THE ISLE OF ANSUR

BETWEEN SHADOWS & LIGHT

Official modding guide

Legend

D — directory

F — file

Types of mods

⚓ Back to Table of Contents

Isle of Ansur currently has four types of mods (or "packs" which is more correct name):

- statpack
- themepack
- worldpack
- scripts

Every mod can constitute one or more of those pack types, as they all interact with different part of the game. Let me shortly introduce them.

I. Statpack

Statpack is type of pack that builds the backbone of whole game by implementing data related to statistics - a core feature of RPG game!
Statpacks let you add new races, classes, religions, origins, attributes, skills, but will also allow for adding new creatures in the future.

II. Themepack

Themepack is usually a pack structured around game's look - it introduces colour theming, images seen as a background for menus, as well as GUI elements.

III. Worldpack

Worldpack is type of pack that is not yet available in the game, however in future updates it will be core part of worldbuilding of Isle of Ansur.
It will allow you to create locations, NPCs, quests, and basically all the environment game puts you in.

IV. Scripts
Scripts aren't pack type per se, because they usually serve as addition of features to one of previous pack types.
Additionally, their implementation right now is far from good or optimised, thus using them risks high possibility of resulting with buggy behaviour.

How to add mods to the game?

⚓ Back to Table of Contents

Adding mods to the game in *pre-alpha 3b* is much simpler than its precedessor, because now you only need to collect all your packs into .zip file, and you can just put the .zip file in *packs* folder.

They will get unpacked by the game automatically, unless manual system is set to true in game settings (it is set off by default).

It is recommended to name your zipped pack with the same name as pack ID, so it can be easily recognised by the game - since game does check zipped packs through pack IDs in specific situations.

More on how to build the mod will be explained in next section.

Creating statpack

⚓ Back to Table of Contents

Statpacks are made in *stats* directory in your IoA folder. Their structure is similar to other packs:

- ❁ Main IoA folder[D]
  - ❁ stats[D]
    - ❁ statpack ID[D]
      - ✳ contents

The ID is very important: it is what let the game distinguish between mods.
One mod can use multiple IDs, however this is rarely useful.
What's more important is that IDs should be unique enough to not be overwritten by another mod.
IDs are currently unchangeable, by which it means that changing the ID will break saved games.

How to write proper ID?

Ideally, IDs should contain only lowercase latin letters and underscores instead of spaces - this ensures it is compatibile everywhere and does not produce buggy behaviour.
However, since there's no guardrails for it yet, game won't put any penalty on not following those rules, except for when you use ":" which is restricted keyword (it can't be put as folder name on Windows, so it shouldn't be even possible to use it for ID).

Statpack structure

While we discussed place where statpack is located, we haven't dived into how statpack structure looks like.

- ❁ Statpack ID[D]
  - ❁ lang[D]
    - ✳ english.toml[F]
  - ❁ classes[D]
  - ❁ races[D]
  - ❁ origins[D]
  - ❁ religions[D]
  - ✳ attributes.yaml[F]
  - ✳ skills.yaml[F]
  - ✳ perks.yaml[F]
  - ✳ info.toml[F]

Two main parts of statpack is language file - located in *lang* folder - and statpack information file - being *info.toml* file. All other elements are entirely optional.
We will cover first one here, and info.toml is explained in configuration tutorial.

Language file

Language files are TOML files that allow you to write translations of your mod elements to English (required) and other languages. They use the same naming as official languages used by IoA, meaning that there are only those languages available at the moment:

- English - being default and required option
- Polish - being indev language, not fully translated even in vanilla

Language file is structured around *keys*, which are semi-IDs made by you in mod features that guide them into specific text in language file.
Let me showcase an example.

```
custom_key = "Translated text"
     (A↑)              (B ↑)

     (A↓)              (B ↓)
another_key = '''
Multiline text, which can use
More lines than one and be formatted <b>with HTML</b>
(making newline uses <br> automatically, though!)
'''
```

In this example, we can see two elements: key (A) and content (B). Just remember about this system, because it will be clearer how this works once we create new element for IoA.

Speaking of which...

Creating Features

⚓ Back to Table of Contents

Statpack elements are made with use of two systems: JSON files put into directories, and YAML files put in main directory.

I. Race

⚓ Back to Table of Contents

Creating race is made using JSON files. You do it simply by creating *races* directory, and file of your choice with .json extension.

- ❁ Statpack ID[D]
  - ❁ lang[D]
    - ✳ english.toml[F]
  - ❁ races[D]
    - ✳ example_race.json[F]
  - ✳ info.toml[F]

Simply adding the file isn't enough and may result in even crashes, but that's how you initialise new races in general.
Remember that race filename is used to create unique race ID, consisting of [statpackID]:[race filename].

Once we created our JSON file, we should fill it with contents. The only thing you need to include as requirement is *key* field, but there are many fields you may want to use as well.

Field	Usage
key	Translation ID that points to respective key in language file
attributes	Attributes that this race has bonus for during character creation
power	Modifiers to four powers IoA bases some of statistics
skills	Skills this race has bonus for during character creation
magic	List of magical skill bonuses given to specific magic types during character creation
names	Lore-friendly names that will be suggested during character creation, separated by genders
perks	Perks that your race may use as their special ability

Feel free to refer to this cheatsheet above to implement fields and understand their purpose.

Below is example race file to understand how fields should look like and how you should write your JSON file.

```
{
    "key": "example_race_key", (IA)
    "attributes": {                (IB)
        "strength": 2,
        "agility":    2
    },
    "power": {                     (IC)
        "void": 2
    },
    "skills": {                    (ID)
        "archery":          2,
        "light_firearms": 1
    },
    "magic": {                     (IE)
        "kind": {
            "destruction": 2
        }
    },
    "names": {                     (IF)
        "male": [
            "Smith",
            "Gregory"
        ],
        "female": [
            "Anna",
            "Eleonora"
        ]
    },
    "perks": {                     (IG)
        "diamond_soul": 100
    }
}
```

IA. Translation key

While translation key system is pretty straightforward - you put your key (in our case, *example_race_key*) in race file, and then head to translation file and writes its translation:

```
example_race_key = "My Best Race Ever Created"
```

There's one small caveat.

You need also one more key being used in translation file, which is made out of your translation key and *_descr* being added.

This is description key and it serves purpose of describing your race to the player.

While it can be one-lined, it is common to use multilined string for it, so it is easier to format the description without using HTML
 symbol too much.

Example translation file for our example can look like that then:

```
example_race_key       = "My Best Race Ever Created"
example_race_key_descr = '''
This race is very good at everything, but not perfect in anything!
It was created as God's creation like Frankenstein, made out of pieces taken away from best mortals: Da Vinci, Einstein, Picasso and thousands others!
Unfortunately somehow God made a mistake and while watching Loki series, he put some trickstery to that race.
'''
```

IB. Attributes

Attributes are those big parts of player's character. They are used as a basis for many actions in game, and also decide heavily on skill improvements.

They can't be raised after character creation except for when your character levels up - so each attribute point pushes character in certain direction.

Attributes are made as a dictionary - those knowing code will already knowing what it means, but for those new to this concept, I'd say that this is exactly the system we are using already - so that pairing of *key* and *value*, like in *"key": "example_race_key"*.

There are only six attributes in vanilla IoA, but more can be added by mods. Balanced race build needs to sum attributes to 4 points (you can use negative values to balance).

Here is cheatsheet of attributes available in vanilla:

Attribute	Description	ID
strength	Strength affects melee fight effectivity, inventory load and other factors relying on muscle power.	ansur:strength
agility	Agility is about speed of various actions, so it tells how quick your character is performing actions.	ansur:agility
endurance	Endurance is art of resilience in situation where character has some physical tension affecting them.	ansur:endurance
charisma	Charisma is all about communication and personal impression on others.	ansur:charisma
perception	Perception tells how good character is in realising certain elements of environment around them.	ansur:perception
intelligence	Intelligence is huge factor on how easily character learns and how good at magic they are.	ansur:intelligence

What's important to mention is that you can use both name of attribute, as well as attribute ID for vanilla attributes - meaning that *strength* is equal to *ansur:strength*.
This is not the case with modded attributes, where ID needs to be explicitly written.

IC. Powers
Powers are currently very in-development aspect of character creation, but eventually they will have heavier impact on player interactions.

Baedoor universe operates on four forces, being opposites within their pairs:
MGCK is opposite to TECH, since magic destabilises technology in many cases, and VOID is opposite to CONN, as void energy is usually pure negation of balance, being chaotic creative force.

Powers set by races can take ranges from -20 to 20, and should not

have positive values when dealing with oppositions, unless perks allowing for it are also in place.

ID. Skills
Skills are minor part of races character development, as main skill aspect is dependant on classes. It is, however, often used, as races also can give additional bonus to skill points.
Balanced build is argued to have up to three skill points being distributed.

There are huge amount of skills added by vanilla, and similarly to attributes, they share similar behaviour of being ID-agnostic, if referred to core ones. Modded ones can be used as well, but they need explicit IDs.

Here is small cheatsheet of skills. For more detailed descriptions, please refer to the ones visible in-game.

Skill	Brief description
Combat category	
Handfight	Ability to fight without weapons
Shortswords	Ability to fight with light-weight bladed weapons
Longswords	Ability to fight with heavier bladed weapons
Axes	Ability to use axes and halberds in fight
Blunts	Ability to use maces and all other blunt weapons
Polearms	Ability to fight with spears
Aim	Ability to target precisely with ranged weapons
Throwing	Ability to use throwable weapons, like knives or javelins
Archery	Ability to use bows of various kinds
Crossbows	Ability to hit enemies with crossbows
Light Firearms	Ability to use revolvers and pistols
Heavy Firearms	Ability to fight with shotguns and rifles
Light Armour	Ability to wear leather armour
Medium Armour	Ability to use chainmails and other lighter metal armour
Heavy Armour	Ability to use heavier metal armours
Blocking	Ability to block damage with shield or weapon

Magic category	
Spellcasting	Ability to cast spells in general
Scrolls writing	Ability to bind spells into scrolls
Mores casting	Ability to bind rune spells into mores
Unsealing	Ability to unbind spells from items
Social category	
Trade	Your proficiency in trading, bargaining, recognising value
Persuasion	How well you can convince others of your views
Craft category	
Hunting	Skill on how well you can hunt and skin hunted animals
Fishing	Ability to find better places to fish and get more fishes
Sewing	Ability to make better clothes
Carpentry	Ability to craft items from wood
Pottery	Ability to create items from clay
Stonemasonry	Ability to make things from stone
Ore Processing	Ability to extract metal from ores and melt alloys
Smithing	Skill allowing you to create various items from metals
Herbalism	Allows you to recognise plants and their use
Alchemy	Ability to merge ingredients to craft useful potions
Cooking	Ability to cook ingredients into tasty meals
Personal category	
Wound Treatment	Ability to heal faster and protect yourself from harms
Horse Riding	Ability to ride a horse
Boat Driving	Ability to drive boats
Animal Taming	Ability to communicate with animals
Musicality	Ability to perform music play with various instruments
Athletics	Affects how well you perform on your body condition
Toughness	Affects your physical resistance
Survival	Ability to use extreme environments to your benefit

IDs of those skills are made by using lowercase for all characters and replacing spaces with _ character.

IE. Magic

Magic is subset of "invisible" magic skills being *common magic* and *path magic*. It is separated from skills because while being entirely beneficial (so you may think of lowering skill advantages if you add magic section to your race), it has very branched form.
In general, it is considered a good practice to either dedicate magic section entirely to common magic or path magic exclusively, however there isn't anything stopping you from doing both.
Balanced race build sums up to six points on all magic branches.
For further explanation, read further.

☀ Common Magic

This is kind of magic that is open for everyone and does not put any limitations on how points distribution affect your gameplay.

Common Magic works on combinations of element-kind - meaning that casting spells use both element of spell and kind of spell, rising them both, but also using them both as factor of successful spellcasting.

There are eight elements and six kinds of common magic in vanilla:

Elements	Kinds
Void	Destruction
Fire	Absorption
Electricity	Illusion
Soul	Transformation
Wind	Evocation
Ice	Restoration
Water	
Earth	

It is not required to make element-kind proportions equal (3 elements-3 kinds), however making them so is suggested.
Non-equal variants give more freedom to player to choose, but also make no initial point.

You write common magic entries by using "elements" and/or "kinds" in such way:

```
"magic": {
   "elements": {
      "void": 2,
      "fire":   1
   },
   "kinds": {
      "destruction": 3
   }
}
```

♠ Path magic

Path magic is different from common one mostly because it conveys the idea that once chosen path is exclusive and do not allow for other paths to be taken. Additionally, choosing path magic weakens your spellcasting in common magic, at least with strongest spells offered.
In exchange, your dedication is rewarded with magic that can cross many boundaries, making you a powerful mage.

Since path magic is so exclusive, it is not used in race selection in vanilla, however the possibility to use it is still available.
While putting more than one path magic is not technically impossible, it may result in error in the future, as purpose of this magic type is to have exclusive choice.

There are seven magic paths for now in vanilla:

Path	Description
Dream magic	Path of wisdom of universes and oneironautics
Death magic	Cult of death and necromancy
Erd magic	Path of levitation and transmutation
Lunatri magic	Path of lunatris, ceremonial and nature-based
Ormath magic	Path set on communication with wind spirits
Terten magic	Path built on pestilence and plague
Keht magic	Path focused on blood sacrifices and vampirism

Similarly to common magic, path magic is created in *magic* section, but uses *path* as keyword:

```
"magic": {
   "path": {
      "dream_magic": 6
   }
}
```

IF. Names
Section dedicated to names offer you possibility to suggest player names for their character, based on the race they chosen.

Names are split by gender, using similar system to attributes/skills of vanilla genders being ID-agnostic.
Vanilla offers you *male*, *female* and *non-binary* genders to fill, however you do not need to use all of them. If you do not fill specific gender (or mod with more genders is added), ane "empty" gender will simply take all names from other genders available.

You can prevent this situation, if you need names being precisely only for genders they are designed for, by writing "*strict": true.*

```
   "names": {
        "strict": true,
        "male": [
            "Smith",
            "Gregory"
        ]
   },
```

Example above will suggest names of Smith and Gregory only to males, but if you remove "strict" keyword, it will offer those names to all genders not being covered.
It is important to notice that names are lists, using square brackets - [] - instead of curved ones - {}.

IG. Perks

Perks are special type of bonus given to specific race that allow them to gain some advantage, usually similar to genetics. Perks are rarely obtainable through quests, but not learnable in different way.

Since they are not polished enough, this part of tutorial will be expanded later.

II. Class

⚓ Back to Table of Contents

Class creation is pretty similar to race creation. Similarly, you are given the same structure of files:

- ❂ Statpack ID[D]
  - ❂ lang[D]
    - ✳ english.toml[F]
  - ❂ classes[D]
    - ✳ example_class.json[F]
  - ✳ info.toml[F]

The difference is mostly that files for class are stored under *classes* folder - but nothing else changes, you write your class in JSON file, and any language keys you use (including description key) work the same way as previously.
It is highly recommended to read previous section to get whole image on how to write class file, because here we will be mostly discussing differences between race and class - in context of the game, and thus in JSON structure.

Example of maximalist *example_class.json* file can look like that:

```
{
    "key": "example_class_key",              (IIA)
    "races_exclusive": ["ansur:saphtri"], (IIB)
    "power": {                               (IIC)
        "tech": 2
    },
    "skills": {                              (IID)
        "archery":          3,
        "light_firearms": 2
    },
    "skills_major": [                        (IIE)
        "archery"
    ],
    "skills_minor": [                        (IIF)
        "light_firearms"
    ],
    "magic": {                               (IIG)
        "kind": {
            "illusion": 4
        }
    },
    "perks": {                               (IIH)
        "diamond_soul": 100
    }
}
```

All elements marked with white colour are the same as with *race* file, so those will not be covered in tutorial.
Attributes, even though not listed above, can be used, as attribute scanner searches also through classes. Despite that, no vanilla class uses it, as it isn't purpose of class concept.
Balanced build tries to be contextual, but usually don't exceed 5 skill points in total and 6 magic points.

There are three new fields introduced with *class*, though, and we will try to expand on them right now.

IIB. Races exclusive
This is pretty simple and optional key, but this allows you to make race-exclusive class that will be available to player only when they select specific race beforehand.

Note that this element is list (using square brackets) and its contents are made out of RIDs (race IDs) - which is ID of the statpack, ":", and race file name without file extension.
Actually, the same system goes for CIDs (class IDs) - they are separate elements, but they are managed through the same idea.

IIE. Major skills
Class main purpose differentiating it from races and other elements is *skill bonus* which allows you to increase experience gain for specific skills.
They do not add skill points at the beginning, like *skills* field, but rather increase speed of learning - since improvement of skills in Isle of Ansur is done by doing tasks related to them.
In case of major skills, the bonus to experience gain is 175 percent of regular gain.

Balanced class build has up to four major skills, and its specific value vary depending on other factors (minor skills, perks, magic), so it can get severely less than that.

IIF. Minor skills
Minor skills work in very similar fashion major ones do, but the experience gain is only 150 percent of regular gain. Which still makes them extremely valuable.

Balanced build has up to four minor skills, but similarly to above, it is very contextual and can get much lower if other factors are overpowered enough.

III. Religions

⚓ Back to Table of Contents

Religions are not exactly expanded aspect of Isle of Ansur at the moment of pre-alpha 3b version. Therefore, even though this section will tell you how to add one, they will have no effect aside of being registered in character creation menu and then saved. This part of the game will be expanded in future versions, so keep an eye to not miss on necessary updates.

Religion can be added through the same form races and classes can be - it follows the same folder-file system:

- ❂ Statpack ID[D]
  - ❂ lang[D]
    - ✳ english.toml[F]
  - ❂ religions[D]
    - ✳ example_religion.json[F]
  - ✳ info.toml[F]

The difference is that religion JSON file contains only key:

```
{
   "key": "religion_example"
}
```

Obviously, filling description key is also required, and everything works in similar vein as previous entries - including creation of RLID (religion ID) by statpack ID and religion filename separated with ":".

IV. Origins

⚓ Back to Table of Contents

Origins share similar fate as religions, with slight difference that they have (non-parsed) section that will be used in future versions, but is written in vanilla origin beforehand.

It follows known folder structure:

- ❁ Statpack ID[D]
  - ❁ lang[D]
    - ✳ english.toml[F]
  - ❁ origins[D]
    - ✳ example_origin.json[F]
  - ✳ info.toml[F]

But the JSON contents, aside of mandatory key (and description key), features additional *new game* key.

```
{
  "key": "origin_example",
  "new_game": {
    "location": "ansur:tutorial",
    "class_inventory": true,
    "inventory": []
  }
}
```

While it is not read by game and so, it doesn't affect anything, it is put there as it will be most important part of origin.
Explaining shortly, origin is what decides on where your game starts and what will be your initial advantages, such as initial inventory contents.

Example here is almost exact vanilla origin, *wanderer*, which has legit planned location (using LID, location ID, ansur:tutorial) and empty inventory - which is, obviously, only a placeholder for minimal equipment.
Class inventory field tells if class can append its own inventory additions or not. It is recommended if you don't plan any specific initial weapons or tools for player.

Since this feature is highly experimental, it may get more advanced over time, for example including inventory based on specific factors. So keep that in mind when updating mod to newer IoA versions.

Creating themepack

⚓ Back to Table of Contents

Themepacks are made to change aesthetical part of Isle of Ansur - by modifying its colouring pallette or applying different banners or background images.
It has very similar structure as statpack, and ideally, if you make mod that includes both, it should have the same ID as statpack, so they will be considered as part of bigger pack.

- ❂ Main IoA folder[D]
  - ❂ themes[D]
    - ❂ themepack ID[D]
      - ✳ contents
    - ✳ guide.toml[F] (optional)

Let's start with elephant in the room - optional *guide.toml* file. It is actually not recommended to use it, but knowledge on what it is used for is required to understand theming system.

The file itself contains currently only from that two lines:

```
[behaviour]
prioritised_id = ""
```

Whole premise behind this file is to guide game to theme *different from default*, that means the one behind ansur ID. In future releases, where all packs will be unpacked during every start of the game, your mod will be able to overwrite the file to put your theme as default, without risk that removing the mod later will irreversibly make vanilla variant gone (vanilla will use unpacking too!).

For now, however, it is recommended to either use script that will allow you to overwrite the file back, or manage it by manually editing it. Its default empty value ("") makes ansur ID prioritised - to see your theme in effect, just put its ID into brackets.

Themepack structure looks like that, and is basically - for now - split into theme file and asset folders:

- ThemepackID[D]
  - assets[D]
    - images[F]
  - fonts[D]
    - fonts[F]
  - theme.toml[F]

Theme file

⚓ Back to Table of Contents

Theme file is heart of whole themepack - it decides on everything that is dependent on theme.
It is set into different sections, but let's take a look at complete one:

```
[backgrounds]                          (I)
icon = "ioa_logo.png"
logo = "ioa_logo.png"
menu = "menu_background.jpg"

[fonts.colours]                        (II)
disabled   = "#4F4C49"
enabled    = "#4E3510"
hovered    = "#7C613B"
background = "#4F3920"
other      = "#B0CA60"

[fonts.general]                        (III)
font_size = 1.0   # fallback value (if not specificed precisely later)

[fonts.menu]                           (III)
_       = ["ferrum.otf",    "1.0"]
```

While all sections kinda explain themselves, let me do some additional context.

I. Backgrounds
Backgrounds present you with ability to change:
✧ icon - game's icon appearing on top of window (non-fullscreen)
✧ logo - logo of the game on main menu
✧ menu - background image that appears in various menus

It's important to mention that menu isn't the same as panorama image - which is the image visible in main menu, not settings menu or character creation menu.
Panoramas are bound to worldpacks in this version, and while this behaviour will most likely change in next versions, it is not managed by themes right now.

Value of those keys should be names of image files that are in *assets* folder. You can use both .jpg and .png files for that purpose.

II. Font colours
This part explains itself basically - the keys here determine type of text that is visualised in game, and the value is hexcode text that represents colour. To get hexcode out of RGB/HSV values, you can use various online tools like this for RGB or this for HSV.

III. Font general & menu-specific
Section dedicated to fonts allow you to decide which font should be represented for each language.
General section is, as comment says, only a fallback value provider when it comes to internal font size, but it is recommended to provide it in sections dedicated to specific part of the game - in current version, the only part available is menu, covering fonts across settings, character creation and main menu. Later, location part will be available.

Menu-specific section consist of _ value, which is used as default register - used both for English language, as well as for any language that does not have font specified.

Value of those are *lists*, which consist of font filename, and text value (thus use of quotation marks) with floating-point number being font size.

It is important to remember that font size is not the same as text size which you can change in game settings - those are two different elements and they both affect text size, but separately, via algorithm that uses them both.

Similarly how backgrounds used their images in *assets* folder, you store fonts in *fonts* folder of themepack. You can use both .otf and .ttf font files.

Creating worldpack

⚓ Back to Table of Contents

Worldpacks are most massive pack types of all, but they are also the ones that are the least developed. While in the future they will be cornerstone of whole Isle of Ansur gameplay, currently they serve only two purposes: drawing main menu images (panoramas) and playing main menu music.
All those features will be moved to themepack in the future, so be aware of it while maintaining your mod.

Structure of worldpack is amazingly similar to previous ones:

- ❁ Main IoA folder[D]
  - ❁ worlds[D]
    - ❁ worldpack ID[D]
      - ✳ contents

Panoramas

⚓ Back to Table of Contents

Each time you run the game, random panorama from list is selected. Panorama list is taken from all worldpacks, listing all images available under specific folder path.

You can add your own panorama by putting .jpg or .png image file in *backgrounds* directory, nested a bit in worldpack folder:

- ❁ WorldpackID[D]
  - ❁ assets[D]
    - ❁ backgrounds[D]
      - ✳ panorama_image.png[F]

What's interesting, however, is that you can make your image being prioritised. By adding %PR_ prefix before image name, you put that image in another queue, which counts only images with this prefix on.

This means that you can set your own preferred image instead of getting randomised one, if you put only one image and prefix it accordingly.

Be aware though that other worldpacks using prioritised prefix will also be included in randomisation batch.

Menu music

⚓ Back to Table of Contents

Music actually works very similar to panoramas, as it shuffles between all music in specific directory - and similarly, it also has separate randomisation batch for those sound files that have %PR_ prefix.

The difference is only the position, as menu music uses different folder names to decide on what is played next.

- ❁ WorldpackID[D]
  - ❁ music[D]
    - ❁ backgrounds[D]
      - ✳ sound.mp3[F]

For music, you can use .mp3, .wav and .ogg file formats.

Additionally, remember that music plays in main menu without breaks, so making it more versatile can be actually beneficial. Unlike panorama, it isn't fixed on only one specific choice - so you can have many songs playing after each other.

Configuration

⚓ Back to Table of Contents

Configuring the pack is done by *info.toml* file, put in your statpack, worldpack or themepack.
If more than one of pack types is used, you only need to put it in one - preferrably, in statpack, then worldpack, since this is order through which game loads the files.

Configuration file is not necessarily required, at least yet, but it is very useful because it helps a lot in mod management.

This is regular *info.toml* contents:

```
name    = "Your Pack Name"
credits = "Whoever Worked on the Pack"
url     = "Your website"
version = "1.1.1" (A)

description  = '''
Whatever you want to write as your pack description
'''

[requirements] # needs to be at the end to not be buggy
# none (B)
```

Most of those fields are self-explainable - you simply write your pack name, your nickname (or your fellow devs, if you work in team), you can also put your website in URL.
You can also write description of your mod that will be visible in-game, when mod manager appears.

There are, however, two special fields - *version* and *[requirements]* which is TOML dictionary.

A. Version

Version field tells about version of your pack, which is important because you can decide to update your pack with new features, remove some, or just change existing ones.
This field allows you to inform player, as well as other mods, that contents of your mod are in specific timespace - and changing the version let you inform that contents may have changed.

While outside of context, this may sound not as crucial, next section will show why using versioning is actually very important.

But before we go into this, let me explain how versioning works. You provide value into quotation marks - "..." - meaning the game will recognise value as a text.
This allows you to use various standards. The ones supported by the game look like this:

System	Example	Description
N	2	Simply number without anything else
N.N	30.12.1899	Number with one dot (floating-point number)
N.N.N	2.3.5	Number with two dots (so called semantic versioning)
N.NX	2.3b	Number with dot and letter (a = 0, b = 1, c = 2, etc.)

You can use any of those to determine your version. The game will automatically convert all of those into semantic versioning (so N.N.N system) when needed.

You can look into this conversion table to see how it works:

Original version	Universal version (converted)
2	2.0.0
2.2	2.2.0
2.2b	2.2.1

B. Requirements
Here is where versioning truly starts having its important purpose. Requirements dictionary (which is visible by use of square brackets without any key) allow you to tell the game that your mod requires specific pack to be alongside yours to work correctly.
Or, that your pack needs specific version of game.
Because, yes, game files are also a pack, just with special properties.

Let's read into how to make requirement system.

TOML dictionary works very similar to what you already saw. You simply put keys and values in it, just like you did with keys before dictionary segment.
The difference is that square brackets separate dictionary keys from regular TOML keys - that's why it's heavily suggested to put dictionaries at the end, as they can get glitchy in specific situations.

Let's work on this (quite expansive) example of dictionary:

```
[requirements]
ansur = "0.3b"
example = "0.2.0 - 0.2.5"
another = ">0.2"
some = ""
```

As you can see, each key of dictionary is responsible for pack ID - with game itself ID being *ansur*.

If you put any recognised versioning value here, the game will check for exact version of the pack, searching through *version* key in *info.toml* file (see, I told you it will be important).
Don't worry if you put not exact version - for example, 1.1 instead of 1.1.0 - game will convert your requirement into universal system and compare with similarly converted value. For table of conversions, look at previous section.

Using "-" symbol let you specify range - so if, as in example, you specify "0.2.0 - 0.2.5", that means 0.2.0, 0.2.1, 0.2.2, 0.2.3, 0.2.4 and 0.2.5 will be compatible.
All range requirements use inclusive system on both sides, since this makes it less chaotic to write.

">", as well as "<" symbol, let you specify minimum and maximum version. They should be put at the beginning of the version name. They are pretty self-explainable as well:

"<n" makes all versions prior to *n+1* compatible, basically setting range between 0.0.0 and *n*, including both.

">n" makes *n* minimum version, making all versions coming after it compatible. On technical terms, the upper limit of versions is 99999.99999.99999, but it is rather impossible for anyone to get to such high numbers without deliberately trying to create the glitch.

If you don't want to specify version, simply write empty string value by putting two quotation marks without anything in them - "". This will skip version check, resulting with error only when pack ID is not existing at all.

If dependency requirements are not met, game will show error message in main menu and disable playing options.

To see what went wrong, you can look at error messages in latest log file (core/logs), or at special file (core/data/pack_manag) called *pack_errors.yaml*, which is used by game itself to list all packages that reported dependency errors.