرور واسم المستخدم
        # [...]

        return redirect(url_for('index'))
    return render_template('login.html', form=form)
```

قمنا بدايةً باستيراد الحزمة `forms` واستهلالها في الدالة. ومن ثم قمنا بتشغيل الدالة `form.validate_on_submit`. تُرجِع هذه الدالة القيمة `True` إن كانت بيانات الاستمارة قد أُرسِلَت (بواسطة طريقة نقل البيانات POST أو PUT مثلاً) وتم التحقق منها من جميع المُصادِقات التي عرفناها في الوحدة *forms.py*.

> **ملاحظة**
>
> – توثيق الدالة الصنفيّة Form.validate_on_submit
>
> – الشيفرة المصدريّة للدالة الصنفيّة Form.validate_on_submit

ستقوم الدالة عندما ترى أنَّ بيانات الاستمارة لم تُرسَل بعد (أي أنَّ الدالة أُستُدعيت بطلب GET) بتمرير الكائن `form` إلى القالب *login.html* وتصييره. يوجد أدناه الشكل الذي ستبدو عليه شيفرة القالب عند استخدام ميزة الحماية من ثغرة تزوير الطلبات.

```
{# ourapp/templates/login.html #}

{% extends "layout.html" %}
<html>
    <head>
        <title>Login Page</title>
    </head>
    <body>
        <form action="{{ url_for('login') }}" method="post">
            <input type="text" name="email" />
            <input type="password" name="password" />
            {{ form.csrf_token }}
        </form>
    </body>
</html>
```

تُنشئ الشيفرة `{{ form.csrf_token }}` حقلاً مخفياً يحتوي على ذلك الرمز الفريد لتتحقق الإضافة منه عند إرسال بيانات الاستمارة. هذا كل ما علينا القيام به وبدون إجراء أي عمليات تحققيّة أخرى.

حماية طلبات الأجاكس من ثغرة تزوير الطلبات

لا تقتصر أهميّة الرموز الفريدة التي تولّدها الإضافة على التحقق من الاستمارات وحسب. فمن الممكن أن تكون هناك أنواع أخرى من الطلبات في التطبيق يُمكِن تزويرها (مثل طلبات الأجاكس)، لذلك عليك استخدام هذه الخاصيّة في تلك الطلبات أيضاً.

> **ملاحظة**
>
> يحتوي توثيق إضافة Flask-WTF على قسم يتكلّم عن استخدام رموز الوقاية من ثغرة تزوير الطلبات في طلبات الأجاكس.

إنشاء مُصادِق مخصص

تتيح لنا الإضافة إمكانيّة تعريف مُصادِقات مخصصة بنا إلى جانب تلك التي توفرها افتراضياً (مثل المُصادِق `Required()` و `Email()`). سأشرح كيفيّة القيام بذلك عبر صنع المُصادِق `Unique()` الذي يقوم بالتحقق من قاعدة البيانات والتأكّد من أنَّ القيمة التي أدخلها المُستخدِم في الحقل الفلاني غير موجودة. يُمكِن استخدام مُصادِق كهذا للتأكد من انَّ البريد الإلكتروني واسم المُستخدم المدخلان في الحقول فريدان. كنّا سنضطر إلى القيام بذلك في دالة عرض الصفحة لو أنّنا لا نستخدم إضافة WTForms، ولكن بما أنَّ

الإضافة توفر لنا هذه الخواص المُيسِّرة فأصبح بإمكاننا عزل الوظيفتين وإنشاء مُصادِق مُنفصِل ومُخصَّص لذلك.

دعنا بدايةً نقوم بتعريف استمارة تسجيل بسيطة لغرضنا التوضيحي:

```
# ourapp/forms.py

from flask_wtf import Form
from wtforms import StringField, PasswordField
from wtforms.validators import DataRequired, Email

class EmailPasswordForm(Form):
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired()])
```

الآن سنقوم بتعريف المُصادِق الذي سيتأكّد أنَّ البريد الإلكتروني فريد وغير موجود في قاعدة البيانات. سأضع المُصادِق في وحدة جديدة في الدليل `util` بالاسم `validators`.

```
# ourapp/util/validators.py
from wtforms.validators import ValidationError

class Unique(object):
    def __init__(self, model, field, message=u'This element already
exists.'):
        self.model = model
        self.field = field

    def __call__(self, form, field):
```

```
        check = self.model.query.filter(self.field ==
field.data).first()
        if check:
            raise ValidationError(self.message)
```

يفترض المُصادق أننا نستخدم إضافة SQLAlchemy في تعريفنا لنماذج قاعدة البيانات. تتوقع إضافة WTForms أن يُعيد المُصادِق نوعاً من الكائنات القابلة للاستدعاء (كالصنوف القابلة للاستدعاء (callable class) على سبيل المثال).

سنقوم بتحديد المُعطيات التي ينبغي تمريرها إلى المُصادِق في الدالة `__init__`. في حالتنا سنرغب بتمرير النموذج (والذي هو النموذج `User`) والحقل المُراد التحقق منه. سيُصدِر المُصادق الخطأ `ValidationError` عندما يُستدعى ويجد أنَّ هناك قيمة مطابقة لبيانات الاستمارة في قاعدة البيانات. قمنا أيضاً بإضافة رسالة افتراضيّة لتُضاف إلى رسالة الخطأ `ValidationError` بواسطة المُعامِل `message`.

الآن أصبح بإمكاننا تعديل الصنف `EmailPasswordForm` لنستخدم فيه المُصادِق `Unique` الذي أنشأناه.

> **ملاحظة**
>
> لا يتعيّن على المُصادِق الذي صنعناه أن يكون صنفاً قابلاً للاستدعاء. حيث يُمكِن أن يُرجِع كائِناً قابلاً للاستدعاء أو أن يُستدعى مباشرةً. يحتوي توثيق إضافة WTForms بعض الأمثلة حول ذلك.

تصيير الاستمارات

توفِر إضافة WTForms خاصيّة مُفيدة أخرى لتساعدنا على تصيير شيفرة الاستمارات. يُستخدَم الصنف `Field` الموجود في الإضافة لتصيير شيفرة لغة ترميز النص الفائق (HTML) المُمثِّلة لحقل ما، وبالتالي جلّ ما علينا القيام به هو استدعاء الحقول التي نرغب بها من داخل القالب لتصييرها ديناميكياً. الكيفيّة مماثلة لِمَ كنّا نقوم به عند تصيير حقل الرمز السري بواسطة الشيفرة `csrf_token`. انظر أدناه إلى المثال الذي يوضِّح كيفيّة استخدام الحقول المبنيّة في إضافة WTForms لإنشاء قالب بسيط لتسجيل الدخول.

```
{# ourapp/templates/login.html #}

{% extends "layout.html" %}
<html>
    <head>
        <title>Login Page</title>
    </head>
    <body>
        <form action="" method="post">
            {{ form.email }}
            {{ form.password }}
            {{ form.csrf_token }}
        </form>
    </body>
</html>
```

يمكننا تخصيص مظهر الحقول عبر تمرير خصائص الحقل كمُعطيات عند الاستدعاء (انظر أدناه).

```
<form action="" method="post">
    {{ form.email.label }}:
{{ form.email(placeholder='yourname@email.com') }}
    <br>
    {{ form.password.label }}: {{ form.password }}
    <br>
    {{ form.csrf_token }}
</form>
```

> **ملاحظة**
>
> استخدم class_="" إذا أردت استخدام الخاصيّة "class" في أحد الحقول، حيث أنَّ `class` كلمة محجوزة في بايثون.

> **ملاحظة**
>
> يحتوي توثيق إضافة WTForms على قائمة بالخواص المتوافرة التي يمكن تمريرها للحقول.

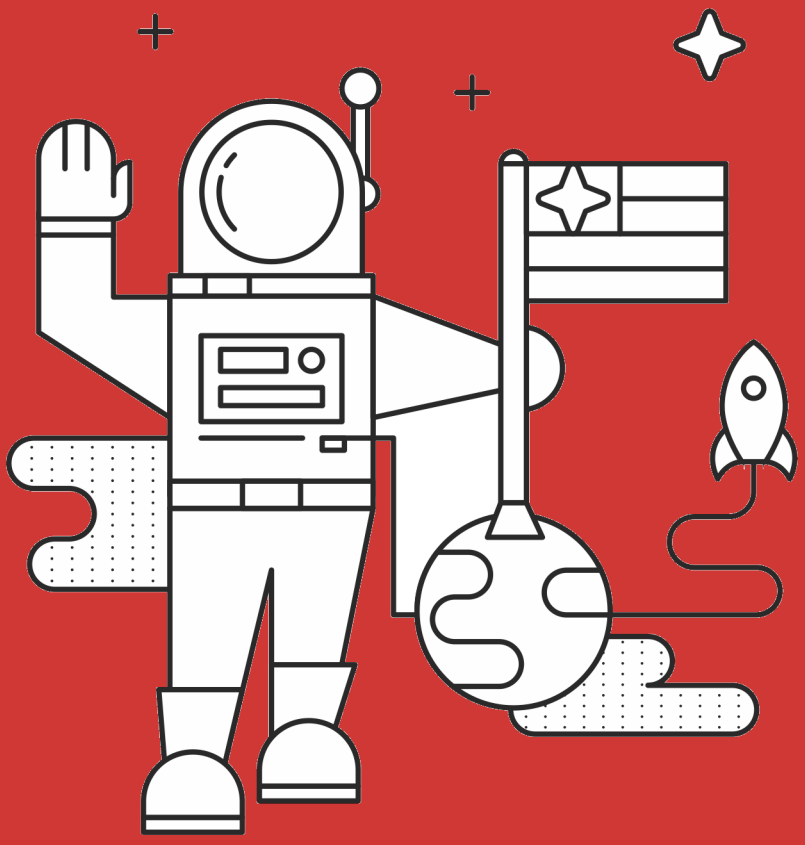
> **ملاحظة**
>
> قد تكون لاحظتَ أنَّه ليس علينا استخدام مُرشِّح جينجا `|safe` عند تصيير الحقول بواسطة إضافة WTForms، وهذا لأنَّ الإضافة تقوم بترشيح المُخرجات تلقائياً لتصييرها بشكل آمن.

ألقِ نظرة على توثيقات الإضافة للمزيد من التفاصيل.

الخلاصة

- يُمكِن أن تكون الاستمارات مُضرَّة ومصدر قلق من الناحية الأمنيّة.
- تجعل إضافة Flask-WTF من السهل تعريف وتأمين وتصيير الاستمارات.
- استخدم ميزة الحماية من ثغرة تزوير الطلبات التي توفرها إضافة Flask-WTF لتأمين استمارات تطبيقك.
- يمكنك استخدام إضافة Flask-WTF لتأمين طلبات الأجاكس أيضاً من هجمات تزوير الطلبات.
- استخدم المُصادِقات لإبقاء عمليّة المُصادقة معزولة خارج دوال العرض.
- استخدم ميّزة تصيير شيفرات حقول الإدخال التلقائيّة من إضافة WTForms لكيلا تضطر إلى تحديث جميع الحقول يدوياً في كل مرَّة تجري فيها تعديلات على تعريف الاستمارة.

الفصل الحادي عشر: أساليب للتعامل مع المستخدمين



أحد أكثر الأمور شيوعاً التي تحتاج تطبيقات الويب العصريّة إلى القيام بها هي التعامل مع المُستخدمين. فالتطبيق الذي يوفر أبسط مزايا التعامل مع حسابات المُستخدمين يحتاج للتعامل مع عشرات الأشياء في الخلفيّة، كإنشاء الحساب، وتأكيد البريد الإلكتروني، وتخزين كلمات المرور بشكل آمن، وإعادة تعيين كلمات المرور بشكل آمن، والمصادقة (authentication)، وغيرها من الأمور. وبما أنَّ العديد من المشاكل الأمنيّة تطوف في هذه الناحيّة، فمن الأفضل عليك استخدام الأساليب المعياريّة والشائعة في مشاريعك.

> **ملاحظة**
>
> أفترض في هذا القسم أنَّك تستخدم إضافة SQLAlchemy للتعامل مع قواعد البيانات وإضافة WTForms للتعامل مع الاستمارات. ستضطر إلى ملائمة الأساليب المذكورة هنا مع أدواتك إن لم تكن تستخدم ما سلف ذكره.

تأكيد البريد الإلكتروني

عادةً ما تقوم تطبيقات الويب بتأكيد البريد الإلكتروني المُدخَل من المُستخدمين الجدد للتأكد من أنَّه صالح وصحيح. فبعد التأكّد من صحة البريد، يمكننا إرسال روابط لإعادة تعيين كلمات المرور وغيرها من المعلومات الحساسة لمُستخدمينا براحة تامة وبدون القلق حول من يتلقى تلك الرسائل.

أحد أكثر الأساليب شيوعاً لإنجاز ذلك هو إرسال رابط لإعادة تعيين كلمة المرور يحوي جزئيّة فريدة، فعند الدخول إلى ذلك الروابط نتيقَّن من أنَّ بريد المُستخدِم صحيح. لنأخذ

مثال على ذلك: مُستخدِم بريده john@gmail.com قام بإنشاء حساب في تطبيقنا. قمنا بتسجيله في قاعدة البيانات بنجاح، وأضفنا القيمة `False` إلى العمود `email_confirmed`، ومن ثم أرسلنا رسالةً إلى بريده الإلكتروني (john@gmail.com) تحوي رابطاً بجزئيّة فريدة. تكون هذه الجزئيّة مميزة وبقيمة مُبعثرة (أو تبدو كذلك)، كهذه:

http://myapp.com/accounts/confirm/Q2hhZCBDYXRsZXR0IHJvY2tzIG15IHNvY2tz. سيضغط صاحب البريد (جون) عند استقباله للرسالة على الرابط المُرفَق ليُؤكِّد حسابه لدينا. سيستقبل التطبيق الجزئيّة الفريدة، ومنها سيعلم البريد الذي عليه تأكيده، وبعدها سيقوم بوضع القيمة `True` في العمود `email_confirmed` في حساب صاحب البريد.

ستتساءل، بالتأكيد، عن الكيفيّة التي سنعلم بها البريد الذي علينا تأكيده من تلك الجزئيّة. إحدى الطرق للقيام بذلك عبر حفظ الجزئيّة في قاعدة البيانات عند إنشائها وإعادة التأكّد منها عند تلقي طلب التأكيد. هذه الطريقة مُزعِجة وستجلب الكثير من الصداع. لحسن حظنا أنّنا لن نحتاج لها.

سنقوم بدلاً من ذلك بترميز البريد الإلكتروني بداخل الجزئيّة. ستحتوي الجزئيّة أيضاً على طابع زمني (timestamp) ليخبرنا بمدّة صلاحيّتها. سنستخدم مكتبة `itsdangerous` للقيام بذلك. تتيح لنا هذه المكتبة إمكانيّة إرسال بيانات حساسة إلى بيئات غير موثوقة (كإرسال رسالة تحوي رمز تفعيل إلى بريد إلكتروني غير مؤكّد، كما في حالتنا) بشكل آمن. سنستخدم الصنف `URLSafeTimedSerializer` هنا من هذه المكتبة.

```
# ourapp/util/security.py

from itsdangerous import URLSafeTimedSerializer

from .. import app

ts = URLSafeTimedSerializer(app.config["SECRET_KEY"])
```

سنستخدم المُسلسِل (serializer) لتوليد رمز تفعيل مُميّز عندما نحصل على البريد الإلكتروني للمُستخدِم. سننفذ أسلوب بسيط لإنشاء حسابات المُستخدمين بهذه الطريقة.

```
# ourapp/views.py

from flask import redirect, render_template, url_for

from . import app, db
from .forms import EmailPasswordForm
from .util import ts, send_email

@app.route('/accounts/create', methods=["GET", "POST"])
def create_account():
    form = EmailPasswordForm()
    if form.validate_on_submit():
        user = User(
            email = form.email.data,
            password = form.password.data
        )
        db.session.add(user)
        db.session.commit()
```

```
        # توجد هنا جزئيّة إرسال البريد الإلكتروني
        subject = "Confirm your email"

        token = ts.dumps(self.email, salt='email-confirm-key')

        confirm_url = url_for(
            'confirm_email',
            token=token,
            _external=True)

        html = render_template(
            'email/activate.html',
            confirm_url=confirm_url)

        # تم تعريفها في الوحدة (send_eamil) دعنا نفترض أنّ هذه الدالة
myapp/util.py
        send_email(user.email, subject, html)

        return redirect(url_for("index"))

    return render_template("accounts/create.html", form=form)
```

تقوم الدالة السابقة بإنشاء حساب المُستخدِم وإرسال رسالة إلى بريده الإلكتروني.

سنستخدم في مثالنا قالب لتوليد رسالة التأكيد الإلكترونيّة (انظر أدناه).

```
{# ourapp/templates/email/activate.html #}

Your account was successfully created. Please click the link below<br>
to confirm your email address and activate your account:
```

```
<p>
<a href="{{ confirm_url }}">{{ confirm_url }}</a>
</p>

<p>
--<br>
Questions? Comments? Email hello@myapp.com.
</p>
```

بقي الآن لدينا أن نقوم بكتابة دالة عرض لتستقبل طلبات التأكيد.

```
# ourapp/views.py

@app.route('/confirm/<token>')
def confirm_email(token):
    try:
        email = ts.loads(token, salt="email-confirm-key",
max_age=86400)
    except:
        abort(404)

    user = User.query.filter_by(email=email).first_or_404()

    user.email_confirmed = True

    db.session.add(user)
    db.session.commit()

    return redirect(url_for('signin'))
```

هذه الدالة مُجرَّد مثال بسيط. قمنا بدايةً بإضافة العبارة `try ... except` للتأكّد ممّا إذا كان رمز التفعيل المُستقبَل صالحاً. سيحتوي رمز التفعيل أيضاً على طابع زمني، ولهذا قمنا باستدعاء الدالة الصنفيّة `ts.loads()` لكي تُصدِر خطأً إذا كان الطابع الزمني المُستقبَل أكبر من الموضوع في المُعامِل `max_age`. قمنا بتعيين قيمة المُعامِل `max_age` في مثالنا إلى 86400 ثانية (أي 24 ساعة).

> **ملاحظة**
>
> يمكنك تنفيذ خاصيّة تحديث البريد الإلكتروني بطريقة مماثلة لتلك. كل ما عليك هو إرسال رابط التأكيد إلى البريد الإلكتروني الجديد وفيه رمز التفعيل ولكن هذه المرَّة يجب أن يحتوي على البريدان الإلكترونيان: القديم والجديد. وعند تلقي الطلب ستقوم بالتحقق ممّا إذا كان صحيحاً والتصرف تبعاً لذلك (تحديث البريد أو إرسال رسالة خطأ للمُستخدِم).

تخزين كلمات المرور

القاعدة رقم واحد في التعامل مع بيانات المُستخدمين هي تهشير (hash) كلمات مرورهم باستخدام خوارزميّة Bcrypt (أو خوارزميّة scrypt، ولكنني سأستخدم Bcrypt هنا) قبل تخزينها. فلا أحد يقوم بتخزين كلمات المرور كنص عادي. قيامك بهذا سيُحدِث مشاكل أمنيّة كبيرة وغير مُنصِفة بحق بيانات مُستخدميك. لا يوجد شيء شاق هنا، فكل الأمور الصعبة والمُعقدة قد أُختصِرَت ويُسِرَت لتُسهِل علينا الطريق. لا عذر لأن لا نتبع الممارسات المعياريّة بهذا الشأن.

> **ملاحظة**
>
> تعد منصة أواسب (OWASP) إحدى أكثر المنصات الموثوقة في المجالات المتعلقة بحمايّة تطبيقات الويب. احرص على إلقاء نظرة على توصياتهم بشأن كتابة كود آمن ونظيف (من المشاكل الأمنيّة).

سنستخدم هنا إضافة Flask-Bcrypt التي تتيح لنا إمكانيّة استخدام مكتبة bcrypt الشهيرة في موقعنا. هذه الإضافة هي مُجرَّد غلاف (wrapper) لمكتبة `py-bcrypt`، ولكنها تقوم ببعض التكتيكات الإضافية وتتعامل مع بضعة أمور من المُزعِج القيام بها يدوياً (كالتأكُّد من الترميز النصي لأكواد الهاش قبل مقارنتها).

```
# ourapp/__init__.py

from flask_bcrypt import Bcrypt

bcrypt = Bcrypt(app)
```

أحد الأسباب الذي يجعل خوارزميّة Bcrypt موصى بها بشدّة في تهشير البيانات أنَّها "مُتكيّفة مُستقبلياً"، بمعنى أنَّه يمكننا زيادة صعوبة التخمين على البيانات المُهشَّرة بواسطتها عبر زيادة القوّة الحاسوبيّة المُستخدمة في العمليّة. فكلّما زادت "الجولات" المُستخدمة لتهشير كلمة مرور كُلّما استغرقت عمليّة تخمينها وقتاً أطولاً. حيث إنْ قمنا بعشرين جولة تهشيريّة، مثلاً، على كلمة مرور قبل خزنِها سيضظر المهاجِم إلى فك تهشير الكلمة عشرين مرَّة أيضاً.

تذكر أنَّ وقت إنهاء العمليّة سيطول مع زيادة الجولات المُستخدمة. مما يعني أنَّه عليك الموازنة ما بين الأمان وقابليّة الاستخدام عند اختيارك لعدد الجولات الذي سيُجرى.

يعتمد عدد الجولات الذي يُمكِن إجرائه في مقدار مُعين من الوقت على الموارد الحاسوبيّة للخادِم التي يمكن للموقع استخدامها. عليك محاولة تجربة عدّة أرقام واختيار رقم يستغرق مابين 0.25 و 0.5 تقريباً لتهشير كلمة المرور الواحدة. عليك المحاولة أيضاً ألا يقل ذلك الرقم عن 12.

جرّب البريمج البايثوني البسيط التالي لاختيار الوقت المناسب لك عند تهشير كلمات المرور:

```
# benchmark.py

from flask_bcrypt import generate_password_hash

# عدِّل عدد الجولات (في المُعطى الثاني) لحين أن يصل الوقت المُستغرَق
# بين 0.25 و 0.5 ثانيّة في عمليّة التهشير الواحدة.
generate_password_hash('password1', 12)
```

والآن قم بتشغيل البريمج عبر أداة `time` لأنظمة يونكس لقياس الوقت المُستغرَق لتنفيذ العمليّة:

```
$ time python test.py

real    0m0.496s
user    0m0.464s
sys     0m0.024s
```

جربت تلك العمليّة على خادم صغير لدي ووجدت أنَّ 12 جولة تستغرق وقت مناسب ولهذا سأضبط مثالنا ليستخدم عدد الجولات ذاك.

```
# config.py

BCRYPT_LOG_ROUNDS = 12
```

والآن يمكننا كتابة جزئيّة عمليّة التهشير بعد إنهاء ضبط إعدادات الإضافة. يمكننا وضع هذه الجزئيّة يدوياً في دالة العرض التي تستقبل طلب التسجيل من استمارة إنشاء الحساب، ولكننا حينها سنضطر إلى إعادة العمليّة يدوياً في دالة استرجاع كلمة المرور وإعادة تعيينها، ولذلك سأقوم بهذا بطريقة مُختصرة وأكثر ديناميكيّة بحيث أنّنا لن نعيدها إطلاقاً في جميع عمليات تخزين كلمات المرور. سأقوم باستخدم خاصيّة **الضابِط** (setter) التي توفرها إضافة SQLAlchemy، حيث أنَّنا سنستخدمها لتقوم تلقائياً بتهشير كلمات المرور باستخدام Bcrypt قبل تخزينها عند استخدام السطر `user.password ='password1'` في أي مكان في تطبيقنا.

```
# ourapp/models.py

from sqlalchemy.ext.hybrid import hybrid_property

from . import bcrypt, db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(64), unique=True)
```

```
    _password = db.Column(db.String(128))

    @hybrid_property
    def password(self):
        return self._password

    @password.setter
    def _set_password(self, plaintext):
        self._password = bcrypt.generate_password_hash(plaintext)
```

استخدمنا هنا المُلحق hybird من إضافة SQLAlchemy لتعريف خاصيّة تملك عدّة دوال تُستدعى من نفس المُلحَق. يُستدعى الضابِط الذي عرفناه عندما نقوم بإسناد قيمة للخاصيّة `user.password`. يُهشِر الضابِط كلمات المرور المسنودة (للخاصيّة) ويقوم بتخزينها في العمود `_password` في جدول المُستخدِم. وبما أنّنا نستخدم الخاصيّة من المُلحق hybird فيمكننا الوصول بسهولة إلى كلمة المرور المُهشَّرة لاحقاً بواسطة `user.password`.

أصبح بإمكاننا الآن كتابة دالة العرض لنُخزِّن كلمات المرور بشكل آمن وبِكُل يُسِر.

```
# ourapp/views.py

from . import app, db
from .forms import EmailPasswordForm
from .models import User

@app.route('/signup', methods=["GET", "POST"])
```

```
def signup():
    form = EmailPasswordForm()
    if form.validate_on_submit():
        user = User(username=form.username.data,
password=form.password.data)
        db.session.add(user)
        db.session.commit()
        return redirect(url_for('index'))

    return render_template('signup.html', form=form)
```

المُصادقة

والآن بعد أن انتهينا من عمليّة تخزين كلمات المرور، وحرصنا على القيام بذلك بشكل آمن، علينا أن نتوجَّه إلى تنفيذ جزئيّة المُصادقة في التطبيق. سيكون سيناريو تسجيل الدخول كالآتي: سنصنع استمارة لنجعل المُستخدم يرسل اسمه وكلمة مروره (أو بريده الإلكتروني وكلمة مروره)، ومن ثُمَّ سنتأكَّد من أنَّ كلمة مروره التي أدخلها صحيحة. سنقوم بعد ذلك، إن صحَّ كل ما أدخله، بتعليمه على أنَّه مُستخدِم مُسجّل (ذو جلسة) عبر وضع كعكة (ملف تعريف ارتباط) في مُتصفحه. بواسطة تلك الكعكة سنستطيع تمييزه في المرّة المُقبلة التي يدخل فيها إلى الموقع.

سنقوم بدايةً بتعريف الصنف `UsernamePassword` باستخدام عناصر مكتبة `WTForms`.

```
# ourapp/forms.py

from flask_wtf import Form
from wtforms import StringField, PasswordField
from wtforms.validators import DataRequired


class UsernamePasswordForm(Form):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
```

سنقوم بعدها بتعريف دالة صنفيّة جديدة في نموذج قاعدة البيانات `User` لتقوم بمقارنة نص كلمة المرور وكلمة المرور المُهشَّرة الموجودة في قاعدة البيانات.

```
# ourapp/models.py

from . import db

class User(db.Model):

    # [...] الأعمدة والخصائص الأخرى هنا

    def is_correct_password(self, plaintext)
        return bcrypt.check_password_hash(self._password, plaintext)
```

إضافة Flask-Login

خطوتنا التالية هي إنشاء دالة العرض التي ستعرض صفحة تسجيل الدخول وستستقبل بيانات استماراتها. ستقوم الدالة، ببساطة، بمُصادقة المُستخدِم بعد التأكّد من بياناته المُدخلة عبر إضافة Flask-Login. تُبسِط تلك الإضافة عمليّة التعامل مع المُستخدمين والمُصادقة.

هناك بعض الأشياء التي علينا إعدادها قبل استخدام إضافة Flask-Login لجعلها جاهزة للعمل.

سنقوم بتعريف الدالة `login_user` في الملف *__init__.py*.

```
# ourapp/__init__.py

from flask_login import LoginManager

# الأكواد المتعلقة بإنشاء وإعداد التطبيق
# [...]

from .models import User

login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view =  "signin"

@login_manager.user_loader
def load_user(userid):
    return User.query.filter(User.id==userid).first()
```

قمنا أعلاه بإنشاء كائن مُشتَق عن `LoginManager` واستهلاله بواسطة الكائن app، ومن ثُمَّ تعريف المتغيّر الصنفي `login_view` الذي يُخبِر الإضافة بالدالة التي عليه الحصول بواسطتها على مُعرِّف (`id`) المُستخدم. تلك هي الأمور الأساسيّة التي عليك تعريفها قبل البدء باستخدام الإضافة.

> **ملاحظة**
>
> انظر إلى المزيد من الطرق لتخصيص إضافة Flask-Login.

الآن سنقوم بتعريف دالة العرض `signin` (التي مرّرناها سابقاً للإضافة) التي ستتولى مهمة القيام بالمُصادقة.

```
# ourapp/views.py

from flask import redirect, url_for

from flask_login import login_user

from . import app
from .forms import UsernamePasswordForm

@app.route('/signin', methods=["GET", "POST"])
def signin():
    form = UsernamePasswordForm()

    if form.validate_on_submit():

        user =
User.query.filter_by(username=form.username.data).first_or_404()
```

```
        if user.is_correct_password(form.password.data):
            login_user(user)

            return redirect(url_for('index'))
        else:
            return redirect(url_for('signin'))
    return render_template('signin.html', form=form)
```

قمنا بمُنتهى البساطة باستيراد الدالة `login_user` من الإضافة واستخدامها للتحقق من بيانات المُستخدِم عبر استدعاء `login_user(user)`. يمكنك إضافة خاصيّة لتسجيل خروج المُستخدِم الحالي عبر استدعاء الدالة `logout_user()`.

```
# ourapp/views.py

from flask import redirect, url_for
from flask_login import logout_user

from . import app

@app.route('/signout')
def signout():
    logout_user()

    return redirect(url_for('index'))
```

خاصيّة "إعادة تعيين كلمة المرور"

غالباً ما ستقوم بإضافة ميزة "إعادة تعيين كلمة المرور" لموقعك. تتيح هذه الميزة لمستخدميك استرجاع حساباتهم، في حال قاموا بنسيان كلمة مرورهم مثلاً، عبر بريدهم الإلكتروني. يعج تطبيق هذه الميزة بالمخاطر والثغرات المحتلمة، وهذا لأنَّ الفكرة وما فيها تقوم على جعل شخص غير مُصادَق بالوصول إلى حساب شخصي لأحد مُستخدميك. سننفذ هذه الميزة باستخدام تكتيكات مماثلة للتي استخدمناها في قسم "تأكيد البريد الإلكتروني".

سنحتاج إلى استمارتين: واحدة لطلب استرجاع الحساب الشخصي وأخرى لاختيار كلمة المرور الجديدة بعد التأكّد من أنَّ صاحب الطلب لديه وصول لبريد الحساب الإلكتروني. أفترض في هذا القسم أنَّ نموذج قاعدة بياناتك يحوي عموداً لكلمة المرور وآخر للبريد الإلكتروني، حيث أنَّ عمود كلمات المرور يستخدم خاصيّة hybird كالتي أنشأناها سابقاً.

> **ملاحظة**
> لا تُرسِل رابط لإعادة تعيين كلمة مرور إلى بريد إلكتروني غير مؤكَّد، فعليك الحرص أنَّك تُرسِل الرابط للشخص المُناسِب.

يوجد أدناه تعريف للصنفين الذين سنستخدمهما في الاستماراتين اللتين سنحتاجهما لتنفيذ الخاصيّة.

```
# ourapp/forms.py

from flask_wtf import Form
from wtforms import StringField, PasswordField
from wtforms.validators import DataRequired, Email

class EmailForm(Form):
    email = StringField('Email', validators=[DataRequired(), Email()])

class PasswordForm(Form):
    password = PasswordField('Password', validators=[DataRequired()])
```

تفترض الشيفرة أعلاه أنّنا سنستخدم حقلاً واحداً في استمارة إدخال كلمة المرور الجديدة. تَطلُب العديد من التطبيقات من مُستخدميها إدخال كلمات مرورهم الجديدة مرّتين للتتأكّد أنَّهم لم يقعوا في أيّة أخطاء إملائيّة أثناء كتابتها. تحقيق ذلك لن يُكلِّف الكثير، فقط أضف حقلاً آخراً من النوع `PasswordField` وأضف المُصادِق `EqualTo` إلى حقل إدخال كلمة المرور الأوّل.

> **ملاحظة**
>
> توجد العديد من المُناقشات المفيدة في مجتمع "تجربة المُستخدِم" حول أفضل طريقة يُمكِن أن تُستخدم للتعامل مع استمارات تسجيل الدخول. أُفضِل شخصياً أفكار المُستخدِم روجر أتريل (Roger Attrill) الذي يقول:
>
> "لا يجدر بنا السؤال عن كلمة المرور مرّتين، فينبغي علينا أن نسأل عنها مرَّة واحدة ونتأكّد أن تعمل ميزة 'إعادة تعيين كلمة المرور' بسلاسة ومثاليّة."

> – اقرأ المزيد حول هذا الشأن في موضوعه في مجتمع تجربة المُستخدِم.
> – يوجد أيضاً بعض الأفكار الرائعة حول تبسيط استمارات تسجيل الدخول وإنشاء الحساب في هذا الموضوع.

سنقوم الآن بكتابة دالة العرض للاستمارة الأولى، والتي يقوم فيها المُستخدِم بطلب إرسال رابط لاسترجاع حسابه إلى بريده الإلكتروني.

```
# ourapp/views.py

from flask import redirect, url_for, render_template

from . import app
from .forms import EmailForm
from .models import User
from .util import send_email, ts

@app.route('/reset', methods=["GET", "POST"])
def reset():
    form = EmailForm()
    if form.validate_on_submit():
        user =
User.query.filter_by(email=form.email.data).first_or_404()


        subject = "Password reset requested"

        # الذي URLSafeTimedSerializer هنا حيث سنقوم باستخدام الصنف
قُمنا بإنشائه

        # في بداية الفصل.
```

```
        token = ts.dumps(user.email, salt='recover-key')

        recover_url = url_for(
            'reset_with_token',
            token=token,
            _external=True)

        html = render_template(
            'email/recover.html',
            recover_url=recover_url)

        # تم تعريفها في الوحدة (send_eamil) دعنا نفترض أنّ هذه الدالة
myapp/util.py
        send_email(user.email, subject, html)

        return redirect(url_for('index'))
    return render_template('reset.html', form=form)
```

ستقوم الاستمارة عند تلقيها للبريد الإلكتروني بتحديد المُستخدم (عبر البريد المُدخَل)، ومن ثُمَّ ستولِّد رمزاً فريداً وستُرسِل للبريد الإلكتروني رابطاً لإعادة تعيين كلمة المرور. سيوجّه الرابط صاحب البريد إلى دالة عرض لتقوم بالتأكّد من صلاحيّة الرمز وتجعل المُستخدِم من بعدها يُعيِّن كلمة مروره الجديدة.

```
# ourapp/views.py

from flask import redirect, url_for, render_template

from . import app, db
from .forms import PasswordForm
from .models import User
from .util import ts

@app.route('/reset/<token>', methods=["GET", "POST"])
def reset_with_token(token):
    try:
        email = ts.loads(token, salt="recover-key", max_age=86400)
    except:
        abort(404)

    form = PasswordForm()

    if form.validate_on_submit():
        user = User.query.filter_by(email=email).first_or_404()

        user.password = form.password.data

        db.session.add(user)
        db.session.commit()

        return redirect(url_for('signin'))

    return render_template('reset_with_token.html', form=form,
token=token)
```

سنستخدم نفس طريقة التأكّد من الرمز التي استخدمناها سابقاً مع ميزة تأكيد البريد الإلكتروني. حيث ستقوم دالة العرض بتمرير رمز التأكيد من الرابط إلى القالب، وبعدها سيستخدم القالب الرمز لإرسال الاستمارة إلى دالة العرض. إليك الكود المُستخدَم في القالب:

```
{# ourapp/templates/reset_with_token.html #}

{% extends "layout.html" %}

{% block body %}
<form action="{{ url_for('reset_with_token', token=token) }}"
method="POST">
    {{ form.password.label }}: {{ form.password }}<br>
    {{ form.csrf_token }}
    <input type="submit" value="Change my password" />
</form>
{% endblock %}
```

الخلاصة

- استخدم مكتبة `itsdangerous` لإنشاء والتأكّد من صلاحيّة رموز التأكيد المُرسلة إلى بريد الإلكتروني لمُستخدميك.
- يُمكِنك استخدام طريقة رموز التأكيد للتحقق من البريد الإلكتروني عندما يقوم المُستخدِم بإنشاء حسابه أو تعديل بريده أو استرجاع كلمة مروره.
- استخدم إضافة Flask-Login لمُصادِقة المُستخدمين لكي تتجنب إدارة بيانات الجلسة يدوياً.
- فكِّر دائماً بعقليّة المهاجم الذي قد يستغل تطبيقك لإحداث الأضرار.

الفصل الثاني عشر: النشر

أصبحنا، وأخيراً، مستعدين لعرض برمجيتنا للعالم أجمع، إنَّه وقت النشر! يمكن لهذه العمليّة أن تكون صعبة ومتعبة لوجود العديد من الأمور المعقدة، كما وتوجَد العديد من الخيارات التي عليك اتخاذها حيال البرمجيات المستعملة في بيئتك الإنتاجيّة. سنتحدث في هذا الفصل عن بعض النقاط المهمة والخيارات المتاحة لاتخاذها.

الاستضافة

سنحتاج في هذه العمليّة إلى خادم لنضع عليه برمجيتنا. يوجد الآلاف من مزودي هذا النوع من الخدمات، ولكن هناك ثلاثة فقط أوصي باستخدامهم. لن أقوم بالتعمق في تفاصيل استخدام هذه الاستضافات، فهذا خارج نطاق كتابنا. فبدلاً من ذلك سأتكلم عن الفوائد والمزايا التي تقدمها كل استضافة عند استخدامها مع البرمجيات المكتوبة بفلاسك.

مجموعة خدمات الويب من أمازون (EC2)

تقدم هذه الخدمة مجموعة من الخدمات المُقدمة من شركة أمازون العالميّة. هناك احتمال لسماعك بهذه الخدمة من قَبل؛ كونها من أكثر الخيارات شعبيّةً للشركات والبرمجيات الناشئة هذه الأيام. سنتحدث هنا عن خدمة EC2 بوجه التحديد، والتي تعني: "خدمة الحوسبة السحابيّة المرنة" (Elastic Compute Cloud). النقطة المثيرة بشأن هذه الخدمة أنَّ الخوادم الافتراضيّة التي تقدمها، أو **النماذِج** (instances) كما تُسميهم الشركة، تعمل على مبدأ الثواني. مما يعني أنَّه يمكنك توسيع إمكانيات تطبيقك عبر زيادة

عدد النماذج وربطهم بموزِّع حمل (load balancer) وحسب (يمكنك استخدام موزِّع الحمل المُقدَم من الشركة نفسها إن أردت).

لا تحتاج لإضافة أي شيء عند استخدام هذه الخدمة مع البرمجيات المكتوبة بفلاسك. كل ما عليك هو تشغيل نموذج يعمل بتوزيعتك اللينكساويّة المفضلة بشكل تقليدي وتثبيت موقعك وحزمة برمجيات خادم الويب التي تريدها بدون أي تعقيدات، مما يعني أيضاً أنَّه يجب أن تملك مهارات إدارة الأنظمة اللينكساويّة لتتمكن من إعداد نظامك.

استضافة هيروكو

هيروكو هي خدمة استضافة تطبيقات تعتمد على خدمات أمازون كخدمة EC2. مما يتيح لنا الاستفادة من السهولة التي توفرها خدمات أمازون وبدون الحاجة لامتلاك خبرة في إدارة الأنظمة.

حيث يمكنك باستخدام هذه الاستضافة نشر تطبيقك بسلاسة عبر دفع التحديثات بواسطة مدير الإصدارات جيت إلى الخادم. سيوفر هذا عليك الكثير من العناء عندما لا تكون راغباً بإنجاز المهام الصعبة يدوياً، كالإتصال بالخادم، وتثبيت وإعداد البرمجيات التي ستستخدمها، وابتكار استراتيجيّة حكيمة للنشر. كل هذه السلاسة تأتي بسعر، ولكن هيروكو وأمازون تُقدِم بعض الخدمات المجانيّة أيضاً.

> **ملاحظة**
>
> يحتوي موقع هيروكو على درس يشرح كيفيّة نشر تطبيقات فلاسك عبر استضافتهم.

ملاحظة

عمليّة إدارة قواعد بياناتك يدوياً يمكن أن تستغرق وقتاً طويلاً كما وتحتاج إلى بعض الخبرة في المجال. إنَّها لفكرة سديدة أن تعرف القليل حول إدارة قواعد بيانات مشاريعك الجانبيّة، ولكنك قد ترغب أحياناً بتوفير الوقت والجهد عبر الاستعانة بمحترفين لأداء هذه المهام نيابةً عنك.

توفر كلاً من شركتي هيروكو وأمازون خدمات لإدارة قواعد البيانات. لم أجرب كلتا الخدمتين حقيقةً بعد، ولكني سمعت أموراً حسنة عن كلتاهما. أظن أنَّ تلك الخدمات تستحق أن توضع في عين الاعتبار إن رغبت يوماً بتأمين بياناتك والحفاظ عليها بدون القيام بذلك يدوياً.

- خدمة هيروكو لإدارة قواعد البيانات.
- خدمة أمازون لإدارة قواعد البيانات.

استضافة ديجيتال أوشن

استضافة ديجيتال أوشن (Digital Ocean) هي خدمة منافسة لخدمة أمازون ظهرت حديثاً. تتيح لك ديجيتال أوشن، كما في أمازون، إنشاء خوادم افتراضيّة، أو **قطيرات** (droplets) كما تُسميها الشركة، بسلاسة وسهولة. جميع القطيرات تعمل على وسائط تخزين من النوع ثابت الحالة (SSD)، وهي ميزة لا نحصل عليها في الباقات العادية في خدمة أمازون. أكبر ميزة برأي تقدمها هذه الخدمة هي الواجهة البسيطة والسهلة مقارنةً مع تلك التي تقدمها أمازون. تعد ديجيتال أوشن إحدى خياراتي المفضلة لاستضافة مشاريعي، وأنصحك بشدة أن تلقي نظرة عليها.

لا تختلف استضافة تطبيقات الفلاسك على هذه الخدمة عن الكيفيّة في خدمات أمازون. كل ما عليك هو بدء قطيّرة جديدة تعمل بتوزيعة لينُكس وتثبيت البرمجيات التي تحتاجها.

> **ملاحظة**
> ساهمت شركة ديجيتال أوشن بسخاء وتبرَّعت لهذا الكتاب. ولكن ذلك لا يعني أنَّ توصيتي لم تكن نابعة من تجربة شخصيّة، فأنا أؤكِّد أنَّها كذلك. فلو أنني لم أكن مُعجباً بخدماتهم لما كُنت طلبت منهم التبرَّع أصلاً.

البرمجيات

سنتكلم في هذا القسم عن بعض البرمجيات التي سنحتاج تثبيتها على خادمنا لإيصال تطبيقنا للعالم أجمع. البرمجيات الأساسيّة التي سنحتاجها هي خادم أمامي (front server) ليستقبل الطلبات ويمرِّرها لمُنفِذ التطبيق الذي يُشغِّل تطبيقنا. عادةً ما نحتاج أيضاً لاستخدام قاعدة بيانات ولذلك سنتكلم قليلاً عن الخيارات المتوفرة لذلك أيضاً.

مُنفِذ التطبيق

الخادم التطويري الذي نستخدمه عادةً لتشغيل تطبيقات فلاسك محلياً ليس جيداً بما فيه الكفايّة للتعامل مع طلبات كثيرة وحقيقيّة في الوضع الإنتاجي. يُفضَل عند نشر التطبيق في بيئة إنتاجيّة استخدام مُنفِذ تطبيقات (application runner) كغونيكورن (Gunicorn). يتعامل غونيكورن مع الطلبات بيُسِر ويهتم بالأمور المُعقدة كالتعامل مع تعدد المسارات الحاسوبيّة (threading).

عليك تثبيت الحزمة `gunicorn` قبل استخدام غونيكورن في بيئة افتراضيّة بواسطة مدير الحزم `pip`. يمكن تشغيل التطبيقات بسهولة مع هذه البرمجيّة، فكل ما يتطلبه الأمر تنفيذ تعليمة بسيطة.

```
# rocket.py

from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello World!"
```

والآن لنقوم بتشغيله بغونيكورن علينا ببساطة تنفيذ الأمر `gunicorn`.

```
(ourapp)$ gunicorn rocket:app
2014-03-19 16:28:54 [62924] [INFO] Starting gunicorn 18.0
2014-03-19 16:28:54 [62924] [INFO] Listening at: http://127.0.0.1:8000
(62924)
2014-03-19 16:28:54 [62924] [INFO] Using worker: sync
2014-03-19 16:28:54 [62927] [INFO] Booting worker with pid: 62927
```

ينبغي الآن أن تكون قادراً على رؤية العبارة "Hello World!" في متصفحك عند الدخول إلى الرابط *http://127.0.0.1:8000*.

لتشغيل الخادم بالخلفيّة (أي جعله يعمل كعفريت - daemon) يمكنك تمرير الخيار `-D` للأمر. وبهذه الطريقة سيظل يعمل حتى ولو أُغلِقت جلسة الطرفيّة الحالية.

قد نواجه صعوبة بإيجاد رقم العمليّة التي يعمل عليها غونيكورن عند جعله يعمل كعفريت. يمكننا لحل ذلك إخبار غونيكورن أنَّ عليه وضع رقم العمليّة التي سيعمل عليها في ملف لنتمكن من إيقافه لاحقاً من الدون البحث في قائمة العمليات. استخدم الخيار `-p <file>` لفعل ذلك.

```
(ourapp)$ gunicorn rocket:app -p rocket.pid -D
(ourapp)$ cat rocket.pid
63101
```

لإعادة تشغيل الخادم أو إيقافه استخدم الأمرين `kill -HUP` و `kill` على التوالي.

```
(ourapp)$ kill -HUP `cat rocket.pid`
(ourapp)$ kill `cat rocket.pid`
```

يعمل خادم غونيكورن افتراضياً على المنفذ 8000. يمكنك تغيير رقم المنفذ عبر استخدام الخيار `-b`.

```
(ourapp)$ gunicorn rocket:app -p rocket.pid -b 127.0.0.1:7999 -D
```

جعل الخادم يتلقى طلبات خارجيّة

> **تحذير**
>
> غونيكورن مخصص ليبقى خلف خادم وكيل عكسي (reverse proxy). فإذا ما أخبرته أن يتنصت على طلبات خارجيّة سيصبح هدفاً سهلاً لهجمات الحرمان من الخدمة. وذلك لأنَّه غير مخصص للتعامل مع هذا النوع من الطلبات. ولذلك لا تجعله يتنصت على الطلبات الخارجيّة إلا إن كنت تريد

> تنقيح تطبيقك، واحرص على جعله يتنصت على الطلبات الداخليّة مجدداً بعد أن تنتهي.

إن قمت بتشغيل غونيكورن كما بالطريقة التي فعلناها سابقاً لن تتمكن من الوصول إلى الخادم من حاسبك الشخصي (أي من خارج الخادم)، وهذا لأنَّ غونيكورن يتنصت افتراضياً على عنوان الشبكة 127.0.0.1، أي أنَّه يتنصت على الطلبات الداخليّة فقط (القادمة من الخادم نفسه). يعمل غونيكورن بهذا الشكل لأنَّه مخصص للاستخدام عندما يكون هناك خادم وكيل عكسي يعمل إلى جانبه. وعلى الرغم من ذلك، إن أردت يوماً إرسال طلبات خارجيّة إلى الخادم، لأهداف تنقيحيّة مثلاً (لتصحيح مشاكل في التطبيق)، يمكنك ضبط غونيكورن ليتنصت على العنوان 0.0.0.0، أي أن يستقبل جميع الطلبات القادمة له (الداخليّة والخارجيّة).

```
(ourapp)$ gunicorn rocket:app -p rocket.pid -b 0.0.0.0:8000 -D
```

> **ملاحظة**
>
> – اقرأ المزيد عن تشغيل ونشر التطبيقات بواسطة غونيكورن في توثيق المشروع.
>
> – تتيح أداة فابريك تشغيل جميع الأوامر التي استخدمناها على الخادم بأريحيّة من جهازك المحلي وبدون الإتصال بالخادم في كل مرَّة.

خادم إنجن إكس

يعمل خادم الوكيل العكسي على التعامل مع طلبات ميثاق نقل النص الفائق (HTTP) وإعادة إرسالها إلى غونيكورن مجدداً ومن ثمَّ إعادة الرد إلى العميل. يعمل إنجن إكس (Nginx) كخادم وكيل عكسي بفعاليّة، كما وينصح بشدّة استخدام غونيكورن معه.

لضبط إنجن إكس ليعمل مع خادم غونيكورن مضبوط ليتنصت على العنوان 127.0.0.1:8000 يمكننا إنشاء ملف لتطبيقنا في المسار *etc/nginx/sites-available/exploreflask.com/* يحتوي على:

```
# /etc/nginx/sites-available/exploreflask.com

# إعادة توجيه الطلبات القادمة من www.exploreflask.com إلى
exploreflask.com

server {
        server_name www.exploreflask.com;
        rewrite ^ http://exploreflask.com/ permanent;
}

# يتعامل مع الطلبات القادمة إلى العنوان exploreflask.com على المنفذ
80
server {
        listen 80;
        server_name exploreflask.com;

                # Handle all locations
```

```
        location / {
                        # Pass the request to Gunicorn
                proxy_pass http://127.0.0.1:8000;

                # Set some HTTP headers so that our app knows where
the
                # request really came from
                proxy_set_header Host $host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
        }
}
```

الآن سنقوم بإنشاء رابط ليّن (symlink) لهذا الملف في المسار *etc/nginx/sites-enabled/* وسنقوم بإعادة تشغيل إنجن إكس.

```
$ sudo ln -s \
/etc/nginx/sites-available/exploreflask.com \
/etc/nginx/sites-enabled/exploreflask.com
```

ينبغي الآن أن يكون إنجن إكس قادر على استقبال طلباتنا وإرسال الرد.

> **ملاحظة**
>
> يشرح قسم إعداد خادم إنجن إكس في توثيقات غونيكورن كيفيّة إعداد إنجن إكس ليعمل مع الخادم.

وحدة ProxyFix

قد تحدث مشاكل نتيجة عدم تعامل فلاسك مع الطلبات المرسلة بشكل صحيح. هذه المشاكل متعلقة بالترويسات التي قمنا بضبطها عند إعداد خادم إنجن إكس. لإصلاح تلك المشاكل يمكننا استخدام وحدة ProxyFix من مكتبة ويركزوغ.

```
# app.py

from flask import Flask

# Import the fixer
from werkzeug.contrib.fixers import ProxyFix

app = Flask(__name__)

# Use the fixer
app.wsgi_app = ProxyFix(app.wsgi_app)

@app.route('/')
def index():
        return "Hello World!"
```

> **ملاحظة**
>
> اقرأ المزيد حول ProxyFix في توثيقات ويركزوغ.

الخلاصة

- هناك ثلاث خدمات أنصحك بها لاستضافة مشاريعك المكتوبة بفلاسك: خدمة أمازون للحوسبة السحابيّة، وخدمة هيروكو، وخدمة ديجيتال أوشن.
- حزمة برمجيات الخادم الأساسيّة التي عليك استخدامها تتألف من تطبيق، ومُنفِذ كغونيكورن، وخادم وكيل عكسي كإنجن إكس.
- ينبغي أن يعمل خادم غونيكورن خلف خادم إنجن إكس وينبغي أن يتنصت على العنوان 127.0.0.1 (الطلبات الداخليّة) وليس على العنوان 0.0.0.0 (الطلبات الخارجيّة).
- استخدم الوحدة ProxyFix من مكتبة ويركزوغ للتعامل مع الترويسات الصحيحة فقط في تطبيق الفلاسك خاصتك.

الخاتمة

لا أشعر أنه يوجد الكثير لأختتمه. آمل أن يساعدك الكتاب في رحلتك مع فلاسك. إن فعل وأفادك، رجاءً تواصل معي! أود أن أستمع لأناس أستمتعوا بقراءة هذا الكتاب. لا تتردد أيضاً عن التواصل معي إذا كنت تملك أية إقتراحات لتحسين الكتاب.

شكراً على قراءتك!

- روبرت