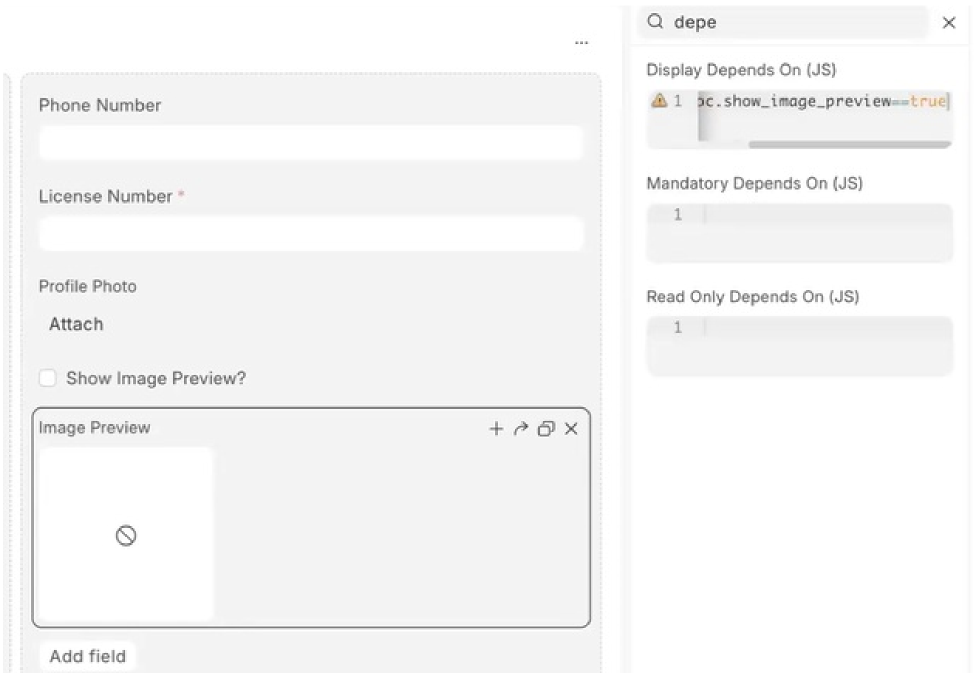
Day 3

Created	@January 14, 2024 5:17 PM
Status	Open
Updated	@September 17, 2024 5:47 PM

Client-Scripting (JS):

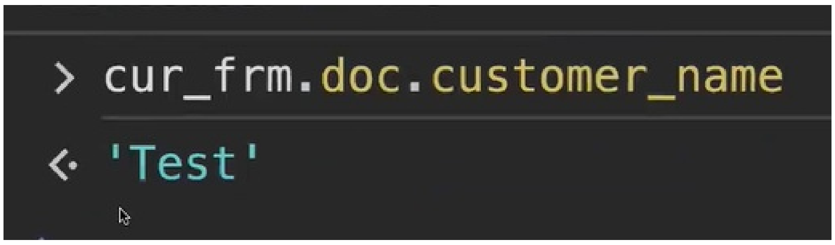


Display Depends On (JS)

```
1 oc.show_image_preview==true
```

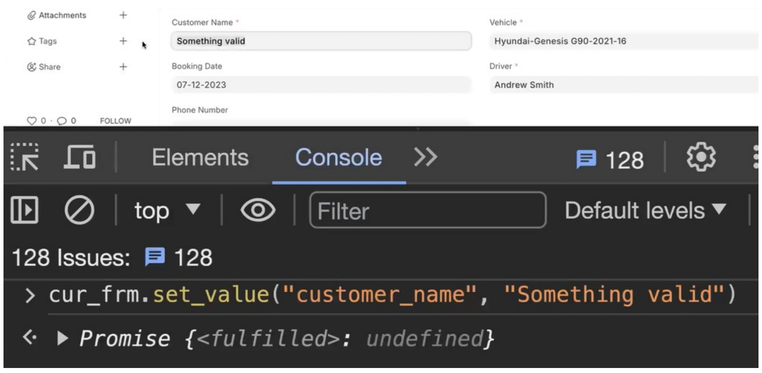
There are options `Depends On` for every field, e.g. `Display Depends On` that will only show the field if the evaluation inside it is true (`eval:doc.show_image_preview==true`). Else, the field will be hidden.

We can even export custom doctypes using Packages if we don't want to migrate it using the site database.



```
> cur_frm.doc.customer_name
<· 'Test'
```

Whenever we are in a form view, we have a special global variable available (for js) called `cur_frm` that refers to the current form we are viewing.

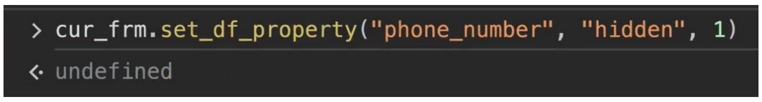


Customer Name *	Something valid
Vehicle *	Hyundai-Genesis G90-2021-16
Booking Date	07-12-2023
Driver *	Andrew Smith
Phone Number	

```
128 Issues: 128
> cur_frm.set_value("customer_name", "Something valid")
<· ▶ Promise {<fulfilled>: undefined}
```

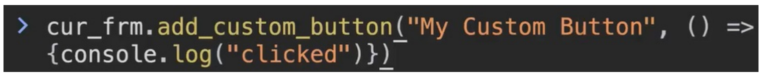
We can also use methods available for the form. For example, we can use .set_value() to change the value of a field. It's not automatically saved, so we can either press the `Save` button or call .save() method.

Example to hide a field:

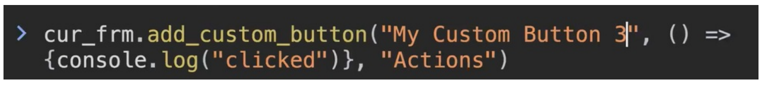


```
> cur_frm.set_df_property("phone_number", "hidden", 1)
<· undefined
```

Example to add a custom button:

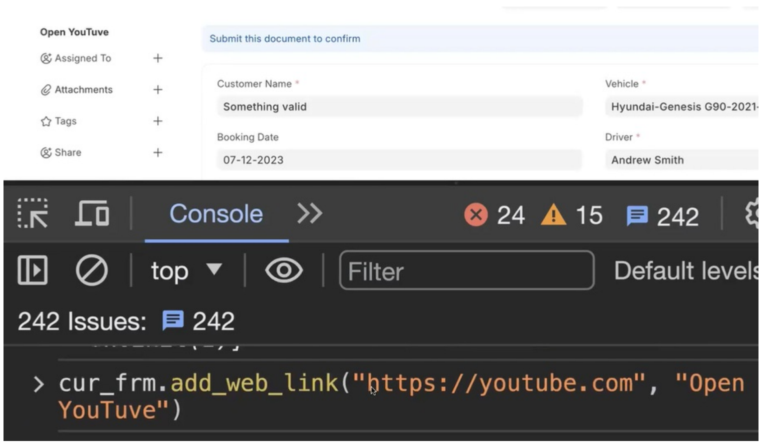


```
> cur_frm.add_custom_button("My Custom Button", () =>
  {console.log("clicked")})
```

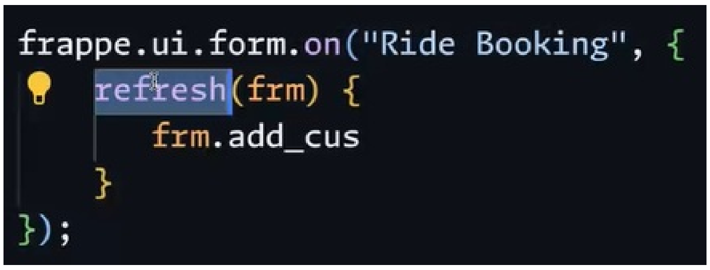


```
> cur_frm.add_custom_button("My Custom Button 3", () =>
  {console.log("clicked")}, "Actions")
```

Example to add a web link to the sidebar:



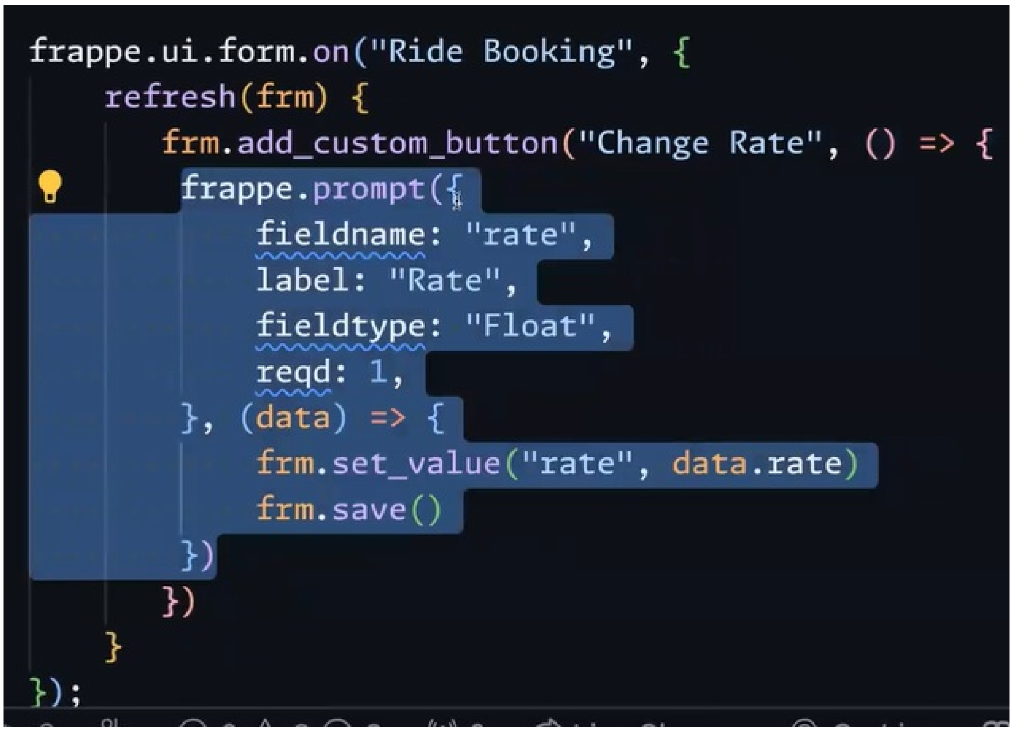
```
> cur_frm.add_web_link("https://youtube.com", "Open
  YouTuve")
```



```
frappe.ui.form.on("Ride Booking", {
    refresh(frm) {
        frm.add_cus
    }
});
```

We can add those console code into the .js file so Frappe can run it every time. For example, here we put it inside `refresh` trigger which will run the code every time the form refreshes. `frm` in the .js is equivalent to `cur_frm` in the console.

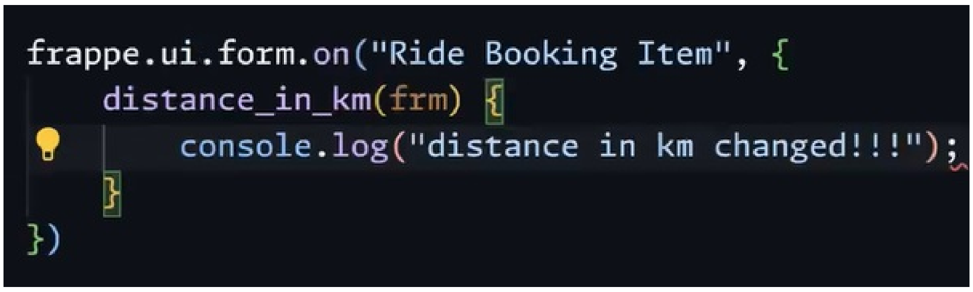
Example for a dialog to change a field's value:



```
frappe.ui.form.on("Ride Booking", {
    refresh(frm) {
        frm.add_custom_button("Change Rate", () => {
            frappe.prompt({
                fieldname: "rate",
                label: "Rate",
                fieldtype: "Float",
                reqd: 1,
            }, (data) => {
                frm.set_value("rate", data.rate)
                frm.save()
            })
        })
    }
});
```

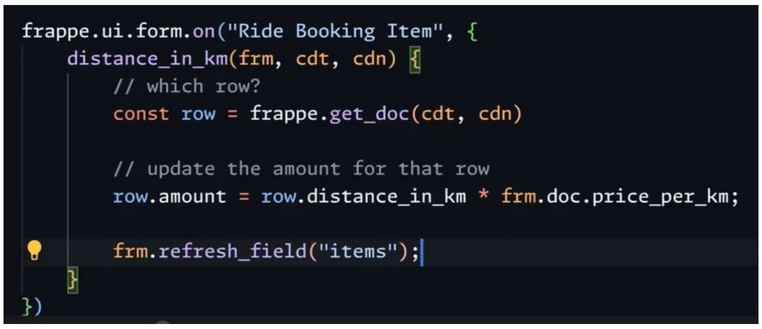
Child-table client-scripting (JS):

We can also put client scripts for the child table that exists inside `Ride Booking`. To do it, just use the same frappe.ui.form syntax, but change it to `Ride Booking Item`.



```
frappe.ui.form.on("Ride Booking Item", {
    distance_in_km(frm) {
        console.log("distance in km changed!!!");
    }
})
```

The code above will trigger every time `distance_in_km` changes.



```
frappe.ui.form.on("Ride Booking Item", {
    distance_in_km(frm, cdt, cdn) {
        // which row?
        const row = frappe.get_doc(cdt, cdn)

        // update the amount for that row
        row.amount = row.distance_in_km * frm.doc.price_per_km;

        frm.refresh_field("items");
    }
})
```

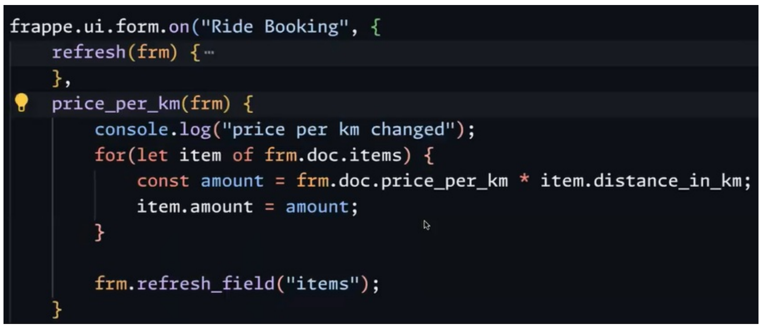
In child table client script, we also get 2 more variables: cdt & cdn.

`cdt`: current doctype (e.g. Ride Booking Item)

`cdn`: current doctype `name` (e.g. 3d8884yd3a)

So, the code above will change the `row.amount` internally, but it won't show the changes live on the UI. Therefore, we need to call `frm.refresh_field()` to refresh the child table and show the changes live on the UI.

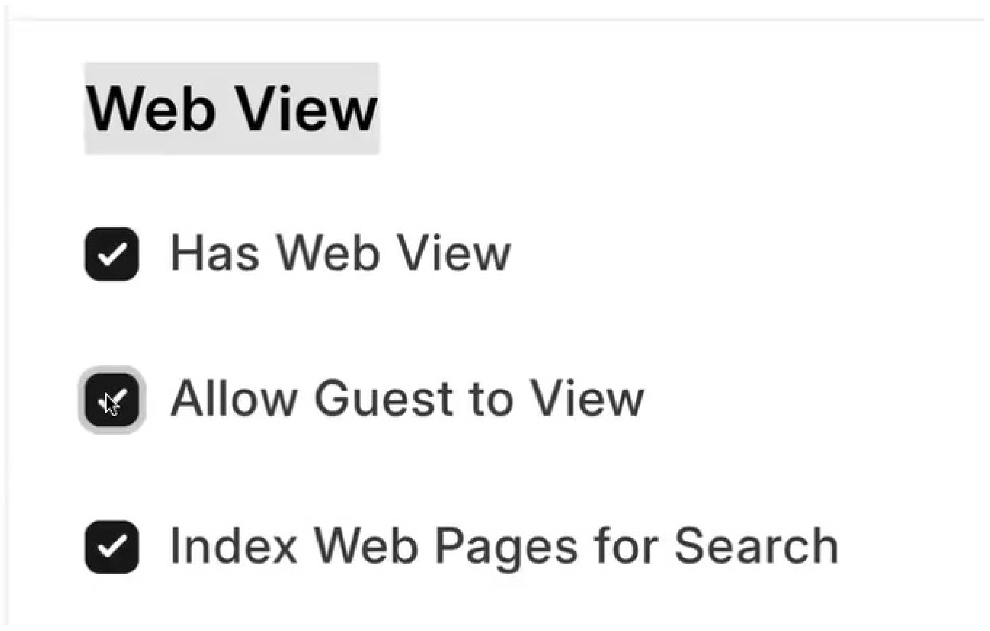
Example to update the amount when a form field changes:



```
frappe.ui.form.on("Ride Booking", {
    refresh(frm) {⋯
    },
    price_per_km(frm) {
        console.log("price per km changed");
        for(let item of frm.doc.items) {
            const amount = frm.doc.price_per_km * item.distance_in_km;
            item.amount = amount;
        }

        frm.refresh_field("items");
    }
```

When we loop the child table, we immediately get the child object, so we set using `item.amount` instead of using `frm.set_value` (frm.set_value is only for the form field).

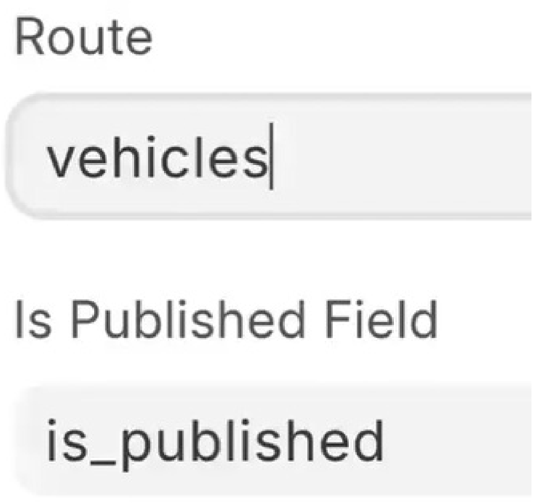
Portal Development:



Web View

- [x] Has Web View
- [x] Allow Guest to View
- [x] Index Web Pages for Search

We first need to check `Has Web View` to enable the doctype records to have web pages. `Allow Guest to View` enables non-authenticated users to access the web page.



Route: vehicles

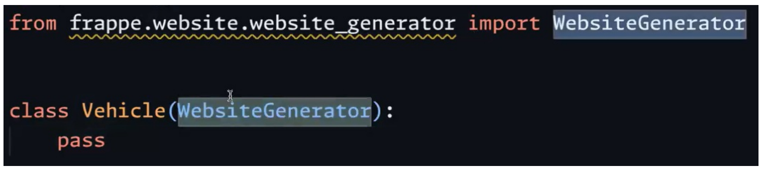
Is Published Field: is_published

Then, we need to determine `Route` (not a field) for the web view, where do we want to route the list view of this doctype?

We also need to have a check field that determines whether the record will be published or not.

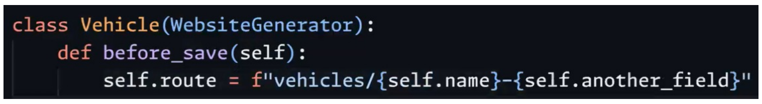
For the detail route, we need to add a field named `route`.

Files Generated & Changed (Web Views):

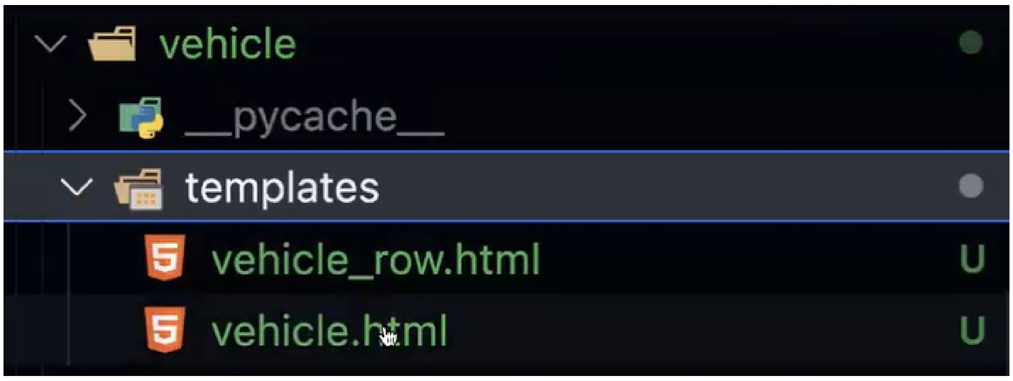


```python
from frappe.website.website_generator import WebsiteGenerator

class Vehicle(WebsiteGenerator):
    pass
```

When we enable web views, the class for the controller **automatically changes from** `Document` **to** `WebsiteGenerator` (extends from Document). This way, we can set the route ourselves:



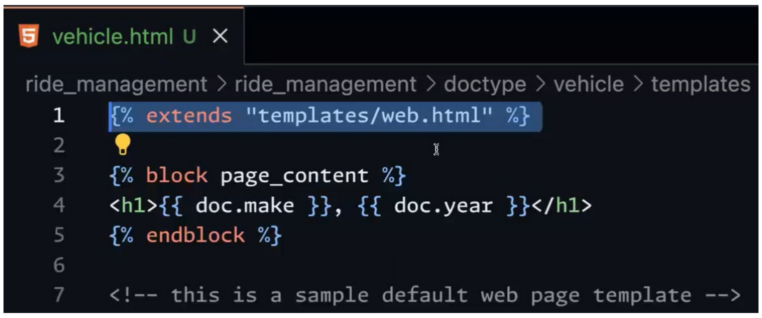
```python
class Vehicle(WebsiteGenerator):
    def before_save(self):
        self.route = f"vehicles/{self.name}-{self.another_field}"
```



```
vehicle
  __pycache__
  templates
    vehicle_row.html    U
    vehicle.html        U
```

Also, whenever we enable web views for a doctype, **a folder is auto-generated called** `templates` which has 2 html files inside: [doctype].html (for detail page) & [doctype]_row.html (for list view page)

WebView file structure



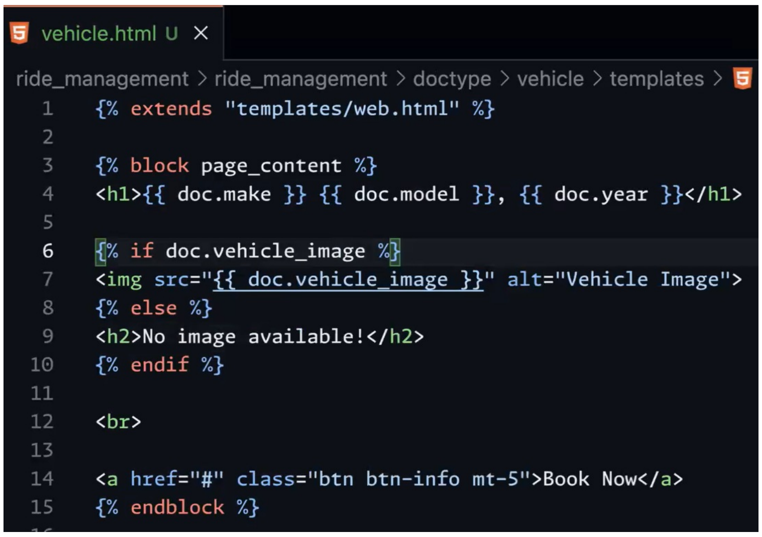
vehicle.html U ×

ride_management > ride_management > doctype > vehicle > templates

```html
{% extends "templates/web.html" %}

{% block page_content %}
<h1>{{ doc.make }}, {{ doc.year }}</h1>
{% endblock %}

<!-- this is a sample default web page template -->
```

The header, footer, etc. came from the `extends` line that extends the web.html from frappe and block page_content.

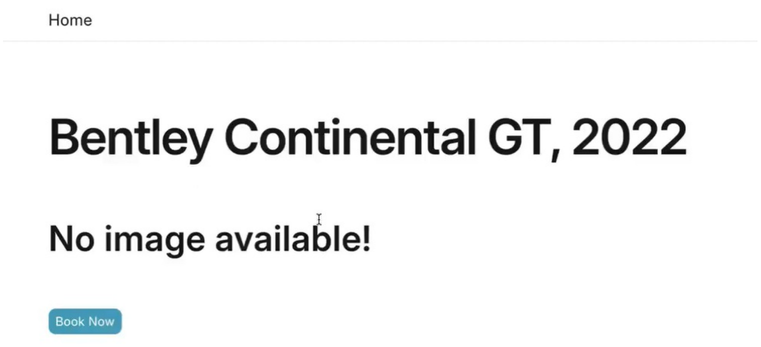
Example use of html, bootstrap, and Jinja2:



vehicle.html U ×

ride_management > ride_management > doctype > vehicle > templates >

```html
{% extends "templates/web.html" %}

{% block page_content %}
<h1>{{ doc.make }} {{ doc.model }}, {{ doc.year }}</h1>

{% if doc.vehicle_image %}
<img src="{{ doc.vehicle_image }}" alt="Vehicle Image">
{% else %}
<h2>No image available!</h2>
{% endif %}

<br>

<a href="#" class="btn btn-info mt-5">Book Now</a>
{% endblock %}
```

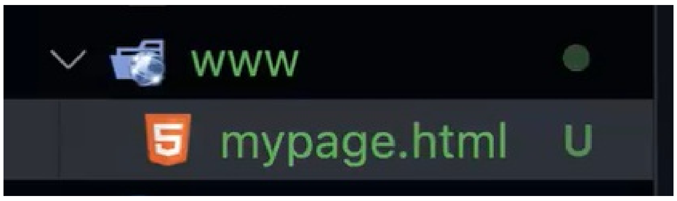

Home

Bentley Continental GT, 2022

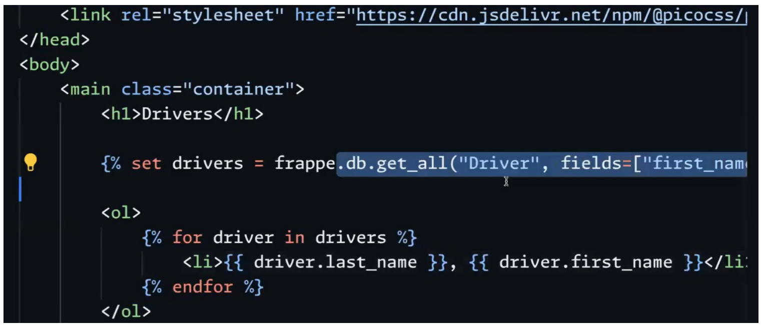
No image available!

Custom Arbitrary WebPage:

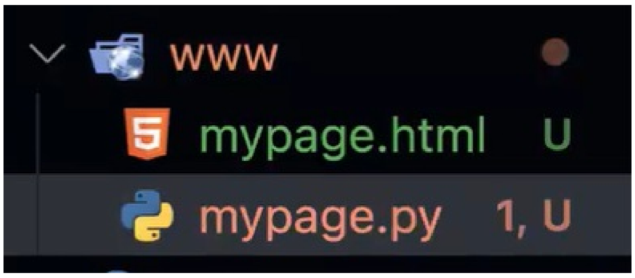
Frappe allows us to have full control and build our own arbitrary web pages using custom app through the `www` directory.



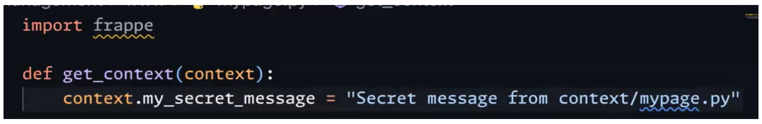
For the route, frappe uses file-based routing. So, we can access the web page using the filename as the route (mypage.html is accessible through [site-url]/mypage). Here we can import 3rd-party stuff like Pico CSS, we can use Jinja2 that has access to Frappe API (Jinja API) as well. Example:


```
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/@picocss/
</head>
<body>
    <main class="container">
        <h1>Drivers</h1>

        {% set drivers = frappe.db.get_all("Driver", fields=["first_nam

        <ol>
            {% for driver in drivers %}
                <li>{{ driver.last_name }}, {{ driver.first_name }}</li
            {% endfor %}
        </ol>
```

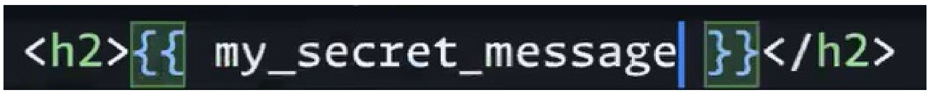
We can also add a .py file to get context for the .html if Jinja API is not enough.



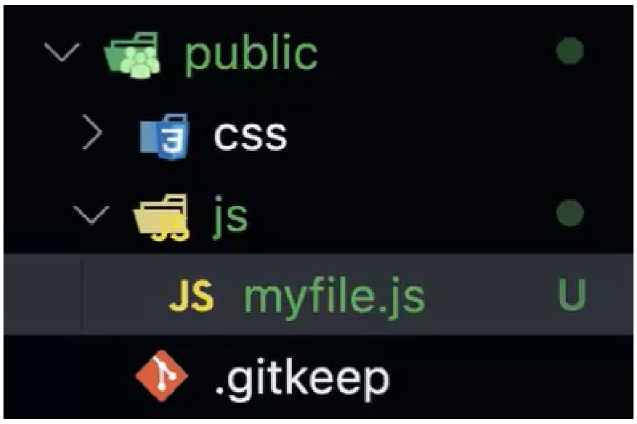
So, this way, we can do whatever we want that is possible through Python (e.g. import pandas etc.):


```
import frappe

def get_context(context):
    context.my_secret_message = "Secret message from context/mypage.py"
```

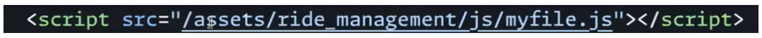
`context` will bring/pass the data to be accessible on our .html


```
<h2>{{ my_secret_message }}</h2>
```

Linking Static Assets:



We can also add a .js file to the www directory, but we would need to extend the base template to make it work. So, a better way is to use the `public` directory where we can put any static assets (css, js, etc.) and use it on our .html. However, we have a special route to access the file:

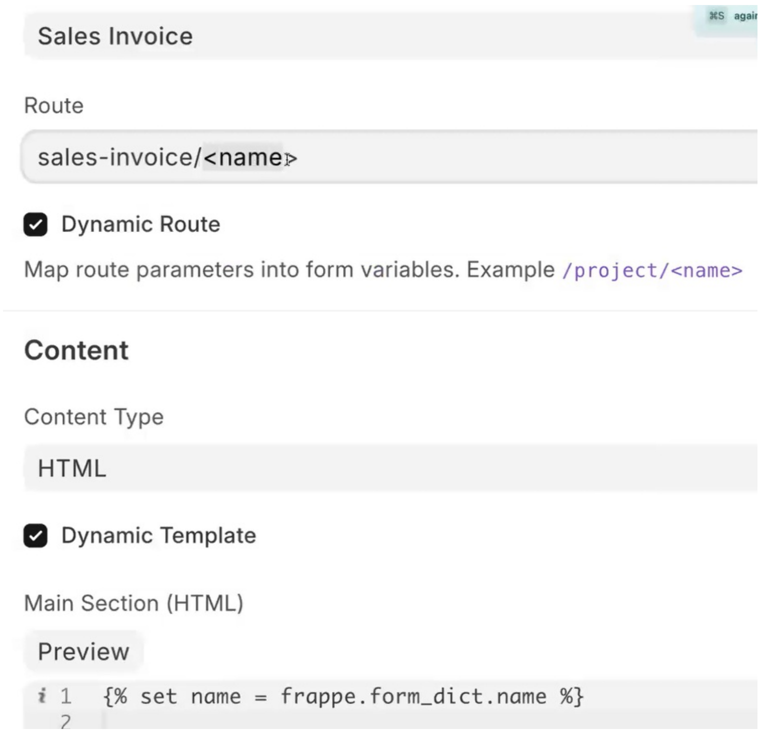


```
<script src="/assets/ride_management/js/myfile.js"></script>
```

`/assets/[app-name]/js/[file-name].js` (so we don't write 'public', frappe does a special thing here). These static assets are served by `nginx` so they are fast.

Dynamic Route & Template:

We can also access the route as variables by creating dynamic pages:



Sales Invoice

Route

sales-invoice/<name>

☑ Dynamic Route

Map route parameters into form variables. Example /project/<name>

Content

Content Type

HTML

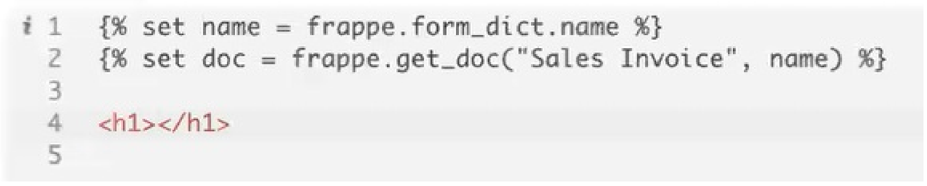
☑ Dynamic Template

Main Section (HTML)

Preview

```
{% set name = frappe.form_dict.name %}
```

By using dynamic route, we can access the `name` on our Jinja2 using Dynamic Template. Use the syntax `frappe.form_dict` and we can access the variables.

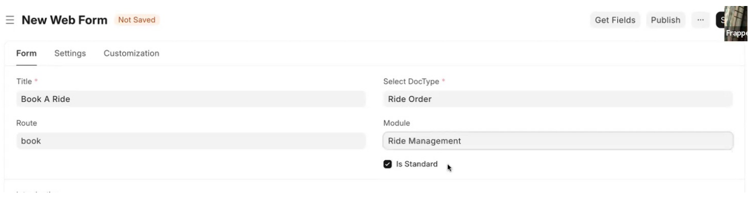


```
{% set name = frappe.form_dict.name %}
{% set doc = frappe.get_doc("Sales Invoice", name) %}

<h1></h1>

```

Web Form:

Web form lets us collect information through our portal.

There are 3 options we can put in Data field: Email, Phone, URL


New Web Form Not Saved

Get Fields Publish ···

Form Settings Customization

Title *: Book A Ride

Select DocType *: Ride Order

Route: book

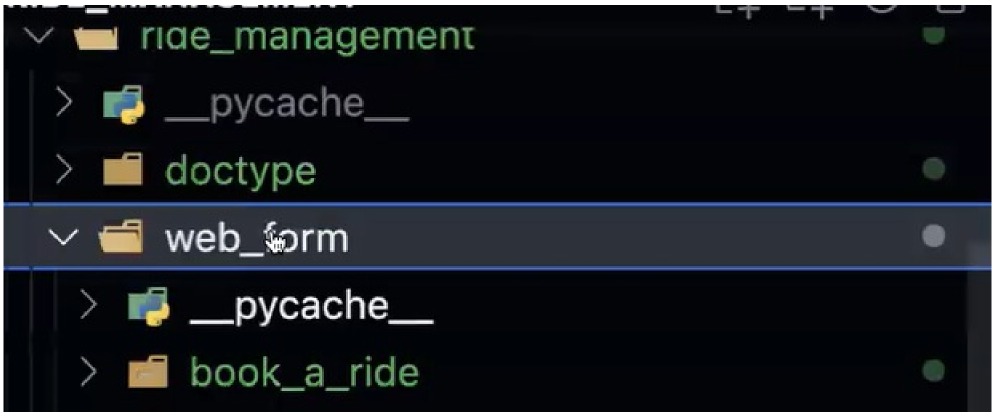
Module: Ride Management

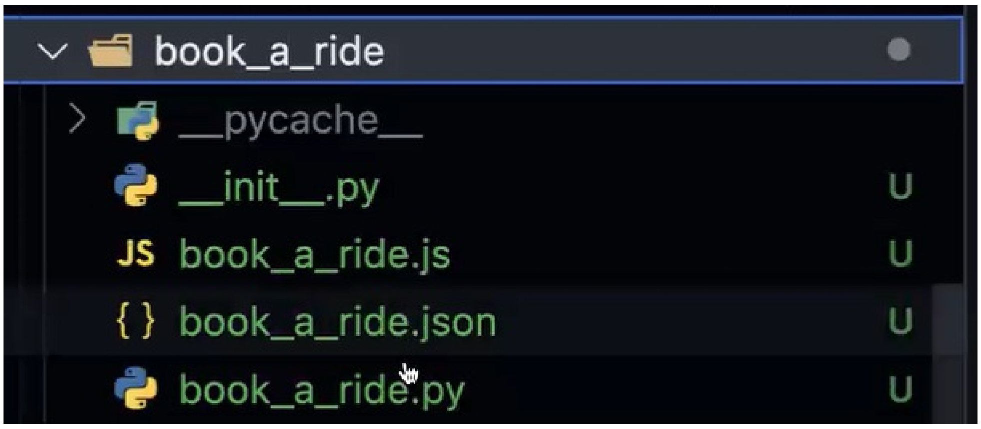
☑ Is Standard

If we want to have the web form on our custom app, we need to check the `Is Standard` field and provide the `Module`. This will generate files for the web form on our app.

Every time the web form is submitted, a new record of the doctype will be created with the data fetched from the web form.

Files Generated (WebForm):

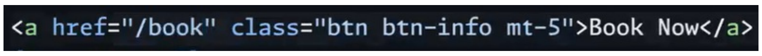

```
ride_management
  > __pycache__
  > doctype
  v web_form
      > __pycache__
      > book_a_ride
```


```
v book_a_ride
    > __pycache__
      __init__.py          U
      book_a_ride.js       U
      book_a_ride.json     U
      book_a_ride.py       U
```

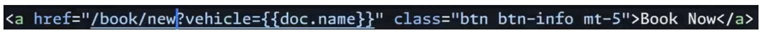
The files generated are in `web_form` directory at the same level of `doctype`. Inside, we'll find folders for each of our standard web form named after the WebForm name. Inside it, we'll find auto-generated .js, .json, and .py

Linking WebPage & WebForm:

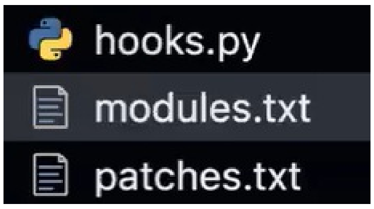
Now, since we have the WebForm created, we can have our WebViews button to redirect the user to the WebForm:


```html
<a href="/book" class="btn btn-info mt-5">Book Now</a>
```

We can also set a default parameter for fields (auto-fill) inside WebForm:

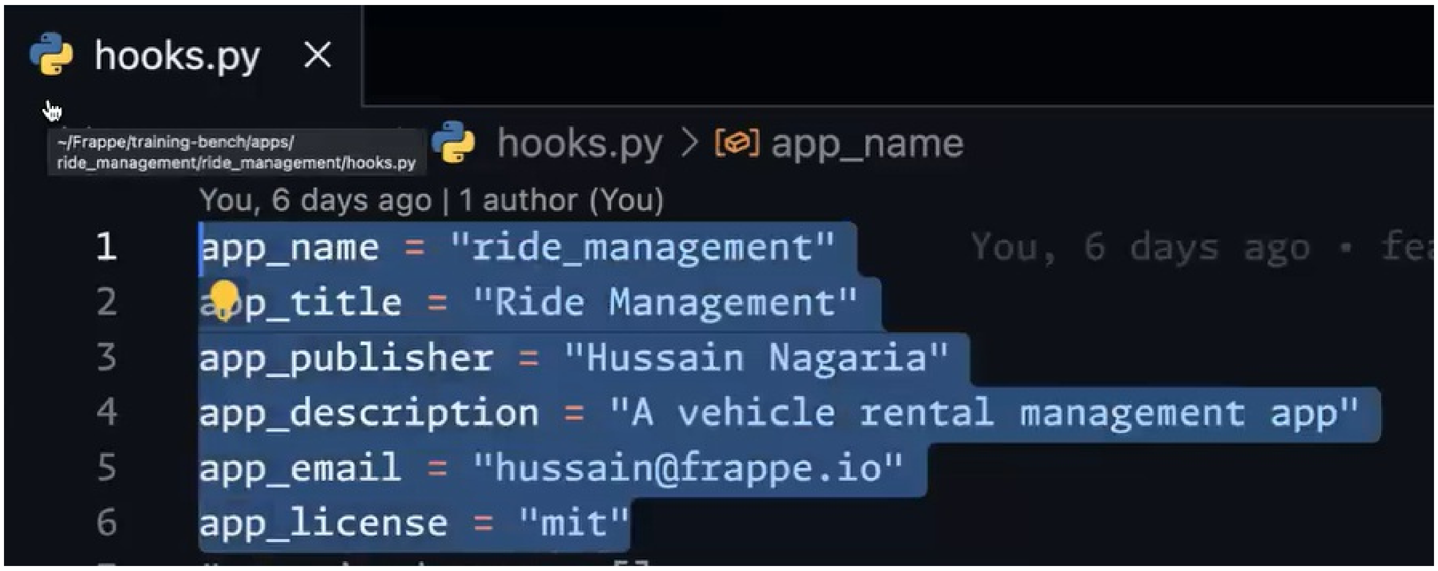

```html
<a href="/book/new?vehicle={{doc.name}}" class="btn btn-info mt-5">Book Now</a>
```

Hooks:



`modules.txt` : contains developer (standard) modules we created

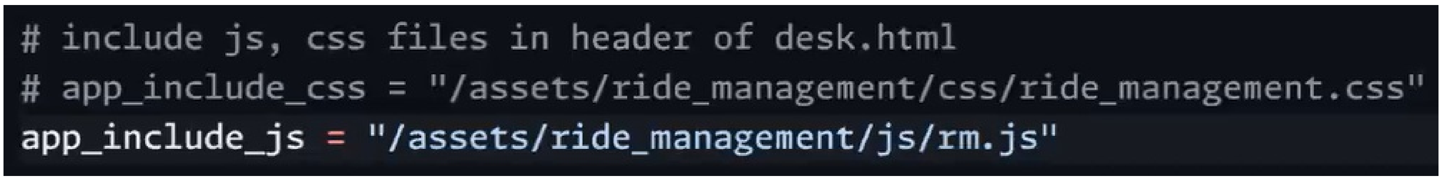
`patches.txt` : contains patches


hooks.py
~/Frappe/training-bench/apps/ride_management/ride_management/hooks.py
hooks.py > app_name
You, 6 days ago | 1 author (You)
You, 6 days ago • fe

```python
app_name = "ride_management"
app_title = "Ride Management"
app_publisher = "Hussain Nagaria"
app_description = "A vehicle rental management app"
app_email = "hussain@frappe.io"
app_license = "mit"
```

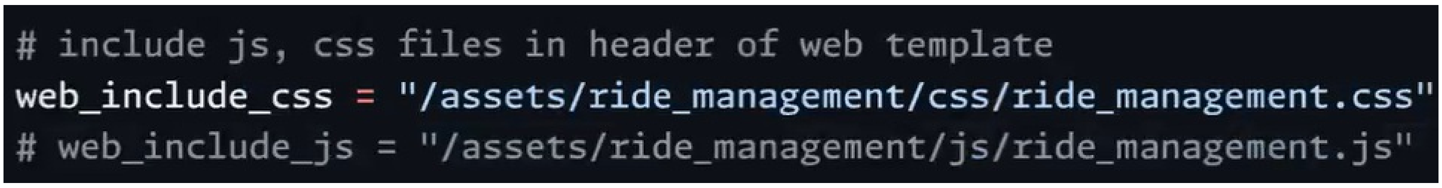
First information on `hooks.py` will be the app information we filled when we create a new app. Everything else will be commented (documentation for what we can do with hooks).

For example, app_include_js means when app (desk) is loaded, we will also run the rm.js located in the public js directory of ride_management app:



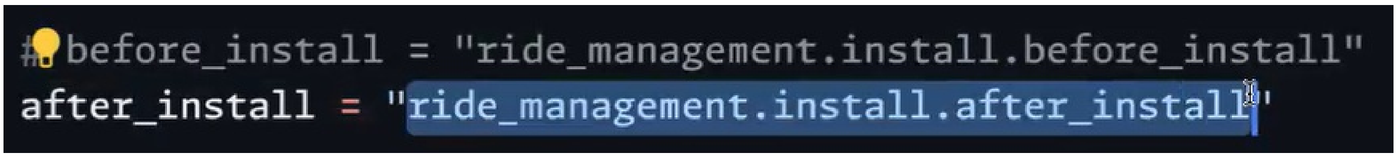
```python
# include js, css files in header of desk.html
# app_include_css = "/assets/ride_management/css/ride_management.css"
app_include_js = "/assets/ride_management/js/rm.js"
```

Another example, these will include the css & js whenever we extended web.html from frappe templates:

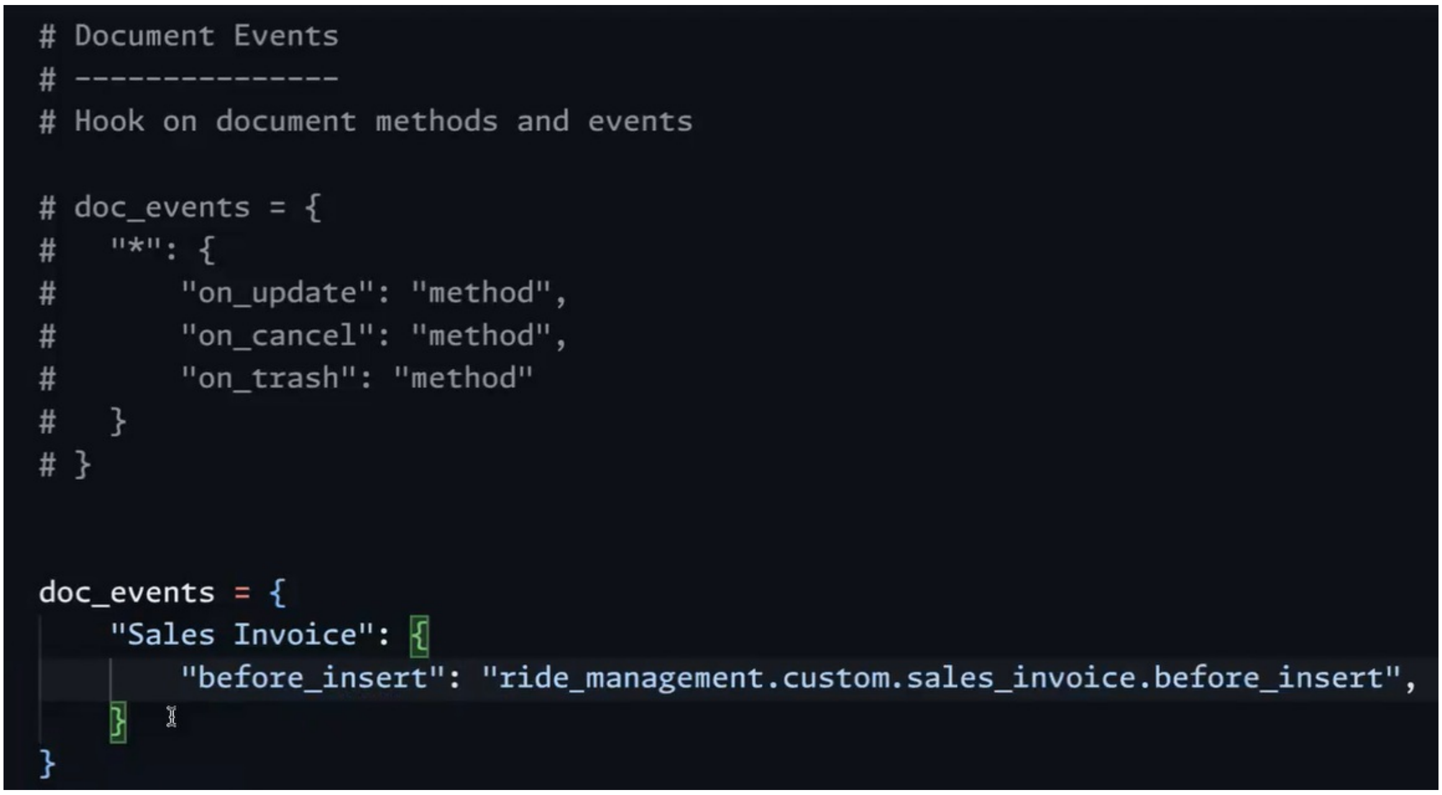


```python
# include js, css files in header of web template
web_include_css = "/assets/ride_management/css/ride_management.css"
# web_include_js = "/assets/ride_management/js/ride_management.js"
```

We can also hook into some specific processes. For example, we can inject code/script after the app is installed using `after_install` hook:

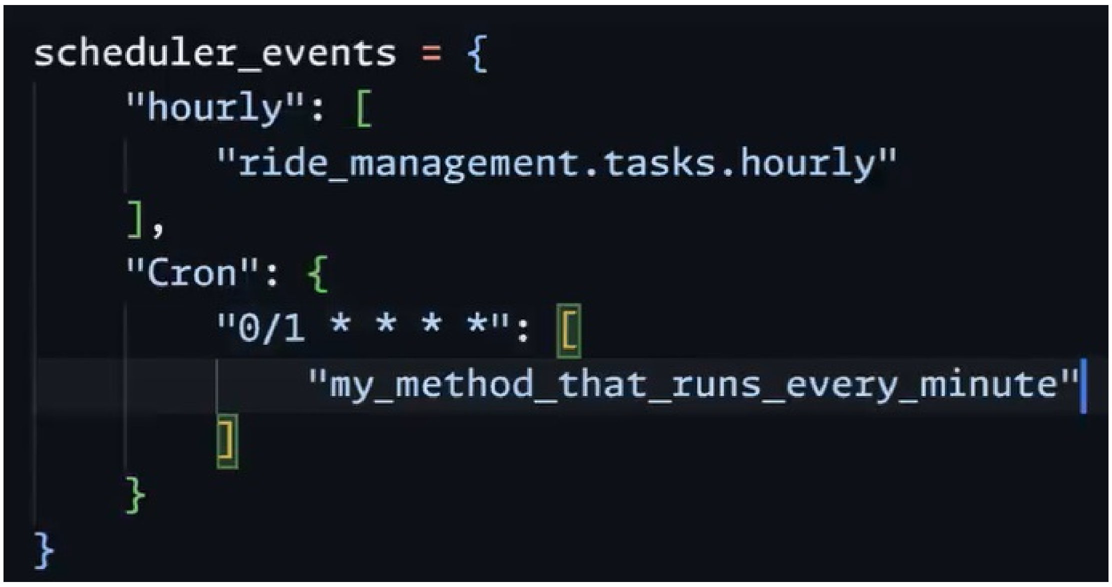


```python
# before_install = "ride_management.install.before_install"
after_install = "ride_management.install.after_install"
```



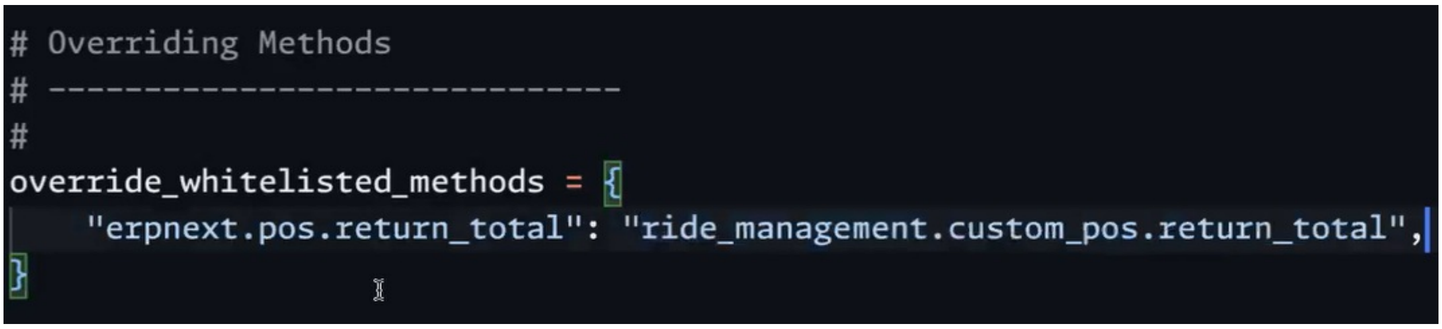
```
# Document Events
# ---------------
# Hook on document methods and events

# doc_events = {
#   "*": {
#       "on_update": "method",
#       "on_cancel": "method",
#       "on_trash": "method"
#   }
# }


doc_events = {
    "Sales Invoice": {
        "before_insert": "ride_management.custom.sales_invoice.before_insert",
    }
}
```

We also have doc_events where we can hook into any doctype methods & events.



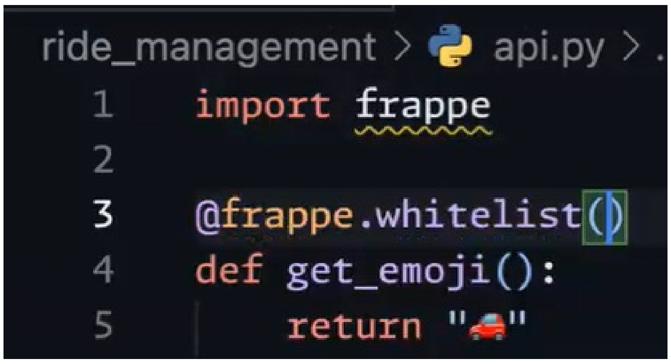
```
scheduler_events = {
    "hourly": [
        "ride_management.tasks.hourly"
    ],
    "Cron": {
        "0/1 * * * *": [
            "my_method_that_runs_every_minute"
        ]
    }
}
```

`scheduler_events` is useful to tell frappe that we want to run specific script/code/method in a specific time interval.



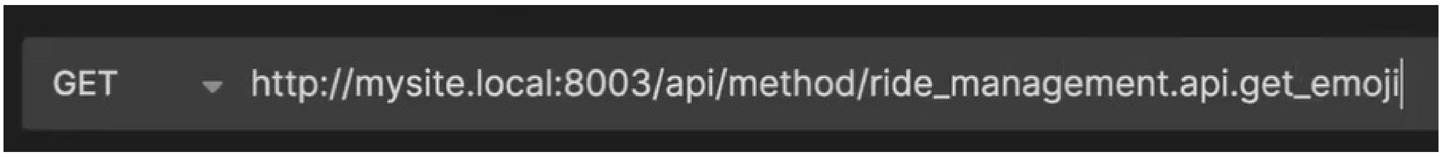
```
# Overriding Methods
# ------------------------------
#
override_whitelisted_methods = {
    "erpnext.pos.return_total": "ride_management.custom_pos.return_total",
}
```

We can also override whitelisted methods (we can't override non-whitelisted ones). So, methods that are decorated with `frappe.whitelist()` are accessible to be called through API. With this, it's possible to create our own custom API.



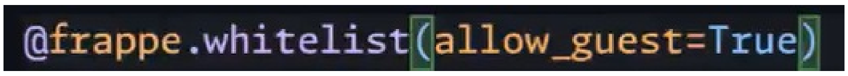
```
ride_management > api.py > .
import frappe

@frappe.whitelist()
def get_emoji():
    return "🚗"
```

Start with creating `api.py` file inside our app directory, define the method, then call it using Bruno (/api/method/[dotted-path]):



```
GET    http://mysite.local:8003/api/method/ride_management.api.get_emoji
```

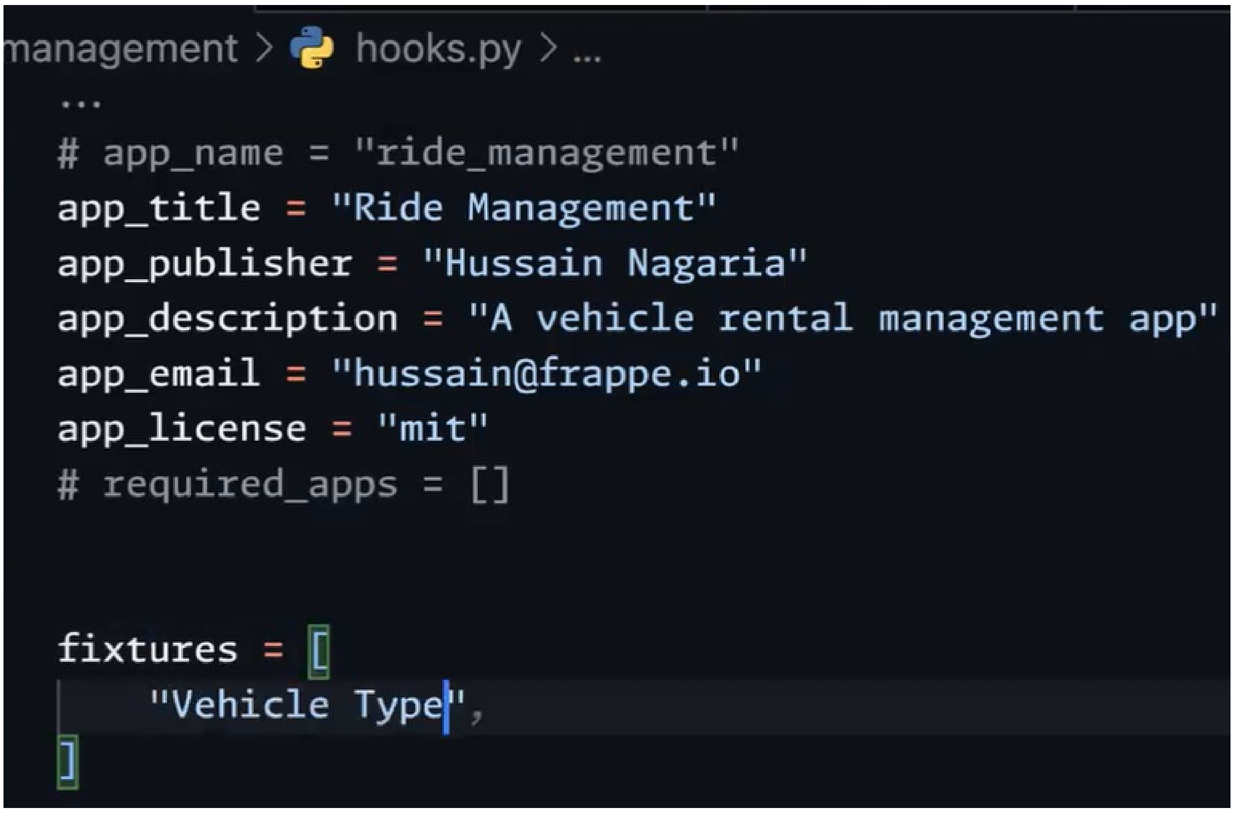
The method above will require the user to provide Authorization header to access the method through API. To allow non-authorized user, add `allow_guest=True`:



```
@frappe.whitelist(allow_guest=True)
```

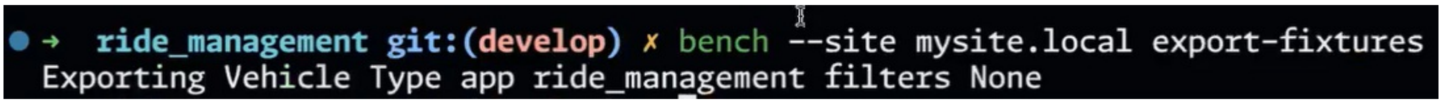
Fixtures:

Fixtures lets us export the documents/records, not just the doctype schema. To do this, we have to include `fixtures` on hooks.py which holds a list of doctypes that we want to export.



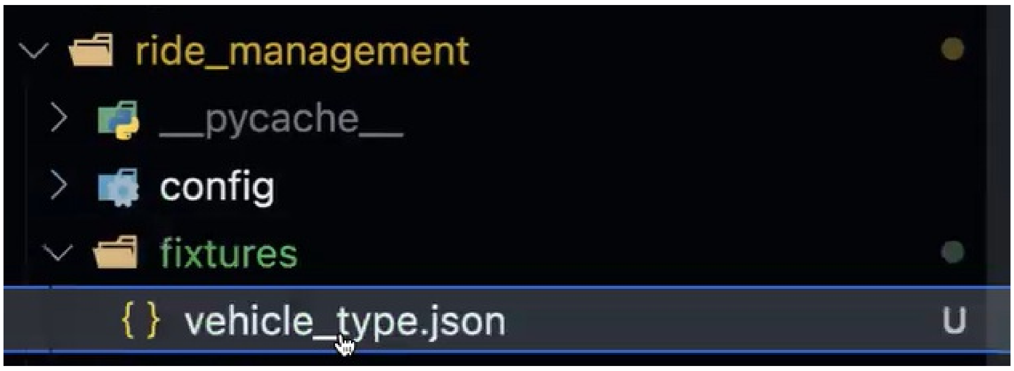
```
management > hooks.py > ...
...
# app_name = "ride_management"
app_title = "Ride Management"
app_publisher = "Hussain Nagaria"
app_description = "A vehicle rental management app"
app_email = "hussain@frappe.io"
app_license = "mit"
# required_apps = []


fixtures = [
    "Vehicle Type",
]
```

To actually export it, we have to call `export-fixtures` bench command:

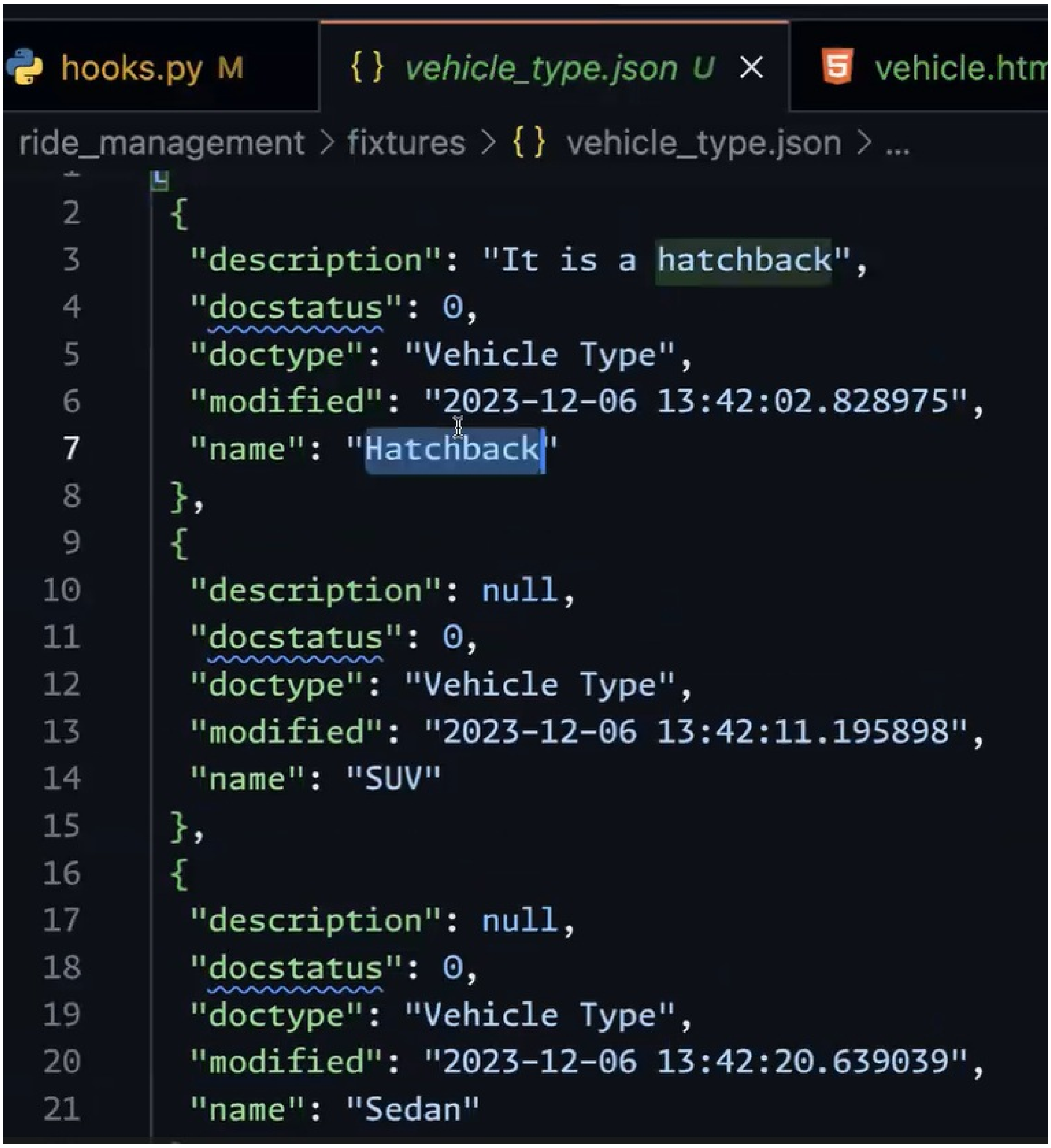


```
● → ride_management git:(develop) ✗ bench --site mysite.local export-fixtures
 Exporting Vehicle Type app ride_management filters None
```

After running the above command, there will be a `fixtures` directory auto-generated with the .json of the exported doctypes inside:



The .json contains the records of that doctype that exists on the site:



```
 {
  "description": "It is a hatchback",
  "docstatus": 0,
  "doctype": "Vehicle Type",
  "modified": "2023-12-06 13:42:02.828975",
  "name": "Hatchback"
 },
 {
  "description": null,
  "docstatus": 0,
  "doctype": "Vehicle Type",
  "modified": "2023-12-06 13:42:11.195898",
  "name": "SUV"
 },
 {
  "description": null,
  "docstatus": 0,
  "doctype": "Vehicle Type",
  "modified": "2023-12-06 13:42:20.639039",
  "name": "Sedan"
```

If there are changes that we want to push (e.g. add a new vehicle type), just run `export-fixtures` again, and run `migrate` to have the changed .json to take effect.

How does Bench Migrate work?

Frappe has a hash-based migration, what it does is it looks at the .json file, it hashes it, then it checks the previous hash it has stored, and matches them. If there is a change, then it will make the **necessary changes**. This is the same case for doctypes' .json files. The changes are made automatically by Frappe so we don't need to create migration files.