1.1 - Routing and Views

During skills lab:

Do the following assignments during the skills lab and bring the results to the Checkpoint.

1.1.A - Project setup

👤 This is an individual assignment.

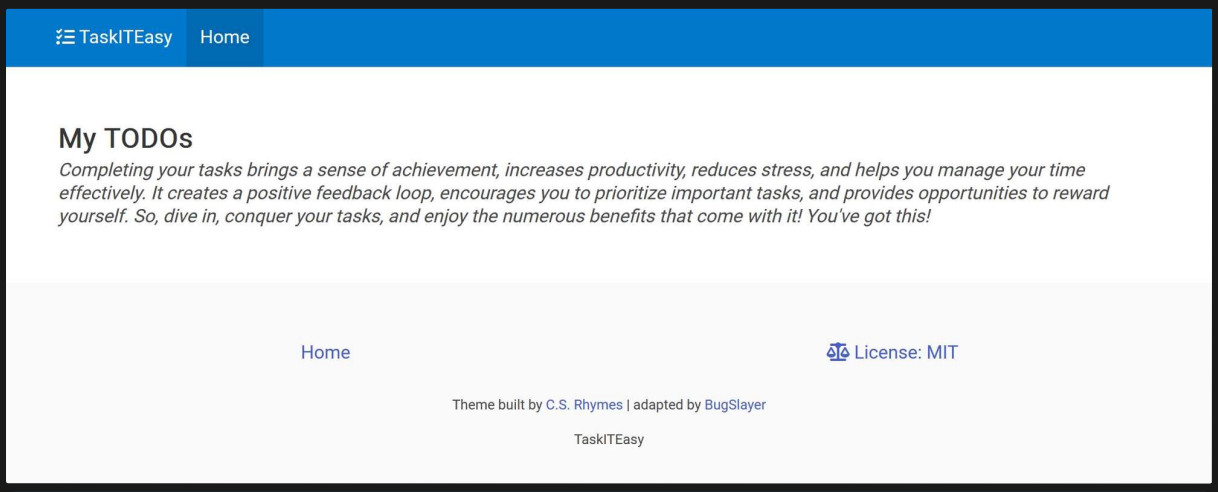
In order to get started, you need to set up the project first. To streamline this process, we have made a zip file available containing a Laravel project with a few modifications already made. This was also discussed in class. You can find the zip file below:

laravel-taskiteasy-starter-code.zip 748.2KB

🛠️ Download the zip file, and install this as an existing Laravel app. See for instructions:

✨ Setting up Laravel Projects

When opening the app in your browser, you should see something like this:



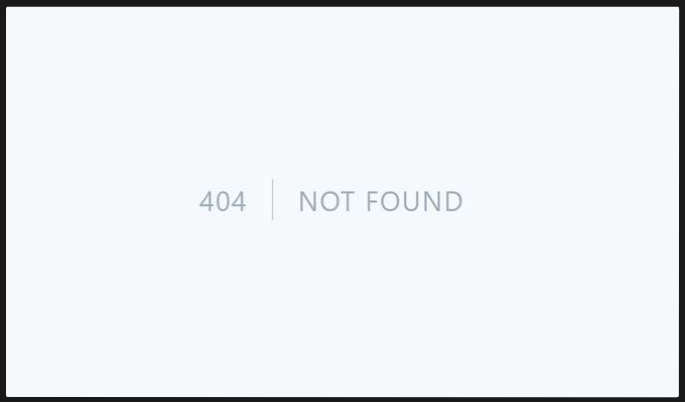
🛠️ Find and open the view that is used by this route. The view contains 109 lines of code, but only 8 lines really represents the page's content. The rest is header, navigation, footer, etc.. Find out which 8 lines contain the actual page content. You will need this in later assignments.

1.1.B - About page

👥 Work in *programming pairs*: a team of two members sitting behind one computer programming as one.

In this section you will create a new route and view. In the following assignments you will discover some exciting stuff around this process, like some error messages

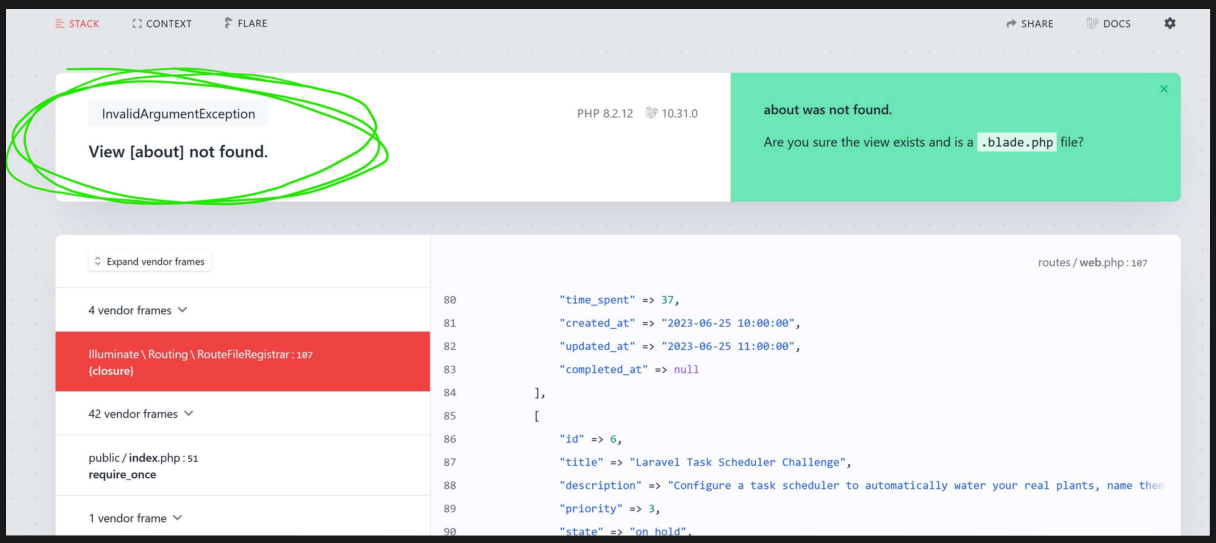
🛠️ Try to open the url: http://localhost/about. The app should respond with a 404 page like:



Obviously, the route does not exist. So, let's just create one. Adding a route in Laravel means that you need to declare it in your project's `web.php`. During class we already discussed the concept of routes. You can read about the details here:

Routing

Create the route that should handle this request in your `web.php` file. Make it return a view called 'about' (do not create that view yet). The app should now return an error message like:



We highlighted the most important part of the error message here. It states that the 'about' view does was not found. Of course this means that this view simply does not exist yet. During class we already introduced the concept of views, but you can read all about the details in the following knowledge bank article:

Views

So, let's create it by using Artisan:

Use the `php artisan make:view` command to create the **about** view. Check if the folder `resources/views` contains a file called `about.blade.php`. Open this file and add the following content to it:

```
OMG it works!
```

Your browser should now show this text upon refresh. Now let's fill it with proper structure and content

🛠️ Build the view by:

1. Copy and paste the code from `welcome.blade.php` to get the HTML *structure*.
2. Remove all the actual content that makes up the welcome page and replace this with the code below.

Note: you decide here on what the difference is between structure and actual content. You should see a internally consistent about page.

```
<h1 class="title is-4">About</h1> <h2 class="title is-5">Welcome to Our App</h2> <p> Discover our innovative app designed to transform your daily life and streamline your productivity. Our app is your perfect companion for task management, allowing you to regain control of your time and focus on what truly matters. Whether you're a busy professional, a student juggling classes and extracurricular activities, or simply looking to better organize your personal life, our app is here to help. With an intuitive and user-friendly interface, we've created a solution that empowers you to achieve your goals and dreams. </p> <h2 class="title is-5 mt-5">Key Features</h2> <p> Our app boasts a wide range of features that make it an indispensable tool for anyone seeking enhanced productivity. You can create and manage tasks with ease, set deadlines, and prioritize them according to your needs. We've also incorporated a reminder system that ensures you never miss an important deadline or appointment. In addition, our app allows for collaboration with others, making it ideal for team projects and group activities. With cloud synchronization, you can access your tasks from anywhere, keeping you on top of your game at all times. </p> <h2 class="title is-5 mt-5">Our Vision</h2> <p> At our core, we are driven by a vision of simplifying your life and enabling you to achieve your aspirations. We believe that effective task management can free up valuable time for personal growth, creativity, and enjoying life's pleasures. Our app is not just a tool but a partner in your journey to success. Join us as we revolutionize the way you work, study, and live. Together, we can make your dreams a reality, one task at a time. </p>
```

1.1.C - Showing tasks

In this section, you will experiment with passing data to views. We do this by creating a page that shows the details of a given task. During the following assignments you will do this and also learn how to work with the `dd()` (dump and die) statement.

🛠️ Create a route handler for the following URI: `tasks/{id}`. Just copy and paste the code fragment below in your `web.php`. Test it in your browser with different URLs: `http://localhost/tasks/1`, `http://localhost/tasks/3123` and `http://localhost/tasks/persist-with-purpose`. Observe the result. Try to come up with a conclusion between the URL and the content on the page.

```
Route::get('tasks/{id}', function ($id) { dd($id); });
```

You now know how users can pass data to the application via the URL, you can use this to find some data in our array that corresponds to the given `$id`

🛠️ Create a function in your `web.php` file. The function must accept one parameter (`$id`) and returns an associative array with Task data that corresponds to the given `$id` value. Name this function `findTask`. When the specified task is not found, it should return `null`. Test this function by calling this function in the `dd()` statement, like:

```
// finds the specified task and dumps it dd(findTask($id));
```

You should now see something like:

```
array:10 [▼ // routes/web.php:132
  "id" => 1
  "title" => "Write a love letter to Laravel"
  "description" => "Compose a heartfelt love letter to Laravel, explaining how it has revolutionized your c
  "priority" => 3
  "state" => "new"
  "time_estimated" => 15
  "time_spent" => 0
  "created_at" => "2023-10-15 09:00:00"
  "updated_at" => "2023-10-15 09:30:00"
  "completed_at" => null
]
```

The next step is to pass this array to a view so we can render the content of a task consistently with the rest of the app.

🛠️ Create a view that can be returned with a command like: `return view('tasks.show')`. Figure out what the correct Artisan command must be to achieve this. Test this by replacing the `dd()` statement with a normal return statement that returns the view.

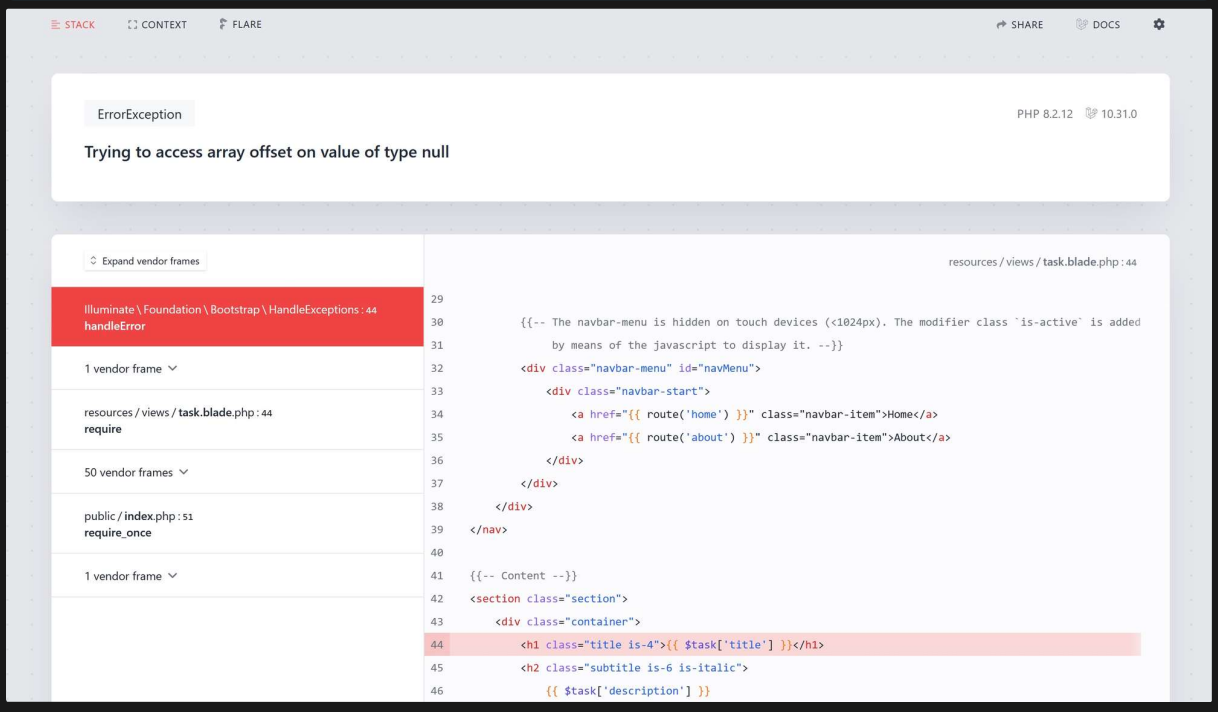
Now we can pass the specific task data to that view

🛠️ Pass the result of the `findTask()` function to the view using something like the statement below. Test this by displaying the title in the view like: `{{ $task['title'] }}`

```
return view('tasks.show', [ 'task' => findTask($id) ]);
```

Now we can render Task data, we can try to break it

🛠️ Find out what happens when the user passes a non-existing id, like: `http://localhost/tasks/3123`. You should see an error message like:



Note that the page shows the error message on top ("Trying to access array offset on value of type null") and highlights one line of code (line 44). This line only has one dynamic element (`$task['title']`). The error message implies that the server tries to do something ('access an array offset') on a 'value' of type null. While there is only one 'value' (the `$task` variable) we can conclude that the value of that variable is `null` . This is logical, because the `findTask()` function was not able to find a task with id 3123.

In order to solve this, you need to use the `abort()` function. This function triggers the creation of an HTTP response representing an error situation, which will be caught and processed by the Laravel built in exception handler, allowing for customised rendering of the response corresponding to the specific HTTP status code.

🛠️ Update your `findTask()` function so it renders a 404 (not found) page when a non-existing id is passed.

1.1.D - Dynamic content and styling

Discover the realm of dynamic styling in this segment. Engage with Blade's empowering features that grant you full control over the rendered elements in the HTML response. Explore how you can manipulate and tailor the appearance through dynamic styling, uncovering the flexibility within Blade's capabilities. Together with this, you will have some fun with icons.

🛠️ First, make sure the HTML structure of the `tasks.show` page matches the welcome and about pages (copy and paste, and replace the actual content)

As you know, a task has lots op properties. In the next exercises you are going to render these properties in different ways.

🛠️ Add the `title` in a `<h3>` element and the `description` in a `<p>` element. The description must be rendered unescaped because this contains HTML code! Test this with different tasks. It should show the marked up description (bold, italics, etc.)

Next, we focus on the priority. The priority is a number between 1 and 4. Pending on the value, the page must show a textual representation:

1. *No priority (shows nothing)*
2. Moderate

3. Urgent
4. Critical!

You can use common PHP to achieve this, but Blade provides convenient shortcuts - called *directives* - for common PHP control structures, such as conditional statements and loops. These shortcuts provide a very clean, terse way of working with PHP control structures while also remaining familiar to their PHP counterparts.

To achieve the requirement above, you can use a conditional directive (`@if` , `@elseif` , `@else` , and `@endif` or `@switch` , `@case` , `@break` , `@default` and `@endswitch`)

🛠️ Use conditional directives to render the tasks priority textual representation. You may spice it up by wrapping the text in a Bulma Tag: https://bulma.io/documentation/elements/tag/. You can then add proper colouring.

Following this, let's direct our attention to the attributes regarding the estimated and spent time. We can show these best near to each other, like: "6/11". Then we can highlight when the task goes over budget with a red font colour: "37/20". Using Bulma you can achieve this by adding the classes `has-text-weight-bold` and `has-text-danger` , like:

```
<span class="tag has-text-weight-bold has-text-danger"> 37/20 </span>
```

This means that you need to render classes pending on the state. Again, this can be done with the `@if` directive, but it might be a bit more readable by using the ternary operator within curly braces, like:

```
{{ $age < 18 ? 'You are too young' : 'Ok, carry on' }}
```

🛠️ Use the method of your choice to render the extra classes to the class attribute when the `time_spent` value is larger than `time_estimated` .

Then try an icon. Icons are popular in use because they offer numerous advantages over plain text: they convey information quickly and universally, transcending language barriers while saving space. Their visual nature aids memory and recognition, enhancing user experience by making interfaces more engaging and aesthetically appealing. When appropriately chosen and labelled, icons also contribute to accessibility, benefiting users with literacy challenges or language barriers. Moreover, icons can strengthen brand recognition, becoming an integral part of a brand's identity and fostering emotional connections with users.

Icons can be used by importing a icon CSS library like Font Awesome. On the other hand, you can also use the SVG format and create your own or download from the internet. We have found a nice icon that can symbolise a complete task:

```
<svg xmlns="http://www.w3.org/2000/svg" width="24" height="24" viewBox="0 0 24 24"> <path fill="currentColor" opacity=".3" d="M5 19h14V5H5v14zm2.41-7.412.58 2.58l6.59-6.59L17.99 9l-8 8L6 13.01l1.41-1.41z"/> <path fill="currentColor" d="M19 3H5c-1.1 0-2 .9-2 2v14c0 1.1.9 2 2 2h14c1.1 0 2-.9 2-2V5c0-1.1-.9-2-2-2zm0 16H5V5h14v14zM17.99 9l-1.41-1.42l-6.59 6.59l-2.58-2.57l-1.42 1.41l4 3.99z"/> </svg>
```

🛠️ Use an `@if` directive to render the SVG-icon above when the `completed_at` attribute is not `null`. Play around with it by i.e. wrapping it in a Bulma tag and different foreground and background colors. Have fun!

Checkpoint

Before visiting the checkpoint, please ensure that you bring along the following items.:

- Code of the about page
- Your code that shows:
  - Location and name of the `tasks.show` view
  - How `findTask()` renders the 404
- Code of the `tasks.show` page

After the Checkpoint

👤 This is an individual assignment.

You can experiment further with your own project.

1.1.E - Catching up: Implementing previously missed features

In this part, you have the flexibility to either implement any missing features from the skills lab assignments in your project or build upon the provided snapshot, which already includes all the features.

🛠️ Make your project ready for the following assignments. Check if all the features are present and bug free. If not, choose to either finish your project or use the provided snapshot by copying the required view files and the content of `web.php` to your project

laravel-taskiteasy-snapshot-1-1-1.zip 752.1KB

1.1.F - Complete the `tasks.show` view

In the previous section, 1.1.D, you began displaying different details about the tasks. Now, we'll continue by showing the rest of the task's information. First up is the `state` attribute. Instead of using `completed_at` to only tell if a task is finished or not, we'll change this to give more specific information about the task's current status.

🛠️ Render the task state attribute. Wrap it in a Bulma tag and use colors to distinguish between different states. Choose your own colors that you think will match the state best (i.e. `success` when task is complete).

The remaining attributes are the three timestamps (`created_at` , `updated_at` and `completed_at`). Laravel uses the Carbon PHP library to work with dates and times. This means that timestamps can be parsed to Carbon objects, that have nice formatting methods, like the following example from the Carbon documentation website:

```
$timestamp = "2023-07-10 17:45:00" $dt =
\Carbon\Carbon::parse($timestamp); var_dump($dt->toDateTimeString() ==
$dt); // bool(true) => uses __toString() echo $dt->toDateString(); //
1975-12-25 echo $dt->toFormattedDateString(); // Dec 25, 1975 echo $dt-
>toFormattedDayDateString(); // Thu, Dec 25, 1975 echo $dt-
>toTimeString(); // 14:15:16 echo $dt->toDateTimeString(); // 1975-12-25
14:15:16 echo $dt->toDayDateTimeString(); // Thu, Dec 25, 1975 2:15 PM //
... of course format() is still available echo $dt->format('l jS \\of F Y
h:i:s A'); // Thursday 25th of December 1975 02:15:16 PM // The reverse
hasFormat method allows you to test if a string looks like a given format
var_dump(Carbon::hasFormat('Thursday 25th December 1975 02:15:16 PM', 'l
jS F Y h:i:s A')); // bool(true)
```

It even provides nice *difference for humans* formats:

```
// The most typical usage is for comments // The instance is the date the
comment was created and its being compared to default now() echo
Carbon::now()->subDays(5)->diffForHumans(); // 5 days ago echo
Carbon::now()->diffForHumans(Carbon::now()->subYear()); // 11 months after
$dt = Carbon::createFromDate(2011, 8, 1); echo $dt->diffForHumans($dt-
>copy()->addMonth()); // 1 month before echo $dt->diffForHumans($dt-
>copy()->subMonth()); // 1 month after echo Carbon::now()->addSeconds(5)-
>diffForHumans(); // 4 seconds from now echo Carbon::now()->subDays(24)-
>diffForHumans(); // 3 weeks ago echo Carbon::now()->subDays(24)-
>longAbsoluteDiffForHumans(); // 3 weeks echo Carbon::parse('2019-08-03')-
>diffForHumans('2019-08-13'); // 1 week before echo Carbon::parse('2000-
01-01 00:50:32')->diffForHumans('@946684800'); // 50 minutes after echo
Carbon::create(2018, 2, 26, 4, 29, 43)-
>longRelativeDiffForHumans(Carbon::create(2016, 6, 21, 0, 0, 0), 6); // 1
year 8 months 5 days 4 hours 29 minutes 43 seconds after
```

🛠️ Create Bulma tags for the 3 timestamp attributes. Use ‘tag addons’ for the name of each attribute and experiment with different timestamp formats. Check the Bulma documentation in order to find out what a tag addon is

1.1.G - a List of Tasks

In this section, we delve a little deeper into the world of Blade directives, specifically focusing on the `@foreach` directive. This directive serves as a robust tool for iterating through collections, making it essential in rendering dynamic content within your web applications.

🛠️ Apply the `@foreach` Blade directive within the `home` view to render the list of tasks stored in your Laravel project's `web.php`. Utilize this directive to loop through the tasks collection, presenting at least 4 attributes —such as titles, descriptions, states, or other associated attributes—in a structured format on the webpage. Ensure a clear and organized representation of the task list, utilizing proper HTML structure with Blade syntax for effective integration. This hands-on task aims to reinforce your grasp of the `@foreach` directive's role in dynamically showcasing task lists, a crucial skill for creating interactive and data-driven web interfaces.

It is common practice that pages that shows lists of items to provide navigation links to pages that show details about each item. So, when the user clicks on, for instance, the name of the item it navigates to the details page showing the item's details.

Generating URLs for items is also common in Laravel and the framework provides several ways to help. The most basic one is the `url()` helper function, for example:

```
$post = [ 'id' => 1, 'title'=> 'Lorem ipsum...', // ... ]; echo
url("/posts/$post['id']"); // http://example.com/posts/1
```

You see that the generated URL will automatically use the scheme (HTTP or HTTPS) and host from the current request being handled by the application. You can read all about generating URLs here:

🔗 URL generation

🛠️ Turn the task title into a link (`<a>`) and use the `url()` function to generate a URL to the `tasks.show` page that shows the task details.

1.1.H - Spicing up

Following are some suggestions how to spice up your app. Consider these as extra challenges. Choose for yourself how many and which of these you MAY want to implement.

Consider named routes

The URL Generation knowledge bank article describes the concept of named routes. This might be helpful for you in the future. You should consider using them from now on.

Internally consistent

Your tasks index and show page are probably not internally consistent with the welcome and about pages. You should fix this by, again, copying the HTML structure of the welcome (or about) page and replace its content with the actual content

Spicing up with icons

You might want to spice up your `tasks.show` and `tasks.index` views with more icons. For instance the state and priority. You can find a good icon repository here:

> Icon Sets • Iconify
>
> Click icon set to see available icons or search icons by keyword.
>
> https://icon-sets.iconify.design/

Browse, select, copy the SVG and play around with it.

Blade Components

Blade components are a relatively new feature that allows the developer to create reusable, dynamic HTML structures. The following knowledge bank article provides an introduction:

Blade components

You SHOULD experiment with these components. For instance by creating a Tag component to render task attributes, or even MAY have a component that renders