2.2 - CRUD → ‘C’reate

During skills lab:

- 2.2.A - Reverse engineering
- 2.2.B - Negative testing
- 2.2.C - Design the Task-create feature

Checkpoint

After the Checkpoint

- 2.2.D - Catching up: Implementing previously missed features
- 2.2.E - Create Post: server side validation
- 2.2.F - Optimizing the code
- 2.2.G - Create Task
- 2.2.H - Spicing up

During skills lab:

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

⚠️ In this workshop you need to use the following snapshot. It already includes all the features of yesterdays assignments and the code for the following Reverse Engineering assignment:

laravel-taskiteasy-snapshot-2-2-0.zip 760.7KB

2.2.A - Reverse engineering

👥 Work in groups of 3-5 students.

In this assignment you will explore a typical create feature implemented in Laravel. You will analyze the existing code and discover its inner workings. a Process called *reverse engineering*.

🛠️ Form a group of 3-5 students. At least one member downloads the Laravel project and sets it up on his working environment so the code can be analyzed and the working app can be tested.

First you explore the functional perspective of the feature.

🛠️ Reverse engineer the functional perspective of the post-create feature, by testing the working version of the app. Sketch a *wireflow* that describes the flow of pages, links and redirects. This wireflow must show each view, each Form-input of the form, and flow(s) for the happy path and should show the alternative flows: cancel and reset.

🎨 **Wireflow diagrams**

Next you find out the technical perspective. This means that you figure out which HTTP requests are sent at what moment during the entire scenario.

🛠️ Reverse engineer the technical perspective of the post-create feature, by analyzing the code and/or analyzing the network traffic. Document at least the following:

- URLs, route handlers, views and route names, i.e. by creating a route list table.
- The routes that are used when navigating links and submitting forms.
- Document each HTTP request as complete as possible but at least the HTTP method, URL and payload (input data)
- Document also the response. Mention response code and relevant response data, like redirect locations. Omit the body, for it might be large

2.2.B - Negative testing

When dealing with forms, it's key to acknowledge that user actions often contain errors. This is why conducting negative testing becomes crucial. Regrettably, in the example app, we intentionally overlooked this aspect. Take some time to delve into this and consider ways to enhance it.

🛠️ Check what happens when the user submits an empty form and a form that contains erroneous input

Again, we deliberately built this wrong. So you should see something a normal user should not see. Now, explore how the real world handles this

🛠️ Brainstorm on what you actually want. Compare with similar features on other websites in the real world that you know of. Sketch a wireflow that illustrates this

Welcome to the world of form validation...

2.2.C - Design the Task-create feature

👥 Work in groups of 3-5 students.

Now you know how the post create feature works from functional and a bit about the technical perspective. a large part of this can be transferred to create features for other models. You will discover this in the following exercises.

🛠️ Design a wireflow for a Create Task feature. Discuss the similarities and changes between this feature and post-create. Also with respect to *consistency and standards*. Consider form design, happy and alternative flows and validation.

Next, you think about the business perspective. When users create a task - mostly in a situation where they are at the beginning of the job, and want to plan or directly start working on it - which fields must be filled in at least, are there any fields optional, have some default value or should even not be there at all?

🛠️ Design the form. Decide on each form input (or actually, each attribute):

- Is it required, optional or not present at all?
- Does it have a default value you can already provide?
- Label text

After and/or together with this, you should also think about the technical perspective

🛠️ Design the technical perspective for the Create Task feature. Consider at least:

- HTTP requests that are sent when navigating links and submitting forms.
- Record for each request the HTTP method, url and data
- Record also the response. Mention response code and relevant response data, like redirect locations
- URLs, route handlers, views and route names

When designing, you can use and enhance a route list table for this. But you may document this in any way you want

Checkpoint

Visit the checkpoint. Bring along the following:

- Functional and technical perspectives of the Create Post feature (reverse engineered)
- Real world observations of 'form error handling'
- Functional and technical perspectives of the Create Task feature, with your observations about the similarities and changes of both features

After the Checkpoint

👤 This is an individual assignment.

You can experiment further with your own project.

2.2.D - Catching up: Implementing previously missed features

In this part, you have the flexibility to either implement any missing features from yesterdays skill assignments in your project or build upon the provided snapshot at the top of this page, which already includes all the features of yesterdays assignments.

> 🛠️ 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.

When you want to work with your own implementation:

> 🛠️ Implement the Create Post feature in your own *TaskITEasy* app (you may copy and paste from the skills lab). Test to see if it works, including the error when submitting an empty form.

2.2.E - Create Post: server side validation

The current solution doesn't implement proper server side validation, which of course it should. See also the following knowledge bank article:

👨‍🎨 Form request validation

> 🛠️ Add server side validation code in the controller method. Each input (title, excerpt and body) should be validated with at least the `required` option. Test to see if submitting an empty form does not result in the error message anymore.

Next you need to provide the user with proper feedback when validation fails. As mentioned in the checkpoint, you have options (from the User Input UI Pattern):

1. The basic "An error has occurred"; simple, but not very helpful for users
2. a More elaborative "Where the error occurred"; where you can highlight erroneous input and even a message stating what the user did wrong;

To implement this in Laravel, you can use the `$errors` variable and/or the `@error` directive.

Besides this, it is also considered a good practice to repopulate the form that the user attempted to submit. In Laravel you can use the `old()` helper function for this.

🛠️ Explore, decide and implement proper user feedback validation errors and repopulating the form again

2.2.F - Optimizing the code

Finally, you can consider refactoring the code making it a bit more readable by:

- Inject the `$request` parameter in the create route handler method (see **Routing - Dependency injection**) and use this instead of the `request()` helper function
- Implementing *mass assignment* to populate the new model object with the validated input in one statement (see: **Eloquent ORM - Mass assignment**)

🛠️ Consider both refactoring options. Experiment with them and experience how beautiful and easy it is to write these kinds of methods and how easy it is to reuse this in your next assignment

2.2.G - Create Task

This assignment is simple:

🛠️ Re-apply everything you've seen and learned to implement the Create Task feature based on your design in 2.2.C (and feedback from the checkpoint) and what you did in 2.2.E. Include (just like before):

- Internal consistency for all the views, forms, etc.
- Input validation, feedback and repopulating

💡 It is common to apply some *smart-copy-and-paste* programming here. This means that you can copy large parts of the Post create form and routes and controller code. The smart thing about this that you need to figure out which can be kept and what needs to change. **Important tip**: Do not forget to keep the `@csrf` directive in the form. Tomorrow you will discover why this is important.

2.2.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 want to implement.

Other validation rules

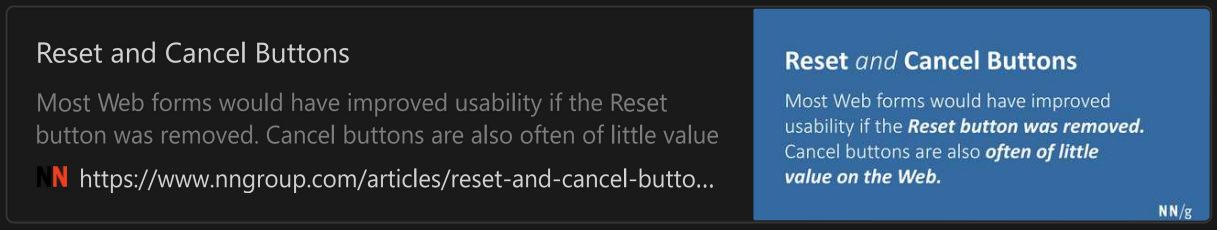
During the Checkpoint we briefly discussed that there are other validation rules possible and Laravel provides a simple way to use these rules. This is especially useful here while Tasks require specific number and date inputs. See the following knowledge bank article:

👨‍🎨 Form request validation

Experiment with the other validation rules and create more specific validations for the number inputs (`priority` , `state` and `time_estimated`)

Alternative flows: Cancel/Reset

Many UI's offer Cancel and even Reset buttons for their forms and dialogs. Consider the following article:



Experiment with Cancel and Reset buttons in your forms.

Other form inputs: the state attribute

Consider the `state` attribute in a Task. This attribute should contain one of the following fixed string values:

- new
- deferred
- cancelled
- planned
- in progress
- on hold
- completed

Therefore, users should not be able to type this value in a normal text input. You could use alternatives like a *select box* or *radio buttons*.

Explore, decide and implement a better user input for the `state` attribute in the Create Task feature. Together with this you can also figure out if there is a validation rule that can check if the input is one of the mentioned values.

Flashing success messages

The most complex part of the Input Feedback UI pattern (in 2.2.E you discovered the other 2) is "let user know that everything went as planned". To implement this you need to work with session flashing, which is described in the following knowledge bank article chapter:

👮‍♂️ **Form request validation - Flashing other feedback**

Implement session flash notifications to let the user know that the Task (and Post) was created successfully.

Client side validation

The Form Request Validation article also describes that it is possible to implement client (JavaScript) side validation.

Feel free to experiment with this as well, but remember that server-side validation is mandatory (for security reasons as well) for this course and client side might be added as a complementary feature to improve the user experience.

Other form inputs: WYSIWYG

Finally, we like to challenge you to implement the WYSIWYG (What You See Is What You Get) UI pattern if you have the time. There are multiple JavaScript libraries available that might help you with this.

Explore the WYSIWYG UI pattern, decide and implement a WYSIWYG editor for the larger text inputs:

- Task description
- Post body
- Post excerpt

Tip: you might try to build a reusable Blade component for this.